

1140729專案



昭儀 江

Shih Chien University

作業系統（Operating System，簡稱OS）是**管理電腦硬體和軟體資源的系統軟體**，它為使用者和應用程式提供了一個介面，以便更方便地與電腦交互並執行各種任務。簡單來說，作業系統就像是電腦的靈魂，負責調度資源、管理檔案、提供使用者介面等，讓電腦可以正常運作。

作業系統的主要功能：

- **資源管理：**
- **程式執行：**
- **檔案管理：**
- **使用者介面：**
- **安全性：**

常見的作業系統類型：

- **個人電腦作業系統：**如Windows、macOS、Linux。
- **行動作業系統：**如Android、iOS。
- **伺服器作業系統：**如Linux、Windows Server。
- **嵌入式作業系統：**用於各種嵌入式設備，如智慧型手錶、汽車導航系統等。

作業系統的重要性：

作業系統是現代電腦系統的核心，它使得電腦能夠正常運作，並為使用者提供方便的交互方式和各種應用程式。沒有作業系統，電腦就無法執行任何程式，也無法提供任何功能。

總之，作業系統是電腦系統中不可或缺的一部分，它扮演著管理、控制和協調的角色，讓使用者可以更有效率地使用電腦。

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

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

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



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

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

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

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

This text was compiled on 05/15/2026

TABLE OF CONTENTS

Operating System

Licensing

1: 位元與位元組

- 1.1: Representing integers
- 1.2: Bitwise operators
- 1.3: Representing floating-point numbers
- 1.4: Unions and memory errors
- 1.5: Representing strings

2: Memory management

- 2.1: Memory errors
- 2.2: Memory leaks
- 2.3: Implementation

3: Caching

- 3.1: How programs run
- 3.2: Cache performance
- 3.3: Locality
- 3.4: Measuring cache performance
- 3.5: Programming for cache performance
- 3.6: The memory hierarchy
- 3.7: Caching policy
- 3.8: Paging

4: Threads

- 4.1: Creating threads
- 4.2: Creating threads
- 4.3: Joining threads
- 4.4: Synchronization errors
- 4.5: Mutex

Index

Glossary

Detailed Licensing

Detailed Licensing

Licensing

A detailed breakdown of this resource's licensing can be found in [Back Matter/Detailed Licensing](#).

CHAPTER OVERVIEW

1: 位元與位元組

- 1.1: Representing integers
- 1.2: Bitwise operators
- 1.3: Representing floating-point numbers
- 1.4: Unions and memory errors
- 1.5: Representing strings

1: 位元與位元組 is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by LibreTexts.

1.1: Representing integers

You probably know that computers represent numbers in base 2, also known as binary. For positive numbers, the binary representation is straightforward; for example, the representation for 5_{10} is $b101$.

For negative numbers, the most obvious representation uses a sign bit to indicate whether a number is positive or negative. But there is another representation, called “two’s complement” that is much more common because it is easier to work with in hardware.

To find the two’s complement of a negative number, $-x$, find the binary representation of x , flip all the bits, and add 1. For example, to represent -5_{10} , start with the representation of 5_{10} , which is $b0000\ 0101$ if we write the 8-bit version. Flipping all the bits and adding 1 yields $b1111\ 1011$.

In two’s complement, the leftmost bit acts like a sign bit; it is 0 for positive numbers and 1 for negative numbers.

To convert from an 8-bit number to 16-bits, we have to add more 0’s for a positive number and add 1’s for a negative number. In effect, we have to copy the sign bit into the new bits. This process is called “sign extension”.

In C all integer types are signed (able to represent positive and negative numbers) unless you declare them `unsigned`. The difference, and the reason this declaration is important, is that operations on unsigned integers don’t use sign extension.

This page titled [1.1: Representing integers](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)).

- [5.1: Representing integers](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

1.2: Bitwise operators

People learning C are sometimes confused about the bitwise operators `&` and `|`. These operators treat integers as bit vectors and compute logical operations on corresponding bits.

For example, `&` computes the AND operation, which yields 1 if both operands are 1, and 0 otherwise. Here is an example of `&` applied to two 4-bit numbers:

```
1100
& 1010
----
1000
```

In C, this means that the expression `12 & 10` has the value 8.

Similarly, `|` computes the OR operation, which yields 1 if either operand is 1, and 0 otherwise.

```
1100
| 1010
----
1110
```

So the expression `12 | 10` has the value 14.

Finally, `^` computes the XOR operation, which yields 1 if either operand is 1, but not both.

```
1100
^ 1010
----
0110
```

So the expression `12 ^ 10` has the value 6.

Most commonly, `&` is used to clear a set of bits from a bit vector, `|` is used to set bits, and `^` is used to flip, or “toggle” bits. Here are the details:

Clearing bits: For any value x , $x \& 0$ is 0, and $x \& 1$ is x . So if you AND a vector with 3, it selects only the two rightmost bits, and sets the rest to 0.

```
XXXX
& 0011
----
00xx
```

In this context, the value 3 is called a “mask” because it selects some bits and masks the rest.

Setting bits: Similarly, for any x , $x | 0$ is x , and $x | 1$ is 1. So if you OR a vector with 3, it sets the rightmost bits, and leaves the rest alone:

```
XXXX
| 0011
```

```
-----  
xx11
```

Toggling bits: Finally, if you XOR a vector with 3, it flips the rightmost bits and leaves the rest alone. As an exercise, see if you can compute the two's complement of 12 using `^`. Hint: what's the two's complement representation of -1?

C also provides shift operators, `<<` and `>>`, which shift bits left and right. Each left shift doubles a number, so `5 << 1` is 10, and `5 << 2` is 20. Each right shift divides by two (rounding down), so `5 >> 1` is 2 and `2 >> 1` is 1.

This page titled [1.2: Bitwise operators](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)).

- [5.2: Bitwise operators](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

1.3: Representing floating-point numbers

Floating-point numbers are represented using the binary version of scientific notation. In decimal notation, large numbers are written as the product of a coefficient and 10 raised to an exponent. For example, the speed of light in m/s is approximately $2.998 \cdot 10^8$.

Most computers use the IEEE standard for floating-point arithmetic. The C type float usually corresponds to the 32-bit IEEE standard; double usually corresponds to the 64-bit standard.

In the 32-bit standard, the leftmost bit is the sign bit, s . The next 8 bits are the exponent, q , and the last 23 bits are the coefficient, c . The value of a floating-point number is

$$(-1)^s \cdot 2^q$$

Well, that's almost correct, but there's one more wrinkle. Floating-point numbers are usually normalized so that there is one digit before the point. For example, in base 10, we prefer $2.998 \cdot 10^8$ rather than $2998 \cdot 10^5$ or any other equivalent expression. In base 2, a normalized number always has the digit 1 before the binary point. Since the digit in this location is always 1, we can save space by leaving it out of the representation.

For example, the integer representation of 13_{10} is $b1101$. In floating point, that's $1.101 \cdot 2^3$, so the exponent is 3 and the part of the coefficient that would be stored is 101 (followed by 20 zeros).

Well, that's almost correct, but there's one more wrinkle. The exponent is stored with a "bias". In the 32-bit standard, the bias is 127, so the exponent 3 would be stored as 130.

To pack and unpack floating-point numbers in C, we can use a union and bitwise operations. Here's an example:

```
union {
    float f;
    unsigned int u;
} p;

p.f = -13.0;
unsigned int sign = (p.u >> 31) & 1;
unsigned int exp = (p.u >> 23) & 0xff;

unsigned int coef_mask = (1 << 23) - 1;
unsigned int coef = p.u & coef_mask;

printf("%d\n", sign);
printf("%d\n", exp);
printf("0x%x\n", coef);
```

This code is in `float.c` in the repository for this book (see Section 0.2).

The union allows us to store a floating-point value using `p.f` and then read it as an unsigned integer using `p.u`.

To get the sign bit, we shift the bits to the right 31 places and then use a 1-bit mask to select only the rightmost bit.

To get the exponent, we shift the bits 23 places, then select the rightmost 8 bits (the hexadecimal value `0xff` has eight 1's).

To get the coefficient, we need to extract the 23 rightmost bits and ignore the rest. We do that by making a mask with 1s in the 23 rightmost places and 0s on the left. The easiest way to do that is by shifting 1 to the left by 23 places and then subtracting 1.

The output of this program is:

```
1
130
0x500000
```

As expected, the sign bit for a negative number is 1. The exponent is 130, including the bias. And the coefficient, which I printed in hexadecimal, is 101 followed by 20 zeros.

As an exercise, try assembling or disassembling a `double`, which uses the 64-bit standard. See http://en.Wikipedia.org/wiki/IEEE_floating_point.

This page titled [1.3: Representing floating-point numbers](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) (Green Tea Press).

- [5.3: Representing floating-point numbers](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

1.4: Unions and memory errors

There are two common uses of C unions. One, which we saw in the previous section, is to access the binary representation of data. Another is to store heterogeneous data. For example, you could use a union to represent a number that might be an integer, float, complex, or rational number.

However, unions are error-prone. It is up to you, as the programmer, to keep track of what type of data is in the union; if you write a floating-point value and then interpret it as an integer, the result is usually nonsense.

Actually, the same thing can happen if you read a location in memory incorrectly. One way that can happen is if you read past the end of an array.

To see what happens, I'll start with a function that allocates an array on the stack and fills it with the numbers from 0 to 99.

```
void f1() {
    int i;
    int array[100];

    for (i=0; i<100; i++) {
        array[i] = i;
    }
}
```

Next I'll define a function that creates a smaller array and deliberately accesses elements before the beginning and after the end:

```
void f2() {
    int x = 17;
    int array[10];
    int y = 123;

    printf("%d\n", array[-2]);
    printf("%d\n", array[-1]);
    printf("%d\n", array[10]);
    printf("%d\n", array[11]);
}
```

If I call `f1` and then `f2`, I get these results:

```
17
123
98
99
```

The details here depend on the compiler, which arranges variables on the stack. From these results, we can infer that the compiler put `x` and `y` next to each other, "below" the array (at a lower address). And when we read past the array, it looks like we are getting values that were left on the stack by the previous function call.

In this example, all of the variables are integers, so it is relatively easy to figure out what is going on. But in general when you read beyond the bounds of an array, the values you read might have any type. For example, if I change `f1` to make an array of floats, the results are:

```
17
123
1120141312
1120272384
```

The latter two values are what you get if you interpret a floating-point value as an integer. If you encountered this output while debugging, you would have a hard time figuring out what's going on.

This page titled [1.4: Unions and memory errors](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)).

- [5.4: Unions and memory errors](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

1.5: Representing strings

Related issues sometimes come up with strings. First, remember that C strings are null-terminated. When you allocate space for a string, don't forget the extra byte at the end.

Also, the letters *and numbers* in C strings are encoded in ASCII. The ASCII codes for the digits “0” through “9” are 48 through 57, *not* 0 through 9. The ASCII code 0 is the NUL character that marks the end of a string. And the ASCII codes 1 through 9 are special characters used in some communication protocols. ASCII code 7 is a bell; on some terminals, printing it makes a sound.

The ASCII code for the letter “A” is 65; the code for “a” is 97. Here are those codes in binary:

```
65 = b0100 0001
97 = b0110 0001
```

A careful observer will notice that they differ by a single bit. And this pattern holds for the rest of the letters; the sixth bit (counting from the right) acts as a “case bit”, 0 for upper-case letters and 1 for lower case letters.

As an exercise, write a function that takes a string and converts from lower-case to upper-case by flipping the sixth bit. As a challenge, you can make a faster version by reading the string 32 or 64 bits at a time, rather than one character at a time. This optimization is made easier if the length of the string is a multiple of 4 or 8 bytes.

If you read past the end of a string, you are likely to see strange characters. Conversely, if you write a string and then accidentally read it as an int or float, the results will be hard to interpret.

For example, if you run:

```
char array[] = "allen";
float *p = array;
printf("%f\n", *p);
```

You will find that the ASCII representation of the first 8 characters of my name, interpreted as a double-precision floating point number, is 69779713878800585457664.

This page titled [1.5: Representing strings](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

- [5.5: Representing strings](#) by Allen B. Downey is licensed [CC BY-NC 4.0](#).

CHAPTER OVERVIEW

2: Memory management

C provides 4 functions for dynamic memory allocation:

- `malloc` , which takes an integer size, in bytes, and returns a pointer to a newly-allocated chunk of memory with (at least) the given size. If it can't satisfy the request, it returns the special pointer value `NULL`.
- `calloc` , which is the same as `malloc` except that it also clears the newly allocated chunk; that is, it sets all bytes in the chunk to 0.
- `free` , which takes a pointer to a previously allocated chunk and deallocates it; that is, it makes the space available for future allocation.
- `realloc` , which takes a pointer to a previously allocated chunk and a new size. It allocates a chunk of memory with the new size, copies data from the old chunk to the new, frees the old chunk, and returns a pointer to the new chunk.

This API is notoriously error-prone and unforgiving. Memory management is one of the most challenging parts of designing large software systems, which is why most modern languages provide higher-level memory management features like garbage collection.

[2.1: Memory errors](#)

[2.2: Memory leaks](#)

[2.3: Implementation](#)

This page titled [2: Memory management](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

2.1: Memory errors

The C memory management API is a bit like Jasper Beardly, a minor character on the animated television program *The Simpsons*; in a few episodes, he appears as a strict substitute teacher who imposes corporal punishment — a “paddlin” — for all infractions.

Here are some of things a program can do that deserve a paddling:

- If you access (read or write) any chunk that has not been allocated, that’s a paddling.
- If you free an allocated chunk and then access it, that’s a paddling.
- If you try to free a chunk that has not been allocated, that’s a paddling.
- If you free the same chunk more than once, that’s a paddling.
- If you call `realloc` with a chunk that was not allocated, or was allocated and then freed, that’s a paddling.

It might not sound difficult to follow these rules, but in a large program a chunk of memory might be allocated in one part of the program, used in several other parts, and freed in yet another part. So changes in one part of the program can require changes in many other parts.

Also, there might be many aliases, or references to the same allocated chunk, in different parts of the program. The chunk should not be freed until all references to the chunk are no longer in use. Getting this right often requires careful analysis across all parts of the program, which is difficult and contrary to fundamental principles of good software engineering.

Ideally, every function that allocates memory should include, as part of the documented interface, information about how that memory is supposed to be freed. Mature libraries often do this well, but in the real world, software engineering practice often falls short of this ideal.

To make matters worse, memory errors can be difficult to find because the symptoms are unpredictable. For example:

- If you read a value from an unallocated chunk, the system *might* detect the error, trigger a runtime error called a “segmentation fault”, and stop the program. Or, the program might read unallocated memory without detecting the error; in that case, the value it gets is whatever happened to be stored at the accessed location, which is unpredictable, and might be different each time the program runs.
- If you write a value to an unallocated chunk, and don’t get a segmentation fault, things are even worse. After you write a value to an invalid location, a long time might pass before it is read and causes problems. At that point it will be very difficult to find the source of the problem.

And things can be even worse than that! One of the most common problems with C-style memory management is that the data structures used to implement `malloc` and `free` (which we will see soon) are often stored along with the allocated chunks. So if you accidentally write past the end of a dynamically-allocated chunk, you are likely to mangle these data structures. The system usually won’t detect the problem until later, when you call `malloc` or `free`, and those functions fail in some inscrutable way.

One conclusion you should draw from this is that safe memory management requires design and discipline. If you write a library or module that allocates memory, you should also provide an interface to free it, and memory management should be part of the API design from the beginning.

If you use a library that allocates memory, you should be disciplined in your use of the API. For example, if the library provides functions to allocate and deallocate storage, you should use those functions and not, for example, call `free` on a chunk you did not `malloc`. And you should avoid keeping multiple references to the same chunk in different parts of your program.

Often there is a trade-off between safe memory management and performance. For example, the most common source of memory errors is writing beyond the bounds of an array. The obvious remedy for this problem is bounds checking; that is, every access to the array should check whether the index is out of bounds. High-level libraries that provide array-like structures usually perform bounds checking. But C arrays and most low-level libraries do not.

This page titled [2.1: Memory errors](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey \(Green Tea Press\)](#).

- [6.1: Memory errors](#) by Allen B. Downey is licensed [CC BY-NC 4.0](#).

2.2: Memory leaks

There is one more memory error that may or may not deserve a paddling. If you allocate a chunk of memory and never free it, that's a "memory leak".

For some programs, memory leaks are ok. For example, if your program allocates memory, performs computations on it, and then exits, it is probably not necessary to free the allocated memory. When the program exits, all of its memory is deallocated by the operating system. Freeing memory immediately before exiting might feel more responsible, but it is mostly a waste of time.

But if a program runs for a long time and leaks memory, its total memory use will increase indefinitely. At that point, a few things might happen:

- At some point, the system runs out of physical memory. On systems without virtual memory, the next call to `malloc` will fail, returning `NULL`.
- On systems with virtual memory, the operating system can move another process's pages from memory to disk and then allocate more space to the leaking process. I explain this mechanism in Section 7.8.
- There might be a limit on the amount of space a single process can allocate; beyond that, `malloc` returns `NULL`.
- Eventually, a process might fill its virtual address space (or the usable part). After that, there are no more addresses to allocate, so `malloc` returns `NULL`.

If `malloc` returns `NULL`, but you persist and access the chunk you think you allocated, you get a segmentation fault. For this reason, it is considered good style to check the result from `malloc` before using it. One option is to add a condition like this after every `malloc` call:

```
void *p = malloc(size);
if (p == NULL) {
    perror("malloc failed");
    exit(-1);
}
```

`perror` is declared in `stdio.h`; it prints an error message and additional information about the last error that occurred.

`exit`, which is declared in `stdlib.h`, causes the process to terminate. The argument is a status code that indicates how the process terminated. By convention, status code 0 indicates normal termination and -1 indicates an error condition. Sometimes other codes are used to indicate different error conditions.

Error-checking code can be a nuisance, and it makes programs harder to read. You can mitigate these problems by wrapping library function calls and their error-checking code in your own functions. For example, here is a `malloc` wrapper that checks the return value.

```
void *check_malloc(int size)
{
    void *p = malloc (size);
    if (p == NULL) {
        perror("malloc failed");
        exit(-1);
    }
    return p;
}
```

Because memory management is so difficult, most large programs, like web browsers, leak memory. To see which programs on your system are using the most memory, you can use the UNIX utilities `ps` and `top`.

This page titled [2.2: Memory leaks](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

- **6.2: Memory leaks** by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

2.3: Implementation

When a process starts, the system allocates space for the text segment and statically allocated data, space for the stack, and space for the heap, which contains dynamically allocated data.

Not all programs allocate data dynamically, so the initial size of the heap might be small or zero. Initially the heap contains only one free chunk.

When `malloc` is called, it checks whether it can find a free chunk that's big enough. If not, it has to request more memory from the system. The function that does that is `sbrk`, which sets the “program break”, which you can think of as a pointer to the end of the heap.

When `sbrk` is called, the OS allocates new pages of physical memory, updates the process's page table, and sets the program break.

In theory, a program could call `sbrk` directly (without using `malloc`) and manage the heap itself. But `malloc` is easier to use and, for most memory-use patterns, it runs fast and uses memory efficiently.

To implement the memory management API (that is, the functions `malloc`, `free`, `calloc`, and `realloc`), most Linux systems use `ptmalloc`, which is based on `dlmalloc`, written by Doug Lea. A short paper that describes key elements of the implementation is available at <http://gee.cs.oswego.edu/dl/html/malloc.html>.

For programmers, the most important elements to be aware of are:

- The run time of `malloc` does not usually depend on the size of the chunk, but might depend on how many free chunks there are. `free` is usually fast, regardless of the number of free chunks. Because `calloc` clears every byte in the chunk, the run time depends on chunk size (as well as the number of free chunks).

`realloc` is sometimes fast, if the new size is smaller than the current size, or if space is available to expand the existing chunk. If not, it has to copy data from the old chunk to the new; in that case, the run time depends on the size of the old chunk.

- Boundary tags: When `malloc` allocates a chunk, it adds space at the beginning and end to store information about the chunk, including its size and the state (allocated or free). These bits of data are called “boundary tags”. Using these tags, `malloc` can get from any chunk to the previous chunk and the next chunk in memory. In addition, free chunks are chained into a doubly-linked list; each free chunk contains pointers to the next and previous chunks in the “free list”.

The boundary tags and free list pointers make up `malloc`'s internal data structures. These data structures are interspersed with program data, so it is easy for a program error to damage them.

- Space overhead: Boundary tags and free list pointers take up space. The minimum chunk size on most systems is 16 bytes. So for very small chunks, `malloc` is not space efficient. If your program requires large numbers of small structures, it might be more efficient to allocate them in arrays.
- Fragmentation: If you allocate and free chunks with varied sizes, the heap will tend to become fragmented. That is, the free space might be broken into many small pieces. Fragmentation wastes space; it also slows the program down by making memory caches less effective.
- Binning and caching: The free list is sorted by size into bins, so when `malloc` searches for a chunk with a particular size, it knows what bin to search in. If you free a chunk and then immediately allocate a chunk with the same size, `malloc` will usually be fast.

This page titled [2.3: Implementation](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)).

- [6.3: Implementation](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

CHAPTER OVERVIEW

3: Caching

- 3.1: How programs run
- 3.2: Cache performance
- 3.3: Locality
- 3.4: Measuring cache performance
- 3.5: Programming for cache performance
- 3.6: The memory hierarchy
- 3.7: Caching policy
- 3.8: Paging

This page titled [3: Caching](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

3.1: How programs run

In order to understand caching, you have to understand how computers execute programs. For a deep understanding of this topic, you should study computer architecture. My goal in this chapter is to provide a simple model of program execution.

When a program starts, the code (or text) is usually on a hard disk or solid state drive. The operating system creates a new process to run the program, then the “loader” copies the text from storage into main memory and starts the program by calling `main`.

While the program is running, most of its data is stored in main memory, but some of the data is in registers, which are small units of memory on the CPU. These registers include:

- The program counter, or PC, which contains the address (in memory) of the next instruction in the program.
- The instruction register, or IR, which contains the machine code instruction currently executing.
- The stack pointer, or SP, which contains the address of the stack frame for the current function, which contains its parameters and local variables.
- General-purpose registers that hold the data the program is currently working with.
- A status register, or flag register, that contains information about the current computation. For example, the flag register usually contains a bit that is set if the result of the previous operation was zero.

When a program is running, the CPU executes the following steps, called the “instruction cycle”:

- Fetch: The next instruction is fetched from memory and stored in the instruction register.
- Decode: Part of the CPU, called the “control unit”, decodes the instruction and sends signals to the other parts of the CPU.
- Execute: Signals from the control unit cause the appropriate computation to occur.

Most computers can execute a few hundred different instructions, called the “instruction set”. But most instructions fall into a few general categories:

- Load: Transfers a value from memory to a register.
- Arithmetic/logic: Loads operands from registers, performs a mathematical operation, and stores the result in a register.
- Store: Transfers a value from a register to memory.
- Jump/branch: Changes the program counter, causing the flow of execution to jump to another location in the program. Branches are usually conditional, which means that they check a flag in the flag register and jump only if it is set.

Some instructions sets, including the ubiquitous x86, provide instructions that combine a load and an arithmetic operation.

During each instruction cycle, one instruction is read from the program text. In addition, about half of the instructions in a typical program load or store data. And therein lies one of the fundamental problems of computer architecture: the “memory bottleneck”.

In current computers, a typical core is capable of executing an instruction in less than 1 ns. But the time it takes to transfer data to and from memory is about 100 ns. If the CPU has to wait 100 ns to fetch the next instruction, and another 100 ns to load data, it would complete instructions 200 times slower than what’s theoretically possible. For many computations, memory is the speed limiting factor, not the CPU.

This page titled [3.1: How programs run](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)).

- [7.1: How programs run](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

3.2: Cache performance

The solution to this problem, or at least a partial solution, is caching. A “cache” is a small, fast memory that is physically close to the CPU, usually on the same chip.

Actually, current computers typically have several levels of cache: the Level 1 cache, which is the smallest and fastest, might be 1–2 MiB with a access times near 1 ns; the Level 2 cache might have access times near 4 ns, and the Level 3 might take 16 ns.

When the CPU loads a value from memory, it stores a copy in the cache. If the same value is loaded again, the CPU gets the cached copy and doesn’t have to wait for memory.

Eventually the cache gets full. Then, in order to bring something new in, we have to kick something out. So if the CPU loads a value and then loads it again much later, it might not be in cache any more.

The performance of many programs is limited by the effectiveness of the cache. If the instructions and data needed by the CPU are usually in cache, the program can run close to the full speed of the CPU. If the CPU frequently needs data that are not in cache, the program is limited by the speed of memory.

The cache “hit rate”, h , is the fraction of memory accesses that find data in cache; the “miss rate”, m , is the fraction of memory accesses that have to go to memory. If the time to process a cache hit is T_h and the time for a cache miss is T_m , the average time for each memory access is

$$hT_h + mT_m$$

Equivalently, we could define the “miss penalty” as the extra time to process a cache miss, $T_p = T_m - T_h$. Then the average access time is

$$T_h + mT_p$$

When the miss rate is low, the average access time can be close to T_h . That is, the program can perform as if memory ran at cache speeds.

This page titled [3.2: Cache performance](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

- [7.2: Cache performance](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

3.3: Locality

When a program reads a byte for the first time, the cache usually loads a “block” or “line” of data that includes the requested byte and some of its neighbors. If the program goes on to read one of the neighbors, it will already be in cache.

As an example, suppose the block size is 64 B; you read a string with length 64, and the first byte of the string happens to fall at the beginning of a block. When you load the first byte, you incur a miss penalty, but after that the rest of the string will be in cache. After reading the whole string, the hit rate will be 63/64, about 98%. If the string spans two blocks, you would incur 2 miss penalties. But even then the hit rate would be 62/64, or almost 97%. If you then read the same string again, the hit rate would be 100%.

On the other hand, if the program jumps around unpredictably, reading data from scattered locations in memory, and seldom accessing the same location twice, cache performance would be poor.

The tendency of a program to use the same data more than once is called “temporal locality”. The tendency to use data in nearby locations is called “spatial locality”. Fortunately, many programs naturally display both kinds of locality:

- Most programs contain blocks of code with no jumps or branches. Within these blocks, instructions run sequentially, so the access pattern has spatial locality.
- In a loop, programs execute the same instructions many times, so the access pattern has temporal locality.
- The result of one instruction is often used immediately as an operand of the next instruction, so the data access pattern has temporal locality.
- When a program executes a function, its parameters and local variables are stored together on the stack; accessing these values has spatial locality.
- One of the most common processing patterns is to read or write the elements of an array sequentially; this pattern also has spatial locality.

The next section explores the relationship between a program’s access pattern and cache performance.

This page titled [3.3: Locality](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

- [7.3: Locality](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

3.4: Measuring cache performance

When I was a graduate student at U.C. Berkeley I was a teaching assistant for Computer Architecture with Brian Harvey. One of my favorite exercises involved a program that iterates through an array and measures the average time to read and write an element. By varying the size of the array, it is possible to infer the size of the cache, the block size, and some other attributes.

My modified version of this program is in the cache directory of the repository for this book (see Section 0.2).

The important part of the program is this loop:

```

iters = 0;
do {
    sec0 = get_seconds();

    for (index = 0; index < limit; index += stride)
        array[index] = array[index] + 1;

    iters = iters + 1;
    sec = sec + (get_seconds() - sec0);

} while (sec < 0.1);

```

The inner `for` loop traverses the array. `limit` determines how much of the array it traverses; `stride` determines how many elements it skips over. For example, if `limit` is 16 and `stride` is 4, the loop would access elements 0, 4, 8, and 12.

`sec` keeps track of the total CPU time used by the inner loop. The outer loop runs until `sec` exceeds 0.1 seconds, which is long enough that we can compute the average time with sufficient precision.

`get_seconds` uses the system call `clock_gettime`, converts to seconds, and returns the result as a `double`:

```

double get_seconds(){
    struct timespec ts;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts);
    return ts.tv_sec + ts.tv_nsec / 1e9;
}

```

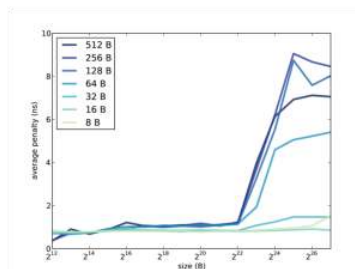


Figure 3.4.1: Average miss penalty as a function of array size and stride.

To isolate the time to access the elements of the array, the program runs a second loop that is almost identical except that the inner loop doesn't touch the array; it always increments the same variable:

```

iters2 = 0;
do {
    sec0 = get_seconds();

```

```
    for (index = 0; index < limit; index += stride)
        temp = temp + index;

    iters2 = iters2 + 1;
    sec = sec - (get_seconds() - sec0);

} while (iters2 < iters);
```

The second loop runs the same number of iterations as the first. After each iteration, it *subtracts* the elapsed time from `sec`. When the loop completes, `sec` contains the total time for all array accesses, minus the total time it took to increment `temp`. This difference is the total miss penalty incurred by all accesses. Finally, we divide by the number of accesses to get the average miss penalty per access, in ns:

```
sec * 1e9 / iters / limit * stride
```

If you compile and run `cache.c` you should see output like this:

```
Size:    4096 Stride:    8 read+write: 0.8633 ns
Size:    4096 Stride:   16 read+write: 0.7023 ns
Size:    4096 Stride:   32 read+write: 0.7105 ns
Size:    4096 Stride:   64 read+write: 0.7058 ns
```

If you have Python and `matplotlib` installed, you can use `graph_data.py` to graph the results. Figure 3.4.1 shows the results when I ran it on a Dell Optiplex 7010. Notice that the array size and stride are reported in bytes, not number of array elements.

Take a minute to consider this graph, and see what you can infer about the cache. Here are some things to think about:

- The program reads through the array many times, so it has plenty of temporal locality. If the entire array fits in cache, we expect the average miss penalty to be near 0.
- When the stride is 4 bytes, we read every element of the array, so the program has plenty of spatial locality. If the block size is big enough to contain 64 elements, for example, the hit rate would be 63/64, even if the array does not fit in cache.
- If the stride is equal to the block size (or greater), the spatial locality is effectively zero, because each time we read a block, we only access one element. In that case we expect to see the maximum miss penalty.

In summary, we expect good cache performance if the array is smaller than the cache size *or* if the stride is smaller than the block size. Performance only degrades if the array is bigger than the cache *and* the stride is large.

In Figure 3.4.1, cache performance is good, for all strides, as long as the array is less than 2^{22} B. We can infer that the cache size is near 4 MiB; in fact, according to the specs, it is 3 MiB.

When the stride is 8, 16, or 32 B, cache performance is good. At 64 B it starts to degrade, and for larger strides the average miss penalty is about 9 ns. We can infer that the block size near 128 B.

Many processors use “multi-level caches” that include a small, fast cache and a bigger, slower cache. In this example, it looks like the miss penalty increases a little when the array size is bigger than 2^{14} B, so it’s possible that this processor also has a 16 KB cache with an access time less than 1 ns.

This page titled [3.4: Measuring cache performance](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)).

- [7.4: Measuring cache performance](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

3.5: Programming for cache performance

Memory caching is implemented in hardware, so most of the time programmers don't need to know much about it. But if you know how caches work, you can write programs that use them more effectively.

For example, if you are working with a large array, it might be faster to traverse the array once, performing several operations with each element, rather than traversing the array several times.

If you are working with a 2-D array, it might be stored as an array of rows. If you traverse through the elements, it would be faster to go row-wise, with stride equal to the element size, rather than column-wise, with stride equal to the row length.

Linked data structures don't always exhibit spatial locality, because the nodes aren't necessarily contiguous in memory. But if you allocate many nodes at the same time, they are usually co-located in the heap. Or, even better, if you allocate an array of nodes all at once, you know they will be contiguous.

Recursive strategies like mergesort often have good cache behavior because they break big arrays into smaller pieces and then work with the pieces. Sometimes these algorithms can be tuned to take advantage of cache behavior.

For applications where performance is critical, it is possible to design algorithms tailored to the size of the cache, the block size, and other hardware characteristics. Algorithms like that are called "cache-aware". The obvious drawback of cache-aware algorithms is that they are hardware-specific.

This page titled [3.5: Programming for cache performance](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

- [7.5: Programming for cache performance](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

3.6: The memory hierarchy

At some point during this chapter, a question like the following might have occurred to you: “If caches are so much faster than main memory, why not make a really big cache and forget about memory?”

Without going too far into computer architecture, there are two reasons: electronics and economics. Caches are fast because they are small and close to the CPU, which minimizes delays due to capacitance and signal propagation. If you make a cache big, it will be slower.

Also, caches take up space on the processor chip, and bigger chips are more expensive. Main memory is usually dynamic random-access memory (DRAM), which uses only one transistor and one capacitor per bit, so it is possible to pack more memory into the same amount of space. But this way of implementing memory is slower than the way caches are implemented.

Also main memory is usually packaged in a dual in-line memory module (DIMM) that includes 16 or more chips. Several small chips are cheaper than one big one.

The trade-off between speed, size, and cost is the fundamental reason for caching. If there were one memory technology that was fast, big, and cheap, we wouldn’t need anything else.

The same principle applies to storage as well as memory. Solid state drives (SSD) are fast, but they are more expensive than hard drives (HDD), so they tend to be smaller. Tape drives are even slower than hard drives, but they can store large amounts of data relatively cheaply.

The following table shows typical access times, sizes, and costs for each of these technologies.

Table 3.6.1: Memory access times, sizes, and costs.

Device	Access time	Typical size	Cost
Register	0.5 ns	256 B	?
Cache	1 ns	2 MiB	?
DRAM	100 ns	4 GiB	\$10 / GiB
SSD	10 μ s	100 GiB	\$1 / GiB
HDD	5 ms	500 GiB	\$0.25 / GiB
Tape	minutes	1-2 TiB	\$0.02 / GiB

The number and size of registers depends on details of the architecture. Current computers have about 32 general-purpose registers, each storing one “word”. On a 32-bit computer, a word is 32 bits or 4 B. On a 64-bit computer, a word is 64 bits or 8 B. So the total size of the register file is 100–300 B.

The cost of registers and caches is hard to quantify. They contribute to the cost of the chips they are on, but consumers don’t see that cost directly.

For the other numbers in the table, I looked at the specifications for typical hardware for sale from online computer hardware stores. By the time you read this, these numbers will be obsolete, but they give you an idea of what the performance and cost gaps looked like at one point in time.

These technologies make up the “memory hierarchy” (note that this use of “memory” also includes storage). Each level of the hierarchy is bigger and slower than the one above it. And in some sense, each level acts as a cache for the one below it. You can think of main memory as a cache for programs and data that are stored permanently on SSDs and HDDs. And if you are working with very large datasets stored on tape, you could use hard drives to cache one subset of the data at a time.

This page titled [3.6: The memory hierarchy](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

- [7.6: The memory hierarchy](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

3.7: Caching policy

The memory hierarchy suggests a framework for thinking about caching. At every level of the hierarchy, we have to address four fundamental questions of caching:

- Who moves data up and down the hierarchy? At the top of the hierarchy, register allocation is usually done by the compiler. Hardware on the CPU handles the memory cache. Users implicitly move data from storage to memory when they execute programs and open files. But the operating system also moves data back and forth between memory and storage. At the bottom of the hierarchy, administrators move data explicitly between disk and tape.
- What gets moved? In general, block sizes are small at the top of the hierarchy and bigger at the bottom. In a memory cache, a typical block size is 128 B. Pages in memory might be 4 KiB, but when the operating system reads a file from disk, it might read 10s or 100s of blocks at a time.
- When does data get moved? In the most basic cache, data gets moved into cache when it is used for the first time. But many caches use some kind of “prefetching”, meaning that data is loaded before it is explicitly requested. We have already seen one form of prefetching: loading an entire block when only part of it is requested.
- Where in the cache does the data go? When the cache is full, we can’t bring anything in without kicking something out. Ideally, we want to keep data that will be used again soon and replace data that won’t.

The answers to these questions make up the “cache policy”. Near the top of the hierarchy, cache policies tend to be simple because they have to be fast and they are implemented in hardware. Near the bottom of the hierarchy, there is more time to make decisions, and well-designed policies can make a big difference.

Most cache policies are based on the principle that history repeats itself; if we have information about the recent past, we can use it to predict the immediate future. For example, if a block of data has been used recently, we expect it to be used again soon. This principle suggests a replacement policy called “least recently used,” or LRU, which removes from the cache a block of data that has not been used recently. For more on this topic, see http://en.Wikipedia.org/wiki/Cache_algorithms.

This page titled [3.7: Caching policy](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

- [7.7: Caching policy](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

3.8: Paging

In systems with virtual memory, the operating system can move pages back and forth between memory and storage. As I mentioned in Section 6.2, this mechanism is called “paging” or sometimes “swapping”.

Here’s how the process works:

1. Suppose Process A calls `malloc` to allocate a chunk. If there is no free space in the heap with the requested size, `malloc` calls `sbrk` to ask the operating system for more memory.
2. If there is a free page in physical memory, the operating system adds it to the page table for Process A, creating a new range of valid virtual addresses.
3. If there are no free pages, the paging system chooses a “victim page” belonging to Process B. It copies the contents of the victim page from memory to disk, then it modifies the page table for Process B to indicate that this page is “swapped out”.
4. Once the data from Process B is written, the page can be reallocated to Process A. To prevent Process A from reading Process B’s data, the page should be cleared.
5. At this point the call to `sbrk` can return, giving `malloc` additional space in the heap. Then `malloc` allocates the requested chunk and returns. Process A can resume.
6. When Process A completes, or is interrupted, the scheduler might allow Process B to resume. When Process B accesses a page that has been swapped out, the memory management unit notices that the page is “invalid” and causes an interrupt.
7. When the operating system handles the interrupt, it sees that the page is swapped out, so it transfers the page back from disk to memory.
8. Once the page is swapped in, Process B can resume.

When paging works well, it can greatly improve the utilization of physical memory, allowing more processes to run in less space. Here’s why:

- Most processes don’t use all of their allocated memory. Many parts of the text segment are never executed, or execute once and never again. Those pages can be swapped out without causing any problems.
- If a program leaks memory, it might leave allocated space behind and never access it again. By swapping those pages out, the operating system can effectively plug the leak.
- On most systems, there are processes like daemons that sit idle most of the time and only occasionally “wake up” to respond to events. While they are idle, these processes can be swapped out.
- A user might have many windows open, but only a few are active at a time. The inactive processes can be swapped out.
- Also, there might be many processes running the same program. These processes can share the same text and static segments, avoiding the need to keep multiple copies in physical memory.

If you add up the total memory allocated to all processes, it can greatly exceed the size of physical memory, and yet the system can still behave well.

Up to a point.

When a process accesses a page that’s swapped out, it has to get the data back from disk, which can take several milliseconds. The delay is often noticeable. If you leave a window idle for a long time and then switch back to it, it might start slowly, and you might hear the disk drive working while pages are swapped in.

Occasional delays like that might be acceptable, but if you have too many processes using too much space, they start to interfere with each other. When Process A runs, it evicts the pages Process B needs. Then when B runs, it evicts the pages A needs. When this happens, both processes slow to a crawl and the system can become unresponsive. This scenario is called “thrashing”.

In theory, operating systems could avoid thrashing by detecting an increase in paging and blocking or killing processes until the system is responsive again. But as far as I can tell, most systems don’t do this, or don’t do it well; it is often left to users to limit their use of physical memory or try to recover when thrashing occurs.

This page titled [3.8: Paging](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)).

- [7.8: Paging](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

CHAPTER OVERVIEW

4: Threads

When I mentioned threads in Section 2.3, I said that a thread is a kind of process. Now I will provide a more careful explanation.

When you create a process, the operating system creates a new address space, which includes the text segment, static segment, and heap; it also creates a new “thread of execution”, which includes the program counter and other hardware state, and the call stack.

The processes we have seen so far are “single-threaded”, which means that only one thread of execution runs in each address space. In this chapter, you will learn about “multi-threaded” processes that have multiple threads running in the same address space.

Within a single process, all threads share the same text segment, so they run the same code. But different threads often run different parts of the code.

And they share the same static segment, so if one thread changes a global variable, other threads see the change. They also share the heap, so threads can share dynamically-allocated chunks.

But each thread has its own stack, so threads can call functions without interfering with each other. Usually threads don’t access each other’s local variables (and sometimes they can’t).

The example code for this chapter is in the repository for this book, in a directory named `counter`. For information on downloading this code, see Section 0.2.

[4.1: Creating threads](#)

[4.2: Creating threads](#)

[4.3: Joining threads](#)

[4.4: Synchronization errors](#)

[4.5: Mutex](#)

This page titled [4: Threads](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)).

4.1: Creating threads

The most popular threading standard used with C is POSIX Threads, or Pthreads for short. The POSIX standard defines a thread model and an interface for creating and controlling threads. Most versions of UNIX provide an implementation of Pthreads.

Using Pthreads is like using most C libraries:

- You include headers files at the beginning of your program.
- You write code that calls functions defined by Pthreads.
- When you compile the program, you link it with the Pthread library.

For my examples, I include the following headers:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
```

The first two are standard; the third is for Pthreads and the fourth is for semaphores. To compile with the Pthread library in `gcc`, you can use the `-lpthread` option on the command line:

```
gcc -g -O2 -o array array.c -lpthread
```

This compiles a source file named `array.c` with debugging info and optimization, links with the Pthread library, and generates an executable named `array`.

This page titled [4.1: Creating threads](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey \(Green Tea Press\)](#).

- [9.1: Creating threads](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

4.2: Creating threads

The Pthread function that creates threads is called `pthread_create`. The following function shows how to use it:

```
pthread_t make_thread(void *(*entry)(void *), Shared *shared)
{
    int n;
    pthread_t thread;

    n = pthread_create(&thread, NULL, entry, (void *)shared);
    if (n != 0) {
        perror("pthread_create failed");
        exit(-1);
    }
    return thread;
}
```

`make_thread` is a wrapper I wrote to make `pthread_create` easier to use, and to provide error-checking.

The return type from `pthread_create` is `pthread_t`, which you can think of as an id or “handle” for the new thread.

If `pthread_create` succeeds, it returns 0 and `make_thread` returns the handle of the new thread. If an error occurs, `pthread_create` returns an error code and `make_thread` prints an error message and exits.

The parameters of `make_thread` take some explaining. Starting with the second, `Shared` is a structure I defined to contain values shared between threads. The following `typedef` statement creates the new type:

```
typedef struct {
    int counter;
} Shared;
```

In this case, the only shared variable is `counter`. `make_shared` allocates space for a `Shared` structure and initializes the contents:

```
Shared *make_shared()
{
    Shared *shared = check_malloc(sizeof (Shared));
    shared->counter = 0;
    return shared;
}
```

Now that we have a shared data structure, let’s get back to `make_thread`. The first parameter is a pointer to a function that takes a void pointer and returns a void pointer. If the syntax for declaring this type makes your eyes bleed, you are not alone. Anyway, the purpose of this parameter is to specify the function where the execution of the new thread will begin. By convention, this function is named `entry`:

```
void *entry(void *arg)
{
    Shared *shared = (Shared *) arg;
    child_code(shared);
}
```

```
pthread_exit(NULL);  
}
```

The parameter of `entry` has to be declared as a `void` pointer, but in this program we know that it is really a pointer to a `Shared` structure, so we can typecast it accordingly and then pass it along to `child_code`, which does the real work.

As a simple example, `child_code` prints the value of the shared counter and increments it.

```
void child_code(Shared *shared)  
{  
    printf("counter = %d\n", shared->counter);  
    shared->counter++;  
}
```

When `child_code` returns, `entry` invokes `pthread_exit` which can be used to pass a value to the thread that joins with this thread. In this case, the child has nothing to say, so we pass `NULL`.

Finally, here is the code that creates the child threads:

```
int i;  
pthread_t child[NUM_CHILDREN];  
  
Shared *shared = make_shared(1000000);  
  
for (i=0; i<NUM_CHILDREN; i++) {  
    child[i] = make_thread(entry, shared);  
}
```

`NUM_CHILDREN` is a compile-time constant that determines the number of child threads. `child` is an array of thread handles.

This page titled [4.2: Creating threads](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)).

- [9.2: Creating threads](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

4.3: Joining threads

When one thread wants to wait for another thread to complete, it invokes `pthread_join`. Here is my wrapper for `pthread_join`:

```
void join_thread(pthread_t thread)
{
    int ret = pthread_join(thread, NULL);
    if (ret == -1) {
        perror("pthread_join failed");
        exit(-1);
    }
}
```

The parameter is the handle of the thread you want to wait for. All the wrapper does is call `pthread_join` and check the result.

Any thread can join any other thread, but in the most common pattern the parent thread creates and joins all child threads. Continuing the example from the previous section, here's the code that waits on the children:

```
for (i=0; i<NUM_CHILDREN; i++) {
    join_thread(child[i]);
}
```

This loop waits for the children one at a time in the order they were created. There is no guarantee that the child threads complete in that order, but this loop works correctly even if they don't. If one of the children is late, the loop might have to wait, and other children might complete in the meantime. But regardless, the loop exits only when all children are done.

If you have downloaded the repository for this book (see Section 0.2), you'll find this example in `counter/counter.c`. You can compile and run it like this:

```
$ make counter
gcc -Wall counter.c -o counter -lpthread
$ ./counter
```

When I ran it with 5 children, I got the following output:

```
counter = 0
counter = 0
counter = 1
counter = 0
counter = 3
```

When you run it, you will probably get different results. And if you run it again, you might get different results each time. What's going on?

This page titled [4.3: Joining threads](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)).

- [9.3: Joining threads](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

4.4: Synchronization errors

The problem with the previous program is that the children access the shared variable, `counter`, without synchronization, so several threads can read the same value of `counter` before any threads increment it.

Here is a sequence of events that could explain the output in the previous section:

```
Child A reads 0
Child B reads 0
Child C reads 0
Child A prints 0
Child B prints 0
Child A sets counter=1
Child D reads 1
Child D prints 1
Child C prints 0
Child A sets counter=1
Child B sets counter=2
Child C sets counter=3
Child E reads 3
Child E prints 3
Child D sets counter=4
Child E sets counter=5
```

Each time you run the program, threads might be interrupted at different points, or the scheduler might choose different threads to run, so the sequence of events, and the results, will be different.

Suppose we want to impose some order. For example, we might want each thread to read a different value of `counter` and increment it, so that the value of `counter` reflects the number of threads that have executed `child_code`.

To enforce that requirement, we can use a “mutex”, which is an object that guarantees “mutual exclusion” for a block of code; that is, only one thread can execute the block at a time.

I have written a small module called `mutex.c` that provides mutex objects. I’ll show you how to use it first; then I’ll explain how it works.

Here’s a version of `child_code` that uses a mutex to synchronize threads:

```
void child_code(Shared *shared)
{
    mutex_lock(shared->mutex);
    printf("counter = %d\n", shared->counter);
    shared->counter++;
    mutex_unlock(shared->mutex);
}
```

Before any thread can access `counter`, it has to “lock” the mutex, which has the effect of barring all other threads. Suppose Thread A has locked the mutex and is in the middle of `child_code`. If Thread B arrives and executes `mutex_lock`, it blocks.

When Thread A is done, it executes `mutex_unlock`, which allows Thread B to proceed. In effect, the threads line up to execute `child_code` one at a time, so they can’t interfere with each other. When I run this code with 5 children, I get:

```
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
```

And that satisfies the requirements. In order for this solution to work, I have to add the Mutex to the Shared struct:

```
typedef struct {
    int counter;
    Mutex *mutex;
} Shared;
```

And initialize it in `make_shared`

```
Shared *make_shared(int end)
{
    Shared *shared = check_malloc(sizeof(Shared));
    shared->counter = 0;
    shared->mutex = make_mutex();    //-- this line is new
    return shared;
}
```

The code in this section is in `counter_mutex.c`. The definition of `Mutex` is in `mutex.c`, which I explain in the next section.

This page titled [4.4: Synchronization errors](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey \(Green Tea Press\)](#).

- [9.4: Synchronization errors](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

4.5: Mutex

My definition of `Mutex` is a wrapper for a type called `pthread_mutex_t`, which is defined in the POSIX threads API.

To create a POSIX mutex, you have to allocate space for a `pthread_mutex_t` type and then call `pthread_mutex_init`.

One of the problems with this API is that `pthread_mutex_t` behaves like a structure, so if you pass it as an argument, it makes a copy, which makes the mutex behave incorrectly. To avoid that, you have to pass `pthread_mutex_t` by address.

My code makes it easier to get that right. It defines a type, `Mutex`, which is just a more readable name for `pthread_mutex_t`:

```
#include <pthread.h>

typedef pthread_mutex_t Mutex;
```

Then it defines `make_mutex`, which allocates space and initializes the mutex:

```
Mutex *make_mutex()
{
    Mutex *mutex = check_malloc(sizeof(Mutex));
    int n = pthread_mutex_init(mutex, NULL);
    if (n != 0) perror_exit("make_lock failed");
    return mutex;
}
```

The return value is a pointer, which you can pass around as an argument without causing unwanted copying.

The functions to lock and unlock the mutex are simple wrappers for POSIX functions:

```
void mutex_lock(Mutex *mutex)
{
    int n = pthread_mutex_lock(mutex);
    if (n != 0) perror_exit("lock failed");
}

void mutex_unlock(Mutex *mutex)
{
    int n = pthread_mutex_unlock(mutex);
    if (n != 0) perror_exit("unlock failed");
}
```

This code is in `mutex.c` and the header file `mutex.h`.

This page titled [4.5: Mutex](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [Allen B. Downey \(Green Tea Press\)](#).

- [9.5: Mutex](#) by [Allen B. Downey](#) is licensed [CC BY-NC 4.0](#).

Index

A

accessibleeducation
[InfoPage](#)

C

cancel
1: 位元與位元組
character
1: 位元與位元組
collaborativelearning
[InfoPage](#)

D

delete
1: 位元與位元組

G

guide
1: 位元與位元組

H

highereducation
[InfoPage](#)

L

learningPath
1: 位元與位元組

M

MindTouch
1: 位元與位元組

O

OER
[InfoPage](#)
openaccess
[InfoPage](#)
openeducation
[InfoPage](#)
OperatingSystem
[Operating System](#)
OS
[Operating System](#)

P

path
1: 位元與位元組

R

remaining
1: 位元與位元組

S

settings
1: 位元與位元組

T

technologyintegration
[InfoPage](#)
textbooks
[InfoPage](#)

W

warning
1: 位元與位元組

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: 1140729專案: A Brief Introduction to Operating Systems 作業系統簡介

Webpages: 36

Applicable Restrictions: Noncommercial

All licenses found:

- [CC BY-NC 4.0](#): 66.7% (24 pages)
- [Undeclared](#): 30.6% (11 pages)
- [CC BY-NC-SA 4.0](#): 2.8% (1 page)

By Page

- 1140729專案: A Brief Introduction to Operating Systems 作業系統簡介 - *Undeclared*
 - Front Matter - *Undeclared*
 - Operating System - *Undeclared*
 - InfoPage - *Undeclared*
 - Table of Contents - *Undeclared*
 - Licensing - *Undeclared*
 - 1: 位元與位元組 - *CC BY-NC-SA 4.0*
 - 1.1: Representing integers - *CC BY-NC 4.0*
 - 1.2: Bitwise operators - *CC BY-NC 4.0*
 - 1.3: Representing floating-point numbers - *CC BY-NC 4.0*
 - 1.4: Unions and memory errors - *CC BY-NC 4.0*
 - 1.5: Representing strings - *CC BY-NC 4.0*
 - 2: Memory management - *CC BY-NC 4.0*
 - 2.1: Memory errors - *CC BY-NC 4.0*
 - 2.2: Memory leaks - *CC BY-NC 4.0*
 - 2.3: Implementation - *CC BY-NC 4.0*
 - 3: Caching - *CC BY-NC 4.0*
 - 3.1: How programs run - *CC BY-NC 4.0*
 - 3.2: Cache performance - *CC BY-NC 4.0*
 - 3.3: Locality - *CC BY-NC 4.0*
 - 3.4: Measuring cache performance - *CC BY-NC 4.0*
 - 3.5: Programming for cache performance - *CC BY-NC 4.0*
 - 3.6: The memory hierarchy - *CC BY-NC 4.0*
 - 3.7: Caching policy - *CC BY-NC 4.0*
 - 3.8: Paging - *CC BY-NC 4.0*
 - 4: Threads - *CC BY-NC 4.0*
 - 4.1: Creating threads - *CC BY-NC 4.0*
 - 4.2: Creating threads - *CC BY-NC 4.0*
 - 4.3: Joining threads - *CC BY-NC 4.0*
 - 4.4: Synchronization errors - *CC BY-NC 4.0*
 - 4.5: Mutex - *CC BY-NC 4.0*
 - Back Matter - *Undeclared*
 - Index - *Undeclared*
 - Glossary - *Undeclared*
 - Detailed Licensing - *Undeclared*
 - Detailed Licensing - *Undeclared*

Detailed Licensing

Overview

Title: 1140729專案: A Brief Introduction to Operating Systems 作業系統簡介

Webpages: 36

Applicable Restrictions: Noncommercial

All licenses found:

- [CC BY-NC 4.0](#): 66.7% (24 pages)
- [Undeclared](#): 30.6% (11 pages)
- [CC BY-NC-SA 4.0](#): 2.8% (1 page)

By Page

- 1140729專案: A Brief Introduction to Operating Systems 作業系統簡介 - [Undeclared](#)
 - [Front Matter](#) - [Undeclared](#)
 - [Operating System](#) - [Undeclared](#)
 - [InfoPage](#) - [Undeclared](#)
 - [Table of Contents](#) - [Undeclared](#)
 - [Licensing](#) - [Undeclared](#)
 - [1: 位元與位元組](#) - [CC BY-NC-SA 4.0](#)
 - [1.1: Representing integers](#) - [CC BY-NC 4.0](#)
 - [1.2: Bitwise operators](#) - [CC BY-NC 4.0](#)
 - [1.3: Representing floating-point numbers](#) - [CC BY-NC 4.0](#)
 - [1.4: Unions and memory errors](#) - [CC BY-NC 4.0](#)
 - [1.5: Representing strings](#) - [CC BY-NC 4.0](#)
 - [2: Memory management](#) - [CC BY-NC 4.0](#)
 - [2.1: Memory errors](#) - [CC BY-NC 4.0](#)
 - [2.2: Memory leaks](#) - [CC BY-NC 4.0](#)
 - [2.3: Implementation](#) - [CC BY-NC 4.0](#)
 - [3: Caching](#) - [CC BY-NC 4.0](#)
 - [3.1: How programs run](#) - [CC BY-NC 4.0](#)
 - [3.2: Cache performance](#) - [CC BY-NC 4.0](#)
 - [3.3: Locality](#) - [CC BY-NC 4.0](#)
 - [3.4: Measuring cache performance](#) - [CC BY-NC 4.0](#)
 - [3.5: Programming for cache performance](#) - [CC BY-NC 4.0](#)
 - [3.6: The memory hierarchy](#) - [CC BY-NC 4.0](#)
 - [3.7: Caching policy](#) - [CC BY-NC 4.0](#)
 - [3.8: Paging](#) - [CC BY-NC 4.0](#)
 - [4: Threads](#) - [CC BY-NC 4.0](#)
 - [4.1: Creating threads](#) - [CC BY-NC 4.0](#)
 - [4.2: Creating threads](#) - [CC BY-NC 4.0](#)
 - [4.3: Joining threads](#) - [CC BY-NC 4.0](#)
 - [4.4: Synchronization errors](#) - [CC BY-NC 4.0](#)
 - [4.5: Mutex](#) - [CC BY-NC 4.0](#)
 - [Back Matter](#) - [Undeclared](#)
 - [Index](#) - [Undeclared](#)
 - [Glossary](#) - [Undeclared](#)
 - [Detailed Licensing](#) - [Undeclared](#)
 - [Detailed Licensing](#) - [Undeclared](#)