OPERATING SYSTEM: THE BASICS

Patrick McClanahan
San Joaquin Delta College

LibreTexts™

Sign in

Sign in

Welcome to the LibreTexts library. For other users, please log in with LibreOne.

Log in with LibreOne

CHAPTER OVERVIEW

1: The Basics - An Overview

1.1: Introduction to Operating Systems
1.2 Starting with the Basics
1.3 The Processor - History
1.3.1: The Processor - Components
1.3.2: The Processor - Bus
1.4 Instruction Cycles
1.4.1 Instruction Cycles - Fetch
1.4.2 Instruction Cycles - Instruction Primer
1.5 Interrupts
1.6 Memory Hierarchy
1.7 Cache Memory
1.7.1 Cache Memory - Multilevel Cache
1.7.2 Cache Memory - Locality of reference
1.8 Direct Memory Access
1.9 Multiprocessor and Multicore Systems

This page titled 1: The Basics - An Overview is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.1: 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.

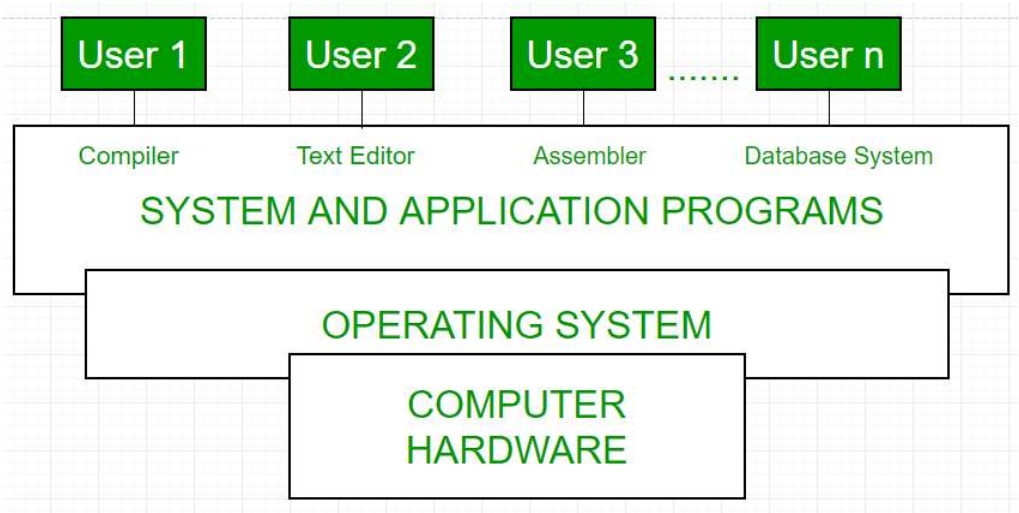


Figure 1.1.1: Conceptual view of a computer system ("Conceptual view of a computer system" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0)

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 –
The module that keeps track of the status of devices is called the I/O traffic controller. Each I/O device has a device handler that resides in a separate process associated with that device.
The I/O subsystem consists of

- A memory Management component that includes buffering caching and spooling.
- A general device driver interface.

Drivers for specific hardware devices.

Assembler – The input to an assembler is an assembly language program. The output is an object program plus information that enables the loader to prepare the object program for execution. At one time, the computer programmer had at their disposal a basic machine that interpreted, through hardware, certain fundamental instructions. The programmer would program the computer by writing a series of ones and zeros (machine language), and place them into the memory of the machine.

Compiler/Interpreter – The high-level languages - for example C/C++, are processed by compilers and interpreters. A compiler is a program that accepts source code written in a "high-level language "and produces a corresponding object program. An interpreter is a program that directly executes a source program as if it was machine language.

Loader – A loader is a routine that loads an object program and prepares it for execution. There are various loading schemes: absolute, relocating and direct-linking. In general, the loader must load, relocate and link the object program. The loader is a program that places programs into memory and prepares them for execution. In a simple loading scheme, the assembler outputs the machine language translation of a program on a secondary device and a loader places it in the core. The loader places into memory the machine language version of the user's program and transfers control to it. Since the loader program is much smaller than the assembler, those make more core available to the user's program.

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

Adapted from:
"Introduction of Operating System – Set 1" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 1.1: Introduction to Operating Systems is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.2 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.

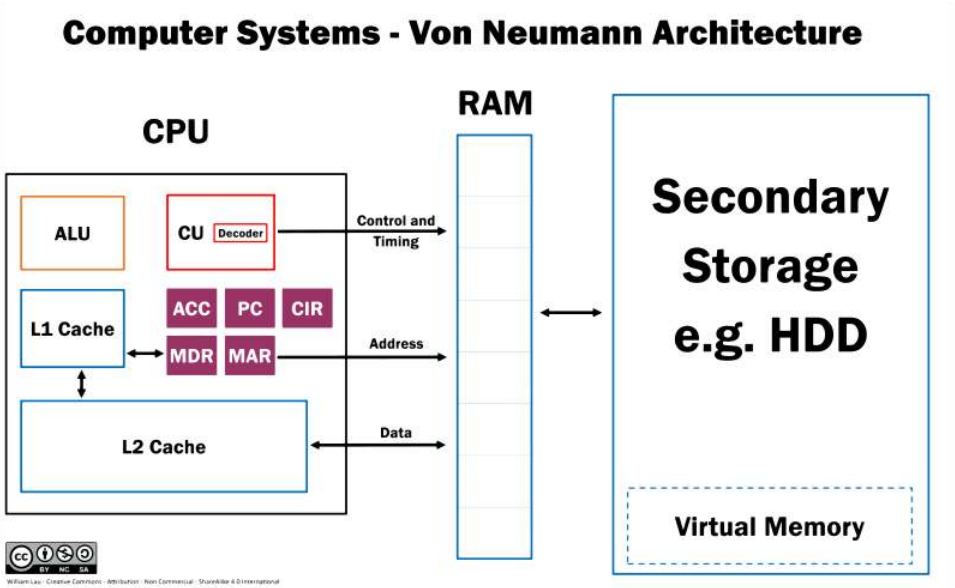


Figure 1: Von Neumann Architecture. ("File:Computer Systems - Von Neumann Architecture Large poster anchor chart.svg" by BotMultichillT, Wikimedia Commons is licensed under CC BY-NC-SA 4.0)

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 memoryand 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 1.2 Starting with the Basics is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.3 The Processor - History

Microprocessor Evolution

Generations of microprocessor:

1. **First generation:** From 1971 to 1972 the era of the first generation came which brought microprocessors like INTEL 4004 Rockwell international PPS-4 INTEL 8008 etc.
2. **Second generation:** The second generation marked the development of 8 bit microprocessors from 1973 to 1978. Processors like INTEL 8085 Motorola 6800 and 6801 etc came into existence.
3. **Third generation:** The third generation brought forward the 16 bit processors like INTEL 8086/80186/80286 Motorola 68000 68010 etc. From 1979 to 1980 this generation used the HMOS technology.
4. **Fourth generation:** The fourth generation came into existence from 1981 to 1995. The 32 bit processors using HMOS fabrication came into existence. INTEL 80386 and Motorola 68020 are some of the popular processors of this generation.
5. **Fifth generation:** From 1995 till now we are in the fifth generation. 64 bit processors like Pentium, Celeron, dual, quad and octa core processors came into existence.

Types of microprocessors:

- **Complex instruction set microprocessor:** The processors are designed to minimize the number of instructions per program and ignore the number of cycles per instructions. The compiler is used to translate a high level language to assembly level language because the length of code is relatively short and an extra RAM is used to store the instructions. These processors can do tasks like downloading, uploading and recalling data from memory. Apart from these tasks these microprocessor can perform complex mathematical calculation in a single command.
  Example: IBM 370/168, VAX 11/780
- **Reduced instruction set microprocessor:** These processor are made according to function. They are designed to reduce the execution time by using the simplified instruction set. They can carry out small things in specific commands. These processors complete commands at faster rate. They require only one clock cycle to implement a result at uniform execution time. There are number of registers and less number of transistors. To access the memory location LOAD and STORE instructions are used.
  Example: Power PC 601, 604, 615, 620
- **Super scalar microprocessor:** These processors can perform many tasks at a time. They can be used for ALUs and multiplier like array. They have multiple operation unit and perform fast by executing multiple commands.
- **Application specific integrated circuit:** These processors are application specific like for personal digital assistant computers. They are designed according to proper specification.
- **Digital signal multiprocessor:** These processors are used to convert signals like analog to digital or digital to analog. The chips of these processors are used in many devices such as RADAR SONAR home theaters etc.

Advantages of microprocessor

1. High processing speed
2. Compact size
3. Easy maintenance
4. Can perform complex mathematics
5. Flexible
6. Can be improved according to requirement

Disadvantages of microprocessors

1. Overheating occurs due to overuse
2. Performance depends on size of data
3. Large board size than microcontrollers
4. Most microprocessors do not support floating point operations

Adapted from:
"Evolution of Microprocessors" by Ayusharma0698, Geeks for Geeks is licensed under CC BY-SA 4.0

This page titled 1.3 The Processor - History is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.3.1: The Processor - Components

Accumulator

In a computer's central processing unit (CPU), the accumulator (ACC in the image below) is a register in which intermediate arithmetic and logic results are stored.

Without a register like an accumulator, it would be necessary to write the result of each calculation (addition, multiplication, shift, etc.) to main memory, perhaps only to be read right back again for use in the next operation.

Access to main memory is slower than access to a register like an accumulator because the technology used for the large main memory is slower (but cheaper) than that used for a register. Early electronic computer systems were often split into two groups, those with accumulators and those without.

Modern computer systems often have multiple general-purpose registers that can operate as accumulators, and the term is no longer as common as it once was. However, to simplify their design, a number of special-purpose processors still use a single accumulator.

Arithmetic logic unit

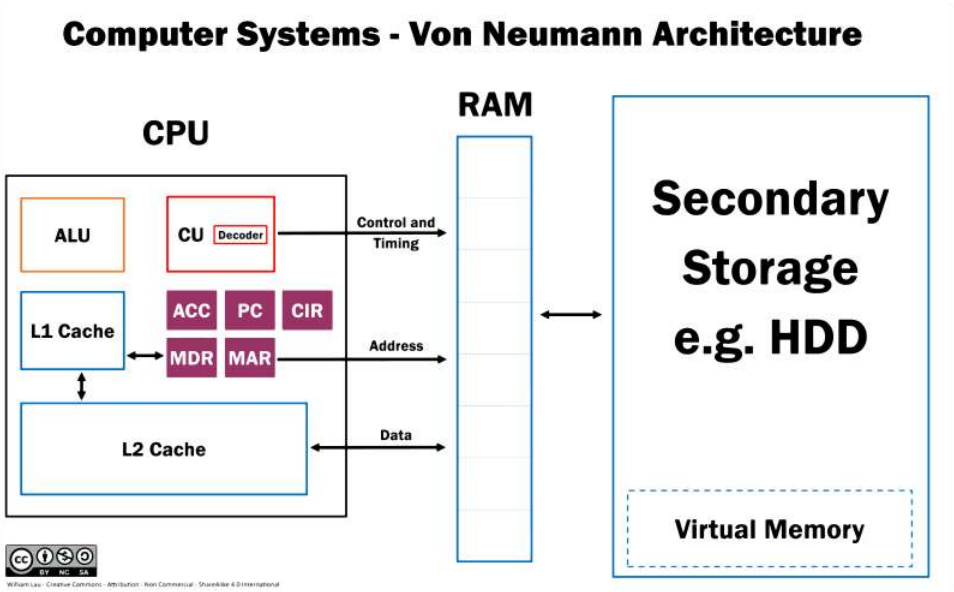
The arithmetic logic unit (ALU) performs the arithmetic and logical functions that are the work of the computer. There are other general purpose registers that hold the input data, and the accumulator receives the result of the operation. The instruction register contains the instruction that the ALU is to perform.

For example, when adding two numbers, one number is placed in one of the general purpose registers register and the other in another general purpose register. The ALU performs the addition and puts the result in the accumulator. If the operation is a logical one, the data to be compared is placed into the one of the general purpose registers. The result of the comparison, a 1 or 0, is put in the accumulator. Whether this is a logical or arithmetic operation, the accumulator content is then placed into the cache location reserved by the program for the result.

Instruction register and pointer

The instruction pointer (CIR in the image below) specifies the location in memory containing the next instruction to be executed by the CPU. When the CPU completes the execution of the current instruction, the next instruction is loaded into the instruction register from the memory location pointed to by the instruction pointer.

After the instruction is loaded into the instruction register, the instruction register pointer is incremented by one instruction address. Incrementing allows it to be ready to move the next instruction into the instruction register.



Memory Address Register

In a computer, the memory address register (MAR) is the CPU register that either stores the memory address from which data will be fetched to the CPU, or the address to which data will be sent and stored.

In other words, This register is used to access data and instructions from memory during the execution phase of instruction. MAR holds the memory location of data that needs to be accessed. When reading from memory, data addressed by MAR is fed into the MDR (memory data register) and then used by the CPU. When writing to memory, the CPU writes data from MDR to the memory location whose address is stored in MAR. MAR, which is found inside the CPU, goes either to the RAM (random access memory) or cache.

Memory Data Register

The memory data register (MDR) is the register that stores the data being transferred to and from the immediate access storage. It contains the copy of designated memory locations specified by the memory address register. It acts as a buffer allowing the processor and memory units to act independently without being affected by minor differences in operation. A data item will be copied to the MDR ready for use at the next clock cycle, when it can be either used by the processor for reading or writing or stored in main memory after being written.

This register holds the contents of the memory which are to be transferred from memory to other components or vice versa. A word to be stored must be transferred to the MDR, from where it goes to the specific memory location, and the arithmetic data to be processed in the ALU first goes to MDR and then to accumulated register, and then it is processed in the ALU.

The MDR is a two-way register. When data is fetched from memory and placed into the MDR, it is written to go in one direction. When there is a write instruction, the data to be written is placed into the MDR from another CPU register, which then puts the data into memory.

Cache

The CPU never directly accesses RAM. Modern CPUs have one or more layers of cache. The CPU's ability to perform calculations is much faster than the RAM's ability to feed data to the CPU.

Cache memory is faster than the system RAM, and it is closer to the CPU because it is physically located on the processor chip. The cache provides data storage and instructions to prevent the CPU from waiting for data to be retrieved from RAM. When the CPU needs data - program instructions are also considered to be data - the cache determines whether the data is already in residence and provides it to the CPU.

If the requested data is not in the cache, it's retrieved from RAM and uses predictive algorithms to move more data from RAM into the cache. The cache controller analyzes the requested data and tries to predict what additional data will be needed from RAM. It loads the anticipated data into the cache. By keeping some data closer to the CPU in a cache that is faster than RAM, the CPU can remain busy and not waste cycles waiting for data.

The simple example CPU has two levels of cache. Levels 2 is designed to predict what data and program instructions will be needed next, move that data from RAM, and move it ever closer to the CPU to be ready when needed. These cache sizes typically range from 1 MB to 32 MB, depending upon the speed and intended use of the processor.

CPU clock and control unit (CU in the image)

All of the CPU components must be synchronized to work together smoothly. The control unit performs this function at a rate determined by the clock speed and is responsible for directing the operations of the other units by using timing signals that extend throughout the CPU.

Random access memory (RAM)

Although the RAM, or main storage, is shown in this diagram and the next, it is not truly a part of the CPU. Its function is to store programs and data so that they are ready for use when the CPU needs them.

Adapted from:
"Accumulator (computing)" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Memory buffer register" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
"Memory address register" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0
William Lau, CC BY-SA 4.0, via Wikimedia Commons

1.3.1: The Processor - Components is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

1.3.2: 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

1.3.2: The Processor - Bus is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

1.4 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 1.4 Instruction Cycles is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.4.1 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).

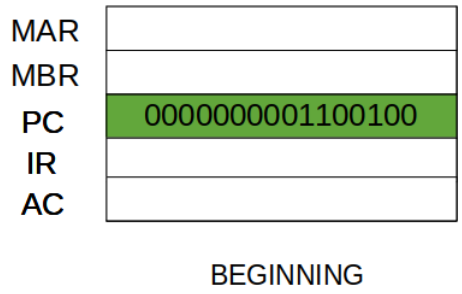


Figure 1: Beginning of the Fetch Cycle. ("Beginning of the Fetch Cycle" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

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.

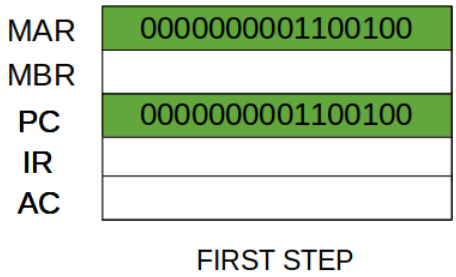


Figure 1: Step #1 of Fetch Cycle. ("Beginning of the Fetch Cycle" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

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)

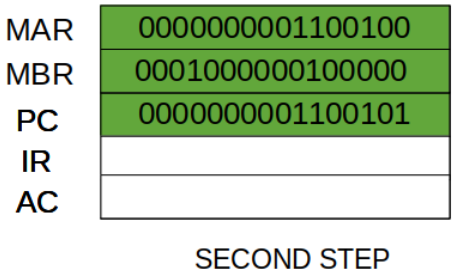


Figure 1: Step #2 of Fetch Cycle. ("Step #2 of the Fetch Cycle" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

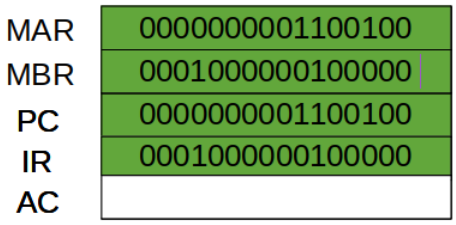


Figure 1: Step #3 of Fetch Cycle. ("Step #3 of the Fetch Cycle" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

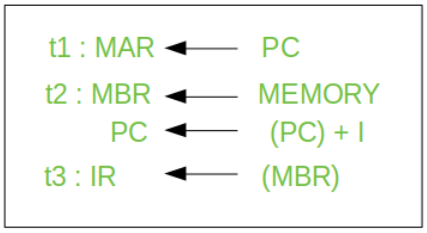


Figure 1: Fetch Cycle Steps. ("Fetch Cycle Steps" by Astha_Singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

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 1.4.1 Instruction Cycles - Fetch is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.4.2 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 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 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.

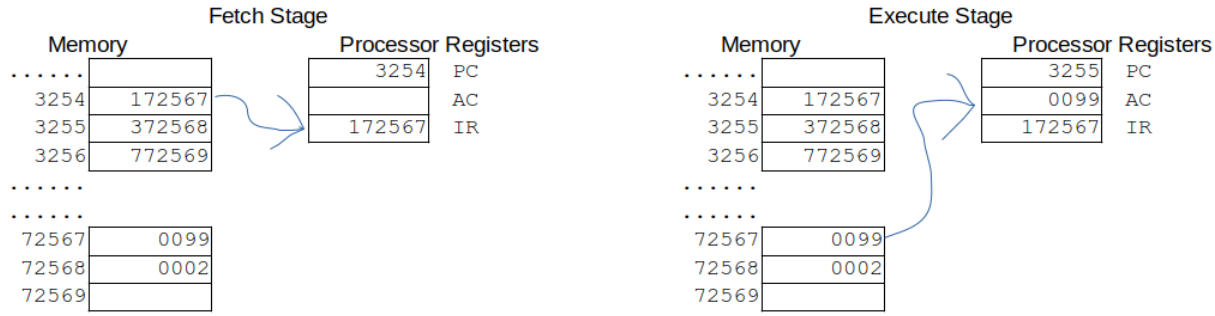


Figure 3: First Cycle. ("First Cycle" by Patrick McClanahan is licensed under CC BY-SA 4.0)

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.

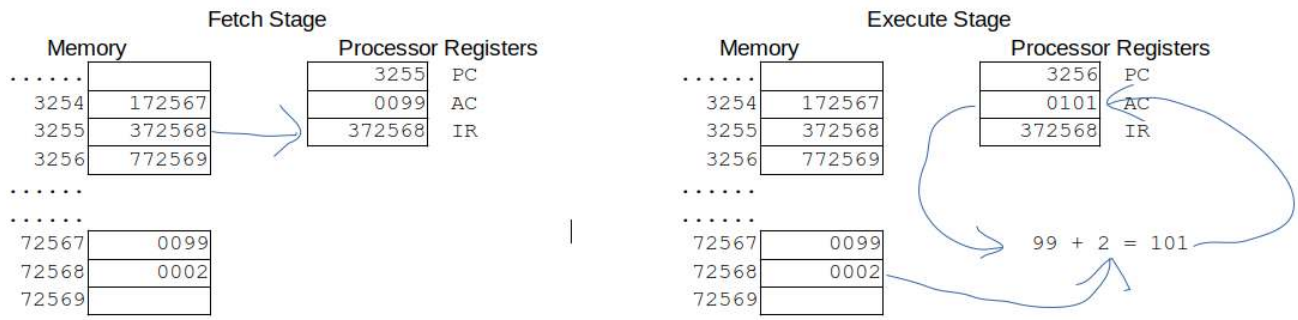


Figure 4: Second Cylce ("Second Cycle" by Patrick McClanahan is licensed under CC BY-SA 4.0)

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.

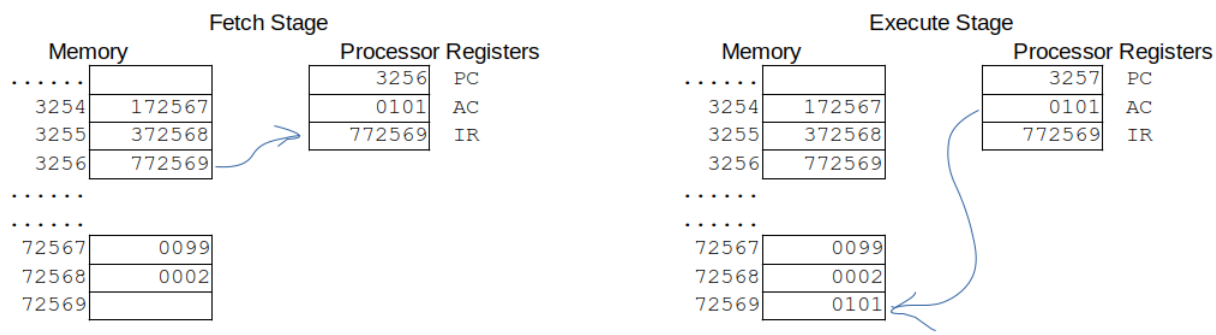


Figure 5: Third Cycle. ("Third Cycle" by Patrick McClanahan is licensed under CC BY-SA 4.0)

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 1.4.2 Instruction Cycles - Instruction Primer is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.5 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:

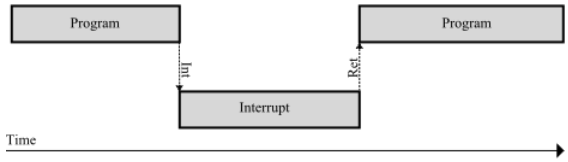


Figure 1: Concept of an interrupt. ("Concept of an Interrupt" by lemilxavier, WikiBooks is licensed under CC BY-SA 3.0)

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.

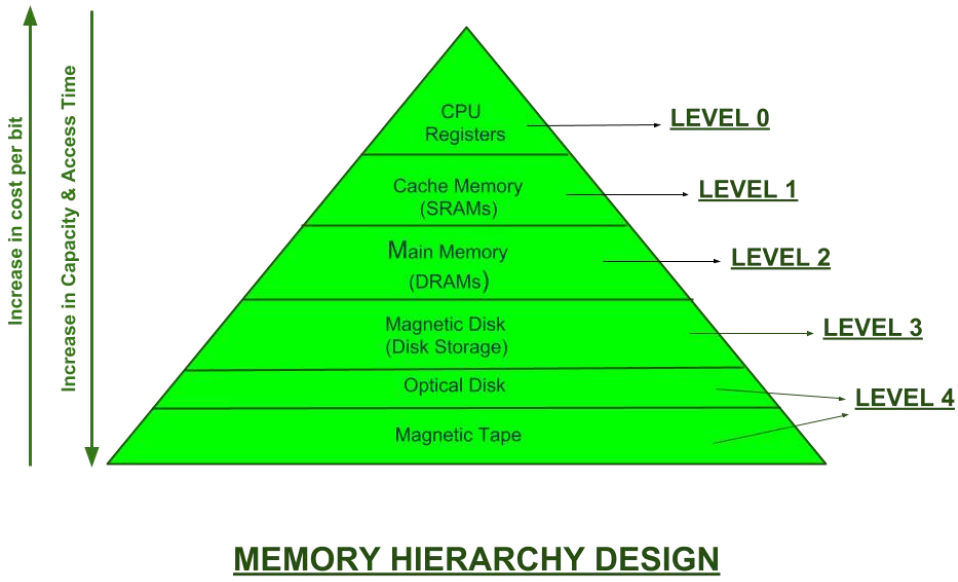
Processors priority is encoded in a few bits of PS (Process Status register). It can be changed by program instructions that write into the PS. Processor is in supervised mode only while executing OS routines. It switches to user mode before executing application programs

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 1.5 Interrupts is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.6 Memory Hierarchy

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 :



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 1.6 Memory Hierarchy is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.7 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.

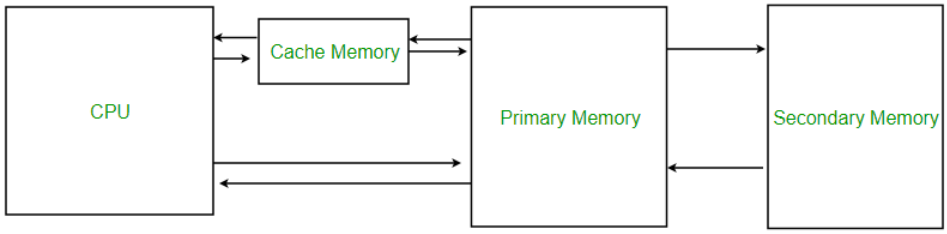


Figure 1: Cahce Memory. ("Cache Memory" by VaibhavRai3, Geeks for Geeks is licensed under CC BY-SA 4.0)

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 1.7 Cache Memory is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.7.1 Cache Memory - Multilevel Cache

Multilevel Cache

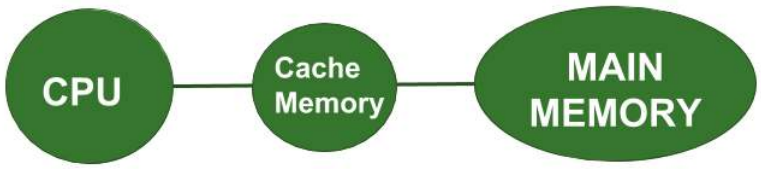
Multilevel cache is one of the techniques to improve cache performance by reducing the “miss penalty”. The term miss penalty refers to the extra time required to bring the data into cache from the main memory whenever there is a “miss” in cache .

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

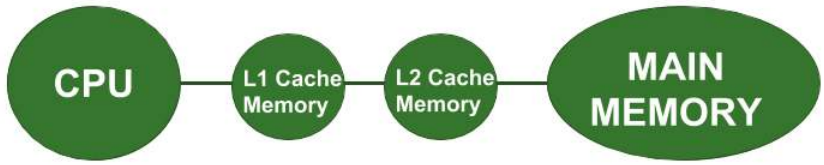
Here the CPU directly communicates with the main memory and no caches are involved.
In this case, the CPU needs to access the main memory 10 times to access the desired information.

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 1.7.1 Cache Memory - Multilevel Cache is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.7.2 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.

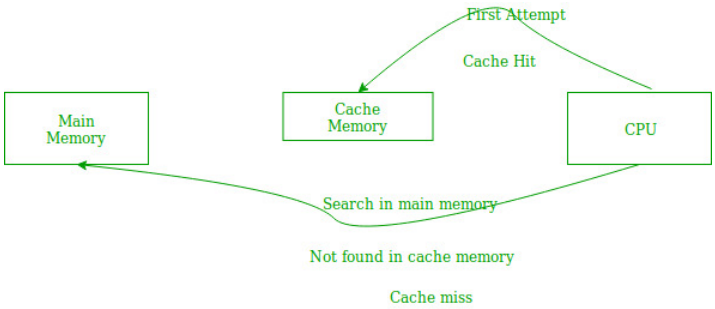


Figure 1: Cache Hit / Cache Miss. ("Cache Hit / Miss" by balwant_singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

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.

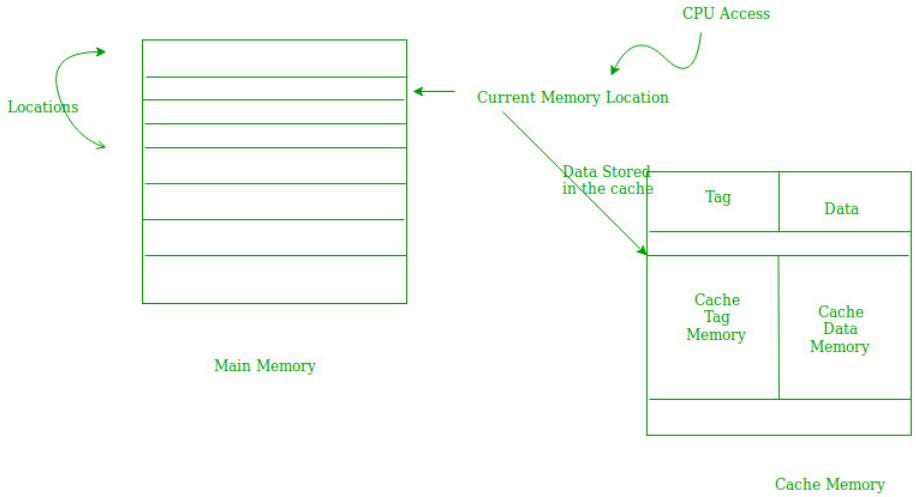


Figure 1: Temporal Locality. ("Temporal Locality" by balwant_singh, Geeks for Geeks is licensed under CC BY-SA 4.0)

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.

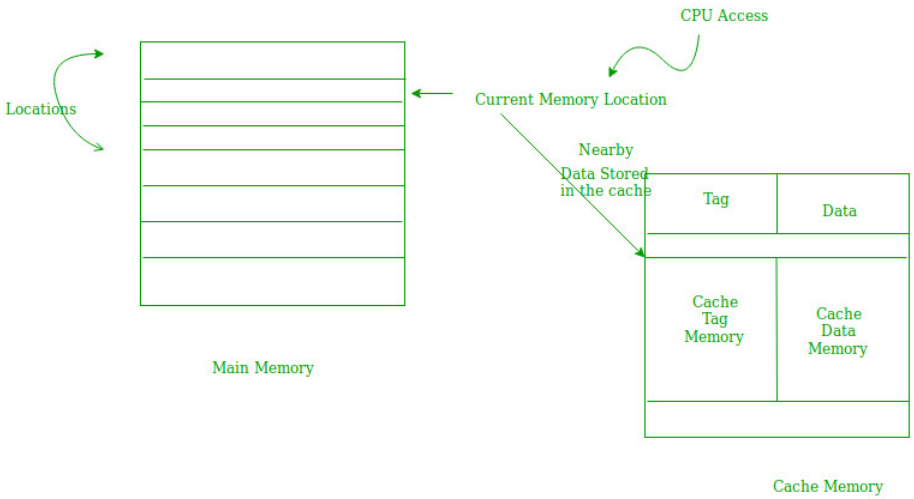


Figure 1: Spatial Locality. ("Spatial Locality" by **balwant_singh**, Geeks for Geeks is licensed under CC BY-SA 4.0)

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 1.7.2 Cache Memory - Locality of reference is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.8 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 1.8 Direct Memory Access is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

1.9 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.

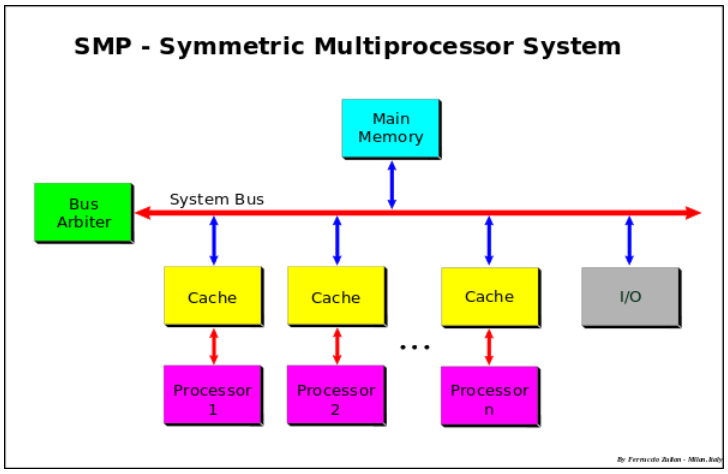


Figure 1: Symmetric multiprocessing system. ("Symmetric multiprocessing system" by Multiple Contributors, Wikipedia Commons is licensed under CC BY-SA 3.0)

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.

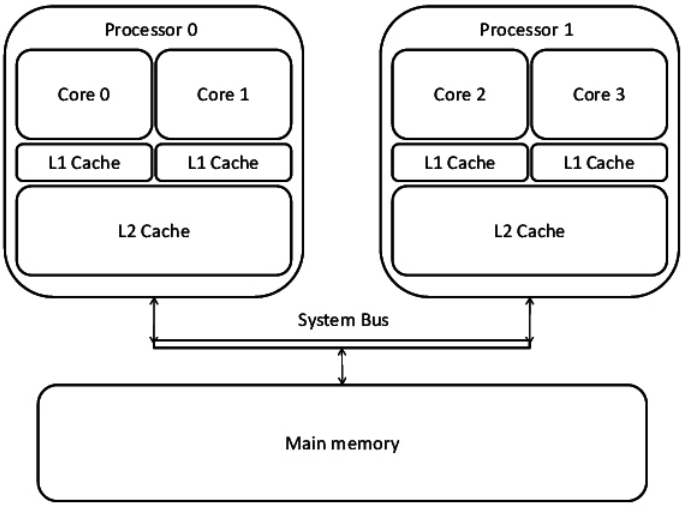


Figure 1: Quad Core Processor. ("Diagram of a generic dual-core processor " by Multiple Contributors, Wikipedia Commons is licensed under CC BY-SA 3.0)

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

CHAPTER OVERVIEW

2: Operating System Overview

2.1: Function of the Operating System

2.2: Types of Operating Systems

2.2.1: Types of Operating Systems (continued)

2.2.2: Types of Operating Systems (continued)

2.3: Difference between multitasking, multithreading and multiprocessing

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

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

This page titled 2: Operating System Overview is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

2.1: 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 2.1: Function of the Operating System is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

2.2: 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.

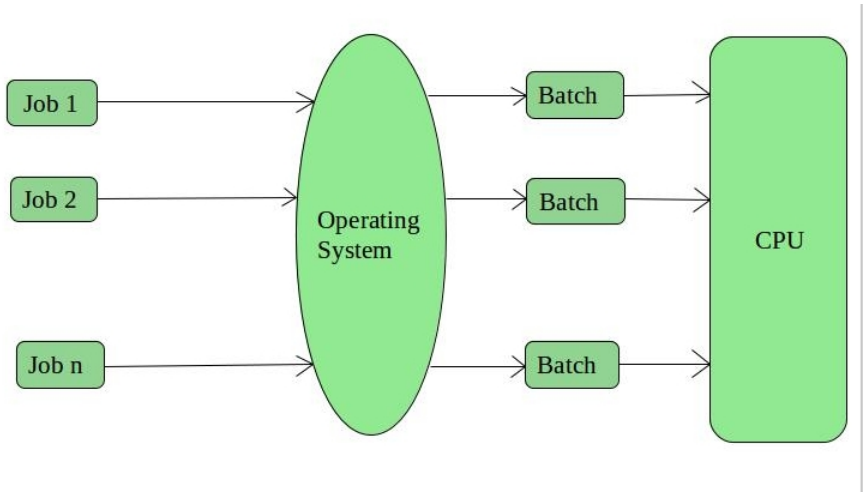


Figure 2.2.1: Depiction of a batch operating system. ("A batch operating system" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0)

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 idle time for batch system is very less
- It is easy to manage large work repeatedly in batch systems

Disadvantages of Batch Operating System:

- The computer operators should be well known with 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 time of CPU as they use single system. These systems are also known as Multitasking Systems. The task can be from single user or from different users also. The time that each task gets to execute is called quantum. After this time interval is over OS switches over to next task.

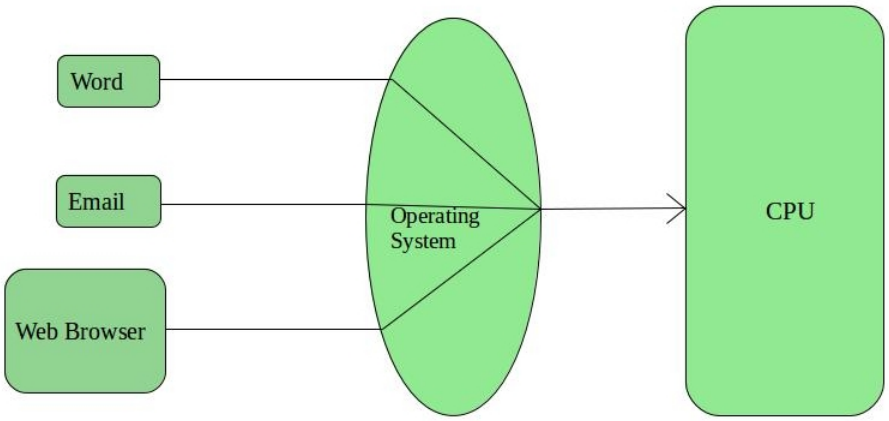


Figure 2.2.1: Time sharing Operating System. ("A time share operating system" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0)

Advantages of Time-Sharing OS:

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

Disadvantages of Time-Sharing OS:

- Reliability problem
- One must have to take care of security and integrity of user programs and data
- Data communication problem

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

2.2.1: 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.

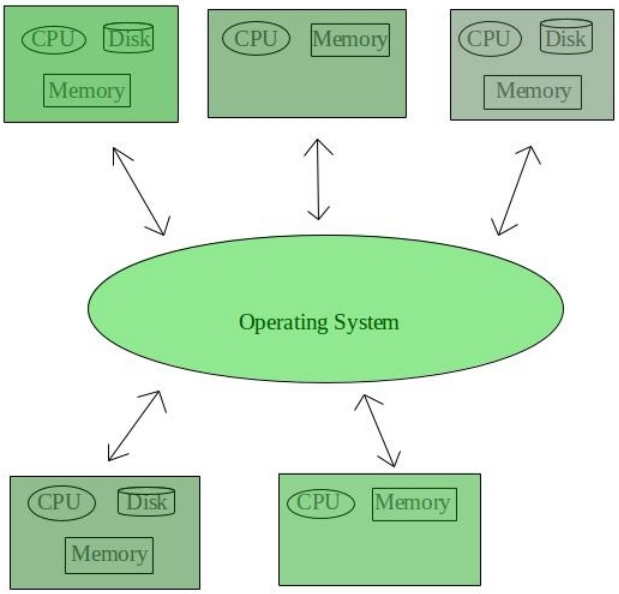


Figure 2.2.1.1: Distributed Operating System. ("Distributed Operating System" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0)

Advantages of Distributed Operating System:

- Failure of one will not affect the other network communication, as all systems are independent from each other
- Electronic mail increases the data exchange speed
- Since resources are being shared, computation is highly fast and durable
- Load on host computer reduces
- These systems are easily scalable as many systems can be easily added to the network
- Delay in data processing reduces

Disadvantages of Distributed Operating System:

- Failure of the main network will stop the entire communication
- To establish distributed systems the language which are used are not well defined yet
- These types of systems are not readily available as they are very expensive. Not only that the underlying software is highly complex and not understood well yet

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.

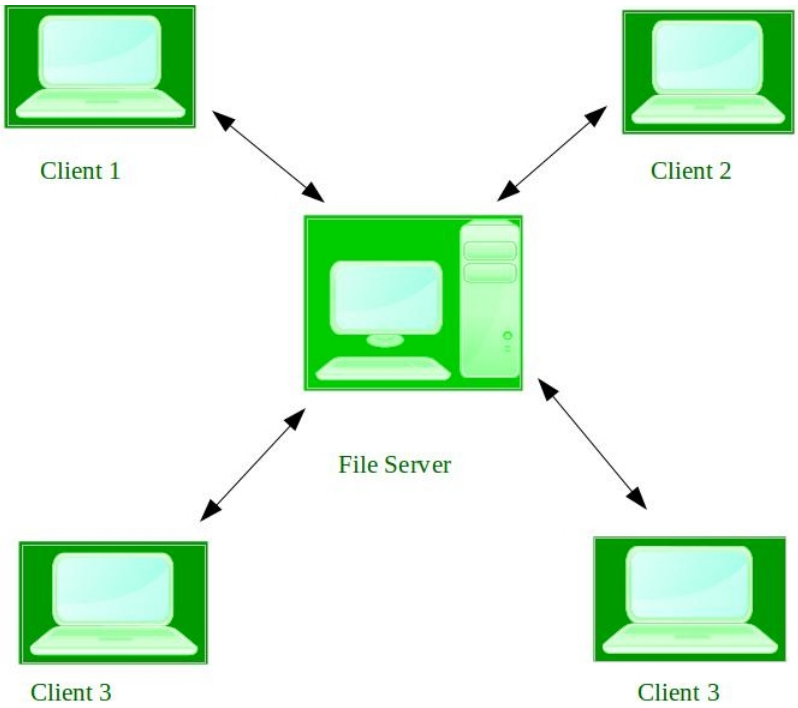


Figure 2.2.1.1: Network Operating System. ("Network Operating System" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

2.2.2: 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.

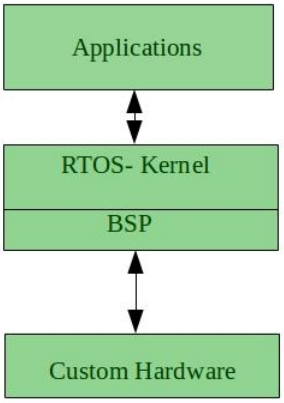


Figure 2.2.2.1: Real Time Operating System. ("Real time operating system" by akash1295, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

2.3: 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.

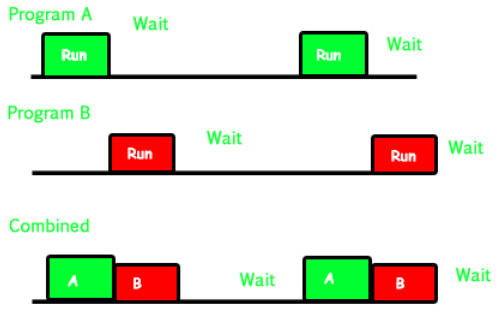


Figure 2.3.1: Multiprogramming. ("Multiprogramming" by Darshan L., Geeks for Geeks is licensed under CC BY-SA 4.0)

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

2.3.1: 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).

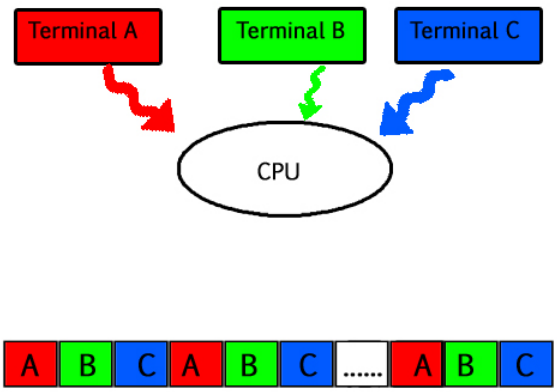


Figure 2.3.1.1: Depiction of Multitasking System. ("Multitasking" by Darshan L., Geeks for Geeks is licensed under CC BY-SA 4.0)

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 2.3.1: 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.

2.3.2: 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:

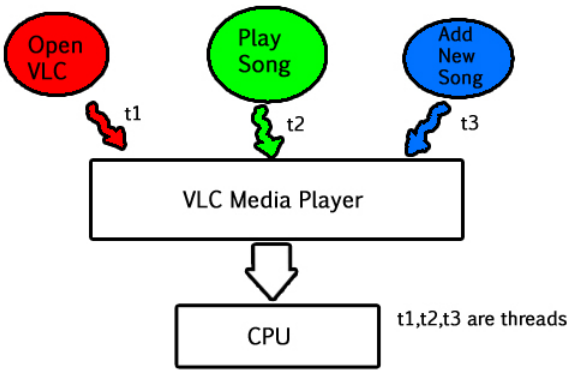


Figure 2.3.2.1: Example of Multithreading. ("Multithreading" by Darshan L., Geeks for Geeks is licensed under CC BY-SA 4.0)

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 2.3.2: 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

3: Processes - What and How

3.1: Processes

3.2: Process States

3.3 Process Description

3.4: Process Control

3.5: Execution within the Operating System

3.5.1 : Execution within the Operating System - Dual Mode

This page titled 3: Processes - What and How is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

3.1: 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.

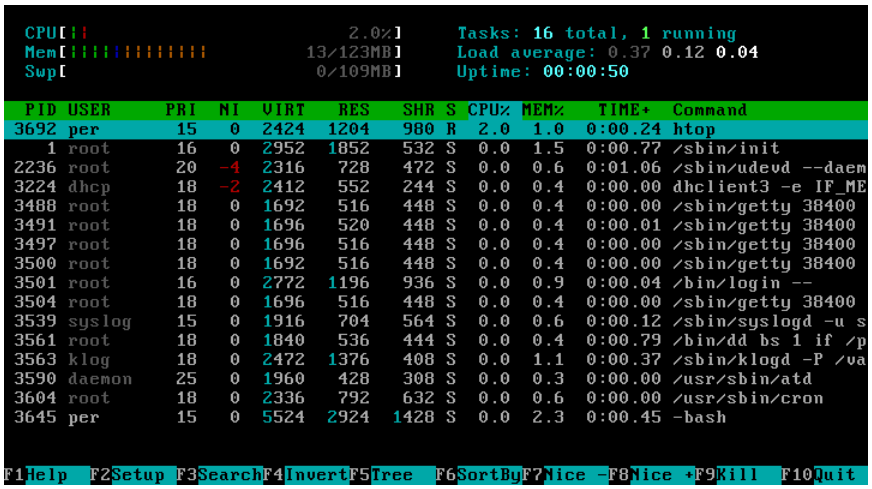


Figure 3.1.1: Output of htop Linux Command. (PER9000, Per Erik Strandberg, CC BY-SA 3.0, via Wikimedia Commons)

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 (described above), 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

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

3.2: 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.

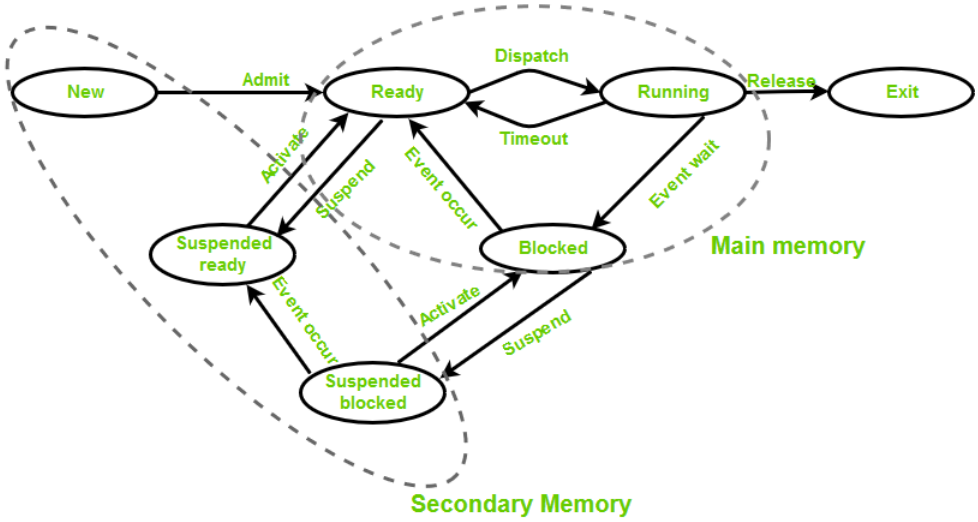


Figure 3.2.1: States of a Process. ("States of a Process" by Aniket_Dusey, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

3.2: Process States is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

3.3 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.

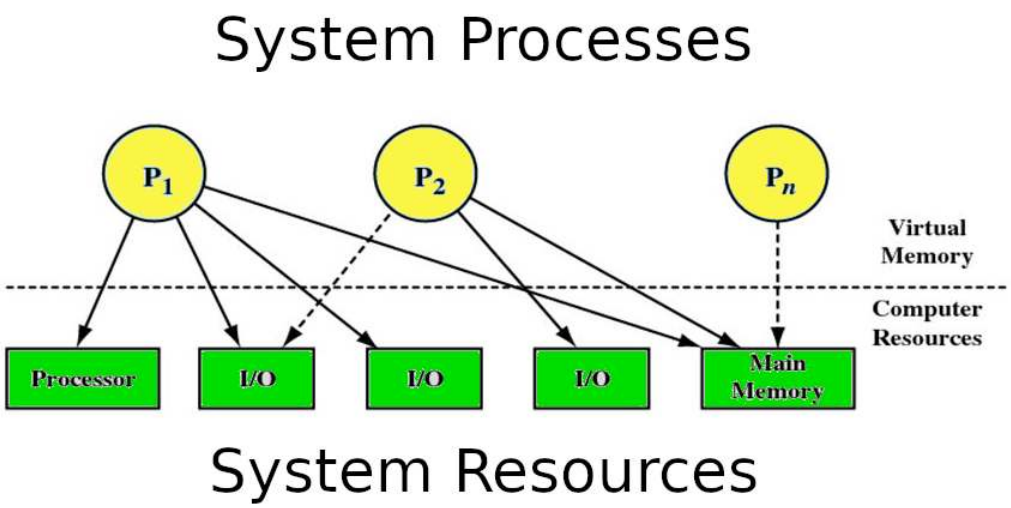


Figure 1: Processes and Resources. (Unknown source)

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

3.3 Process Description is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

3.4: 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.

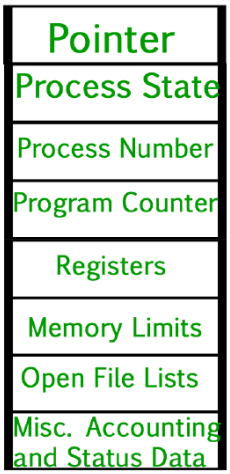


Figure 3.4.1: Process Control Block. ("Process Control Block" by magbene, Geeks for Geeks is licensed under CC BY-SA 4.0)

- **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.

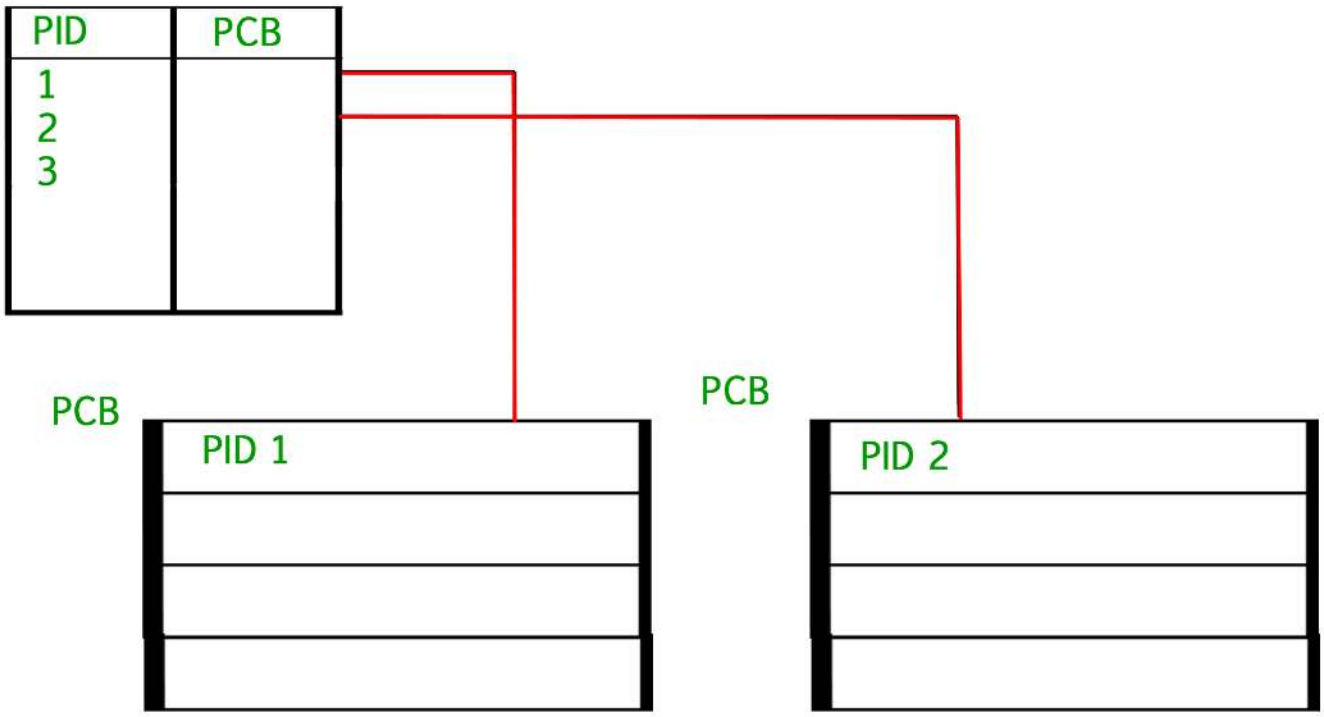


Figure 3.4.1: Process Table and Process Control Block. ("Process Table and Process Control Block" by magbene, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

3.4: Process Control is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

3.5: 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

3.5: Execution within the Operating System is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

3.5.1 : 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)

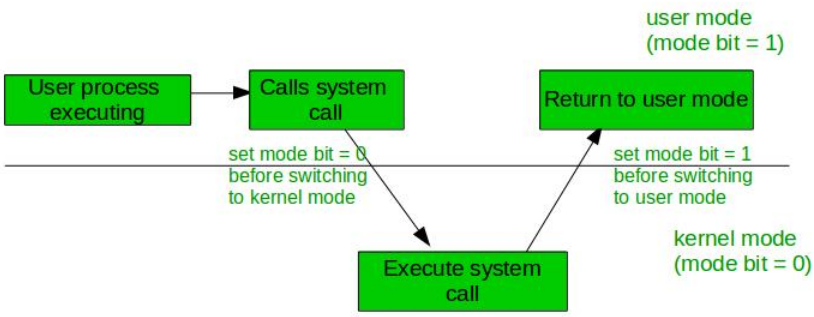


Figure 3.5.1.1: User Process Makes System Call. ("User Mode to Kernel Mode Switch" by shivani.mittal, Geeks for Geeks is licensed under CC BY-SA 4.0)

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.

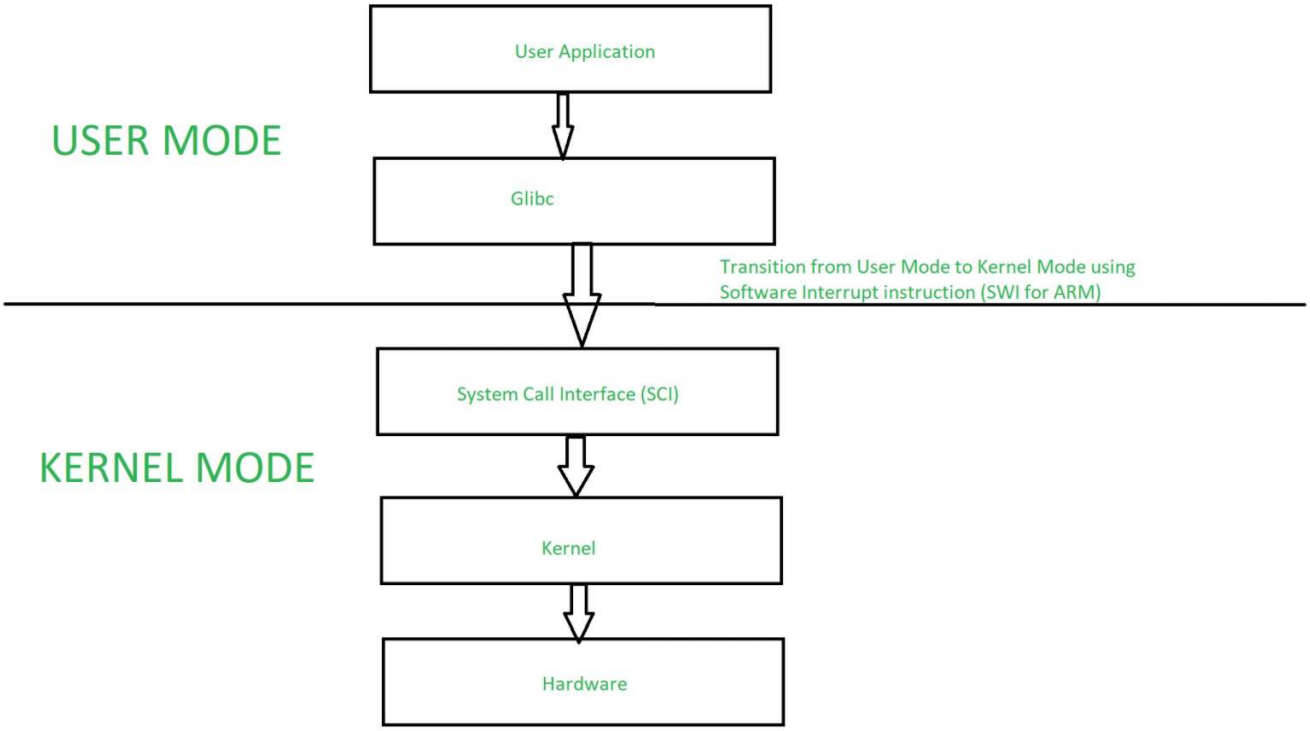


Figure 3.5.1.1: User Mode to Kernel Mode Transition. ("user mode to kernel mode transition" by sandeepjainlinux, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

3.5.1 : Execution within the Operating System - Dual Mode is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

4: Threads

4.1: Process and Threads

4.2: Thread Types

4.2.1: Thread Types - Models

4.3: Thread Relationships

4.4: Benefits of Multithreading

This page titled 4: Threads is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

4.1: 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

4.1: Process and Threads is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

4.2: 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.

4.2: Thread Types is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

4.2.1: 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.

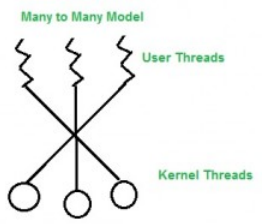


Figure 4.2.1.1: Many to Many Thread Model. ("Many to Many" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0)

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.

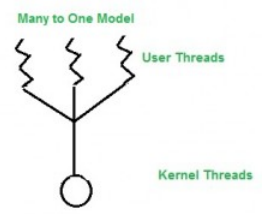


Figure 4.2.1.1: Many to One Thread Model. ("Many to One" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0)

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.

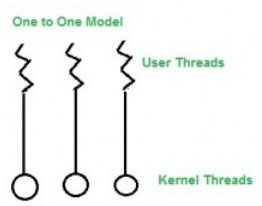


Figure 4.2.1.1: One to One Thread Model. ("One to One" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0)

Adapted from:
"Multi Threading Models in Process Management" by Unknown, Geeks for Geeks is licensed under CC BY-SA 4.0

4.2.1: Thread Types - Models is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

4.3: 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.

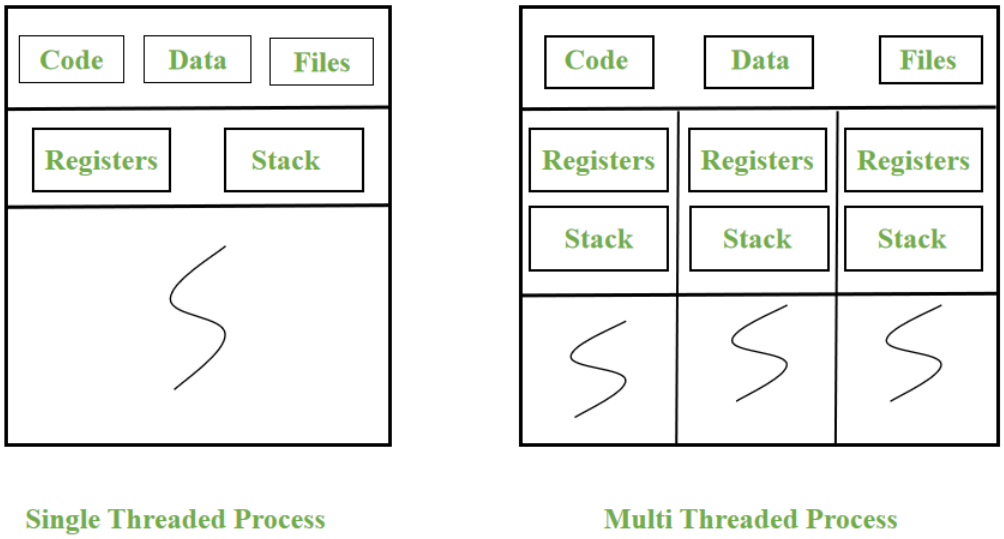


Figure 4.3.1: Single and Multi Thread Processes. ("SIngle versus Multi Threads" by maha93427, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

4.3: Thread Relationships is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

4.4: 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

4.4: Benefits of Multithreading is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

5: Concurrency and Process Synchronization

5.1: Introduction to Concurrency

5.2: Process Synchronization

5.3: Mutual Exclusion

5.4: Interprocess Communication

5.4.1: IPC - Semaphores

5.4.2: IPC - Monitors

5.4.3: IPC - Message Passing / Shared Memory

This page titled 5: Concurrency and Process Synchronization is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

5.1: 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.[1]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

5.1: Introduction to Concurrency is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.2: 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

5.2: Process Synchronization is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.3: 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

5.3: Mutual Exclusion is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.4: 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

5.4: Interprocess Communication is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.4.1: 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

5.4.1: IPC - Semaphores is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.4.2: 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

5.4.2: IPC - Monitors is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

5.4.3: 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.

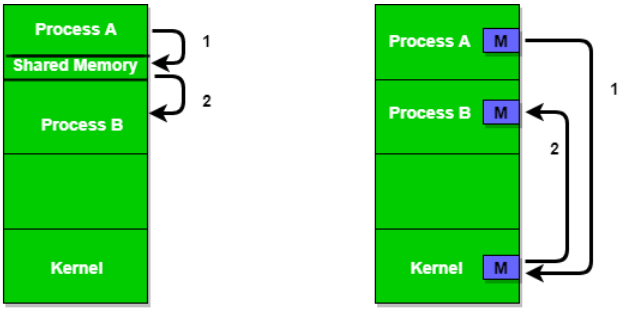


Figure 5.4.3.1: Shared Memory and Message Passing. ("Shared Memory and Message Passing" by ShubhamMaurya3, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

5.4.3: IPC - Message Passing / Shared Memory is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

6: Concurrency: Deadlock and Starvation

6.1: Concept and Principles of Deadlock
6.2: Deadlock Detection and Prevention
6.3: Starvation
6.4: Dining Philosopher Problem

This page titled 6: Concurrency: Deadlock and Starvation is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

6.1: 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.

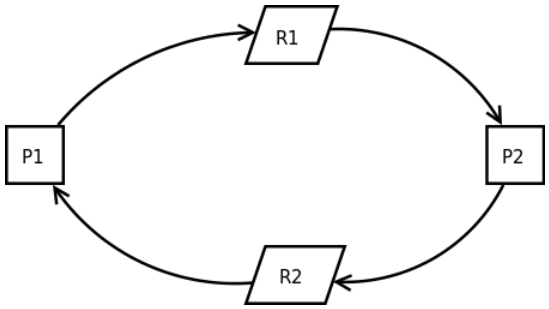


Figure 6.1.1: Both processes need resources to continue execution. *P1* requires additional resource *R1* and is in possession of resource *R2*, *P2* requires additional resource *R2* and is in possession of *R1*; neither process can continue.
("Process deadlock" by Wikimedia Commons is licensed under 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.

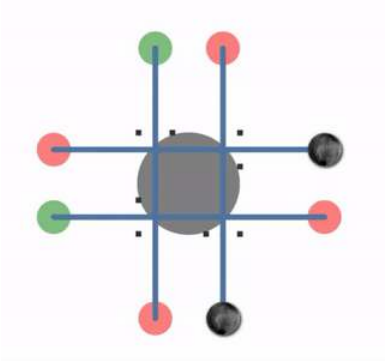


Figure 6.1.1: Four processes (blue lines) compete for one resource (grey circle), following a right-before-left policy. A deadlock occurs when all processes lock the resource simultaneously (black lines). The deadlock can be resolved by breaking the symmetry. ("Marble Machine" by Wikimedia Commons is licensed under CC BY-SA 4.0)

Adapted from:
"Deadlock" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0

6.1: Concept and Principles of Deadlock is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

6.2: 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.

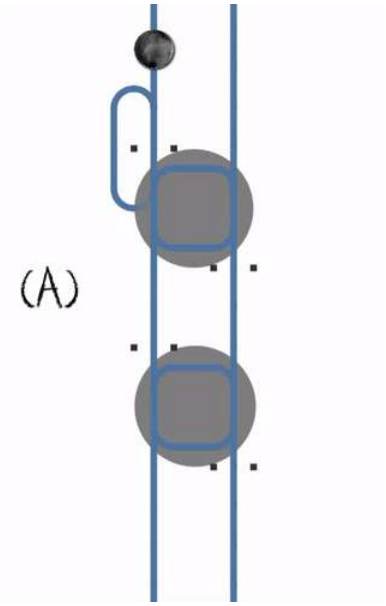


Figure 6.2.1: Two processes concurring for two resources. A deadlock occurs when the first process locks the first resource at the same time as the second process locks the second resource. The deadlock can be resolved by cancelling and restarting the first process.
("Two Processes - Two Resources" by 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.

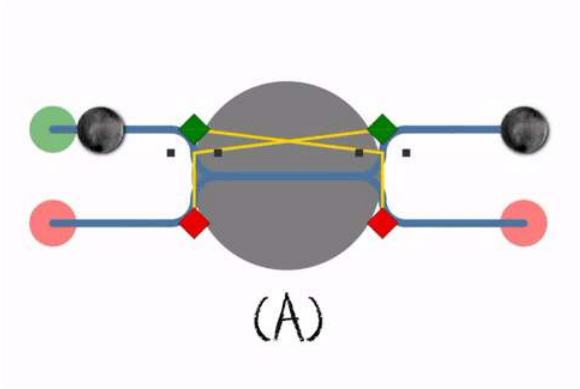


Figure 6.2.1: (A) Two processes concurring for one resource, following a first-come, first-served policy. (B) A deadlock occurs when both processes lock the resource simultaneously. (C) The deadlock can be resolved by breaking the symmetry of the locks. (D) The deadlock can be avoided by breaking the 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

6.2: Deadlock Detection and Prevention is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

6.3: 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

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

6.4: 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

6.4: Dining Philosopher Problem is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

7: Memory Management

7.1: Random Access Memory (RAM) and Read Only Memory (ROM)
7.2: Memory Hierarchy
7.3: Requirements for Memory Management
7.4: Memory Partitioning
7.4.1: Fixed Partitioning
7.4.2: Variable Partitioning
7.4.3: Buddy System
7.5: Logical vs Physical Address
7.6: Paging
7.7: Segmentation

This page titled 7: Memory Management is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

7.1: 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.

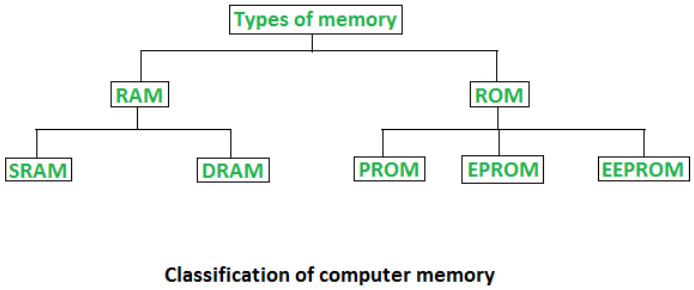


Figure 7.1.1: ("Classification of Computer Memory" by Deepanshi_Mittal, Geeks for Geeks is licensed under CC BY-SA 4.0)

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 7.1.1: 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 7.1.1: ("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 7.1: 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.

7.2: 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 :

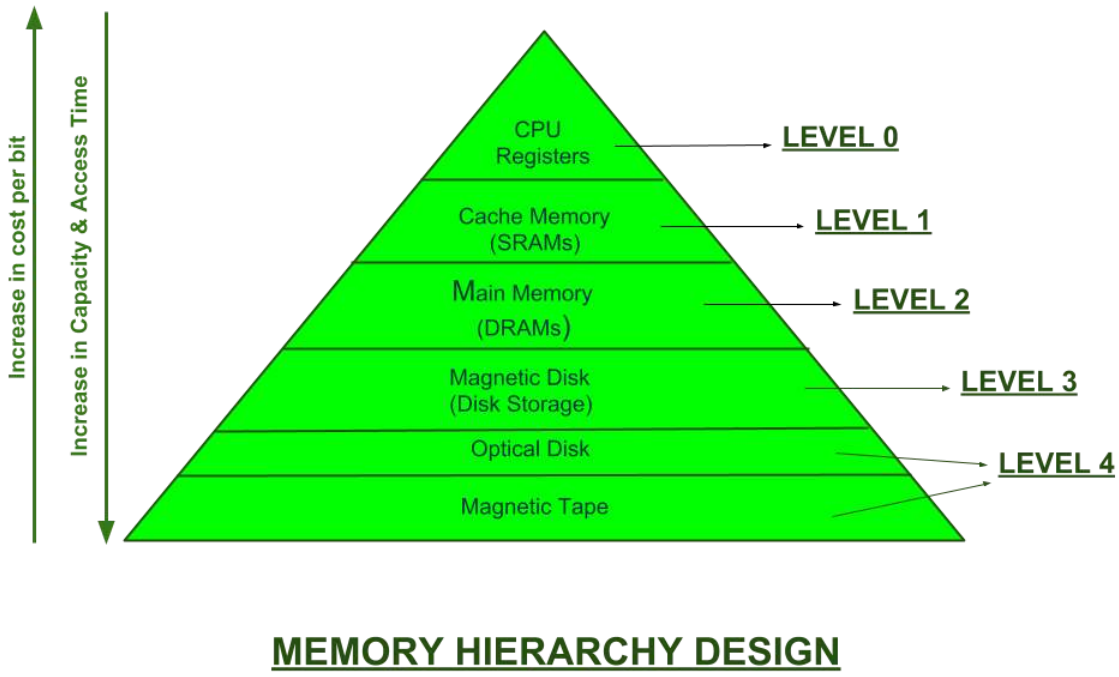


Figure 7.2.1: ("Memory Hierarchy" by RishabhJain12, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

7.2: Memory Hierarchy is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

7.3: 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.

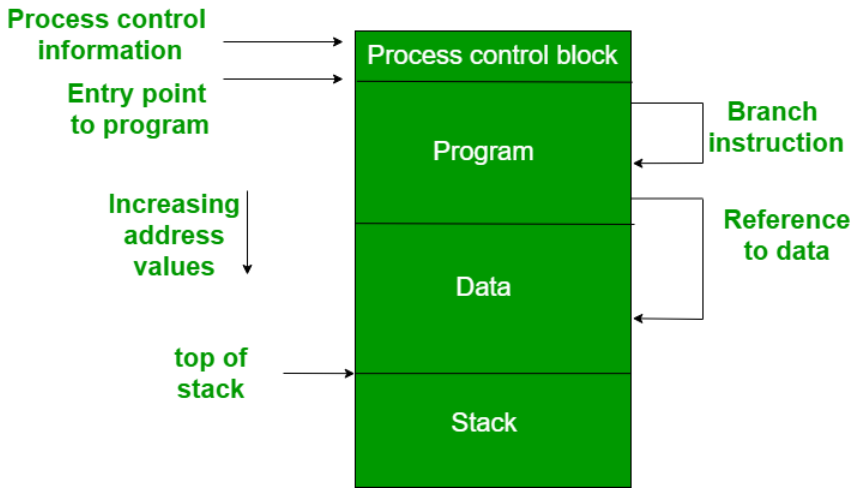


Figure 7.3.1: A process occupying a continuous region of main memory.
("Process Image" by Aditya_04, Geeks for Geeks is licensed under CC BY-SA 4.0)

   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

7.3: Requirements for Memory Management is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

7.4: 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 7.4.1: 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.

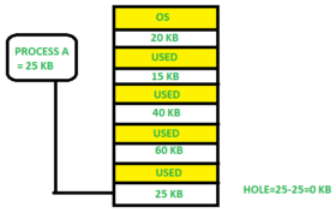


Figure 7.4.1: Best Fit. ("Best Fit" by deepakmkoshy, Geeks for Geeks is licensed under CC BY-SA 4.0)

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.

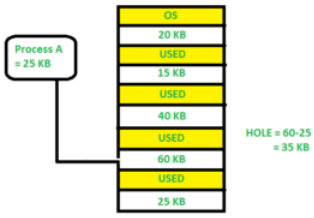


Figure 7.4.1: Worst Fit. ("Worst Fit" by deepakmkoshy, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

7.4: Memory Partitioning is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

7.4.1: 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.

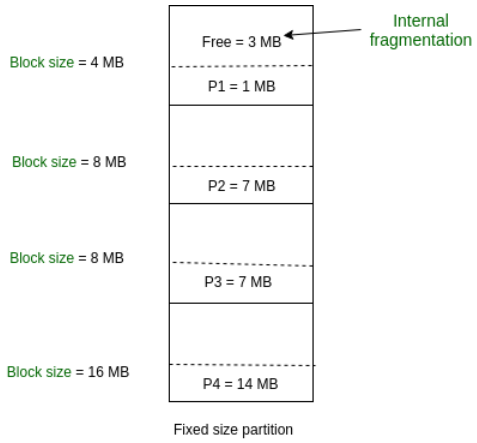


Figure 7.4.1.1: Fixed Sized partition Example. ("Fixed Sized Partition" by Vidhayak_Chacha, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

7.4.1: Fixed Partitioning is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

7.4.2: 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.

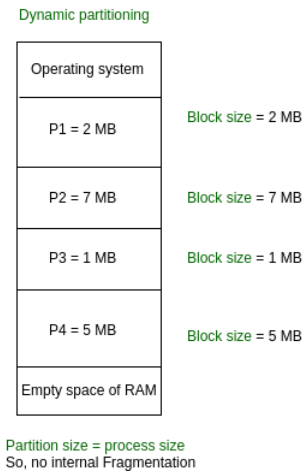


Figure 7.4.2.1: Variable Partitioned Memory. ("Variable Patitiong" by **Vidhayak_Chacha**, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

7.4.2: Variable Partitioning is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

7.4.3: 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

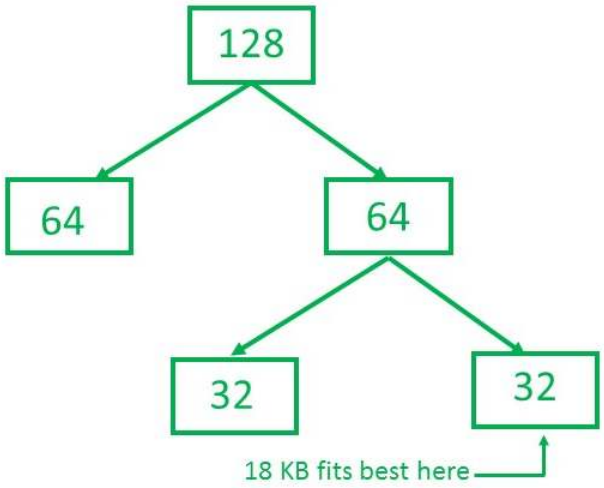


Figure 7.4.3.1: Buddy System Partitioning
("Buddy System Partitioning" by Samit Mandal, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

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

7.5: 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.

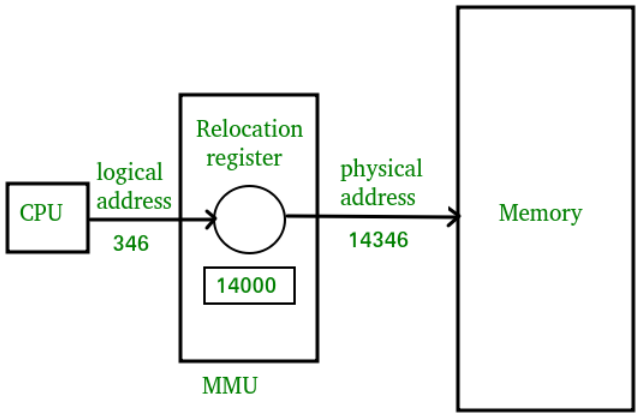


Figure 7.5.1: MMU Operation.
("MMU Operation" by Ankit_Bisht, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

7.5: Logical vs Physical Address is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

7.6: 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).

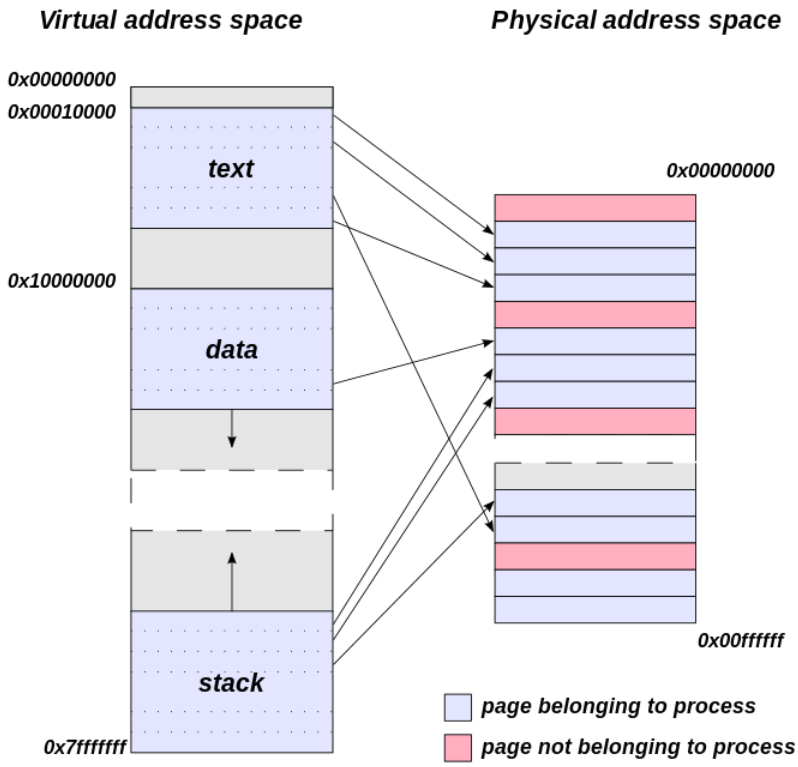


Figure 7.6.1: Mapping Virtual Memory to Physical Memory.
("Mapping Virtual Addresses to Physical Addresses" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0)

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.

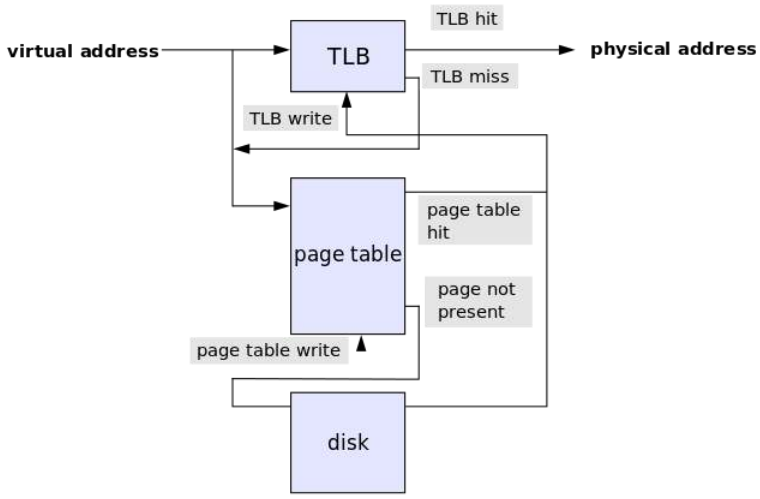


Figure 7.6.1: Actions taken upon a virtual to physical address translation request.
("Actions taken upon a virtual to physical address translation request" by Multiple Contributors, Wikipedia is licensed under CC BY-SA 3.0)

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

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

7.7: 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.

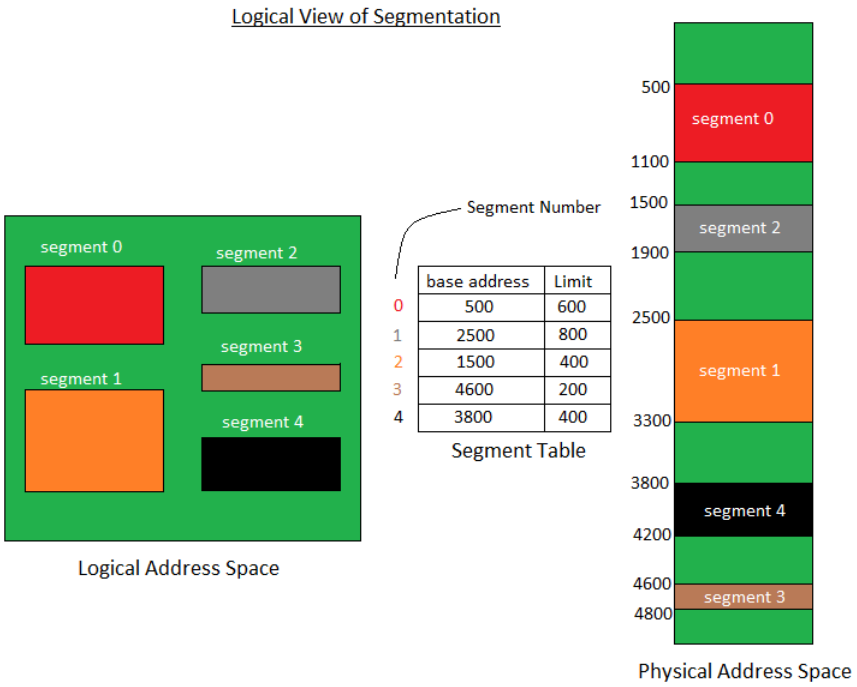


Figure 7.7.1: Segmentation Table Mapping to Physical Address.
("Segmentation Table Mapping to Physical Address" by VaibhavRai3, Geeks for Geeks is licensed under CC BY-SA 4.0)

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.

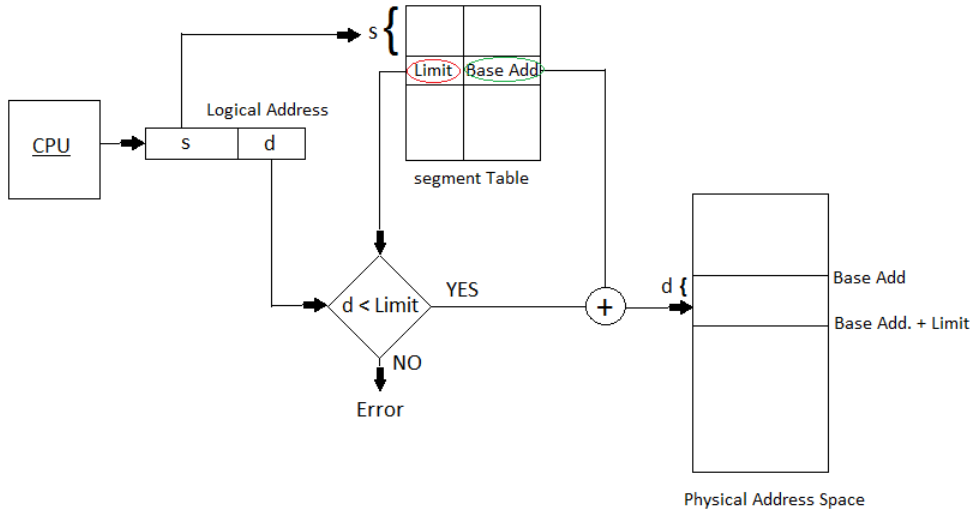


Figure 7.7.1: Translate Logical Address to Physical Address.
("Translate Logical Address to Physical Address" by VaibhavRai3, Geeks for Geeks is licensed under CC BY-SA 4.0)

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

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

CHAPTER OVERVIEW

8: Virtual Memory

8.1: Memory Paging

8.1.1: Memory Paging - Page Replacement

8.2: Virtual Memory in the Operating System

This page titled 8: Virtual Memory is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

8.1: 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

8.1: Memory Paging is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

8.1.1: 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

8.1.1: Memory Paging - Page Replacement is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

8.2: 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

8.2: Virtual Memory in the Operating System is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

9: Uniprocessor CPU Scheduling

9.1: Types of Processor Scheduling
9.2: Scheduling Algorithms

This page titled 9: Uniprocessor CPU Scheduling is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

9.1: 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

9.1: Types of Processor Scheduling is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

9.2: 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

9.2: Scheduling Algorithms is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

10: Multiprocessor Scheduling

10.1: The Question
10.2: Multiprocessor Scheduling

This page titled 10: Multiprocessor Scheduling is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

10.1: 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.

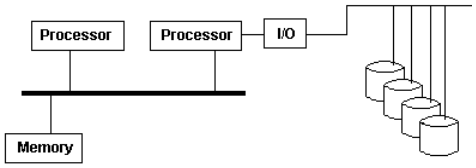


Figure 10.1.1: Asymmetric multiprocessing. ("Asmp 2.gif" by G7a, Wikimedia Commons is licensed under CC BY-SA 3.0)

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."

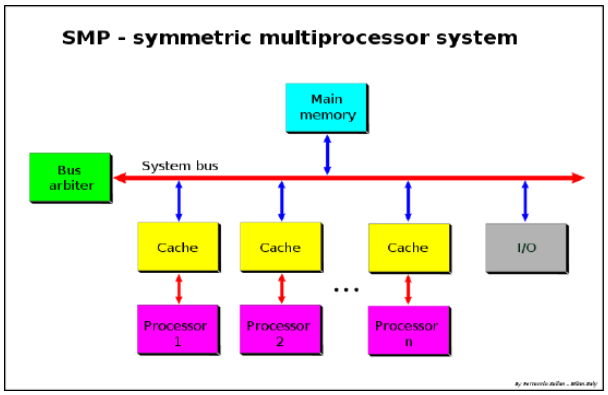


Figure 10.1.1: SMP - Symmetric Multiprocessor System. ("SMP - Symmetric Multiprocessor System" by Ferry24.Milan, Wikimedia Commons is licensed under CC BY-SA 3.0)

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

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

10.2: 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

10.2: Multiprocessor Scheduling is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

12: File Management

12.1: Overview

12.2: Files

12.2.1: Files (continued)

12.3: Directory

12.4: File Sharing

This page titled 12: File Management is shared under a CC BY-SA license and was authored, remixed, and/or curated by Patrick McClanahan.

12.1: 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.

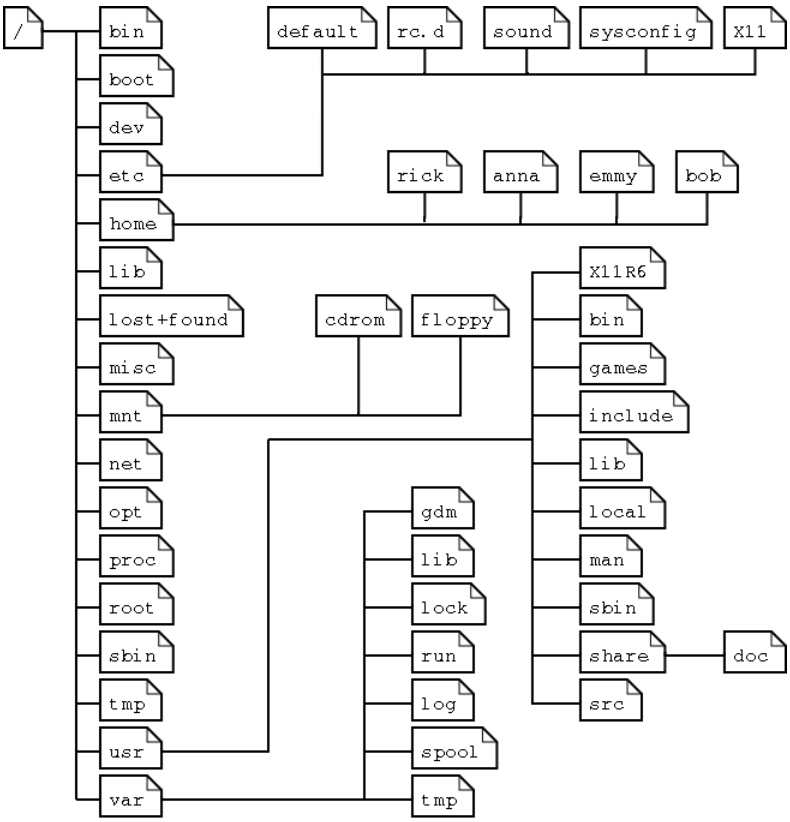


Figure 12.1.1: **Linux file system layout**. ("Linux file system layout" by Machtelt Garrels, The Linux Documentation Project is licensed under LDP Manifesto)

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

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

12.2: 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

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

12.2.1: 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

12.2.1: Files (continued) is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.

12.3: 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.

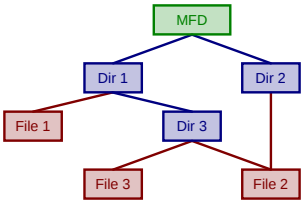


Figure 12.3.1: A typical Files-11 directory hierarchy. ("A typical Files-11 directory hierarchy." by Stannered, Wikimedia Commons is in the Public Domain, CC0)

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

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

12.4: 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

12.4: File Sharing is shared under a not declared license and was authored, remixed, and/or curated by LibreTexts.