INTRODUCTION TO OPERATING SYSTEMS

LibreTexts™

Introduction to Operating Systems

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.

LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

This text was compiled on 05/15/2026

TABLE OF CONTENTS

Licensing

1: Binary and Number Representation

- 1.1: How Computers See Numbers
- 1.2: Types and Number Representation

2: The Basics - An Overview

- 2.1: Introduction to Operating Systems
- 2.2: Starting with the Basics

3: The Operating System

- 3.1: The Role of the Operating System
- 3.2: Operating System Organisation
- 3.3: System Calls
- 3.4: Privileges
- 3.5: Function of the Operating System
- 3.6: Types of Operating Systems
  - 3.6.1: Types of Operating Systems (continued)
  - 3.6.2: Types of Operating Systems (continued)
- 3.7: Difference between multitasking, multithreading and multiprocessing
  - 3.7.1: Difference between multitasking, multithreading and multiprocessing (continued)
  - 3.7.2: Difference between Multiprogramming, multitasking, multithreading and multiprocessing (continued)

4: Computer Architecture - the CPU

- 4.1: Instruction Cycles
  - 4.1.1: Instruction Cycles - Fetch
  - 4.1.2: Instruction Cycles - Instruction Primer
- 4.2: Interrupts

5: Computer Architecture - Memory

- 5.1: Memory Hierarchy
- 5.2: Memory Hierarchy (continued)
- 5.3: Cache Memory
  - 5.3.1: Cache Memory - Multilevel Cache
  - 5.3.2: Cache Memory - Locality of reference
- 5.4: Direct Memory Access

6: Computer Architecture - Peripherals and Buses

- 6.1: Peripherals and buses
- 6.2: The Processor - Bus

7: Small to Big Systems

- 7.1: Symmetric Multi-Processing Systems
- 7.2: Multiprocessor and Multicore Systems

8: Processes

- 8.1: The Process
- 8.2: Processes
- 8.3: Elements of a process
- 8.4: Process States
- 8.5: Process Description
- 8.6: Process Control
- 8.7: Fork and Exec
- 8.8: Scheduling
- 8.9: Execution within the Operating System
  - 8.9.1: Execution within the Operating System - Dual Mode
- 8.10: The Shell
- 8.11: Signals

9: Threads

- 9.1: Process and Threads
- 9.2: Thread Types
  - 9.2.1: Thread Types - Models
- 9.3: Thread Relationships
- 9.4: Benefits of Multithreading

10: Concurrency and Process Synchronization

- 10.1: Introduction to Concurrency
- 10.2: Process Synchronization
- 10.3: Mutual Exclusion
- 10.4: Interprocess Communication
  - 10.4.1: IPC - Semaphores
  - 10.4.2: IPC - Monitors
  - 10.4.3: IPC - Message Passing / Shared Memory

11: Concurrency- Deadlock and Starvation

- 11.1: Concept and Principles of Deadlock
- 11.2: Deadlock Detection and Prevention
- 11.3: Starvation
- 11.4: Dining Philosopher Problem

12: Memory Management

- 12.1: Random Access Memory (RAM) and Read Only Memory (ROM)
- 12.2: Memory Hierarchy
- 12.3: Requirements for Memory Management
- 12.4: Memory Partitioning
  - 12.4.1: Fixed Partitioning

- 12.4.2: Variable Partitioning
- 12.4.3: Buddy System
- 12.5: Logical vs Physical Address
- 12.6: Paging
- 12.7: Segmentation

13: Virtual Memory

- 13.1: Memory Paging
  - 13.1.1: Memory Paging - Page Replacement
- 13.2: Virtual Memory in the Operating System

14: Uniprocessor CPU Scheduling

- 14.1: Types of Processor Scheduling
- 14.2: Scheduling Algorithms

15: Multiprocessor Scheduling

- 15.1: The Question
- 15.2: Multiprocessor Scheduling

16: I/O and Disk Management

- 16.1: Input / Output
- 16.2: Types of I/O
- 16.3: I/O Buffering
- 16.4: Disk Drives in the OS
- 16.5: Disk Drive Scheduling
- 16.6: RAID

17: File Management

- 17.1: Overview
- 17.2: Files
  - 17.2.1: Files (continued)
- 17.3: Directory
- 17.4: File Sharing

Index

Glossary

Detailed Licensing

Licensing

*A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.*

CHAPTER OVERVIEW

1: Binary and Number Representation

1.1: How Computers See Numbers
1.2: Types and Number Representation

1: Binary and Number Representation is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

1.1: How Computers See Numbers

Binary — the basis of computing

Binary Theory

Introduction

Binary is a base-2 number system that uses two mutually exclusive states to represent information. A binary number is made up of elements called *bits* where each bit can be in one of the two possible states. Generally, we represent them with the numerals `1` and `0`. We also talk about them being true and false. Electrically, the two states might be represented by high and low voltages or some form of switch turned on or off.

We build binary numbers the same way we build numbers in our traditional base 10 system. However, instead of a one's column, a 10's column, a 100's column (and so on) we have a one's column, a two's columns, a four's column, an eight's column, and so on, as illustrated below.

Table 2.1. Binary

$2^{\cdots}$	2^6	2^5	2^4	2^3	2^2	2^1	2^0
...	64	32	16	8	4	2	1

For example, to represent the number 203 in base 10, we know we place a `3` in the `1's` column, a `0` in the `10's` column and a `2` in the `100's` column. This is expressed with exponents in the table below.

Table 2.2. 203 in base 10

10^2	10^1	10^0
2	0	3

Or, in other words, $2 \times 10^2 + 3 \times 10^0 = 200 + 3 = 203$. To represent the same thing in binary, we would have the following table.

Table 2.3. 203 in base 2

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	0	1	0	1	1

That equates to $2^7 + 2^6 + 2^3 + 2^1 + 2^0 = 128 + 64 + 8 + 2 + 1 = 203$.

The basis of computing

You may be wondering how a simple number is the basis of all the amazing things a computer can do. Believe it or not, it is! The processor in your computer has a complex but ultimately limited set of *instructions* it can perform on values such as addition, multiplication, etc. Essentially, each of these instructions is assigned a number so that an entire program (add this to that, multiply by that, divide by this and so on) can be represented by a just a stream of numbers. For example, if the processor knows operation `2` is addition, then `252` could mean "add 5 and 2 and store the output somewhere". The reality is of course much more complicated (see Chapter 3, *Computer Architecture*) but, in a nutshell, this is what a computer is.

In the days of punch-cards, one could see with their eye the one's and zero's that make up the program stream by looking at the holes present on the card. Of course this moved to being stored via the polarity of small magnetic particles rather quickly (tapes, disks) and onto the point today that we can carry unimaginable amounts of data in our pocket.

Translating these numbers to something useful to humans is what makes a computer so useful. For example, screens are made up of millions of discrete *pixels*, each too small for the human eye to distinguish but combining to make a complete image. Generally each pixel has a certain red, green and blue component that makes up it's display color. Of course, these values can be represented

by numbers, which of course can be represented by binary! Thus any image can be broken up into millions of individual dots, each dot represented by a *tuple* of three values representing the red, green and blue values for the pixel. Thus given a long string of such numbers, formatted correctly, the video hardware in your computer can convert those numbers to electrical signals to turn on and off individual pixels and hence display an image.

As you read on, we will build up the entire modern computing environment from this basic building block; *from the bottom-up* if you will!

Bits and Bytes

As discussed above, we can essentially choose to represent anything by a number, which can be converted to binary and operated on by the computer. For example, to represent all the letters of the alphabet we would need at least enough different combinations to represent all the lower case letters, the upper case letters, numbers and punctuation, plus a few extras. Adding this up means we need probably around 80 different combinations.

If we have two bits, we can represent four possible unique combinations (`00 01 10 11`). If we have three bits, we can represent 8 different combinations. In general, with `n` bits we can represent 2^n unique combinations.

8 bits gives us $2^8 = 256$ unique representations, more than enough for our alphabet combinations. We call a group of 8 bits a *byte*. Guess how big a C `char` variable is? One byte.

ASCII

Given that a byte can represent any of the values 0 through 255, anyone could arbitrarily make up a mapping between characters and numbers. For example, a video card manufacturer could decide that `1` represents `A` , so when value `1` is sent to the video card it displays a capital 'A' on the screen. A printer manufacturer might decide for some obscure reason that `1` represented a lower-case 'z', meaning that complex conversions would be required to display and print the same thing.

To avoid this happening, the *American Standard Code for Information Interchange* or ASCII was invented. This is a *7-bit* code, meaning there are 2^7 or 128 available codes.

The range of codes is divided up into two major parts; the non-printable and the printable. Printable characters are things like characters (upper and lower case), numbers and punctuation. Non-printable codes are for control, and do things like make a carriage-return, ring the terminal bell or the special `NULL` code which represents nothing at all.

127 unique characters is sufficient for American English, but becomes very restrictive when one wants to represent characters common in other languages, especially Asian languages which can have many thousands of unique characters.

To alleviate this, modern systems are moving away from ASCII to *Unicode*, which can use up to 4 bytes to represent a character, giving *much* more room!

Parity

ASCII, being only a 7-bit code, leaves one bit of the byte spare. This can be used to implement *parity* which is a simple form of error checking. Consider a computer using punch-cards for input, where a hole represents 1 and no hole represents 0. Any inadvertent covering of a hole will cause an incorrect value to be read, causing undefined behaviour.

Parity allows a simple check of the bits of a byte to ensure they were read correctly. We can implement either *odd* or *even* parity by using the extra bit as a *parity bit*.

In odd parity, if the number of 1's in the 7 bits of information is odd, the parity bit is set, otherwise it is not set. Even parity is the opposite; if the number of 1's is even the parity bit is set to 1.

In this way, the flipping of one bit will case a parity error, which can be detected.

XXX more about error correcting

16, 32 and 64 bit computers

Numbers do not fit into bytes; hopefully your bank balance in dollars will need more range than can fit into one byte! Modern architectures are at least *32 bit* computers. This means they work with 4 bytes at a time when processing and reading or writing to memory. We refer to 4 bytes as a *word*; this is analogous to language where letters (bits) make up words in a sentence, except in

computing every word has the same size! The size of a C `int` variable is 32 bits. Modern architectures are 64 bits, which doubles the size the processor works with to 8 bytes.

Kilo, Mega and Giga Bytes

Computers deal with a lot of bytes; that's what makes them so powerful! We need a way to talk about large numbers of bytes, and a natural way is to use the "International System of Units" (SI) prefixes as used in most other scientific areas. So for example, kilo refers to 10^3 or 1000 units, as in a kilogram has 1000 grams.

1000 is a nice round number in base 10, but in binary it is `1111101000` which is not a particularly "round" number. However, 1024 (or 2^{10}) is a round number — (`10000000000` — and happens to be quite close to the base 10 meaning value of "kilo" (1000 as opposed to 1024). Thus 1024 bytes naturally became known as a *kilobyte*. The next SI unit is "mega" for 10^6 and the prefixes continue upwards by 10^3 (corresponding to the usual grouping of three digits when writing large numbers). As it happens, 2^{20} is again close to the SI base 10 definition for mega; 1048576 as opposed to 1000000. Increasing the base 2 units by powers of 10 remains functionally close to the SI base 10 value, although each increasing factor diverges slightly further from the base SI meaning. Thus the SI base-10 units are "close enough" and have become the commonly used for base 2 values.

Table 2.4. Base 2 and 10 factors related to bytes

Name	Base 2 Factor	Bytes	Close Base 10 Factor	Base 10 bytes
1 Kilobyte	2^{10}	1,024	10^3	1,000
1 Megabyte	2^{20}	1,048,576	10^6	1,000,000
1 Gigabyte	2^{30}	1,073,741,824	10^9	1,000,000,000
1 Terabyte	2^{40}	1,099,511,627,776	10^{12}	1,000,000,000,000
1 Petabyte	2^{50}	1,125,899,906,842,624	10^{15}	1,000,000,000,000,000
1 Exabyte	2^{60}	1,152,921,504,606,846,976	10^{18}	1,000,000,000,000,000,000

SI units compared in base 2 and base 10

It can be very useful to commit the base 2 factors to memory as an aid to quickly correlate the relationship between number-of-bits and "human" sizes. For example, we can quickly calculate that a 32 bit computer can address up to four gigabytes of memory by noting the recombination of 2^2 (4) + 2^{30}. A 64-bit value could similarly address up to 16 exabytes (2^4 + 2^{60}); you might be interested in working out just how big a number this is. To get a feel for how big that number is, calculate how long it would take to count to 2^{64} if you incremented once per second.

Kilo, Mega and Giga Bits

Apart from the confusion related to the overloading of SI units between binary and base 10, capacities will often be quoted in terms of *bits* rather than bytes. Generally this happens when talking about networking or storage devices; you may have noticed that your ADSL connection is described as something like 1500 kilobits/second. The calculation is simple; multiply by 1000 (for the kilo), divide by 8 to get bytes and then 1024 to get kilobytes (so 1500 kilobits/s=183 kilobytes per second).

The SI standardisation body has recognised these dual uses and has specified unique prefixes for binary usage. Under the standard 1024 bytes is a `kibibyte`, short for *kilo binary* byte (shortened to KiB). The other prefixes have a similar prefix (Mebibyte, MiB, for example). Tradition largely prevents use of these terms, but you may seem them in some literature.

Conversion

The easiest way to convert between bases is to use a computer, after all, that's what they're good at! However, it is often useful to know how to do conversions by hand.

The easiest method to convert between bases is *repeated division*. To convert, repeatedly divide the quotient by the base, until the quotient is zero, making note of the remainders at each step. Then, write the remainders in reverse, starting at the bottom and appending to the right each time. An example should illustrate; since we are converting to binary we use a base of 2.

Table 2.5. Convert 203 to binary

Quotient		Remainder	
$203_{10} \div 2 =$	101	1	
$101_{10} \div 2 =$	50	1	↑
$50_{10} \div 2 =$	25	0	↑
$25_{10} \div 2 =$	12	1	↑
$12_{10} \div 2 =$	6	0	↑
$6_{10} \div 2 =$	3	0	↑
$3_{10} \div 2 =$	1	1	↑
$1_{10} \div 2 =$	0	1	↑

Reading from the bottom and appending to the right each time gives `11001011` , which we saw from the previous example was 203.

Boolean Operations

George Boole was a mathematician who discovered a whole area of mathematics called *Boolean Algebra*. Whilst he made his discoveries in the mid 1800's, his mathematics are the fundamentals of all computer science. Boolean algebra is a wide ranging topic, we present here only the bare minimum to get you started.

Boolean operations simply take a particular input and produce a particular output following a rule. For example, the simplest boolean operation, `not` simply inverts the value of the input operand. Other operands usually take two inputs, and produce a single output.

The fundamental Boolean operations used in computer science are easy to remember and listed below. We represent them below with *truth tables*; they simply show all possible inputs and outputs. The term *true* simply reflects `1` in binary.

Not

Usually represented by `!` , `not` simply inverts the value, so `0` becomes `1` and `1` becomes `0`

Table 2.6. Truth table for *not*

Input	Output
1	0
0	1

And

To remember how the and operation works think of it as "if one input *and* the other are true, result is true

Table 2.7. Truth table for *and*

Input 1	Input 2	Output
0	0	0
1	0	0
0	1	0
1	1	1

Or

To remember how the `or` operation works think of it as "if one input *or* the other input is true, the result is true

Table 2.8. Truth table for *or*

Input 1	Input 2	Output
0	0	0
1	0	1
0	1	1
1	1	1

Exclusive Or (xor)

Exclusive or, written as `xor` is a special case of `or` where the output is true if one, and *only* one, of the inputs is true. This operation can surprisingly do many interesting tricks, but you will not see a lot of it in the kernel.

Table 2.9. Truth table for *xor*

Input 1	Input 2	Output
0	0	0
1	0	1
0	1	1
1	1	0

How computers use boolean operations

Believe it or not, essentially everything your computer does comes back to the above operations. For example, the half adder is a type of circuit made up from boolean operations that can add bits together (it is called a half adder because it does not handle carry bits). Put more half adders together, and you will start to build something that can add together long binary numbers. Add some external memory, and you have a computer.

Electronically, the boolean operations are implemented in *gates* made by *transistors*. This is why you might have heard about transistor counts and things like Moore's Law. The more transistors, the more gates, the more things you can add together. To create the modern computer, there are an awful lot of gates, and an awful lot of transistors. Some of the latest Itanium processors have around 460 million transistors.

Working with binary in C

In C we have a direct interface to all of the above operations. The following table describes the operators

Table 2.10. Boolean operations in C

Operation	Usage in C
not	`!`
and	`&`
or	`|`
xor	`^`

We use these operations on variables to modify the bits within the variable. Before we see examples of this, first we must divert to describe hexadecimal notation.

Hexadecimal refers to a base 16 number system. We use this in computer science for only one reason, it makes it easy for humans to think about binary numbers. Computers only ever deal in binary and hexadecimal is simply a shortcut for us humans trying to work with the computer.

So why base 16? Well, the most natural choice is base 10, since we are used to thinking in base 10 from our every day number system. But base 10 does not work well with binary -- to represent 10 different elements in binary, we need four bits. Four bits, however, gives us sixteen possible combinations. So we can either take the very tricky road of trying to convert between base 10 and binary, or take the easy road and make up a base 16 number system -- hexadecimal!

Hexadecimal uses the standard base 10 numerals, but adds `A B C D E F` which refer to `10 11 12 13 14 15` (n.b. we start from zero).

Traditionally, any time you see a number prefixed by `0x` this will denote a hexadecimal number.

As mentioned, to represent 16 different patterns in binary, we would need exactly four bits. Therefore, each hexadecimal numeral represents exactly four bits. You should consider it an exercise to learn the following table off by heart.

Table 2.11. Hexadecimal, Binary and Decimal

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Of course there is no reason not to continue the pattern (say, assign G to the value 16), but 16 values is an excellent trade off between the vagaries of human memory and the number of bits used by a computer (occasionally you will also see base 8 used, for example for file permissions under UNIX). We simply represent larger numbers of bits with more numerals. For example, a sixteen

bit variable can be represented by 0xAB12, and to find it in binary simply take each individual numeral, convert it as per the table and join them all together (so `0xAB12` ends up as the 16-bit binary number `1010101100010010`). We can use the reverse to convert from binary back to hexadecimal.

We can also use the same repeated division scheme to change the base of a number. For example, to find 203 in hexadecimal

Table 2.12. Convert 203 to hexadecimal

Quotient		Remainder	
$203_{10} \div 16 =$	12	11 (0xB)	
$12_{10} \div 16 =$	0	12 (0xC)	↑

Hence 203 in hexadecimal is `0xCB` .

Practical Implications

Use of binary in code

Whilst binary is the underlying language of every computer, it is entirely practical to program a computer in high level languages without knowing the first thing about it. However, for the low level code we are interested in a few fundamental binary principles are used repeatedly.

Masking and Flags

Masking

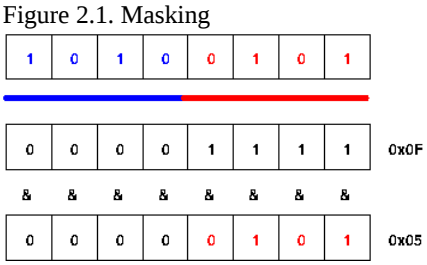
In low level code, it is often important to keep your structures and variables as space efficient as possible. In some cases, this can involve effectively packing two (generally related) variables into one.

Remember each bit represents two states, so if we know a variable only has, say, 16 possible states it can be represented by 4 bits (i.e. 2^4=16 unique values). But the smallest type we can declare in C is 8 bits (a `char`), so we can either waste four bits, or find some way to use those left over bits.

We can easily do this by the process of *masking*. This uses the rules of logical operations to extract values.

The process is illustrated in the figure below. We can keep two separate 4-bit values "inside" a single 8-bit character. We consider the upper four-bits as one value (blue) and the lower 4-bits (red) as another. To extract the lower four bits, we set our mask to have the lower-4 bits set to `1` (`0x0F`). Since the `logical and` operation will only set the bit if *both* bits are `1` , those bits of the mask set to `0` effectively hide the bits we are not interested in.

Figure 2.1. Masking



To get the top (blue) four bits, we would invert the mask; in other words, set the top 4 bits to `1` and the lower 4-bits to `0` . You will note this gives a result of `1010 0000` (or, in hexadecimal `0xA0`) when really we want to consider this as a unique 4-bit value `1010` (`0x0A`). To get the bits into the right position we use the `right shift` operation 4 times, giving a final value of `0000 1010` .

Example 2.1. Using masks

```
1 #include <stdio.h>

  #define LOWER_MASK 0x0F
```

```
#define UPPER_MASK 0xF0

int main(int argc, char* argv[])
{
        /* Two 4-bit values stored in one
         * 8-bit variable */
        char value = 0xA5;
        char lower = value & LOWER_MASK;
        char upper = (value & UPPER_MASK) >> 4;

        printf("Lower: %x\n", lower);
        printf("Upper: %x\n", upper);
}
```

Setting the bits requires the `logical or` operation. However, rather than using `1` 's as the mask, we use `0` 's. You should draw a diagram similar to the above figure and work through setting bits with the `logical or` operation.

Flags

Often a program will have a large number of variables that only exist as *flags* to some condition. For example, a state machine is an algorithm that transitions through a number of different states but may only be in one at a time. Say it has 8 different states; we could easily declare 8 different variables, one for each state. But in many cases it is better to declare *one 8 bit variable* and assign each bit to *flag* flag a particular state.

Flags are a special case of masking, but each bit represents a particular boolean state (on or off). An *n* bit variable can hold *n* different flags. See the code example below for a typical example of using flags -- you will see variations on this basic code very often.

Example 2.2. Using flags

```
#include <stdio.h>

/*
 *  define all 8 possible flags for an 8 bit variable
 *      name  hex     binary
 */
#define FLAG1 0x01 /* 00000001 */
#define FLAG2 0x02 /* 00000010 */
#define FLAG3 0x04 /* 00000100 */
#define FLAG4 0x08 /* 00001000 */
/* ... and so on */
#define FLAG8 0x80 /* 10000000 */

int main(int argc, char *argv[])
{
    char flags = 0; /* an 8 bit variable */

    /* set flags with a logical or */
    flags = flags | FLAG1; /* set flag 1 */
```

```
    flags = flags | FLAG3; /* set flag 3

    /* check flags with a logical and.  If the flag is set (1)
     * then the logical and will return 1, causing the if
     * condition to be true. */
    if (flags & FLAG1)
            printf("FLAG1 set!\n");

    /* this of course will be untrue. */
    if (flags & FLAG8)
            printf("FLAG8 set!\n");

    /* check multiple flags by using a logical or
     * this will pass as FLAG1 is set */
    if (flags & (FLAG1|FLAG4))
            printf("FLAG1 or FLAG4 set!\n");

    return 0;
}
```

This page titled 1.1: How Computers See Numbers is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

1.2: Types and Number Representation

Types and Number Representation

C Standards

Although a slight divergence, it is important to understand a bit of history about the C language.

C is the common languge of the systems programming world. Every operating system and its associated system libraries in common use is written in C, and every system provides a C compiler. To stop the language diverging across each of these systems where each would be sure to make numerous incompatible changes, a strict standard has been written for the language.

Officially this standard is known as ISO/IEC 9899:1999(E), but is more commonly referred to by its shortened name *C99*. The standard is maintained by the International Standards Organisation (ISO) and the full standard is available for purchase online. Older standards versions such as C89 (the predecessor to C99 released in 1989) and ANSI C are no longer in common usage and are encompassed within the latest standard. The standard documentation is very technical, and details most every part of the language. For example it explains the syntax (in Backus Naur form), standard `#define` values and how operations should behave.

It is also important to note what the C standards does *not* define. Most importantly the standard needs to be appropriate for every architecture, both present and future. Consequently it takes care *not* to define areas that are architecture dependent. The "glue" between the C standard and the underlying architecture is the Application Binary Interface (or ABI) which we discuss below. In several places the standard will mention that a particular operation or construct has an unspecified or implementation dependent result. Obviously the programmer can not depend on these outcomes if they are to write portable code.

GNU C

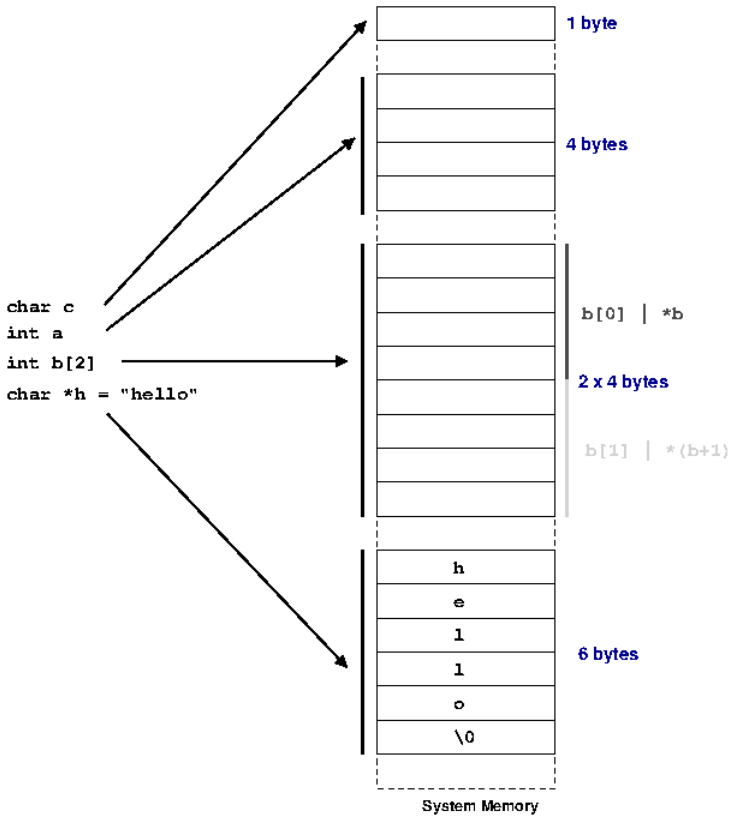
The GNU C Compiler, more commonly referred to as gcc, almost completely implements the C99 standard. However it also implements a range of extensions to the standard which programmers will often use to gain extra functionality, at the expense of portability to another compiler. These extensions are usually related to very low level code and are much more common in the system programming field; the most common extension being used in this area being inline assembly code. Programmers should read the gcc documentation and understand when they may be using features that diverge from the standard.

gcc can be directed to adhere strictly to the standard (the `-std=c99` flag for example) and warn or create an error when certain things are done that are not in the standard. This is obviously appropriate if you need to ensure that you can move your code easily to another compiler.

Types

As programmers, we are familiar with using variables to represent an area of memory to hold a value. In a *typed* language, such as C, every variable must be declared with a *type*. The type tells the compiler about what we expect to store in a variable; the compiler can then both allocate sufficient space for this usage and check that the programmer does not violate the rules of the type. In the example below, we see an example of the space allocated for some common types of variables.

Figure 2.2. Types



The C99 standard purposely only mentions the *smallest* possible size of each of the types defined for C. This is because across different processor architectures and operating systems the best size for types can be wildly different.

To be completely safe programmers need to never assume the size of any of their variables, however a functioning system obviously needs agreements on what sizes types are going to be used in the system. Each architecture and operating system conforms to an *Application Binary Interface* or *ABI*. The ABI for a system fills in the details between the C standard and the requirements of the underlying hardware and operating system. An ABI is written for a specific processor and operating system combination.

Table 2.13. Standard Integer Types and Sizes

Type	C99 minimum size (bits)	Common size (32 bit architecture)
`char`	8	8
`short`	16	16
`int`	16	32
`long`	32	32
`long long`	64	64
Pointers	Implementation dependent	32

Above we can see the only divergence from the standard is that `int` is commonly a 32 bit quantity, which is twice the strict minimum 16 bit size that the C99 requires.

Pointers are really just an address (i.e. their value is an address and thus "points" somewhere else in memory) therefore a pointer needs to be sufficient in size to be able to address any memory in the system.

64 bit

One area that causes confusion is the introduction of 64 bit computing. This means that the processor can handle addresses 64 bits in length (specifically the registers are 64 bits wide; a topic we discuss in Chapter 3, *Computer Architecture*).

This firstly means that all pointers are required to be a 64 bits wide so they can represent any possible address in the system. However, system implementers must then make decisions about the size of the other types. Two common models are widely used, as shown below.

Table 2.14. Standard Scalar Types and Sizes

Type	C99 minimum size (bits)	Common size (LP64)	Common size (Windows)
`char`	8	8	8
`short`	16	16	16
`int`	16	32	32
`long`	32	64	32
`long long`	64	64	64
Pointers	Implementation dependent	64	64

You can see that in the LP64 (long-pointer 64) model `long` values are defined to be 64 bits wide. This is different to the 32 bit model we showed previously. The LP64 model is widely used on UNIX systems.

In the other model, `long` remains a 32 bit value. This maintains maximum compatibility with 32 code. This model is in use with 64 bit Windows.

There are good reasons why the size of `int` was not increased to 64 bits in either model. Consider that if the size of `int` is increased to 64 bits you leave programmers no way to obtain a 32 bit variable. The only possibly is redefining `shorts` to be a larger 32 bit type.

A 64 bit variable is so large that it is not generally required to represent many variables. For example, loops very rarely repeat more times than would fit in a 32 bit variable (4294967296 times!). Images usually are usually represented with 8 bits for each of a red, green and blue value and an extra 8 bits for extra (alpha channel) information; a total of 32 bits. Consequently for many cases, using a 64 bit variable will be wasting at least the top 32 bits (if not more). Not only this, but the size of an integer array has now doubled too. This means programs take up more system memory (and thus more cache; discussed in detail in Chapter 3, *Computer Architecture*) for no real improvement. For the same reason Windows elected to keep their long values as 32 bits; since much of the Windows API was originally written to use long variables on a 32 bit system and hence does not require the extra bits this saves considerable wasted space in the system without having to re-write all the API.

If we consider the proposed alternative where `short` was redefined to be a 32 bit variable; programmers working on a 64 bit system could use it for variables they know are bounded to smaller values. However, when moving back to a 32 bit system their same `short` variable would now be only 16 bits long, a value which is much more realistically overflowed (65536).

By making a programmer request larger variables when they know they will be needed strikes a balance with respect to portability concerns and wasting space in binaries.

Type qualifiers

The C standard also talks about some qualifiers for variable types. For example `const` means that a variable will never be modified from its original value and `volatile` suggests to the compiler that this value might change outside program execution flow so the compiler must be careful not to re-order access to it in any way.

`signed` and `unsigned` are probably the two most important qualifiers; and they say if a variable can take on a negative value or not. We examine this in more detail below.

Qualifiers are all intended to pass extra information about how the variable will be used to the compiler. This means two things; the compiler can check if you are violating your own rules (e.g. writing to a `const` value) and it can make optimisations based upon the extra knowledge (examined in later chapters).

Standard Types

C99 realises that all these rules, sizes and portability concerns can become very confusing very quickly. To help, it provides a series of special types which can specify the exact properties of a variable. These are defined in `<stdint.h>` and have the form `qtypes_t` where `q` is a qualifier, `type` is the base type, `s` is the width in bits and `_t` is an extension so you know you are using the C99 defined types.

So for example `uint8_t` is an unsigned integer exactly 8 bits wide. Many other types are defined; the complete list is detailed in C99 17.8 or (more cryptically) in the header file. [3]

It is up to the system implementing the C99 standard to provide these types for you by mapping them to appropriate sized types on the target system; on Linux these headers are provided by the system libraries.

Types in action

Below in Example 2.3, “Example of warnings when types are not matched” we see an example of how types place restrictions on what operations are valid for a variable, and how the compiler can use this information to warn when variables are used in an incorrect fashion. In this code, we firstly assign an integer value into a `char` variable. Since the `char` variable is smaller, we loose the correct value of the integer. Further down, we attempt to assign a pointer to a `char` to memory we designated as an `integer`. This operation can be done; but it is not safe. The first example is run on a 32-bit Pentium machine, and the correct value is returned. However, as shown in the second example, on a 64-bit Itanium machine a pointer is 64 bits (8 bytes) long, but an integer is only 4 bytes long. Clearly, 8 bytes can not fit into 4! We can attempt to "fool" the compiler by *casting* the value before assigning it; note that in this case we have shot ourselves in the foot by doing this cast and ignoring the compiler warning since the smaller variable can not hold all the information from the pointer and we end up with an invalid address.

Example 2.3. Example of warnings when types are not matched

```c
/*
 * types.c
 */

#include <stdio.h>
#include <stdint.h>

int main(void)
{
    char a;
    char *p = "hello";

    int i;

    // moving a larger variable into a smaller one
    i = 0x12341234;
    a = i;
    i = a;
    printf("i is %d\n", i);
```

```
    // moving a pointer into an integer
    printf("p is %p\n", p);
    i = p;
    // "fooling" with casts
    i = (int)p;
    p = (char*)i;
    printf("p is %p\n", p);

    return 0;
}
```

```
$ uname -m
i686

$ gcc -Wall -o types types.c
types.c: In function 'main':
types.c:19: warning: assignment makes integer from pointer without a cast

$ ./types
i is 52
p is 0x80484e8
p is 0x80484e8

$ uname -m
ia64

$ gcc -Wall  -o types types.c
types.c: In function 'main':
types.c:19: warning: assignment makes integer from pointer without a cast
types.c:21: warning: cast from pointer to integer of different size
types.c:22: warning: cast to pointer from integer of different size

$ ./types
i is 52
p is 0x40000000000009e0
p is 0x9e0
```

Number Representation

Negative Values

With our modern base 10 numeral system we indicate a negative number by placing a minus (-) sign in front of it. When using binary we need to use a different system to indicate negative numbers.

There is only one scheme in common use on modern hardware, but C99 defines three acceptable methods for negative value representation.

Sign Bit

The most straight forward method is to simply say that one bit of the number indicates either a negative or positive value depending on it being set or not.

This is analogous to mathematical approach of having a `+` and `-` . This is fairly logical, and some of the original computers did represent negative numbers in this way. But using binary numbers opens up some other possibilities which make the life of hardware designers easier.

However, notice that the value `0` now has two equivalent values; one with the sign bit set and one without. Sometimes these values are referred to as `+0` and `-0` respectively.

One's Complement

One's complement simply applies the *not* operation to the positive number to represent the negative number. So, for example the value -90 (-0x5A) is represented by `~01011010 = 10100101` [4]

With this scheme the biggest advantage is that to add a negative number to a positive number no special logic is required, except that any additional carry left over must be added back to the final value. Consider

Table 2.15. One's Complement Addition

Decimal	Binary	Op
-90	10100101	+
100	01100100	
---	--------	
10	^{1}00001001	9
	00001010	10

If you add the bits one by one, you find you end up with a carry bit at the end (highlighted above). By adding this back to the original we end up with the correct value, 10

Again we still have the problem with two zeros being represented. Again no modern computer uses one's complement, mostly because there is a better scheme.

Two's Complement

Two's complement is just like one's complement, except the negative representation has *one* added to it and we discard any left over carry bit. So to continue with the example from before, `-90` would be `~01011010+1=10100101+1 = 10100110` .

This means there is a slightly odd symmetry in the numbers that can be represented; for example with an 8 bit integer we have `2^8 = 256` possible values; with our sign bit representation we could represent -127 thru 127 but with two's complement we can represent -127 thru 128. This is because we have removed the problem of having two zeros; consider that "negative zero" is `(~00000000 +1)=(11111111+1)=00000000` (note discarded carry bit).

Table 2.16. Two's Complement Addition

Decimal	Binary	Op
-90	10100110	+
100	01100100	
---	--------	
10	00001010	

You can see that by implementing two's complement hardware designers need only provide logic for addition circuits; subtraction can be done by two's complement negating the value to be subtracted and then adding the new value.

Similarly you could implement multiplication with repeated addition and division with repeated subtraction. Consequently two's complement can reduce all simple mathematical operations down to addition!

All modern computers use two's complement representation.

Sign-extension

Because of two's complement format, when increasing the size of signed value, it is important that the additional bits be *sign-extended*; that is, copied from the top-bit of the existing value.

For example, the value of an 32-bit `int` `-10` would be represented in two's complement binary as `11111111111111111111111111110110` . If one were to cast this to a 64-bit `long long int` , we would need to ensure that the additional 32-bits were set to `1` to maintain the same sign as the original.

Thanks to two's complement, it is sufficient to take the top bit of the exiting value and replace all the added bits with this value. This processes is referred to as *sign-extension* and is usually handled by the compiler in situations as defined by the language standard, with the processor generally providing special instructions to take a value and sign-extended it to some larger value.

Floating Point

So far we have only discussed integer or whole numbers; the class of numbers that can represent decimal values is called *floating point*.

To create a decimal number, we require some way to represent the concept of the decimal place in binary. The most common scheme for this is known as the *IEEE-754 floating point standard* because the standard is published by the Institute of Electric and Electronics Engineers. The scheme is conceptually quite simple and is somewhat analogous to "scientific notation".

In scientific notation the value `123.45` might commonly be represented as 1.2345×10^2 . We call `1.2345` the *mantissa* or *significand*, `10` is the *radix* and `2` is the *exponent*.

In the IEEE floating point model, we break up the available bits to represent the sign, mantissa and exponent of a decimal number. A decimal number is represented by $\text{sign} \times \text{significand} \times 2^{\text{exponent}}$.

The sign bit equates to either `1` or `-1` . Since we are working in binary, we always have the implied radix of `2` .

There are differing widths for a floating point value -- we examine below at only a 32 bit value. More bits allows greater precision.

Table 2.17. IEEE Floating Point

Sign	Exponent	Significand/Mantissa
S	EEEEEEEE	MMMMMMMMMMMMMMMMMMMMMMM

The other important factor is *bias* of the exponent. The exponent needs to be able to represent both positive and negative values, thus an implied value of `127` is subtracted from the exponent. For example, an exponent of `0` has an exponent field of `127` , `128` would represent `1` and `126` would represent `-1` .

Each bit of the significand adds a little more precision to the values we can represent. Consider the scientific notation representation of the value `198765` . We could write this as 1.98765×10^6 , which corresponds to a representation below

Table 2.18. Scientific Notation for 1.98765x10^6

10^0	.	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
1	.	9	8	7	6	5

Each additional digit allows a greater range of decimal values we can represent. In base 10, each digit after the decimal place increases the precision of our number by 10 times. For example, we can represent `0.0` through `0.9` (10 values) with one digit of decimal place, `0.00` through `0.99` (100 values) with two digits, and so on. In binary, rather than each additional digit

giving us 10 times the precision, we only get two times the precision, as illustrated in the table below. This means that our binary representation does not always map in a straight-forward manner to a decimal representation.

Table 2.19. Significands in binary

2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
1	.	1/2	1/4	1/8	1/16	1/32
1	.	0.5	0.25	0.125	0.0625	0.03125

With only one bit of precision, our fractional precision is not very big; we can only say that the fraction is either `0` or `0.5` . If we add another bit of precision, we can now say that the decimal value is one of either `0,0.25,0.5,0.75` . With another bit of precision we can now represent the values `0,0.125,0.25,0.375,0.5,0.625,0.75,0.875` .

Increasing the number of bits therefore allows us greater and greater precision. However, since the range of possible numbers is infinite we will never have enough bits to represent *any* possible value.

For example, if we only have two bits of precision and need to represent the value `0.3` we can only say that it is closest to `0.25` ; obviously this is insufficient for most any application. With 22 bits of significand we have a much finer resolution, but it is still not enough for most applications. A `double` value increases the number of significand bits to 52 (it also increases the range of exponent values too). Some hardware has an 84-bit float, with a full 64 bits of significand. 64 bits allows a tremendous precision and should be suitable for all but the most demanding of applications (XXX is this sufficient to represent a length to less than the size of an atom?)

Example 2.4. Floats versus Doubles

```
$ cat float.c
#include <stdio.h>

int main(void)
{
        float a = 0.45;
        float b = 8.0;

        double ad = 0.45;
        double bd = 8.0;

        printf("float+float, 6dp    : %f\n", a+b);
        printf("double+double, 6dp  : %f\n", ad+bd);
        printf("float+float, 20dp   : %10.20f\n", a+b);
        printf("dobule+double, 20dp : %10.20f\n", ad+bd);

        return 0;
}

$ gcc -o float float.c

$ ./float
float+float, 6dp    : 8.450000
double+double, 6dp  : 8.450000
float+float, 20dp   : 8.44999998807907104492
```

```
dobule+double, 20dp : 8.44999999999999928946

$ python
Python 2.4.4 (#2, Oct 20 2006, 00:23:25)
[GCC 4.1.2 20061015 (prerelease) (Debian 4.1.1-16.1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 8.0 + 0.45
8.4499999999999993
```

A practical example is illustrated above. Notice that for the default 6 decimal places of precision given by `printf` both answers are the same, since they are rounded up correctly. However, when asked to give the results to a larger precision, in this case 20 decimal places, we can see the results start to diverge. The code using `doubles` has a more accurate result, but it is still not *exactly* correct. We can also see that programmers not explicitly dealing with `float` values still have problems with precision of variables!

Normalised Values

In scientific notation, we can represent a value in many different ways. For example, $10023 \times 10^{0} = 1002.3 \times 10^{1} = 100.23 \times 10^{2}$. We thus define the *normalised* version as the one where `1/radix <= significand < 1` . In binary this ensures that the leftmost bit of the significand is *always one*. Knowing this, we can gain an extra bit of precision by having the standard say that the leftmost bit being one is implied.

Table 2.20. Example of normalising 0.375

2^{0}	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	Exponent	Calculation
0	.	0	1	1	0	0	2^{0}	(0.25+0.125) × 1 = 0.375
0	.	1	1	0	0	0	2^{-1}	(0.5+0.25)×.5=0.375
1	.	1	0	0	0	0	2^{-2}	(1+0.5)×0.25=0.375

As you can see above, we can make the value normalised by moving the bits upwards as long as we compensate by increasing the exponent.

Normalisation Tricks

A common problem programmers face is finding the first set bit in a bitfield. Consider the bitfield `0100` ; from the right the first set bit would be bit `2` (starting from zero, as is conventional).

The standard way to find this value is to shift right, check if the uppermost bit is a `1` and either terminate or repeat. This is a slow process; if the bitfield is 64 bits long and only the very last bit is set, you must go through all the preceding 63 bits!

However, if this bitfield value were the signficand of a floating point number and we were to normalise it, the value of the exponent would tell us how many times it was shifted. The process of normalising a number is generally built into the floating point hardware unit on the processor, so operates very fast; usually much faster than the repeated shift and check operations.

The example program below illustrates two methods of finding the first set bit on an Itanium processor. The Itanium, like most server processors, has support for an 80-bit *extended* floating point type, with a 64-bit significand. This means a `unsigned long` neatly fits into the significand of a `long double` . When the value is loaded it is normalised, and and thus by reading the exponent value (minus the 16 bit bias) we can see how far it was shifted.

Example 2.5. Program to find first set bit

```
#include <stdio.h>

int main(void)
{
    //  in binary = 1000 0000 0000 0000
    //  bit num     5432 1098 7654 3210
    int i = 0x8000;
    int count = 0;
    while ( !(i & 0x1) ) {
            count ++;
            i = i >> 1;
    }
    printf("First non-zero (slow) is %d\n", count);

    // this value is normalised when it is loaded
    long double d = 0x8000UL;
    long exp;

    // Itanium "get floating point exponent" instruction
    asm ("getf.exp %0=%1" : "=r"(exp) : "f"(d));

    // note exponent include bias
    printf("The first non-zero (fast) is %d\n", exp - 65535);

}
```

Bringing it together

In the example code below we extract the components of a floating point number and print out the value it represents. This will only work for a 32 bit floating point value in the IEEE format; however this is common for most architectures with the `float` type.

Example 2.6. Examining Floats

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* return 2^n */
int two_to_pos(int n)
{
    if (n == 0)
            return 1;
    return 2 * two_to_pos(n - 1);
}

double two_to_neg(int n)
```

```
{
    if (n == 0)
            return 1;
    return 1.0 / (two_to_pos(abs(n)));
}

double two_to(int n)
{
    if (n >= 0)
            return two_to_pos(n);
    if (n < 0)
            return two_to_neg(n);
    return 0;
}

/* Go through some memory "m" which is the 24 bit significand of a
   floating point number.  We work "backwards" from the bits
   furthest on the right, for no particular reason. */
double calc_float(int m, int bit)
{
    /* 23 bits; this terminates recursion */
    if (bit > 23)
            return 0;

    /* if the bit is set, it represents the value 1/2^bit */
    if ((m >> bit) & 1)
            return 1.0L/two_to(23 - bit) + calc_float(m, bit + 1);

    /* otherwise go to the next bit */
    return calc_float(m, bit + 1);
}

int main(int argc, char *argv[])
{
    float f;
    int m,i,sign,exponent,significand;

    if (argc != 2)
    {
            printf("usage: float 123.456\n");
            exit(1);
    }

    if (sscanf(argv[1], "%f", &f) != 1)
    {
            printf("invalid input\n");
```

```
            exit(1);
        }

        /* We need to "fool" the compiler, as if we start to use casts
           (e.g. (int)f) it will actually do a conversion for us.  We
           want access to the raw bits, so we just copy it into a same
           sized variable. */
        memcpy(&m, &f, 4);

        /* The sign bit is the first bit */
        sign = (m >> 31) & 0x1;

        /* Exponent is 8 bits following the sign bit */
        exponent = ((m >> 23) & 0xFF) - 127;

        /* Significand fills out the float, the first bit is implied
           to be 1, hence the 24 bit OR value below. */
        significand = (m & 0x7FFFFF) | 0x800000;

        /* print out a power representation */
        printf("%f = %d * (", f, sign ? -1 : 1);
        for(i = 23 ; i >= 0 ; i--)
        {
                if ((significand >> i) & 1)
                        printf("%s1/2^%d", (i == 23) ? "" : " + ",
                               23-i);
        }
        printf(") * 2^%d\n", exponent);

        /* print out a fractional representation */
        printf("%f = %d * (", f, sign ? -1 : 1);
        for(i = 23 ; i >= 0 ; i--)
        {
                if ((significand >> i) & 1)
                        printf("%s1/%d", (i == 23) ? "" : " + ",
                               (int)two_to(23-i));
        }
        printf(") * 2^%d\n", exponent);

        /* convert this into decimal and print it out */
        printf("%f = %d * %.12g * %f\n",
               f,
               (sign ? -1 : 1),
               calc_float(significand, 0),
               two_to(exponent));

```

```
        /* do the math this time */
        printf("%f = %.12g\n",
               f,
               (sign ? -1 : 1) *
               calc_float(significand, 0) *
               two_to(exponent)
                );

        return 0;
}
```

Sample output of the value `8.45`, which we previously examined, is shown below.

Example 2.7. Analysis of `8.45`

```
$ ./float 8.45
8.450000 = 1 * (1/2^0 + 1/2^5 + 1/2^6 + 1/2^7 + 1/2^10 + 1/2^11 + 1/2^14 + 1/2^15 +
8.450000 = 1 * (1/1 + 1/32 + 1/64 + 1/128 + 1/1024 + 1/2048 + 1/16384 + 1/32768 + 1/
8.450000 = 1 * 1.05624997616 * 8.000000
8.450000 = 8.44999980927
```

From this example, we get some idea of how the inaccuracies creep into our floating point numbers.

---

[3] Note that C99 also has portability helpers for `printf`. The `PRI` macros in `<inttypes.h>` can be used as specifiers for types of specified sizes. Again see the standard or pull apart the headers for full information.

[4] The `~` operator is the C language operator to apply `NOT` to the value. It is also occasionally called the one's complement operator, for obvious reasons now!

This page titled 1.2: Types and Number Representation is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

2: The Basics - An Overview

2.1: Introduction to Operating Systems
2.2: Starting with the Basics

2: The Basics - An Overview is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

2.1: Introduction to Operating Systems

Introduction to Operating System

An operating system acts as an intermediary between the user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.

An operating system is a software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

Operating System

Definition:

- An operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware.
- A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being application programs.
- An operating system is concerned with the allocation of resources and services, such as memory, processors, devices, and information. The operating system correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system.

Functions of Operating system

Operating system performs three functions:

1. **Convenience:** An OS makes a computer more convenient to use.
2. **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.
3. **Ability to Evolve:** An OS should be constructed in such a way as to permit the effective development, testing and introduction of new system functions at the same time without interfering with service.

Operating system as User Interface –

1. User
2. System and application programs
3. Operating system
4. Hardware

Every general-purpose computer consists of the hardware, operating system, system programs, and application programs. The hardware consists of memory, CPU, ALU, and I/O devices, peripheral device, and storage device. System program consists of compilers, loaders, editors, OS, etc. The application program consists of business programs, database programs.

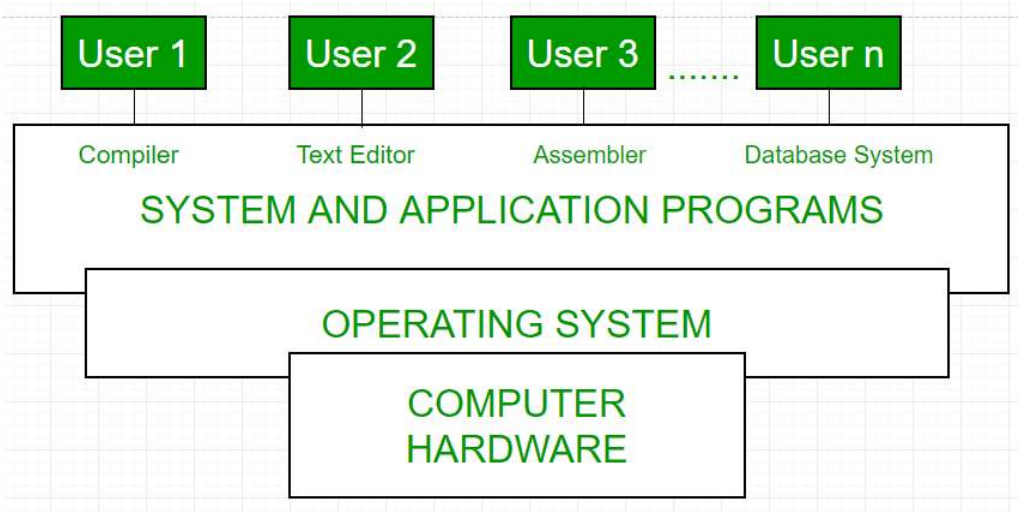


Figure 2.1.1: Conceptual view of a computer system ("Conceptual view of a computer system" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0)

Every computer must have an operating system to run other programs. The operating system coordinates the use of the hardware among the various system programs and application programs for various users. It simply provides an environment within which other programs can do useful work.

The operating system is a set of special programs that run on a computer system that allows it to work properly. It performs basic tasks such as recognizing input from the keyboard, keeping track of files and directories on the disk, sending output to the display screen and controlling peripheral devices.
OS is designed to serve two basic purposes:

1. It controls the allocation and use of the computing System's resources among the various user and tasks.
2. It provides an interface between the computer hardware and the programmer that simplifies and makes feasible for coding, creation, debugging of application programs.

The Operating system must support the following tasks. The task are:

1. Provides the facilities to create, modification of programs and data files using an editor.
2. Access to the compiler for translating the user program from high level language to machine language.
3. Provide a loader program to move the compiled program code to the computer's memory for execution.
4. Provide routines that handle the details of I/O programming.

I/O System Management

One of the important jobs of an Operating System is to manage the operations of various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

The I/O system of an OS works by taking I/O request from an application software and sending it to the physical device, which could be an input or output device then it takes whatever response comes back from the device and sends it to the application.

Components of I/O Hardware

- I/O Device
- Device Driver
- Device Controller

I/O Device:

I/O devices such as storage, communications, user-interface, and others communicate with the computer via signals sent over wires or through the air. Devices connect with the computer via ports, e.g. a serial or parallel port. A common set of wires connecting

multiple devices is termed a bus.

I/O devices can be divided into two categories:

- Block devices − A block device is one with which the device driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-OnKey etc.
- Character devices − A character device is one with which the device driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc

Device Driver: Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

Device Controller: The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

History of Operating system

Operating system has been evolving through the years. Following Table shows the history of OS.

Generation	Year	Electronic device used	Types of OS Device
First	1945-55	Vacuum Tubes	Plug Boards
Second	1955-65	Transistors	Batch Systems
Third	1965-80	Integrated Circuits(IC)	Multiprogramming
Fourth	Since 1980	Large Scale Integration	PC

This page titled 2.1: Introduction to Operating Systems is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

2.2: Starting with the Basics

Processor

The processor is an important part of a computer architecture, without it nothing would happen. It is a programmable device that takes input, perform some arithmetic and logical operations and produce some output. In simple words, a processor is a digital device on a chip which can fetch instruction from memory, decode and execute them and provide results.

Basics of a Processor –
A processor takes a bunch of instructions in machine language and executes them, telling the processor what it has to do. Processors performs three basic operations while executing the instruction:

1. It performs some basic operations like addition, subtraction, multiplication, division and some logical operations using its Arithmetic and Logical Unit (ALU).
2. Data in the processor can move from one location to another.
3. It has a Program Counter (PC) register that stores the address of next instruction based on the value of PC.

A typical processor structure looks like this.

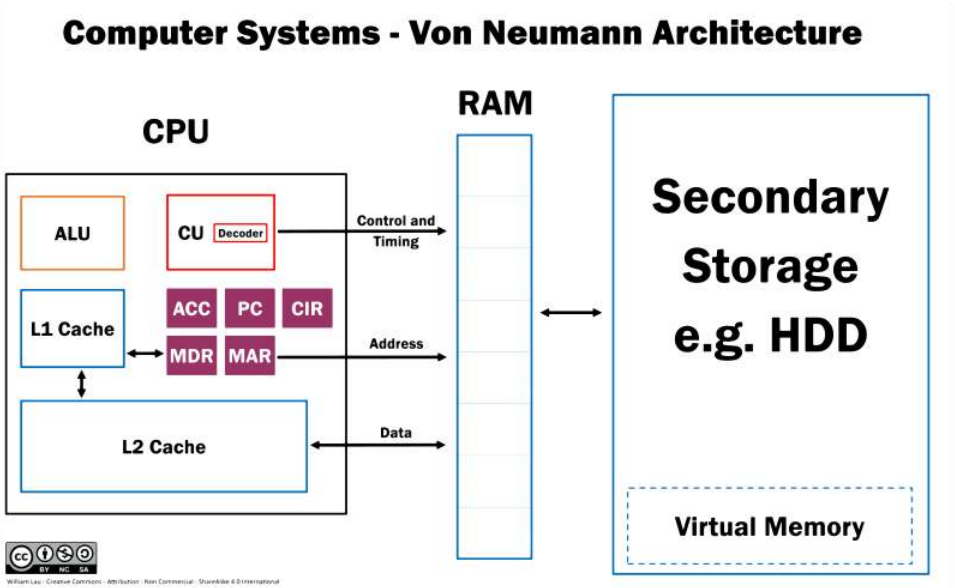


Figure 2.2.1: Von Neumann Architecture. ("File:Computer Systems - Von Neumann Architecture Large poster anchor chart.svg" by BotMultichillT, Wikimedia Commons is licensed under CC BY-NC-SA 4.0)

Basic Processor Terminology

- **Control Unit (CU)**
  A control unit (CU) handles all processor control signals. It directs all input and output flow, fetches the code for instructions and controlling how data moves around the system.
- **Arithmetic and Logic Unit (ALU)**
  The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons. It performs Logical Operations, Bit Shifting Operations, and Arithmetic Operation.
- **Main Memory Unit (Registers)**
  1. **Accumulator (ACC):** Stores the results of calculations made by ALU.
  2. **Program Counter (PC):** Keeps track of the memory location of the next instructions to be dealt with. The PC then passes this next address to Memory Address Register (MAR).

3. **Memory Address Register (MAR):** It stores the memory locations of instructions that need to be fetched from memory or stored into memory.
4. **Memory Data Register (MDR):** It stores instructions fetched from memory or any data that is to be transferred to, and stored in, memory.
5. **Current Instruction Register (CIR):** It stores the most recently fetched instructions while it is waiting to be coded and executed.
6. **Instruction Buffer Register (IBR):** The instruction that is not to be executed immediately is placed in the instruction buffer register IBR.

- **Input/Output Devices –** Program or data is read into main memory from the *input device* or secondary storage under the control of CPU input instruction. *Output devices* are used to output the information from a computer.
- **Buses –** Data is transmitted from one part of a computer to another, connecting all major internal components to the CPU and memory, by the means of Buses. Types:
  1. **Data Bus (Data):** It carries data among the memory unit, the I/O devices, and the processor.
  2. **Address Bus (Address):** It carries the address of data (not the actual data) between memory and processor.
  3. **Control Bus (Control and Timing):** It carries control commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer.

Memory

Memory attached to the CPU is used for storage of data and instructions and is called internal memory The internal memory is divided into many storage locations, each of which can store data or instructions. Each memory location is of the same size and has an address. With the help of the address, the computer can read any memory location easily without having to search the entire memory. when a program is executed, it's data is copied to the internal memory and is stored in the memory till the end of the execution. The internal memory is also called the Primary memory or Main memory. This memory is also called as RAM, i.e. Random Access Memory. The time of access of data is independent of its location in memory, therefore this memory is also called Random Access memory (RAM).

I/O Modules

The method that is used to transfer information between main memory and external I/O devices is known as the I/O interface, or I/O modules. The CPU is interfaced using special communication links by the peripherals connected to any computer system. These communication links are used to resolve the differences between CPU and peripheral. There exists special hardware components between CPU and peripherals to supervise and synchronize all the input and output transfers that are called interface units.

Mode of Transfer:

The binary information that is received from an external device is usually stored in the memory unit. The information that is transferred from the CPU to the external device is originated from the memory unit. CPU merely processes the information but the source and target is always the memory unit. Data transfer between CPU and the I/O devices may be done in different modes.

Data transfer to and from the peripherals may be done in any of the three possible ways

1. **Programmed I/O:** is the result of the I/O instructions written in the program's code. Each data transfer is initiated by an instruction in the program. Usually the transfer is from a CPU register and/or memory. In this case it requires constant monitoring by the CPU of the peripheral devices.
2. **Interrupt- initiated I/O:** using an interrupt facility and special commands to issue an interrupt request signal whenever data is available from any device. In the meantime the CPU can proceed processing other programs. The interface meanwhile keeps monitoring the device. When it is determined that the device is ready for a data transfer it initiates an interrupt request signal to the CPU. Upon detection of an external interrupt signal the CPU momentarily stops the task it was processing, and services program that was waiting on the interrupt to process the I/O transfer> Once the interrupt is satisfied, the CPU then return to the task it was originally processing.
3. **Direct memory access(DMA)**: The data transfer between a fast storage media such as magnetic disk and main memory is limited by the speed of the CPU. Thus we can allow the peripherals directly communicate with each other using the memory

buses, removing the intervention of the CPU. This type of data transfer technique is known as direct memory access, or DMA. During DMA the CPU is idle and it has no control over the memory buses. The DMA controller takes over the buses to manage the transfer directly between the I/O devices and the memory unit.

Adapted from:
"Introduction of Microprocessor" by DikshaTewari, **Geeks for Geeks** is licensed under CC BY-SA 4.0
"Last Minute Notes Computer Organization" by Geeks for Geeks is licensed under CC BY-SA 4.0
"Functional Components of a Computer" by aishwaryaagarwal2, Geeks for Geeks is licensed under CC BY-SA 4.0
"System Bus Design" by deepak, Geeks for Geeks is licensed under CC BY-SA 4.0
"I/O Interface (Interrupt and DMA Mode)" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 2.2: Starting with the Basics is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

3: The Operating System

3.1: The Role of the Operating System
3.2: Operating System Organisation
3.3: System Calls
3.4: Privileges
3.5: Function of the Operating System
3.6: Types of Operating Systems
3.6.1: Types of Operating Systems (continued)
3.6.2: Types of Operating Systems (continued)
3.7: Difference between multitasking, multithreading and multiprocessing
3.7.1: Difference between multitasking, multithreading and multiprocessing (continued)
3.7.2: Difference between Multiprogramming, multitasking, multithreading and multiprocessing (continued)

3: The Operating System is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

The role of the operating system

The operating system underpins the entire operation of the modern computer.

Abstraction of hardware

The fundamental operation of the operating system (OS) is to abstract the hardware to the programmer and user. The operating system provides generic interfaces to services provided by the underlying hardware.

In a world without operating systems, every programmer would need to know the most intimate details of the underlying hardware to get anything to run. Worse still, their programs would not run on other hardware, even if that hardware has only slight differences.

Multitasking

We expect modern computers to do many different things at once, and we need some way to arbitrate between all the different programs running on the system. It is the operating systems job to allow this to happen seamlessly.

The operating system is responsible for *resource management* within the system. Many tasks will be competing for the resources of the system as it runs, including processor time, memory, disk and user input. The job of the operating system is to arbitrate these resources to the multiple tasks and allow them access in an orderly fashion. You have probably experienced when this *fails* as it usually ends up with your computer crashing (the famous "blue screen of death" for example).

Standardised Interfaces

Programmers want to write programs that will run on as many different hardware platforms as possible. By having operating system support for standardised interfaces, programmers can get this functionality.

For example, if the function to open a file on one system is `open()` , on another is `open_file()` and on yet another `openf()` programmers will have the dual problem of having to remember what each system does and their programs will not work on multiple systems.

The Portable Operating System Interface (POSIX)[9] is a very important standard implemented by UNIX type operating systems. Microsoft Windows has similar proprietary standards.

Security

On multi-user systems, security is very important. As the arbitrator of access to the system the operating system is responsible for ensuring that only those with the correct permissions can access resources.

For example if a file is owned by one user, another user should not be allowed to open and read it. However there also need to be mechanisms to share that file safely between the users should they want it.

Operating systems are large and complex programs, and often security issues will be found. Often a virus or worm will take advantage of these bugs to access resources it should not be allowed to, such as your files or network connection; to fight them you must install *patches* or updates provided by your operating system vendor.

Performance

As the operating system provides so many services to the computer, its performance is critical. Many parts of the operating system run extremely frequently, so even an overhead of just a few processor cycles can add up to a big decrease in overall system performance.

The operating system needs to exploit the features of the underlying hardware to make sure it is getting the best possible performance for the operations, and consequently systems programmers need to understand the intimate details of the architecture they are building for.

In many cases the systems programmers job is about deciding on policies for the system. Often the case that the side effects of making one part of the operating system run faster will make another part run slower or less efficiently. Systems programmers need

to understand all these trade offs when they are building their operating system.

---

[9] The X comes from *Unix*, from which the standard grew. Today, POSIX is the same thing as the Single UNIX Specification Version 3 or ISO/IEC 9945:2002. This is a free standard, available online.

Once upon a time, the Single UNIX specification and the POSIX Standards were separate entities. The Single UNIX specification was released by a consortium called the "Open Group", and was freely available as per their requirements. The latest version is The Single Unix Specification Version 3.

The IEEE POSIX standards were released as IEEE Std 1003.[insert various years, revisions here], and were not freely available. The latest version is IEEE 1003.1-2001 and is equivalent to the Single Unix Specification Version 3.

Thus finally the two separate standards were merged into what is known as the Single UNIX Specification Version 3, which is also standardised by the ISO under ISO/IEC 9945:2002. This happened early in 2002. So when people talk about POSIX, SUS3 or ISO/IEC 9945:2002 they all mean the same thing!

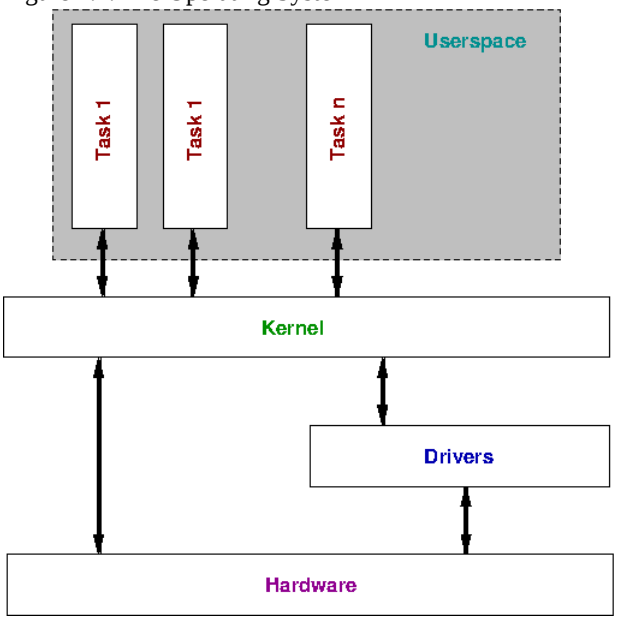
This page titled 3.1: The Role of the Operating System is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

3.2: Operating System Organisation

Operating System Organisation

The operating system is roughly organised as in the figure below.

Figure 4.1. The Operating System



The organisation of the kernel. Processes the kernel is running live in *userspace*, and the kernel talks both directly to hardware and through *drivers*.

The Kernel

The kernel *is* the operating system. As the figure illustrates, the kernel communicates to hardware both directly and through *drivers*.

Just as the kernel abstracts the hardware to user programs, drivers abstract hardware to the kernel. For example there are many different types of graphic card, each one with slightly different features. As long as the kernel exports an API, people who have access to the specifications for the hardware can write drivers to implement that API. This way the kernel can access many different types of hardware.

The kernel is generally what we called *privileged*. As you will learn, the hardware has important roles to play in running multiple tasks and keeping the system secure, but these rules do not apply to the kernel. We know that the kernel must handle programs that crash (remember it is the operating systems job arbitrate between multiple programs running on the same system, and there is no guarantee that they will behave), but if any internal part of the operating system crashes chances are the entire system will become useless. Similarly security issues can be exploited by user processes to escalate themselves to the privilege level of the kernel; at that point they can access any part of the system completely unchecked.

Monolithic v Microkernels

One debate that is often comes up surrounding operating systems is whether the kernel should be a *microkernel* or *monolithic*.

The monolithic approach is the most common, as taken by most common Unixes (such as Linux). In this model the core privileged kernel is large, containing hardware drivers, file system accesses controls, permissions checking and services such as Network File System (NFS).

Since the kernel is always privileged, if any part of it crashes the whole system has the potential to comes to a halt. If one driver has a bug it can overwrite any memory in the system with no problems, ultimately causing the system to crash.

A microkernel architecture tries to minimise this possibility by making the privileged part of the kernel as small as possible. This means that most of the system runs as unprivileged programs, limiting the harm that any one crashing component can influence. For example, drivers for hardware can run in separate processes, so if one goes astray it can not overwrite any memory but that allocated to it.

Whilst this sounds like the most obvious idea, the problem comes back two main issues

1. Performance is decreased. Talking between many different components can decrease performance.
2. It is slightly more difficult for the programmer.

Both of these criticisms come because to keep separation between components most microkernels are implemented with a *message passing* based system, commonly referred to as *inter-process communication* or IPC. Communicating between individual components happens via discrete messages which must be bundled up, sent to the other component, unbundled, operated upon, re-bundled up and sent back, and then unbundled again to get the result.

This is a lot of steps for what might be a fairly simple request from a foreign component. Obviously one request might make the other component do more requests of even more components, and the problem can multiply. Slow message passing implementations were largely responsible for the poor performance of early microkernel systems, and the concepts of passing messages are slightly harder for programmers to program for. The enhanced protection from having components run separately was not sufficient to overcome these hurdles in early microkernel systems, so they fell out of fashion.

In a monolithic kernel calls between components are simple function calls, as all programmers are familiar with.

There is no definitive answer as to which is the best organisation, and it has started many arguments in both academic and non-academic circles. Hopefully as you learn more about operating systems you will be able to make up your own mind!

Modules

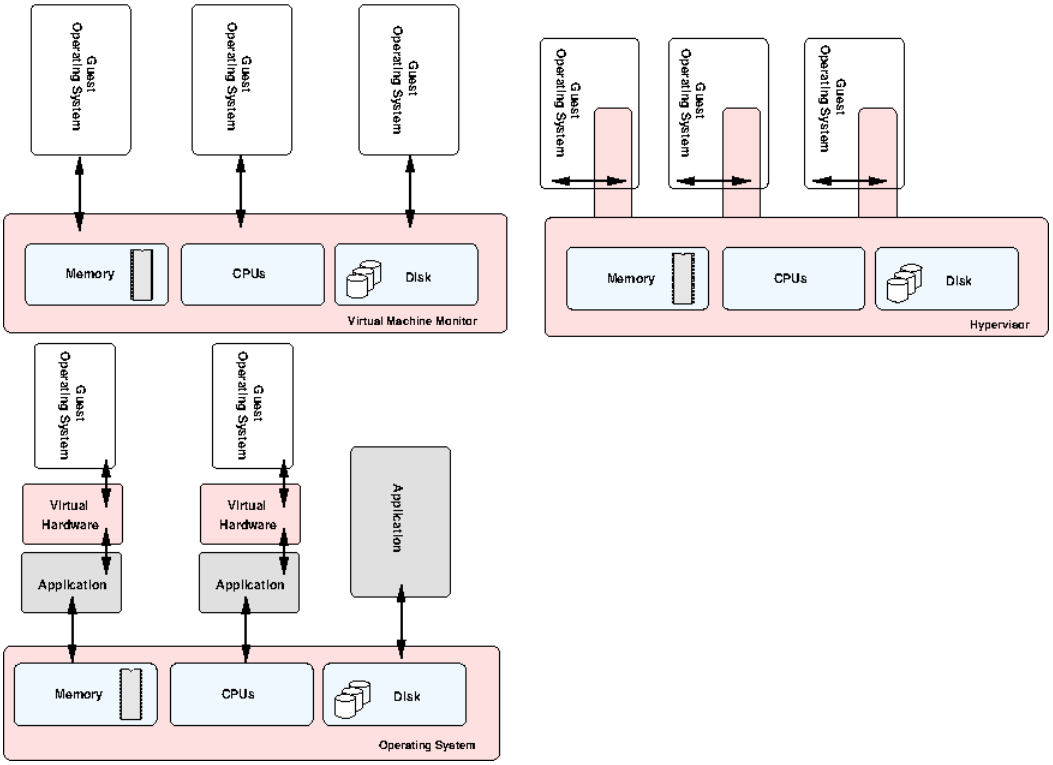
The Linux kernel implements a module system, where drivers can loaded into the running kernel "on the fly" as they are required. This is good in that drivers, which make up a large part of operating system code, are not loaded for devices that are not present in the system. Someone who wants to make the most generic kernel possible (i.e. runs on lots of different hardware, such as RedHat or Debian) can include most drivers as modules which are only loaded if the system it is running on has the hardware available.

However, the modules are loaded directly in the privileged kernel and operate at the same privilege level as the rest of the kernel, so the system is still considered a monolithic kernel.

Virtualisation

Closely related to kernel is the concept of virtualisation of hardware. Modern computers are very powerful, and often it is useful to not think of them as one whole system but split a single physical computer up into separate "virtual" machines. Each of these virtual machines looks for all intents and purposes as a completely separate machine, although physically they are all in the same box, in the same place.

Figure 4.2. The Operating System



Some different virtualisation methods.

This can be organised in many different ways. In the simplest case, a small *virtual machine monitor* can run directly on the hardware and provide an interface to the guest operating systems running on top. This VMM is often often called a hypervisor (from the word "supervisor")[10]. In fact, the operating system on top may have no idea that the hypervisor is even there at all, as the hypervisor presents what appears to be a complete system. It intercepts operations between the guest operating system and hardware and only presents a subset of the system resources to each.

This is often used on large machines (with many CPUs and much RAM) to implement *partitioning*. This means the machine can be split up into smaller virtual machines. Often you can allocate more resources to running systems on the fly, as requirements dictate. The hypervisors on many large IBM machines are actually quite complicated affairs, with many millions of lines of code. It provides a multitude of system management services.

Another option is to have the operating system aware of the underlying hypervisor, and request system resources through it. This is sometimes referred to as *paravirtualisation* due to its halfway nature. This is similar to the way early versions of the Xen system works and is a compromise solution. It hopefully provides better performance since the operating system is explicitly asking for system resources from the hypervisor when required, rather than the hypervisor having to work things out dynamically.

Finally, you may have a situation where an application running on top of the existing operating system presents a virtualised system (including CPU, memory, BIOS, disk, etc) which a plain operating system can run on. The application converts the requests to hardware through to the underlying hardware via the existing operating system. This is similar to how VMWare works. This approach has many overheads, as the application process has to emulate an entire system and convert everything to requests from the underlying operating system. However, this lets you emulate an entirely different architecture all together, as you can dynamically translate the instructions from one processor type to another (as the Rosetta system does with Apple software which moved from the PowerPC processor to Intel based processors).

Performance is major concern when using any of these virtualisation techniques, as what was once fast operations directly on hardware need to make their way through layers of abstraction.

Intel have discussed hardware support for virtualisation soon to be coming in their latest processors. These extensions work by raising a special exception for operations that might require the intervention of a virtual machine monitor. Thus the processor looks

the same as a non-virtualised processor to the application running on it, but when that application makes requests for resources that might be shared between other guest operating systems the virtual machine monitor can be invoked.

This provides superior performance because the virtual machine monitor does not need to monitor every operation to see if it is safe, but can wait until the processor notifies that something *unsafe* has happened.

Covert Channels

This is a digression, but an interesting security flaw relating to virtualised machines. If the partitioning of the system is not static, but rather *dynamic*, there is a potential security issue involved.

In a dynamic system, resources are allocated to the operating systems running on top as required. Thus if one is doing particularly CPU intensive operations whilst the other is waiting on data to come from disks, more of the CPU power will be given to the first task. In a static system, each would get 50% an the unused portion would go to waste.

Dynamic allocation actually opens up a communications channel between the two operating systems. Anywhere that two states can be indicated is sufficient to communicate in binary. Imagine both systems are extremely secure, and no information should be able to pass between one and the other, ever. Two people with access could collude to pass information between themselves by writing two programs that try to take large amounts of resources at the same time.

When one takes a large amount of memory there is less available for the other. If both keep track of the maximum allocations, a bit of information can be transferred. Say they make a pact to check every second if they can allocate this large amount of memory. If the target can, that is considered binary 0, and if it can not (the other machine has all the memory), that is considered binary 1. A data rate of one bit per second is not astounding, but information is flowing.

This is called a *covert channel*, and whilst admittedly far fetched there have been examples of security breaches from such mechanisms. It just goes to show that the life of a systems programmer is never simple!

Userspace

We call the theoretical place where programs run by the user *userspace*. Each program runs in userspace, talking to the kernel through *system calls* (discussed below).

As previously discussed, userspace is *unprivileged*. User programs can only do a limited range of things, and should never be able to crash other programs, even if they crash themselves.

---

[10] In fact, the hypervisor shares much in common with a micro-kernel; both strive to be small layers to present the hardware in a safe fashion to layers above it.

This page titled 3.2: Operating System Organisation is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

3.3: System Calls

System Calls

Overview

System calls are how userspace programs interact with the kernel. The general principle behind how they work is described below.

System call numbers

Each and every system call has a *system call number* which is known by both the userspace and the kernel. For example, both know that system call number 10 is `open()`, system call number 11 is `read()`, etc.

The *Application Binary Interface* (ABI) is very similar to an API but rather than being for software is for hardware. The API will define which register the system call number should be put in so the kernel can find it when it is asked to do the system call.

Arguments

System calls are no good without arguments; for example `open()` needs to tell the kernel exactly *what* file to open. Once again the ABI will define which registers arguments should be put into for the system call.

The trap

To actually perform the system call, there needs to be some way to communicate to the kernel we wish to make a system call. All architectures define an instruction, usually called `break` or something similar, that signals to the hardware we wish to make a system call.

Specifically, this instruction will tell the hardware to modify the instruction pointer to point to the kernels system call handler (when the operating system sets its self up it tells the hardware where its system call handler lives). So once the userspace calls the break instruction, it has lost control of the program and passed it over to the kernel.

The rest of the operation is fairly straight forward. The kernel looks in the predefined register for the system call number, and looks it up in a table to see which function it should call. This function is called, does what it needs to do, and places its return value into *another* register defined by the ABI as the return register.

The final step is for the kernel to make a jump instruction back to the userspace program, so it can continue off where it left from. The userpsace program gets the data it needs from the return register, and continues happily on its way!

Although the details of the process can get quite hairy, this is basically all their is to a system call.

libc

Although you can do all of the above by hand for each system call, system libraries usually do most of the work for you. The standard library that deals with system calls on UNIX like systems is `libc`; we will learn more about its roles in future weeks.

Analysing a system call

As the system libraries usually deal with making systems call for you, we need to do some low level hacking to illustrate exactly how the system calls work.

We will illustrate how probably the most simple system call, `getpid()`, works. This call takes no arguments and returns the ID of the currently running program (or process; we'll look more at the process in later weeks).

Example 4.1. getpid() example

```
#include <stdio.h>

/* for syscall() */
#include <sys/syscall.h>
#include <unistd.h>
```

```
/* system call numbers */
#include <asm/unistd.h>

void function(void)
{
    int pid;

    pid = __syscall(__NR_getpid);
}
```

We start by writing a small C program which we can start to illustrate the mechanism behind system calls. The first thing to note is that there is a `syscall` argument provided by the system libraries for directly making system calls. This provides an easy way for programmers to directly make systems calls without having to know the exact assembly language routines for making the call on their hardware. So why do we use `getpid()` at all? Firstly, it is much clearer to use a symbolic function name in your code. However, more importantly, `getpid()` may work in very different ways on different systems. For example, on Linux the `getpid()` call can be cached, so if it is run twice the system library will not take the penalty of having to make an entire system call to find out the same information again.

By convention under Linux, system calls numbers are defined in the `asm/unistd.h` file from the kernel source. Being in the `asm` subdirectory, this is different for each architecture Linux runs on. Again by convention, system calls numbers are given a `#define` name consisting of `__NR_` . Thus you can see our code will be making the `getpid` system call, storing the value in `pid` .

We will have a look at how several architectures implement this code under the hood. We're going to look at real code, so things can get quite hairy. But stick with it -- this is *exactly* how your system works!

PowerPC

PowerPC is a RISC architecture common in older Apple computers, and the core of devices such as the latest version of the Xbox.

Example 4.2. PowerPC system call example

```

/* On powerpc a system call basically clobbers the same registers like a
 * function call, with the exception of LR (which is needed for the
 * "sc; bnslr" sequence) and CR (where only CR0.SO is clobbered to signal
 * an error return status).
 */

#define __syscall_nr(nr, type, name, args...)                                \
    unsigned long __sc_ret, __sc_err;                                 \
    {                                                                 \
            register unsigned long __sc_0  __asm__ ("r0");            \
            register unsigned long __sc_3  __asm__ ("r3");            \
            register unsigned long __sc_4  __asm__ ("r4");            \
            register unsigned long __sc_5  __asm__ ("r5");            \
            register unsigned long __sc_6  __asm__ ("r6");            \
            register unsigned long __sc_7  __asm__ ("r7");            \
                                                                      \
            __sc_loadargs_##nr(name, args);                           \
```

```
                __asm__ __volatile__                                    \
                        ("sc           \n\t"                            \
                         "mfcr %0      "                                \
                        : "=&r" (__sc_0),                               \
                          "=&r" (__sc_3),  "=&r" (__sc_4),              \
                          "=&r" (__sc_5),  "=&r" (__sc_6),              \
                          "=&r" (__sc_7)                                \
                        : __sc_asm_input_##nr                           \
                        : "cr0", "ctr", "memory",                       \
                          "r8", "r9", "r10","r11", "r12");              \
                __sc_ret = __sc_3;                                      \
                __sc_err = __sc_0;                                      \
        }                                                               \
        if (__sc_err & 0x10000000)                                      \
        {                                                               \
                errno = __sc_ret;                                       \
                __sc_ret = -1;                                          \
        }                                                               \
        return (type) __sc_ret

#define __sc_loadargs_0(name, dummy...)                                 \
    __sc_0 = __NR_##name
#define __sc_loadargs_1(name, arg1)                                     \
    __sc_loadargs_0(name);                                              \
    __sc_3 = (unsigned long) (arg1)
#define __sc_loadargs_2(name, arg1, arg2)                               \
    __sc_loadargs_1(name, arg1);                                        \
    __sc_4 = (unsigned long) (arg2)
#define __sc_loadargs_3(name, arg1, arg2, arg3)                         \
    __sc_loadargs_2(name, arg1, arg2);                                  \
    __sc_5 = (unsigned long) (arg3)
#define __sc_loadargs_4(name, arg1, arg2, arg3, arg4)                   \
    __sc_loadargs_3(name, arg1, arg2, arg3);                            \
    __sc_6 = (unsigned long) (arg4)
#define __sc_loadargs_5(name, arg1, arg2, arg3, arg4, arg5)             \
    __sc_loadargs_4(name, arg1, arg2, arg3, arg4);                      \
    __sc_7 = (unsigned long) (arg5)

#define __sc_asm_input_0 "0" (__sc_0)
#define __sc_asm_input_1 __sc_asm_input_0, "1" (__sc_3)
#define __sc_asm_input_2 __sc_asm_input_1, "2" (__sc_4)
#define __sc_asm_input_3 __sc_asm_input_2, "3" (__sc_5)
#define __sc_asm_input_4 __sc_asm_input_3, "4" (__sc_6)
#define __sc_asm_input_5 __sc_asm_input_4, "5" (__sc_7)

#define _syscall0(type,name)                                            \
```

```
type name(void)                                                         \
{                                                                       \
    __syscall_nr(0, type, name);                                        \
}

#define _syscall1(type,name,type1,arg1)                                 \
type name(type1 arg1)                                                   \
{                                                                       \
    __syscall_nr(1, type, name, arg1);                                  \
}

#define _syscall2(type,name,type1,arg1,type2,arg2)                      \
type name(type1 arg1, type2 arg2)                                       \
{                                                                       \
    __syscall_nr(2, type, name, arg1, arg2);                            \
}

#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)           \
type name(type1 arg1, type2 arg2, type3 arg3)                           \
{                                                                       \
    __syscall_nr(3, type, name, arg1, arg2, arg3);                      \
}

#define _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
type name(type1 arg1, type2 arg2, type3 arg3, type4 arg4)               \
{                                                                       \
    __syscall_nr(4, type, name, arg1, arg2, arg3, arg4);                \
}

#define _syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,a
type name(type1 arg1, type2 arg2, type3 arg3, type4 arg4, type5 arg5)   \
{                                                                       \
    __syscall_nr(5, type, name, arg1, arg2, arg3, arg4, arg5);          \
}
```

This code snippet from the kernel header file `asm/unistd.h` shows how we can implement system calls on PowerPC. It looks very complicated, but it can be broken down step by step.

Firstly, jump to the end of the example where the `_syscallN` macros are defined. You can see there are many macros, each one taking progressively one more argument. We'll concentrate on the most simple version, `_syscall0` to start with. It only takes two arguments, the return type of the system call (e.g. a C `int` or `char`, etc) and the name of the system call. For `getpid` this would be done as `_syscall0(int,getpid)`.

Easy so far! We now have to start pulling apart `__syscall_nr` macro. This is not dissimilar to where we were before, we take the number of arguments as the first parameter, the type, name and then the actual arguments.

The first step is declaring some names for registers. What this essentially does is says `__sc_0` refers to `r0` (i.e. register 0). The compiler will usually use registers how it wants, so it is important we give it constraints so that it doesn't decide to go using

register we need in some ad-hoc manner.

We then call `sc_loadargs` with the interesting `##` parameter. That is just a *paste* command, which gets replaced by the `nr` variable. Thus for our example it expands to `__sc_loadargs_0(name, args);` . `__sc_loadargs` we can see below sets `__sc_0` to be the system call number; notice the paste operator again with the `__NR_` prefix we talked about, and the variable name that refers to a specific register.

So, all this tricky looking code actually does is puts the system call number in register 0! Following the code through, we can see that the other macros will place the system call arguments into `r3` through `r7` (you can only have a maximum of 5 arguments to your system call).

Now we are ready to tackle the `__asm__` section. What we have here is called *inline assembly* because it is assembler code mixed right in with source code. The exact syntax is a little to complicated to go into right here, but we can point out the important parts.

Just ignore the `__volatile__` bit for now; it is telling the compiler that this code is unpredictable so it shouldn't try and be clever with it. Again we'll start at the end and work backwards. All the stuff after the colons is a way of communicating to the compiler about what the inline assembly is doing to the CPU registers. The compiler needs to know so that it doesn't try using any of these registers in ways that might cause a crash.

But the interesting part is the two assembly statements in the first argument. The one that does all the work is the `sc` call. That's all you need to do to make your system call!

So what happens when this call is made? Well, the processor is interrupted knows to transfer control to a specific piece of code setup at system boot time to handle interrupts. There are many interrupts; system calls are just one. This code will then look in register 0 to find the system call number; it then looks up a table and finds the right function to jump to to handle that system call. This function receives its arguments in registers 3 - 7.

So, what happens once the system call handler runs and completes? Control returns to the next instruction after the `sc` , in this case a *memory fence* instruction. What this essentially says is "make sure everything is committed to memory"; remember how we talked about pipelines in the superscalar architecture? This instruction ensures that everything we think has been written to memory actually has been, and isn't making its way through a pipeline somewhere.

Well, we're almost done! The only thing left is to return the value from the system call. We see that `__sc_ret` is set from r3 and `__sc_err` is set from r0. This is interesting; what are these two values all about?

One is the return value, and one is the error value. Why do we need two variables? System calls can fail, just as any other function. The problem is that a system call can return any possible value; we can not say "a negative value indicates failure" since a negative value might be perfectly acceptable for some particular system call.

So our system call function, before returning, ensures its result is in register r3 and any error code is in register r0. We check the error code to see if the top bit is set; this would indicate a negative number. If so, we set the global `errno` value to it (this is the standard variable for getting error information on call failure) and set the return to be `-1` . Of course, if a valid result is received we return it directly.

So our calling function should check the return value is not `-1` ; if it is it can check errno to find the exact reason why the call failed.

And that is an entire system call on a PowerPC!

x86 system calls

Below we have the same interface as implemented for the x86 processor.

Example 4.3. x86 system call example

```
/* user-visible error numbers are in the range -1 - -124: see <asm-i386/errno.h>

#define __syscall_return(type, res)                                \
```

```
do {                                                                    \
        if ((unsigned long)(res) >= (unsigned long)(-125)) {            \
                errno = -(res);                                         \
                res = -1;                                       \
        }                                                       \
        return (type) (res);                                            \
} while (0)

/* XXX - _foo needs to be __foo, while __NR_bar could be _NR_bar. */
#define _syscall0(type,name)                            \
type name(void)                                         \
{                                               \
long __res;                                     \
__asm__ volatile ("int $0x80"                           \
        : "=a" (__res)                                  \
        : "0" (__NR_##name));                           \
__syscall_return(type,__res);
}

#define _syscall1(type,name,type1,arg1)                         \
type name(type1 arg1)                                           \
{                                                       \
long __res;                                             \
__asm__ volatile ("int $0x80"                                   \
        : "=a" (__res)                                          \
        : "0" (__NR_##name),"b" ((long)(arg1)));        \
__syscall_return(type,__res);
}

#define _syscall2(type,name,type1,arg1,type2,arg2)                      \
type name(type1 arg1,type2 arg2)                                        \
{                                                                       \
long __res;                                                             \
__asm__ volatile ("int $0x80"                                                   \
        : "=a" (__res)                                                          \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2))); \
__syscall_return(type,__res);
}

#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)                   \
type name(type1 arg1,type2 arg2,type3 arg3)                             \
{                                                                       \
long __res;                                                             \
__asm__ volatile ("int $0x80"                                                   \
        : "=a" (__res)                                                          \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),  \
```

```
                  "d" ((long)(arg3)));                                    \
__syscall_return(type,__res);                                             \
}

#define _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)    \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4)                \
{                                                                         \
long __res;                                                               \
__asm__ volatile ("int $0x80"
        : "=a" (__res)
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),        \
          "d" ((long)(arg3)),"S" ((long)(arg4)));                         \
__syscall_return(type,__res);
}

#define _syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,    \
          type5,arg5)
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5)        \
{                                                                         \
long __res;                                                               \
__asm__ volatile ("int $0x80"
        : "=a" (__res)
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),        \
          "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)));
__syscall_return(type,__res);
}

#define _syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,
          type5,arg5,type6,arg6)
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5,type6 arg6)
{
long __res;
__asm__ volatile ("push %%ebp ; movl %%eax,%%ebp ; movl %1,%%eax ; int $0x80 ; p
        : "=a" (__res)
        : "i" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),
          "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)),
          "0" ((long)(arg6)));
__syscall_return(type,__res);
}
```

The x86 architecture is very different from the PowerPC that we looked at previously. The x86 is classed as a CISC processor as opposed to the RISC PowerPC, and has dramatically less registers.

Start by looking at the most simple `_syscall0` macro. It simply calls the `int` instruction with a value of `0x80`. This instruction makes the CPU raise interrupt 0x80, which will jump to code that handles system calls in the kernel.

We can start inspecting how to pass arguments with the longer macros. Notice how the PowerPC implementation cascaded macros downwards, adding one argument per time. This implementation has slightly more copied code, but is a little easier to follow.

x86 register names are based around letters, rather than the numerical based register names of PowerPC. We can see from the zero argument macro that only the `A` register gets loaded; from this we can tell that the system call number is expected in the `EAX` register. As we start loading registers in the other macros you can see the short names of the registers in the arguments to the `__asm__` call.

We see something a little more interesting in `__syscall6`, the macro taking 6 arguments. Notice the `push` and `pop` instructions? These work with the stack on x86, "pushing" a value onto the top of the stack in memory, and popping the value from the stack back into memory. Thus in the case of having six registers we need to store the value of the `ebp` register in memory, put our argument in in (the `mov` instruction), make our system call and then restore the original value into `ebp`. Here you can see the disadvantage of not having enough registers; stores to memory are expensive so the more you can avoid them, the better.

Another thing you might notice there is nothing like the *memory fence* instruction we saw previously with the PowerPC. This is because on x86 the effect of all instructions will be guaranteed to be visible when the complete. This is easier for the compiler (and programmer) to program for, but offers less flexibility.

The only thing left to contrast is the return value. On the PowerPC we had two registers with return values from the kernel, one with the value and one with an error code. However on x86 we only have one return value that is passed into `__syscall_return`. That macro casts the return value to `unsigned long` and compares it to an (architecture and kernel dependent) range of negative values that might represent error codes (note that the `errno` value is positive, so the negative result from the kernel is negated). However, this means that system calls can not return small negative values, since they are indistinguishable from error codes. Some system calls that have this requirement, such as `getpriority()`, add an offset to their return value to force it to always be positive; it is up to the userspace to realise this and subtract this constant value to get back to the "real" value.

This page titled 3.3: System Calls is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

3.4: Privileges

Privileges

Hardware

We mentioned how one of the major tasks of the operating system is to implement security; that is to not allow one application or user to interfere with any other that is running in the system. This means applications should not be able to overwrite each others memory or files, and only access system resources as dictated by system policy.

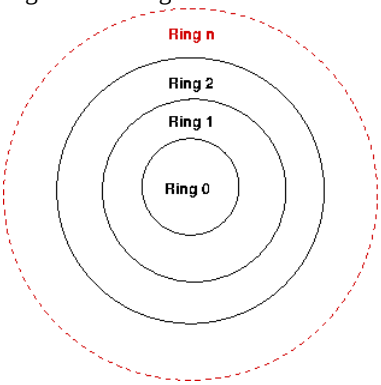
However, when an application is running it has exclusive use of the processor. We see how this works when we examine processes in the next chapter. Ensuring the application only accesses memory it owns is implemented by the virtual memory system, which we examine in the chapter after next. The essential point is that the hardware is responsible for enforcing these rules.

The system call interface we have examined is the gateway to the application getting to system resources. By forcing the application to request resources through a system call into the kernel, the kernel can enforce rules about what sort of access can be provided. For example, when an application makes an `open()` system call to open a file on disk, it will check the permissions of the user against the file permissions and allow or deny access.

Privilege Levels

Hardware protection can usually be seen as a set of concentric rings around a core set of operations.

Figure 4.3. Rings



Privilege levels on x86

In the inner most ring are the most protected instructions; those that only the kernel should be allowed to call. For example, the `HLT` instruction to halt the processor should not be allowed to be run by a user application, since it would stop the entire computer from working. However, the kernel needs to be able to call this instruction when the computer is legitimately shut down. [11]

Each inner ring can access any instructions protected by a further out ring, but not any protected by a further in ring. Not all architectures have multiple levels of rings as above, but most will either provide for at least a "kernel" and "user" level.

386 protection model

The 386 protection model has four rings, though most operating systems (such as Linux and Windows) only use two of the rings to maintain compatibility with other architectures that do now allow as many discrete protection levels.

386 maintains privileges by making each piece of application code running in the system have a small descriptor, called a *code descriptor*, which describes, amongst other things, its privilege level. When running application code makes a jump into some other code outside the region described by its code descriptor, the privilege level of the target is checked. If it is higher than the currently running code, the jump is disallowed by the hardware (and the application will crash).

Raising Privilege

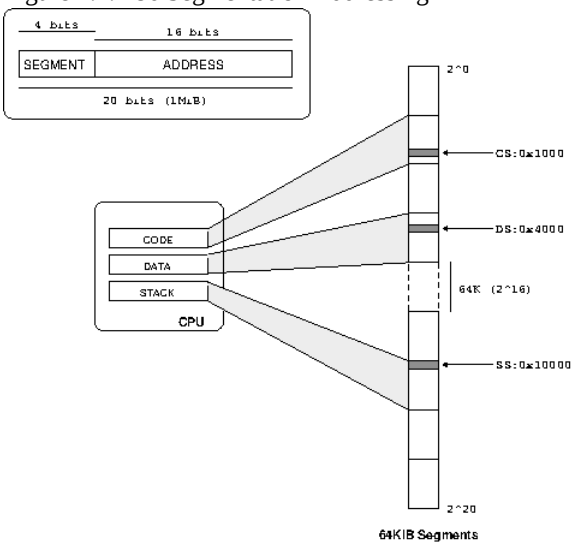
Applications may only raise their privilege level by specific calls that allow it, such as the instruction to implement a system call. These are usually referred to as a *call gate* because they function just as a physical gate; a small entry through an otherwise impenetrable wall. When that instruction is called we have seen how the hardware completely stops the running application and hands control over to the kernel. The kernel must act as a gatekeeper; ensuring that nothing nasty is coming through the gate. This means it must check system call arguments carefully to make sure it will not be fooled into doing anything it shouldn't (if it can be, that is a security bug). As the kernel runs in the innermost ring, it has permissions to do any operation it wants; when it is finished it will return control back to the application which will again be running with its lower privilege level.

Fast System Calls

One problem with traps as described above is that they are very expensive for the processor to implement. There is a lot of state to be saved before context can switch. Modern processors have realised this overhead and strive to reduce it.

To understand the call-gate mechanism described above requires investigation of the ingenious but complicated segmentation scheme used by the processor. The original reason for segmentation was to be able to use more than the 16 bits available in a register for an address, as illustrated in Figure 4.4, “x86 Segmentation Addressing”.

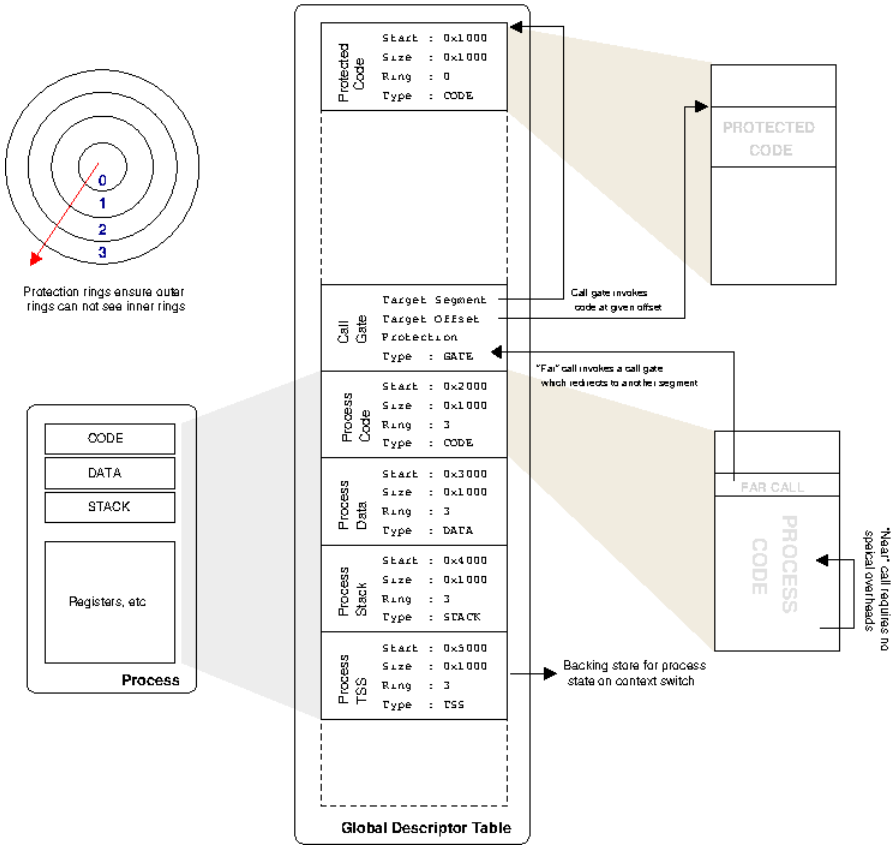
Figure 4.4. x86 Segmentation Addressing



Segmentation expanding the address space of a processor by dividing it into chunks. The processor keeps special segment registers, and addresses are specified by a segment register and offset combination. The value of the segment register is added to the offset portion to find a final address.

When x86 moved to 32 bit registers, the segmentation scheme remained but in a different format. Rather than fixed segment sizes, segments are allowed to be any size. This means the processor needs to keep track of all these different segments and their sizes, which it does using *descriptors*. The segment descriptors available to everyone are kept in the *global descriptor table* or GDT for short. Each process has a number of registers which point to entries in the GDT; these are the segments the process can access (there are also *local* descriptor tables, and it all interacts with task state segments, but that's not important now). The overall situation is illustrated in Figure 4.5, “x86 segments”.

Figure 4.5. x86 segments



x86 segments in action. Notice how a "far-call" passes via a call-gate which redirects to a segment of code running at a lower ring level. The only way to modify the code-segment selector, implicitly used for all code addresses, is via the call mechanism. Thus the call-gate mechanism ensures that to choose a new segment descriptor, and hence possibly change protection levels, you must transition via a known entry point.

Since the operating system assigns the segment registers as part of the process state, the processor hardware knows what segments of memory the currently running process can access and can enforce *protection* to ensure the process doesn't touch anything it is not supposed to. If it does go out of bounds, you receive a *segmentation fault*, which most programmers are familiar with.

The picture becomes more interesting when running code needs to make calls into code that resides in *another* segment. As discussed in the section called “386 protection model”, x86 does this with *rings*, where ring 0 is the highest permission, ring 3 is the lowest, and inner rings can access outer rings but not vice-versa.

As discussed in the section called “Raising Privilege”, when ring 3 code wants to jump into ring 0 code, it is essentially modifying its code segment selector to point to a different segment. To do this, it must use a special *far-call* instruction which hardware ensures passes through the call gate. There is no other way for the running process to choose a new code-segment descriptor, and hence the processor will start executing code at the known offset within the ring 0 segment, which is responsible for maintaining integrity (e.g. not reading arbitrary and possibly malicious code and executing it. Of course nefarious attackers will always look for ways to make your code do what you did not intend it to!).

This allows a whole hierarchy of segments and permissions between them. You might have noticed a cross segment call sounds exactly like a system call. If you've ever looked at Linux x86 assembly the standard way to make a system call is `int 0x80` , which raises interrupt `0x80` . An interrupt stops the processor and goes to an interrupt gate, which then works the same as a call gate -- it changes privilege level and bounces you off to some other area of code .

The problem with this scheme is that it is *slow*. It takes a lot of effort to do all this checking, and many registers need to be saved to get into the new code. And on the way back out, it all needs to be restored again.

On a modern x86 system segmentation and the four-level ring system is not used thanks to virtual memory, discussed fully in Chapter 6, *Virtual Memory*. The only thing that really happens with segmentation switching is system calls, which essentially switch from mode 3 (userspace) to mode 0 and jump to the system call handler code inside the kernel. Thus the processor provides extra *fast system call* instructions called `sysenter` (and `sysexit` to get back) which speed up the whole process over a `int 0x80` call by removing the general nature of a far-call — that is the possibility of transitioning into any segment at any ring level — and restricting the call to only transition to ring 0 code at a specific segment and offset, as stored in registers.

Because the general nature has been replaced with so much prior-known information, the whole process can be speed up, and hence we have a the aforementioned *fast system call*. The other thing to note is that state is not preserved when the kernel gets control. The kernel has to be careful to not to destroy state, but it also means it is free to only save as little state as is required to do the job, so can be much more efficient about it. This is a very RISC philosophy, and illustrates how the line blurs between RISC and CISC processors.

For more information on how this is implemented in the Linux kernel, see the section called “Kernel Library”.

Other ways of communicating with the kernel

ioctl

about ioctls

File Systems

about proc, sysfs, debugfs, etc

---

[11] What happens when a "naughty" application calls that instruction anyway? The hardware will usually raise an exception, which will involve jumping to a specified handler in the operating system similar to the system call handler. The operating system will then probably terminate the program, usually giving the user some error about how the application has crashed.

This page titled 3.4: Privileges is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

3.5: Function of the Operating System

What is the Purpose of an OS?

An **operating system** acts as a communication bridge (interface) between the user and computer hardware. The purpose of an operating system is to provide a platform on which a user can execute programs in a convenient and efficient manner.

An operating system is a piece of software that manages the allocation of computer hardware. The coordination of the hardware must be appropriate to ensure the correct working of the computer system and to prevent user programs from interfering with the proper working of the system.
Example: Just like a boss gives order to his employee, in the similar way we request or pass our orders to the operating system. The main goal of the operating system is to thus make the computer environment more convenient to use and the secondary goal is to use the resources in the most efficient manner.

What is operating system ?
An operating system is a program on which application programs are executed and acts as an communication bridge (interface) between the user and the computer hardware.

The main task an operating system carries out is the allocation of resources and services, such as allocation of: memory, devices, processors and information. The operating system also includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system.

Important functions of an operating system:

1. **Security**
   The operating system uses password protection to protect user data and similar other techniques. it also prevents unauthorized access to programs and user data.
2. **Control over system performance**
   Monitors overall system health to help improve performance. records the response time between service requests and system response to have a complete view of the system health. This can help improve performance by providing important information needed to troubleshoot problems.
3. **Job accounting**
   Operating system keeps track of time and resources used by various tasks and users, this information can be used to track resource usage for a particular user or group of user.
4. **Error detecting aids**
   Operating system constantly monitors the system to detect errors and avoid the malfunctioning of computer system.
5. **Coordination between other software and users**
   Operating systems also coordinate and assign interpreters, compilers, assemblers and other software to the various users of the computer systems.
6. **Memory Management**
   The operating system manages the primary memory or main memory. Main memory is made up of a large array of bytes or words where each byte or word is assigned a certain address. Main memory is a fast storage and it can be accessed directly by the CPU. For a program to be executed, it should be first loaded in the main memory. An operating system performs the following activities for memory management:

   It keeps tracks of primary memory, i.e., which bytes of memory are used by which user program. The memory addresses that have already been allocated and the memory addresses of the memory that has not yet been used. In multi programming, the OS decides the order in which process are granted access to memory, and for how long. It Allocates the memory to a process when the process requests it and deallocates the memory when the process has terminated or is performing an I/O operation.

7. **Processor Management**
   In a multi programming environment, the OS decides the order in which processes have access to the processor, and how much processing time each process has. This function of OS is called process scheduling. An operating system performs the following activities for processor management.

Keeps tracks of the status of processes. The program which perform this task is known as traffic controller. Allocates the CPU that is processor to a process. De-allocates processor when a process is no more required.

8. **Device Management**
   An OS manages device communication via their respective drivers. It performs the following activities for device management. Keeps tracks of all devices connected to system. designates a program responsible for every device known as the Input/Output controller. Decides which process gets access to a certain device and for how long. Allocates devices in an effective and efficient way. Deallocates devices when they are no longer required.
9. **File Management**
   A file system is organized into directories for efficient or easy navigation and usage. These directories may contain other directories and other files. An operating system carries out the following file management activities. It keeps track of where information is stored, user access settings and status of every file and more… These facilities are collectively known as the file system.

Moreover, operating system also provides certain services to the computer system in one form or the other.

The operating system provides certain services to the users which can be listed in the following manner:

1. **Program Execution**
   The operating system is responsible for execution of all types of programs whether it be user programs or system programs. The operating system utilizes various resources available for the efficient running of all types of functionalities.
2. **Handling Input/Output Operations**
   The operating system is responsible for handling all sort of inputs, i.e, from keyboard, mouse, desktop, etc. The operating system does all interfacing in the most appropriate manner regarding all kind of inputs and outputs.
   For example, there is difference in nature of all types of peripheral devices such as mouse or keyboard, then operating system is responsible for handling data between them.
3. **Manipulation of File System**
   The operating system is responsible for making of decisions regarding the storage of all types of data or files, i.e, floppy disk/hard disk/pen drive, etc. The operating system decides as how the data should be manipulated and stored.
4. **Error Detection and Handling**
   The operating system is responsible for detection of any types of error or bugs that can occur while any task. The well secured OS sometimes also acts as countermeasure for preventing any sort of breach to the computer system from any external source and probably handling them.
5. **Resource Allocation**
   The operating system ensures the proper use of all the resources available by deciding which resource to be used by whom for how much time. All the decisions are taken by the operating system.
6. **Accounting**
   The operating system tracks an account of all the functionalities taking place in the computer system at a time. All the details such as the types of errors occurred are recorded by the operating system.
7. **Information and Resource Protection**
   The operating system is responsible for using all the information and resources available on the machine in the most protected way. The operating system must foil an attempt from any external resource to hamper any sort of data or information.

All these services are ensured by the operating system for the convenience of the users to make the programming task easier. All different kinds of operating system more or less provide the same services.

Adapted from:
"Functions of operating system" by Amaninder.Singh, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 3.5: Function of the Operating System is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

3.6: Types of Operating Systems

What are the Types of Operating Systems

An **Operating System** performs all the basic tasks like managing file,process, and memory. Thus operating system acts as manager of all the resources, i.e. **resource manager**. Thus operating system becomes an interface between user and machine.

Types of Operating Systems: Some of the widely used operating systems are as follows-

1. Batch Operating System

This type of operating system does not interact with the computer directly. There is an operator which takes similar jobs having same requirement and group them into batches. It is the responsibility of operator to sort the jobs with similar needs.

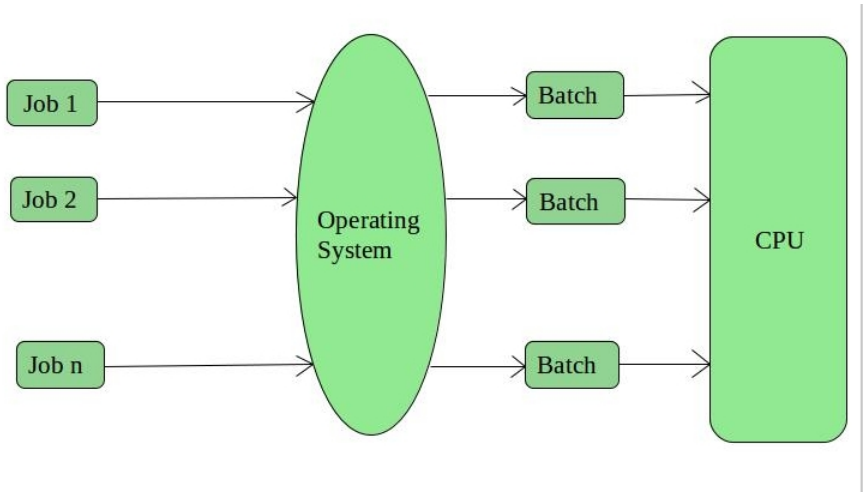


Figure 3.6.1: Depiction of a batch operating system. ("A batch operating system" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0)

Advantages of Batch Operating System:

- It is very difficult to guess or know the time required by any job to complete. Processors of the batch systems know how long the job would be when it is in queue
- Multiple users can share the batch systems
- The processor idle time for batch system is very low
- It is easy to manage large tasks repeatedly in batch systems

Disadvantages of Batch Operating System:

- The computer operators should have a good understanding of batch systems
- Batch systems are hard to debug
- It is sometime costly
- The other jobs will have to wait for an unknown time if any job fails

Examples of Batch based Operating System: Payroll System, Bank Statements etc.

2. Time-Sharing Operating Systems

Each task is given some time to execute, so that all the tasks work smoothly. Each user gets a time time slot on the CPU. These systems are also known as Multitasking Systems. The task can be from single user or from different users. The time that each task gets to execute is called quantum. After this time interval is over OS switches to next task.

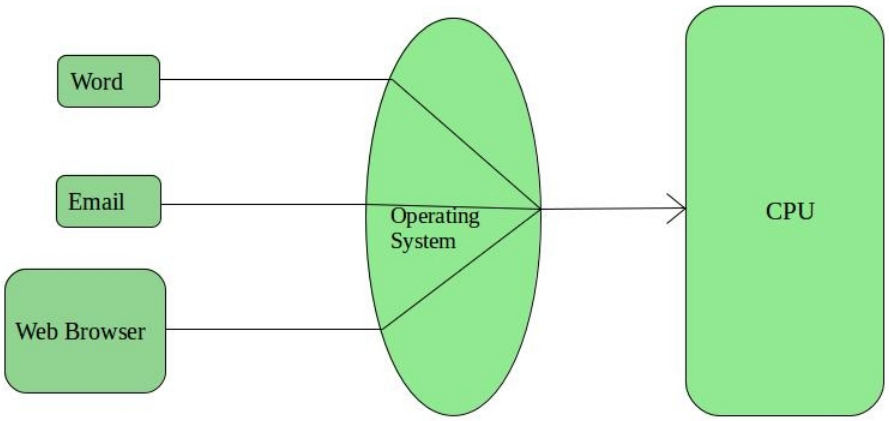


Figure 3.6.1: Time sharing Operating System. ("A time share operating system" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0)

Advantages of Time-Sharing OS:

- Each task gets an equal time on the processor
- Less chances of duplication of software
- CPU idle time can be reduced

Disadvantages of Time-Sharing OS:

- The operating system must take care of security and integrity of user programs and data
- Data communication problems can arise with if the data storage or users are remotely located

Examples of Time-Sharing OSs are: Linux, Unix etc.

Adapted from:
"Types of Operating Systems" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 3.6: Types of Operating Systems is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

3.6.1: Types of Operating Systems (continued)

3. Distributed Operating System

Various autonomous interconnected computers communicate each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred as **loosely coupled systems** or distributed systems. These system's processors differ in size and function. The major benefit of working with these types of operating system is that it is always possible that one user can access the files or software which are not actually present on his system but on some other system connected within this network i.e., remote access is enabled within the devices connected in that network.

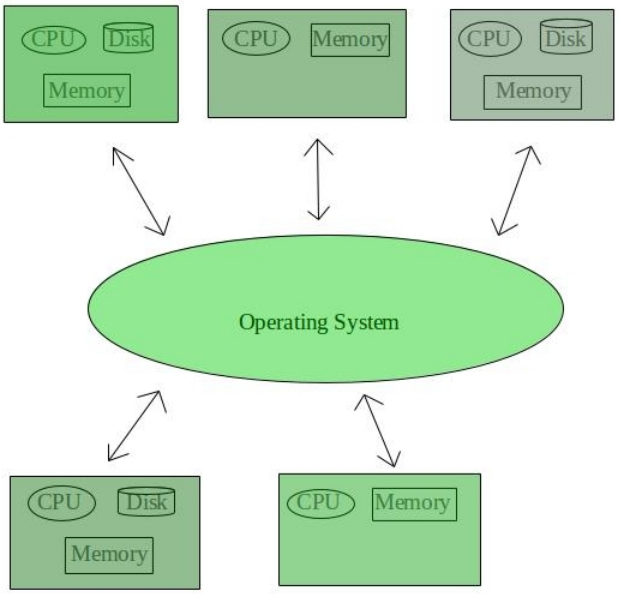


Figure 3.6.1.1: Distributed Operating System. ("Distributed Operating System" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0)

Advantages of Distributed Operating System:

- Failure of one node will not affect the other network communication, since all systems are independent from each other
- Since resources are being shared, computation can be very fast
- Load on host computer is reduced
- These systems are easily scalable as many systems can be easily added to the network

Disadvantages of Distributed Operating System:

- Failure of the main network will stop the entire communication
- With the increase of telecommunications capability and the Internet, some of the disadvantages have disappeared

Examples of Distributed Operating System are- LOCUS .

4. Network Operating System

Historically operating systems with networking capabilities were described as network operating system, because they allowed personal computers (PCs) to participate in computer networks and shared file and printer access within a local area network (LAN). This description of operating systems is now largely historical, as common operating systems include a network stack to support a client–server model.

These limited client/server networks were gradually replaced by Peer-to-peer networks, which used networking capabilities to share resources and files located on a variety of computers of all sizes. A peer-to-peer network sets all connected computers equal; they all share the same abilities to use resources available on the network. The most popular peer-to-peer networks as of 2020 are Ethernet, Wi-Fi and the Internet protocol suite. Software that allowed users to interact with these networks, despite a lack of

networking support in the underlying manufacturer's operating system, was sometimes called a network operating system. Examples of such add-on software include Phil Karn's KA9Q NOS (adding Internet support to CP/M and MS-DOS), PC/TCP Packet Drivers (adding Ethernet and Internet support to MS-DOS), and LANtastic (for MS-DOS, Microsoft Windows and OS/2), and Windows for Workgroups (adding NetBIOS to Windows). Examples of early operating systems with peer-to-peer networking capabilities built-in include MacOS (using AppleTalk and LocalTalk), and the Berkeley Software Distribution.

Today, distributed computing and groupware applications have become the norm. Computer operating systems include a networking stack as a matter of course. During the 1980s the need to integrate dissimilar computers with network capabilities grew and the number of networked devices grew rapidly. Partly because it allowed for multi-vendor interoperability, and could route packets globally rather than being restricted to a single building, the Internet protocol suite became almost universally adopted in network architectures. Thereafter, computer operating systems and the firmware of network devices tended to support Internet protocols.

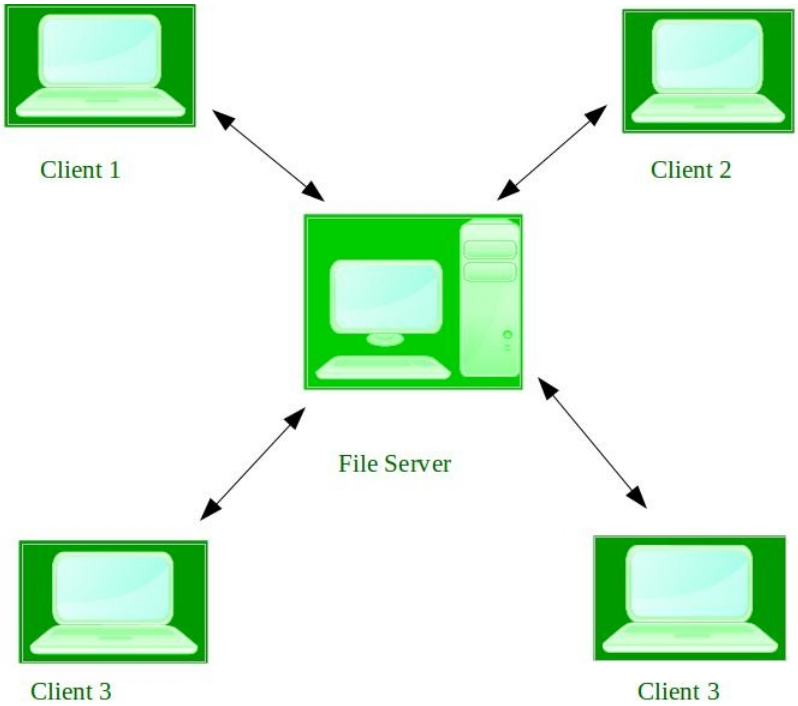


Figure 3.6.1.1: Network Operating System. ("Network Operating System" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0)

Advantages of Network Operating System:

- Highly stable centralized servers
- Security concerns are handled through servers
- New technologies and hardware up-gradation are easily integrated to the system
- Server access are possible remotely from different locations and types of systems

Disadvantages of Network Operating System:

- Servers are costly
- User has to depend on central location for most operations
- Maintenance and updates are required regularly

Examples of Network Operating System are: Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD etc.

Adapted from:
"Types of Operating Systems" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0

"Network operating system" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 3.6.1: Types of Operating Systems (continued) is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

3.6.2: Types of Operating Systems (continued)

5. Real-Time Operating System

These types of OSs are used in real-time systems. The time interval required to process and respond to inputs is very small. This time interval is called **response time**.

Real-time systems are used when there are time requirements are very strict like missile systems, air traffic control systems, robots etc.

Two types of Real-Time Operating System which are as follows:

- **Hard Real-Time Systems:**
  These OSs are meant for the applications where time constraints are very strict and even the shortest possible delay is not acceptable. These systems are built for saving life like automatic parachutes or air bags which are required to be readily available in case of any accident. Virtual memory is almost never found in these systems.
- **Soft Real-Time Systems:**
  These OSs are for applications where for time-constraint is less strict.

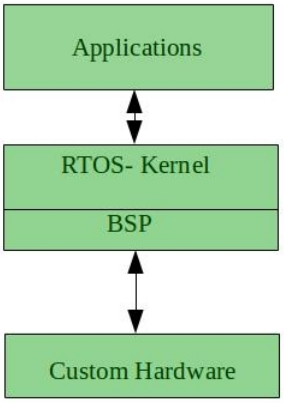


Figure 3.6.2.1: Real Time Operating System. ("Real time operating system" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0)

Advantages of RTOS:

- **Maximum Consumption:** Maximum utilization of devices and system,thus more output from all the resources
- **Task Shifting:** Time assigned for shifting tasks in these systems are very less. For example in older systems it takes about 10 micro seconds in shifting one task to another and in latest systems it takes 3 micro seconds.
- **Focus on Application:** Focus on running applications and less importance to applications which are in queue.
- **Real time operating system in embedded system:** Since size of programs are small, RTOS can also be used in embedded systems like in transport and others.
- **Error Free:** These types of systems MUST be able to deal with any exceptions, so they are not really error free, but handle error conditions without halting the system.
- **Memory Allocation:** Memory allocation is best managed in these type of systems.

Disadvantages of RTOS:

- **Limited Tasks:** Very few tasks run at the same time and their concentration is very less on few applications to avoid errors.
- **Use heavy system resources:** Sometimes the system resources are not so good and they are expensive as well.
- **Complex Algorithms:** The algorithms are very complex and difficult for the designer to write on.
- **Device driver and interrupt signals:** It needs specific device drivers and interrupt signals to response earliest to interrupts.

- **Thread Priority:** It is not good to set thread priority as these systems are very less prone to switching tasks.

Examples of Real-Time Operating Systems are: Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

Adapted from:
"Types of Operating Systems" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 3.6.2: Types of Operating Systems (continued) is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

3.7: Difference between multitasking, multithreading and multiprocessing

Multi-programming

In a modern computing system, there are usually several concurrent application processes which want to execute. It is the responsibility of the operating system to manage all the processes effectively and efficiently. One of the most important aspects of an operating system is to provide the capability to multi-program.

In a computer system, there are multiple processes waiting to be executed, i.e. they are waiting while the CPU is allocated to other processes. The main memory is too small to accommodate all of these processes or jobs. Thus, these processes are initially kept in an area called job pool. This job pool consists of all those processes awaiting allocation of main memory and CPU.

The scheduler selects a job out of the job pool, brings it into main memory and begins executing it. The processor executes one job until one of several factors interrupt its processing: 1) the process uses up its allotted time; 2) some other interrupt (we will talk more about interrupts) causes the processor to stop executing this process; 3) the process goes into a wait state waiting on an I/O request.

Non-multi-programmed system concepts:

- In a non multi-programmed system, as soon as one job hits any type of interrupt or wait state, the CPU becomes idle. The CPU keeps waiting and waiting until this job (which was executing earlier) comes back and resumes its execution with the CPU. So CPU remains idle for a period of time.
- There are drawbacks when the CPU remains idle for a very long period of time. Other jobs which are waiting for the processor will not get a chance to execute because the CPU is still allocated to the job that is in a wait state.
  This poses a very serious problem - even though other jobs are ready to execute, the CPU is not available to them because it is still allocated to a job which is not even utilizing it.
- It is possible that one job is using the CPU for an extended period of time, while other jobs sit in the queue waiting for access to the CPU. In order to work around such scenarios like this the concept of multi-programming developed to increase the CPU utilization and thereby the overall efficiency of the system.

The main idea of multi-programming is to maximize the CPU time.

Multi-programmed system concepts:

- In a multi-programmed system, as soon as one job goes gets interrupted or goes into a wait state, the cpu selects the next job from the scheduler and starts its execution. Once the previous job resolves the reason for its interruption - perhaps the I/O completes - goes back into the job pool. If the second job goes into a wait state, the CPU chooses a third job and starts executing it.
- This makes for much more efficient use of the CPU. Therefore, the ultimate goal of multi-programming is to keep the CPU busy as long as there are processes ready to execute. This way, multiple programs can be executed on a single processor by executing a part of a program at one time, a part of another program after this, then a part of another program and so on, hence executing multiple programs
- In the image below, program A runs for some time and then goes to waiting state. In the mean time program B begins its execution. So the CPU does not waste its resources and gives program B an opportunity to run. There is still time slots where the processor is waiting - other programs could be run if necessary.

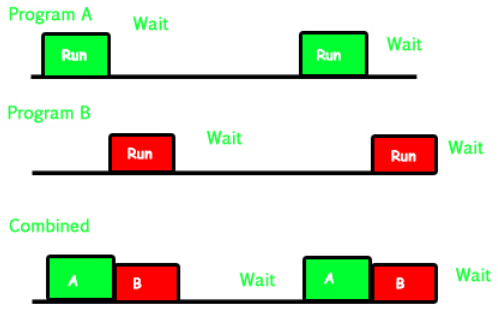


Figure 3.7.1: Multiprogramming. ("Multiprogramming" by Darshan L., Geeks for Geeks is licensed under CC BY-SA 4.0)

Adapted from:
"Difference between Multiprogramming, multitasking, multithreading and multiprocessing" by Darshan L., Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 3.7: Difference between multitasking, multithreading and multiprocessing is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

3.7.1: Difference between multitasking, multithreading and multiprocessing (continued)

2. Multiprocessing

In a uni-processor system, only one process executes at a time. Multiprocessing makes use of two or more CPUs (processors) within a single computer system. The term also refers to the ability of a system to support more than one processor within a single computer system. Since there are multiple processors available, multiple processes can be executed at a time. These multiprocessors share the computer bus, sometimes the clock, memory and peripheral devices also.

Multiprocessing system's working –

- With the help of multiprocessing, many processes can be executed simultaneously. Say processes P1, P2, P3 and P4 are waiting for execution. Now in a single processor system, firstly one process will execute, then the other, then the other and so on.
- But with multiprocessing, each process can be assigned to a different processor for its execution. If its a dual-core processor (2 processors), two processes can be executed simultaneously and thus will be two times faster, similarly a quad core processor will be four times as fast as a single processor.

Why use multiprocessing

- The main advantage of multiprocessor system is to get more work done in a shorter period of time. These types of systems are used when very high speed is required to process a large volume of data. multiprocessing systems can save money in comparison to single processor systems because the processors can share peripherals and power supplies.
- It also provides increased reliability in that if one processor fails, the work does not halt, it only slows down. e.g. if we have 10 processors and 1 fails, then the work does not halt, rather the remaining 9 processors can share the work of the 10th processor. Thus the whole system runs only 10 percent slower, rather than failing altogether

Multiprocessing refers to the hardware (i.e., the CPU units) rather than the software (i.e., running processes). If the underlying hardware provides more than one processor then that is multiprocessing. It is the ability of the system to leverage multiple processors' computing power.

Difference between multiprogramming and multiprocessing

- A system can be both multi programmed by having multiple programs running at the same time and multiprocessing by having more than one physical processor. The difference between multiprocessing and multi programming is that Multiprocessing is basically executing multiple processes at the same time on multiple processors, whereas multi programming is keeping several programs in main memory and executing them concurrently using a single CPU only.
- Multiprocessing occurs by means of parallel processing whereas Multi programming occurs by switching from one process to other (phenomenon called as context switching).

3. Multitasking

As the name itself suggests, multitasking refers to execution of multiple tasks (say processes, programs, threads etc.) at a time. In the modern operating systems, we are able to play MP3 music, edit documents in Microsoft Word, surf the Google Chrome all simultaneously, this is accomplished by means of multitasking.

Multitasking is a logical extension of multi programming. The major way in which multitasking differs from multi programming is that multi programming works solely on the concept of context switching whereas multitasking is based on time sharing alongside the concept of context switching.

Multi tasking system's concepts

- In a time sharing system, each process is assigned some specific quantum of time for which a process is meant to execute. Say there are 4 processes P1, P2, P3, P4 ready to execute. So each of them are assigned some time quantum for which they will execute e.g time quantum of 5 nanoseconds (5 ns). As one process begins execution (say P2), it executes for that quantum of time (5 ns). After 5 ns the CPU starts the execution of the other process (say P3) for the specified quantum of time.

- Thus the CPU makes the processes to share time slices between them and execute accordingly. As soon as time quantum of one process expires, another process begins its execution.
- Here also basically a context switch is occurring but it is occurring so fast that the user is able to interact with each program separately while it is running. This way, the user is given the illusion that multiple processes/ tasks are executing simultaneously. But actually only one process/ task is executing at a particular instant of time. In multitasking, time sharing is best manifested because each running process takes only a fair quantum of the CPU time.

In a more general sense, multitasking refers to having multiple programs, processes, tasks, threads running at the same time. This term is used in modern operating systems when multiple tasks share a common processing resource (e.g., CPU and Memory).

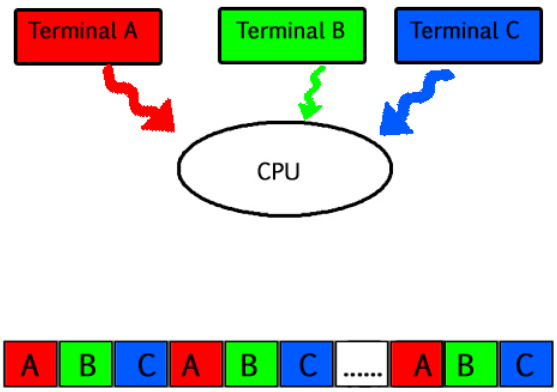


Figure 3.7.1.1: Depiction of Multitasking System. ("Multitasking" by Darshan L., Geeks for Geeks is licensed under CC BY-SA 4.0)

As depicted in the above image, At any time the CPU is executing only one task while other tasks are waiting for their turn. The illusion of parallelism is achieved when the CPU is reassigned to another task. i.e all the three tasks A, B and C are appearing to occur simultaneously because of time sharing.

So for multitasking to take place, firstly there should be multiprogramming i.e. presence of multiple programs ready for execution. And secondly the concept of time sharing.

Adapted from:
"Difference between Multiprogramming, multitasking, multithreading and multiprocessing" by Darshan L., Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 3.7.1: Difference between multitasking, multithreading and multiprocessing (continued) is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

3.7.2: Difference between Multiprogramming, multitasking, multithreading and multiprocessing (continued)

Multithreading

A thread is a basic unit of CPU utilization. Multithreading is an execution model that allows a single process to have multiple code segments (i.e., threads) running concurrently within the “context” of that process.
e.g. VLC media player, where one thread is used for opening the VLC media player, one thread for playing a particular song and another thread for adding new songs to the playlist.

Multithreading is the ability of a process to manage its use by more than one user at a time and to manage multiple requests by the same user without having to have multiple copies of the program.

Multithreading system examples

Example 1

- Say there is a web server which processes client requests. Now if it executes as a single threaded process, then it will not be able to process multiple requests at a time. First one client will make its request and finish its execution and only then, the server will be able to process another client request. This is quite inefficient, time consuming and tiring task. To avoid this, we can take advantage of multithreading.
- Now, whenever a new client request comes in, the web server simply creates a new thread for processing this request and resumes its execution to process more client requests. So the web server has the task of listening to new client requests and creating threads for each individual request. Each newly created thread processes one client request, thus reducing the burden on web server.

Example 2

- We can think of threads as child processes that share the parent process resources but execute independently. Take the case of a GUI. Say we are performing a calculation on the GUI (which is taking very long time to finish). Now we can not interact with the rest of the GUI until this command finishes its execution. To be able to interact with the rest of the GUI, this calculation should be assigned to a separate thread. So at this point of time, 2 threads will be executing i.e. one for calculation, and one for the rest of the GUI. Hence here in a single process, we used multiple threads for multiple functionality.

The image helps to describe the VLC player example:

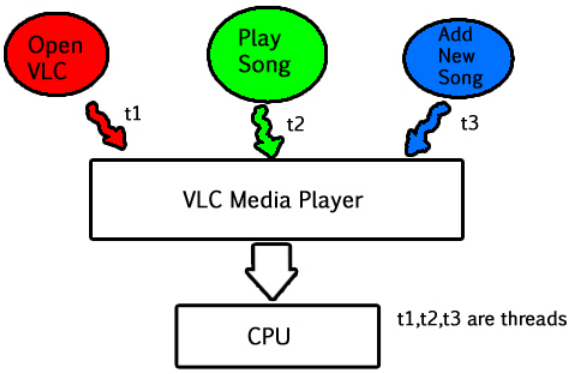


Figure 3.7.2.1: Example of Multithreading. ("Multithreading" by Darshan L., Geeks for Geeks is licensed under CC BY-SA 4.0)

Advantages of multithreading

- Benefits of multithreading include increased responsiveness. Since there are multiple threads in a program, so if one thread is taking too long to execute or if it gets blocked, the rest of the threads keep executing without any problem. Thus the whole program remains responsive to the user by means of remaining threads.

- Another advantage of multithreading is that it is less costly. Creating brand new processes and allocating resources is a time consuming task, but since threads share resources of the parent process, creating threads and switching between them is comparatively easy. Hence multithreading is the need of modern Operating Systems.

Adapted from:
"Difference between Multiprogramming, multitasking, multithreading and multiprocessing" by Darshan L., Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 3.7.2: Difference between Multiprogramming, multitasking, multithreading and multiprocessing (continued) is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

4: Computer Architecture - the CPU

4.1: Instruction Cycles

4.1.1: Instruction Cycles - Fetch

4.1.2: Instruction Cycles - Instruction Primer

4.2: Interrupts

4: Computer Architecture - the CPU is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

4.1: Instruction Cycles

Instruction Cycles

The **instruction cycle** (also known as the **fetch–decode–execute cycle**, or simply the **fetch-execute cycle**) is the cycle that the central processing unit (CPU) follows from boot-up until the computer has shut down in order to process instructions. It is composed of three main stages: the fetch stage, the decode stage, and the execute stage.

Role of components

The program counter (PC) is a special register that holds the memory address of the next instruction to be executed. During the fetch stage, the address stored in the PC is copied into the memory address register (MAR) and then the PC is incremented in order to "point" to the memory address of the next instruction to be executed. The CPU then takes the instruction at the memory address described by the MAR and copies it into the memory data register (MDR). The MDR also acts as a two-way register that holds data fetched from memory or data waiting to be stored in memory (it is also known as the memory buffer register (MBR) because of this). Eventually, the instruction in the MDR is copied into the current instruction register (CIR) which acts as a temporary holding ground for the instruction that has just been fetched from memory.

During the decode stage, the control unit (CU) will decode the instruction in the CIR. The CU then sends signals to other components within the CPU, such as the arithmetic logic unit (ALU) and the floating point unit (FPU). The ALU performs arithmetic operations such as addition and subtraction and also multiplication via repeated addition and division via repeated subtraction. It also performs logic operations such as AND, OR, NOT, and binary shifts as well. The FPU is reserved for performing floating-point operations.

Summary of stages

Each computer's CPU can have different cycles based on different instruction sets, but will be similar to the following cycle:

1. **Fetch Stage**: The next instruction is fetched from the memory address that is currently stored in the program counter and stored into the instruction register. At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.
2. **Decode Stage**: During this stage, the encoded instruction presented in the instruction register is interpreted by the decoder.
   - **Read the effective address**: In the case of a memory instruction (direct or indirect), the execution phase will be during the next clock pulse. If the instruction has an indirect address, the effective address is read from main memory, and any required data is fetched from main memory to be processed and then placed into data registers (clock pulse: T_3). If the instruction is direct, nothing is done during this clock pulse. If this is an I/O instruction or a register instruction, the operation is performed during the clock pulse.
3. **Execute Stage**: The control unit of the CPU passes the decoded information as a sequence of control signals to the relevant functional units of the CPU to perform the actions required by the instruction, such as reading values from registers, passing them to the ALU to perform mathematical or logic functions on them, and writing the result back to a register. If the ALU is involved, it sends a condition signal back to the CU. The result generated by the operation is stored in the main memory or sent to an output device. Based on the feedback from the ALU, the PC may be updated to a different address from which the next instruction will be fetched.
4. **Repeat Cycle**

Registers Involved In Each Instruction Cycle:

- **Memory address registers(MAR)** : It is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
- **Memory Buffer Register(MBR)** : It is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from the memory.
- **Program Counter(PC)** : Holds the address of the next instruction to be fetched.
- **Instruction Register(IR)** : Holds the last instruction fetched.

Adapted from:
"Instruction cycle" by Multiple Contributors, Wikipedia is licensed under CC BY-NC 3.0

This page titled 4.1: Instruction Cycles is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

4.1.1: Instruction Cycles - Fetch

The Fetch Cycle

At the beginning of the fetch cycle, the address of the next instruction to be executed is in the *Program Counter*(PC).

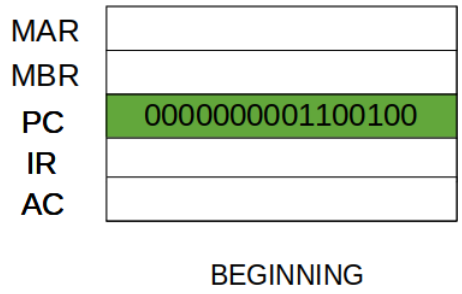


Figure 4.1.1.1: Beginning of the Fetch Cycle. ("Beginning of the Fetch Cycle" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

Step 1: The address in the program counter is moved to the memory address register(MAR), as this is the only register which is connected to address lines of the system bus.

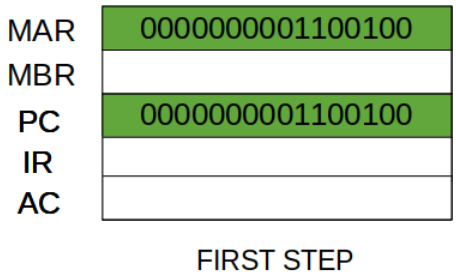


Figure 4.1.1.1: Step #1 of Fetch Cycle. ("Beginning of the Fetch Cycle" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

Step 2: The address in MAR is placed on the address bus, now the control unit issues a READ command on the control bus, and the result appears on the data bus and is then copied into the memory buffer register(MBR). Program counter is incremented by one, to get ready for the next instruction.(These two action can be performed simultaneously to save time)

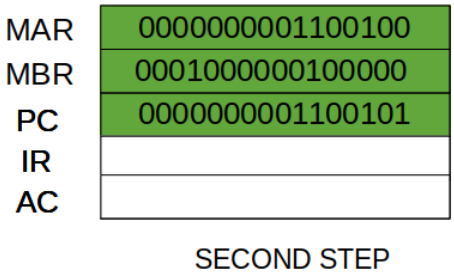


Figure 4.1.1.1: Step #2 of Fetch Cycle. ("Step #2 of the Fetch Cycle" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

Step 3: The content of the MBR is moved to the instruction register(IR).

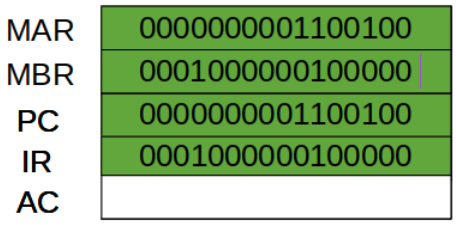


Figure 4.1.1.1: Step #3 of Fetch Cycle. ("Step #3 of the Fetch Cycle" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

Thus, a simple *Fetch Cycle* consist of three steps and four micro-operation. Symbolically, we can write these sequence of events as follows:-

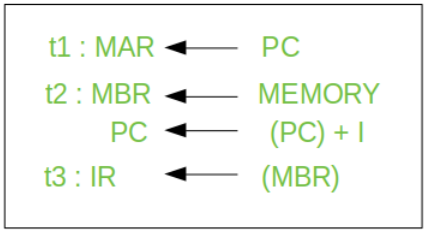


Figure 4.1.1.1: Fetch Cycle Steps. ("Fetch Cycle Steps" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

Here 'I' is the instruction length. The notations (t1, t2, t3) represents successive time units. We assume that a clock is available for timing purposes and it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are of equal duration. Each micro-operation can be performed within the time of a single time unit.

First time unit: Move the contents of the PC to MAR. The contents of the Program Counter contains the address (location) of the instruction being executed at the current time. As each instruction gets fetched, the program counter increases its stored value by 1. After each instruction is fetched, the program counter points to the next instruction in the sequence. When the computer restarts or is reset, the program counter normally reverts to 0. The MAR stores this address.

Second time unit: Move contents of memory location specified by MAR to MBR. Remember - the MDB contains the value to be stored in memory or the last value read from the memory, in this example it is an instruction to be executed. Also in this time unit the PC content gets incremented by 1.

Third time unit: Move contents of MBR to IR. Now the instructin register contains the instruction we need to execute.
Note: Second and third micro-operations both take place during the second time unit.

Adapted from:
"Computer Organization | Different Instruction Cycles" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 4.1.1: Instruction Cycles - Fetch is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

4.1.2: Instruction Cycles - Instruction Primer

How Instructions Work

On the previous page we showed how the fetched instruction is loaded into the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor will read the instruction and performs the necessary action. In general, these actions fall into four categories:

- **Processor - memory:** Data may be transferred from processor to memory, or from memory to processor.
- **Processor - I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and the system's I/O module.
- **Data processing:** The processor may perform some arithmetic or logic operation on data, such as addition or comparisons.
- **Control:** An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which specifies that the next instruction be from location 182. The processor sets the program counter to 182. Thus, on the next fetch stage, the instruction will be fetched from location 182 rather than 150.

It is not the intent of this course to cover assembly language programming, or the concepts taught there. However, in order to provide an example the following details are provided.

In this simplistic example, both the instructions and data are 16 bits long. The instructions have to following format:

Figure 4.1.2.1: Instruction Layout. ("Instruction Layout" by Patrick McClanahan is licensed under CC BY-SA 4.0)

The opcode is a numerical representation of the instruction to be performed, instructions such as mathematical or logic operations. The opcode tells the processor what it needs to do. The second part of the instructions tells the processor WHERE to find the data that is being operated on. (we will see more specifically how this works in a moment).

Figure 4.1.2.2: Data Layout. ("Data Layout" by Patrick McClanahan is licensed under CC BY-SA 4.0)

When reading data from a memory location the first bit is a sign bit, and the other 15 bits are for the actual data.

In our example, we include an Accumulator (AC) which will be used as a temporary storage location for the data.

There will be 3 opcodes - PLEASE understand these are sample opcode for this example...do NOT confuse these with actual processor opcodes.

1. 0001 - this opcode tells the processor to load the accumulator (AC) from the given memory address.
2. 0011 - this opcode tells the processor to add to the value currently stored in the AC from the specified memory address.
3. 0111 - this opcode tells the processor to move the value in the AC to the specified memory address.

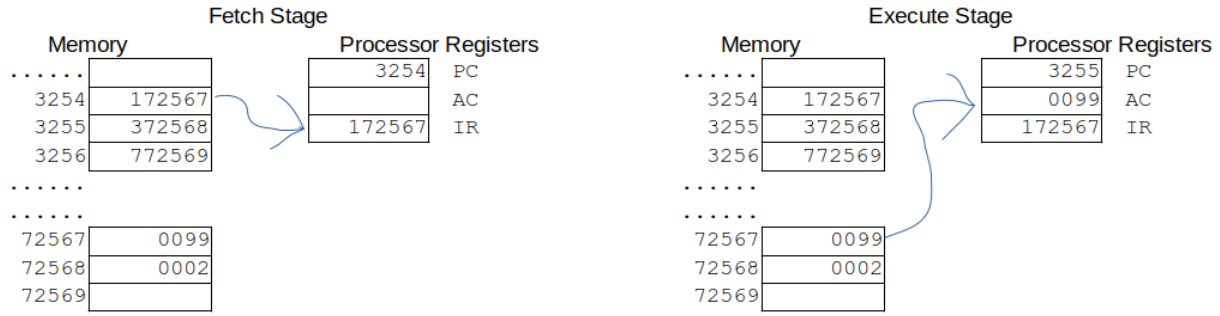


Figure 4.1.2.3: First Cycle. ("First Cycle" by Patrick McClanahan is licensed under CC BY-SA 4.0)

1. At the beginning the PC contains 3254, which is the memory address of the next instruction to be executed. In this example we have skipped the micro-steps, showing the IR receiving the value at the specified address.
2. The instruction at address 3254 is 172567. Remember from above - the first 4 bits are the opcode, in this case it is the number 1 (0001 in binary).
3. This opcode tells the processor to load the AC from the memory address located in the last 12 bits of the instruction - 72567.
4. Go to address 72567 and load that value, 0099, into the accumulator.

ALSO NOTICE - the PC has been incremented by 1, so it now points to the next instruction in memory.

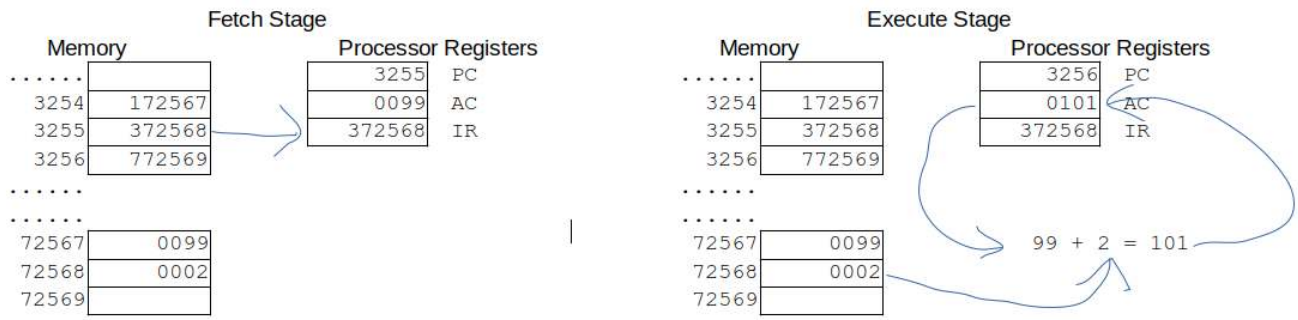


Figure 4.1.2.4: Second Cylce ("Second Cycle" by Patrick McClanahan is licensed under CC BY-SA 4.0)

1. Again - we start with the PC - and move the contents found at the memory address, 3255, into the IR.
2. The instruction in the IR has an opcode of 3 (0011 in binary).
3. This opcode in our example tells the processor to add the value currently stored in the AC, 0099, to the value stored at the given memory address, 72568, which is the value 2.
4. This value, 101, is stored back into the AC.

AGAIN, the PC has been incremented by one as well.

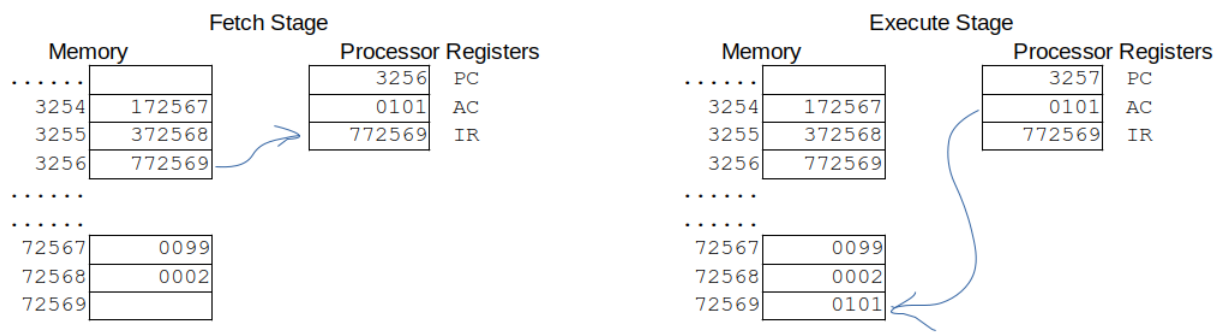


Figure 4.1.2.5: Third Cycle. ("Third Cycle" by Patrick McClanahan is licensed under CC BY-SA 4.0)

1. The PC points to 3256, the value 772569 is moved to the IR.
2. This instruction has an opcode of 7 (0111 in binary)
3. This opcode tells the processor to move the value in the AC, 101, to the specified memory address, 72569.

The PC has again been incremented by one - and when our simple 3 instructions are completed, whatever instruction was at that address would be executed.

This page titled 4.1.2: Instruction Cycles - Instruction Primer is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

4.2: Interrupts

What is an Interrupt

An interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention. It alerts the processor to a high priority process requiring interruption of the current working process. In I/O devices one of the bus control lines is dedicated for this purpose and is called the *Interrupt Service Routine (ISR)*.

When a device raises an interrupt at lets say process i, the processor first completes the execution of instruction i. Then it loads the Program Counter (PC) with the address of the first instruction of the ISR. Before loading the Program Counter with the address, the address of the interrupted instruction is moved to a temporary location. Therefore, after handling the interrupt the processor can continue with process i+1.

While the processor is handling the interrupts, it must inform the device that its request has been recognized so that it stops sending the interrupt request signal. Also, saving the registers so that the interrupted process can be restored in the future, increases the delay between the time an interrupt is received and the start of the execution of the ISR. This is called Interrupt Latency.

Hardware Interrupts:
In a hardware interrupt, all the devices are connected to the Interrupt Request Line. A single request line is used for all the n devices. To request an interrupt, a device closes its associated switch. When a device requests an interrupts, the value of INTR is the logical OR of the requests from individual devices.

Sequence of events involved in handling an IRQ:

1. Devices raise an IRQ.
2. Processor interrupts the program currently being executed.
3. Device is informed that its request has been recognized and the device deactivates the request signal.
4. The requested action is performed.
5. Interrupt is enabled and the interrupted program is resumed.

Conceptually an interrupt causes the following to happen:

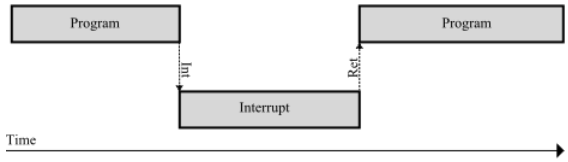


Figure 4.2.1: Concept of an interrupt. ("Concept of an Interrupt" by lemilxavier, WikiBooks is licensed under CC BY-SA 3.0)

The grey bars represent the control flow. The top line is the program that is currently running, and the bottom bar is the interrupt service routine (ISR). Notice that when the interrupt (**Int**) occurs, the program stops executing and the microcontroller begins to execute the ISR. Once the ISR is complete, the microcontroller returns to processing the program where it left off.

Handling Multiple Devices:
When more than one device raises an interrupt request signal, then additional information is needed to decide which which device to be considered first. The following methods are used to decide which device to select: Polling, Vectored Interrupts, and Interrupt Nesting. These are explained as following below.

1. **Polling:**
   In polling, the first device encountered with with IRQ bit set is the device that is to be serviced first. Appropriate ISR is called to service the same. It is easy to implement but a lot of time is wasted by interrogating the IRQ bit of all devices.
2. **Vectored Interrupts:**
   In vectored interrupts, a device requesting an interrupt identifies itself directly by sending a special code to the processor over the bus. This enables the processor to identify the device that generated the interrupt. The special code can be the starting address of the ISR or where the ISR is located in memory, and is called the interrupt vector.

3. **Interrupt Nesting:**
   In this method, I/O device is organized in a priority structure. Therefore, interrupt request from a higher priority device is recognized where as request from a lower priority device is not. To implement this each process/device (even the processor). Processor accepts interrupts only from devices/processes having priority more than it.

What happens when external hardware requests another interrupt while the processor is already in the middle of executing the ISR for a previous interrupt request?

When the first interrupt was requested, hardware in the processor causes it to finish the current instruction, disable further interrupts, and jump to the interrupt handler.

The processor ignores further interrupts until it gets to the part of the interrupt handler that has the "return from interrupt" instruction, which re-enables interrupts.

If an interrupt request occurs while interrupts were turned off, some processors will immediately jump to that interrupt handler as soon as interrupts are turned back on. With this sort of processor, an interrupt storm "starves" the main loop background task. Other processors execute at least one instruction of the main loop before handling the interrupt, so the main loop may execute extremely slowly, but at least it never "starves".

A few processors have an interrupt controller that supports "round robin scheduling", which can be used to prevent a different kind of "starvation" of low-priority interrupt handlers.

Processors priority is encoded in a few bits of PS (Process Status register). It can be changed by program instructions that write into the PS. Processor is in supervised mode only while executing OS routines. It switches to user mode before executing application programs

Adapted from:
"Interrupts" by lemilxavier, Geeks for Geeks is licensed under CC BY-SA 4.0
"Microprocessor Design/Interrupts" by lemilxavier, WikiBooks is licensed under CC BY-SA 3.0

This page titled 4.2: Interrupts is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

5: Computer Architecture - Memory

5.1: Memory Hierarchy

5.2: Memory Hierarchy (continued)

5.3: Cache Memory

5.3.1: Cache Memory - Multilevel Cache

5.3.2: Cache Memory - Locality of reference

5.4: Direct Memory Access

5: Computer Architecture - Memory is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.1: Memory Hierarchy

Memory

Memory Hierarchy

The CPU can only directly fetch instructions and data from cache memory, located directly on the processor chip. Cache memory must be loaded in from the main system memory (the Random Access Memory, or RAM). RAM however, only retains its contents when the power is on, so needs to be stored on more permanent storage.

We call these layers of memory the *memory hierarchy*

Table 3.1. Memory Hierarchy

Speed	Memory	Description
Fastest	Cache	Cache memory is memory actually embedded inside the CPU. Cache memory is very fast, typically taking only once cycle to access, but since it is embedded directly into the CPU there is a limit to how big it can be. In fact, there are several sub-levels of cache memory (termed L1, L2, L3) all with slightly increasing speeds.
	RAM	All instructions and storage addresses for the processor must come from RAM. Although RAM is very fast, there is still some significant time taken for the CPU to access it (this is termed *latency*). RAM is stored in separate, dedicated chips attached to the motherboard, meaning it is much larger than cache memory.
Slowest	Disk	We are all familiar with software arriving on a floppy disk or CDROM, and saving our files to the hard disk. We are also familiar with the long time a program can take to load from the hard disk -- having physical mechanisms such as spinning disks and moving heads means disks are the slowest form of storage. But they are also by far the largest form of storage.

The important point to know about the memory hierarchy is the trade offs between speed and size — the faster the memory the smaller it is. Of course, if you can find a way to change this equation, you'll end up a billionaire!

The reason caches are effective is because computer code generally exhibits two forms of locality

1. *Spatial* locality suggests that data within blocks is likely to be accessed together.
2. *Temporal* locality suggests that data that was used recently will likely be used again shortly.

This means that benefits are gained by implementing as much quickly accessible memory (temporal) storing small blocks of relevant information (spatial) as practically possible.

Cache in depth

Cache is one of the most important elements of the CPU architecture. To write efficient code developers need to have an understanding of how the cache in their systems works.

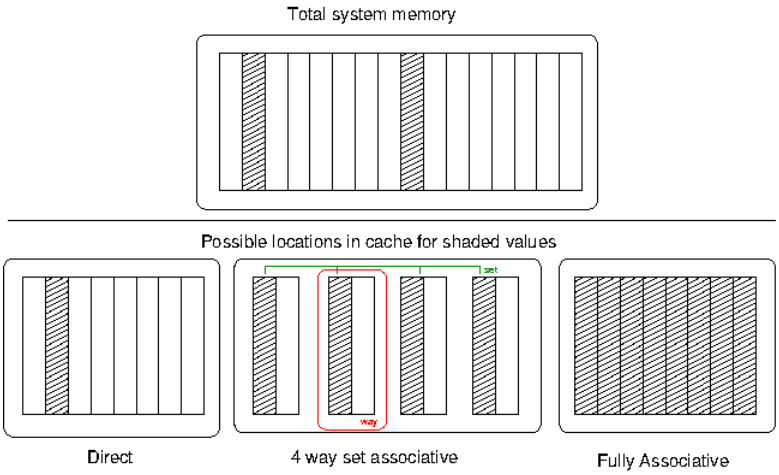
The cache is a very fast copy of the slower main system memory. Cache is much smaller than main memories because it is included inside the processor chip alongside the registers and processor logic. This is prime real estate in computing terms, and there are both economic and physical limits to its maximum size. As manufacturers find more and more ways to cram more and more transistors onto a chip cache sizes grow considerably, but even the largest caches are tens of megabytes, rather than the gigabytes of main memory or terabytes of hard disk otherwise common.

The cache is made up of small chunks of mirrored main memory. The size of these chunks is called the *line size*, and is typically something like 32 or 64 bytes. When talking about cache, it is very common to talk about the line size, or a cache line, which refers to one chunk of mirrored main memory. The cache can only load and store memory in sizes a multiple of a cache line.

Caches have their own hierarchy, commonly termed L1, L2 and L3. L1 cache is the fastest and smallest; L2 is bigger and slower, and L3 more so.

L1 caches are generally further split into instruction caches and data, known as the "Harvard Architecture" after the relay based Harvard Mark-1 computer which introduced it. Split caches help to reduce pipeline bottlenecks as earlier pipeline stages tend to reference the instruction cache and later stages the data cache. Apart from reducing contention for a shared resource, providing separate caches for instructions also allows for alternate implementations which may take advantage of the nature of instruction streaming; they are read-only so do not need expensive on-chip features such as multi-porting, nor need to handle handle sub-block reads because the instruction stream generally uses more regular sized accesses.

Figure 3.4. Cache Associativity



A given cache line may find a valid home in one of the shaded entries.

During normal operation the processor is constantly asking the cache to check if a particular address is stored in the cache, so the cache needs some way to very quickly find if it has a valid line present or not. If a given address can be cached anywhere within the cache, every cache line needs to be searched every time a reference is made to determine a hit or a miss. To keep searching fast this is done in parallel in the cache hardware, but searching every entry is generally far too expensive to implement for a reasonable sized cache. Thus the cache can be made simpler by enforcing limits on where a particular address must live. This is a trade-off; the cache is obviously much, much smaller than the system memory, so some addresses must *alias* others. If two addresses which alias each other are being constantly updated they are said to *fight* over the cache line. Thus we can categorise caches into three general types, illustrated in Figure 3.4, “Cache Associativity”.

- *Direct mapped* caches will allow a cache line to exist only in a singe entry in the cache. This is the simplest to implement in hardware, but as illustrated in Figure 3.4, “Cache Associativity” there is no potential to avoid aliasing because the two shaded addresses must share the same cache line.
- *Fully Associative* caches will allow a cache line to exist in any entry of the cache. This avoids the problem with aliasing, since any entry is available for use. But it is very expensive to implement in hardware because every possible location must be looked up simultaneously to determine if a value is in the cache.

- *Set Associative* caches are a hybrid of direct and fully associative caches, and allow a particular cache value to exist in some subset of the lines within the cache. The cache is divided into even compartments called *ways*, and a particular address could be located in any way. Thus an *n*-way set associative cache will allow a cache line to exist in any entry of a set sized total blocks mod n — Figure 3.4, "Cache Associativity" shows a sample 8-element, 4-way set associative cache; in this case the two addresses have four possible locations, meaning only half the cache must be searched upon lookup. The more ways, the more possible locations and the less aliasing, leading to overall better performance.

Once the cache is full the processor needs to get rid of a line to make room for a new line. There are many algorithms by which the processor can choose which line to evict; for example *least recently used* (LRU) is an algorithm where the oldest unused line is discarded to make room for the new line.

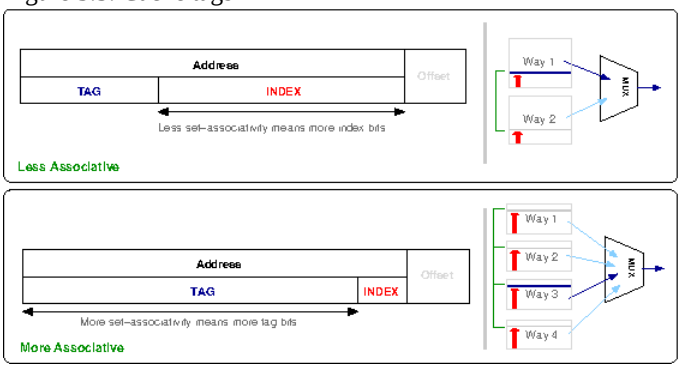
When data is only read from the cache there is no need to ensure consistency with main memory. However, when the processor starts writing to cache lines it needs to make some decisions about how to update the underlying main memory. A *write-through* cache will write the changes directly into the main system memory as the processor updates the cache. This is slower since the process of writing to the main memory is, as we have seen, slower. Alternatively a *write-back* cache delays writing the changes to RAM until absolutely necessary. The obvious advantage is that less main memory access is required when cache entries are written. Cache lines that have been written but not committed to memory are referred to as *dirty*. The disadvantage is that when a cache entry is evicted, it may require two memory accesses (one to write dirty data main memory, and another to load the new data).

If an entry exists in both a higher-level and lower-level cache at the same time, we say the higher-level cache is *inclusive*. Alternatively, if the higher-level cache having a line removes the possibility of a lower level cache having that line, we say it is *exclusive*. This choice is discussed further in the section called "Cache exclusivity in SMP systems".

Cache Addressing

So far we have not discussed how a cache decides if a given address resides in the cache or not. Clearly, caches must keep a directory of what data currently resides in the cache lines. The cache directory and data may co-located on the processor, but may also be separate — such as in the case of the POWER5 processor which has an on-core L3 directory, but actually accessing the data requires traversing the L3 bus to access off-core memory. An arrangement like this can facilitate quicker hit/miss processing without the other costs of keeping the entire cache on-core.

Figure 3.5. Cache tags



Tags need to be checked in parallel to keep latency times low; more tag bits (i.e. less set associativity) requires more complex hardware to achieve this. Alternatively more set associativity means less tags, but the processor now needs hardware to multiplex the output of the many sets, which can also add latency.

To quickly decide if an address lies within the cache it is separated into three parts; the *tag* and the *index* and the *offset*.

The offset bits depend on the line size of the cache. For example, a 32-byte line size would use the last 5-bits (i.e. 2^5) of the address as the offset into the line.

The *index* is the particular cache line that an entry may reside in. As an example, let us consider a cache with 256 entries. If this is a direct-mapped cache, we know the data may reside in only one possible line, so the next 8-bits (2^8) after the offset describe the line to check - between 0 and 255.

Now, consider the same 256 element cache, but divided into two ways. This means there are two groups of 128 lines, and the given address may reside in either of these groups. Consequently only 7-bits are required as an index to offset into the 128-entry ways. For a given cache size, as we increase the number of ways, we decrease the number of bits required as an index since each way gets smaller.

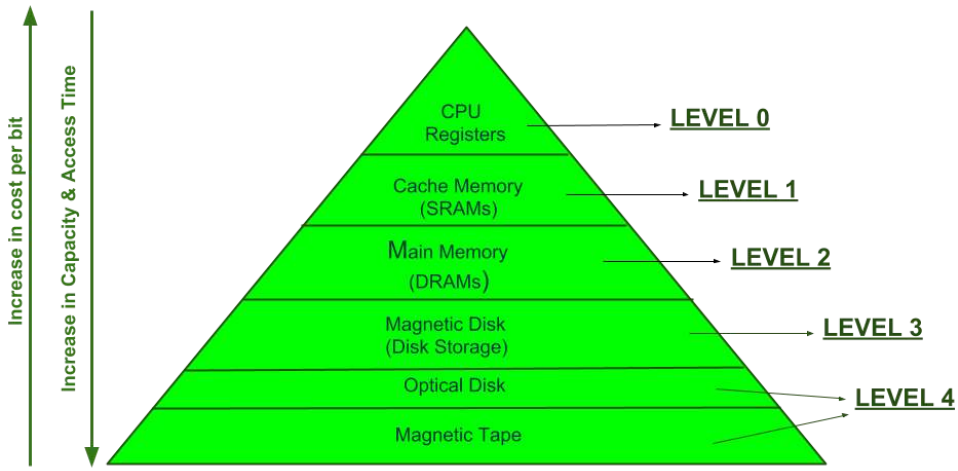
The cache directory still needs to check if the particular address stored in the cache is the one it is interested in. Thus the remaining bits of the address are the *tag* bits which the cache directory checks against the incoming address tag bits to determine if there is a cache hit or not. This relationship is illustrated in Figure 3.5, "Cache tags".

When there are multiple ways, this check must happen in parallel within each way, which then passes its result into a multiplexor which outputs a final *hit* or *miss* result. As describe above, the more associative a cache is, the less bits are required for index and the more as tag bits — to the extreme of a fully-associative cache where no bits are used as index bits. The parallel matching of tags bits is the expensive component of cache design and generally the limiting factor on how many lines (i.e, how big) a cache may grow.

This page titled 5.1: Memory Hierarchy is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

5.2: Memory Hierarchy (continued)

Memory Hierarchy is an enhancement to organize the memory such that it can minimize the access time. The Memory Hierarchy was developed based on a program behavior known as locality of references.The figure below clearly demonstrates the different levels of memory hierarchy :



MEMORY HIERARCHY DESIGN

This Memory Hierarchy Design is divided into 2 main types:

1. **External Memory or Secondary Memory –**
   Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via I/O Module.
2. **Internal Memory or Primary Memory –**
   Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.

We can infer the following characteristics of Memory Hierarchy Design from above figure:

1. **Capacity:**
   It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.
2. **Access Time:**
   It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.
3. **Performance:**
   Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the most significant ways to increase system performance is minimizing how far down the memory hierarchy one has to go to manipulate data.
4. **Cost per bit:**
   As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

Adapted from:
"Memory Hierarchy Design and its Characteristics" by RishabhJain12, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 5.2: Memory Hierarchy (continued) is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

5.3: Cache Memory

Cache Memory

Cache Memory is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data.

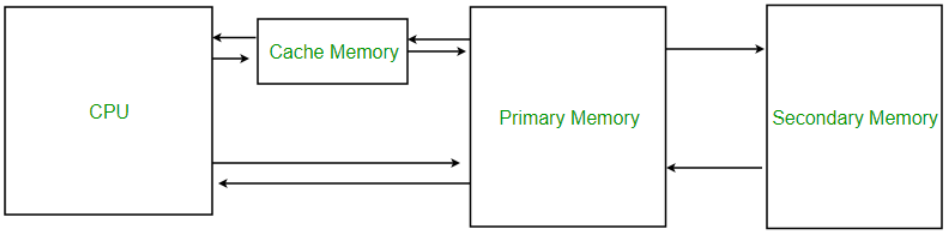


Figure 5.3.1: Cahce Memory. ("Cache Memory" by VaibhavRai3, Geeks for Geeks is licensed under CC BY-SA 4.0)

Levels of memory:

- **Level 1 or Register**
  It is a type of memory in which data is stored and accepted that are immediately stored in CPU. Most commonly used register is accumulator, Program counter, address register etc.
- **Level 2 or Cache memory**
  It is the fastest memory which has faster access time where data is temporarily stored for faster access.
- **Level 3 or Main Memory (Primary Memory in the image above)**
  It is memory on which computer works currently. It is small in size and once power is off data no longer stays in this memory.
- **Level 4 or Secondary Memory**
  It is external memory which is not as fast as main memory but data stays permanently in this memory.

Cache Performance:
When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.

- If the processor finds that the memory location is in the cache, a **cache hit** has occurred and data is read from cache
- If the processor **does not** find the memory location in the cache, a **cache miss** has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

The performance of cache memory is frequently measured in terms of a quantity called **Hit ratio.**

```
Hit ratio = hit / (hit + miss) =  no. of hits/total accesses
```

We can improve Cache performance using higher cache block size, higher associativity, reduce miss rate, reduce miss penalty, and reduce the time to hit in the cache.

Application of Cache Memory

1. Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory.
2. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

Types of Cache

- **Primary Cache**
  A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers.
- **Secondary Cache**
  Secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the level 2 (L2) cache. Often, the Level 2 cache is also housed on the processor chip.

Locality of reference

Since size of cache memory is less as compared to main memory. So to check which part of main memory should be given priority and loaded in cache is decided based on locality of reference. Locality will be discussed in greater detail later on.

Adapted from:
"Cache Memory in Computer Organization" by VaibhavRai3, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 5.3: Cache Memory is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

5.3.1: Cache Memory - Multilevel Cache

Multilevel Cache

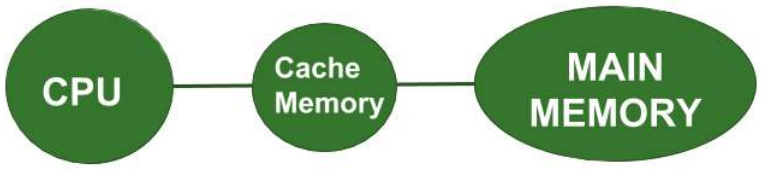
Multilevel cache is one of the techniques to improve cache performance by reducing the “miss penalty”. The term miss penalty refers to the extra time required to bring the data into cache from the main memory whenever there is a “miss” in cache .

For clear understanding let us consider an example where CPU requires 10 memory references for accessing the desired information and consider this scenario in the following 3 cases of System design :

System Design without cache memory

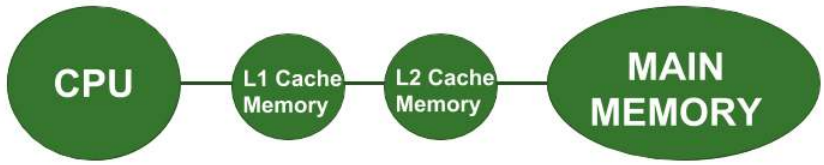
Here the CPU directly communicates with the main memory and no caches are involved.
In this case, the CPU needs to access the main memory 10 times to access the desired information.

System Design with cache memory



Here the CPU at first checks whether the desired data is present in the cache memory or not i.e. whether there is a “hit” in cache or “miss” in cache. Suppose there are 3 miss in cache memory then the main memory will be accessed only 3 times. We can see that here the miss penalty is reduced because the main memory is accessed a lesser number of times than that in the previous case.

System Design with Multilevel cache memory



Here the cache performance is optimized further by introducing multilevel Caches. As shown in the above figure, we are considering 2 level cache Design. Suppose there are *3 miss* in the L1 cache memory and out of these 3 misses there are *2 miss* in the L2 cache memory then the Main Memory will be accessed only 2 times. It is clear that here the miss penalty is reduced considerably than that in the previous case thereby improving the performance of cache memory.

NOTE :
We can observe from the above 3 cases that we are trying to decrease the number of main memory references and thus decreasing the miss penalty in order to improve the overall system performance. Also, it is important to note that in the multilevel cache design, L1 cache is attached to the CPU and it is small in size but fast. Although, L2 cache is attached to the primary cache i.e. L1 cache and it is larger in size and slower but still faster than the main memory.

```
Effective Access Time = Hit rate * Cache access time
                      + Miss rate * Lower level access time
```

Average access Time For Multilevel Cache:(T_{avg})

$T_{avg} = H_1 * C_1 + (1 – H_1) * (H_2 * C_2 +(1 – H_2) *M)$

where
H1 is the Hit rate in the L1 caches.
H2 is the Hit rate in the L2 cache.
C1 is the Time to access information in the L1 caches.
C2 is the Miss penalty to transfer information from the L2 cache to an L1 cache.
M is the Miss penalty to transfer information from the main memory to the L2 cache.

Adapted from:
"Multilevel Cache Organisation" by shreya garg 4, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 5.3.1: Cache Memory - Multilevel Cache is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

5.3.2: Cache Memory - Locality of reference

Locality of Reference

Locality of reference refers to a phenomenon in which a computer program tends to access same set of memory locations for a particular time period. In other words, Locality of Reference refers to the tendency of the computer program to access instructions whose addresses are near one another. The property of locality of reference is mainly shown by:

1. Loops in program cause the CPU to repeatedly execute a set of instructions that constitute the loop.
2. Subroutine calls, cause the set of instructions are fetched from memory each time the subroutine gets called.
3. References to data items also get localized, meaning the same data item is referenced again and again.

Even though accessing memory is quite fast, it is possible for repeated calls for data from main memory can become a bottleneck. By using faster cache memory, it is possible to speed up the retrieval of frequently used instructions or data.

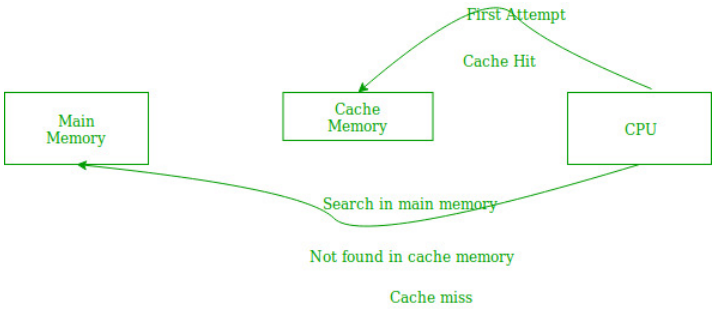


Figure 5.3.2.1: Cache Hit / Cache Miss. ("Cache Hit / Miss" by balwant_singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

In the above figure, you can see that the CPU wants to read or fetch the data or instruction. First, it will access the cache memory as it is near to it and provides very fast access. If the required data or instruction is found, it will be fetched. This situation is known as a cache hit. But if the required data or instruction is not found in the cache memory then this situation is known as a cache miss. Now the main memory will be searched for the required data or instruction that was being searched and if found will go through one of the two ways:

1. The inefficent method is to have the CPU fetch the required data or instruction from main memory and use it. When the same data or instruction is required again the CPU again has to access the main memory to retrieve it again .
2. A much more efficient method is to store the data or instruction in the cache memory so that if it is needed soon again in the near future it could be fetched in a much faster manner.

Cache Operation:

This concept is based on the idea of locality of reference. There are two ways in which data or instruction are fetched from main memory then get stored in cache memory:

1. **Temporal Locality**
   Temporal locality means current data or instruction that is being fetched may be needed soon. So we should store that data or instruction in the cache memory so that we can avoid again searching in main memory for the same data.

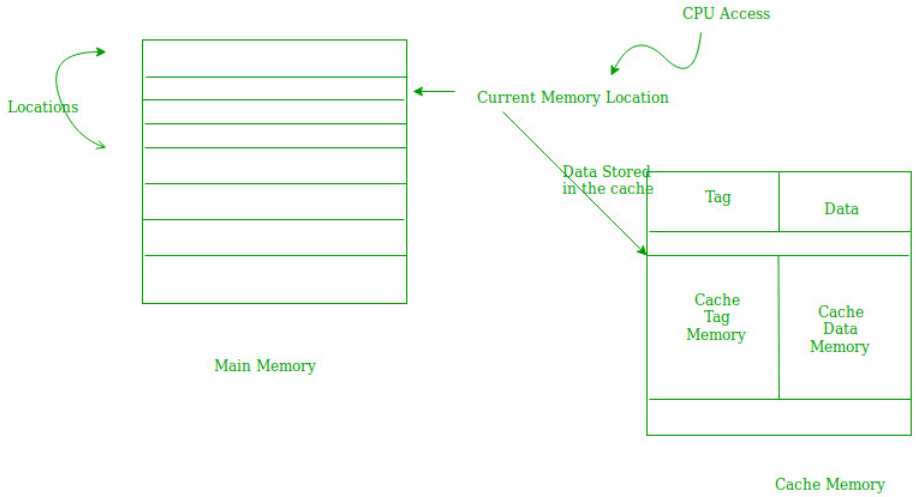


Figure 5.3.2.1: Temporal Locality. ("Temporal Locality" by balwant_singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

2. When CPU accesses the current main memory location for reading required data or instruction, it also gets stored in the cache memory which is based on the fact that same data or instruction may be needed in near future. This is known as temporal locality. If some data is referenced, then there is a high probability that it will be referenced again in the near future.
3. **Spatial Locality**
   Spatial locality means instruction or data near to the current memory location that is being fetched, may be needed by the processor soon. This is different from the temporal locality in that we are making a guess that the data/instructions will be needed soon. With temporal locality we were talking about the actual memory location that was being fetched.

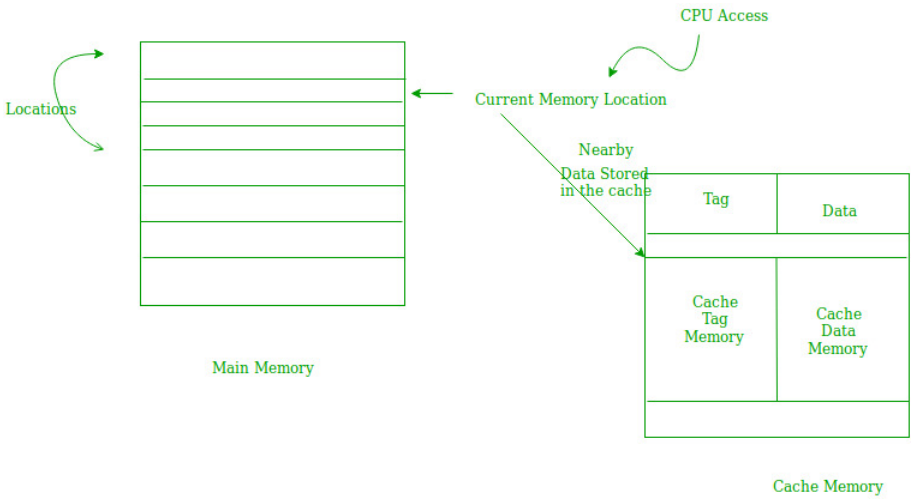


Figure 5.3.2.1: Spatial Locality. ("Spatial Locality" by **balwant_singh**, Geeks for Geeks is licensed under CC BY-SA 4.0)

Adapted From:
"Locality of Reference and Cache Operation in Cache Memory" by balwant_singh, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 5.3.2: Cache Memory - Locality of reference is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

5.4: Direct Memory Access

DMA

There are three techniques used for I/O operations: programmed I/O, interrupt-driven I/O, and direct memory access (DMA). As long as we are discussing DMA, we will also discuss the other two techniques.

The method that is used to transfer information between internal storage and external I/O devices is known as I/O interface. The CPU is interfaced using special communication links by the peripherals connected to any computer system. These communication links are used to resolve the differences between CPU and peripheral. There exists special hardware components between CPU and peripherals to supervise and synchronize all the input and output transfers that are called interface units.

Mode of Transfer:

The binary information that is received from an external device is usually stored in the memory unit. The information that is transferred from the CPU to the external device is originated from the memory unit. CPU merely processes the information but the source and target is always the memory unit. Data transfer between CPU and the I/O devices may be done in different modes.

Data transfer to and from the peripherals may be done in any of the three possible ways

1. Programmed I/O.
2. Interrupt- initiated I/O.
3. Direct memory access(DMA).

Now let's discuss each mode one by one.

1. **Programmed I/O:** It is due to the result of the I/O instructions that are written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually the transfer is from a CPU register and memory. In this case it requires constant monitoring by the CPU of the peripheral devices.

   Example of Programmed I/O: In this case, the I/O device does not have direct access to the memory unit. A transfer from I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from device to the CPU and store instruction to transfer the data from CPU to memory. In programmed I/O, the CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This is a time consuming process since it needlessly keeps the CPU busy. This situation can be avoided by using an interrupt facility. This is discussed below.

2. **Interrupt- initiated I/O:** Since in the above case we saw the CPU is kept busy unnecessarily. This situation can very well be avoided by using an interrupt driven method for data transfer. By using interrupt facility and special commands to inform the interface to issue an interrupt request signal whenever data is available from any device. In the meantime the CPU can proceed for any other program execution. The interface meanwhile keeps monitoring the device. Whenever it is determined that the device is ready for data transfer it initiates an interrupt request signal to the computer. Upon detection of an external interrupt signal the CPU stops momentarily the task that it was already performing, branches to the service program to process the I/O transfer, and then return to the task it was originally performing. \

Note: Both the methods programmed I/O and Interrupt-driven I/O require the active intervention of the processor to transfer data between memory and the I/O module, and any data transfer must transverse a path through the processor. Thus both these forms of I/O suffer from two inherent drawbacks.

- The I/O transfer rate is limited by the speed with which the processor can test and service a device.
- The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

3. **Direct Memory Access**: The data transfer between a fast storage media such as magnetic disk and memory unit is limited by the speed of the CPU. Thus we can allow the peripherals directly communicate with each other using the memory buses, removing the intervention of the CPU. This type of data transfer technique is known as DMA or direct memory access. During

DMA the CPU is idle and it has no control over the memory buses. The DMA controller takes over the buses to manage the transfer directly between the I/O devices and the memory unit.

Bus Request : It is used by the DMA controller to request the CPU to relinquish the control of the buses.

Bus Grant : It is activated by the CPU to Inform the external DMA controller that the buses are in high impedance state and the requesting DMA can take control of the buses. Once the DMA has taken the control of the buses it transfers the data. This transfer can take place in many ways.

Types of DMA transfer using DMA controller (DMAC):

Burst Transfer :
DMA returns the bus after complete data transfer. A register is used as a byte count, being decremented for each byte transfer, and upon the byte count reaching zero, the DMAC will release the bus. When the DMAC operates in burst mode, the CPU is halted for the duration of the data transfer.

1. Bus grant request time.
2. Transfer the entire block of data at transfer rate of device because the device is usually slow than the speed at which the data can be transferred to CPU.
3. Release the control of the bus back to CPU
   So, total time taken to transfer the N bytes = Bus grant request time + (N) * (memory transfer rate) + Bus release control time.

Cyclic Stealing : An alternative method in which DMA controller transfers one word at a time after which it must return the control of the buses to the CPU. The CPU delays its operation only for one memory cycle to allow the direct memory I/O transfer to “steal” one memory cycle.

Steps Involved are:

1. Buffer the byte into the buffer
2. Inform the CPU that the device has 1 byte to transfer (i.e. bus grant request)
3. Transfer the byte (at system bus speed)
4. Release the control of the bus back to CPU.

Before moving on transfer next byte of data, device performs step 1 again so that bus isn’t tied up and the transfer won’t depend upon the transfer rate of device.

Adapted from:
"I/O Interface (Interrupt and DMA Mode)" by saripallisriharsha2, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 5.4: Direct Memory Access is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

6: Computer Architecture - Peripherals and Buses

6.1: Peripherals and buses
6.2: The Processor - Bus

This page titled 6: Computer Architecture - Peripherals and Buses is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

6.1: Peripherals and buses

Peripherals and buses

Peripherals are any of the many external devices that connect to your computer. Obviously, the processor must have some way of talking to the peripherals to make them useful.

The communication channel between the processor and the peripherals is called a *bus*.

Peripheral Bus concepts

A device requires both input and output to be useful. There are a number of common concepts required for useful communication with peripherals.

Interrupts

An interrupt allows the device to literally interrupt the processor to flag some information. For example, when a key is pressed, an interrupt is generated to deliver the key-press event to the operating system. Each device is assigned an interrupt by some combination of the operating system and BIOS.

Devices are generally connected to an *programmable interrupt controller* (PIC), a separate chip that is part of the motherboard which buffers and communicates interrupt information to the main processor. Each device has a physical *interrupt line* between it an one of the PIC's provided by the system. When the device wants to interrupt, it will modify the voltage on this line.

A very broad description of the PIC's role is that it receives this interrupt and converts it to a message for consumption by the main processor. While the exact procedure varies by architecture, the general principle is that the operating system has configured an *interrupt descriptor table* which pairs each of the possible interrupts with a code address to jump to when the interrupt is received. This is illustrated in Figure 6.1.1, “Overview of handling an interrupt”.

Writing this *interrupt handler* is the job of the device driver author in conjunction with the operating system.

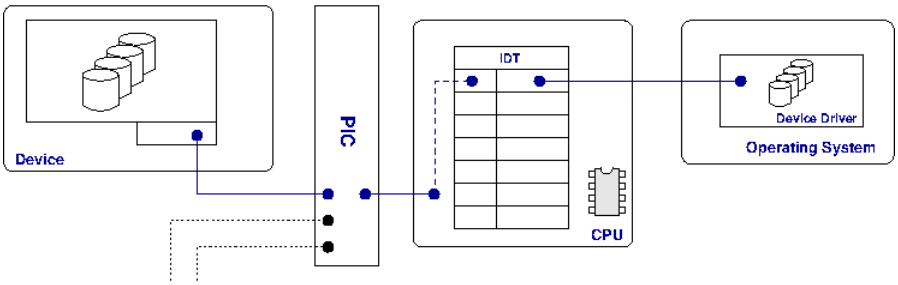


Figure 6.1.1: Overview of handling an interrupt. ("Computer Science from the Bottom Up" by Ian Wienand is licensed under CC BY-SA 3.0)

A generic overview of handling an interrupt. The device raises the interrupt to the interrupt controller, which passes the information onto the processor. The processor looks at its descriptor table, filled out by the operating system, to find the code to handle the fault.

Most drivers will split up handling of interrupts into *bottom* and *top* halves. The bottom half will acknowledge the interrupt, queue actions for processing and return the processor to what it was doing quickly. The top half will then run later when the CPU is free and do the more intensive processing. This is to stop an interrupt hogging the entire CPU.

Saving state

Since an interrupt can happen at any time, it is important that you can return to the running operation when finished handling the interrupt. It is generally the job of the operating system to ensure that upon entry to the interrupt handler, it saves any *state*; i.e. registers, and restores them when returning from the interrupt handler. In this way, apart from some lost time, the interrupt is completely transparent to whatever happens to be running at the time.

Interrupts v traps and exceptions

While an interrupt is generally associated with an external event from a physical device, the same mechanism is useful for handling internal system operations. For example, if the processor detects conditions such as an access to invalid memory, an attempt to divide-by-zero or an invalid instruction, it can internally raise an *exception* to be handled by the operating system. It is also the mechanism used to trap into the operating system for *system calls*. Although generated internally rather than from an external source, the principles of asynchronously interrupting the running code remains the same.

Types of interrupts

There are two main ways of signalling interrupts on a line — *level* and *edge* triggered.

Level-triggered interrupts define voltage of the interrupt line being held high to indicate an interrupt is pending. Edge-triggered interrupts detect *transitions* on the bus; that is when the line voltage goes from low to high. With an edge-triggered interrupt, a square-wave pulse is detected by the PIC as signalling and interrupt has been raised.

The difference is pronounced when devices share an interrupt line. In a level-triggered system, the interrupt line will be high until all devices that have raised an interrupt have been processed and un-asserted their interrupt.

In an edge-triggered system, a pulse on the line will indicate to the PIC that an interrupt has occurred, which it will signal to the operating system for handling.

The issue with level-triggered interrupts is that it may require some considerable amount of time to handle an interrupt for a device. During this time, the interrupt line remains high and it is not possible to determine if any other device has raised an interrupt on the line. This means there can be considerable unpredictable latency in servicing interrupts.

With edge-triggered interrupts, a long-running interrupt can be noticed and queued, but other devices sharing the line can still transition (and hence raise interrupts) while this happens. However, this introduces new problems; if two devices interrupt at the same time it may be possible to miss one of the interrupts, or environmental or other interference may create a *spurious* interrupt which should be ignored.

Non-maskable interrupts

It is important for the system to be able to *mask* or prevent interrupts at certain times. Generally, it is possible to put interrupts on hold, but a particular class of interrupts, called *non-maskable interrupts* (NMI), are the exception to this rule. The typical example is the *reset* interrupt.

NMIs can be useful for implementing things such as a system watchdog, where a NMI is raised periodically and sets some flag that must be acknowledged by the operating system. If the acknowledgement is not seen before the next periodic NMI, then system can be considered to be not making forward progress. Another common usage is for profiling a system. A periodic NMI can be raised and used to evaluate what code the processor is currently running; over time this builds a profile of what code is being run and create a very useful insight into system performance.

IO Space

Obviously the processor will need to communicate with the peripheral device, and it does this via IO operations. The most common form of IO is so called *memory mapped IO* where registers on the device are *mapped* into memory.

This means that to communicate with the device, you need simply read or write to a specific address in memory. TODO: expand

DMA

Since the speed of devices is far below the speed of processors, there needs to be some way to avoid making the CPU wait around for data from devices.

Direct Memory Access (DMA) is a method of transferring data directly between an peripheral and system RAM.

The driver can setup a device to do a DMA transfer by giving it the area of RAM to put its data into. It can then start the DMA transfer and allow the CPU to continue with other tasks.

Once the device is finished, it will raise an interrupt and signal to the driver the transfer is complete. From this time the data from the device (say a file from a disk, or frames from a video capture card) is in memory and ready to be used.

Other Buses

Other buses connect between the PCI bus and external devices.

USB

From an operating system point of view, a USB device is a group of end-points grouped together into an interface. An end-point can be either *in* or *out* and hence transfers data in one direction only. End-points can have a number of different types:

- *Control* end-points are for configuring the device, etc.
- *Interrupt* end-points are for transferring small amounts of data. They have higher priority than ...
- *Bulk* end-points, which transfer large amounts of data but do not get guaranteed time constraints.
- *Isochronous* transfers are high-priority real-time transfers, but if they are missed they are not re-tried. This is for streaming data like video or audio where there is no point sending data again.

There can be many interfaces (made of multiple end-points) and interfaces are grouped into *configurations*. However most devices only have a single configuration.

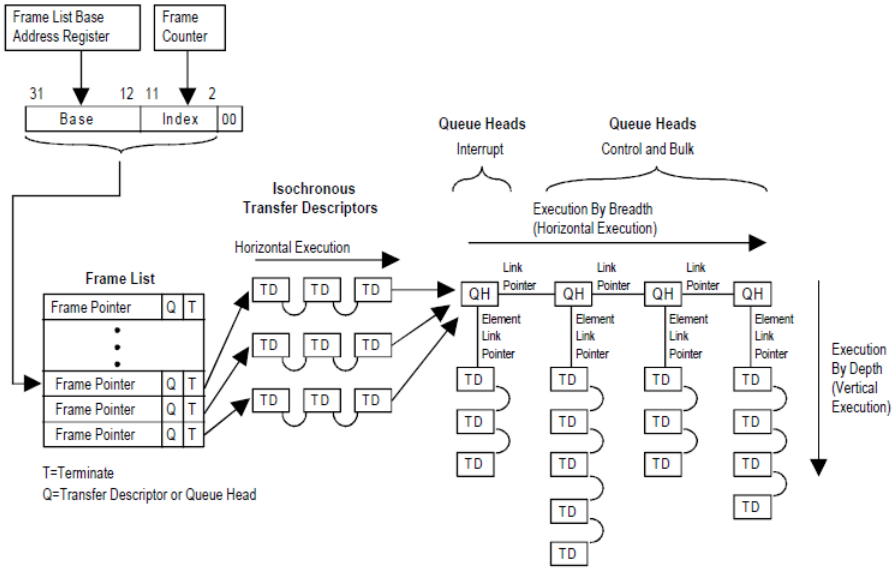


Figure 6.1.2: Overview of a UHCI controller operation ("Computer Science from the Bottom Up" by Ian Wienand is licensed under CC BY-SA 3.0)

An overview of a UCHI controller, taken from Intel documentation.

Figure 6.1.2, "Overview of a UHCI controller operation" shows an overview of a universal host controller interface, or UHCI. It provides an overview of how USB data is moved out of the system by a combination of hardware and software. Essentially, the software sets up a template of data in a specified format for the host controller to read and send across the USB bus.

Starting at the top-left of the overview, the controller has a *frame* register with a counter which is incremented periodically — every millisecond. This value is used to index into a *frame list* created by software. Each entry in this table points to a queue of *transfer descriptors*. Software sets up this data in memory, and it is read by the host controller which is a separate chip the drives the USB bus. Software needs to schedule the work queues so that 90% of a frame time is given to isochronous data, and 10% left for interrupt, control and bulk data..

As you can see from the diagram, the way the data is linked means that transfer descriptors for isochronous data are associated with only one particular frame pointer — in other words only one particular time period — and after that will be discarded. However, the interrupt, control and bulk data are all *queued* after the isochronous data and thus if not transmitted in one frame (time period) will be done in the next.

The USB layer communicates through USB *request blocks*, or URBs. A URB contains information about what end-point this request relates to, data, any related information or attributes and a call-back function to be called when the URB is complete. USB drivers submit URBs in a fixed format to the USB core, which manages them in co-ordination with the USB host controller as above. Your data gets sent off to the USB device by the USB core, and when its done your call-back is triggered.

This page titled 6.1: Peripherals and buses is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

6.2: The Processor - Bus

Control Bus

In computer architecture, a control bus is part of the system bus, used by CPUs for communicating with other devices within the computer. While the address bus carries the information about the device with which the CPU is communicating and the data bus carries the actual data being processed, the control bus carries commands from the CPU and returns status signals from the devices. For example, if the data is being read or written to the device the appropriate line (read or write) will be active (logic one).

Address bus

An address bus is a bus that is used to specify a physical address. When a processor or DMA-enabled device needs to read or write to a memory location, it specifies that memory location on the address bus (the value to be read or written is sent on the data bus). The width of the address bus determines the amount of memory a system can address. For example, a system with a 32-bit address bus can address 232 (4,294,967,296) memory locations. If each memory location holds one byte, the addressable memory space is 4 GiB.

Data / Memory Bus

The memory bus is the computer bus which connects the main memory to the memory controller in computer systems. Originally, general-purpose buses like VMEbus and the S-100 bus were used, but to reduce latency, modern memory buses are designed to connect directly to DRAM chips, and thus are designed by chip standards bodies such as JEDEC. Examples are the various generations of SDRAM, and serial point-to-point buses like SLDRAM and RDRAM. An exception is the Fully Buffered DIMM which, despite being carefully designed to minimize the effect, has been criticized for its higher latency.

Adapted from:
"Bus (computing)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Memory bus" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Control bus" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 6.2: The Processor - Bus is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

7: Small to Big Systems

7.1: Symmetric Multi-Processing Systems
7.2: Multiprocessor and Multicore Systems

7: Small to Big Systems is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

7.1: Symmetric Multi-Processing Systems

Small to big systems

As Moore's law has predicted, computing power has been growing at a furious pace and shows no signs of slowing down. It is relatively uncommon for any high end servers to contain only a single CPU. This is achieved in a number of different fashions.

Symmetric Multi-Processing

Symmetric Multi-Processing, commonly shortened to *SMP*, is currently the most common configuration for including multiple CPUs in a single system.

The symmetric term refers to the fact that all the CPUs in the system are the same (e.g. architecture, clock speed). In a SMP system there are multiple processors that share other all other system resources (memory, disk, etc).

Cache Coherency

For the most part, the CPUs in the system work independently; each has its own set of registers, program counter, etc. Despite running separately, there is one component that requires strict synchronisation.

This is the CPU cache; remember the cache is a small area of quickly accessible memory that mirrors values stored in main system memory. If one CPU modifies data in main memory and another CPU has an old copy of that memory in its cache the system will obviously not be in a consistent state. Note that the problem only occurs when processors are writing to memory, since if a value is only read the data will be consistent.

To co-ordinate keeping the cache coherent on all processors an SMP system uses *snooping*. Snooping is where a processor listens on a bus which all processors are connected to for cache events, and updates its cache accordingly.

One protocol for doing this is the *MOESI* protocol; standing for Modified, Owner, Exclusive, Shared, Invalid. Each of these is a state that a cache line can be in on a processor in the system. There are other protocols for doing as much, however they all share similar concepts. Below we examine MOESI so you have an idea of what the process entails.

When a processor requires reading a cache line from main memory, it firstly has to snoop all other processors in the system to see if they currently know anything about that area of memory (e.g. have it cached). If it does not exist in any other process, then the processor can load the memory into cache and mark it as *exclusive*. When it writes to the cache, it then changes state to be *modified*. Here the specific details of the cache come into play; some caches will immediately write back the modified cache to system memory (known as a *write-through* cache, because writes go through to main memory). Others will not, and leave the modified value only in the cache until it is evicted, when the cache becomes full for example.

The other case is where the processor snoops and finds that the value is in another processors cache. If this value has already been marked as *modified*, it will copy the data into its own cache and mark it as *shared*. It will send a message for the other processor (that we got the data from) to mark its cache line as *owner*. Now imagine that a third processor in the system wants to use that memory too. It will snoop and find both a *shared* and a *owner* copy; it will thus take its value from the *owner* value. While all the other processors are only reading the value, the cache line stays *shared* in the system. However, when one processor needs to update the value it sends an *invalidate* message through the system. Any processors with that cache line must then mark it as invalid, because it not longer reflects the "true" value. When the processor sends the invalidate message, it marks the cache line as *modified* in its cache and all others will mark as *invalid* (note that if the cache line is *exclusive* the processor knows that no other processor is depending on it so can avoid sending an invalidate message).

From this point the process starts all over. Thus whichever processor has the *modified* value has the responsibility of writing the true value back to RAM when it is evicted from the cache. By thinking through the protocol you can see that this ensures consistency of cache lines between processors.

There are several issues with this system as the number of processors starts to increase. With only a few processors, the overhead of checking if another processor has the cache line (a read snoop) or invalidating the data in every other processor (invalidate snoop) is manageable; but as the number of processors increase so does the bus traffic. This is why SMP systems usually only scale up to around 8 processors.

Having the processors all on the same bus starts to present physical problems as well. Physical properties of wires only allow them to be laid out at certain distances from each other and to only have certain lengths. With processors that run at many gigahertz the speed of light starts to become a real consideration in how long it takes messages to move around a system.

Note that system software usually has no part in this process, although programmers should be aware of what the hardware is doing underneath in response to the programs they design to maximise performance.

Cache exclusivity in SMP systems

In the section called "Cache in depth" we described *inclusive* v *exclusive* caches. In general, L1 caches are usually inclusive — that is all data in the L1 cache also resides in the L2 cache. In a multiprocessor system, an inclusive L1 cache means that only the L2 cache need snoop memory traffic to maintain coherency, since any changes in L2 will be guaranteed to be reflected by L1. This reduces the complexity of the L1 and de-couples it from the snooping process allowing it to be faster.

Again, in general, most all modern high-end (e.g. not targeted at embedded) processors have a write-through policy for the L1 cache, and a write-back policy for the lower level caches. There are several reasons for this. Since in this class of processors L2 caches are almost exclusively on-chip and generally quite fast the penalties from having L1 write-through are not the major consideration. Further, since L1 sizes are small, pools of written data unlikely to be read in the future could cause pollution of the limited L1 resource. Additionally, a write-through L1 does not have to be concerned if it has outstanding dirty data, hence can pass the extra coherency logic to the L2 (which, as we mentioned, already has a larger part to play in cache coherency).

Hyperthreading

Much of the time of a modern processor is spent waiting for much slower devices in the memory hierarchy to deliver data for processing.

Thus strategies to keep the pipeline of the processor full are paramount. One strategy, known as hyperthreading, is to include enough registers and state logic such that two instruction streams can be processed at the same time. This makes one CPU look for all intents and purposes like two CPUs.

While each CPU has its own registers, they still have to share the core logic, cache and input and output bandwidth from the CPU to memory. So while two instruction streams can keep the core logic of the processor busier, the performance increase will not be as great has having two physically separate CPUs. Typically the performance improvement is below 20% (XXX check), however it can be drastically better or worse depending on the workloads.

Multi Core

With increased ability to fit more and more transistors on a chip, it became possible to put two or more processors in the same physical package. Most common is dual-core, where two processor cores are in the same chip. These cores, unlike hyperthreading, are full processors and so appear as two physically separate processors a la a SMP system.

While generally the processors have their own L1 cache, they do have to share the bus connecting to main memory and other devices. Thus performance is not as great as a full SMP system, but considerably better than a hyperthreading system (in fact, each core can still implement hyperthreading for an additional enhancement).

Multi core processors also have some advantages not performance related. As we mentioned, external physical buses between processors have physical limits; by containing the processors on the same piece of silicon extremely close to each other some of these problems can be worked around. The power requirements for multi core processors are much less than for two separate processors. This means that there is less heat needing to be dissipated which can be a big advantage in data centre applications where computers are packed together and cooling considerations can be considerable. By having the cores in the same physical package it makes muti-processing practical in applications where it otherwise would not be, such as laptops. It is also considerably cheaper to only have to produce one chip rather than two.

Clusters

Many applications require systems much larger than the number of processors a SMP system can scale to. One way of scaling up the system further is a *cluster*.

A cluster is simply a number of individual computers which have some ability to talk to each other. At the hardware level the systems have no knowledge of each other; the task of stitching the individual computers together is left up to software.

Software such as MPI allow programmers to write their software and then "farm out" parts of the program to other computers in the system. For example, image a loop that executes several thousand times performing independent action (that is no iteration of the loop affects any other iteration). With four computers in a cluster, the software could make each computer do 250 loops each.

The interconnect between the computers varies, and may be as slow as an internet link or as fast as dedicated, special buses (Infiniband). Whatever the interconnect, however, it is still going to be further down the memory hierarchy and much, much slower than RAM. Thus a cluster will not perform well in a situation when each CPU requires access to data that may be stored in the RAM of another computer; since each time this happens the software will need to request a copy of the data from the other computer, copy across the slow link and into local RAM before the processor can get any work done.

However, many applications *do not* require this constant copying around between computers. One large scale example is SETI@Home, where data collected from a radio antenna is analysed for signs of Alien life. Each computer can be distributed a few minutes of data to analyse, and only needs report back a summary of what it found. SETI@Home is effectively a very large, dedicated cluster.

Another application is rendering of images, especially for special effects in films. Each computer can be handed a single frame of the movie which contains the wire-frame models, textures and light sources which needs to be combined (rendered) into the amazing special effects we now take for grained. Since each frame is static, once the computer has the initial input it does not need any more communication until the final frame is ready to be sent back and combined into the move. For example the block-buster Lord of the Rings had their special effects rendered on a huge cluster running Linux.

Non-Uniform Memory Access

Non-Uniform Memory Access, more commonly abbreviated to NUMA, is almost the opposite of a cluster system mentioned above. As in a cluster system it is made up of individual nodes linked together, however the linkage between nodes is highly specialised (and expensive!). As opposed to a cluster system where the hardware has no knowledge of the linkage between nodes, in a NUMA system the *software* has no (well, less) knowledge about the layout of the system and the hardware does all the work to link the nodes together.

The term *non uniform memory access* comes from the fact that RAM may not be local to the CPU and so data may need to be accessed from a node some distance away. This obviously takes longer, and is in contrast to a single processor or SMP system where RAM is directly attached and always takes a constant (uniform) time to access.

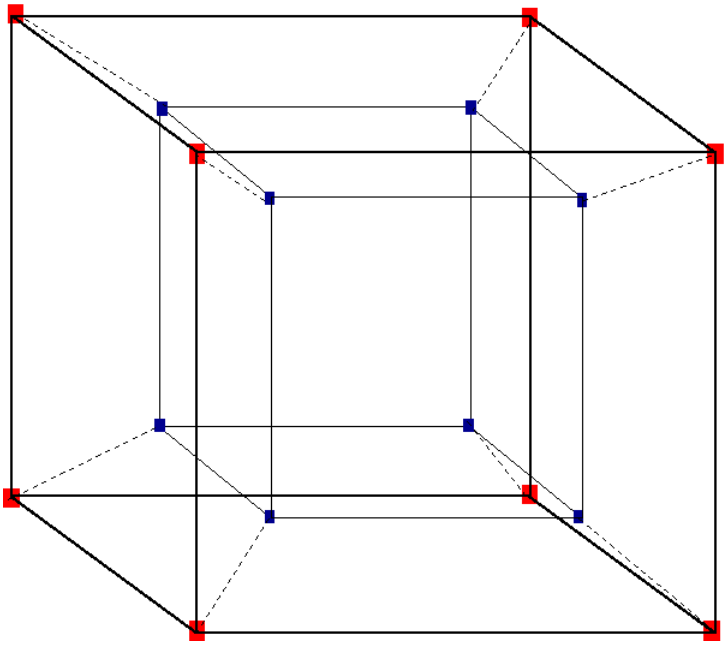
NUMA Machine Layout

With so many nodes talking to each other in a system, minimising the distance between each node is of paramount importance. Obviously it is best if every single node has a direct link to every other node as this minimises the distance any one node needs to go to find data. This is not a practical situation when the number of nodes starts growing into the hundreds and thousands as it does with large supercomputers; if you remember your high school maths the problem is basically a combination taken two at a time (each node talking to another), and will grow `n!/2*(n-2)!` .

To combat this exponential growth alternative layouts are used to trade off the distance between nodes with the interconnects required. One such layout common in modern NUMA architectures is the hypercube.

A hypercube has a strict mathematical definition (way beyond this discussion) but as a cube is a 3 dimensional counterpart of a square, so a hypercube is a 4 dimensional counterpart of a cube.

Figure 3.8. A Hypercube



An example of a hypercube. Hypercubes provide a good trade off between distance between nodes and number of interconnections required.

Above we can see the outer cube contains four 8 nodes. The maximum number of paths required for any node to talk to another node is 3. When another cube is placed inside this cube, we now have double the number of processors but the maximum path cost has only increased to 4. This means as the number of processors grow by 2^n the maximum path cost grows only linearly.

Cache Coherency

Cache coherency can still be maintained in a NUMA system (this is referred to as a cache-coherent NUMA system, or ccNUMA). As we mentioned, the broadcast based scheme used to keep the processor caches coherent in an SMP system does not scale to hundreds or even thousands of processors in a large NUMA system. One common scheme for cache coherency in a NUMA system is referred to as a *directory based model*. In this model processors in the system communicate to special cache directory hardware. The directory hardware maintains a consistent picture to each processor; this abstraction hides the working of the NUMA system from the processor.

The Censier and Feautrier directory based scheme maintains a central directory where each memory block has a flag bit known as the *valid bit* for each processor and a single *dirty* bit. When a processor reads the memory into its cache, the directory sets the valid bit for that processor.

When a processor wishes to write to the cache line the directory needs to set the dirty bit for the memory block. This involves sending an invalidate message to those processors who are using the cache line (and only those processors whose flag are set; avoiding broadcast traffic).

After this should any other processor try to read the memory block the directory will find the dirty bit set. The directory will need to get the updated cache line from the processor with the valid bit currently set, write the dirty data back to main memory and then provide that data back to the requesting processor, setting the valid bit for the requesting processor in the process. Note that this is transparent to the requesting processor and the directory may need to get that data from somewhere very close or somewhere very far away.

Obviously having thousands of processors communicating to a single directory does also not scale well. Extensions to the scheme involve having a hierarchy of directories that communicate between each other using a separate protocol. The directories can use a more general purpose communications network to talk between each other, rather than a CPU bus, allowing scaling to much larger systems.

NUMA systems are best suited to the types of problems that require much interaction between processor and memory. For example, in weather simulations a common idiom is to divide the environment up into small "boxes" which respond in different ways (oceans and land reflect or store different amounts of heat, for example). As simulations are run, small variations will be fed in to see what the overall result is. As each box influences the surrounding boxes (e.g. a bit more sun means a particular box puts out more heat, affecting the boxes next to it) there will be much communication (contrast that with the individual image frames for a rendering process, each of which does not influence the other). A similar process might happen if you were modelling a car crash, where each small box of the simulated car folds in some way and absorbs some amount of energy.

Although the software has no directly knowledge that the underlying system is a NUMA system, programmers need to be careful when programming for the system to get maximum performance. Obviously keeping memory close to the processor that is going to use it will result in the best performance. Programmers need to use techniques such as *profiling* to analyse the code paths taken and what consequences their code is causing for the system to extract best performance.

Memory ordering, locking and atomic operations

The multi-level cache, superscalar multi-processor architecture brings with it some interesting issues relating to how a programmer sees the processor running code.

Imagine program code is running on two processors simultaneously, both processors sharing effectively one large area of memory. If one processor issues a store instruction, to put a register value into memory, when can it be sure that the other processor does a load of that memory it will see the correct value?

In the simplest situation the system could guarantee that if a program executes a store instruction, any subsequent load instructions will see this value. This is referred to as *strict memory ordering*, since the rules allow no room for movement. You should be starting to realise why this sort of thing is a serious impediment to performance of the system.

Much of the time, the memory ordering is not required to be so strict. The programmer can identify points where they need to be sure that all outstanding operations are seen globally, but in between these points there may be many instructions where the semantics are not important.

Take, for example, the following situation.

Example 3.1. Memory Ordering

```
typedef struct {
int a;
int b;
} a_struct;

/*
 * Pass in a pointer to be allocated as a new structure
 */
void get_struct(a_struct *new_struct)
{
    void *p = malloc(sizeof(a_struct));

    /* We don't particularly care what order the following two
     * instructions end up acutally executing in */
    p->a = 100;
    p->b = 150;
```

```
        /* However, they must be done before this instruction.
         * Otherwise, another processor who looks at the value of p
         * could find it pointing into a structure whose values have
         * not been filled out.
         */
        new_struct = p;
    }

```

In this example, we have two stores that can be done in any particular order, as it suits the processor. However, in the final case, the pointer must only be updated once the two previous stores are known to have been done. Otherwise another processor might look at the value of `p`, follow the pointer to the memory, load it, and get some completely incorrect value!

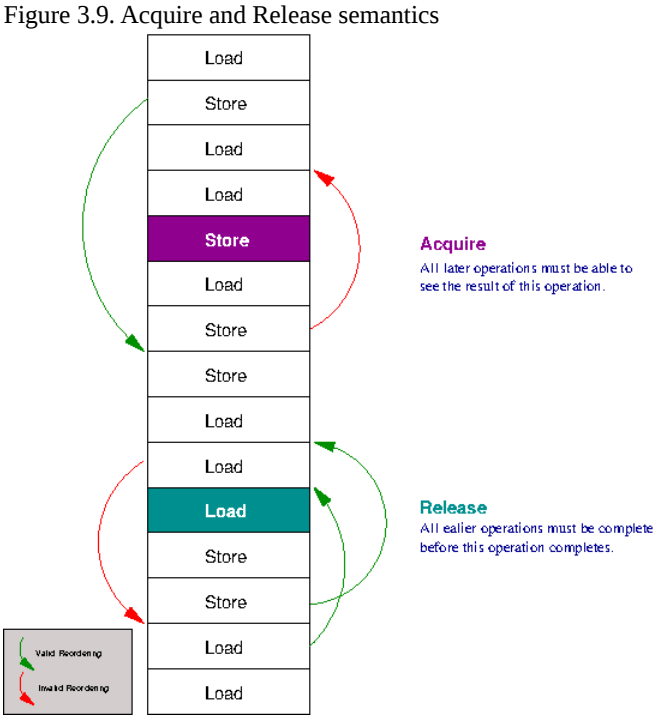
To indicate this, loads and stores have to have *semantics* that describe what behaviour they must have. Memory semantics are described in terms of *fences* that dictate how loads and stores may be reordered around the load or store.

By default, a load or store can be re-ordered anywhere.

Acquire semantics is like a fence that only allows load and stores to move downwards through it. That is, when this load or store is complete you can be guaranteed that any later load or stores will see the value (since they can not be moved above it).

Release semantics is the opposite, that is a fence that allows any load or stores to be done before it (move upwards), but nothing before it to move downwards past it. Thus, when load or store with release semantics is processed, you can be store that any earlier load or stores will have been complete.

Figure 3.9. Acquire and Release semantics



An illustration of valid reorderings around operations with acquire and release semantics.

A *full memory fence* is a combination of both; where no loads or stores can be reordered in any direction around the current load or store.

The strictest memory model would use a full memory fence for every operation. The weakest model would leave every load and store as a normal re-orderable instruction.

Different processors implement different memory models.

The x86 (and AMD64) processor has a quite strict memory model; all stores have release semantics (that is, the result of a store is guaranteed to be seen by any later load or store) but all loads have normal semantics. lock prefix gives memory fence.

Itanium allows all load and stores to be normal, unless explicitly told. XXX

Locking

Knowing the memory ordering requirements of each architecture is not practical for all programmers, and would make programs difficult to port and debug across different processor types.

Programmers use a higher level of abstraction called *locking* to allow simultaneous operation of programs when there are multiple CPUs.

When a program acquires a lock over a piece of code, no other processor can obtain the lock until it is released. Before any critical pieces of code, the processor must attempt to take the lock; if it can not have it, it does not continue.

You can see how this is tied into the naming of the memory ordering semantics in the previous section. We want to ensure that before we *acquire* a lock, no operations that should be protected by the lock are re-ordered before it. This is how acquire semantics works.

Conversely, when we *release* the lock, we must be sure that every operation we have done whilst we held the lock is complete (remember the example of updating the pointer previously?). This is release semantics.

There are many software libraries available that allow programmers to not have to worry about the details of memory semantics and simply use the higher level of abstraction of `lock()` and `unlock()` .

Locking difficulties

Locking schemes make programming more complicated, as it is possible to *deadlock* programs. Imagine if one processor is currently holding a lock over some data, and is currently waiting for a lock for some other piece of data. If that other processor is waiting for the lock the first processor holds before unlocking the second lock, we have a deadlock situation. Each processor is waiting for the other and neither can continue without the others lock.

Often this situation arises because of a subtle *race condition*; one of the hardest bugs to track down. If two processors are relying on operations happening in a specific order in time, there is always the possibility of a race condition occurring. A gamma ray from an exploding star in a different galaxy might hit one of the processors, making it skip a beat, throwing the ordering of operations out. What will often happen is a deadlock situation like above. It is for this reason that program ordering needs to be ensured by semantics, and not by relying on one time specific behaviours. (XXX not sure how i can better word that).

A similar situation is the opposite of deadlock, called *livelock*. One strategy to avoid deadlock might be to have a "polite" lock; one that you give up to anyone who asks. This politeness might cause two threads to be constantly giving each other the lock, without either ever taking the lock long enough to get the critical work done and be finished with the lock (a similar situation in real life might be two people who meet at a door at the same time, both saying "no, you first, I insist". Neither ends up going through the door!).

Locking strategies

Underneath, there are many different strategies for implementing the behaviour of locks.

A simple lock that simply has two states - locked or unlocked, is referred to as a *mutex* (short for mutual exclusion; that is if one person has it the other can not have it).

There are, however, a number of ways to implement a mutex lock. In the simplest case, we have what its commonly called a *spinlock*. With this type of lock, the processor sits in a tight loop waiting to take the lock; equivalent to it saying "can I have it now" constantly much as a young child might ask of a parent.

The problem with this strategy is that it essentially wastes time. Whilst the processor is sitting constantly asking for the lock, it is not doing any useful work. For locks that are likely to be only held locked for a very short amount of time this may be appropriate, but in many cases the amount of time the lock is held might be considerably longer.

Thus another strategy is to *sleep* on a lock. In this case, if the processor can not have the lock it will start doing some other work, waiting for notification that the lock is available for use (we see in future chapters how the operating system can switch processes and give the processor more work to do).

A mutex is however just a special case of a *semaphore*, famously invented by the Dutch computer scientist Dijkstra. In a case where there are multiple resources available, a semaphore can be set to count accesses to the resources. In the case where the number of resources is one, you have a mutex. The operation of semaphores can be detailed in any algorithms book.

These locking schemes still have some problems however. In many cases, most people only want to read data which is updated only rarely. Having all the processors wanting to only read data require taking a lock can lead to *lock contention* where less work gets done because everyone is waiting to obtain the same lock for some data.

This page titled 7.1: Symmetric Multi-Processing Systems is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

7.2: Multiprocessor and Multicore Systems

MultiProcessor System

Two or more processors or CPUs present in same computer, sharing system bus, memory and I/O is called MultiProcessing System. It allows parallel execution of different processors. These systems are reliable since failure of any single processor does not affect other processors. A quad-processor system can execute four processes at a time while an octa-processor can execute eight processes at a time. The memory and other resources may be shared or distributed among processes.

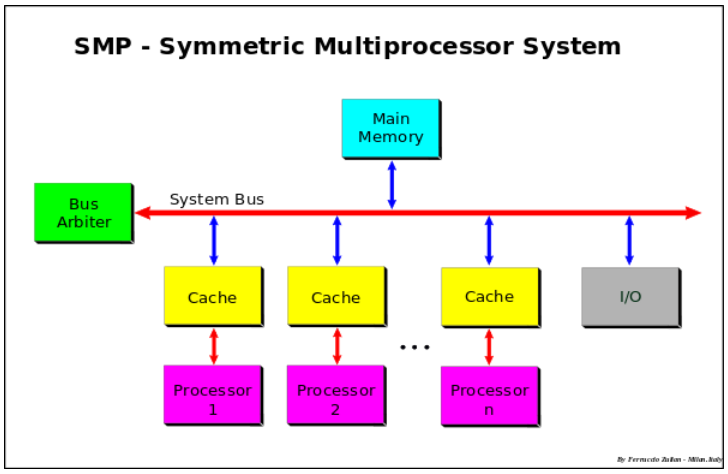


Figure 7.2.1: Symmetric multiprocessing system. ("Symmetric multiprocessing system" by Multiple Contributors, Wikipedia Commons is licensed under CC BY-SA 3.0)

Advantages :

- Since more than one processor are working at the same time, throughput will get increased.
- More reliable since failure in one CPU does not affect other.
- It needs little complex configuration.
- Parallel processing (more than one process executing at same time) is achieved through MultiProcessing.

Disadvantages :

- It will have more traffic (distances between two will require longer time).
- Throughput may get reduced in shared resources system where one processor using some I/O then another processor has to wait for its turn.
- As more than processors are working at particular instant of time. So, coordination between these is very complex.

Multicore System

A processor that has more than one core is called Multicore Processor while one with single core is called Unicore Processor or Uniprocessor. Nowadays, most of systems have four cores (Quad-core) or eight cores (Octa-core). These cores can individually read and execute program instructions, giving feel like computer system has several processors but in reality, they are cores and not processors. Instructions can be calculation, data transferring instruction, branch instruction, etc. Processor can run instructions on separate cores at same time. This increases overall speed of program execution in system. Thus heat generated by processor gets reduced and increases overall speed of execution.

Multicore systems support MultiThreading and Parallel Computing**.** Multicore processors are widely used across many application domains, including general-purpose, embedded, network, digital signal processing (DSP), and graphics (GPU). Efficient software algorithms should be used for implementation of cores to achieve higher performance. Software that can run in parallel is preferred because the desire is to achieve parallel execution with the help of multiple cores.

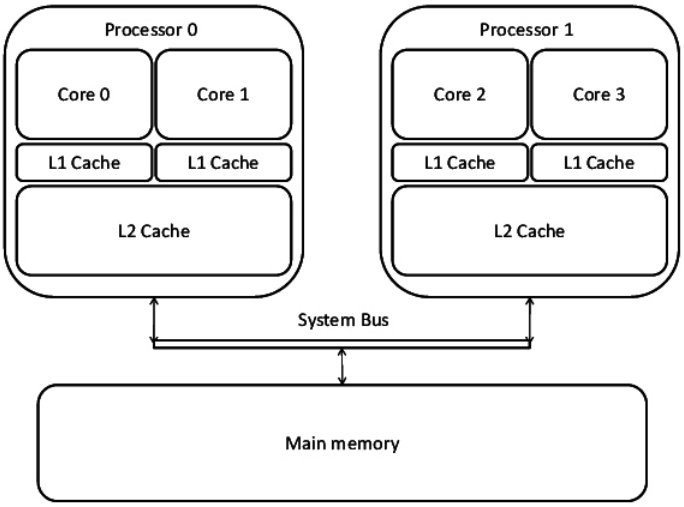


Figure 7.2.1: Quad Core Processor. ("Diagram of a generic dual-core processor " by Multiple Contributors, Wikipedia Commons is licensed under CC BY-SA 3.0)

Advantages :

- These cores are usually integrated into single IC (integrated circuit) die, or onto multiple dies but in single chip package. Thus allowing higher **Cache Coherency.**
- These systems are energy efficient since they allow higher performance at lower energy. A challenge in this, however, is additional overhead of writing parallel code.
- It will have less traffic(cores integrated into single chip and will require less time).

Disadvantages :

- Dual-core processor do not work at twice speed of single processor. They get only 60-80% more speed.
- Some Operating systems are still using single core processor.
- OS compiled for multi-core processor will run slightly slower on single-core processor

Adapted from:
"Difference between MultiCore and MultiProcessor System" by Ganeshchowdharysadanala, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 7.2: Multiprocessor and Multicore Systems is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

8: Processes

8.1: The Process
8.2: Processes
8.3: Elements of a process
8.4: Process States
8.5: Process Description
8.6: Process Control
8.7: Fork and Exec
8.8: Scheduling
8.9: Execution within the Operating System
8.9.1: Execution within the Operating System - Dual Mode
8.10: The Shell
8.11: Signals

8: Processes is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

8.1: The Process

The role of the operating system

The operating system underpins the entire operation of the modern computer.

Abstraction of hardware

The fundamental operation of the operating system (OS) is to abstract the hardware to the programmer and user. The operating system provides generic interfaces to services provided by the underlying hardware.

In a world without operating systems, every programmer would need to know the most intimate details of the underlying hardware to get anything to run. Worse still, their programs would not run on other hardware, even if that hardware has only slight differences.

Multitasking

We expect modern computers to do many different things at once, and we need some way to arbitrate between all the different programs running on the system. It is the operating systems job to allow this to happen seamlessly.

The operating system is responsible for *resource management* within the system. Many tasks will be competing for the resources of the system as it runs, including processor time, memory, disk and user input. The job of the operating system is to arbitrate these resources to the multiple tasks and allow them access in an orderly fashion. You have probably experienced when this *fails* as it usually ends up with your computer crashing (the famous "blue screen of death" for example).

Standardised Interfaces

Programmers want to write programs that will run on as many different hardware platforms as possible. By having operating system support for standardised interfaces, programmers can get this functionality.

For example, if the function to open a file on one system is `open()` , on another is `open_file()` and on yet another `openf()` programmers will have the dual problem of having to remember what each system does and their programs will not work on multiple systems.

The Portable Operating System Interface (POSIX)[9] is a very important standard implemented by UNIX type operating systems. Microsoft Windows has similar proprietary standards.

Security

On multi-user systems, security is very important. As the arbitrator of access to the system the operating system is responsible for ensuring that only those with the correct permissions can access resources.

For example if a file is owned by one user, another user should not be allowed to open and read it. However there also need to be mechanisms to share that file safely between the users should they want it.

Operating systems are large and complex programs, and often security issues will be found. Often a virus or worm will take advantage of these bugs to access resources it should not be allowed to, such as your files or network connection; to fight them you must install *patches* or updates provided by your operating system vendor.

Performance

As the operating system provides so many services to the computer, its performance is critical. Many parts of the operating system run extremely frequently, so even an overhead of just a few processor cycles can add up to a big decrease in overall system performance.

The operating system needs to exploit the features of the underlying hardware to make sure it is getting the best possible performance for the operations, and consequently systems programmers need to understand the intimate details of the architecture they are building for.

In many cases the systems programmers job is about deciding on policies for the system. Often the case that the side effects of making one part of the operating system run faster will make another part run slower or less efficiently. Systems programmers need

to understand all these trade offs when they are building their operating system.

---

[9] The X comes from *Unix*, from which the standard grew. Today, POSIX is the same thing as the Single UNIX Specification Version 3 or ISO/IEC 9945:2002. This is a free standard, available online.

Once upon a time, the Single UNIX specification and the POSIX Standards were separate entities. The Single UNIX specification was released by a consortium called the "Open Group", and was freely available as per their requirements. The latest version is The Single Unix Specification Version 3.

The IEEE POSIX standards were released as IEEE Std 1003.[insert various years, revisions here], and were not freely available. The latest version is IEEE 1003.1-2001 and is equivalent to the Single Unix Specification Version 3.

Thus finally the two separate standards were merged into what is known as the Single UNIX Specification Version 3, which is also standardised by the ISO under ISO/IEC 9945:2002. This happened early in 2002. So when people talk about POSIX, SUS3 or ISO/IEC 9945:2002 they all mean the same thing!

This page titled 8.1: The Process is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

8.2: Processes

What is a Process

In computing, a process is the instance of a computer program that is being executed by one or many threads. It contains the program code and its activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

While a computer program is a passive collection of instructions, a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often results in more than one process being executed.

Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU (core) executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish (preemption). Depending on the operating system implementation, switches could be performed when tasks initiate and wait for completion of input/output operations, when a task voluntarily yields the CPU, on hardware interrupts, and when the operating system scheduler decides that a process has expired its fair share of CPU time (e.g, by the Completely Fair Scheduler of the Linux kernel).

A common form of multitasking is provided by CPU's time-sharing that is a method for interleaving the execution of users processes and threads, and even of independent kernel tasks - although the latter feature is feasible only in preemptive kernels such as Linux. Preemption has an important side effect for interactive process that are given higher priority with respect to CPU bound processes, therefore users are immediately assigned computing resources at the simple pressing of a key or when moving a mouse. Furthermore, applications like video and music reproduction are given some kind of real-time priority, preempting any other lower priority process. In time-sharing systems, context switches are performed rapidly, which makes it seem like multiple processes are being executed simultaneously on the same processor. This simultaneous execution of multiple processes is called concurrency.

For security and reliability, most modern operating systems prevent direct communication between independent processes, providing strictly mediated and controlled inter-process communication functionality.

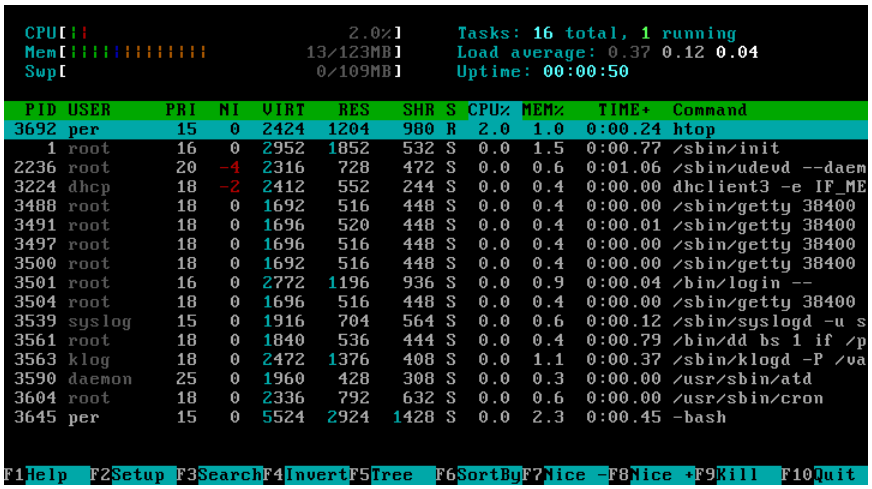


Figure 8.2.1: Output of htop Linux Command. (PER9000, Per Erik Strandberg, CC BY-SA 3.0, via Wikimedia Commons)

Child process

A child process in computing is a process created by another process (the parent process). This technique pertains to multitasking operating systems, and is sometimes called a subprocess or traditionally a subtask.

A child process inherits most of its attributes, such as file descriptors, from its parent. In Linux, a child process is typically created as a copy of the parent. The child process can then overlay itself with a different program as required.

Each process may create many child processes but will have at most one parent process; if a process does not have a parent this usually indicates that it was created directly by the kernel. In some systems, including Linux-based systems, the very first process

is started by the kernel at booting time and never terminates; other parentless processes may be launched to carry out various tasks in userspace. Another way for a process to end up without a parent is if its parent dies, leaving an orphan process; but in this case it will shortly be adopted by the main process.

Representation

In general, a computer system process consists of (or is said to own) the following resources:

- Operating system descriptors of resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows), and data sources and sinks.
- Security attributes, such as the process owner and the process' set of permissions (allowable operations).
- Processor state (context), such as the content of registers and physical memory addressing. The state is typically stored in computer registers when the process is executing, and in memory otherwise.

The operating system holds most of this information about active processes in data structures called process control blocks. Any subset of the resources, typically at least the processor state, may be associated with each of the process' threads in operating systems that support threads or child processes.

The operating system keeps its processes separate and allocates the resources they need, so that they are less likely to interfere with each other and cause system failures (e.g., deadlock or thrashing). The operating system may also provide mechanisms for inter-process communication to enable processes to interact in safe and predictable ways.

Attributes or Characteristics of a Process

A process has following attributes.

```
1. Process Id:    A unique identifier assigned by the operating system
2. Process State: Can be ready, running, etc.
3. CPU registers: Like the Program Counter (CPU registers must be saved and restored w
4. I/O status information: For example, devices allocated to the process,
                           open files, etc
5. CPU scheduling information: For example, Priority (Different processes may have dif
                               a short process may be assigned a low priority
                               in the shortest job first scheduling)
6. Various accounting information
```

All of the above attributes of a process are also known as the ***context of the process***.

Context Switching

The process of saving the context of one process and loading the context of another process is known as Context Switching. In simple terms, it is like loading and unloading the process from running state to ready state.

When does context switching happen?
1. When a high-priority process comes to ready state (i.e. with higher priority than the running process)
2. An Interrupt occurs
3. User and kernel mode switch (It is not necessary though)
4. Preemptive CPU scheduling used.

Context Switch vs Mode Switch

A mode switch occurs when CPU privilege level is changed, for example when a system call is made or a fault occurs. The kernel works in more a privileged mode than a standard user task. If a user process wants to access things which are only accessible to the kernel, a mode switch must occur. The currently executing process need not be changed during a mode switch.
A mode switch typically occurs for a process context switch to occur. Only the kernel can cause a context switch.

CPU-Bound vs I/O-Bound Processes

A CPU-bound process requires more CPU time or spends more time in the running state.
An I/O-bound process requires more I/O time and less CPU time. An I/O-bound process spends more time in the waiting state.

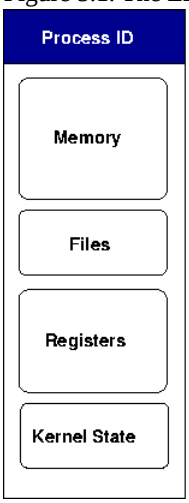
Adapted from:
"Process (computing)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Introduction of Process Management" by SarthakSinghal1, Geeks for Geeks is licensed under CC BY-SA 4.0
"Child process" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 8.2: Processes is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

8.3: Elements of a process

Elements of a process

Figure 5.1. The Elements of a Process



Process ID

The *process ID* (or the PID) is assigned by the operating system and is unique to each running process.

Memory

We will learn exactly how a process gets its memory in the following weeks -- it is one of the most fundamental parts of how the operating system works. However, for now it is sufficient to know that each process gets its own section of memory.

In this memory all the program code is stored, along with variables and any other allocated storage.

Parts of the memory can be shared between processes (called, not surprisingly *shared memory*). You will often see this called *System Five Shared Memory* (or SysV SHM) after the original implementation in an older operating system.

Another important concept a process may utilise is that of *mmap*ing a file on disk to memory. This means that instead of having to open the file and use commands such as `read()` and `write()` the file looks as if it were any other type of RAM. `mmaped` areas have permissions such as read, write and execute which need to be kept track of. As we know, it is the job of the operating system to maintain security and stability, so it needs to check if a process tries to write to a read only area and return an error.

Code and Data

A process can be further divided into *code* and `data` sections. Program code and data should be kept separately since they require different permissions from the operating system and separation facilitates sharing of code (as you see later). The operating system needs to give program code permission to be read and executed, but generally not written to. On the other hand data (variables) require read and write permissions but should not be executable[12].

The Stack

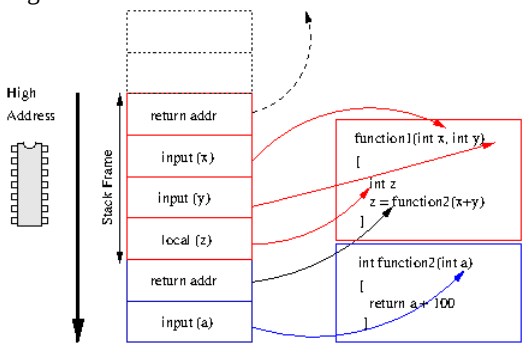
One other very important part of a process is an area of memory called *the stack*. This can be considered part of the data section of a process, and is intimately involved in the execution of any program.

A stack is generic data structure that works exactly like a stack of plates; you can *push* an item (put a plate on top of a stack of plates), which then becomes the top item, or you can *pop* an item (take a plate off, exposing the previous plate).

Stacks are fundamental to function calls. Each time a function is called it gets a new `stack frame`. This is an area of memory which usually contains, at a minimum, the address to return to when complete, the input arguments to the function and space for local variables.

By convention, stacks usually *grow down*[13] . This means that the stack starts at a high address in memory and progressively gets lower.

Figure 5.2. The Stack



We can see how having a stack brings about many of the features of functions.

- Each function has its own copy of its input arguments. This is because each function is allocated a new stack frame with its arguments in a fresh area of memory.
- This is the reason why a variable defined inside a function can not be seen by other functions. Global variables (which can be seen by any function) are kept in a separate area of data memory.
- This facilitates *recursive* calls. This means a function is free to call itself again, because a new stack frame will be created for all its local variables.
- Each frame contains the address to return to. C only allows a single value to be returned from a function, so by convention this value is returned to the calling function in a specified register, rather than on the stack.
- Because each frame has a reference to the one before it, a debugger can "walk" backwards, following the pointers up the stack. From this it can produce a *stack trace* which shows you all functions that were called leading into this function. This is extremely useful for debugging.

  You can see how the way functions works fits exactly into the nature of a stack. Any function can call any other function, which then becomes the up most function (put on top of the stack). Eventually that function will return to the function that called it (takes itself off the stack).
- Stacks do make calling functions slower, because values must be moved out of registers and into memory. Some architectures allow arguments to be passed in registers directly; however to keep the semantics that each function gets a unique copy of each argument the registers must *rotate*.
- You may have heard of the term a *stack overflow*. This is a common way of hacking a system by passing bogus values. If you as a programmer accept arbitrary input into a stack variable (say, reading from the keyboard or over the network) you need to explicitly say how big that data is going to be.

  Allowing any amount of data unchecked will simply overwrite memory. Generally this leads to a crash, but some people realised that if they overwrote just enough memory to place a specific value in the return address part of the stack frame, when the function completed rather than returning to the correct place (where it was called from) they could make it return into the data they just sent. If that data contains binary executable code that hacks the system (e.g. starts a terminal for the user with root privileges) then your computer has been compromised.

  This happens because the stack grows downwards, but data is read in "upwards" (i.e. from lower address to higher addresses).

  There are several ways around this; firstly as a programmer you must ensure that you always check the amount of data you are receiving into a variable. The operating system can help to avoid this on behalf of the programmer by ensuring that the stack is marked as *not executable*; that is that the processor will not run any code, even if a malicious user tries to pass some into your program. Modern architectures and operating systems support this functionality.

- Stacks are ultimately managed by the compiler, as it is responsible for generating the program code. To the operating system the stack just looks like any other area of memory for the process.

To keep track of the current growth of the stack, the hardware defines a register as the *stack pointer*. The compiler (or the programmer, when writing in assembler) uses this register to keep track of the current top of the stack.

Example 5.1. Stack pointer example

```
$ cat sp.c
void function(void)
{
        int i = 100;
        int j = 200;
        int k = 300;
}

$ gcc -fomit-frame-pointer -S sp.c

$ cat sp.s
        .file   "sp.c"
        .text
.globl function
        .type   function, @function
function:
        subl    $16, %esp
        movl    $100, 4(%esp)
        movl    $200, 8(%esp)
        movl    $300, 12(%esp)
        addl    $16, %esp
        ret
        .size   function, .-function
        .ident  "GCC: (GNU) 4.0.2 20050806 (prerelease) (Debian 4.0.1-4)"
        .section        .note.GNU-stack,"",@progbits
```

Above we show a simple function allocating three variables on the stack. The disassembly illustrates the use of the stack pointer on the x86 architecture[14]. Firstly we allocate some space on the stack for our local variables. Since the stack grows down, we subtract from the value held in the stack pointer. The value 16 is a value large enough to hold our local variables, but may not be exactly the size required (for example with 3 4 byte `int` values we really only need 12 bytes, not 16) to keep alignment of the stack in memory on certain boundaries as the compiler requires.

Then we move the values into the stack memory (and in a real function, use them). Finally, before returning to our parent function we "pop" the values off the stack by moving the stack pointer back to where it was before we started.

The Heap

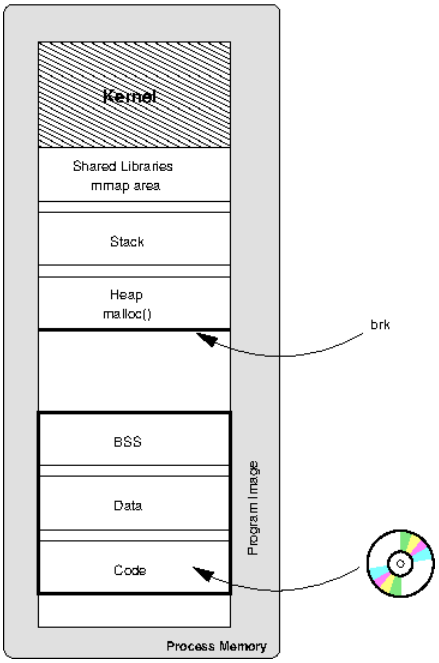
The heap is an area of memory that is managed by the process for on the fly memory allocation. This is for variables whose memory requirements are not known at compile time.

The bottom of the heap is known as the *brk*, so called for the system call which modifies it. By using the `brk` call to grow the area downwards the process can request the kernel allocate more memory for it to use.

The heap is most commonly managed by the `malloc` library call. This makes managing the heap easy for the programmer by allowing them to simply allocate and free (via the `free` call) heap memory. `malloc` can use schemes like a *buddy allocator* to manage the heap memory for the user. `malloc` can also be smarter about allocation and potentially use *anonymous mmaps* for extra process memory. This is where instead of mmaping a *file* into the process memory it directly maps an area of system RAM. This can be more efficient. Due to the complexity of managing memory correctly, it is very uncommon for any modern program to have a reason to call `brk` directly.

Memory Layout

Figure 5.3. Process memory layout



As we have seen a process has smaller areas of memory allocated to it, each with a specific purpose.

An example of how the process is laid out in memory by the kernel is given above. Starting from the top, the kernel reserves itself some memory at the top of the process (we see with virtual memory how this memory is actually shared between all processes).

Underneath that is room for `mmaped` files and libraries. Underneath that is the stack, and below that the heap.

At the bottom is the program image, as loaded from the executable file on disk. We take a closer look at the process of loading this data in later chapters.

File Descriptors

In the first week we learnt about `stdin`, `stdout` and `stderr`; the default files given to each process. You will remember that these files always have the same file descriptor number (0,1,2 respectively).

Thus, file descriptors are kept by the kernel individually for each process.

File descriptors also have permissions. For example, you may be able to read from a file but not write to it. When the file is opened, the operating system keeps a record of the processes permissions to that file in the file descriptor and doesn't allow the process to do anything it shouldn't.

Registers

We know from the previous chapter that the processor essentially performs generally simple operations on values in registers. These values are read (and written) to memory -- we mentioned above that each process is allocated memory which the kernel keeps track of.

So the other side of the equation is keeping track of the registers. When it comes time for the currently running process to give up the processor so another process can run, it needs to save its current state. Equally, we need to be able to restore this state when the

process is given more time to run on the CPU. To do this the operating system needs to store a copy of the CPU registers to memory. When it is time for the process to run again, the operating system will copy the register values back from memory to the CPU registers and the process will be right back where it left off.

Kernel State

Internally, the kernel needs to keep track of a number of elements for each process.

Process State

Another important element for the operating system to keep track of is the process state. If the process is currently running it makes sense to have it in a *running* state.

However, if the process has requested to read a file from disk we know from our memory hierarchy that this may take a significant amount of time. The process should give up its current execution to allow another process to run, but the kernel need not let the process run again until the data from the disk is available in memory. Thus it can mark the process as *disk wait* (or similar) until the data is ready.

Priority

Some processes are more important than others, and get a higher priority. See the discussion on the scheduler below.

Statistics

The kernel can keep statistics on each processes behaviour which can help it make decisions about how the process behaves; for example does it mostly read from disk or does it mostly do CPU intensive operations?

---

[12] Not all architectures support this, however. This has lead to a wide range of security problems on many architectures.

[13] Some architectures, such as PA-RISC from HP, have stacks that grow upwards. On some other architectures, such as IA64, there are other storage areas (the register backing store) that grow from the bottom toward the stack.

[14] Note we used the special flag to gcc `-fomit-frame-pointer` which specifies that an extra register should *not* be used to keep a pointer to the start of the stack frame. Having this pointer helps debuggers to walk upwards through the stack frames, however it makes one less register available for other applications.

This page titled 8.3: Elements of a process is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

8.4: Process States

Process state

In a multitasking computer system, processes may occupy a variety of states. These distinct states may not be recognized as such by the operating system kernel. However, they are a useful abstraction for the understanding of processes.

Primary process states

If we look at the following diagram there are several things to note:

- Memory:
  - We have some states of a process where the process is kept in main memory - RAM.
  - We have some states that are stored in secondary memory - that is what we call swap space and is actually part of the hard disk set aside for use by the operating system.
- There are numerous actions: Admint, dispatch, Event wait, Event occur, Suspend, Activate, Timeout, and Release. These all have an impact on the process during its lifetime.
- In the following diagram there are 7 states. Depending on who the author is there may be 5, 6 or 7. Sometimes the 2 Suspended states are not shown, but we shown them for clarity here.

Let's follow a process through a lifecycle.

1. A new process gets created. For example, a user opens up a word processor, this requires a new process.
2. Once the process has completed its initialization, it is placed in a Ready state with all of the other processes waiting to take its turn on the processor.
3. When the process's turn comes, it is dispatched to the Running state and executes on the processor. From here one of 3 things happens:
   1. the process completes and is Released and moves to the Exit state
   2. it uses up it turn and so Timeout and is returned to the Ready state
   3. some event happens, such as waiting for user input, and it is moved to the Blocked state.
      1. In the Blocked state, once the event it is waiting on occurs, it can return to the Ready state.
      2. If the event doesn't occur for an extended period of time, the process can get moved to the Suspended blocked state. Since it is suspended it is now in virtual memory - which is actually disk space set aside for temporary storage.
4. Once the event this process is waiting on does occur it is moved to the Suspended ready state, then waits to get moved from secondary storage, into main memory in the Ready state.
   1. On very busy systems processes can get moved from the Ready state to the Suspended ready state as well.
5. Eventually every process will Exit.

The following typical process states are possible on computer systems of all kinds. In most of these states, processes are "stored" on main memory.

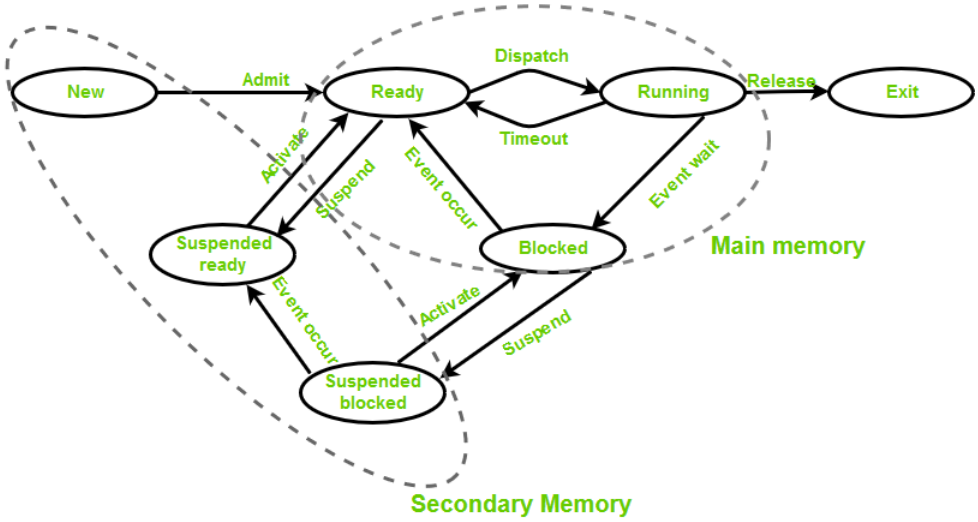


Figure 8.4.1: States of a Process. ("States of a Process" by Aniket_Dusey, Geeks for Geeks is licensed under CC BY-SA 4.0)

New or Created

When a process is first created, it occupies the "created" or "new" state. In this state, the process awaits admission to the "ready" state. Admission will be approved or delayed by a long-term, or admission, scheduler. Typically in most desktop computer systems, this admission will be approved automatically. However, for real-time operating systems this admission may be delayed. In a realtime system, admitting too many processes to the "ready" state may lead to oversaturation and overcontention of the system's resources, leading to an inability to meet process deadlines.

Operating systems need some ways to create processes. In a very simple system designed for running only a single application (e.g., the controller in a microwave oven), it may be possible to have all the processes that will ever be needed be present when the system comes up. In general-purpose systems, however, some way is needed to create and terminate processes as needed during operation.
There are four principal events that cause a process to be created:

- System initialization.
- Execution of process creation system call by a running process.
- A user request to create a new process.
- Initiation of a batch job.

When an operating system is booted, typically several processes are created. Some of these are foreground processes, that interact with a (human) user and perform work for them. Others are background processes, which are not associated with particular users, but instead have some specific function. For example, one background process may be designed to accept incoming e-mails, sleeping most of the day but suddenly springing to life when an incoming e-mail arrives. Another background process may be designed to accept an incoming request for web pages hosted on the machine, waking up when a request arrives to service that request.

There are several steps involved in process creation. The first step is the validation of whether the parent process has sufficient authorization to create a process. Upon successful validation, the parent process is copied almost entirely, with changes only to the unique process id, parent process, and user-space. Each new process gets its own user space.

Ready

A "ready" or "waiting" process has been loaded into main memory and is awaiting execution on a CPU (to be context switched onto the CPU by the dispatcher, or short-term scheduler). There may be many "ready" processes at any one point of the system's

execution—for example, in a one-processor system, only one process can be executing at any one time, and all other "concurrently executing" processes will be waiting for execution.

A ready queue or run queue is used in computer scheduling. Modern computers are capable of running many different programs or processes at the same time. However, the CPU is only capable of handling one process at a time. Processes that are ready for the CPU are kept in a queue for "ready" processes. Other processes that are waiting for an event to occur, such as loading information from a hard drive or waiting on an internet connection, are not in the ready queue.

Running

A process moves into the running state when it is chosen for execution. The process's instructions are executed by one of the CPUs (or cores) of the system. There is at most one running process per CPU or core. A process can run in either of the two modes, namely kernel mode or user mode.

Kernel mode

- Processes in kernel mode can access both: kernel and user addresses.
- Kernel mode allows unrestricted access to hardware including execution of privileged instructions.
- Various instructions (such as I/O instructions and halt instructions) are privileged and can be executed only in kernel mode.
- A system call from a user program leads to a switch to kernel mode.

User mode

- Processes in user mode can access their own instructions and data but not kernel instructions and data (or those of other processes).
- When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.
- User mode avoids various catastrophic failures:
  - There is an isolated virtual address space for each process in user mode.
  - User mode ensures isolated execution of each process so that it does not affect other processes as such.
  - No direct access to any hardware device is allowed.

Blocked

A process transitions to a blocked state when it is waiting for some event, such as a resource becoming available or the completion of an I/O operation. In a multitasking computer system, individual processes, must share the resources of the system. Shared resources include: the CPU, network and network interfaces, memory and disk.

For example, a process may block on a call to an I/O device such as a printer, if the printer is not available. Processes also commonly block when they require user input, or require access to a critical section which must be executed atomically.

Suspend ready

Process that was initially in the ready state but were swapped out of main memory(refer Virtual Memory topic) and placed onto external storage by scheduler are said to be in suspend ready state. The process will transition back to ready state whenever the process is again brought onto the main memory.

Suspend wait or suspend blocked

Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory.
When work is finished it may go to suspend ready.

Terminated

A process may be terminated, either from the "running" state by completing its execution or by explicitly being killed. In either of these cases, the process moves to the "terminated" state. The underlying program is no longer executing, but the process remains in the process table as a zombie process until its parent process calls the wait system call to read its exit status, at which point the

process is removed from the process table, finally ending the process's lifetime. If the parent fails to call wait, this continues to consume the process table entry (concretely the process identifier or PID), and causes a resource leak.

There are many reasons for process termination:

- Batch job issues halt instruction
- User logs off
- Process executes a service request to terminate
- Error and fault conditions
- Normal completion
- Time limit exceeded
- Memory unavailable
- Bounds violation; for example: attempted access of (non-existent) 11th element of a 10-element array
- Protection error; for example: attempted write to read-only file
- Arithmetic error; for example: attempted division by zero
- Time overrun; for example: process waited longer than a specified maximum for an event
- I/O failure
- Invalid instruction; for example: when a process tries to execute data (text)
- Privileged instruction
- Data misuse
- Operating system intervention; for example: to resolve a deadlock
- Parent terminates so child processes terminate (cascading termination)
- Parent request

Adapted from:
"States of a Process in Operating Systems" by Aniket_Dusey, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 8.4: Process States is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

8.5: Process Description

System Processes / System Resources

In a multiprocessing system, there are numerous processes competing to the system's resources. As each process takes its turn executing in the processor, the state of the other processes must be kept in the state that they were interrupted at so as to restore the next process to execute.

Processes run, make use of I/O resources, consume memory. At times processes block waiting for I/O, allowing other processes to run on the processors. Some processes are swapped out in order to make room in physical memory for other processes' needs. P_1 is running, accessing I/O and memory. P_2 is blocked waiting for P1 to complete I/O. P_n is swapped out waiting to return to main memory and further processing.

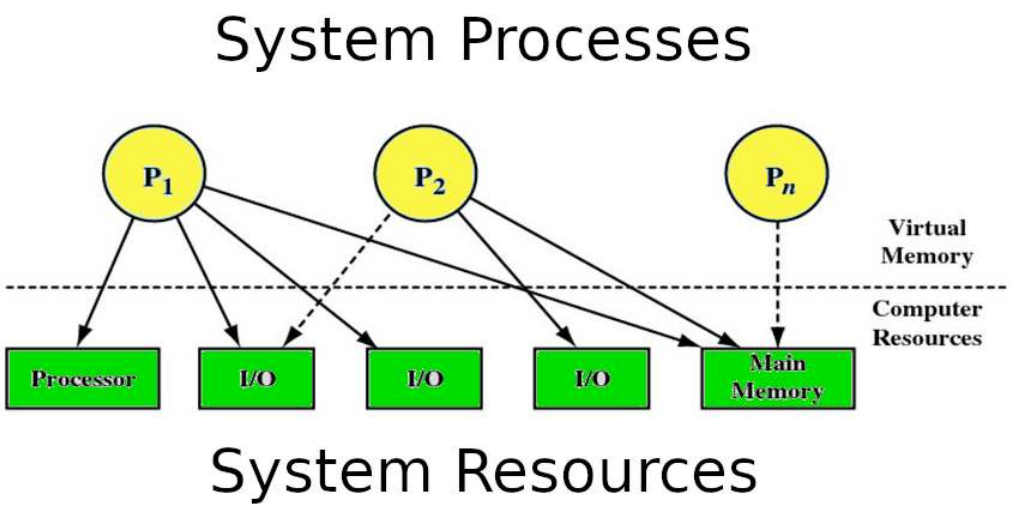


Figure 8.5.1: Processes and Resources. (Unknown source)

Process description and control

As the operating system manages processes and resources, it must maintain information about the current status of each process and the resources in use. The approach to maintaining this information is for the operating system to construct and maintain various tables of information about each entity to be managed.

Operating system maintains four internal components: 1) memory; 2) devices; 3) files; and 4) processes. Details differ from one operating system to another, but all operating systems maintain information in these four categories.

Memory tables: Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address. Interaction is achieved through a sequence of reads or writes of specific memory address. The CPU fetches from and stores in memory.

In order to improve both the utilization of CPU and the speed of the computer's response to its users, several processes must be kept in memory. There are many different algorithms depends on the particular situation. Selection of a memory management scheme for a specific system depends upon many factor, but especially upon the hardware design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management.

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and deallocate memory space as needed.

We will cover memory management in a later section.

Device tables: One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in Linux, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O system. The I/O system consists of:

- A buffer caching system
- A general device driver code
- Drivers for specific hardware devices.

Only the device driver knows the peculiarities of a specific device. We will cover the details later in this course

File tables: File management is one of the most visible services of an operating system. Computers can store information in several different physical forms; disk - both magnetic disks and newer SSD devices, various USB devices, and to the cloud. Each of these devices has it own characteristics and physical organization.

For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped, by the operating system, onto physical devices.

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general a files is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. It is a very general concept.

The operating system implements the abstract concept of the file by managing mass storage device, such as types and disks. Also files are normally organized into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed.

The operating system is responsible for the following activities in connection with file management:

- The creation and deletion of files
- The creation and deletion of directory
- The support of primitives for manipulating files and directories
- The mapping of files onto disk storage.
- Backup of files on stable (non volatile) storage.

This portion of the operating system will also be dealt with later.

Processes: The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general.

In general, a process will need certain resources such as CPU time, memory, files, I/O devices, etc., to accomplish its task. These resources are given to the process when it is created. In addition to the various physical and logical resources that a process obtains when its is created, some initialization data (input) may be passed along. For example, a process whose function is to display on the screen of a terminal the status of a file, say F1, will get as an input the name of the file F1 and execute the appropriate program to obtain the desired information.

We emphasize that a program by itself is not a process; a program is a passive entity, while a process is an active entity. It is known that two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating system processes, those that execute system code, and the rest being user processes, those that execute user code. All of those processes can potentially execute concurrently.

The operating system is responsible for the following activities in connection with processes managed.

- The creation and deletion of both user and system processes
- The suspension are resumption of processes.
- The provision of mechanisms for process synchronization
- The provision of mechanisms for deadlock handling

This page titled 8.5: Process Description is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

8.6: Process Control

Process Table and Process Control Block (PCB)

While creating a process the operating system performs several operations. To identify the processes, it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc. All these information is required and must be saved when the process is switched from one state to another. When the process makes a transition from one state to another, the operating system must update information in the process's PCB.

A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCB's, that means logically contains a PCB for all of the current processes in the system.

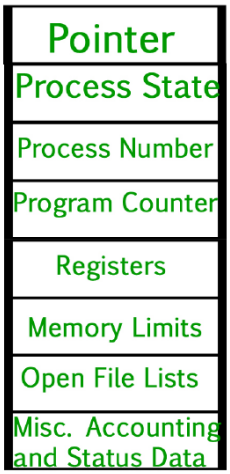


Figure 8.6.1: Process Control Block. ("Process Control Block" by magbene, Geeks for Geeks is licensed under CC BY-SA 4.0)

- **Process scheduling state** –The state of the process in terms of "ready", "suspended", etc., and other scheduling information as well, such as priority value, the amount of time elapsed since the process gained control of the CPU or since it was suspended. Also, in case of a suspended process, event identification data must be recorded for the event the process is waiting for.
- **Process structuring information** – the process's children id's, or the id's of other processes related to the current one in some functional way, which may be represented as a queue, a ring or other data structures
- **Pointer –** It is a stack pointer which is required to be saved when the process is switched from one state to another to retain the current position of the process.
- **Process number –** Every process is assigned with a unique id known as process ID or PID which stores the process identifier.
- **Program counter –** It stores the counter which contains the address of the next instruction that is to be executed for the process.
- **Register –** These are the CPU registers which includes: accumulator, base, registers and general purpose registers.
- **Memory Management Information –** This field contains the information about memory management system used by operating system. This may include the page tables, segment tables etc.
- **Open files list –** This information includes the list of files opened for a process.
- **Interprocess communication information** – flags, signals and messages associated with the communication among independent processes
- **Process Privileges** – allowed/disallowed access to system resources
- **Process State** – new, ready, running, waiting, dead
- **Process Number (PID)** – unique identification number for each process (also known as Process ID)
- **Program Counter (PC)** – A pointer to the address of the next instruction to be executed for this process
- **CPU Scheduling Information** – information scheduling CPU time
- **Accounting Information** – amount of CPU used for process execution, time limits, execution ID etc.

- **I/O Status Information** – list of I/O devices allocated to the process.

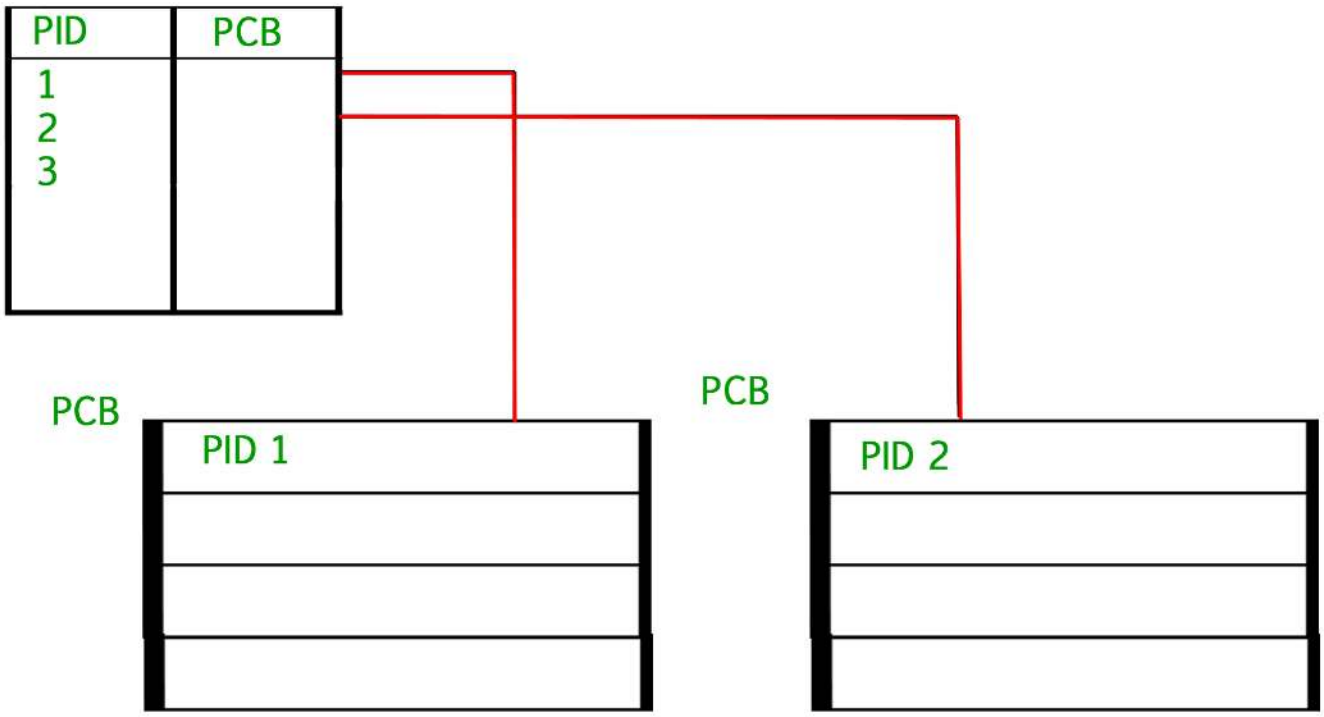


Figure 8.6.1: Process Table and Process Control Block. ("Process Table and Process Control Block" by magbene, Geeks for Geeks is licensed under CC BY-SA 4.0)

Adapted from:

"Process Table and Process Control Block (PCB)" by magbene, Geeks for Geeks is licensed under CC BY-SA 4.0

"Process control block" by ultiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 8.6: Process Control is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

8.7: Fork and Exec

Fork and Exec

New processes are created by the two related interfaces `fork` and `exec` .

Fork

When you come to metaphorical "fork in the road" you generally have two options to take, and your decision effects your future. Computer programs reach this fork in the road when they hit the `fork()` system call.

At this point, the operating system will create a new process that is exactly the same as the parent process. This means all the state that was talked about previously is copied, including open files, register state and all memory allocations, which includes the program code.

The return value from the system call is the only way the process can determine if it was the existing process or a new one. The return value to the parent process will be the Process ID (PID) of the child, whilst the child will get a return value of 0.

At this point, we say the process has `forked` and we have the parent-child relationship as described above.

Exec

Forking provides a way for an existing process to start a new one, but what about the case where the new process is not part of the same program as parent process? This is the case in the shell; when a user starts a command it needs to run in a new process, but it is unrelated to the shell.

This is where the `exec` system call comes into play. `exec` will *replace* the contents of the currently running process with the information from a program binary.

Thus the process the shell follows when launching a new program is to firstly `fork` , creating a new process, and then `exec` (i.e. load into memory and execute) the program binary it is supposed to run.

How Linux actually handles fork and exec

`clone`

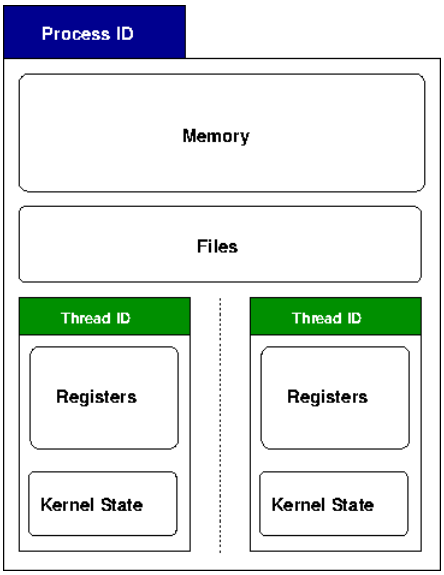
In the kernel, fork is actually implemented by a `clone` system call. This `clone` interfaces effectively provides a level of abstraction in how the Linux kernel can create processes.

`clone` allows you to explicitly specify which parts of the new process are copied into the new process, and which parts are shared between the two processes. This may seem a bit strange at first, but allows us to easily implement *threads* with one very simple interface.

Threads

While `fork` copies all of the attributes we mentioned above, imagine if everything was copied for the new process *except* for the memory. This means the parent and child share the same memory, which includes program code and data.

Figure 5.4. Threads



This hybrid child is called a *thread*. Threads have a number of advantages over where you might use *fork*

1. Separate processes can not see each others memory. They can only communicate with each other via other system calls.

   Threads however, share the same memory. So you have the advantage of multiple processes, with the expense of having to use system calls to communicate between them.

   The problem that this raises is that threads can very easily step on each others toes. One thread might increment a variable, and another may decrease it without informing the first thread. These type of problems are called *concurrency problems* and they are many and varied.

   To help with this, there are userspace libraries that help programmers work with threads properly. The most common one is called `POSIX threads` or, as it more commonly referred to `pthreads`

2. Switching processes is quite expensive, and one of the major expenses is keeping track of what memory each process is using. By sharing the memory this overhead is avoided and performance can be significantly increased.

There are many different ways to implement threads. On the one hand, a userspace implementation could implement threads within a process without the kernel having any idea about it. The threads all look like they are running in a single process to the kernel.

This is suboptimal mainly because the kernel is being withheld information about what is running in the system. It is the kernels job to make sure that the system resources are utilised in the best way possible, and if what the kernel thinks is a single process is actually running multiple threads it may make suboptimal decisions.

Thus the other method is that the kernel has full knowledge of the thread. Under Linux, this is established by making all processes able to share resources via the `clone` system call. Each thread still has associated kernel resources, so the kernel can take it into account when doing resource allocations.

Other operating systems have a hybrid method, where some threads can be specified to run in userspace only ("hidden" from the kernel) and others might be a *light weight process*, a similar indication to the kernel that the processes is part of a thread group.

Copy on write

As we mentioned, copying the entire memory of one process to another when `fork` is called is an expensive operation.

One optimisation is called *copy on write*. This means that similar to threads above, the memory is actually shared, rather than copied, between the two processes when fork is called. If the processes are only going to be reading the memory, then actually copying the data is unnecessary.

However, when a process writes to its memory, it needs to be a private copy that is not shared. As the name suggests, copy on write optimises this by only doing the actual copy of the memory at the point when it is written to.

Copy on write also has a big advantage for `exec`. Since `exec` will simply be overwriting all the memory with the new program, actually copying the memory would waste a lot of time. Copy on write saves us actually doing the copy.

The init process

We discussed the overall goal of the init process previously, and we are now in a position to understand how it works.

On boot the kernel starts the init process, which then forks and execs the systems boot scripts. These fork and exec more programs, eventually ending up forking a login process.

The other job of the `init` process is "reaping". When a process calls `exit` with a return code, the parent usually wants to check this code to see if the child exited correctly or not.

However, this exit code is part of the process which has just called `exit`. So the process is "dead" (e.g. not running) but still needs to stay around until the return code is collected. A process in this state is called a *zombie* (the traits of which you can contrast with a mystical zombie!)

A process stays as a zombie until the parent collects the return code with the `wait` call. However, if the parent exits before collecting this return code, the zombie process is still around, waiting aimlessly to give its status to someone.

In this case, the zombie child will be *reparented* to the init process which has a special handler that *reaps* the return value. Thus the process is finally free and the descriptor can be removed from the kernels process table.

Zombie example

Example 5.3. Zombie example process

```
$ cat zombie.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        pid_t pid;

        printf("parent : %d\n", getpid());

        pid = fork();

        if (pid == 0) {
                printf("child : %d\n", getpid());
                sleep(2);
                printf("child exit\n");
                exit(1);
        }

        /* in parent */
        while (1)
        {
                sleep(1);
        }
}

ianw@lime:~$ ps ax | grep [z]ombie
```

```
    16168 pts/9    S      0:00 ./zombie
    16169 pts/9    Z      0:00 [zombie] <defunct>
```

Above we create a zombie process. The parent process will sleep forever, whilst the child will exit after a few seconds.

Below the code you can see the results of running the program. The parent process (16168) is in state `S` for sleep (as we expect) and the child is in state `Z` for zombie. The ps output also tells us that the process is `defunct` in the process description.[16]

---

[16] The square brackets around the "z" of "zombie" are a little trick to remove the grep processes itself from the ps output. grep interprets everything between the square brackets as a character class, but because the process name will be "grep [z]ombie" (with the brackets) this will not match!

This page titled 8.7: Fork and Exec is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

8.8: Scheduling

Scheduling

A running system has many processes, maybe even into the hundreds or thousands. The part of the kernel that keeps track of all these processes is called the *scheduler* because it schedules which process should be run next.

Scheduling algorithms are many and varied. Most users have different goals relating to what they want their computer to do, so this affects scheduling decisions. For example, for a desktop PC you want to make sure that your graphical applications for your desktop are given plenty of time to run, even if system processes take a little longer. This will increase the responsiveness the user feels, as their actions will have more immediate responses. For a server, you might want your web server application to be given priority.

People are always coming up with new algorithms, and you can probably think of your own fairly easily. But there are a number of different components of scheduling.

Preemptive v co-operative scheduling

Scheduling strategies can broadly fall into two categories

1. *Co-operative* scheduling is where the currently running process voluntarily gives up executing to allow another process to run. The obvious disadvantage of this is that the process may decide to never give up execution, probably because of a bug causing some form of infinite loop, and consequently nothing else can ever run.
2. *Preemptive* scheduling is where the process is interrupted to stop it to allow another process to run. Each process gets a *time-slice* to run in; at the point of each context switch a timer will be reset and will deliver and interrupt when the time-slice is over.

   We know that the hardware handles the interrupt independently of the running process, and so at this point control will return to the operating system. At this point, the scheduler can decide which process to run next.

   This is the type of scheduling used by all modern operating systems.

Realtime

Some processes need to know exactly how long their time-slice will be, and how long it will be before they get another time-slice to run. Say you have a system running a heart-lung machine; you don't want the next pulse to be delayed because something else decided to run in the system!

Hard realtime systems make guarantees about scheduling decisions like the maximum amount of time a process will be interrupted before it can run again. They are often used in life critical applications like medical, aircraft and military applications.

Soft realtime is a variation on this, where guarantees aren't as strict but general system behaviour is predictable. Linux can be used like this, and it is often used in systems dealing with audio and video. If you are recording an audio stream, you don't want to be interrupted for long periods of time as you will loose audio data which can not be retrieved.

Nice value

UNIX systems assign each process a *nice* value. The scheduler looks at the nice value and can give priority to those processes that have a higher "niceness".

A brief look at the Linux Scheduler

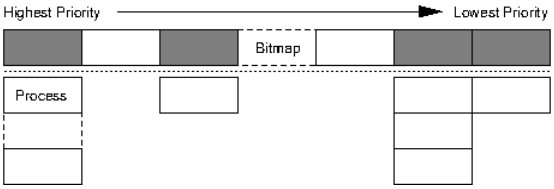
The Linux scheduler has and is constantly undergoing many changes as new developers attempt to improve its behaviour.

The current scheduler is known as the O(1) scheduler, which refers to the property that no matter how many processes the scheduler has to choose from, it will choose the next one to run in a constant amount of time[17].

Previous incarnations of the Linux scheduler used the concept of *goodness* to determine which process to run next. All possible tasks are kept on a *run queue*, which is simply a linked list of processes which the kernel knows are in a "runnable" state (i.e. not waiting on disk activity or otherwise asleep). The problem arises that to calculate the next process to run, every possible runnable

process must have its goodness calculated and the one with the highest goodness ``wins''. You can see that for more tasks, it will take longer and longer to decide which processes will run next.

Figure 5.5. The O(1) scheduler



In contrast, the O(1) scheduler uses a run queue structure as shown above. The run queue has a number of *buckets* in priority order and a bitmap that flags which buckets have processes available. Finding the next process to run is a matter of reading the bitmap to find the first bucket with processes, then picking the first process off that bucket's queue. The scheduler keeps two such structures, an *active* and *expired* array for processes that are runnable and those which have utilised their entire time slice respectively. These can be swapped by simply modifying pointers when all processes have had some CPU time.

The really interesting part, however, is how it is decided where in the run queue a process should go. Some of the things that need to be taken into account are the nice level, processor affinity (keeping processes tied to the processor they are running on, since moving a process to another CPU in a SMP system can be an expensive operation) and better support for identifying interactive programs (applications such as a GUI which may spend much time sleeping, waiting for user input, but when the user *does* get around to interacting with it wants a fast response).

---

[17] *Big-O* notation is a way of describing how long an algorithm takes to run given increasing inputs. If the algorithm takes twice as long to run for twice as much input, this is increasing linearly. If another algorithm takes four times as long to run given twice as much input, then it is increasing exponentially. Finally if it takes the same amount of time now matter how much input, then the algorithm runs in constant time. Intuitively you can see that the slower the algorithm grows for more input, the better it is. Computer science text books deal with algorithm analysis in more detail.

This page titled 8.8: Scheduling is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

8.9: Execution within the Operating System

There are some concepts we need to clarify as we discuss the concept of execution within the operating system.

What is the kernel

The kernel is a computer program at the core of a computer's operating system that has complete control over everything in the system. It is the "portion of the operating system code that is always resident in memory", and facilitates interactions between hardware and software components. On most systems, the kernel is one of the first programs loaded on startup (after the bootloader). It handles the rest of startup as well as memory, peripherals, and input/output (I/O) requests from software, translating them into data-processing instructions for the central processing unit.

Introduction to System Call

In computing, a **system call** when a program program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to interact with the operating system. Application programs are NOT allowed to perform certain tasks, such as open a file, or create a new process. System calls provide the services of the operating system to the application programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes, that is the applications that users are running on the system, to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

Services Provided by System Calls :

1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

Types of System Calls : There are 5 different categories of system calls

1. Process control: end, abort, create, terminate, allocate and free memory.
2. File management: create, open, close, delete, read file etc.
3. Device management
4. Information maintenance
5. Communication

The following are some examples of system calls in Windows and Linux. So, if a user is running a word processing tool, and wants to save the document - the word processor asks the operating system to create a file, or open a file, to save the current set of changes. If the application has permission to write to the requested file then the operating system performs the task. Otherwise, the operating system returns a status telling the user they do not have permission to write to the requested file. This concept of user versus kernel allows the operating system to maintain a certain level of control.

	Windows	Linux
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	

Adapted from:
"Kernel (operating system)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Introduction of System Call" by Samit Mandal, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 8.9: Execution within the Operating System is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

8.9.1: Execution within the Operating System - Dual Mode

Dual Mode Operations in an Operating System

An error in one program can adversely affect many processes, it might modify data of another program, or also can affect the operating system. For example, if a process stuck in infinite loop then this infinite loop could affect correct operation of other processes. So to ensure the proper execution of the operating system, there are two modes of operation:

User mode –
When the computer system run user applications like creating a text document or using any application program, then the system is in the user mode. When the user application requests for a service from the operating system or an interrupt occurs or **system call**, then there will be a transition from user to kernel mode to fulfill the requests.

Given below image describes what happen when an interrupt occurs: (do not worry about the comments about the mode bit)

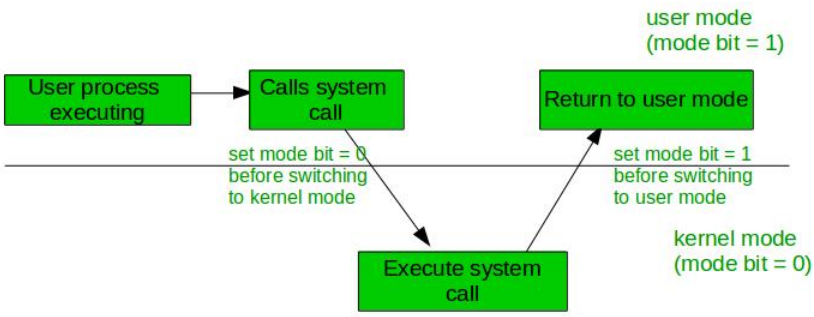


Figure 8.9.1.1: User Process Makes System Call. ("User Mode to Kernel Mode Switch" by shivani.mittal, Geeks for Geeks is licensed under CC BY-SA 4.0)

Kernel Mode –
While the system is running the certain processes operate in kernel mode because the processes needs access to operating system calls. This provides protection by controlling which processes can access kernel mode operations. As shown in the diagram above, the system will allow certain user mode processes to execute system calls by allowing the process to temporarily run in kernel mode. While in kernel mode the process is allowed to have direct access to all hardware and memory in the system (also called privileged mode). If a user process attempts to run privileged instructions in user mode then it will treat instruction as illegal and traps to OS. Some of the privileged instructions are:

1. Handling system interrupts
2. To switch from user mode to kernel mode.
3. Management of I/O devices.

User Mode and Kernel Mode Switching

In it's life span a process executes in user mode **AND** kernel mode. The user mode is normal mode where the process has limited access. While the kernel mode is the privileged mode where the process has unrestricted access to system resources like hardware, memory, etc. A process can access services like hardware I/O by executing accessing kernel data in kernel mode. Anything related to process management, I/O hardware management, and memory management requires process to execute in Kernel mode.

The kernel provides System Call Interface (SCI), which are entry points for user processes to enter kernel mode. System calls are the only way through which a process can go into kernel mode from user mode. Below diagram explains user mode to kernel mode transition in detail.

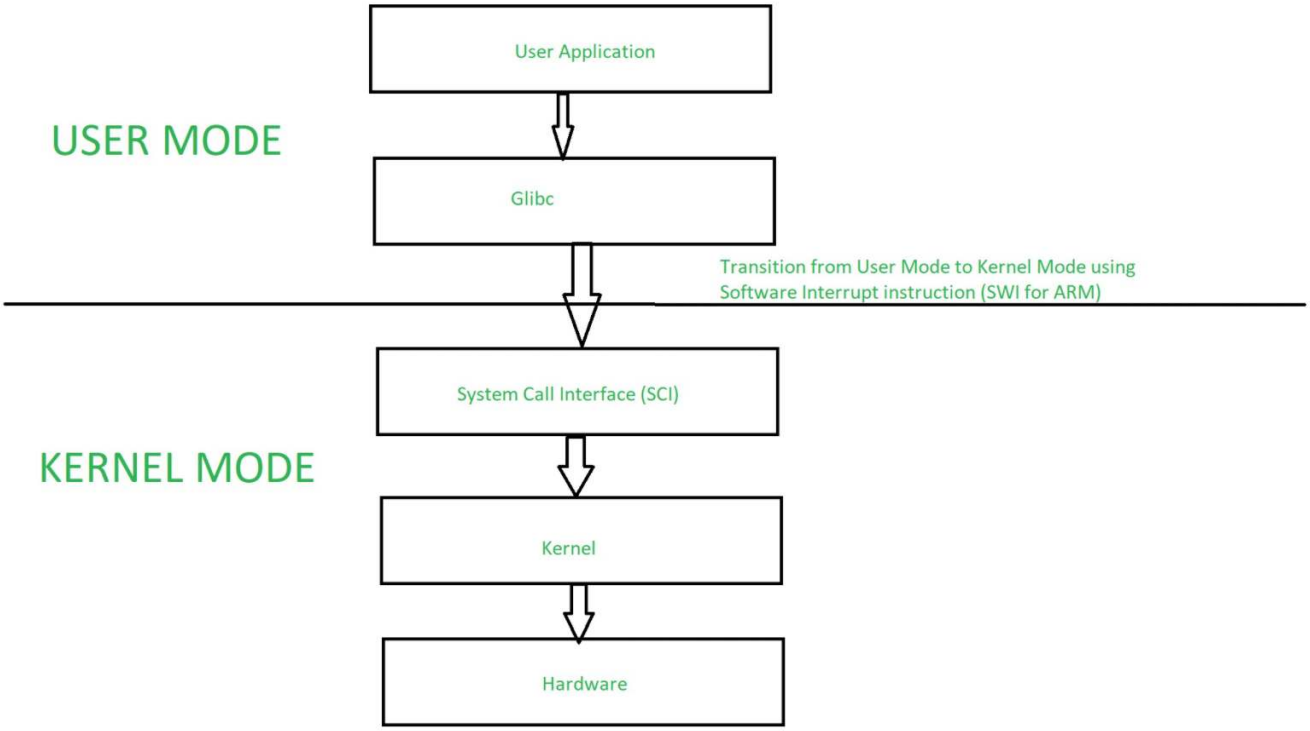


Figure 8.9.1.1: User Mode to Kernel Mode Transition. ("user mode to kernel mode transition" by sandeepjainlinux, Geeks for Geeks is licensed under CC BY-SA 4.0)

When in user mode, the application process makes a call to Glibc, which is a library used by software programmers. This call alerts the operating system kernel that the application desires to do something that only the kernel has the privilege to do. The operating system/kernel will check to ensure that this application process has the proper authority to perform the requested action. If it does have the necessary permission - the operating system allows the operation to proceed, otherwise an error message is sent to the user.

Why?

You may be wondering why do operating system designers go to all of this trouble of creating dual modes. Why not just allow everything to operate in kernel mode and save the over head of all this switching?

There are 2 main reasons:

1. If everything were to run in a single mode we end up with the issue that Microsoft had in the earlier versions of Windows. If a process were able to exploit a vulnerability that process then had the ability to control the system.
2. There are certain conditions known as a trap, also known as an exception or a system fault, is typically caused by an exceptional condition such as division by zero, invalid memory access etc. If the process is running in kernel mode such a trap situation can crash the entire operating system. A process in user mode that encounters a trap situation it only crashes the user mode process.

So, the overhead of switching is acceptable to ensure a more stable, secure system.

Adapted from:
"Dual Mode operations in OS" by shivani.mittal, Geeks for Geeks is licensed under CC BY-SA 4.0
"User mode and Kernel mode Switching" by sandeepjainlinux, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 8.9.1: Execution within the Operating System - Dual Mode is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

8.10: The Shell

The Shell

On a UNIX system, the shell is the standard interface to handling processes on your system. Once the shell was the primary interface, however modern Linux systems have a GUI and provide a shell via a "terminal application" or similar. The primary job of the shell is to help the user handle starting, stopping and otherwise controlling processes running in the system.

When you type a command at the prompt of the shell, it will `fork` a copy of itself and `exec` the command that you have specified.

The shell then, by default, waits for that process to finish running before returning to a prompt to start the whole process over again.

As an enhancement, the shell also allows you to *background* a job, usually by placing an `&` after the command name. This is simply a signal that the shell should fork and execute the command, but *not* wait for the command to complete before showing you the prompt again.

The new process runs in the background, and the shell is ready waiting to start a new process should you desire. You can usually tell the shell to *foreground* a process, which means we do actually want to wait for it to finish.

XXX : a bit of history about bourne shell

This page titled 8.10: The Shell is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

8.11: Signals

Signals

Processes running in the system require a way to be told about events that influence them. On UNIX there is infrastructure between the kernel and processes called *signals* which allows a process to receive notification about events important to it.

When a signal is sent to a process, the kernel invokes a *handler* which the process must register with the kernel to deal with that signal. A handler is simply a designed function in the code that has been written to specifically deal with interrupt. Often the signal will be sent from inside the kernel itself, however it is also common for one process to send a signal to another process (one form of *interprocess communication*). The signal handler gets called *asynchronously*; that is the currently running program is interrupted from what it is doing to process the signal event.

For example, one type of signal is an *interrupt* (defined in system headers as `SIGINT`) is delivered to the process when the `ctrl-c` combination is pressed.

As a process uses the `read` system call to read input from the keyboard, the kernel will be watching the input stream looking for special characters. Should it see a `ctrl-c` it will jump into signal handling mode. The kernel will look to see if the process has registered a handler for this interrupt. If it has, then execution will be passed to that function where the function will *handle* it. Should the process have not registered a handler for this particular signal, then the kernel will take some default action. With `ctrl-c` the default action is to terminate the process.

A process can choose to ignore some signals, but other signals are not allowed to be ignored. For example, `SIGKILL` is the signal sent when a process should be terminated. The kernel will see that the process has been sent this signal and terminate the process from running, no questions asked. The process can not ask the kernel to ignore this signal, and the kernel is very careful about which process is allowed to send this signal to another process; you may only send it to processes owned by you unless you are the root user. You may have seen the command `kill -9` ; this comes from the implementation `SIGKILL` signal. It is commonly known that `SIGKILL` is actually defined to be `0x9` , and so when specified as an argument to the `kill` program means that the process specified is going to be stopped immediately. Since the process can not choose to ignore or handle this signal, it is seen as an avenue of last resort, since the program will have no chance to clean up or exit cleanly. It is considered better to first send a `SIGTERM` (for terminate) to the process first, and if it has crashed or otherwise will not exit then resort to the `SIGKILL` . As a matter of convention, most programs will install a handler for `SIGHUP` (hangup -- a left over from days of serial terminals and modems) which will reload the program, perhaps to pick up changes in a configuration file or similar.

If you have programmed on a Unix system you would be familiar with `segmentation faults` when you try to read or write to memory that has not been allocated to you. When the kernel notices that you are touching memory outside your allocation, it will send you the segmentation fault signal. Usually the process will not have a handler installed for this, and so the default action to terminate the program ensues (hence your program "crashes"). In some cases a program may install a handler for segmentation faults, although reasons for doing this are limited.

This raises the question of what happens after the signal is received. Once the signal handler has finished running, control is returned to the process which continues on from where it left off.

Example

The following simple program introduces a lot of signals to run!

Example 5.4. Signals Example

```
$ cat signal.c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void sigint_handler(int signum)
```

```
{
        printf("got SIGINT\n");
}

int main(void)
{
        signal(SIGINT, sigint_handler);
        printf("pid is %d\n", getpid());
        while (1)
                sleep(1);
}
$ gcc -Wall -o signal signal.c
$ ./signal
pid is 2859
got SIGINT # press ctrl-c
           # press ctrl-z
[1]+  Stopped                 ./signal

$ kill -SIGINT 2859
$ fg
./signal
got SIGINT
Quit # press ctrl-\

$
```

We have simple program that simply defines a handler for the `SIGINT` signal, which is sent when the user presses `ctrl-c`. All the signals for the system are defined in `signal.h`, including the `signal` function which allows us to register the handling function.

The program simply sits in a tight loop doing nothing until it quits. When we start the program, we try pressing `ctrl-c` to make it quit. Rather than taking the default action, or handler is invoked and we get the output as expected.

We then press `ctrl-z` which sends a `SIGSTOP` which by default puts the process to sleep. This means it is not put in the queue for the scheduler to run and is thus dormant in the system.

As an illustration, we use the kill program to send the same signal from another terminal window. This is actually implemented with the `kill` system call, which takes a signal and PID to send to (this function is a little misnamed because not all signals do actually kill the process, as we are seeing, but the `signal` function was already taken to register the handler). As the process is stopped, the signal gets *queued* for the process. This means the kernel takes note of the signal and will deliver it when appropriate.

At this point we wake the process up by using the command `fg`. This actually sends a `SIGCONT` signal to the process, which by default will wake the process back up. The kernel knows to put the process on the run queue and give it CPU time again. We see at this point the queued signal is delivered.

In desperation to get rid of the program, we finally try `ctrl-\` which sends a `SIGQUIT` (abort) to the process. But if the process has aborted, where did the `Quit` output come from?

You guessed it, more signals! When a parent child has a process that dies, it gets a `SIGCHLD` signal back. In this case the shell was the parent process and so it got the signal. Remember how we have the zombie process that needs to be reaped with the `wait` call to get the return code from the child process? Well another thing it also gives the parent is the signal number that the

child may have died from. Thus the shell knows that child process died from a `SIGABRT` and as an informational service prints as much for the user (the same process happens to print out "Segmentation Fault" when the child process dies from a `SIGSEGV`).

You can see how in even a simple program, around 5 different signals were used to communicate between processes and the kernel and keep things running. There are many other signals, but these are certainly amongst the most common. Most have system functions defined by the kernel, but there are a few signals reserved for users to use for their own purposes within their programs (`SIGUSR`).

This page titled 8.11: Signals is shared under a CC BY-SA 3.0 license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

9: Threads

9.1: Process and Threads

9.2: Thread Types

9.2.1: Thread Types - Models

9.3: Thread Relationships

9.4: Benefits of Multithreading

This page titled 9: Threads is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

9.1: Process and Threads

Definition

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its dynamically allocated variables and non-thread-local global variables at any given time.

Difference between Process and Thread

Process:
Process means any program is in execution. Process control block controls the operation of any process. Process control block contains information about processes for example Process priority, process id, process state, CPU, register, etc. A process can creates other processes which are known as **Child Processes**. Process takes more time to terminate and it is isolated means it does not share memory with any other process.

The process can have the following states like new, ready, running, waiting, terminated, suspended.

Thread:
Thread is the segment of a process means a process can have multiple threads and these multiple threads are contained within a process. A thread have 3 states: running, ready, and blocked.

Thread takes less time to terminate as compared to process and like process threads do not isolate.

Difference between Process and Thread:

	Process	Thread
1.	Process means any program is in execution.	Thread means segment of a process.
2.	Process takes more time to terminate.	Thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	Process is less efficient in term of communication.	Thread is more efficient in term of communication.
6.	Process consume more resources.	Thread consume less resources.
7.	Process is isolated.	Threads share memory.
8.	Process is called heavy weight process.	Thread is called light weight process.
9.	Process switching uses interface in operating system.	Thread switching does not require to call a operating system and cause an interrupt to the kernel.
10.	If one server process is blocked no other server process can execute until the first process unblocked.	Second thread in the same task could run, while one server thread is blocked.
11.	Process has its own Process Control Block, Stack and Address Space.	Thread has Parents' PCB, its own Thread Control Block and Stack and common Address space.

Threads in Operating Systems

What is a Thread?
A thread is a path of execution within a process. A process can contain multiple threads.

Why Multithreading?

A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc. More advantages of multithreading are discussed below

Process vs Thread?

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.

Threads vs. processes pros and cons

Threads differ from traditional multitasking operating-system processes in several ways:

- processes are typically independent, while threads exist as subsets of a process
- processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
- processes have separate address spaces, whereas threads share their address space
- processes interact only through system-provided inter-process communication m`1echanisms
- context switching between threads in the same process typically occurs faster than context switching between processes

Systems such as Windows NT and OS/2 are said to have cheap threads and expensive processes; in other operating systems there is not so great a difference except in the cost of an address-space switch, which on some architectures (notably x86) results in a translation lookaside buffer (TLB) flush.

Advantages and disadvantages of threads vs processes include:

- Lower resource consumption of threads: using threads, an application can operate using fewer resources than it would need when using multiple processes.
- Simplified sharing and communication of threads: unlike processes, which require a message passing or shared memory mechanism to perform inter-process communication (IPC), threads can communicate through data, code and files they already share.
- Thread crashes a process: due to threads sharing the same address space, an illegal operation performed by a thread can crash the entire process; therefore, one misbehaving thread can disrupt the processing of all the other threads in the application.

Adapted from:
"Difference between Process and Thread" by MKS075, Geeks for Geeks is licensed under CC BY-SA 4.0
"Thread in Operating System" by chrismaher37, Geeks for Geeks is licensed under CC BY-SA 4.0
"Thread (computing)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 9.1: Process and Threads is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

9.2: Thread Types

Threads and its types in Operating System

Thread is a single sequence stream within a process. Threads have same properties as of the process so they are called as light weight processes. Threads are executed one after another but gives the illusion as if they are executing in parallel. Each thread has different states. Each thread has

1. A program counter
2. A register set
3. A stack space

Threads are not independent of each other as they share the code, data, OS resources etc.

Similarity between Threads and Processes –

- Only one thread or process is active at a time
- Within process both execute sequentiall
- Both can create children

Differences between Threads and Processes –

- Threads are not independent, processes are.
- Threads are designed to assist each other, processes may or may not do it

Types of Threads:

1. **User Level thread (ULT)**
   Is implemented in the user level library, they are not created using the system calls. Thread switching does not need to call OS and to cause interrupt to Kernel. Kernel doesn't know about the user level thread and manages them as if they were single-threaded processes.

   Advantages of ULT

   - Can be implemented on an OS that does't support multithreading.
   - Simple representation since thread has only program counter, register set, stack space.
   - Simple to create since no intervention of kernel.
   - Thread switching is fast since no OS calls need to be made.

   Disadvantages of ULT

   - No or less co-ordination among the threads and Kernel.
   - If one thread causes a page fault, the entire process blocks.

   Difference between Process and User Level Thread:

PROCESS	USER LEVEL THREAD
Process is a program being executed.	User level thread is the thread managed at user level.
It is high overhead.	It is low overhead.
There is no sharing between processes.	User level threads share address space.
Process is scheduled by operating system.	User level thread is scheduled by thread library.
Blocking one process does not affect the other processes.	Blocking one user Level thread will block whole process of the thread.
Process is scheduled using process table.	User level thread is scheduled using thread table.
It is heavy weight activity.	It is light weight as compared to process.
It can be suspended.	It can not be suspended.
Suspension of a process does not affect other processes.	Suspension of user level thread leads to all the threads stop running.
Its types are – user process and system process.	Its types are – user level single thread and user level multi thread.
Each process can run on different processor.	All threads should run on only one processor.
Processes are independent from each other.	User level threads are dependent.
Process supports parallelism.	User level threads do not support parallelism.

2. **Kernel Level Thread (KLT)**
   Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads.

   Advantages of KLT

   - Since kernel has full knowledge about the threads in the system, scheduler may decide to give more time to processes having large number of threads.
   - Good for applications that frequently block.

   Disadvantages of KLT

   - Slow and inefficient.
   - It requires thread control block so it is an overhead.

   Difference between Process and Kernel Thread:

PROCESS	KERNEL THREAD
Process is a program being executed.	Kernel thread is the thread managed at kernel level.
It is high overhead.	It is medium overhead.
There is no sharing between processes.	Kernel threads share address space.
Process is scheduled by operating system using process table.	Kernel thread is scheduled by operating system using thread table.
It is heavy weight activity.	It is light weight as compared to process.
It can be suspended.	It can not be suspended.
Suspension of a process does not affect other processes.	Suspension of kernel thread leads to all the threads stop running.
Its types are – user process and system process.	Its types are – kernel level single thread and kernel level multi thread.

This page titled 9.2: Thread Types is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

9.2.1: Thread Types - Models

Multi Threading Models in Process Management

Many operating systems support kernel thread and user thread in a combined way. Example of such system is Solaris. Multi threading model are of three types.

```
Many to many model.
Many to one model.
one to one model.
```

Many to Many Model

In this model, we have multiple user threads connected to the same or lesser number of kernel level threads. The number of kernel level threads are specific to the type of hardware. The advantage of this model is if a user thread is blocked for any reason, we can schedule others user threads to other kernel threads and the process itself continues to execute. Therefore, the entire process doesn't block if a single thread is blocked.

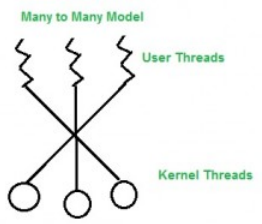


Figure 9.2.1.1: Many to Many Thread Model. ("Many to Many" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0)

Many to One Model

In this model, we have multiple user threads mapped to a single kernel thread. In this model if a user thread gets blocked by a system call, the process itself is blocked. Since we have only one kernel thread then only one user thread can access kernel at a time, so multiple user threads are not able access system calls at the same time.

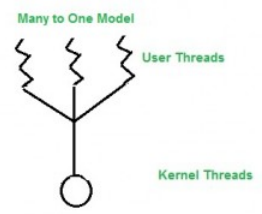


Figure 9.2.1.1: Many to One Thread Model. ("Many to One" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0)

One to One Model

In this model, there is a one to one relationship between kernel and user threads. Multiple thread can run on their own processor in a multiprocessor system. The problem with this model is that creating a user thread requires the creation of a kernel thread.

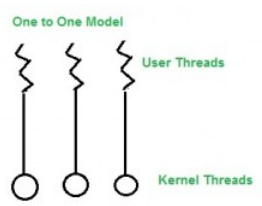


Figure 9.2.1.1: One to One Thread Model. ("One to One" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0)

Adapted from:
"Multi Threading Models in Process Management" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 9.2.1: Thread Types - Models is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

9.3: Thread Relationships

Relationship between User level thread and Kernel level thread

A task is accomplished on the execution of a program, which results in a process. Every task incorporates one or many sub tasks, whereas these sub tasks are carried out as functions within a program by the threads. The operating system (kernel) is unaware of the threads in the user space.

There are two types of threads, User level threads (ULT) and Kernel level threads (KLT).

1. **User Level Threads :**
   Threads in the user space designed by the application developer using a thread library to perform unique subtask.
2. **Kernel Level Threads :**
   Threads in the kernel space designed by the os developer to perform unique functions of OS. Similar to a interrupt handler.

There exist a strong a relationship between user level threads and kernel level threads.

Dependencies between ULT and KLT :

1. **Use of Thread Library :**
   Thread library acts as an interface for the application developer to create number of threads (according to the number of subtasks) and to manage those threads. This API for a process can be implemented in kernel space or user space. In real-time application, the necessary thread library is implemented in user space. This reduces the system call to kernel whenever the application is in need of thread creation, scheduling or thread management activities. Thus, the thread creation is faster as it requires only function calls within the process. The user address space for each thread is allocated at run-time. Overall it reduces various interface and architectural overheads as all these functions are independent of kernel support.
2. **Synchronization :**
   The subtasks (functions) within each task (process) can be executed concurrently or in parallel depending on the application. In that case, a single-threaded process is not suitable. These subtaks require multithreaded process. A unique subtask is allocated to every thread within the process. These threads may use the same data section or different data section. Typically, threads within the same process will share the code section, data section, address space, open files etc...BUT...each thread has its own set of registers, and its own stack memory.

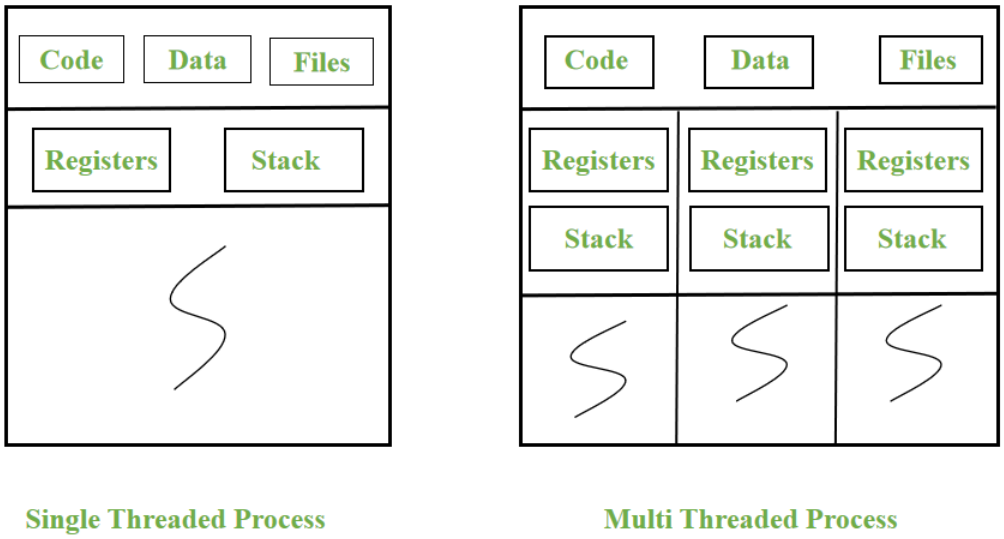


Figure 9.3.1: Single and Multi Thread Processes. ("SIngle versus Multi Threads" by maha93427, Geeks for Geeks is licensed under CC BY-SA 4.0)

When subtasks are concurrently performed by sharing the code section, it may result in data inconsistency. Ultimately, requires suitable synchronization techniques to maintain the control flow to access the shared data.

In a multithreaded process, synchronization has four different models :

1. **Mutex Locks –** This allows only one thread at a time to access the shared resource.
2. **Read/Write Locks –** This allows exclusive writes and concurrent read of a shared resource.
3. **Counting Semaphore –** This count refers to the number of shared resource that can be accessed simultaneously at a time. Once the count limit is reached, the remaining threads are blocked.
4. **Conditional Variables –** This blocks the thread until the condition satisfies (Busy Waiting).
   All these synchronization models are carried out within each process using thread library. The memory space for the lock variables is allocated in the user address space. Thus, requires no kernel intervention.

1. Scheduling :
The application developer during the thread creation sets the priority and scheduling policy of each ULT thread using the thread library. On the execution of program, based on the defined attributes the scheduling takes place by the thread library. In this case, the system scheduler has no control over thread scheduling as the kernel is unaware of the ULT threads.

2. Context Switching :
Switching from one ULT thread to other ULT thread is faster within the same process, as each thread has its own unique thread control block, registers, stack. Thus, registers are saved and restored. Does not require any change of address space. Entire switching takes place within the user address space under the control of thread library.

3. Asynchronous I/O :
After an I/O request ULT threads remains in blocked state, until it receives the acknowledgment(ack) from the receiver. Although it follows asynchronous I/O, it creates a synchronous environment to the application user. This is because the thread library itself schedules an other ULT to execute until the blocked thread sends *sigpoll* as an ack to the process thread library. Only then the thread library, reschedules the blocked thread.For example, consider a program to copy the content(read) from one file and to paste(write) in the other file. Additionally, a pop-up that displays the percentage of progress completion.

Dependency between ULT and KLT :
The one and only major dependency between KLT and ULT occurs when an ULT is in need of the **Kernel resources**. Every ULT thread is associated to a virtual processor called a Light-weight process. This is created and bound to the ULT by the thread library. Whenever a system call is invoked, a kernel level thread is created and scheduled by the system scheduler. These KLT are scheduled to access the kernel resources by the system scheduler - the scheduler is unaware of the ULT. Whereas the KLT themselves are aware of each ULT associated with each KLT.

What if the relationship does not exist ?
If there is no association between KLT and ULT, then every process is a single-threaded process.

Adapted from:
"Relationship between User level thread and Kernel level thread" by maha93427, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 9.3: Thread Relationships is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

9.4: Benefits of Multithreading

Benefits of Multithreading in Operating System

The benefits of multi threaded programming can be broken down into four major categories:

1. **Responsiveness –**
   Multithreading in an interactive application may allow a program to continue running even if a part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

   In a non multi threaded environment, a server listens to the port for some request and when the request comes, it processes the request and then resume listening to another request. The time taken while processing of request makes other users wait unnecessarily. Instead a better approach would be to pass the request to a worker thread and continue listening to port.

   For example, a multi threaded web browser allow user interaction in one thread while an video is being loaded in another thread. So instead of waiting for the whole web-page to load the user can continue viewing some portion of the web-page.

2. **Resource Sharing –**
   Processes may share resources only through techniques such as-
   - Message Passing
   - Shared Memory

   Such techniques must be explicitly organized by programmer. However, threads share the memory and the resources of the process to which they belong by default.
   The benefit of sharing code and data is that it allows an application to have several threads of activity within same address space.

3. **Economy –**
   Allocating memory and resources for process creation is a costly job in terms of time and space.
   Since, threads share memory with the process it belongs, it is more economical to create and context switch threads. Generally much more time is consumed in creating and managing processes than in threads.
   In Solaris, for example, creating process is 30 times slower than creating threads and context switching is 5 times slower.
4. **Scalability –**
   The benefits of multi-programming greatly increase in case of multiprocessor architecture, where threads may be running parallel on multiple processors. If there is only one thread then it is not possible to divide the processes into smaller tasks that different processors can perform.
   Single threaded process can run only on one processor regardless of how many processors are available.
   Multi-threading on a multiple CPU machine increases parallelism.

Adapted from:
"Benefits of Multithreading in Operating System" by aastha98, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 9.4: Benefits of Multithreading is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

10: Concurrency and Process Synchronization

10.1: Introduction to Concurrency

10.2: Process Synchronization

10.3: Mutual Exclusion

10.4: Interprocess Communication

10.4.1: IPC - Semaphores

10.4.2: IPC - Monitors

10.4.3: IPC - Message Passing / Shared Memory

This page titled 10: Concurrency and Process Synchronization is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

10.1: Introduction to Concurrency

Concurrency in Operating System

Concurrency is the execution of a set of multiple instruction sequences at the same time. This occurs when there are several process threads running in parallel. These threads communicate with the other threads/processes through a concept of shared memory or through message passing. Because concurrency results in the sharing of system resources - instructions, memory, files - problems can occur. like deadlocks and resources starvation. (we will talk about starvation and deadlocks in the next module).

Principles of Concurrency :
With current technology such as multi core processors, and parallel processing, which allow for multiple processes/threads to be executed concurrently - that is at the same time - it is possible to have more than a single process/thread accessing the same space in memory, the same declared variable in the code, or even attempting to read/write to the same file.

The amount of time it takes for a process to execute is not easily calculated, so we are unable to predict which process will complete first, thereby allowing us to implement algorithms to deal with the issues that concurrency creates. The amount of time a process takes to complete depends on the following:

- The activities of other processes
- The way operating system handles interrupts
- The scheduling policies of the operating system

Problems in Concurrency :

- **Sharing global resources**
  Sharing of global resources safely is difficult. If two processes both make use of a global variable and both make changes to the variables value, then the order in which various changes take place are executed is critical.
- **Optimal allocation of resources**
  It is difficult for the operating system to manage the allocation of resources optimally.
- **Locating programming errors**
  It is very difficult to locate a programming error because reports are usually not reproducible due to the different states of the shared components each time the code runs.
- **Locking the channel**
  It may be inefficient for the operating system to simply lock the resource and prevent its use by other processes.

Advantages of Concurrency :

- **Running of multiple applications**
  Having concurrency allows the operating system to run multiple applications at the same time.
- **Better resource utilization**
  Having concurrency allows the resources that are NOT being used by one application can be used for other applications.
- **Better average response time**
  Without concurrency, each application has to be run to completion before the next one can be run.
- **Better performance**
  Concurrency provides better performance by the operating system. When one application uses only the processor and another application uses only the disk drive then the time to concurrently run both applications to completion will be shorter than the time to run each application consecutively.

Drawbacks of Concurrency :

- When concurrency is used, it is pretty much required to protect multiple processes/threads from one another.
- Concurrency requires the coordination of multiple processes/threads through additional sequences of operations within the operating system.
- Additional performance enhancements are necessary within the operating systems to provide for switching among applications.
- Sometimes running too many applications concurrently leads to severely degraded performance.

Issues of Concurrency :

- **Non-atomic**
  Operations that are non-atomic but interruptible by multiple processes can cause problems. (an atomic operation is one that runs completely independently of any other processes/threads - any process that is dependent on another process/thread is **non-atomic**)
- **Race conditions**
  A race condition is a behavior which occurs in software applications where the output is dependent on the timing or sequence of other uncontrollable events. Race conditions also occur in software which supports multithreading, use a distributed environment or are interdependent on shared resources
- **Blocking**
  A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation.[Processes can block waiting for resources. A process could be blocked for long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable.
- **Starvation**
  A problem encountered in concurrent computing where a process is perpetually denied necessary resources to process its work. Starvation may be caused by errors in a scheduling or mutual exclusion algorithm, but can also be caused by resource leaks
- **Deadlock**
  In concurrent computing, a deadlock is a state in which each member of a group waits for another member, including itself, to take action, such as sending a message or more commonly releasing a lock. Deadlocks are a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to arbitrate shared resources and implement process synchronization

Adapted from:
"Concurrency in Operating System" by pp_pankaj, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 10.1: Introduction to Concurrency is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

10.2: Process Synchronization

Introduction of Process Synchronization

When we discuss the concept of synchronization, processes are categorized as one of the following two types:

1. **Independent Process** : Execution of one process does not affects the execution of other processes.
2. **Cooperative Process** : Execution of one process affects the execution of other processes.

Process synchronization problems are most likely when dealing with cooperative processes because the process resources are shared between the multiple processes/threads.

Race Condition

When more than one processes is executing the same code, accessing the same memory segment or a shared variable there is the possibility that the output or the value of the shared variable is incorrect. This can happen when multiple processes are attempting to alter a memory location, this can create a race condition - where multiple processes have accessed the current value at a memory location, each process has changed that value, and now they need to write the new back...BUT...each process has a different new value. So, which one is correct? Which one is going to be the new value.

Operating system need to have a process to manage these shared components/memory segments. This is called synchronization, and is a critical concept in operating system.

Usually race conditions occur inside what is known as a critical section of the code. Race conditions can be avoided if the critical section is treated as an atomic instruction, that is an operation that run completely independently of any other processes, making use of software locks or atomic variables which will prevent race conditions. We will take a look at this concept below.

Critical Section Problem

In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access. This protected section is the critical section or critical region. It cannot be executed by more than one process at a time. Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that would not operate correctly in the context of multiple concurrent accesses.

Different codes or processes may consist of the same variable or other resources that need to be read or written but whose results depend on the order in which the actions occur. For example, if a variable *x* is to be read by process A, and process B has to write to the same variable *x* at the same time, process A might get either the old or new value of *x*.

The following example is VERY simple - sometimes the critical section can be more than a single line of code.

Process A:

```
// Process A
.
.
b = x + 5;                // instruction executes at time = Tx, meaning some unknown ti
.
```

Process B:

```
// Process B
.
.
```

```
x = 3 + z;                // instruction executes at time = Tx, meaning some unknown ti
.
```

In cases like these, a critical section is important. In the above case, if A needs to read the updated value of *x*, executing Process A and Process B at the same time may not give required results. To prevent this, variable *x* is protected by a critical section. First, B gets the access to the section. Once B finishes writing the value, A gets the access to the critical section and variable *x* can be read.

By carefully controlling which variables are modified inside and outside the critical section, concurrent access to the shared variable are prevented. A critical section is typically used when a multi-threaded program must update multiple related variables without a separate thread making conflicting changes to that data. In a related situation, a critical section may be used to ensure that a shared resource, for example, a printer, can only be accessed by one process at a time.

Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**
  Exclusive access of each process to the shared memory. Only one process can be in it's critical section at any given time.
- **Progress**
  If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.
- **Bounded Waiting**
  After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section. The purpose of this condition is to make sure that every process gets the chance to actually enter its critical section so that no process starves forever.

Adapted from:
"Critical section" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 10.2: Process Synchronization is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

10.3: Mutual Exclusion

Mutual Exclusion Explained

The problem which mutual exclusion addresses is a problem of resource sharing: how can a software system control multiple processes' access to a shared resource, when each process needs exclusive control of that resource while doing its work? The mutual-exclusion solution to this makes the shared resource available only while the process is in a specific code segment called the critical section. It controls access to the shared resource by controlling each mutual execution of that part of its program where the resource would be used.

A successful solution to this problem must have at least these two properties:

- It must implement mutual exclusion: only one process can be in the critical section at a time.
- It must be free of deadlocks: if processes are trying to enter the critical section, one of them must eventually be able to do so successfully, provided no process stays in the critical section permanently.

Hardware solutions

On single-processor systems, the simplest solution to achieve mutual exclusion is to disable interrupts when a process is in a critical section. This will prevent any interrupt service routines (such as the system timer, I/O interrupt request, etc) from running (effectively preventing a process from being interrupted). Although this solution is effective, it leads to many problems. If a critical section is long, then the system clock will drift every time a critical section is executed because the timer interrupt (which keeps the system clock in sync) is no longer serviced, so tracking time is impossible during the critical section. Also, if a process halts during its critical section, control will never be returned to another process, effectively halting the entire system. A more elegant method for achieving mutual exclusion is the busy-wait.

Busy-waiting is effective for both single-processor and multiprocessor systems. The use of shared memory and an atomic (remember - we talked about atomic) test-and-set instruction provide the mutual exclusion. A process can test-and-set on a variable in a section of shared memory, and since the operation is atomic, only one process can set the flag at a time. Any process that is unsuccessful in setting the flag (it is unsuccessful because the process can NOT gain access to the variable until the other process releases it) can either go on to do other tasks and try again later, release the processor to another process and try again later, or continue to loop while checking the flag until it is successful in acquiring it. Preemption is still possible, so this method allows the system to continue to function—even if a process halts while holding the lock.

Software solutions

In addition to hardware-supported solutions, some software solutions exist that use busy waiting to achieve mutual exclusion.

It is often preferable to use synchronization facilities provided by an operating system's multithreading library, which will take advantage of hardware solutions if possible but will use software solutions if no hardware solutions exist. For example, when the operating system's lock library is used and a thread tries to acquire an already acquired lock, the operating system could suspend the thread using a context switch and swap it out with another thread that is ready to be run, or could put that processor into a low power state if there is no other thread that can be run

Adapted from:
"Mutual exclusion" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 10.3: Mutual Exclusion is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

10.4: Interprocess Communication

IPC (InterProcess Communication)

In computer science, inter-process communication or interprocess communication (IPC) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data. Typically, applications using IPC, are categorized as clients and servers, where the client requests data and the server responds to client requests. Many applications are both clients and servers, as commonly seen in distributed computing.

An independent process is not affected by the execution of other processes while cooperating processes can be affected by, and may affect, other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations where the co-operative nature can be utilized for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions.

IPC is very important to the design process for operating system kernels that desire to be kept small, therefore reduce the number of functionalities provided by the kernel. Those functionalities are then obtained by communicating with servers via IPC, leading to a large increase in communication when compared to a regular type of operating system kernel, which provides a lot more functionality.

Methods in Interprocess Communication

There are several different ways to implement IPC. IPC is set of programming interfaces, used by programs to communicate between series of processes. This allows running programs concurrently in an Operating System. Below are the methods in IPC:

1. **Pipes (Same Process)**
   This allows flow of data in one direction only. Analogous to simplex systems (Keyboard). Data from the output is usually buffered until input process receives it which must have a common origin.
2. **Names Pipes (Different Processes)**
   This is a pipe with a specific name it can be used in processes that don't have a shared common process origin. E.g. is FIFO where the details written to a pipe is first named.
3. **Message Queuing**
   This allows messages to be passed between processes using either a single queue or several message queue. This is managed by system kernel these messages are coordinated using an API.
4. **Semaphores**
   This is used in solving problems associated with synchronization and to avoid race condition. These are integer values which are greater than or equal to 0.
5. **Shared memory**
   This allows the interchange of data through a defined area of memory. Semaphore values have to be obtained before data can get access to shared memory.
6. **Sockets**
   This method is mostly used to communicate over a network between a client and a server. It allows for a standard connection which is computer and OS independent.

We will discuss a couple of these concepts.

Adapted from:
"Inter Process Communication (IPC)" by ShubhamMaurya3, Geeks for Geeks is licensed under CC BY-SA 4.0
"Methods in Interprocess Communication" by Aniket_Dusey, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 10.4: Interprocess Communication is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

10.4.1: IPC - Semaphores

What is a sempahore

In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple processes and avoid critical section problems in a concurrent system such as a multitasking operating system. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions.

A useful way to think of a semaphore as used in a real-world system is as a record of how many units of a particular resource are available, coupled with operations to adjust that record safely (i.e., to avoid race conditions) as units are acquired or become free, and, if necessary, wait until a unit of the resource becomes available.

Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores and are used to implement locks.

Library analogy

Suppose a library has 10 identical study rooms, to be used by one student at a time. Students must request a room from the front desk if they wish to use a study room. If no rooms are free, students wait at the desk until someone relinquishes a room. When a student has finished using a room, the student must return to the desk and indicate that one room has become free.

In the simplest implementation, the clerk at the front desk knows only the number of free rooms available, which they only know correctly if all of the students actually use their room while they've signed up for them and return them when they're done. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. The room can be used for as long as desired, and so it is not possible to book rooms ahead of time.

In this scenario the front desk count-holder represents a counting semaphore, the rooms are the resource, and the students represent processes/threads. The value of the semaphore in this scenario is initially 10, with all rooms empty. When a student requests a room, they are granted access, and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7 and so on. If someone requests a room and the current value of the semaphore is 0,[3] they are forced to wait until a room is freed (when the count is increased from 0). If one of the rooms was released, but there are several students waiting, then any method can be used to select the one who will occupy the room (like FIFO or flipping a coin). And of course, a student needs to inform the clerk about releasing their room only after really leaving it, otherwise, there can be an awkward situation when such student is in the process of leaving the room (they are packing their textbooks, etc.) and another student enters the room before they leave it.

Important observations

When used to control access to a pool of resources, a semaphore tracks only how many resources are free; it does not keep track of which of the resources are free. Some other mechanism (possibly involving more semaphores) may be required to select a particular free resource.

The paradigm is especially powerful because the semaphore count may serve as a useful trigger for a number of different actions. The librarian above may turn the lights off in the study hall when there are no students remaining, or may place a sign that says the rooms are very busy when most of the rooms are occupied.

The success of the semaphore requires applications to follow it correctly. Fairness and safety are likely to be compromised (which practically means a program may behave slowly, act erratically, hang or crash) if even a single process acts incorrectly. This includes:

- requesting a resource and forgetting to release it;
- releasing a resource that was never requested;
- holding a resource for a long time without needing it;
- using a resource without requesting it first (or after releasing it).

Adapted from:
"Semaphore (programming)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 10.4.1: IPC - Semaphores is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

10.4.2: IPC - Monitors

Monitors in Process Synchronization

The monitor is one of the ways to achieve process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. Not all programming languages provide for monitors.

The proper basic usage of a monitor is: (the italicized text are all comments explaining what is going on)

```
acquire(m); // Acquire this monitor's lock - this prevents other processes from being
while (!condition) { // While the condition that we are waiting for is not true (in pr
        wait(m, condition); // Wait on this monitor's lock (tht is the variable m) and
}
// ... Critical section of code goes here ...
signal(condition2); // condition2 might be the same as condition or different.
release(m); // Release this monitor's lock, now one of the processes sitting in the wa
```

Advantages of Monitor:
Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore.

Disadvantages of Monitor:
Monitors have to be implemented as part of the programming language . The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. Some languages that do support monitors are Java,C#,Visual Basic,Ada and concurrent Euclid.

Adapted from:
"Monitors in Process Synchronization" by shivanshukumarsingh1, Geeks for Geeks is licensed under CC BY-SA 4.0
"Monitor (synchronization)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 10.4.2: IPC - Monitors is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

10.4.3: IPC - Message Passing / Shared Memory

IPC through Message Passing

As we previously discussed - a process can be one of two different types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations when co-operative nature can be utilised for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through either of these techniques:

1. Shared Memory
2. Message passing

The Figure 1 below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

An operating system can implement both method of communication. First, there is the shared memory method of communication. Communication between processes using shared memory requires processes to share some variable and it is usually left up to the programmer to implement it. Sharing memory works in this manner: process1 and process2 are executing simultaneously and they share some resources. Process1 generates data based on computations in the code. Process1 stores this data in shared memory. When process2 needs to use the shared data, it will check in the shared memory segment and use the data that process1 placed there. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

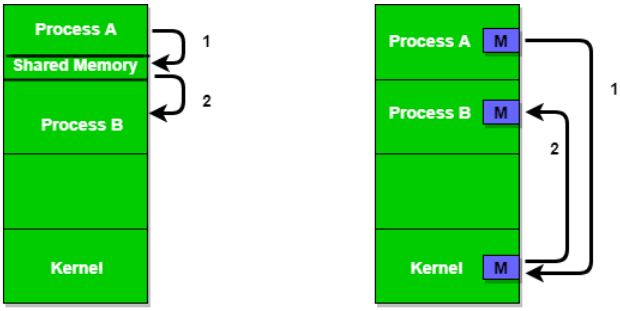


Figure 10.4.3.1: Shared Memory and Message Passing. ("Shared Memory and Message Passing" by ShubhamMaurya3, Geeks for Geeks is licensed under CC BY-SA 4.0)

Second, there is communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using a system's library functions send() and receive().
  We need at least two primitives:
  – **send**(message, destination) or **send**(message)
  – **receive**(message, host) or **receive**(message)

To send a message, Process A, sends a message via the communication link that has been opened between the 2 processes. Using the send() function it send the necessary message. Process B, which is monitoring the communication link, uses the receive() function to pick up the message and performs the necessary processing based on the message it has received. The message size can

be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer.

Adapted from:
"Inter Process Communication (IPC)" by ShubhamMaurya3, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 10.4.3: IPC - Message Passing / Shared Memory is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

11: Concurrency- Deadlock and Starvation

11.1: Concept and Principles of Deadlock

11.2: Deadlock Detection and Prevention

11.3: Starvation

11.4: Dining Philosopher Problem

This page titled 11: Concurrency- Deadlock and Starvation is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

11.1: Concept and Principles of Deadlock

Deadlock

In concurrent computing, a deadlock is a state in which each member of a group waits for another member, including itself, to take action, such as sending a message or more commonly releasing a lock. Deadlocks are a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to arbitrate shared resources and implement process synchronization.

In an operating system, a deadlock occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system is said to be in a deadlock.

In a communications system, deadlocks occur mainly due to lost or corrupt signals rather than resource contention.

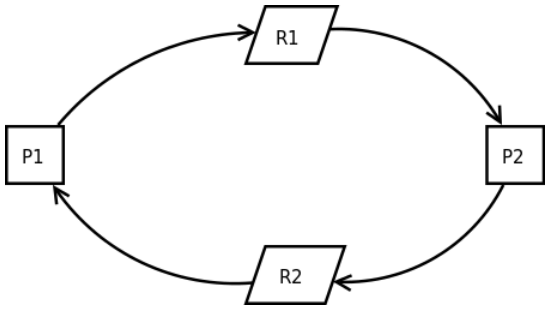


Figure 11.1.1: Both processes need resources to continue execution. *P1* requires additional resource *R1* and is in possession of resource *R2*, *P2* requires additional resource *R2* and is in possession of *R1*; neither process can continue.
("Process deadlock" by Wikimedia Commons is licensed under CC BY-SA 4.0)

The previous image show a simple instance of deadlock. Two resources are "stuck", because the other process has control of the resource that the process needs to continue to process. While this can occure quite easily, there is usually code in place to keep this from happening. As we discussed in the previous module there are various inter-process communication techniques that can actually keep processes from becoming deadlocked due to resource contention. So, often times when we hit a deadlock like this it is something to do with the IPC that is not handling this situation properly.

Necessary conditions

A deadlock situation on a resource can arise if and only if all of the following conditions hold simultaneously in a system:

- **Mutual exclusion**: At least one resource must be held in a non-shareable mode. Otherwise, the processes would not be prevented from using the resource when necessary. Only one process can use the resource at any given instant of time.
- **Hold and wait or resource holding**: a process is currently holding at least one resource and requesting additional resources which are being held by other processes.
- **No preemption**: a resource can be released only voluntarily by the process holding it.
- **Circular wait**: each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, P = {P1, P2, …, PN}, such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3 and so on until PN is waiting for a resource held by P1.

These four conditions are known as the Coffman conditions from their first description in a 1971 article by Edward G. Coffman, Jr.

While these conditions are sufficient to produce a deadlock on single-instance resource systems, they only indicate the possibility of deadlock on systems having multiple instances of resources.

The following image shows 4 processes and a single resource. The image shows 2 processes contending for the single reource (the grey circl in the middle), the it shows 3 processes contending for that resource, then finally how it looks when 4 processes contend

for the same resource. The image depicts the processes waiting to gain access to the resource - only one resource at a time can have access.

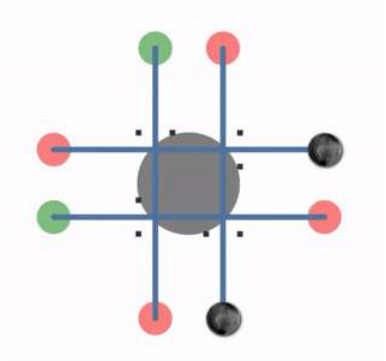


Figure 11.1.1: Four processes (blue lines) compete for one resource (grey circle), following a right-before-left policy. A deadlock occurs when all processes lock the resource simultaneously (black lines). The deadlock can be resolved by breaking the symmetry. ("Marble Machine" by Wikimedia Commons is licensed under CC BY-SA 4.0)

Adapted from:
"Deadlock" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 11.1: Concept and Principles of Deadlock is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

11.2: Deadlock Detection and Prevention

Deadlock Handling

Most current operating systems cannot prevent deadlocks. When a deadlock occurs, different operating systems respond to them in different non-standard manners. Most approaches work by preventing one of the four Coffman conditions from occurring, especially the fourth one. Major approaches are as follows.

Ignoring deadlock

In this approach, it is assumed that a deadlock will never occur. This is also an application of the Ostrich algorithm. This approach was initially used by MINIX and UNIX. This is used when the time intervals between occurrences of deadlocks are large and the data loss incurred each time is tolerable.

Ignoring deadlocks can be safely done if deadlocks are formally proven to never occur.

Detection

Under the deadlock detection, deadlocks are allowed to occur. Then the state of the system is examined to detect that a deadlock has occurred and subsequently it is corrected. An algorithm is employed that tracks resource allocation and process states, it rolls back and restarts one or more of the processes in order to remove the detected deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler of the operating system.

After a deadlock is detected, it can be corrected by using one of the following methods

1. Process termination: one or more processes involved in the deadlock may be aborted. One could choose to abort all competing processes involved in the deadlock. This ensures that deadlock is resolved with certainty and speed. But the expense is high as partial computations will be lost. Or, one could choose to abort one process at a time until the deadlock is resolved. This approach has high overhead because after each abort an algorithm must determine whether the system is still in deadlock. Several factors must be considered while choosing a candidate for termination, such as priority and age of the process.
2. Resource preemption: resources allocated to various processes may be successively preempted and allocated to other processes until the deadlock is broken.

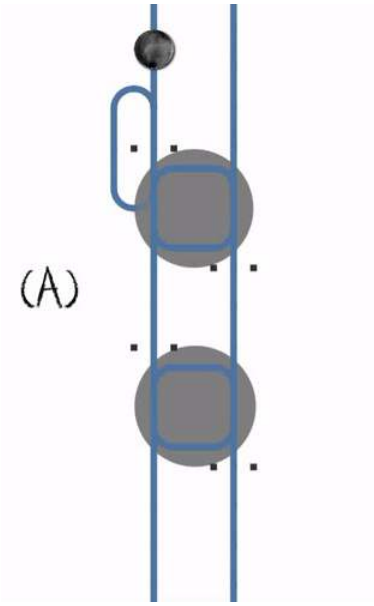


Figure 11.2.1: Two processes concurring for two resources. A deadlock occurs when the first process locks the first resource at the same time as the second process locks the second resource. The deadlock can be resolved by cancelling and restarting the first process.
("Two Processes - Two Resources" by Wikimedia Commons is licensed under CC BY-SA 4.0)

In the above image - there are 2 resources. Initially only one process is using both resources. When a second process attempts to access one of the resources, it is temporarily blocked, until the resource is released by the other process.. When 2 processes each have control of one resource there is a deadlock, as the process can not gain access to the other process it need to continue to process. Eventually, the one process is canceled, allowing the system to block the other resource and allow one of the processes to complete, which then frees up both resources for the other process.

Prevention

Deadlock prevention works by preventing one of the four Coffman conditions from occurring.

- Removing the mutual exclusion condition means that no process will have exclusive access to a resource. This proves impossible for resources that cannot be spooled. But even with spooled resources, the deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.
- The hold and wait or resource holding conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations). This advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to request resources only when it has none; First they must release all their currently held resources before requesting all the resources they will need from scratch. This too is often impractical. It is so because resources may be allocated and remain unused for long periods. Also, a process requiring a popular resource may have to wait indefinitely, as such a resource may always be allocated to some process, resulting in resource starvation.
- The no preemption condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, the inability to enforce preemption may interfere with a priority algorithm. Preemption of a "locked out" resource generally implies a rollback, and is to be avoided since it is very costly in overhead. Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control. If a process holding some resources and requests for some another resource(s) that cannot be immediately allocated to it, the condition may be removed by releasing all the currently being held resources of that process.
- The final condition is the circular wait condition. Approaches that avoid circular waits include disabling interrupts during critical sections and using a hierarchy to determine a partial ordering of resources. If no obvious hierarchy exists, even the memory address of resources has been used to determine ordering and resources are requested in the increasing order of the enumeration.

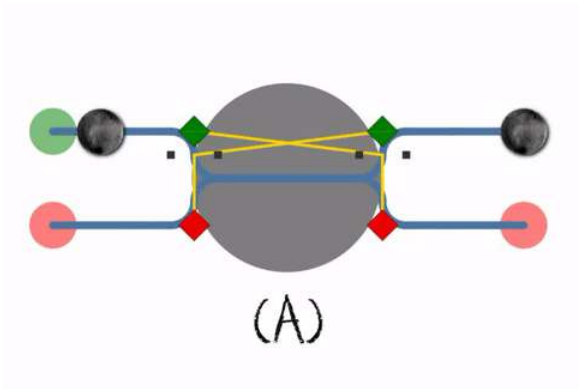


Figure 11.2.1: (A) Two processes concurring for one resource, following a first-come, first-served policy. (B) A deadlock occurs when both processes lock the resource simultaneously. (C) The deadlock can be resolved by breaking the symmetry of the locks. (D) The deadlock can be avoided by breaking the symmetry of the locking mechanism.
("Avoiding Deadlock" by **Wikimedia Commons** is licensed under CC BY-SA 4.0)

In the above image notice the yellow line - if it is the same on both sides, a deadlock can develop (scenario A shows that one process gets there first...it is difficult to see in the gif - but that is why there is NOT a deadlock - first come - first serve). Watch - when the yellow lines, representing the locking mechanism, are different on each side then we have a method to break the deadlock and allow the left side process tom complete and freeing up the resource for the right and resource to complete.

Adapted from:
"Deadlock" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 11.2: Deadlock Detection and Prevention is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

11.3: Starvation

Concept of Starvation

Starvation is usually caused by an overly simplistic scheduling algorithm. For example, if a system always switches between the first two tasks while a third never gets to run, then the third task is being starved of CPU time. The scheduling algorithm, which is part of the kernel, is supposed to allocate resources equally among all processes; that is, the algorithm should allocate resources so that no process continually is blocked from accessing the resources it needs to execute to completion.

Many operating system schedulers employ the concept of process priority. Each process gets a priority - usually the lower the number the higher the priority - making a priority of zero the highest priority a process can have. A high priority process A will run before a low priority process B. If the high priority process (process A) blocks and never gives up control of the processor, the low priority process (B) will (in some systems) never be scheduled—it will experience starvation. If there is an even higher priority process X, which is dependent on a result from process B, then process X might never finish, even though it is the most important process in the system. This condition is called a priority inversion. Modern scheduling algorithms normally contain code to guarantee that all processes will receive a minimum amount of each important resource (most often CPU time) in order to prevent any process from being subjected to starvation.

Starvation is normally caused by a deadlock that causes a process to freeze waiting for resources. Two or more processes become deadlocked when each of them is doing nothing while waiting for a resource occupied by another program in the same set, the two (or more) processes that are waiting can starve while waiting on the one process that has control of the resource. On the other hand, a process is in starvation when it is waiting for a resource that is continuously given to other processes because it can never complete without access to the necessary resource. Starvation-freedom is a stronger guarantee than the absence of deadlock: a mutual exclusion algorithm that must choose to allow one of two processes into a critical section and picks one arbitrarily is deadlock-free, but not starvation-free.

A possible solution to starvation is to use a scheduling algorithm with priority queue that also uses the aging technique. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if a process X has a priority of 100, it would probably be near the bottom of the priority list, it would get very little processing time on a busy system. Using the concept of aging, over some set period of time, process X's priority would decrease, to say 50. If process X still did not get enough resources or processing time, after another period of time the priority would again decrease, to say 25. Eventually process X would get to a high enough priority (low number) that it would be scheduled for access to resources/processor and would complete in a proper fashion.

Adapted from:
"Starvation (computer science)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 11.3: Starvation is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

11.4: Dining Philosopher Problem

In computer science, the dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

It was originally formulated in 1965 by Edsger Dijkstra as a student exam exercise, presented in terms of computers competing for access to tape drive peripherals.

Problem statement

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can only take the fork on their right or the one on their left as they become available and they cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; *i.e.*, each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

Problems

The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. To see that a proper solution to this problem is not obvious, consider a proposal in which each philosopher is instructed to behave as follows:

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

This attempted solution fails because it allows the system to reach a deadlock state, in which no progress is possible. This is a state in which each philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available. With the given instructions, this state can be reached, and when it is reached, each philosopher will eternally wait for another (the one to the right) to release a fork.[4]

Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example, there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of livelock. If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

Mutual exclusion is the basic idea of the problem; the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources. These issues are studied in concurrent programming. The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties exemplified by the dining philosophers problem arise far more often when multiple processes access sets of data that are being updated. Complex systems such as operating system kernels use thousands of locks and synchronizations that require strict adherence to methods and protocols if such problems as deadlock, starvation, and data corruption are to be avoided.

Resource hierarchy solution

This solution to the problem is the one originally proposed by Dijkstra. It assigns a partial order to the resources (the forks, in this case), and establishes the convention that all resources will be requested in order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources (forks) will be numbered 1 through 5 and each unit of work (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks they plan to use. The order in which each philosopher puts down the forks does not matter. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the highest-numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so he will be able to eat using two forks.

While the resource hierarchy solution avoids deadlocks, it is not always practical, especially when the list of required resources is not completely known in advance. For example, if a unit of work holds resources 3 and 5 and then determines it needs resource 2, it must release 5, then 3 before acquiring 2, and then it must re-acquire 3 and 5 in that order. Computer programs that access large numbers of database records would not run efficiently if they were required to release all higher-numbered records before accessing a new record, making the method impractical for that purpose.

The resource hierarchy solution is not *fair*. If philosopher 1 is slow to take a fork, and if philosopher 2 is quick to think and pick its forks back up, then philosopher 1 will never get to pick up both forks. A fair solution must guarantee that each philosopher will eventually eat, no matter how slowly that philosopher moves relative to the others.

Arbitrator solution

Another approach is to guarantee that a philosopher can only pick up both forks or none by introducing an arbitrator, e.g., a waiter. In order to pick up the forks, a philosopher must ask permission of the waiter. The waiter gives permission to only one philosopher at a time until the philosopher has picked up both of their forks. Putting down a fork is always allowed. The waiter can be implemented as a mutex. In addition to introducing a new central entity (the waiter), this approach can result in reduced parallelism: if a philosopher is eating and one of his neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available.

Adapted from:
"Dining philosophers problem" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 11.4: Dining Philosopher Problem is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

12: Memory Management

12.1: Random Access Memory (RAM) and Read Only Memory (ROM)
12.2: Memory Hierarchy
12.3: Requirements for Memory Management
12.4: Memory Partitioning
12.4.1: Fixed Partitioning
12.4.2: Variable Partitioning
12.4.3: Buddy System
12.5: Logical vs Physical Address
12.6: Paging
12.7: Segmentation

This page titled 12: Memory Management is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.1: Random Access Memory (RAM) and Read Only Memory (ROM)

Memory Basics

Memory is the most essential element of a computing system because without it computer can't perform simple tasks. Computer memory is of two basic type – Primary memory(RAM and ROM) and Secondary memory(hard drive,CD,etc.). Random Access Memory (RAM) is primary-volatile memory and Read Only Memory (ROM) is primary-non-volatile memory.

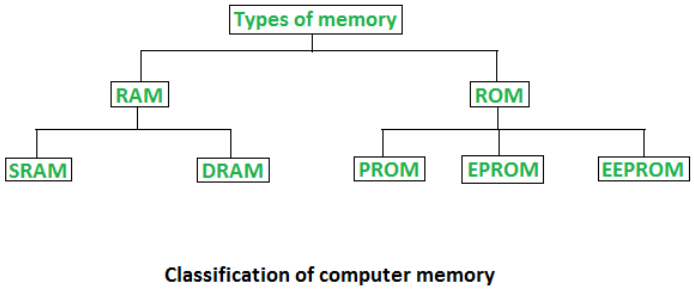


Figure 12.1.1: ("Classification of Computer Memory" by Deepanshi_Mittal, Geeks for Geeks is licensed under CC BY-SA 4.0)

1. **Random Access Memory (RAM)**

- It is also called as *read write memory* or the *main memory* or the *primary memory*.
- The programs and data that the CPU requires during execution of a program are stored in this memory.
- It is a volatile memory as the data loses when the power is turned off.
- RAM is further classified into two types- *SRAM (Static Random Access Memory)* and *DRAM (Dynamic Random Access Memory)*.

DRAM	SRAM
1. Constructed of tiny capacitors that leak electricity.	1.Constructed of circuits similar to D flip-flops.
2.Requires a recharge every few milliseconds to maintain its data.	2.Holds its contents as long as power is available.
3.Inexpensive.	3.Expensive.
4. Slower than SRAM.	4. Faster than DRAM.
5. Can store many bits per chip.	5. Can not store many bits per chip.
6. Uses less power.	6.Uses more power.
7.Generates less heat.	7.Generates more heat.
8. Used for main memory.	8. Used for cache.

Difference between SRAM and DRAM

Figure 12.1.1: Difference between SRAM and DRAM. ("Difference between SRAM and DRAM" by Deepanshi_Mittal, Geeks for Geeks is licensed under CC BY-SA 4.0)

2. **Read Only Memory (ROM)**

- Stores crucial information essential to operate the system, like the program essential to boot the computer.
- It is not volatile.
- Always retains its data.
- Used in embedded systems or where the programming needs no change.
- Used in calculators and peripheral devices.

- ROM is further classified into 4 types- *ROM*, *PROM*, *EPROM*, and *EEPROM*.

Types of Read Only Memory (ROM) –

1. **PROM (Programmable read-only memory)** – It can be programmed by user. Once programmed, the data and instructions in it cannot be changed.
2. **EPROM (Erasable Programmable read only memory)** – It can be reprogrammed. To erase data from it, expose it to ultra violet light. To reprogram it, erase all the previous data.
3. **EEPROM (Electrically erasable programmable read only memory)** – The data can be erased by applying electric field, no need of ultra violet light. We can erase only portions of the chip.

RAM	ROM
1. Temporary Storage.	1. Permanent storage.
2. Store data in MBs.	2. Store data in GBs.
3. Volatile.	3. Non-volatile.
4.Used in normal operations.	4. Used for startup process of computer.
5. Writing data is faster.	5. Writing data is slower.

Difference between RAM and ROM

Figure 12.1.1: ("Difference between RAM and ROM" by Deepanshi_Mittal, Geeks for Geeks is licensed under CC BY-SA 4.0)

Adapted from:
"Random Access Memory (RAM) and Read Only Memory (ROM)" by Deepanshi_Mittal, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 12.1: Random Access Memory (RAM) and Read Only Memory (ROM) is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.2: Memory Hierarchy

Memory Hierarchy Design and its Characteristics

In computer systems design, the concept of memory hierarchy is an enhancement to organize the computer's memory such that access time to memory is minimized. Memory hierarchy was developed based on a software program's behavior known as locality of references.The figure below depicts the different levels of memory hierarchy :

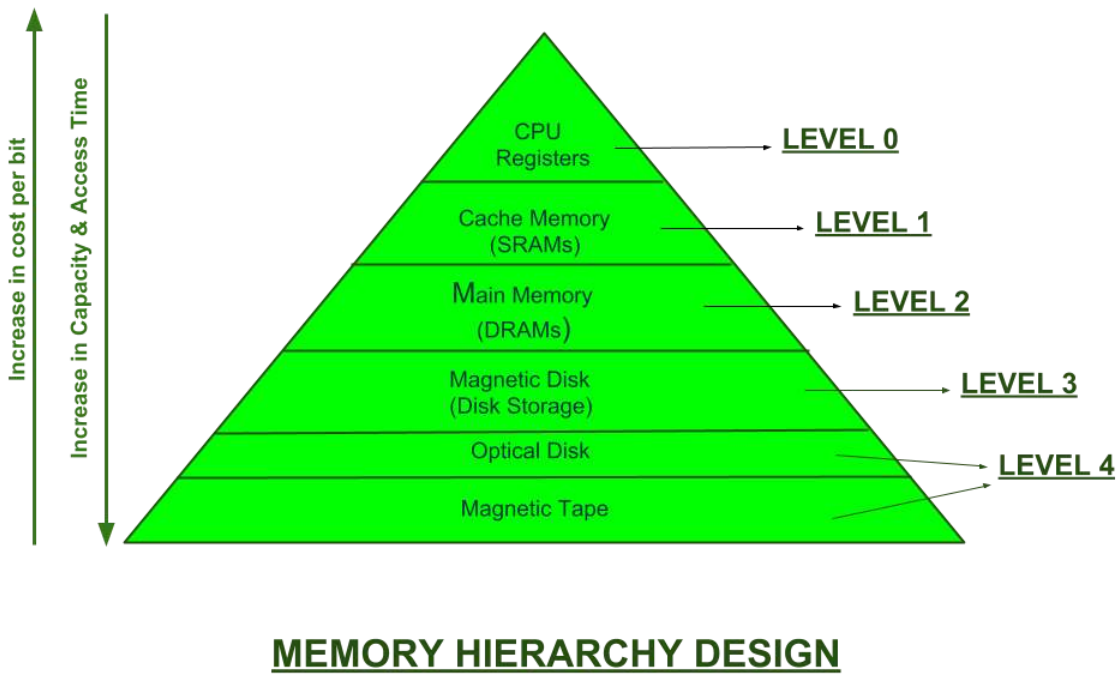


Figure 12.2.1: ("Memory Hierarchy" by RishabhJain12, Geeks for Geeks is licensed under CC BY-SA 4.0)

This Memory Hierarchy Design is divided into 2 main types:

1. **External Memory or Secondary Memory**
   This level is comprised of peripheral storage devices which are accessible by the processor via I/O Module.
2. **Internal Memory or Primary Memory**
   This level is comprised of memory that is directly accessible by the processor.

We can infer the following characteristics of Memory Hierarchy Design from the above figure:

1. **Capacity:**
   As we move from top to bottom in the hierarchy, the capacity increases.
2. **Access Time:**
   This represents the time interval between the read/write request and the availability of the data. As we move from top to bottom in the hierarchy, the access time increases.
3. **Performance:**
   In erly computer systems that were designed without the idea of memory hierarchy design, the speed gap increased between the CPU registers and main memory due to difference in access time. This results in lower system performance, an enhancement was required. This enhancement was memory hierarchy design which provided the system with greater performance. One of the most significant ways to increase system performance is to minimize how far down the memory hierarchy one has to go to manipulate data. If we can keep system using lower numbered levels (higher up the hierarchy) then we get better performance.
4. **Cost per bit:**
   As we move up the hierarchy - from bottom to top - the cost per bit increases i.e. internal memory is costlier than external memory.

Adapted from:
"Memory Hierarchy Design and its Characteristics" by RishabhJain12, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 12.2: Memory Hierarchy is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.3: Requirements for Memory Management

Requirements of Memory Management System

Memory management keeps track of the status of each memory location, whether it is allocated or free. It allocates the memory dynamically to the programs at their request and frees it for reuse when it is no longer needed.

Memory management is meant to satisfy the following requirements:

1. **Relocation –** The available memory is generally shared among a number of processes in a multiprogramming system, so it is not possible to know in advance which other programs will be resident in main memory at the time of execution of his program. Swapping the active processes in and out of the main memory enables the operating system to have a larger pool of ready-to-execute process.

   When a program gets swapped out to a disk memory, then it is not always possible that when it is swapped back into main memory then it occupies the previous memory location, since the location may still be occupied by another process. We may need to **relocate** the process to a different area of memory. Thus there is a possibility that program may be moved in main memory due to swapping.

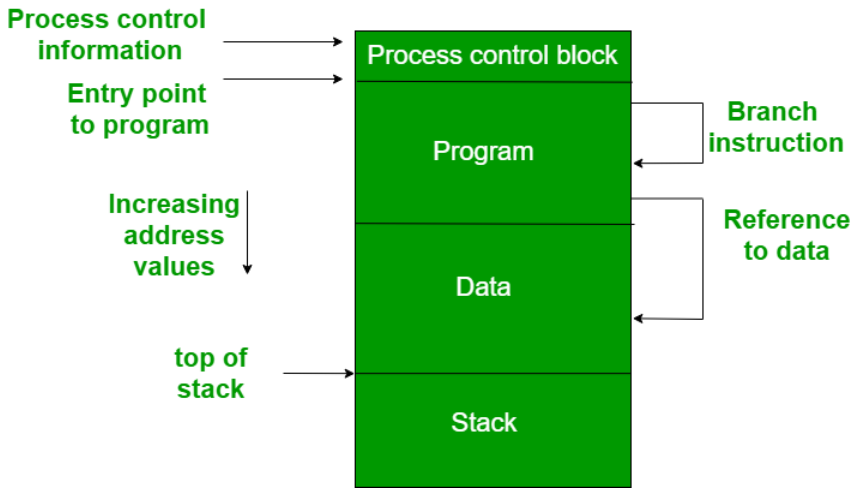


Figure 12.3.1: A process occupying a continuous region of main memory.
("Process Image" by Aditya_04, Geeks for Geeks is licensed under CC BY-SA 4.0)

   The figure depicts a process image. Every process looks like this in memory. Each process contains: 1) process control blocks; 2) a program entry point - this is the instruction where the program starts execution; 3) a program section; 4) a data section; and 5) a stack. The process image is occupying a continuous region of main memory. The operating system will need to know many things including the location of process control information, the execution stack, and the code entry. Within a program, there are memory references in various instructions and these are called logical addresses.

   After loading of the program into main memory, the processor and the operating system must be able to translate logical addresses into physical addresses. Branch instructions contain the address of the next instruction to be executed. Data reference instructions contain the address of byte or word of data referenced.
2. **Protection –** There is always a danger when we have multiple programs executing at the same time - one program may write to the address space of another program. So every process must be protected against unwanted interference if one process tries to write into the memory space of another process - whether accidental or incidental. The operating system makes a trade-off between relocation and protection requirement: in order to satisfy the relocation requirement the difficulty of satisfying the protection requirement increases in difficulty.

   It is impossible to predict the location of a program in main memory, which is why it is impossible to determine the absolute address at compile time and thereby attempt to assure protection. Most programming languages provide for dynamic calculation

of address at run time. The memory protection requirement must be satisfied by the processor rather than the operating system because the operating system does not necessarily control a process when it occupies the processor. Thus it is not possible to check the validity of memory references.

3. **Sharing –** A protection mechanism must allow several processes to access the same portion of main memory. This must allow for each processes the ability to access the same copy of the program rather than have their own separate copy.

   This concept has an advantage. For example, multiple processes may use the same system file and it is natural to load one copy of the file in main memory and let it shared by those processes. It is the task of memory management to allow controlled access to the shared areas of memory without compromising the protection. Mechanisms are used to support relocation supported sharing capabilities.
4. **Logical organization –** Main memory is organized as linear or it can be a one-dimensional address space which consists of a sequence of bytes or words. Most of the programs can be organized into modules, some of those are unmodifiable (read-only, execute only) and some of those contain data that can be modified. To effectively deal with a user program, the operating system and computer hardware must support a basic module to provide the required protection and sharing. It has the following advantages:
   - Modules are written and compiled independently and all the references from one module to another module are resolved by the system at run time.
   - Different modules are provided with different degrees of protection.
   - There are mechanisms by which modules can be shared among processes. Sharing can be provided on a module level that lets the user specify the sharing that is desired.
5. **Physical organization –** The structure of computer memory has two levels referred to as main memory and secondary memory. Main memory is relatively very fast and costly as compared to the secondary memory. Main memory is volatile. Thus secondary memory is provided for storage of data on a long-term basis while the main memory holds currently used programs. The major system concern between main memory and secondary memory is the flow of information and it is impractical for programmers to understand this for two reasons:
   - The programmer may engage in a practice known as overlaying when the main memory available for a program and its data may be insufficient. It allows different modules to be assigned to the same region of memory. One disadvantage is that it is time-consuming for the programmer.
   - In a multiprogramming environment, the programmer does not know how much space will be available at the time of coding and where that space will be located inside the memory.

Adapted from:
"Requirements of Memory Management System" by Aditya_04, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 12.3: Requirements for Memory Management is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.4: Memory Partitioning

Partition Allocation Methods in Memory Management

In the world of computer operating system, there are four common memory management techniques. They are:

1. **Single contiguous allocation:** Simplest allocation method used by MS-DOS. All memory (except some reserved for OS) is available to a process.
2. **Partitioned allocation:** Memory is divided into different blocks or partitions. Each process is allocated according to the requirement.
3. **Paged memory management:** Memory is divided into fixed-sized units called page frames, used in a virtual memory environment.
4. **Segmented memory management:** Memory is divided into different segments (a segment is a logical grouping of the process' data or code).In this management, allocated memory doesn't have to be contiguous.

Most of the operating systems (for example Windows and Linux) use segmentation with paging. A process is divided into segments and individual segments have pages.

In p**artition allocation**, when there is more than one partition freely available to accommodate a process's request, a partition must be selected. To choose a particular partition, a partition allocation method is needed. A partition allocation method is considered better if it avoids internal fragmentation.

When it is time to load a process into the main memory and if there is more than one free block of memory of sufficient size then the OS decides which free block to allocate.

There are different Placement Algorithm:

1. **First Fit**: In the first fit, the partition is allocated which is the first sufficient block from the top of main memory. It scans memory from the beginning and chooses the first available block that is large enough. Thus it allocates the first hole that is large enough.

Figure 12.4.1: First Fit. ("First Fit" by deepakmkoshy, Geeks for Geeks is licensed under CC BY-SA 4.0)

2. **Best Fit** Allocate the process to the partition which is the first smallest sufficient partition among the free available partition. It searches the entire list of holes to find the smallest hole whose size is greater than or equal to the size of the process.

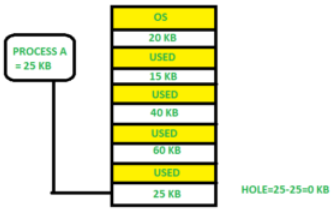


Figure 12.4.1: Best Fit. ("Best Fit" by deepakmkoshy, Geeks for Geeks is licensed under CC BY-SA 4.0)

3. **Worst Fit** Allocate the process to the partition which is the largest sufficient among the freely available partitions available in the main memory. It is opposite to the best-fit algorithm. It searches the entire list of holes to find the largest hole and allocate it

to process.

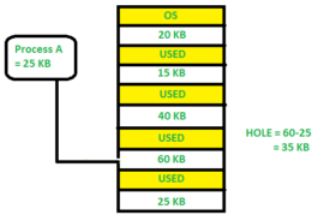


Figure 12.4.1: Worst Fit. ("Worst Fit" by deepakmkoshy, Geeks for Geeks is licensed under CC BY-SA 4.0)

4. **Next Fit:** Next fit is similar to the first fit but it will search for the first sufficient partition from the last allocation point.

Adapted from:
"Partition Allocation Methods in Memory Management" by deepakmkoshy, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 12.4: Memory Partitioning is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.4.1: Fixed Partitioning

Fixed (or static) Partitioning in Operating System

This is the oldest and simplest technique that allows more than one processes to be loaded into main memory. In this partitioning method the number of partitions (non-overlapping) in RAM are all a fixed size, but they may or may not be same size. This method of partitioning provides for contiguous allocation, hence no spanning is allowed. The partition sizes are made before execution or during system configuration.

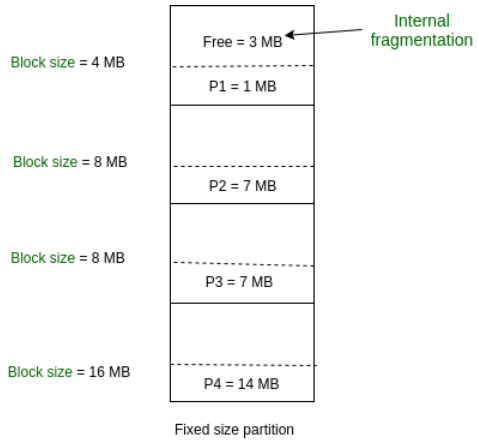


Figure 12.4.1.1: Fixed Sized partition Example. ("Fixed Sized Partition" by Vidhayak_Chacha, Geeks for Geeks is licensed under CC BY-SA 4.0)

As illustrated in above figure, first process is only consuming 1MB out of 4MB in the main memory.
Hence, Internal Fragmentation in first block is (4-1) = 3MB.
Sum of Internal Fragmentation in every block = (4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7MB.

Suppose process P5 of size 7MB comes. But this process cannot be accommodated inspite of available free space because of contiguous allocation (as spanning is not allowed). Hence, 7MB becomes part of External Fragmentation.

There are some advantages and disadvantages of fixed partitioning.

Advantages of Fixed Partitioning

- **Easy to implement:**
  Algorithms needed to implement Fixed Partitioning are easy to implement. It simply requires putting a process into certain partition without focussing on the emergence of Internal and External Fragmentation.
- **Little OS overhead:**
  Processing of Fixed Partitioning require lesser excess and indirect computational power.

Disadvantages of Fixed Partitioning

- **Internal Fragmentation:**
  Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.
- **External Fragmentation:**
  The total unused space (as stated above) of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form (as spanning is not allowed).
- **Limit process size:**
  Process of size greater than size of partition in Main Memory cannot be accommodated. Partition size cannot be varied according to the size of incoming process's size. Hence, process size of 32MB in above stated example is invalid.
- **Limitation on Degree of Multiprogramming:**
  Partition in Main Memory are made before execution or during system configure. Main Memory is divided into fixed number

of partition. Suppose if there are $n1$ partitions in RAM and $n2$ are the number of processes, then $n2 <= n1$ condition must be fulfilled. Number of processes greater than number of partitions in RAM is invalid in Fixed Partitioning.

Adapted from:
"Fixed (or static) Partitioning in Operating System" by Vidhayak_Chacha, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 12.4.1: Fixed Partitioning is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.4.2: Variable Partitioning

Variable (or dynamic) Partitioning in Operating System

Variable partitioning is part of the contiguous allocation technique. It is used to alleviate the problem faced by fixed partitioning. As opposed to fixed partitioning, in variable partitioning, partitions are not created until a process executes. At the time it is read into main memory, the process is given exactly the amount of memory needed. This technique, like the fixed partitioning scheme previously discussed have been replaced by more complex and efficient techniques.

Various features associated with variable partitioning.

- Initially RAM is empty and partitions are made during the run-time according to process's need instead of partitioning during system configure.
- The size of partition will be equal to incoming process.
- The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilisation of RAM.
- Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.

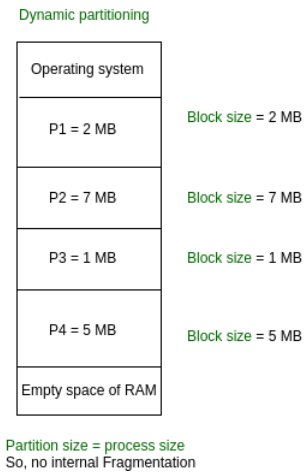


Figure 12.4.2.1: Variable Partitioned Memory. ("Variable Patitiong" by **Vidhayak_Chacha**, Geeks for Geeks is licensed under CC BY-SA 4.0)

Advantages of Variable Partitioning

1. **No Internal Fragmentation:**
   In variable Partitioning, space in main memory is allocated strictly according to the need of process, hence there is no case of internal fragmentation. There will be no unused space left in the partition.
2. **No restriction on Degree of Multiprogramming:**
   More number of processes can be accommodated due to absence of internal fragmentation. A process can be loaded until the memory is empty.
3. **No Limitation on the size of the process:**
   In Fixed partitioning, the process with the size greater than the size of the largest partition could not be loaded and process can not be divided as it is invalid in contiguous allocation technique. Here, In variable partitioning, the process size can't be restricted since the partition size is decided according to the process size.

Disadvantages of Variable Partitioning

1. **Difficult Implementation:**
   Implementing variable Partitioning is difficult as compared to Fixed Partitioning as it involves allocation of memory during run-time rather than during system configure.
2. **External Fragmentation:**
   There will be external fragmentation inspite of absence of internal fragmentation.

For example, suppose in above example- process P1(2MB) and process P3(1MB) completed their execution. Hence two spaces are left i.e. 2MB and 1MB. Let's suppose process P5 of size 3MB comes. The empty space in memory cannot be allocated as no spanning is allowed in contiguous allocation. The rule says that process must be contiguously present in main memory to get executed. Hence it results in External Fragmentation.

Adapted from:
"Variable (or dynamic) Partitioning in Operating System" by Vidhayak_Chacha, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 12.4.2: Variable Partitioning is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.4.3: Buddy System

Buddy System – Memory allocation technique

Static partition schemes suffer from the **limitation** of having the fixed number of active processes and the usage of space may also not be optimal. The **buddy system** is a memory allocation and management algorithm that manages memory in **power of two increments**. Assume the memory size is 2^U, suppose a size of S is required.

- **If 2^{U-1}<S<=2^U:** Allocate the whole block
- **Else:** Recursively divide the block equally and test the condition at each time, when it satisfies, allocate the block and get out the loop.

System also keep the record of all the unallocated blocks each and can merge these different size blocks to make one big chunk.

Advantage

- Easy to implement a buddy system
- Allocates block of correct size
- It is easy to merge adjacent holes
- Fast to allocate memory and de-allocating memory

Disadvantage

- It requires all allocation unit to be powers of two
- It leads to internal fragmentation

Example

Consider a system having buddy system with physical address space 128 KB.Calculate the size of partition for 18 KB process.

Solution

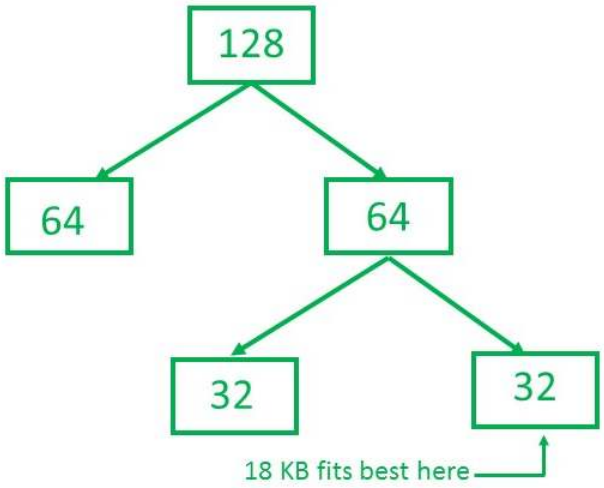


Figure 12.4.3.1: Buddy System Partitioning
("Buddy System Partitioning" by Samit Mandal, Geeks for Geeks is licensed under CC BY-SA 4.0)

So, size of partition for 18 KB process = 32 KB. It divides by 2, till possible to get minimum block to fit 18 KB.

Adapted from:
"Buddy System – Memory allocation technique" by Samit Mandal, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 12.4.3: Buddy System is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.5: Logical vs Physical Address

Logical and Physical Addresses in an Operating System

A logical address is generated by CPU while a program is running. Since a logical address does not physically exists it is also known as a virtual address. This address is used as a reference by the CPU to access the actual physical memory location.

There is a hardware device called Memory-Management Unit is used for mapping logical address to its corresponding physical address.

A physical address identifies the physical location of a specific data element in memory. The user never directly deals with the physical address but can determine the physical address by its corresponding logical address. The user program generates the logical address and believes that the program is running in this logical address space, but the program needs physical memory for its execution, therefore, the logical address must be mapped to the physical address by the MMU before the addresses are used. The term physical address space is used for all physical addresses corresponding to the logical addresses in a logical address space.

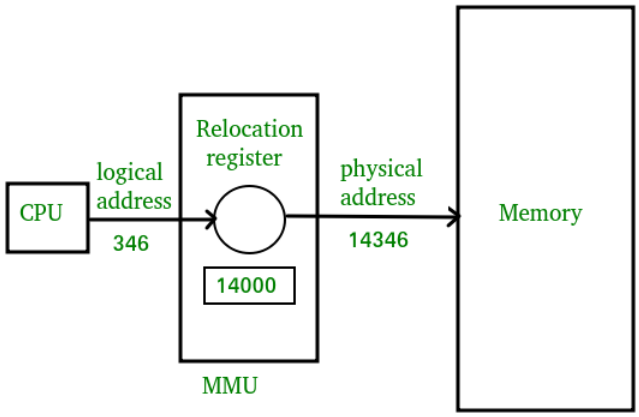


Figure 12.5.1: MMU Operation.
("MMU Operation" by Ankit_Bisht, Geeks for Geeks is licensed under CC BY-SA 4.0)

Differences Between Logical and Physical Address in Operating System

1. The basic difference between Logical and physical address is that Logical address is generated by CPU in perspective of a program whereas the physical address is a location that exists in the memory unit.
2. Logical Address Space is the set of all logical addresses generated by CPU for a program whereas the set of all physical address mapped to corresponding logical addresses is called Physical Address Space.
3. The logical address does not exist physically in the memory whereas physical address is a location in the memory that can be accessed physically.
4. Identical logical addresses are generated by Compile-time and Load time address binding methods whereas they differs from each other in run-time address binding method.
5. The logical address is generated by the CPU while the program is running whereas the physical address is computed by the Memory Management Unit (MMU).

Comparison Chart:

Paramenter	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	generated by CPU	location in a memory unit
Address Space	Logical Address Space is set of all logical addresses generated by CPU in reference to a program.	Physical Address is set of all physical addresses mapped to the corresponding logical addresses.
Visibility	User can view the logical address of a program.	User can never view physical address of program.
Generation	generated by the CPU	Computed by MMU
Access	The user can use the logical address to access the physical address.	The user can indirectly access physical address but not directly.

Mapping Virtual Addresses to Physical Addresses

Memory consists of large array addresses. It is the responsibility of the CPU to fetch instruction address from the program counter. These instruction may cause loading or storing to specific memory address.

Address binding is the process of mapping from one address space to another address space. Logical addresses are generated by CPU during execution whereas physical address refers to location in a physical memory unit (the one that is loaded into memory). Note that users deal only with logical address (virtual address). The logical address is translated by the MMU. The output of this process is the appropriate physical address of the data in RAM.

An address binding can be done in three different ways:

Compile Time – If at compile time you know where a process will reside in memory then an absolute address can be generated - that is a physical address is generated in the program executable during compilation. Loading such an executable into memory is very fast. But if the generated address space is occupied by other process, then the program crashes and it becomes necessary to recompile the program to use a virtual address space.

Load time – If it is not known at the compile time where process will reside then relocatable addresses will be generated. The loader translates the relocatable address to absolute address. The base address of the process in main memory is added to all logical addresses by the loader to generate absolute address. If the base address of the process changes then we need to reload the process again.

Execution time- The instructions are already loaded into memory and are processed by the CPU. Additional memory may be allocated and/or deallocated at this time. This process is used if the process can be moved from one memory to another during execution (dynamic linking-Linking that is done during load or run time). e.g – Compaction.

MMU(Memory Management Unit)-
The run time mapping between virtual address and physical address is done by a hardware device known as MMU.

In memory management, Operating System will handle the processes and moves the processes between disk and memory for execution . It keeps the track of available and used memory.

Adapted from:
"Logical and Physical Address in Operating System" by Ankit_Bisht, Geeks for Geeks is licensed under CC BY-SA 4.0
"Mapping Virtual Addresses to Physical Addresses" by NEERAJ NEGI, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 12.5: Logical vs Physical Address is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.6: Paging

Memory Paging

A page, memory page, or virtual page is a fixed-length contiguous block of virtual memory, described by a single entry in the page table. It is the smallest unit of data for memory management in a virtual memory operating system. Similarly, a page frame is the smallest fixed-length contiguous block of physical memory into which memory pages are mapped by the operating system

In computer operating systems, memory paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory. In this scheme, the operating system retrieves data from secondary storage (usually the swap space on the disk) in same-size blocks called pages. Paging is an important part of virtual memory implementations in modern operating systems, using secondary storage to let programs exceed the size of available physical memory.

Page Table

Part of the concept of paging is the page table, which is a data structure used by the virtual memory system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the executed program, while physical addresses are used by the hardware, or more specifically, by the RAM subsystem. The page table is a key component of virtual address translation which is necessary to access data in memory.

Role of the page table

In operating systems that use virtual memory, every process is given the impression that it is working with large, contiguous sections of memory. Physically, the memory of each process may be dispersed across different areas of physical memory, or may have been moved (paged out) to another storage, typically to a hard disk drive or solid state drive.

When a process requests access to data in its memory, it is the responsibility of the operating system to map the virtual address provided by the process to the physical address of the actual memory where that data is stored. The page table is where the operating system stores its mappings of virtual addresses to physical addresses, with each mapping also known as a *page table entry* (PTE).

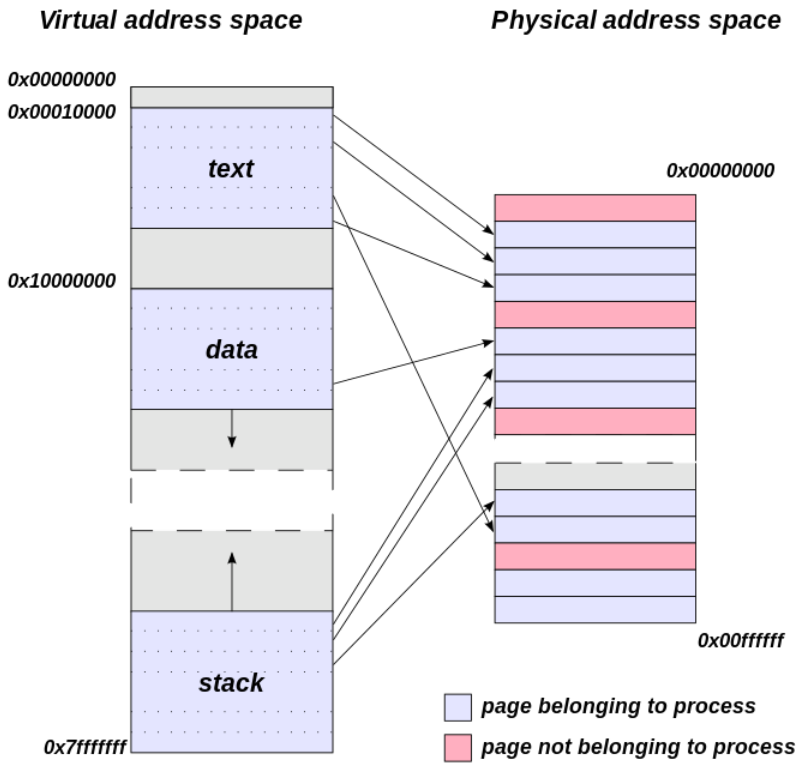


Figure 12.6.1: Mapping Virtual Memory to Physical Memory.
("Mapping Virtual Addresses to Physical Addresses" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0)

The above image shows the relationship between pages addressed by virtual addresses and the pages in physical memory, within a simple address space scheme. Physical memory can contain pages belonging to many processes. If a page is not used for a period of time, the operating system can, if deemed necessary, move that page to secondary storage. The purple indicates where in physical memory the pieces of the executing processes reside - BUT - in the virtual environments, the memory is contiguous.

The translation process

The CPU's memory management unit (MMU) stores a cache of recently used mappings from the operating system's page table. This is called the translation lookaside buffer (TLB), which is an associative cache.

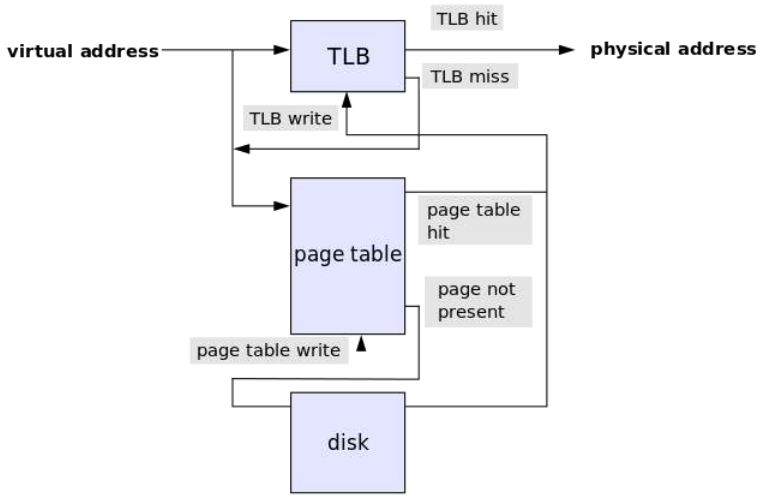


Figure 12.6.1: Actions taken upon a virtual to physical address translation request.
("Actions taken upon a virtual to physical address translation request" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0)

When a virtual address needs to be translated into a physical address, the TLB is searched first. If a match is found (a *TLB hit*), the physical address is returned and memory access can continue. However, if there is no match (called a *TLB miss*), the memory management unit, or the operating system TLB miss handler, will typically look up the address mapping in the page table to see whether a mapping exists (a *page walk*). If one exists, it is written back to the TLB (this must be done, as the hardware accesses memory through the TLB in a virtual memory system), and the faulting instruction is restarted (this may happen in parallel as well). The subsequent translation will find a TLB hit, and the memory access will continue.

Translation failures

The page table lookup may fail, triggering a page fault, for two reasons:

- The lookup may fail if there is no translation available for the virtual address, meaning that virtual address is invalid. This will typically occur because of a programming error, and the operating system must take some action to deal with the problem. On modern operating systems, it will cause a segmentation fault signal being sent to the offending program.
- The lookup may also fail if the page is currently not resident in physical memory. This will occur if the requested page has been moved out of physical memory to make room for another page. In this case the page is paged out to a secondary store located on a medium such as a hard disk drive (this secondary store, or "backing store", is often called a "swap partition" if it is a disk partition, or a *swap file*, "swapfile" or "page file" if it is a file). When this happens the page needs to be taken from disk and put back into physical memory. A similar mechanism is used for memory-mapped files, which are mapped to virtual memory and loaded to physical memory on demand.

When physical memory is not full this is a simple operation; the page is written back into physical memory, the page table and TLB are updated, and the instruction is restarted. However, when physical memory is full, one or more pages in physical memory will need to be paged out to make room for the requested page. The page table needs to be updated to mark that the pages that were previously in physical memory are no longer there, and to mark that the page that was on disk is now in physical memory. The TLB also needs to be updated, including removal of the paged-out page from it, and the instruction restarted. Which page to page out is the subject of page replacement algorithms.

Some MMUs trigger a page fault for other reasons, whether or not the page is currently resident in physical memory and mapped into the virtual address space of a process:

- Attempting to write when the page table has the read-only bit set causes a page fault. This is a normal part of many operating system's implementation of copy-on-write; it may also occur when a write is done to a location from which the process is allowed to read but to which it is not allowed to write, in which case a signal is delivered to the process.
- Attempting to execute code when the page table has the NX bit (no-execute bit) set in the page table causes a page fault. This can be used by an operating system, in combination with the read-only bit, to provide a Write XOR Execute feature that stops some kinds of exploits

Adapted from:
"Memory paging" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Page (computer memory)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Page table" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 12.6: Paging is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.7: Segmentation

Segmentation in Operating System

A process is divided into segments. The segments are not required to be of the same sizes.

There are 2 types of segmentation:

1. **Virtual memory segmentation**
   Each process is divided into a number of segments, not all of which are resident at any one point in time.
2. **Simple segmentation**
   Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

There is no simple relationship between logical addresses and physical addresses in segmentation. A table stores the information about all such segments and is called Segment Table.

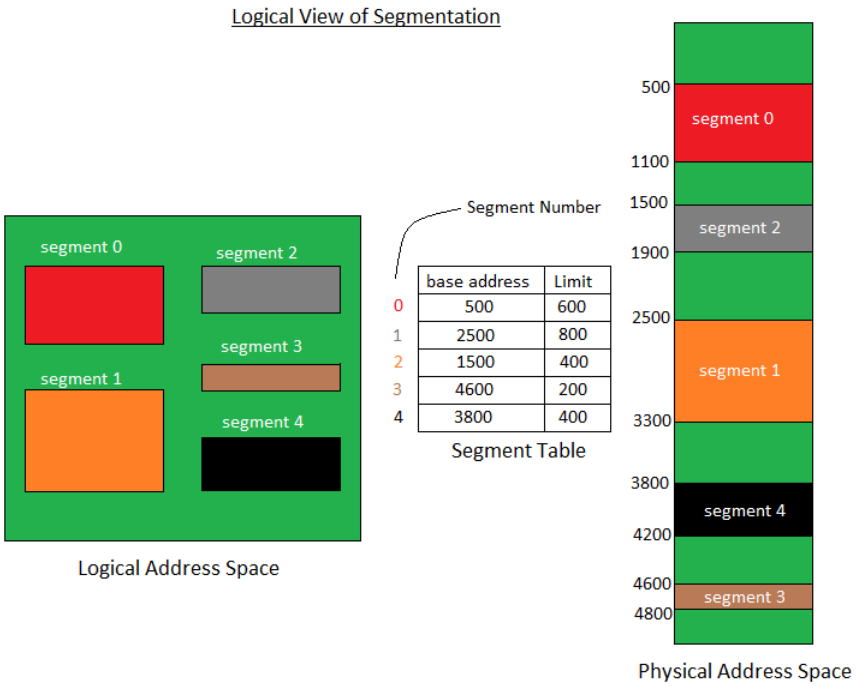


Figure 12.7.1: Segmentation Table Mapping to Physical Address.
("Segmentation Table Mapping to Physical Address" by VaibhavRai3, Geeks for Geeks is licensed under CC BY-SA 4.0)

The Segment Table maps the logical address, made up of the base address and the limit, into one-dimensional physical address. It's each table entry has:

- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Limit:** It specifies the length of the segment.

Translation of a two dimensional Logical Address to one dimensional Physical Address.

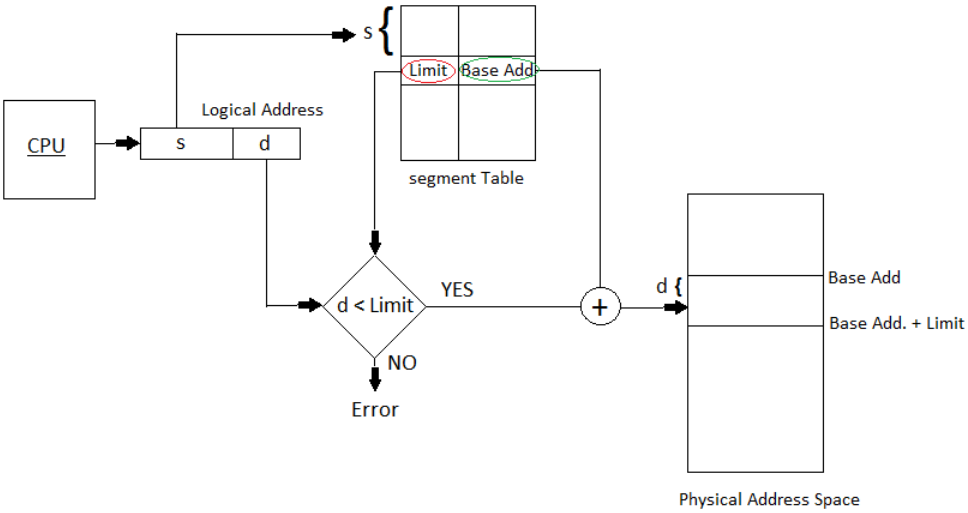


Figure 12.7.1: Translate Logical Address to Physical Address.
("Translate Logical Address to Physical Address" by VaibhavRai3, Geeks for Geeks is licensed under CC BY-SA 4.0)

Address generated by the CPU is divided into:

- **Segment number (s):** Number of bits required to represent the segment.
- **Segment offset (d):** Number of bits required to represent the size of the segment.

Walking through the diagram above:

1. CPU generates a 2 part logival address.
2. The segment number is used to get the Limit and the Base Address value from the segment table.
3. If the segment offset (d) is less than the Limit value from the segment table then
   - The Base Address returned from the segment table, points to the beginning of the segment
   - The Limit value points to the end of the segment in physcial memory.

Advantages of Segmentation

- No Internal fragmentation.
- Segment Table consumes less space in comparison to Page table in paging.

Disadvantage of Segmentation

- As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing External fragmentation.

Adapted from:
"Segmentation in Operating System" by VaibhavRai3, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 12.7: Segmentation is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

13: Virtual Memory

13.1: Memory Paging

13.1.1: Memory Paging - Page Replacement

13.2: Virtual Memory in the Operating System

This page titled 13: Virtual Memory is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

13.1: Memory Paging

Although memory paging is NOT specific to virtual memory, paging is discussed a lot in the discussion of virtual memory. So, we will spend a moment making sure we are up on the concepts we need to properly study virtual memory.

Memory Paging

In computer operating systems, memory paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory. In this scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. Paging is an important part of virtual memory implementations in modern operating systems, using secondary storage to let programs exceed the size of available physical memory.

For simplicity, main memory is called "RAM" (an acronym of "random-access memory") and secondary storage is called "disk" (a shorthand for "hard disk drive, drum memory or solid-state drive"), but the concepts do not depend on whether these terms apply literally to a specific computer system.

Page faults

When a process tries to reference a page not currently present in RAM, the processor treats this invalid memory reference as a page fault and transfers control from the program to the operating system. The operating system must:

- Determine the location of the data on disk.
- Obtain an empty page frame in RAM to use as a container for the data.
- Load the requested data into the available page frame.
- Update the page table to refer to the new page frame.
- Return control to the program, transparently retrying the instruction that caused the page fault.

When all page frames are in use, the operating system must select a page frame to reuse for the page the program now needs. If the evicted page frame was dynamically allocated by a program to hold data, or if a program modified it since it was read into RAM (in other words, if it has become "dirty"), it must be written out to disk before being freed. If a program later references the evicted page, another page fault occurs and the page must be read back into RAM.

The method the operating system uses to select the page frame to reuse, which is its page replacement algorithm, is important to efficiency. The operating system predicts the page frame least likely to be needed soon, often through the least recently used (LRU) algorithm or an algorithm based on the program's working set. To further increase responsiveness, paging systems may predict which pages will be needed soon, preemptively loading them into RAM before a program references them.

Thrashing

After completing initialization, most programs operate on a small number of code and data pages compared to the total memory the program requires. The pages most frequently accessed are called the working set.

When the working set is a small percentage of the system's total number of pages, virtual memory systems work most efficiently and an insignificant amount of computing is spent resolving page faults. As the working set grows, resolving page faults remains manageable until the growth reaches a critical point. Then faults go up dramatically and the time spent resolving them overwhelms time spent on the computing the program was written to do. This condition is referred to as thrashing. Thrashing occurs on a program that works with huge data structures, as its large working set causes continual page faults that drastically slow down the system. Satisfying page faults may require freeing pages that will soon have to be re-read from disk. "Thrashing" is also used in contexts other than virtual memory systems; for example, to describe cache issues in computing or silly window syndrome in networking.

A worst case might occur on VAX processors. A single MOVL crossing a page boundary could have a source operand using a displacement deferred addressing mode, where the longword containing the operand address crosses a page boundary, and a destination operand using a displacement deferred addressing mode, where the longword containing the operand address crosses a page boundary, and the source and destination could both cross page boundaries. This single instruction references ten pages; if not all are in RAM, each will cause a page fault. As each fault occurs the operating system needs to go through the extensive memory management routines perhaps causing multiple I/Os which might including writing other process pages to disk and reading pages

of the active process from disk. If the operating system could not allocate ten pages to this program, then remedying the page fault would discard another page the instruction needs, and any restart of the instruction would fault again.

To decrease excessive paging and resolve thrashing problems, a user can increase the number of pages available per program, either by running fewer programs concurrently or increasing the amount of RAM in the computer.

Sharing

In multi-programming or in a multi-user environment, many users may execute the same program, written so that its code and data are in separate pages. To minimize RAM use, all users share a single copy of the program. Each process's page table is set up so that the pages that address code point to the single shared copy, while the pages that address data point to different physical pages for each process.

Different programs might also use the same libraries. To save space, only one copy of the shared library is loaded into physical memory. Programs which use the same library have virtual addresses that map to the same pages (which contain the library's code and data). When programs want to modify the library's code, they use copy-on-write, so memory is only allocated when needed.

Shared memory is an efficient way of communication between programs. Programs can share pages in memory, and then write and read to exchange data.

Adapted from:
"Virtual memory" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Demand paging" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Memory paging" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Page replacement algorithm" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 13.1: Memory Paging is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

13.1.1: Memory Paging - Page Replacement

Page replacement algorithm

In a computer operating system that uses paging for virtual memory management, page replacement algorithms decide which memory pages to page out, sometimes called swap out, or write to disk, when a page of memory needs to be allocated. Page replacement happens when a requested page is not in memory (page fault) and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold.

When the page that was selected for replacement and paged out is referenced again it has to be paged in (read in from disk), and this involves waiting for I/O completion. This determines the quality of the page replacement algorithm: the less time waiting for page-ins, the better the algorithm. A page replacement algorithm looks at the limited information about accesses to the pages provided by hardware, and tries to guess which pages should be replaced to minimize the total number of page misses, while balancing this with the costs (primary storage and processor time) of the algorithm itself.

Local vs. global replacement

Replacement algorithms can be local or global.

When a process incurs a page fault, a local page replacement algorithm selects for replacement some page that belongs to that same process (or a group of processes sharing a memory partition). A global replacement algorithm is free to select any page in memory.

Local page replacement assumes some form of memory partitioning that determines how many pages are to be assigned to a given process or a group of processes. Most popular forms of partitioning are fixed partitioning and balanced set algorithms based on the working set model. The advantage of local page replacement is its scalability: each process can handle its page faults independently, leading to more consistent performance for that process. However global page replacement is more efficient on an overall system basis.

Detecting which pages are referenced and modified

Modern general purpose computers and some embedded processors have support for virtual memory. Each process has its own virtual address space. A page table maps a subset of the process virtual addresses to physical addresses. In addition, in most architectures the page table holds an "access" bit and a "dirty" bit for each page in the page table. The CPU sets the access bit when the process reads or writes memory in that page. The CPU sets the dirty bit when the process writes memory in that page. The operating system can modify the access and dirty bits. The operating system can detect accesses to memory and files through the following means:

By clearing the access bit in pages present in the process' page table. After some time, the OS scans the page table looking for pages that had the access bit set by the CPU. This is fast because the access bit it set automatically by the CPU and inaccurate because the OS does not immediately receives notice of the access nor does it have information about the order in which the process accessed these pages.

By removing pages from the process' page table without necessarily removing them from physical memory. The next access to that page is detected immediately because it causes a page fault. This is slow because a page fault involves a context switch to the OS, software lookup for the corresponding physical address, modification of the page table and a context switch back to the process and accurate because the access is detected immediately after it occurs.

Directly when the process makes system calls that potentially access the page cache like read and write in POSIX.

Precleaning

Most replacement algorithms simply return the target page as their result. This means that if target page is dirty (that is, contains data that have to be written to the stable storage before page can be reclaimed), I/O has to be initiated to send that page to the stable storage (to clean the page). In the early days of virtual memory, time spent on cleaning was not of much concern, because virtual memory was first implemented on systems with full duplex channels to the stable storage, and cleaning was customarily overlapped with paging. Contemporary commodity hardware, on the other hand, does not support full duplex transfers, and cleaning of target pages becomes an issue.

To deal with this situation, various precleaning policies are implemented. Precleaning is the mechanism that starts I/O on dirty pages that are (likely) to be replaced soon. The idea is that by the time the precleaned page is actually selected for the replacement, the I/O will complete and the page will be clean. Precleaning assumes that it is possible to identify pages that will be replaced next. Precleaning that is too eager can waste I/O bandwidth by writing pages that manage to get re-dirtied before being selected for replacement.

Demand Paging Basic concept

Demand paging follows that pages should only be brought into memory if the executing process demands them. This is often referred to as lazy evaluation as only those pages demanded by the process are swapped from secondary storage to main memory. Contrast this to pure swapping, where all memory for a process is swapped from secondary storage to main memory during the process startup.

Commonly, to achieve this process a page table implementation is used. The page table maps logical memory to physical memory. The page table uses a bitwise operator to mark if a page is valid or invalid. A valid page is one that currently resides in main memory. An invalid page is one that currently resides in secondary memory. When a process tries to access a page, the following steps are generally followed:

- Attempt to access page.
- If page is valid (in memory) then continue processing instruction as normal.
- If page is invalid then a page-fault trap occurs.
- Check if the memory reference is a valid reference to a location on secondary memory. If not, the process is terminated (illegal memory access). Otherwise, we have to page in the required page.
- Schedule disk operation to read the desired page into main memory.
- Restart the instruction that was interrupted by the operating system trap.

Advantages

Demand paging, as opposed to loading all pages immediately:

- Only loads pages that are demanded by the executing process.
- As there is more space in main memory, more processes can be loaded, reducing the context switching time, which utilizes large amounts of resources.
- Less loading latency occurs at program startup, as less information is accessed from secondary storage and less information is brought into main memory.
- As main memory is expensive compared to secondary memory, this technique helps significantly reduce the bill of material (BOM) cost in smart phones for example. Symbian OS had this feature.

Disadvantages

- Individual programs face extra latency when they access a page for the first time.
- Low-cost, low-power embedded systems may not have a memory management unit that supports page replacement.
- Memory management with page replacement algorithms becomes slightly more complex.
- Possible security risks, including vulnerability to timing attacks; see Percival, Colin (2005-05-13). "Cache missing for fun and profit" (PDF). BSDCan 2005. (specifically the virtual memory attack in section 2).
- Thrashing which may occur due to repeated page faults.

Anticipatory paging

Some systems attempt to reduce latency of Demand paging by guessing which pages not in RAM are likely to be needed soon, and pre-loading such pages into RAM, before that page is requested. (This is often in combination with pre-cleaning, which guesses which pages currently in RAM are not likely to be needed soon, and pre-writing them out to storage).

When a page fault occurs, "anticipatory paging" systems will not only bring in the referenced page, but also the next few consecutive pages (analogous to a prefetch input queue in a CPU).

The swap prefetch mechanism goes even further in loading pages (even if they are not consecutive) that are likely to be needed soon.

Adapted from:
"Virtual memory" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Demand paging" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Memory paging" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Page replacement algorithm" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 13.1.1: Memory Paging - Page Replacement is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

13.2: Virtual Memory in the Operating System

Virtual Memory Intro

In computing, virtual memory, or virtual storage is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine" which "creates the illusion to users of a very large (main) memory".

The computer's operating system, using a combination of hardware and software, maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory. Main storage, as seen by a process or task, appears as a contiguous address space or collection of contiguous segments. The operating system manages virtual address spaces and the assignment of real memory to virtual memory. Address translation hardware in the CPU, often referred to as a memory management unit (MMU), automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities to provide a virtual address space that can exceed the capacity of real memory and thus reference more memory than is physically present in the computer.

The primary benefits of virtual memory include freeing applications from having to manage a shared memory space, ability to share memory used by libraries between processes, increased security due to memory isolation, and being able to conceptually use more memory than might be physically available, using the technique of paging or segmentation.

Properties of Virtual Memory

Virtual memory makes application programming easier by hiding fragmentation of physical memory; by delegating to the kernel the burden of managing the memory hierarchy (eliminating the need for the program to handle overlays explicitly); and, when each process is run in its own dedicated address space, by obviating the need to relocate program code or to access memory with relative addressing.

Paged virtual memory

Nearly all current implementations of virtual memory divide a virtual address space into pages, blocks of contiguous virtual memory addresses. Pages on contemporary systems are usually at least 4 kilobytes in size; systems with large virtual address ranges or amounts of real memory generally use larger page sizes

Page tables

Page tables are used to translate the virtual addresses seen by the application into physical addresses used by the hardware to process instructions; such hardware that handles this specific translation is often known as the memory management unit. Each entry in the page table holds a flag indicating whether the corresponding page is in real memory or not. If it is in real memory, the page table entry will contain the real memory address at which the page is stored. When a reference is made to a page by the hardware, if the page table entry for the page indicates that it is not currently in real memory, the hardware raises a page fault exception, invoking the paging supervisor component of the operating system.

Systems can have one page table for the whole system, separate page tables for each application and segment, a tree of page tables for large segments or some combination of these. If there is only one page table, different applications running at the same time use different parts of a single range of virtual addresses. If there are multiple page or segment tables, there are multiple virtual address spaces and concurrent applications with separate page tables redirect to different real addresses.

Some earlier systems with smaller real memory sizes, such as the SDS 940, used page registers instead of page tables in memory for address translation.

Paging supervisor

This part of the operating system creates and manages page tables. If the hardware raises a page fault exception, the paging supervisor accesses secondary storage, returns the page that has the virtual address that resulted in the page fault, updates the page tables to reflect the physical location of the virtual address and tells the translation mechanism to restart the request.

When all physical memory is already in use, the paging supervisor must free a page in primary storage to hold the swapped-in page. The supervisor uses one of a variety of page replacement algorithms such as least recently used to determine which page to

free.

Pinned pages

Operating systems have memory areas that are pinned (never swapped to secondary storage). Other terms used are locked, fixed, or wired pages. For example, interrupt mechanisms rely on an array of pointers to their handlers, such as I/O completion and page fault. If the pages containing these pointers or the code that they invoke were pageable, interrupt-handling would become far more complex and time-consuming, particularly in the case of page fault interruptions. Hence, some part of the page table structures is not pageable.

Some pages may be pinned for short periods of time, others may be pinned for long periods of time, and still others may need to be permanently pinned. For example:

- The paging supervisor code and drivers for secondary storage devices on which pages reside must be permanently pinned, as otherwise paging wouldn't even work because the necessary code wouldn't be available.
- Timing-dependent components may be pinned to avoid variable paging delays.
- Data buffers that are accessed directly by peripheral devices that use direct memory access or I/O channels must reside in pinned pages while the I/O operation is in progress because such devices and the buses to which they are attached expect to find data buffers located at physical memory addresses; regardless of whether the bus has a memory management unit for I/O, transfers cannot be stopped if a page fault occurs and then restarted when the page fault has been processed.

Thrashing

When paging and page stealing are used, a problem called "thrashing" can occur, in which the computer spends an unsuitably large amount of time transferring pages to and from a backing store, hence slowing down useful work. A task's working set is the minimum set of pages that should be in memory in order for it to make useful progress. Thrashing occurs when there is insufficient memory available to store the working sets of all active programs. Adding real memory is the simplest response, but improving application design, scheduling, and memory usage can help. Another solution is to reduce the number of active tasks on the system. This reduces demand on real memory by swapping out the entire working set of one or more processes.

Segmented virtual memory

Some systems use segmentation instead of paging, dividing virtual address spaces into variable-length segments. A virtual address here consists of a segment number and an offset within the segment. Segmentation and paging can be used together by dividing each segment into pages; systems with this memory structure are usually paging-predominant, segmentation providing memory protection.

In some processors, the segments reside in a 32-bit linear, paged address space. Segments can be moved in and out of that space; pages there can "page" in and out of main memory, providing two levels of virtual memory; few if any operating systems do so, instead using only paging. Early non-hardware-assisted virtualization solutions combined paging and segmentation because paging offers only two protection domains whereas a VMM / guest OS / guest applications stack needs three. The difference between paging and segmentation systems is not only about memory division; segmentation is visible to user processes, as part of memory model semantics. Hence, instead of memory that looks like a single large space, it is structured into multiple spaces.

This difference has important consequences; a segment is not a page with variable length or a simple way to lengthen the address space. Segmentation that can provide a single-level memory model in which there is no differentiation between process memory and file system consists of only a list of segments (files) mapped into the process's potential address space.

Adapted from:
"Virtual memory" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 13.2: Virtual Memory in the Operating System is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

14: Uniprocessor CPU Scheduling

14.1: Types of Processor Scheduling
14.2: Scheduling Algorithms

This page titled 14: Uniprocessor CPU Scheduling is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

14.1: Types of Processor Scheduling

In computing, scheduling is the method by which work is assigned to resources that complete the work. The work may be virtual computation elements such as threads, processes or data flows, which are in turn scheduled onto hardware resources such as processors, network links or expansion cards.

A scheduler is what carries out the scheduling activity. Schedulers are often implemented so they keep all computer resources busy (as in load balancing), allow multiple users to share system resources effectively, or to achieve a target quality of service. Scheduling is fundamental to computation itself, and an intrinsic part of the execution model of a computer system; the concept of scheduling makes it possible to have computer multitasking with a single central processing unit (CPU).

Goals of a Scheduler

A scheduler may aim at one or more goals, for example: maximizing throughput (the total amount of work completed per time unit); minimizing wait time (time from work becoming ready until the first point it begins execution); minimizing latency or response time (time from work becoming ready until it is finished in case of batch activity, or until the system responds and hands the first output to the user in case of interactive activity); or maximizing fairness (equal CPU time to each process, or more generally appropriate times according to the priority and workload of each process). In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is measured by any one of the concerns mentioned above, depending upon the user's needs and objectives.

In real-time environments, such as embedded systems for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines; this is crucial for keeping the system stable. Scheduled tasks can also be distributed to remote devices across a network and managed through an administrative back end.

Types of operating system schedulers

The scheduler is an operating system module that selects the next jobs to be admitted into the system and the next process to run. Operating systems may feature up to three distinct scheduler types: a long-term scheduler (also known as an admission scheduler or high-level scheduler), a mid-term or medium-term scheduler, and a short-term scheduler. The names suggest the relative frequency with which their functions are performed.

Process scheduler

The process scheduler is a part of the operating system that decides which process runs at a certain point in time. It usually has the ability to pause a running process, move it to the back of the running queue and start a new process; such a scheduler is known as a preemptive scheduler, otherwise it is a cooperative scheduler.

We distinguish between "long-term scheduling", "medium-term scheduling", and "short-term scheduling" based on how often decisions must be made.

Long-term scheduling

The long-term scheduler, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue (in main memory); that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler. Thus, this scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time – whether many or few processes are to be executed concurrently, and how the split between I/O-intensive and CPU-intensive processes is to be handled. The long-term scheduler is responsible for controlling the degree of multiprogramming.

In general, most processes can be described as either I/O-bound or CPU-bound. An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that a long-term scheduler selects a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O-bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. On the other hand, if all processes are CPU-bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of

CPU-bound and I/O-bound processes. In modern operating systems, this is used to make sure that real-time processes get enough CPU time to finish their tasks.

Long-term scheduling is also important in large-scale systems such as batch processing systems, computer clusters, supercomputers, and render farms. For example, in concurrent systems, co-scheduling of interacting processes is often required to prevent them from blocking due to waiting on each other. In these cases, special-purpose job scheduler software is typically used to assist these functions, in addition to any underlying admission scheduling support in the operating system.

Some operating systems only allow new tasks to be added if it is sure all real-time deadlines can still be met. The specific heuristic algorithm used by an operating system to accept or reject new tasks is the admission control mechanism.

Medium-term scheduling

The medium-term scheduler temporarily removes processes from main memory and places them in secondary memory (such as a hard disk drive) or vice versa, which is commonly referred to as "swapping out" or "swapping in" (also incorrectly as "paging out" or "paging in"). The medium-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource.

In many systems today (those that support mapping virtual address space to secondary storage other than the swap file), the medium-term scheduler may actually perform the role of the long-term scheduler, by treating binaries as "swapped out processes" upon their execution. In this way, when a segment of the binary is required it can be swapped in on demand, or "lazy loaded", also called demand paging.

Short-term scheduling

The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes is to be executed (allocated a CPU) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers – a scheduling decision will at a minimum have to be made after every time slice, and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is unable to "force" processes off the CPU.

A preemptive scheduler relies upon a programmable interval timer which invokes an interrupt handler that runs in kernel mode and implements the scheduling function.

Dispatcher

Another component that is involved in the CPU-scheduling function is the dispatcher, which is the module that gives control of the CPU to the process selected by the short-term scheduler. It receives control in kernel mode as the result of an interrupt or system call. The functions of a dispatcher mop the following:

- Context switches, in which the dispatcher saves the state (also known as context) of the process or thread that was previously running; the dispatcher then loads the initial or previously saved state of the new process.
- Switching to user mode.
- Jumping to the proper location in the user program to restart that program indicated by its new state.

The dispatcher should be as fast as possible, since it is invoked during every process switch. During the context switches, the processor is virtually idle for a fraction of time, thus unnecessary context switches should be avoided. The time it takes for the dispatcher to stop one process and start another is known as the dispatch latency.

Adapted from:
"Scheduling (computing)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 14.1: Types of Processor Scheduling is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

14.2: Scheduling Algorithms

Scheduling Algorithms

Scheduling algorithms are used for distributing resources among parties which simultaneously and asynchronously request them. Scheduling disciplines are used in routers (to handle packet traffic) as well as in operating systems (to share CPU time among both threads and processes), disk drives (I/O scheduling), printers (print spooler), most embedded systems, etc.

The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources. Scheduling deals with the problem of deciding which of the outstanding requests is to be allocated resources. There are many different scheduling algorithms. In this section, we introduce several of them.

In packet-switched computer networks and other statistical multiplexing, the notion of a scheduling algorithm is used as an alternative to first-come first-served queuing of data packets.

The simplest best-effort scheduling algorithms are round-robin, fair queuing (a max-min fair scheduling algorithm), proportionally fair scheduling and maximum throughput. If differentiated or guaranteed quality of service is offered, as opposed to best-effort communication, weighted fair queuing may be utilized.

In advanced packet radio wireless networks such as HSDPA (High-Speed Downlink Packet Access) 3.5G cellular system, channel-dependent scheduling may be used to take advantage of channel state information. If the channel conditions are favourable, the throughput and system spectral efficiency may be increased. In even more advanced systems such as LTE, the scheduling is combined by channel-dependent packet-by-packet dynamic channel allocation, or by assigning OFDMA multi-carriers or other frequency-domain equalization components to the users that best can utilize them.

First come, first served

First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue. This is commonly used for a task queue, for example as illustrated in this section.

- Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.
- Throughput can be low, because long processes can be holding the CPU, causing the short processes to wait for a long time (known as the convoy effect).
- No starvation, because each process gets chance to be executed after a definite time.
- Turnaround time, waiting time and response time depend on the order of their arrival and can be high for the same reasons above.
- No prioritization occurs, thus this system has trouble meeting process deadlines.
- The lack of prioritization means that as long as every process eventually completes, there is no starvation. In an environment where some processes might not complete, there can be starvation.
- It is based on queuing.

Shortest remaining time first

Similar to shortest job first (SJF). With this strategy the scheduler arranges processes with the least estimated processing time remaining to be next in the queue. This requires advanced knowledge or estimations about the time required for a process to complete.

- If a shorter process arrives during another process' execution, the currently running process is interrupted (known as preemption), dividing that process into two separate computing blocks. This creates excess overhead through additional context switching. The scheduler must also place each incoming process into a specific place in the queue, creating additional overhead.
- This algorithm is designed for maximum throughput in most scenarios.
- Waiting time and response time increase as the process's computational requirements increase. Since turnaround time is based on waiting time plus processing time, longer processes are significantly affected by this. Overall waiting time is smaller than FIFO, however since no process has to wait for the termination of the longest process.

- No particular attention is given to deadlines, the programmer can only attempt to make processes with deadlines as short as possible.
- Starvation is possible, especially in a busy system with many small processes being run.
- To use this policy we should have at least two processes of different priority

Fixed priority pre-emptive scheduling

The operating system assigns a fixed priority rank to every process, and the scheduler arranges the processes in the ready queue in order of their priority. Lower-priority processes get interrupted by incoming higher-priority processes.

- Overhead is not minimal, nor is it significant.
- FPPS has no particular advantage in terms of throughput over FIFO scheduling.
- If the number of rankings is limited, it can be characterized as a collection of FIFO queues, one for each priority ranking. Processes in lower-priority queues are selected only when all of the higher-priority queues are empty.
- Waiting time and response time depend on the priority of the process. Higher-priority processes have smaller waiting and response times.
- Deadlines can be met by giving processes with deadlines a higher priority.
- Starvation of lower-priority processes is possible with large numbers of high-priority processes queuing for CPU time.

Round-robin scheduling

The scheduler assigns a fixed time unit per process, and cycles through them. If process completes within that time-slice it gets terminated otherwise it is rescheduled after giving a chance to all other processes.

- RR scheduling involves extensive overhead, especially with a small time unit.
- Balanced throughput between FCFS/ FIFO and SJF/SRTF, shorter jobs are completed faster than in FIFO and longer processes are completed faster than in SJF.
- Good average response time, waiting time is dependent on number of processes, and not average process length.
- Because of high waiting times, deadlines are rarely met in a pure RR system.
- Starvation can never occur, since no priority is given. Order of time unit allocation is based upon process arrival time, similar to FIFO.
- If Time-Slice is large it becomes FCFS /FIFO or if it is short then it becomes SJF/SRTF.

Multilevel queue scheduling

This is used for situations in which processes are easily divided into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. It is very useful for shared memory problems.

Work-conserving schedulers

A work-conserving scheduler is a scheduler that always tries to keep the scheduled resources busy, if there are submitted jobs ready to be scheduled. In contrast, a non-work conserving scheduler is a scheduler that, in some cases, may leave the scheduled resources idle despite the presence of jobs ready to be scheduled.

Choosing a scheduling algorithm

When designing an operating system, a programmer must consider which scheduling algorithm will perform best for the use the system is going to see. There is no universal "best" scheduling algorithm, and many operating systems use extended or combinations of the scheduling algorithms above.

For example, Windows NT/XP/Vista uses a multilevel feedback queue, a combination of fixed-priority preemptive scheduling, round-robin, and first in, first out algorithms. In this system, threads can dynamically increase or decrease in priority depending on if it has been serviced already, or if it has been waiting extensively. Every priority level is represented by its own queue, with round-robin scheduling among the high-priority threads and FIFO among the lower-priority ones. In this sense, response time is short for most threads, and short but critical system threads get completed very quickly. Since threads can only use one time unit of the round-robin in the highest-priority queue, starvation can be a problem for longer high-priority threads.

Adapted from:
"Scheduling (computing)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 14.2: Scheduling Algorithms is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

15: Multiprocessor Scheduling

15.1: The Question
15.2: Multiprocessor Scheduling

This page titled 15: Multiprocessor Scheduling is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

15.1: The Question

What is the purpose of multiprocessing

In computer science, multiprocessor scheduling is an optimization problem involving the scheduling of computational tasks in a multiprocessor environment. The problem statement is: "**Given a set J of jobs where job j_i has length l_i and a number of processors m, what is the minimum possible time required to schedule all jobs in J on m processors such that none overlap?**". The problem is often called the minimum makespan problem: the makespan of a schedule is defined as the time it takes the system to complete all processes, and the goal is to find a schedule that minimizes the makespan. The problem has many variants.

Approaches to Multiple-Processor Scheduling

Asymmetric multiprocessing

An asymmetric multiprocessing (AMP or ASMP) system is a multiprocessor computer system where not all of the multiple interconnected central processing units (CPUs) are treated equally. For example, a system might allow (either at the hardware or operating system level) only one CPU to execute operating system code or might allow only one CPU to perform I/O operations. Other AMP systems might allow any CPU to execute operating system code and perform I/O operations, so that they were symmetric with regard to processor roles, but attached some or all peripherals to particular CPUs, so that they were asymmetric with respect to the peripheral attachment.

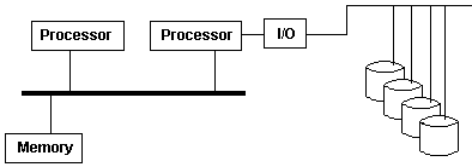


Figure 15.1.1: Asymmetric multiprocessing. ("Asmp 2.gif" by G7a, Wikimedia Commons is licensed under CC BY-SA 3.0)

Asymmetric multiprocessing was the only method for handling multiple CPUs before symmetric multiprocessing (SMP) was available. It has also been used to provide less expensive options on systems where SMP was available.

Symmetric multiprocessing

Symmetric multiprocessing (SMP) involves a multiprocessor computer hardware and software architecture where two or more identical processors are connected to a single, shared main memory, have full access to all input and output devices, and are controlled by a single operating system instance that treats all processors equally, reserving none for special purposes. Most multiprocessor systems today use an SMP architecture. In the case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.

Professor John D. Kubiatowicz considers traditionally SMP systems to contain processors without caches. Culler and Pal-Singh in their 1998 book "Parallel Computer Architecture: A Hardware/Software Approach" mention: "The term SMP is widely used but causes a bit of confusion. The more precise description of what is intended by SMP is a shared memory multiprocessor where the cost of accessing a memory location is the same for all processors; that is, it has uniform access costs when the access actually is to memory. If the location is cached, the access will be faster, but cache access times and memory access times are the same on all processors."

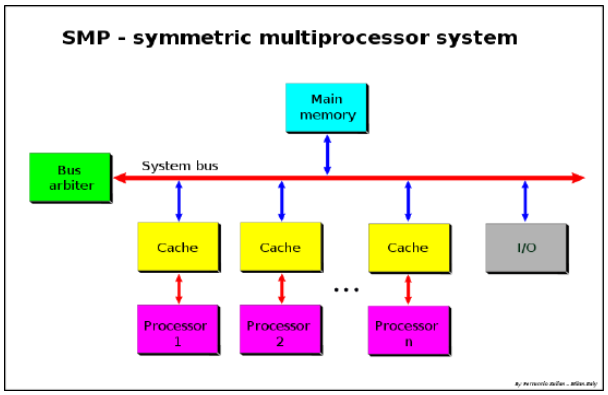


Figure 15.1.1: SMP - Symmetric Multiprocessor System. ("SMP - Symmetric Multiprocessor System" by Ferry24.Milan, Wikimedia Commons is licensed under CC BY-SA 3.0)

SMP systems are tightly coupled multiprocessor systems with a pool of homogeneous processors running independently of each other. Each processor, executing different programs and working on different sets of data, has the capability of sharing common resources (memory, I/O device, interrupt system and so on) that are connected using a system bus or a crossbar.

Adapted from:
"Asymmetric multiprocessing" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Symmetric multiprocessing" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 15.1: The Question is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

15.2: Multiprocessor Scheduling

MP Scheduling

So, we come back to the question: "Given a set J of jobs where job ji has length li and a number of processors m, what is the minimum possible time required to schedule all jobs in J on m processors such that none overlap?".

This is a complex question when we have multiple processors, some of which may run at different speeds. Scheduling is not as straight forward as it was with the single processor, the algorithms are more complex due to the nature of multiprocessors being present.

There are several different concepts that have been studied and implemented for multiprocessor thread scheduling and processor assignment. A few of these concepts are discussed below approaches seem to be well accepted:

- ### Gang scheduling

  In computer science, gang scheduling is a scheduling algorithm for parallel systems that schedules related threads or processes to run simultaneously on different processors. Usually these will be threads all belonging to the same process, but they may also be from different processes, where the processes could have a producer-consumer relationship or come from the same MPI program.

  Gang scheduling is used to ensure that if two or more threads or processes communicate with each other, they will all be ready to communicate at the same time. If they were not gang-scheduled, then one could wait to send or receive a message to another while it is sleeping, and vice versa. When processors are over-subscribed and gang scheduling is not used within a group of processes or threads which communicate with each other, each communication event could suffer the overhead of a context switch.

  Gang scheduling is based on a data structure called the Ousterhout matrix. In this matrix each row represents a time slice, and each column a processor. The threads or processes of each job are packed into a single row of the matrix. During execution, coordinated context switching is performed across all nodes to switch from the processes in one row to those in the next row.

  Gang scheduling is stricter than co-scheduling. It requires all threads of the same process to run concurrently, while co-scheduling allows for fragments, which are sets of threads that do not run concurrently with the rest of the gang.

- ### Processor affinity

  Processor affinity, or CPU pinning or "cache affinity", enables the binding and unbinding of a process or a thread to a central processing unit (CPU) or a range of CPUs, so that the process or thread will execute only on the designated CPU or CPUs rather than any CPU. This can be viewed as a modification of the native central queue scheduling algorithm in a symmetric multiprocessing operating system. Each item in the queue has a tag indicating its kin processor. At the time of resource allocation, each task is allocated to its kin processor in preference to others.

  Processor affinity takes advantage of the fact that remnants of a process that was run on a given processor may remain in that processor's state (for example, data in the cache memory) after another process was run on that processor. Scheduling that process to execute on the same processor improves its performance by reducing performance-degrading events such as cache misses. A practical example of processor affinity is executing multiple instances of a non-threaded application, such as some graphics-rendering software.

  Scheduling-algorithm implementations vary in adherence to processor affinity. Under certain circumstances, some implementations will allow a task to change to another processor if it results in higher efficiency. For example, when two processor-intensive tasks (A and B) have affinity to one processor while another processor remains unused, many schedulers will shift task B to the second processor in order to maximize processor use. Task B will then acquire affinity with the second processor, while task A will continue to have affinity with the original processor.

  #### Usage

  Processor affinity can effectively reduce cache problems, but it does not reduce the persistent load-balancing problem.[1] Also note that processor affinity becomes more complicated in systems with non-uniform architectures. For example, a system with

two dual-core hyper-threaded CPUs presents a challenge to a scheduling algorithm.

There is complete affinity between two virtual CPUs implemented on the same core via hyper-threading, partial affinity between two cores on the same physical processor (as the cores share some, but not all, cache), and no affinity between separate physical processors. As other resources are also shared, processor affinity alone cannot be used as the basis for CPU dispatching. If a process has recently run on one virtual hyper-threaded CPU in a given core, and that virtual CPU is currently busy but its partner CPU is not, cache affinity would suggest that the process should be dispatched to the idle partner CPU. However, the two virtual CPUs compete for essentially all computing, cache, and memory resources. In this situation, it would typically be more efficient to dispatch the process to a different core or CPU, if one is available. This could incur a penalty when process repopulates the cache, but overall performance could be higher as the process would not have to compete for resources within the CPU.

- ### Load balancing

In computing, load balancing refers to the process of distributing a set of tasks over a set of resources (computing units), with the aim of making their overall processing more efficient. Load balancing techniques can optimize the response time for each task, avoiding unevenly overloading compute nodes while other compute nodes are left idle.

Load balancing is the subject of research in the field of parallel computers. Two main approaches exist: static algorithms, which do not take into account the state of the different machines, and dynamic algorithms, which are usually more general and more efficient, but require exchanges of information between the different computing units, at the risk of a loss of efficiency.

Load Balancing Considerations

A load balancing algorithm always tries to answer a specific problem. Among other things, the nature of the tasks, the algorithmic complexity, the hardware architecture on which the algorithms will run as well as required error tolerance, must be taken into account. Therefore compromise must be found to best meet application-specific requirements.

Nature of tasks

The efficiency of load balancing algorithms critically depends on the nature of the tasks. Therefore, the more information about the tasks is available at the time of decision making, the greater the potential for optimization.

Size of tasks

A perfect knowledge of the execution time of each of the tasks allows to reach an optimal load distribution (see algorithm of prefix sum). Unfortunately, this is in fact an idealized case. Knowing the exact execution time of each task is an extremely rare situation.

For this reason, there are several techniques to get an idea of the different execution times. First of all, in the fortunate scenario of having tasks of relatively homogeneous size, it is possible to consider that each of them will require approximately the average execution time. If, on the other hand, the execution time is very irregular, more sophisticated techniques must be used. One technique is to add some metadata to each task. Depending on the previous execution time for similar metadata, it is possible to make inferences for a future task based on statistics.

Dependencies

In some cases, tasks depend on each other. These interdependencies can be illustrated by a directed acyclic graph. Intuitively, some tasks cannot begin until others are completed.

Assuming that the required time for each of the tasks is known in advance, an optimal execution order must lead to the minimization of the total execution time. Although this is an NP-hard problem and therefore can be difficult to be solved exactly. There are algorithms, like job scheduler, that calculate optimal task distributions using metaheuristic methods.

Segregation of tasks

Another feature of the tasks critical for the design of a load balancing algorithm is their ability to be broken down into subtasks during execution. The "Tree-Shaped Computation" algorithm presented later takes great advantage of this specificity.

Load Balancing Approaches

Static distribution with full knowledge of the tasks

If the tasks are independent of each other, and if their respective execution time and the tasks can be subdivided, there is a simple and optimal algorithm.

By dividing the tasks in such a way as to give the same amount of computation to each processor, all that remains to be done is to group the results together. Using a prefix sum algorithm, this division can be calculated in logarithmic time with respect to the number of processors.

Static load distribution without prior knowledge

Even if the execution time is not known in advance at all, static load distribution is always possible.

Round-Robin

In this simple algorithm, the first request is sent to the first server, then the next to the second, and so on down to the last. Then it is started again, assigning the next request to the first server, and so on.

This algorithm can be weighted such that the most powerful units receive the largest number of requests and receive them first.

Randomized static

Randomized static load balancing is simply a matter of randomly assigning tasks to the different servers. This method works quite well. If, on the other hand, the number of tasks is known in advance, it is even more efficient to calculate a random permutation in advance. This avoids communication costs for each assignment. There is no longer a need for a distribution master because every processor knows what task is assigned to it. Even if the number of tasks is unknown, it is still possible to avoid communication with a pseudo-random assignment generation known to all processors.

The performance of this strategy (measured in total execution time for a given fixed set of tasks) decreases with the maximum size of the tasks

Master-Worker Scheme

Master-Worker schemes are among the simplest dynamic load balancing algorithms. A master distributes the workload to all workers (also sometimes referred to as "slaves"). Initially, all workers are idle and report this to the master. The master answers worker requests and distributes the tasks to them. When he has no more tasks to give, he informs the workers so that they stop asking for tasks.

The advantage of this system is that it distributes the burden very fairly. In fact, if one does not take into account the time needed for the assignment, the execution time would be comparable to the prefix sum seen above.

The problem of this algorithm is that it has difficulty to adapt to a large number of processors because of the high amount of necessary communications. This lack of scalability makes it quickly inoperable in very large servers or very large parallel computers. The master acts as a bottleneck.

Adapted from:
"Multiprocessor scheduling" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Processor affinity" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Load balancing (computing)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Gang scheduling" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 15.2: Multiprocessor Scheduling is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

16: I/O and Disk Management

16.1: Input / Output
16.2: Types of I/O
16.3: I/O Buffering
16.4: Disk Drives in the OS
16.5: Disk Drive Scheduling
16.6: RAID

This page titled 16: I/O and Disk Management is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

16.1: Input / Output

Input Devices

In computing, an input device is a piece of equipment used to provide data and control signals to an information processing system such as a computer or information appliance. Examples of input devices include keyboards, mouse, scanners, cameras, joysticks, and microphones. Although,with advancements in technology there are numerous other input devices: network interfaces, Bluetooth devices, voice recognition, gesture recognition, a plethora of sensors, etc.

Input devices can be categorized based on:

- modality of input - mechanical motion, audio, visual, etc
- whether the input is discrete - pressing of key - or continuous - a mouse's position, though digitized into a discrete quantity, is fast enough to be considered continuous
- the number of degrees of freedom involved - two-dimensional traditional mice, or three-dimensional navigators designed for CAD applications

Output Devices

An output device is the opposite of an input device, in that the output device takes information from the computer and presents it in some manner to the user. The examples we tend to think of are the monitor/screen where we see the output, or a printer. Again, with the advances in technology we are seeing a lot more devices that meet this definition. Other output devices include: network interfaces, Bluetooth devices, the Cloud, smartphones, tablets, etc.

Input/Output

In computing, input/output, better known as I/O, is the communication between an information processing system, such as a computer, and the outside world, possibly a human or another information processing system. Inputs are the signals or data received by the system and outputs are the signals or data sent from it. The term can also be used as part of an action; to "perform I/O" is to perform an input or output operation.

I/O devices are the pieces of hardware used by a human (or other system) to communicate with a computer. For instance, a keyboard or computer mouse is an input device for a computer, while monitors and printers are output devices. Devices for communication between computers, such as modems and network cards, typically perform both input and output operations.

The designation of a device as either input or output depends on perspective. Mice and keyboards take physical movements that the human user outputs and convert them into input signals that a computer can understand; the output from these devices is the computer's input. Similarly, printers and monitors take signals that computers output as input, and they convert these signals into a representation that human users can understand. From the human user's perspective, the process of reading or seeing these representations is receiving output; this type of interaction between computers and humans is studied in the field of human–computer interaction. A further complication is that a device traditionally considered an input device, e.g., card reader, keyboard, may accept control commands to, e.g., select stacker, display keyboard lights, while a device traditionally considered as an output device may provide status data, e.g., low toner, out of paper, paper jam.

In computer architecture, the combination of the CPU and main memory, to which the CPU can read or write directly using individual instructions, is considered the brain of a computer. Any transfer of information to or from the CPU/memory combo, for example by reading data from a disk drive, is considered I/O. The CPU and its supporting circuitry may provide memory-mapped I/O that is used in low-level computer programming, such as in the implementation of device drivers, or may provide access to I/O channels. An I/O algorithm is one designed to exploit locality and perform efficiently when exchanging data with a secondary storage device, such as a disk drive.

Interface

An I/O interface is required whenever the I/O device is driven by a processor. Typically a CPU communicates with devices via a bus. The interface must have the necessary logic to interpret the device address generated by the processor. Handshaking should be implemented by the interface using appropriate commands (like BUSY, READY, and WAIT), and the processor can communicate

with an I/O device through the interface. If different data formats are being exchanged, the interface must be able to convert serial data to parallel form and vice versa. Because it would be a waste for a processor to be idle while it waits for data from an input device there must be provision for generating interrupts and the corresponding type numbers for further processing by the processor if required.

A computer that uses memory-mapped I/O accesses hardware by reading and writing to specific memory locations, using the same assembly language instructions that computer would normally use to access memory. An alternative method is via instruction-based I/O which requires that a CPU have specialized instructions for I/O. Both input and output devices have a data processing rate that can vary greatly. With some devices able to exchange data at very high speeds direct access to memory (DMA) without the continuous aid of a CPU is required.

Higher-level implementation

Higher-level operating system and programming facilities employ separate, more abstract I/O concepts and primitives. For example, most operating systems provide application programs with the concept of files. The C and C++ programming languages, and operating systems in the Unix family, traditionally abstract files and devices as streams, which can be read or written, or sometimes both. The C standard library provides functions for manipulating streams for input and output.

Adapted from:
"Input device" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Input/output" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 16.1: Input / Output is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

16.2: Types of I/O

Channel I/O

Channel I/O is a high-performance I/O architecture that is implemented in various forms on a number of computer architectures, especially on mainframe computers. In the past, channels were generally implemented with custom devices, variously named channel, I/O processor, I/O controller, I/O synchronizer, or DMA controller.

Description

Many I/O tasks can be complex and require logic to be applied to the data to convert formats and other similar duties. In these situations, the simplest solution is to ask the CPU to handle the logic, but because I/O devices are relatively slow, a CPU could waste time (in computer perspective) waiting for the data from the device. This situation is called 'I/O bound'.

Channel architecture avoids this problem by using a logically independent, low-cost facility. Channel processors are simple, but self-contained, with minimal logic and sufficient scratch-pad memory (working storage) to handle I/O tasks. They are typically not powerful or flexible enough to be used as a computer on their own and can be construed as a form of coprocessor. On some systems the channels use memory or registers addressable by the central processor as their scratchpad memory, while on other systems it is present in the channel hardware. Typically there are standard interfaces between channels and external peripheral devices, and multiple channels can operate concurrently.

A CPU typically designates a block of storage or sends a relatively small channel program to the channel in order to handle I/O tasks, which the channel and controller can, in many cases, complete without further intervention from the CPU (exception: those channel programs which utilize 'program controlled interrupts', PCIs, to facilitate program loading, demand paging and other essential system tasks).

When I/O transfer is complete or an error is detected, the controller communicates with the CPU through the channel using an interrupt. Since the channel has direct access to the main memory, it is also often referred to as a direct memory access (DMA) controller.

In the most recent implementations, the channel program is initiated and the channel processor performs all required processing until either an ending condition. This eliminates much of the CPU—Channel interaction and greatly improves overall system performance. The channel may report several different types of ending conditions, which may be unambiguously normal, may unambiguously indicate an error or whose meaning may depend on the context and the results of a subsequent sense operation. In some systems an I/O controller can request an automatic retry of some operations without CPU intervention. In earlier implementations, any error, no matter how small, required CPU intervention, and the overhead was, consequently, much higher. A program-controlled interruption (PCI) is still supported for certain "legacy" operations, but the trend is to move away from such PCIs, except where unavoidable.

Memory-mapped I/O

Memory-mapped I/O (MMIO) and port-mapped I/O (PMIO) are two complementary methods of performing input/output (I/O) between the central processing unit (CPU) and peripheral devices in a computer. These are both alternative approach the channel based I/O discussed above.

Memory-mapped I/O uses the same address space to address both memory and I/O devices. The memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is accessed by the CPU, it may refer to a portion of physical RAM, or it can instead refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register. To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory. The reservation may be permanent, or temporary (as achieved via bank switching). An example of the latter is found in the Commodore 64, which uses a form of memory mapping to cause RAM or I/O hardware to appear in the 0xD000-0xDFFF range.

Description

Different CPU-to-device communication methods, such as memory mapping, do not affect the direct memory access (DMA) for a device, because, by definition, DMA is a memory-to-device communication method that bypasses the CPU.

Hardware interrupts are another communication method between the CPU and peripheral devices, however, for a number of reasons, interrupts are always treated separately. An interrupt is device-initiated, as opposed to the methods mentioned above, which are CPU-initiated. It is also unidirectional, as information flows only from device to CPU. Lastly, each interrupt line carries only one bit of information with a fixed meaning, namely "an event that requires attention has occurred in a device on this interrupt line".

I/O operations can slow memory access if the address and data buses are shared. This is because the peripheral device is usually much slower than main memory. In some architectures, port-mapped I/O operates via a dedicated I/O bus, alleviating the problem.

One merit of memory-mapped I/O is that, by discarding the extra complexity that port I/O brings, a CPU requires less internal logic and is thus cheaper, faster, easier to build, consumes less power and can be physically smaller; this follows the basic tenets of reduced instruction set computing, and is also advantageous in embedded systems. The other advantage is that, because regular memory instructions are used to address devices, all of the CPU's addressing modes are available for the I/O as well as the memory, and instructions that perform an ALU operation directly on a memory operand (loading an operand from a memory location, storing the result to a memory location, or both) can be used with I/O device registers as well. In contrast, port-mapped I/O instructions are often very limited, often providing only for simple load-and-store operations between CPU registers and I/O ports, so that, for example, to add a constant to a port-mapped device register would require three instructions: read the port to a CPU register, add the constant to the CPU register, and write the result back to the port.

As 16-bit processors have become obsolete and replaced with 32-bit and 64-bit in general use, reserving ranges of memory address space for I/O is less of a problem, as the memory address space of the processor is usually much larger than the required space for all memory and I/O devices in a system. Therefore, it has become more frequently practical to take advantage of the benefits of memory-mapped I/O. However, even with address space being no longer a major concern, neither I/O mapping method is universally superior to the other, and there will be cases where using port-mapped I/O is still preferable.

Memory-mapped I/O is preferred in x86-based architectures because the instructions that perform port-based I/O are limited to one register: EAX, AX, and AL are the only registers that data can be moved into or out of, and either a byte-sized immediate value in the instruction or a value in register DX determines which port is the source or destination port of the transfer. Since any general-purpose register can send or receive data to or from memory and memory-mapped I/O devices, memory-mapped I/O uses fewer instructions and can run faster than port I/O. AMD did not extend the port I/O instructions when defining the x86-64 architecture to support 64-bit ports, so 64-bit transfers cannot be performed using port I/O.

Port-mapped I/O

Port-mapped I/O often uses a special class of CPU instructions designed specifically for performing I/O, such as the in and out instructions found on microprocessors based on the x86 and x86-64 architectures. Different forms of these two instructions can copy one, two or four bytes (outb, outw and outl, respectively) between the EAX register or one of that register's subdivisions on the CPU and a specified I/O port which is assigned to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire bus dedicated to I/O. Because the address space for I/O is isolated from that for main memory, this is sometimes referred to as isolated I/O.

Direct Memory Access

Direct memory access (DMA) is a feature of computer systems that allows certain hardware subsystems to access main system memory (random-access memory) independent of the central processing unit (CPU).

Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work. With DMA, the CPU first initiates the transfer, then it does other operations while the transfer is in progress, and it finally receives an interrupt from the DMA controller (DMAC) when the operation is done. This feature is useful at any time that the CPU cannot keep up with the rate of data transfer, or when the CPU needs to perform work while waiting for a relatively slow I/O data transfer. Many hardware systems use DMA, including disk drive

controllers, graphics cards, network cards and sound cards. DMA is also used for intra-chip data transfer in multi-core processors. Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without DMA channels. Similarly, a processing element inside a multi-core processor can transfer data to and from its local memory without occupying its processor time, allowing computation and data transfer to proceed in parallel.

DMA can also be used for "memory to memory" copying or moving of data within memory. DMA can offload expensive memory operations, such as large copies or scatter-gather operations, from the CPU to a dedicated DMA engine. An implementation example is the I/O Acceleration Technology. DMA is of interest in network-on-chip and in-memory computing architectures.

Description

There are three common modes DMA uses, they are described below.

Burst mode

In burst mode, an entire block of data is transferred in one contiguous sequence. Once the DMA controller is granted access to the system bus by the CPU, it transfers all bytes of data in the data block before releasing control of the system buses back to the CPU, but renders the CPU inactive for relatively long periods of time. The mode is also called "Block Transfer Mode".

Cycle stealing mode

The cycle stealing mode is used in systems in which the CPU should not be disabled for the length of time needed for burst transfer modes. In the cycle stealing mode, the DMA controller obtains access to the system bus the same way as in burst mode, using BR (Bus Request) and BG (Bus Grant) signals, which are the two signals controlling the interface between the CPU and the DMA controller. However, in cycle stealing mode, after one byte of data transfer, the control of the system bus is deasserted to the CPU via BG. It is then continually requested again via BR, transferring one byte of data per request, until the entire block of data has been transferred. By continually obtaining and releasing the control of the system bus, the DMA controller essentially interleaves instruction and data transfers. The CPU processes an instruction, then the DMA controller transfers one data value, and so on. On the one hand, the data block is not transferred as quickly in cycle stealing mode as in burst mode, but on the other hand the CPU is not idled for as long as in burst mode. Cycle stealing mode is useful for controllers that monitor data in real time.

Transparent mode

Transparent mode takes the most time to transfer a block of data, yet it is also the most efficient mode in terms of overall system performance. In transparent mode, the DMA controller transfers data only when the CPU is performing operations that do not use the system buses. The primary advantage of transparent mode is that the CPU never stops executing its programs and the DMA transfer is free in terms of time, while the disadvantage is that the hardware needs to determine when the CPU is not using the system buses, which can be complex. This is also called "Hidden DMA data transfer mode".

Adapted from:
"Memory-mapped I/O" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Channel I/O" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Direct memory access" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 16.2: Types of I/O is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

16.3: I/O Buffering

What is a Buffer

A data buffer (or just buffer) is a region of a physical memory storage used to temporarily store data while it is being moved from one place to another. Typically, the data is stored in a buffer as it is retrieved from an input device (such as a microphone) or just before it is sent to an output device (such as speakers). However, a buffer may be used when moving data between processes within a computer. This is comparable to buffers in telecommunication. Buffers can be implemented in a fixed memory location in hardware—or by using a virtual data buffer in software, pointing at a location in the physical memory. In all cases, the data stored in a data buffer are stored on a physical storage medium. A majority of buffers are implemented in software, which typically use the faster RAM to store temporary data, due to the much faster access time compared with hard disk drives. Buffers are typically used when there is a difference between the rate at which data is received and the rate at which it can be processed, or in the case that these rates are variable, for example in a printer spooler or in online video streaming. In the distributed computing environment, data buffer is often implemented in the form of burst buffer that provides distributed buffering service.

Uses of Buffers

Buffers are often used in conjunction with I/O to hardware, such as disk drives, sending or receiving data to or from a network, or playing sound on a speaker. A line to a rollercoaster in an amusement park shares many similarities. People who ride the coaster come in at an unknown and often variable pace, but the roller coaster will be able to load people in bursts (as a coaster arrives and is loaded). The queue area acts as a buffer—a temporary space where those wishing to ride wait until the ride is available. Buffers are usually used in a FIFO (first in, first out) method, outputting data in the order it arrived.

Buffers can increase application performance by allowing synchronous operations such as file reads or writes to complete quickly instead of blocking while waiting for hardware interrupts to access a physical disk subsystem; instead, an operating system can immediately return a successful result from an API call, allowing an application to continue processing while the kernel completes the disk operation in the background. Further benefits can be achieved if the application is reading or writing small blocks of data that do not correspond to the block size of the disk subsystem, allowing a buffer to be used to aggregate many smaller read or write operations into block sizes that are more efficient for the disk subsystem, or in the case of a read, sometimes to completely avoid having to physically access a disk.

Types of Buffers

There are several common types of buffering used. We will talk about just a couple.

Single I/O Buffer

The simplest form of buffering is using a single buffer implementation. As a user process executes, it issues an I/O request, This causes the OS to assign a buffer in the system portion of main memory to the I/O operation. The OS then send the request to the proper device. The I/O device responds by sending data in response to the request. The OS simply writes the data into the buffer the OS had previously created. When te buffer gets full, or at some OS specified timed interval, the data is transferred to the process in a single transfer.

Figure 16.3.1: Single I/O Buffering. ("Single I/O Buffering" by Pbmacsodak, Wikimedia is licensed under CC BY-SA 4.0)

This process can operate in the other direction as well. The process can be creating data that needs to be stored on a disk drive - or to some other output device. As the process creates the data, it is stored in the buffer, until the device is free to accept the data, at which time, the OS sends the contents of the buffer to the output device.

Double I/O Buffer

This concept simply adds a second buffer to be used to increase efficiency. When the first buffer gets full, the OS immediately switches to the other buffer. As the second buffer is filled, the OS is also moving the data out of the first buffer. So, by the time the first buffer is needed again it is empty, and the OS then begins filling it again, and the OS empties the other buffer.

Figure 16.3.1: Double I/O Buffering. ("Double I/O Buffering" by Pbmacsodak, Wikimedia is licensed under CC BY-SA 4.0)

Again, this concept works with either a read request or a write request from the user process. The example aboe is showing a read request, but the same process takes place on a write request.

Circular Buffer

A circular buffer first starts out empty and has a set length. In the diagram below is a 7-element buffer:

Figure 16.3.1: Circular Buffer - Empty. ("Circular_buffer_-_empty.svg" by I, Cburnett, Wikimedia is licensed under CC BY-SA 3.0)

Assume that 1 is written in the center of a Circular Buffer (the exact starting location is not important in a Circular Buffer):

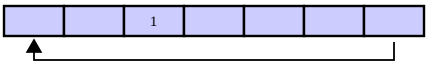


Figure 16.3.1: Circular Buffer 1. ("Circular_buffer_-_XX1XXXX.svg" by I, Cburnett, Wikimedia is licensed under CC BY-SA 3.0)

Then assume that two more elements are added to the Circular Buffer — 2 & 3 — which get put after 1:

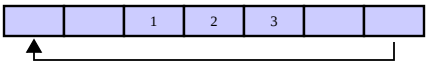


Figure 16.3.1: Circular Buffer 123. ("Circular_buffer_-_XX123XXXX.svg" by I, Cburnett, Wikimedia is licensed under CC BY-SA 3.0)

If two elements are removed, the two oldest values inside of the Circular Buffer would be removed. Circular Buffers use FIFO (First In, First Out) logic. In the example 1 & 2 were the first to enter the Circular Buffer, they are the first to be removed, leaving 3 inside of the Buffer.

Figure 16.3.1: Circular Buffer 3. ("Circular_buffer_-_XXXX3XX.svg" by I, Cburnett, Wikimedia is licensed under CC BY-SA 3.0)

If the buffer has 7 elements, then it is completely full:

Figure 16.3.1: Circular Buffer 6789345. ("Circular_buffer_-_6789345.svg" by I, Cburnett, Wikimedia is licensed under CC BY-SA 3.0)

A property of the circular buffer is that when it is full and a subsequent write is performed, then it starts overwriting the oldest data. In the current example, two more elements — A & B — are added and they *overwrite* the 3 & 4:

Figure 16.3.1: Circular Buffer 6789AB5. ("Circular_buffer_-_6789AB5.svg" by I, Cburnett, Wikimedia is licensed under CC BY-SA 3.0)

Alternatively, the routines that manage the buffer could prevent overwriting the data and return an error or raise an exception. Whether or not data is overwritten is up to the semantics of the buffer routines or the application using the circular buffer.

Finally, if two elements are now removed then what would be returned is **not** 3 & 4 but 5 & 6 because A & B overwrote the 3 & the 4 yielding the buffer with:

Figure 16.3.1: Circular Buffer 789AB. ("Circular_buffer_-_X789ABX.svg" by I, Cburnett, Wikimedia is licensed under CC BY-SA 3.0)

Uses

Adapted from:

"Data buffer" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

"Circular buffer" by **Multiple Contributors**, Wikipedia is licensed under CC BY-SA 3.0

This page titled 16.3: I/O Buffering is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

16.4: Disk Drives in the OS

Hard Disks

A hard disk drive (HDD), hard disk, hard drive, or fixed disk is an electro-mechanical data storage device that stores and retrieves digital data using magnetic storage and one or more rigid rapidly rotating platters coated with magnetic material. The platters are paired with magnetic heads, usually arranged on a moving actuator arm, which read and write data to the platter surfaces. Data is accessed in a random-access manner, meaning that individual blocks of data can be stored and retrieved in any order. HDDs are a type of non-volatile storage, retaining stored data even when powered off.

Introduced by IBM in 1956, HDDs were the dominant secondary storage device for general-purpose computers beginning in the early 1960s. HDDs maintained this position into the modern era of servers and personal computers, though personal computing devices produced in large volume, like cell phones and tablets, rely on flash memory storage devices. More than 224 companies have produced HDDs historically, though after extensive industry consolidation most units are manufactured by Seagate, Toshiba, and Western Digital. HDDs dominate the volume of storage produced (exabytes per year) for servers. Though production is growing slowly (by exabytes shipped), sales revenues and unit shipments are declining because solid-state drives (SSDs) have higher data-transfer rates, higher areal storage density, better reliability, and much lower latency and access times.

Performance

The are several factors which impact the performance of a disk drive. A drive's performance has an impact on the over all performance of the operating system, since much of the computer's processing does interface with the disk drive. The factors that limit the time to access the data on an HDD are mostly related to the mechanical nature of the rotating disks and moving heads.

Access Time

The access time or response time of a rotating drive is a measure of the time it takes before the drive can actually transfer data. The factors that control this time on a rotating drive are mostly related to the mechanical nature of the rotating disks and moving heads. It is composed of a few independently measurable elements that are added together to get a single value when evaluating the performance of a storage device. The access time can vary significantly, so it is typically provided by manufacturers or measured in benchmarks as an average.

The key components that are typically added together to obtain the access time are:

- Seek time
- Rotational latency
- Command processing time
- Settle time

Seek time

With rotating drives, the seek time measures the time it takes the head assembly on the actuator arm to travel to the track of the disk where the data will be read or written. The data on the media is stored in sectors which are arranged in parallel circular tracks (concentric or spiral depending upon the device type) and there is an actuator with an arm that suspends a head that can transfer data with that media. When the drive needs to read or write a certain sector it determines in which track the sector is located. It then uses the actuator to move the head to that particular track. If the initial location of the head was the desired track then the seek time would be zero. If the initial track was the outermost edge of the media and the desired track was at the innermost edge then the seek time would be the maximum for that drive. Seek times are not linear compared with the seek distance traveled because of factors of acceleration and deceleration of the actuator arm.

Rotational Latency

Rotational latency (sometimes called rotational delay or just latency) is the delay waiting for the rotation of the disk to bring the required disk sector under the read-write head. It depends on the rotational speed of a disk (or spindle motor), measured in revolutions per minute (RPM). For most magnetic media-based drives, the average rotational latency is typically based on the empirical relation that the average latency in milliseconds for such a drive is one-half the rotational period. Maximum rotational

latency is the time it takes to do a full rotation excluding any spin-up time (as the relevant part of the disk may have just passed the head when the request arrived)

Other

The command processing time or command overhead is the time it takes for the drive electronics to set up the necessary communication between the various components in the device so it can read or write the data. This is of the order of 3 µs, very much less than other overhead times, so it is usually ignored when benchmarking hardware.

The settle time is the time it takes the heads to settle on the target track and stop vibrating so they do not read or write off track. This time is usually very small, typically less than 100 µs, and modern HDD manufacturers account for it in their seek time specifications

Data Transfer Rate

The data transfer rate of a drive (also called throughput) covers both the internal rate (moving data between the disk surface and the controller on the drive) and the external rate (moving data between the controller on the drive and the host system). The measurable data transfer rate will be the lower (slower) of the two rates. The sustained data transfer rate or sustained throughput of a drive will be the lower of the sustained internal and sustained external rates. The sustained rate is less than or equal to the maximum or burst rate because it does not have the benefit of any cache or buffer memory in the drive. The internal rate is further determined by the media rate, sector overhead time, head switch time, and cylinder switch time.

Solid State Drives

Over time, the performance gap between the central processing units (CPUs) and electromechanical storage (hard disk drives and their RAID setups) widened, requiring advancements in the secondary storage technology. A solution was found in flash memory, which is an electronic non-volatile computer storage media that can be electrically erased and reprogrammed. Solid-state storage typically uses the NAND type of flash memory, which may be written and read in chunks much smaller than the entire size of the storage device. The size of a minimal chunk (page) for read operations is much smaller than the minimal chunk size (block) for write/erase operations, resulting in an undesirable phenomenon called write amplification that limits the random write performance and write endurance of flash-based solid-state storage devices. Another type of solid-state storage devices uses volatile random-access memory (RAM) combined with a battery that allows the contents of RAM to be preserved for a limited amount of time after the device's power supply is interrupted. As an advantage, RAM-based solid-state storage is much faster compared to flash, and does not experience write amplification.

As a result of having no moving mechanical parts, solid-state storage virtually eliminates the data access latencies present in electromechanical storage devices, and allows significantly higher rates of I/O operations per second (IOPS). Additionally, solid-state storage allows much faster sequential access to stored data, consumes less power, has better physical shock resistance, and produces less heat and no vibrations during operation. As a downside, solid-state storage devices have much higher per-megabyte prices than electromechanical storage devices, and generally come in significantly smaller per-device capacities. Moreover, flash-based devices experience the memory wear that reduces their service life by imposing a limited amount of data that may be written to them, resulting from the limitations of flash memory that impose a finite number of program–erase cycles used to write data. As a result, solid-state storage is frequently used for the creation of hybrid drives, in which solid-state storage serves as a cache for frequently accessed data instead of being a complete substitute for the traditional secondary storage.

Adapted from:
"Hard disk drive" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Hard disk drive performance characteristics" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

"Solid-state storage" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 16.4: Disk Drives in the OS is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

16.5: Disk Drive Scheduling

Input/output (I/O) scheduling is the method that computer operating systems use to decide in which order the block I/O operations will be submitted to storage volumes. I/O scheduling is sometimes called disk scheduling.

I/O scheduling

I/O scheduling usually has to work with hard disk drives that have long access times for requests placed far away from the current position of the disk head (this operation is called a seek). To minimize the effect this has on system performance, most I/O schedulers implement a variant of the elevator algorithm that reorders the incoming randomly ordered requests so the associated data would be accessed with minimal arm/head movement.

I/O schedulers can have many purposes depending on the goals; common purposes include the following

- To minimize time wasted by hard disk seeks
- To prioritize a certain processes' I/O requests
- To give a share of the disk bandwidth to each running process
- To guarantee that certain requests will be issued before a particular deadline

Disk Scheduling Algorithms

First in First Out (also known as First-Come First Served - FCFS)

In computing and in systems theory, FIFO (an acronym for first in, first out) is a method for organizing the manipulation of a data structure (often, specifically a data buffer) where the oldest (first) entry, or "head" of the queue, is processed first.

Such processing is analogous to servicing people in a queue area on a first-come, first-served basis, in the same sequence in which they had arrived at the queue's tail.

The FIFO algorithm is also an operating system scheduling algorithm, which gives every process central processing unit (CPU) time in the order in which it is demanded. FIFO's opposite is LIFO, last-in-first-out, where the youngest entry or "top of the stack" is processed first. A priority queue is neither FIFO or LIFO but may adopt similar behavior temporarily or by default. Queuing theory encompasses these methods for processing data structures, as well as interactions between strict-FIFO queues.

Shortest seek first

This is a direct improvement upon a first-come first-served (FCFS) algorithm. The drive maintains an incoming buffer of requests, and tied with each request is a cylinder number of the request. Lower cylinder numbers indicate that the cylinder is closer to the spindle, while higher numbers indicate the cylinder is farther away. The shortest seek first algorithm determines which request is closest to the current position of the head, and then services that request next.

The shortest seek first algorithm has the direct benefit of simplicity and is clearly advantageous in comparison to the FIFO method, in that overall arm movement is reduced, resulting in lower average response time.

However, since the buffer is always getting new requests, these can skew the service time of requests that may be farthest away from the disk head's current location, if the new requests are all close to the current location; in fact, starvation may result, with the faraway requests never being able to make progress.

The elevator algorithm is one way of reducing arm movement/response time, and ensuring consistent servicing of requests.

Anticipatory scheduling

"Deceptive idleness" is a situation where a process appears to be finished reading from the disk when it is actually processing data in preparation of the next read operation. This will cause a normal work-conserving I/O scheduler to switch to servicing I/O from an unrelated process. This situation is detrimental to the throughput of synchronous reads, as it degenerates into a seeking workload. Anticipatory scheduling overcomes deceptive idleness by pausing for a short time (a few milliseconds) after a read operation in anticipation of another close-by read requests.

Anticipatory scheduling yields significant improvements in disk utilization for some workloads. In some situations the Apache web server may achieve up to 71% more throughput from using anticipatory scheduling.

The Linux anticipatory scheduler may reduce performance on disks using Tagged Command Queuing (TCQ), high performance disks, and hardware RAID arrays. An anticipatory scheduler (AS) was the default Linux kernel scheduler between 2.6.0 and 2.6.18, by which time it was replaced by the CFQ scheduler.

As of kernel version 2.6.33, the Anticipatory scheduler has been removed from the Linux kernel. The reason being that while useful, the scheduler's effects could be achieved through tuned use of other schedulers (mostly CFQ, which can also be configured to idle with the slice_idle tunable). Since the anticipatory scheduler added maintenance overhead while not improving the workload coverage of the Linux kernel, it was deemed redundant.

Adapted from:
"FIFO (computing and electronics)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Shortest seek first" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Anticipatory scheduling" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 16.5: Disk Drive Scheduling is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

16.6: RAID

RAID

RAID - "Redundant Array of Inexpensive Disks" or "Redundant Array of Independent Disks") is a data storage virtualization technology that combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy, performance improvement, or both. This was in contrast to the previous concept of highly reliable mainframe disk drives referred to as "single large expensive disk"

Data is distributed across the drives in one of several ways, referred to as RAID levels, depending on the required level of redundancy and performance. The different schemes, or data distribution layouts, are named by the word "RAID" followed by a number, for example RAID 0 or RAID 1. Each scheme, or RAID level, provides a different balance among the key goals: reliability, availability, performance, and capacity. RAID levels greater than RAID 0 provide protection against unrecoverable sector read errors, as well as against failures of whole physical drives.

RAID Levels

Originally, there were five standard levels of RAID, but many variations have evolved, including several nested levels and many non-standard levels (mostly proprietary). RAID levels and their associated data formats are standardized by the Storage Networking Industry Association (SNIA) in the Common RAID Disk Drive Format (DDF) standard:

RAID 0 consists of striping, but no mirroring or parity. Compared to a spanned volume, the capacity of a RAID 0 volume is the same; it is the sum of the capacities of the drives in the set. But because striping distributes the contents of each file among all drives in the set, the failure of any drive causes the entire RAID 0 volume and all files to be lost. In comparison, a spanned volume preserves the files on the unfailing drives. The benefit of RAID 0 is that the throughput of read and write operations to any file is multiplied by the number of drives because, unlike spanned volumes, reads and writes are done concurrently. The cost is increased vulnerability to drive failures—since any drive in a RAID 0 setup failing causes the entire volume to be lost, the average failure rate of the volume rises with the number of attached drives.

RAID 1 consists of data mirroring, without parity or striping. Data is written identically to two or more drives, thereby producing a "mirrored set" of drives. Thus, any read request can be serviced by any drive in the set. If a request is broadcast to every drive in the set, it can be serviced by the drive that accesses the data first (depending on its seek time and rotational latency), improving performance. Sustained read throughput, if the controller or software is optimized for it, approaches the sum of throughputs of every drive in the set, just as for RAID 0. Actual read throughput of most RAID 1 implementations is slower than the fastest drive. Write throughput is always slower because every drive must be updated, and the slowest drive limits the write performance. The array continues to operate as long as at least one drive is functioning.

RAID 2 consists of bit-level striping with dedicated Hamming-code parity. All disk spindle rotation is synchronized and data is striped such that each sequential bit is on a different drive. Hamming-code parity is calculated across corresponding bits and stored on at least one parity drive. This level is of historical significance only; although it was used on some early machines (for example, the Thinking Machines CM-2), as of 2014 it is not used by any commercially available system.

RAID 3 consists of byte-level striping with dedicated parity. All disk spindle rotation is synchronized and data is striped such that each sequential byte is on a different drive. Parity is calculated across corresponding bytes and stored on a dedicated parity drive. [Although implementations exist, RAID 3 is not commonly used in practice.

RAID 4 consists of block-level striping with dedicated parity. This level was previously used by NetApp, but has now been largely replaced by a proprietary implementation of RAID 4 with two parity disks, called RAID-DP. The main advantage of RAID 4 over RAID 2 and 3 is I/O parallelism: in RAID 2 and 3, a single read I/O operation requires reading the whole group of data drives, while in RAID 4 one I/O read operation does not have to spread across all data drives. As a result, more I/O operations can be executed in parallel, improving the performance of small transfers.

RAID 5 consists of block-level striping with distributed parity. Unlike RAID 4, parity information is distributed among the drives, requiring all drives but one to be present to operate. Upon failure of a single drive, subsequent reads can be calculated from the distributed parity such that no data is lost. RAID 5 requires at least three disks. Like all single-parity concepts, large RAID 5

implementations are susceptible to system failures because of trends regarding array rebuild time and the chance of drive failure during rebuild (see "Increasing rebuild time and failure probability" section, below). Rebuilding an array requires reading all data from all disks, opening a chance for a second drive failure and the loss of the entire array.

RAID 6 consists of block-level striping with double distributed parity. Double parity provides fault tolerance up to two failed drives. This makes larger RAID groups more practical, especially for high-availability systems, as large-capacity drives take longer to restore. RAID 6 requires a minimum of four disks. As with RAID 5, a single drive failure results in reduced performance of the entire array until the failed drive has been replaced. With a RAID 6 array, using drives from multiple sources and manufacturers, it is possible to mitigate most of the problems associated with RAID 5. The larger the drive capacities and the larger the array size, the more important it becomes to choose RAID 6 instead of RAID 5. RAID 10 also minimizes these problems.

Adapted from:
"RAID" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 16.6: RAID is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

CHAPTER OVERVIEW

17: File Management

17.1: Overview

17.2: Files

17.2.1: Files (continued)

17.3: Directory

17.4: File Sharing

This page titled 17: File Management is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

17.1: Overview

File system

A file system controls how data is stored and retrieved. Without a file system, data placed in a storage medium would be one large body of data with no way to tell where one piece of data stops and the next begins. By separating the data into pieces and giving each piece a name, the data is easily isolated and identified. Taking its name from the way paper-based data management system is named, each group of data is called a "file." The structure and logic rules used to manage the groups of data and their names is called a "file system."

There are many different kinds of file systems. Each one has different structure and logic, properties of speed, flexibility, security, size and more. Some file systems have been designed to be used for specific applications. For example, the ISO 9660 file system is designed specifically for optical disks.

File systems can be used on numerous different types of storage devices that use different kinds of media. As of 2019, hard disk drives have been key storage devices and are projected to remain so for the foreseeable future. Other kinds of media that are used include SSDs, magnetic tapes, and optical disks. In some cases, such as with tmpfs, the computer's main memory (random-access memory, RAM) is used to create a temporary file system for short-term use.

Some file systems are used on local data storage devices; others provide file access via a network protocol (for example, NFS, SMB, or 9P clients). Some file systems are "virtual", meaning that the supplied "files" (called virtual files) are computed on request (such as procfs and sysfs) or are merely a mapping into a different file system used as a backing store. The file system manages access to both the content of files and the metadata about those files. It is responsible for arranging storage space; reliability, efficiency, and tuning with regard to the physical storage medium are important design considerations.

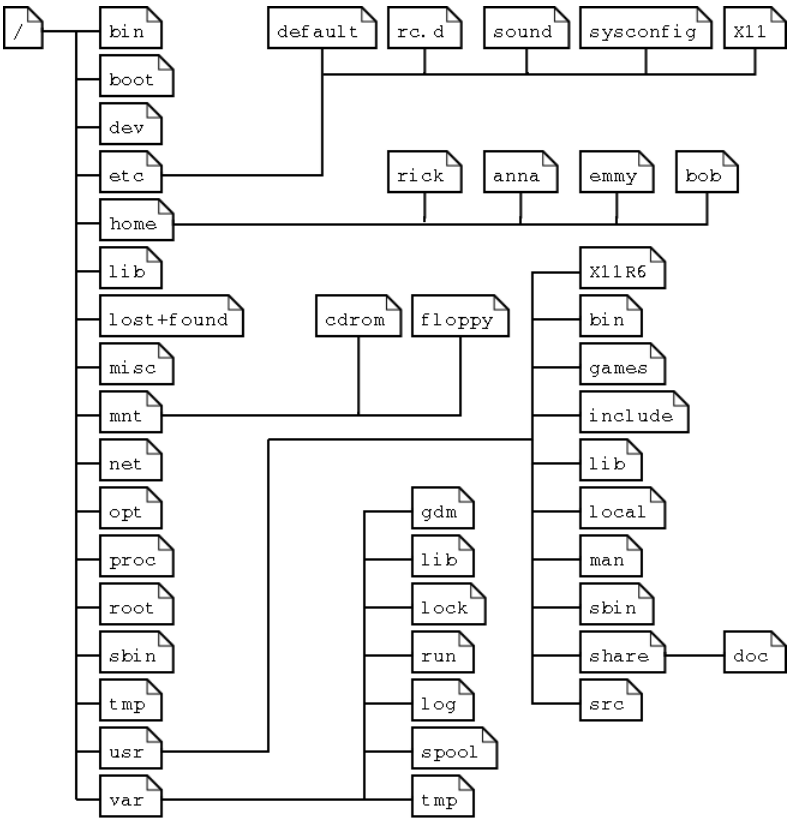


Figure 17.1.1: **Linux file system layout**. ("Linux file system layout" by Machtelt Garrels, The Linux Documentation Project is licensed under LDP Manifesto)

Architecture

A file system consists of two or three layers. Sometimes the layers are explicitly separated, and sometimes the functions are combined.

The logical file system is responsible for interaction with the user application. It provides the application program interface (API) for file operations — OPEN, CLOSE, READ, etc., and passes the requested operation to the layer below it for processing. The logical file system "manage open file table entries and per-process file descriptors". This layer provides "file access, directory operations, security and protection".

The second optional layer is the virtual file system. "This interface allows support for multiple concurrent instances of physical file systems, each of which is called a file system implementation".

The third layer is the physical file system. This layer is concerned with the physical operation of the storage device (e.g. disk). It processes physical blocks being read or written. It handles buffering and memory management and is responsible for the physical placement of blocks in specific locations on the storage medium. The physical file system interacts with the device drivers or with the channel to drive the storage device.

Aspects of file systems

Space management

File systems allocate space in a granular manner, usually multiple physical units on the device. The file system is responsible for organizing files and directories, and keeping track of which areas of the media belong to which file and which are not being used. This results in unused space when a file is not an exact multiple of the allocation unit, sometimes referred to as *slack space*. For a 512-byte allocation, the average unused space is 256 bytes. For 64 KB clusters, the average unused space is 32 KB. The size of the allocation unit is chosen when the file system is created. Choosing the allocation size based on the average size of the files expected to be in the file system can minimize the amount of unusable space. Frequently the default allocation may provide reasonable usage. Choosing an allocation size that is too small results in excessive overhead if the file system will contain mostly very large files.

File system fragmentation occurs when unused space or single files are not contiguous. As a file system is used, files are created, modified and deleted. When a file is created, the file system allocates space for the data. Some file systems permit or require specifying an initial space allocation and subsequent incremental allocations as the file grows. As files are deleted, the space they were allocated eventually is considered available for use by other files. This creates alternating used and unused areas of various sizes. This is free space fragmentation. When a file is created and there is not an area of contiguous space available for its initial allocation, the space must be assigned in fragments. When a file is modified such that it becomes larger, it may exceed the space initially allocated to it, another allocation must be assigned elsewhere and the file becomes fragmented.

Filenames

A **filename** (or **file name**) is used to identify a storage location in the file system. Most file systems have restrictions on the length of filenames. In some file systems, filenames are not case sensitive (i.e., the names `MYFILE` and `myfile` refer to the same file in a directory); in others, filenames are case sensitive (i.e., the names `MYFILE` , `MyFile` , and `myfile` refer to three separate files that are in the same directory).

Most modern file systems allow filenames to contain a wide range of characters from the Unicode character set. However, they may have restrictions on the use of certain special characters, disallowing them within filenames; those characters might be used to indicate a device, device type, directory prefix, file path separator, or file type.

Directories

File systems typically have **directories** (also called **folders**) which allow the user to group files into separate collections. This may be implemented by associating the file name with an index in a table of contents or an inode in a Unix-like file system. Directory structures may be flat (i.e. linear), or allow hierarchies where directories may contain subdirectories.

Metadata

Other bookkeeping information is typically associated with each file within a file system. The length of the data contained in a file may be stored as the number of blocks allocated for the file or as a byte count. The time that the file was last modified may be stored as the file's timestamp. File systems might store the file creation time, the time it was last accessed, the time the file's metadata was changed, or the time the file was last backed up. Other information can include the file's device type (e.g. block, character, socket, subdirectory, etc.), its owner user ID and group ID, its access permissions and other file attributes (e.g. whether the file is read-only, executable, etc.).

A file system stores all the metadata associated with the file—including the file name, the length of the contents of a file, and the location of the file in the folder hierarchy—separate from the contents of the file.

Most file systems store the names of all the files in one directory in one place—the directory table for that directory—which is often stored like any other file. Many file systems put only some of the metadata for a file in the directory table, and the rest of the metadata for that file in a completely separate structure, such as the inode.

Adapted from:
"File system" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 17.1: Overview is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

17.2: Files

A computer file is a computer resource for recording data in a computer storage device. Just as words can be written to paper, so can data be written to a computer file. Files can be edited and transferred through the Internet on that particular computer system.

Different types of computer files are designed for different purposes. A file may be designed to store a picture, a written message, a video, a computer program, or a wide variety of other kinds of data. Certain files can store multiple data types at once.

By using computer programs, a person can open, read, change, save, and close a computer file. Computer files may be reopened, modified, and copied an arbitrary number of times.

Files are typically organized in a file system, which tracks file locations on the disk and enables user access.

File contents

On most modern operating systems, files are organized into one-dimensional arrays of bytes. The format of a file is defined by its content since a file is solely a container for data, although on some platforms the format is usually indicated by its filename extension, specifying the rules for how the bytes must be organized and interpreted meaningfully. For example, the bytes of a plain text file (.txt in Windows) are associated with either ASCII or UTF-8 characters, while the bytes of image, video, and audio files are interpreted otherwise. Most file types also allocate a few bytes for metadata, which allows a file to carry some basic information about itself.

Some file systems can store arbitrary (not interpreted by the file system) file-specific data outside of the file format, but linked to the file, for example extended attributes or forks. On other file systems this can be done via sidecar files or software-specific databases. All those methods, however, are more susceptible to loss of metadata than are container and archive file formats.

File Size

At any instant in time, a file might have a size, normally expressed as number of bytes, that indicates how much storage is associated with the file. In most modern operating systems the size can be any non-negative whole number of bytes up to a system limit. Many older operating systems kept track only of the number of blocks or tracks occupied by a file on a physical storage device. In such systems, software employed other methods to track the exact byte count.

Operations

The most basic operations that programs can perform on a file are:

- Create a new file
- Change the access permissions and attributes of a file
- Open a file, which makes the file contents available to the program
- Read data from a file
- Write data to a file
- Delete a file
- Close a file, terminating the association between it and the program
- Truncate a file, shortening it to a specified size within the file system without rewriting any content

Files on a computer can be created, moved, modified, grown, shrunk (truncated), and deleted. In most cases, computer programs that are executed on the computer handle these operations, but the user of a computer can also manipulate files if necessary. For instance, Microsoft Word files are normally created and modified by the Microsoft Word program in response to user commands, but the user can also move, rename, or delete these files directly by using a file manager program such as Windows Explorer (on Windows computers) or by command lines (CLI).

In Unix-like systems, user space programs do not operate directly, at a low level, on a file. Only the kernel deals with files, and it handles all user-space interaction with files in a manner that is transparent to the user-space programs. The operating system provides a level of abstraction, which means that interaction with a file from user-space is simply through its filename (instead of its inode). For example, rm filename will not delete the file itself, but only a link to the file. There can be many links to a file, but when they are all removed, the kernel considers that file's memory space free to be reallocated. This free space is commonly

considered a security risk (due to the existence of file recovery software). Any secure-deletion program uses kernel-space (system) functions to wipe the file's data.

File moves within a file system complete almost immediately because the data content does not need to be rewrittten. Only the paths need to be changed.

File Organization

Continuous Allocation

A single continuous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre-allocation strategy, using variable size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. This method is best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block. For example, if a file starts at block b, and the ith block of the file is wanted, its location on secondary storage is simply b+i-1.

Disadvantage

- External fragmentation will occur, making it difficult to find contiguous blocks of space of sufficient length. Compaction algorithm will be necessary to free up additional space on disk.
- Also, with pre-allocation, it is necessary to declare the size of the file at the time of creation.

Linked Allocation(Non-contiguous allocation)

Allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Again the file table needs just a single entry for each file, showing the starting block and the length of the file. Although pre-allocation is possible, it is more common simply to allocate blocks as needed. Any free block can be added to the chain. The blocks need not be continuous. Increase in file size is always possible if free disk block is available. There is no external fragmentation because only one block at a time is needed but there can be internal fragmentation but it exists only in the last disk block of file.

Disadvantage

- Internal fragmentation exists in last disk block of file.
- There is an overhead of maintaining the pointer in every disk block.
- If the pointer of any disk block is lost, the file will be truncated.
- It supports only the sequential access of files.

Indexed Allocation

It addresses many of the problems of contiguous and chained allocation. In this case, the file allocation table contains a separate one-level index for each file: The index has one entry for each block allocated to the file. Allocation may be on the basis of fixed-size blocks or variable-sized blocks. Allocation by blocks eliminates external fragmentation, whereas allocation by variable-size blocks improves locality. This allocation technique supports both sequential and direct access to the file and thus is the most popular form of file allocation.

Just as the space that is allocated to files must be managed ,so the space that is not currently allocated to any file must be managed. To perform any of the file allocation techniques,it is necessary to know what blocks on the disk are available. Thus we need a disk allocation table in addition to a file allocation table.The following are the approaches used for free space management.

Bit Tables

This method uses a vector containing one bit for each block on the disk. Each entry for a 0 corresponds to a free block and each 1 corresponds to a block in use.
For example: 00011010111100110001

In this vector every bit correspond to a particular block and 0 implies that, that particular block is free and 1 implies that the block is already occupied. A bit table has the advantage that it is relatively easy to find one or a contiguous group of free blocks. Thus, a bit table works well with any of the file allocation methods. Another advantage is that it is as small as possible.

Free Block List

In this method, each block is assigned a number sequentially and the list of the numbers of all free blocks is maintained in a reserved block of the disk.

Adaptred from:
"Computer file" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"File Systems in Operating System" by Aakansha yadav, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 17.2: Files is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

17.2.1: Files (continued)

File Access Methods

When a file is used, information is read and accessed into computer memory and there are several ways to access this information of the file. Some systems provide only one access method for files. Other systems, such as those of IBM, support many access methods, and choosing the right one for a particular application is a major design problem.

Sequential Access

It is the simplest access method. Information in the file is processed in order starting at the beginning of the file, one record after the other. This mode of access is by far the most common; for example, editor and compiler usually access the file in this fashion.

Read and write make up the bulk of the operation on a file. A read operation reads from the current file pointer position. Usually, the software reads some pre-determined amount of bytes. The read operation also moves the file pointer to the new position where the system has stopped reading the file. Similarly, for the write command writes at the point of the current pointer.

Direct Access

Another method is *direct access method* also known as *relative access method*. A fixed-length logical record that allows the program to read and write records rapidly, in no particular order. Direct access is based on the disk model of a file since disk allows random access to any file block. For direct access, the file is viewed as a numbered sequence of block or record. Thus, we may read block 14 then block 59 and then we can write block 17. There is no restriction on the order of reading and writing for a direct access file.

A block number provided by the user to the operating system is normally a *relative block number*, the first relative block of the file is 0 and then 1 and so on.

Index Sequential Access

It is a method of accessing a file which is built on top of the sequential access method. This method constructs an index for the file. The index, like an index in the back of a book, contains the pointer to the various blocks. To find a record in the file, we first search the index and then by the help of pointer we access the file directly.

Adapted from:
"File Access Methods in Operating System" by AshishVishwakarma1, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 17.2.1: Files (continued) is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

17.3: Directory

A directory is a file system cataloging structure which contains references to other computer files, and possibly other directories. On many computers, directories are known as folders, or drawers, analogous to a workbench or the traditional office filing cabinet.

Files are organized by storing related files in the same directory. In a hierarchical file system (that is, one in which files and directories are organized in a manner that resembles a tree), a directory contained inside another directory is called a subdirectory. The terms parent and child are often used to describe the relationship between a subdirectory and the directory in which it is cataloged, the latter being the parent. The top-most directory in such a filesystem, which does not have a parent of its own, is called the root directory.

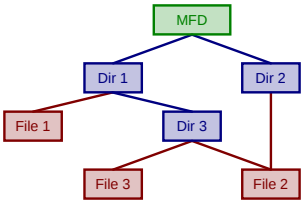


Figure 17.3.1: A typical Files-11 directory hierarchy. ("A typical Files-11 directory hierarchy." by Stannered, Wikimedia Commons is in the Public Domain, CC0)

FILE DIRECTORIES

Collection of files is a file directory. The directory contains information about the files, including attributes, location and ownership. Much of this information, especially that is concerned with storage, is managed by the operating system. The directory is itself a file, accessible by various file management routines. Different operating systems have different structures for their directories.

Information frequently contained in a directories structure

- Name of the directory
- Type of file - not supported on all file systems
- Current length - of the directory
- Maximum length
- Date last accessed
- Date last updated
- Owner id
- Protection information

Operation usually allowed on directory

- Search for a file
- Create a file or directory
- Delete a file or directory
- List the contents of the directory or sub-directory
- Rename a file

Advantages of maintaining directories are

- Efficiency**:** A file can be located more quickly.
- Naming**:** It becomes convenient for users- for example two users can have same name for different files or may have different name for same file.
- Grouping**:** Logical grouping of files can be done by properties e.g. all java programs, all games etc.

Adapted from:
"File system" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"File Systems in Operating System" by Aakansha yadav, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 17.3: Directory is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

17.4: File Sharing

Restricting and Permitting Access

There are several mechanisms used by file systems to control access to data. Usually the intent is to prevent reading or modifying files by a user or group of users. Another reason is to ensure data is modified in a controlled way so access may be restricted to a specific program. Examples include passwords stored in the metadata of the file or elsewhere and file permissions in the form of permission bits, access control lists, or capabilities. The need for file system utilities to be able to access the data at the media level to reorganize the structures and provide efficient backup usually means that these are only effective for polite users but are not effective against intruders.

Methods for encrypting file data are sometimes included in the file system. This is very effective since there is no need for file system utilities to know the encryption seed to effectively manage the data. The risks of relying on encryption include the fact that an attacker can copy the data and use brute force to decrypt the data. Additionally, losing the seed means losing the data.

Maintaining Integrity

One significant responsibility of a file system is to ensure that the file system structures in secondary storage remain consistent, regardless of the actions by programs accessing the file system. This includes actions taken if a program modifying the file system terminates abnormally or neglects to inform the file system that it has completed its activities. This may include updating the metadata, the directory entry and handling any data that was buffered but not yet updated on the physical storage media.

Other failures which the file system must deal with include media failures or loss of connection to remote systems.

In the event of an operating system failure or "soft" power failure, special routines in the file system must be invoked similar to when an individual program fails.

The file system must also be able to correct damaged structures. These may occur as a result of an operating system failure for which the OS was unable to notify the file system, a power failure, or a reset.

The file system must also record events to allow analysis of systemic issues as well as problems with specific files or directories.

User Data

The most important purpose of a file system is to manage user data. This includes storing, retrieving and updating data.

Some file systems accept data for storage as a stream of bytes which are collected and stored in a manner efficient for the media. When a program retrieves the data, it specifies the size of a memory buffer and the file system transfers data from the media to the buffer. A runtime library routine may sometimes allow the user program to define a record based on a library call specifying a length. When the user program reads the data, the library retrieves data via the file system and returns a record.

Some file systems allow the specification of a fixed record length which is used for all writes and reads. This facilitates locating the nth record as well as updating records.

An identification for each record, also known as a key, makes for a more sophisticated file system. The user program can read, write and update records without regard to their location. This requires complicated management of blocks of media usually separating key blocks and data blocks. Very efficient algorithms can be developed with pyramid structures for locating records.

Adapted from:
"File system" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

This page titled 17.4: File Sharing is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

Index

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: Introduction to Operating Systems

Webpages: 112

All licenses found:

- CC BY-SA 4.0: 73.2% (82 pages)
- Undeclared: 15.2% (17 pages)
- CC BY-SA 3.0: 11.6% (13 pages)

By Page

- Introduction to Operating Systems - *Undeclared*
  - Front Matter - *Undeclared*
    - TitlePage - *Undeclared*
    - InfoPage - *Undeclared*
    - Table of Contents - *Undeclared*
    - Licensing - *Undeclared*
  - 1: Binary and Number Representation - *Undeclared*
    - 1.1: How Computers See Numbers - *CC BY-SA 4.0*
    - 1.2: Types and Number Representation - *CC BY-SA 3.0*
  - 2: The Basics - An Overview - *Undeclared*
    - 2.1: Introduction to Operating Systems - *CC BY-SA 4.0*
    - 2.2: Starting with the Basics - *CC BY-SA 4.0*
  - 3: The Operating System - *Undeclared*
    - 3.1: The Role of the Operating System - *CC BY-SA 3.0*
    - 3.2: Operating System Organisation - *CC BY-SA 3.0*
    - 3.3: System Calls - *CC BY-SA 3.0*
    - 3.4: Privileges - *CC BY-SA 3.0*
    - 3.5: Function of the Operating System - *CC BY-SA 4.0*
    - 3.6: Types of Operating Systems - *CC BY-SA 4.0*
      - 3.6.1: Types of Operating Systems (continued) - *CC BY-SA 4.0*
      - 3.6.2: Types of Operating Systems (continued) - *CC BY-SA 4.0*
    - 3.7: Difference between multitasking, multithreading and multiprocessing - *CC BY-SA 4.0*
      - 3.7.1: Difference between multitasking, multithreading and multiprocessing (continued) - *CC BY-SA 4.0*
      - 3.7.2: Difference between Multiprogramming, multitasking, multithreading and multiprocessing (continued) - *CC BY-SA 4.0*
  - 4: Computer Architecture - the CPU - *Undeclared*
    - 4.1: Instruction Cycles - *CC BY-SA 4.0*
      - 4.1.1: Instruction Cycles - Fetch - *CC BY-SA 4.0*
      - 4.1.2: Instruction Cycles - Instruction Primer - *CC BY-SA 4.0*
    - 4.2: Interrupts - *CC BY-SA 4.0*
  - 5: Computer Architecture - Memory - *Undeclared*
    - 5.1: Memory Hierarchy - *CC BY-SA 4.0*
    - 5.2: Memory Hierarchy (continued) - *CC BY-SA 4.0*
    - 5.3: Cache Memory - *CC BY-SA 4.0*
      - 5.3.1: Cache Memory - Multilevel Cache - *CC BY-SA 4.0*
      - 5.3.2: Cache Memory - Locality of reference - *CC BY-SA 4.0*
    - 5.4: Direct Memory Access - *CC BY-SA 4.0*
  - 6: Computer Architecture - Peripherals and Buses - *CC BY-SA 4.0*
    - 6.1: Peripherals and buses - *CC BY-SA 3.0*
    - 6.2: The Processor - Bus - *CC BY-SA 4.0*
  - 7: Small to Big Systems - *Undeclared*
    - 7.1: Symmetric Multi-Processing Systems - *CC BY-SA 3.0*
    - 7.2: Multiprocessor and Multicore Systems - *CC BY-SA 4.0*
  - 8: Processes - *Undeclared*
    - 8.1: The Process - *CC BY-SA 3.0*
    - 8.2: Processes - *CC BY-SA 4.0*
    - 8.3: Elements of a process - *CC BY-SA 3.0*
    - 8.4: Process States - *CC BY-SA 4.0*
    - 8.5: Process Description - *CC BY-SA 4.0*
    - 8.6: Process Control - *CC BY-SA 4.0*
    - 8.7: Fork and Exec - *CC BY-SA 3.0*
    - 8.8: Scheduling - *CC BY-SA 3.0*

- 8.9: Execution within the Operating System - *CC BY-SA 4.0*
  - 8.9.1: Execution within the Operating System - Dual Mode - *CC BY-SA 4.0*
- 8.10: The Shell - *CC BY-SA 3.0*
- 8.11: Signals - *CC BY-SA 3.0*

- 9: Threads - *CC BY-SA 4.0*
  - 9.1: Process and Threads - *CC BY-SA 4.0*
  - 9.2: Thread Types - *CC BY-SA 4.0*
    - 9.2.1: Thread Types - Models - *CC BY-SA 4.0*
  - 9.3: Thread Relationships - *CC BY-SA 4.0*
  - 9.4: Benefits of Multithreading - *CC BY-SA 4.0*
- 10: Concurrency and Process Synchronization - *CC BY-SA 4.0*
  - 10.1: Introduction to Concurrency - *CC BY-SA 4.0*
  - 10.2: Process Synchronization - *CC BY-SA 4.0*
  - 10.3: Mutual Exclusion - *CC BY-SA 4.0*
  - 10.4: Interprocess Communication - *CC BY-SA 4.0*
    - 10.4.1: IPC - Semaphores - *CC BY-SA 4.0*
    - 10.4.2: IPC - Monitors - *CC BY-SA 4.0*
    - 10.4.3: IPC - Message Passing / Shared Memory - *CC BY-SA 4.0*
- 11: Concurrency- Deadlock and Starvation - *CC BY-SA 4.0*
  - 11.1: Concept and Principles of Deadlock - *CC BY-SA 4.0*
  - 11.2: Deadlock Detection and Prevention - *CC BY-SA 4.0*
  - 11.3: Starvation - *CC BY-SA 4.0*
  - 11.4: Dining Philosopher Problem - *CC BY-SA 4.0*
- 12: Memory Management - *CC BY-SA 4.0*
  - 12.1: Random Access Memory (RAM) and Read Only Memory (ROM) - *CC BY-SA 4.0*
  - 12.2: Memory Hierarchy - *CC BY-SA 4.0*
  - 12.3: Requirements for Memory Management - *CC BY-SA 4.0*
  - 12.4: Memory Partitioning - *CC BY-SA 4.0*
    - 12.4.1: Fixed Partitioning - *CC BY-SA 4.0*
    - 12.4.2: Variable Partitioning - *CC BY-SA 4.0*
    - 12.4.3: Buddy System - *CC BY-SA 4.0*
  - 12.5: Logical vs Physical Address - *CC BY-SA 4.0*
  - 12.6: Paging - *CC BY-SA 4.0*
  - 12.7: Segmentation - *CC BY-SA 4.0*
- 13: Virtual Memory - *CC BY-SA 4.0*
  - 13.1: Memory Paging - *CC BY-SA 4.0*
    - 13.1.1: Memory Paging - Page Replacement - *CC BY-SA 4.0*
  - 13.2: Virtual Memory in the Operating System - *CC BY-SA 4.0*
- 14: Uniprocessor CPU Scheduling - *CC BY-SA 4.0*
  - 14.1: Types of Processor Scheduling - *CC BY-SA 4.0*
  - 14.2: Scheduling Algorithms - *CC BY-SA 4.0*
- 15: Multiprocessor Scheduling - *CC BY-SA 4.0*
  - 15.1: The Question - *CC BY-SA 4.0*
  - 15.2: Multiprocessor Scheduling - *CC BY-SA 4.0*
- 16: I/O and Disk Management - *CC BY-SA 4.0*
  - 16.1: Input / Output - *CC BY-SA 4.0*
  - 16.2: Types of I/O - *CC BY-SA 4.0*
  - 16.3: I/O Buffering - *CC BY-SA 4.0*
  - 16.4: Disk Drives in the OS - *CC BY-SA 4.0*
  - 16.5: Disk Drive Scheduling - *CC BY-SA 4.0*
  - 16.6: RAID - *CC BY-SA 4.0*
- 17: File Management - *CC BY-SA 4.0*
  - 17.1: Overview - *CC BY-SA 4.0*
  - 17.2: Files - *CC BY-SA 4.0*
    - 17.2.1: Files (continued) - *CC BY-SA 4.0*
  - 17.3: Directory - *CC BY-SA 4.0*
  - 17.4: File Sharing - *CC BY-SA 4.0*
- Back Matter - *Undeclared*
  - Index - *Undeclared*
  - Glossary - *Undeclared*
  - Detailed Licensing - *Undeclared*