Contents

3 — The File System **3**

4 — Processes **9**

5 — Processes in UNIX **16**

6 — Inter-Process Communication **21**

7 — Sockets **28**

8 — Network Communication **34**

9 — Pipes and Shared Memory **40**

10 — Threads **47**

11 — Threads and Concurrency **52**

12 — Concurrency: Synchronization and Atomicity **58**

13 — Semaphores **63**

14 — Synchronization Patterns **68**

15 — The Producer-Consumer Problem **74**

16 — The Readers-Writers Problem **79**

17 — Deadlock **84**

18 — Deadlock Avoidance **90**

19 — Deadlock Detection & Recovery **95**

20 — Memory **101**

21 — Dynamic Memory Allocation **107**

22 — Memory: Segmentation and Paging **112**

23 — Caching **119**

24 — Virtual Memory **124**

25 — Virtual Memory II **129**

26 — Uniprocessor Scheduling **134**

27 — Scheduling Algorithms **140**

28 — Scheduling: Idling, Priorities, Multiprocessor **146**

29 — Real-Time Scheduling, Windows & UNIX **152**

30 — Scheduling in Linux **156**

3 — The File System

File Systems

The file system is important and very useful to programs. It is more than just the way of storing data and programs persistently; it also provides organization for the files through a directory structure and maintains metadata related to files.

But what is a file? The snarky UNIX answer is, "Everything is a file!", but using the word in the definition is rather bad form. As far as the computer is concerned, any data is just 1s and 0s (bytes). The file is just a logical unit to organize these. So an area of disk is designated as belonging to a file.

Files can contain programs (e.g., `word.exe`) and/or data (e.g., `technical-report.docx`). The content of a file is defined by its creator. The creator could be a user if he or she is using notepad or something, or it could be a program, like a compiler creating an output binary file.

The UNIX approach is treating everything like a file, whether it is a device, regular file, or anything else (we'll see some examples later). This means that the functions that we use to interact with a file can be applied in several other contexts as well.

Files typically have attributes, which, although they can vary, tend generally to include the following things [SGG13]:

1. **Name**: The symbolic file name, in human-readable form.
2. **Identifier**: The unique identifier, usually a number, that identifies the file inside the file system.
3. **Type**: Information about what kind of file it is.
4. **Location**: The physical location of the file, including what device (e.g., hard drive) it is on.
5. **Size**: The current, and possibly maximum, size of the file.
6. **Protection**: Access-control information, including who owns the file, who may read, write, and execute it...
7. **Time, Date, User ID**: The owner of the file, time of creation, last access, last change... any sort of data that is useful for protection, security, usage monitoring...

Files are maintained in a directory structure. The directory structure is generally quite familiar to us as the folders on the system. Directories, really, are just like files; they are information about what files are in what locations, and they too will be stored on disk.

File Operations

It makes some sense to consider a file to be a structure; a file has some data (fields, metadata) and some operations (methods). The OS provides these operations to allow users to work with and on files. Six basic operations are required for a file system to be useful, though other things like renaming and so on are nice to have [SGG13]:

1. Creating a file.

2. Writing a file.
3. Reading a file.
4. Repositioning within a file.
5. Deleting a file.
6. Truncating a file.

Let's examine each of these briefly. As you will see, there is a certain similarity between file operations and memory operations, with which we should already be familiar. In all cases, we can look at some equivalent system calls (in the context of a trivial program) with thanks to [Sal] as the source of some examples.

We already saw in an earlier example how to open and close a file:

```
FILE* f = fopen( argv[1], "r");
if ( f == NULL ) {
    printf("Unable␣to␣open␣file!␣%s␣is␣invalid␣name?\n", argv[1] );
    return -1;
}
readfile( f );
fclose( f );
```

Repeatedly opening and closing the same file is unnecessary and inefficient. If we plan to write some data into the file, then do more work, then write more data, it is probably wasted effort to close and reopen the file in between. It is okay to keep a file open when the file isn't being actively worked on, if you expect you will use that file again in the near future.

Creating a File. Like allocating memory, creating a new file has multiple essential steps: first, find a place to put the file, allocate that space and mark that file as being allocated, and finally put the file in its appropriate directory.

Writing a File. Writing a file requires the name or identifier of the file and the data to be written to the file. Using the name or identifier, the system finds that file and can then start putting data in the file. A write operation may replace the existing contents or append (write at the end) to the existing contents. A pointer will be needed to keep track of where the next write will take place, and will be updated after each write (but we don't have to manage this ourselves if we don't want to).

Reading a File. This requires the name or identifier of the file and where in memory the next block of the file should be put. A pointer will also be required to indicate where the next read will take place (and like the write pointer, we don't have to manage this ourselves usually).

Truncating a File. If a file should be erased but its attributes maintained (e.g., all the metadata), we can truncate it: cut off all the contents. The file length is reset to zero and its data area is marked as free, but the rest of the attributes remain the same.

It turns out that creating, reading, writing, and truncating all involve the open call. The call is `FILE * fopen( const char* filename, const char* mode )`. In addition to the filename as the first parameter, the function is called with the mode as the second parameter. In the last example, the mode we provided was a string literal of `r`. The modes are:

- `r` – Open the file for reading only.
- `w` – Open the file for writing. If a file with the name exists, it's overwritten.
- `a` – Open the file, writing data at the end of the file.
- `r+` – Open for read and update (the file must exist).
- `w+` – Create a file and open it for update; if the file exists it's overwritten.
- `a+` – Open a file for update, with new output operations at the end of the file.

If we combine this with a `b`, such as `rb` then we are opening the file as a binary file. Also, as of the C 2011 standard, there is a new add-on `x` which can be used to make any write operation fail if the file exists.

Repositioning within a File. Since a file may be read or written, but usually only one at a time, the pointer for the write location may in fact be the same pointer for the memory location. If so, we might call it a current position pointer, and this operation is just repositioning it within the file. Repositioning is also sometimes called a seek operation.

In C this is done with the `fseek()` call. This adjusts the pointer for reading or writing. This should be done with caution though, because you can go to an arbitrary location, even the middle of a two (or more) byte character. And we can't seek when a file is opened for append.

Generally – seeking in the file is only necessary if you want to "skip ahead" or go back in the file; if you read n bytes of the file, that advances the read pointer for you automatically; if you read 48 bytes and then seek 48 bytes forward, at the end of both operations your pointer is 96 bytes away from where it started.

Deleting a File. Deletion works pretty much as we would expect: find the file, mark its space as free, and remove it from the directory listing. This is a "simple" deletion and it does not actually get rid of any of the data, it just makes the file system forget the existence of the file. However, it might be possible to recover the data if the space it previously occupied has not been overwritten. This is a bit like a freed pointer in C possibly still being accessible. Some systems offer a more secure deletion routine that overwrites the space the file used to occupy with zeros.

In C a file is deleted with the `remove()` function. This simple program deletes whatever file is provided as the second argument. In reality, we would more likely delete a temporary file that the program has used for some purpose, at the end of execution.

```
int main( int argc, char** argv ) {
  if (argc != 2) {
    return -1;
  }

  remove( argv[1] );

  return 0;
}
```

These six operations can be combined for most of the other things we may want to do. To copy a file, for example, create a new file, read from the old file, and write it into the new file. We may also have operations to allow a user to access or set various attributes such as the owner, security descriptors, size on disk, et cetera [SGG13].

Aside from creation and deletion, all the other operations are restricted to files that are open. When a file is opened, a program gets a reference to it, and the operating system keeps track of which files are currently open in which process. It is good behaviour for a process to close a file when it is no longer using it, but eventually when the process terminates, that will automatically close any open files (hopefully).

Some operating systems support file locks. Locks may be exclusive, or non-exclusive. When a file is locked by one process, other processes will be advised that opening failed due to someone having a lock on that file. Similarly, files in use cannot be deleted while that file is in use.

Windows, for example, uses locking and any file that is open in some program cannot be deleted. UNIX, however, does not, so UNIX-compatible programs can, if they need, lock a file, but by default this does not happen. In UNIX if a file is open in a program, another user can still delete the file and it will be removed from the directory. As long as that program remains open and retains that reference to the file, it can still operate on that file. However, once the file is no longer open in a program, its storage space will be marked as free.

To lock a file in Linux, the call for this is `flock()`. Think "File-Lock", not "flock of seagulls". It takes two parameters, the file descriptor and the type of lock we wish to have. But we have opened a file with `fopen()` and it returns a `FILE` pointer. To convert that to a file descriptor, there is a function `fileno()`.

```
FILE* f = fopen("myfile.txt", "r");
int file_desc = fileno( f );
int result = flock( file_desc, LOCK_EX );
```

This example locks the file exclusively. A shared lock would be `LOCK_SH`, and to unlock the parameter is `LOCK_UN`.

Reading and Writing

Writing to a file is easy enough because it works like `printf`. In fact, the function call for it is `fprintf` and the only real difference between that and `printf` is that the first argument to this is the file pointer where you'd like the data to be written to:

```
void write_points_to_file( point* p, FILE f ) {
  while( p != NULL ) {
    fprintf(f, "(%d, %d, %d)\n", p->x, p->y, p->z);
    p = p->next;
  }
}
```

Reading from a file involves the use of `fscanf` which is a mirror image of `fprintf`. The format specifiers are the same. Let's look at an example from [Sal]:

```
int main( int argc, char** argv ){
  FILE *fp;
  int i, isquared;

  fp = fopen("results.dat", "r");
  if (fp == NULL) {
    return -1;
  }

  while (fscanf(fp, "%d,%d\n", &i, &isquared) == 2) {
    printf("i: %d,  isquared: %d\n", i, isquared);
  }

  fclose(fp);
  return 0;
}
```

The return value of this function call is the number of elements successfully read, which in this case is supposed to be two. But it's worth noting that there's no space in the read from the file, because the `%d` skips leading whitespaces (occasionally leading to hard-to-find bugs).

If reading user input, there is of course, regular `scanf`. We can, of course, read a file with `getline` as in the earlier examples.

File Types

Files we are familiar with often have extensions separated from the file name by a period, like `fork.txt`. The `txt` extension tells us some information about the file, i.e. that it is a text file. These things are mostly hints to the OS or user about what sort of file it is. In most operating systems, any program can open arbitrary files... that it has a `.docx` extension is only a suggestion that it should be opened by a word processing program, but nothing stops people from opening it in any other program. OSes typically allow setting a default program for the extension: e.g., always open `.docx` files with LibreOffice.

Directories

A directory is really just a symbol table that translates file names (user-readable representations) to their directory entries. A directory should support several common operations [SGG13]:

1. **Search**. We want to be able to find a file, and searching is typically not just on the file name but may include the contents of files as well, if their content is human-readable data.
2. **Add a File**. Add a file to the directory.
3. **Remove a File**. Remove a file from the directory.

4. **List a Directory**. List the files of the directory and the contents of the directory entry.
5. **Rename a File**. Change the user-friendly file name, possibly changing the file's position in the directory if it is sorted by name.
6. **Navigate the File System**. It should be possible to open subdirectories, go to parent directories, and so on.

There are some simple file systems where there are no such things as subdirectories, but they don't really require any examination. Textbooks may also bring up a structure where each user has his or her own directory but cannot have subdirectories either. Also rather uninteresting. The kind of directory we are most familiar with is tree-structured: there is a root directory, and every file in the system has a unique name when the name and path to it (from the root) are combined.

In UNIX the root directory is just called `/` (forward slash) and from there we can navigate to any file. If we would like to run the `ls` command, we will find it in the `bin` directory as `/bin/ls`. This is an example of an absolute path. Most of the time we do not have to use the absolute path (the full file name); a relative path (the path from the current directory) will suffice. As an example, if you want to compile something with a command like `gcc code/example.c`, the file `example.c` is in a subdirectory of the current directory called `code` and the system will work out that we need to start from the current directory (e.g., `/home/jz/ece252/`) and prepend that to the given file name, to produce the absolute path of `/home/jz/ece252/code/example.c`.

OS designers have to make a choice about deletion of directories, if a directory is not empty. If it is empty, just removing the directory is enough. If it contains some files, either the system can refuse to delete the directory until it is empty, or automatically delete the files and subdirectories in their entirety. Also, what does it mean to delete a file or folder? In modern operating systems, the delete command sometimes does not necessarily actually delete the file or folder, but instead moves it to some deleted files directory (recycle bin, trash can, whatever you want to call it). If it is deleted from there then it is really gone, but while it is in that deleted file directory it can be restored.

File systems may also support the sharing of files: there is one copy of the file but it has more than one name. In UNIX this concept is called a *link* and this is effectively a pointer to another file. Links are either "hardlinks" or "symlinks".

Symlinks, or symbolic links, are just references by file name. So if a symbolic link is created to a file like `/Users/jz/file.txt`, the symbolic link will just be a "shortcut" to that file. If the file is later deleted, the symbolic link is left pointing to nothing. A future attempt to use this pointer will result in an error, so it is something to check on. It would be expensive, though possible, to search through the file system to find all links and remove them.

Creating a hardlink means creating a second pointer to the underlying file in the file system. If a hardlink exists and the user deletes that file, the file still remains on disk until the last hardlink is removed. We will see later on, when we examine the file system implementation, how this actually works, but the short answer is: reference counting. The file structure maintains a count of how many hardlinks reference a file, and it is only really deleted if the count falls to zero.

File Permissions

To protect users from one another and to maintain the confidentiality and integrity of data, files usually have some permissions associated with them, which may control access to the following operations [SGG13]:

1. Read
2. Write
3. Execute
4. Append (write at the end of the file)
5. Delete
6. List (view the attributes of the file)

UNIX-Style Permissions. UNIX-Style permissions are commonly used still today in a lot of UNIX and UNIX-like systems. Each file has an owner and a group, and a set of permissions that can be assigned for the owner, the group, and for everyone. There are three basic permissions: read, write, and execute (run as a program). The permissions are represented using 10 bits, where a 1 indicates true and a 0 indicates false. The first bit is the directory bit and indicates if the file being examined is actually a directory. The next three bits are the read, write, and execute bits for the owner, followed by the read, write, and execute bits for the group, and finally the read, write, and execute bits for everyone.

Effective permissions are determined by the user: the owner of the file gets the owner permissions even if different permissions are assigned to the group or everyone. Precedence goes from left to right: owner takes precedence over the group; group takes precedence over the permissions for everyone.

The permissions can be shown to the screen in a human-readable format which is ten characters long. The order is always the same, and so a dash (`-`) appears if a bit is zero (permission does not exist). The character `d` is used to indicate a directory, `r` to indicate read access, `w` to indicate write access, and `x` to indicate execute access.

Example: permissions of `-rwxr-----` indicate that a file is not a directory; the owner can read, write, and execute; other members of the group can read it only, and everyone else has no access to the file (cannot read, write, or execute).

Permissions can also be written in octal (base 8) where r = 4, w = 2, and x = 1. To get the octal representation, start with 0, and then add the value of the permissions that are present, using zero where permissions are absent. You might then have a permission that reads `750` - meaning that the owner has read, write and execute access (0 + 4 + 2 + 1 = 7); group members have read and execute access (0 + 4 + 0 + 1 = 5); everyone else has no access to it at all.

There are some more details like what the permissions mean on directories, and some advanced topics like `setuid`, `setgid`, and "sticky bit", but we will not cover them in this course.

The obvious shortcoming of this approach is that it is very coarse-grained: there are only three groups for whom we can specify permissions. Within Linux and other similar systems there is a trend now towards using SELinux (Security Enhanced Linux) which is an Access Control List system. For the moment we'll leave off the discussion of access control lists and just proceed forward.

4 — Processes

Processes

Early computers, as well as many modern embedded systems, did exactly one thing, or at least, exactly one thing at a time. At that time, the program had access to all the resources available in the system. Now, we expect that the OS supports multiple programs running concurrently. For that to work reliably, the operating system needs a way to manage the complexity and this has resulted in the notion of a *process*. We've already worked with processes, but most likely we didn't know it at the time.

A process is a program in execution. It is composed of three things:

1. The instructions and data of the program (the compiled executable).
2. The current state of the program.
3. Any resources that are needed to execute the program.

Having two instances of the same program running counts as two separate processes. Thus, you may have two windows open for Microsoft Word, and even though they are the same program, they are separate processes. Similarly, two users who both use Firefox at the same time on a terminal server are interacting with two different processes.

The Process Control Block

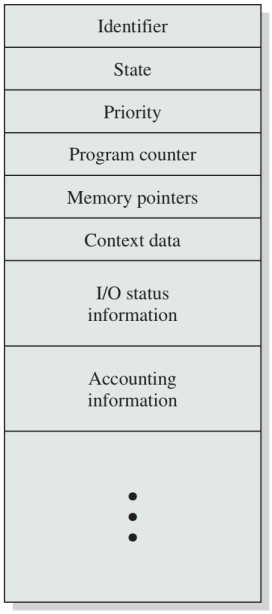
We will take a slight detour to the behind-the-scenes view of how the operating system manages a process, so that we can have a mental model of what may happen when a program is executing. The operating system's data structure for managing processes is the *Process Control Block* (PCB). This is a data structure containing what the OS needs to know about the program. It is created and updated by the OS for each running process and can be thrown away when the program has finished executing and cleaned everything up. The blocks are held in memory and maintained in some container (e.g., a list) by the kernel.

The process control block will (usually) have [Sta14]:

- **Identifier.** A unique ID associated with the process; usually a simple integer that increments when a new process is created and reset when the system is rebooted.
- **State.** The current state of the process.
- **Priority.** How important this process is (compared to the others).
- **Program Counter.** A place to store the address of the next instruction to be executed (*when needed).
- **Register Data.** A place to store the current values of the registers (*when needed); also called context data.
- **Memory Pointers.** Pointers to the code as well as data associated with this process, and any memory that the OS has allocated by request.

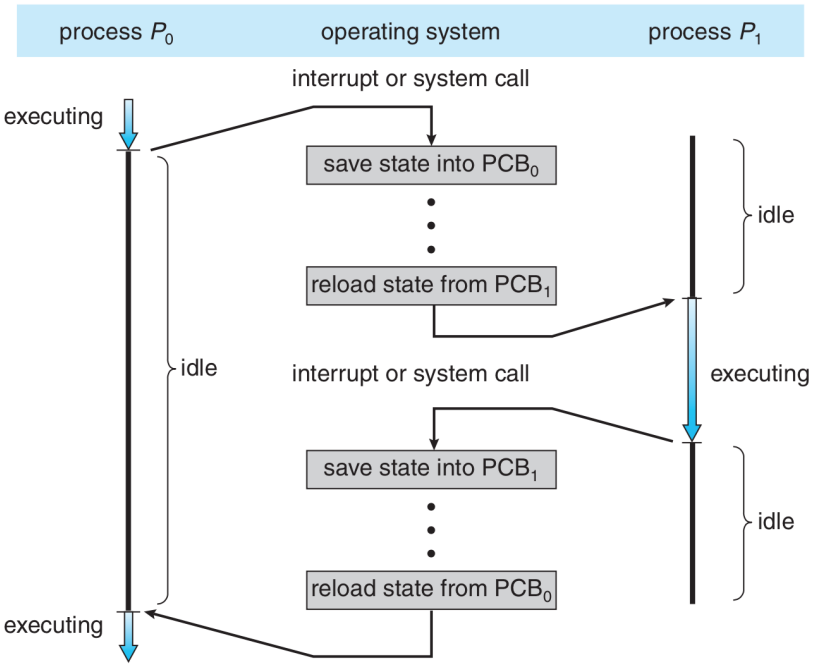
- **I/O Status Information.** Any outstanding requests, files, or I/O devices currently assigned to this process.
- **Accounting Information.** Some data about this process's use of resources. This is optional (but common).

To represent this visually:



A simplified Process Control Block [Sta14].

Almost all of the above will be kept up to date constantly as the process executes. Two of the items, notably the program counter and the register data are asterisked with the words "when needed". When the program is running, these values do not need to be updated. However, when a system call (trap) or process switch occurs, and the execution of that process is suspended, the OS will save the state of the process into the PCB. This includes the Program Counter variable (so the program can resume from exactly where it left off) and the Register variables (so the state of the CPU goes back to how it was). The diagram below shows the sequence as the OS switches between the execution of process P_0 and process P_1.



A process switch from P_0 to P_1 and back again [SGG13].

The Circle of Life

Unlike energy, processes may be created and destroyed. Upon creation, the OS will create a new PCB for the process and initialize the data in that block. This means setting the variables to their initial values: setting the initial program state, setting the instruction pointer to the first instruction in `main`, and so on. The PCB will then be added to the set of PCBs the OS maintains. After the program is terminated and cleaned up, the OS may collect some data (like a summary of accounting information) and then it can remove the PCB from its list of active processes and carry on.

Process Creation

There are, generally speaking, three main events that may lead to the creation of a process [Tan08]:

1. System boot up.
2. User request to start a new process.
3. One process spawns another.

When the computer boots up, the OS is started and begins creating processes. An embedded system might have all the processes it will ever run created by this initialization process, but general-purpose operating systems will allow at least one of the other routes (if not both of them).

At boot time the OS starts up various processes, some of which will be in the foreground (visible to the user) and some in the background. A user-visible process might be the log in screen; a background process might be the server that shares media on the local network. The UNIX term for a background process is *Daemon*. You have already worked with one of these if you have ever used the `ssh` (Secure Shell) command to log into a Linux system; when you attempt to connect it is the `sshd` (Secure Shell Daemon) that responds to your connection attempt.

Users are well known for starting up processes whenever they feel like it, much to the chagrin of system designers everywhere. Every time you double-click an icon or enter a command line command (like `ssh` above) that will result in the creation of a process.

An already-executing process may spawn another. If you receive an e-mail with a link in it and click on that link[1], the e-mail program will start up the web browser (another process) to open the web page. Or a program may break its work up into different logical parts to be parcelled out to subprograms that run as their own process (to promote parallelism or fault tolerance). When an already-executing program spawns another process, we say the spawning process is the *parent* and the one spawned is the *child*. Later on, we will return to the subject of relations between processes in UNIX.

Process Destruction

Eventually, most processes die. This is sad, but it can happen in one of four ways [Tan08]:

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal Error (involuntary)
4. Killed by another process (involuntary)

Most of the time, the process finishes because they are finished or the user asks them to. If the command is to compile some piece of code, when the compiler process is finished, it terminates normally. When you are finished writing a document in a text editor, you may click the close button on the window and this will terminate the program normally.

[1] Security advice: don't click on links you receive by e-mail.

Sometimes there is voluntary exit, but with an error. If the user attempts to run a program that requires write access to the temporary directory, and it checks for the permission on startup and does not find it, it may exit voluntarily with an error code. Similarly, the compiler will exit with an error if you ask it to compile a non-existent file [Tan08]. In either case, the program has chosen to terminate (not continue) because of the error and it is a voluntary termination.

The third reason for termination is a fatal error occurring in the program, like a stack overflow error or division by zero. The OS will detect this error and send it to the program. Very often, this results in the involuntary termination of the offending program. A process may tell the OS it wishes to handle some kinds of errors (like in Java/C# with the `try-catch-finally` syntax) in which case the OS will send the error to the program which can hopefully deal with it. If so, the process may continue, otherwise, the unhandled exception will result in the involuntary termination.

The last reason for termination is that one process might be killed by another (yes, processes can murder one another. Is no-one safe?!). Typically this is a user request: a program is stuck or consuming too much CPU and the user opens task manager in Windows or uses the `ps` command (in UNIX) to find the offender and then terminates it with the "End Process" button (in Windows) or the `kill` command (in UNIX). However, programs can, without user intervention, theoretically kill other processes, such as a parent process killing a child it believes to be stuck (or timed out).

Obviously, there are restrictions on killing process: a user or process must have the rights to execute the victim. Typically a user may only kill a process he or she has created, unless that user is a system administrator. While killing processes may be fun, it is something that should be reserved for when it is needed.

Sometimes when a process is killed, all the processes it has spawned are killed as well (this is really cruel). Neither UNIX nor Windows works this way, however: a parent can outlive the death of its child and vice-versa.

Process Family Tree

In UNIX, but not in Windows, the relationship between the parent process and child process(es), if any, is maintained, forming a hierarchy. A process, unlike most plants and animals, reproduces asexually: a process has one parent, but may have zero or more children. A process and all its descendants form a *process group* and certain operations like sending a signal (e.g., the terminate signal `Ctrl-C`) can be sent to the whole group, letting each process decide what to do with it. [Tan08].

In UNIX the first process created is called `init` and it is the parent of all processes (eventually), much like the `Object` class in Java is the superclass of all classes in the system. Thus in UNIX we may represent all processes as a tree structure, where each node is a process, each node may have zero or more children, and moving up the hierarchy will eventually take us to `init`.

In Windows, a process that spawns another process gets a reference to its child, allowing it to exercise some measure of control over the child. However, this reference may be given to another process (so, the concept of adoption exists) meaning there is no real hierarchy. A process in UNIX cannot disinherit a child [Tan08].

When a process terminates, voluntarily or otherwise, it does so with a return code, just as a function often returns a value. If the command is issued on the command line (e.g., `cat /var/log/syslog`) or from double clicking an icon, the return value is generally ignored (or at least, not presented to the user). In UNIX, when a parent process spawns a child, it can get the code that process returns. Usually, a return value of zero indicates success and other values indicate an error of some sort. Normally there is some sort of understanding between the parent and child processes about what a particular code means.

When a child process finishes execution, until such time as the parent comes by to collect the return value, the child continues in a state of "undeath" we call a *zombie*. This does not mean that the process then shuffles around the system attempting to eat the brains of other processes; it just means that the process is dead but not gone. The program has finished executing, there is still an entry in the PCB list, and the process holds on to its allocated resources until such time as the return value is collected. Only after the return value is collected can it be cleaned up. Usually, a child process's result is eagerly awaited by its parent and the `wait` call collects the value right away, allowing the child to be cleaned up (or, more grimly, "reaped"). If there is some delay for some reason, the process is considered a zombie until that value is collected.

If a child process's parent should die before the child does, the process is called an *orphan*. In UNIX any orphan process is automatically adopted by the `init` process, making sure all processes have a good home. By default, `init` will just `wait` on all its child processes (and do nothing with the return values), ensuring that when they are finished, they do not become zombies. Sometimes a program is intentionally orphaned: it is spawned to run in the background (e.g., when starting up a service or daemon on the system). This would be cruel, except that processes, as far as anyone knows, do not have feelings.

The Five-State Model

As you might imagine, at any given time, a process is running or not running. The first two states of the model are therefore "Running" and "Ready".

A program that requests a resource like I/O or memory may not get it right away. This is not to say the program will never get it, just that it does not have it right now. Sometimes the program needs user input, and as far as the computer is concerned, the user moves at glacial speed. In any case, the program wants to continue but cannot until it gets what it is waiting for. If the scheduler picks a process that is waiting for user input, nothing will be happening while the program is waiting for input, so the CPU's time would be wasted. Thus, we should be able to mark a process as "not ready to proceed", which gives us our "Blocked" state.

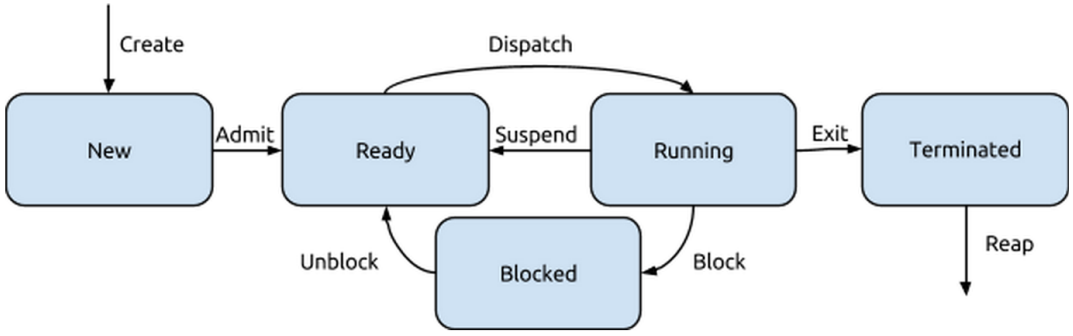
Earlier we discussed that a UNIX process may be finished but a zombie, because its parent has not (yet) come by to collect its return value. The process is not ready to run (it has finished running) and it is not waiting for a resource, so it does not qualify for Ready or Blocked. Thus, we need a state to represent that it is finished but not yet cleaned up: Terminated.

The fifth and final state will be the "New" state: a process that has just been defined. Suppose a user wants to run a new process. The OS will first perform the necessary administrative tasks: define an identifier for the process, instantiate the PCB object, and put the process in the New state. The OS has created the process but has not committed itself to execution thereof. This may be because the system limits the number of concurrent processes for performance reasons. When the process is in the New state is is typically not in memory, but on disk instead [Sta14].

Thus, with the two new states added, the five states of a process in the system are:

1. **Running:** Actively executing right now.
2. **Ready:** Not running, but ready to execute if selected by the scheduler.
3. **Blocked:** Not running, and not able to run until some event happens.
4. **New:** Just created but not yet added to the list of processes ready to run.
5. **Terminated:** Finished executing, but not yet cleaned up (reaped).

With five states, we will have significantly more transitions between the states. The diagram below shows the five-state model:



State diagram for the five-state model.

There are now eight transitions, most of which are similar to what we have seen before:

- **Create:** The process is created and enters the New state.
- **Admit:** A process in the New state is added to the list of processes ready to start, in the Ready state.
- **Dispatch:** A process that is not currently running begins executing and moves to the Running state.
- **Suspend:** A running program pauses execution, but can still run if allowed, and moves to the Ready state.
- **Exit:** A running program finishes and moves to the Terminated state; its return value is available.
- **Block:** A running program requests a resource, does not get it right away, and cannot proceed.
- **Unblock:** A program, currently blocked, receives the resource it was waiting for; it moves to the Ready state.
- **Reap:** A terminated program's return value is collected by a `wait` and its resources can be released.

There are two additional "Exit" transitions that may happen but are not shown. In theory, a process that is in the Ready or Blocked state might transition directly to the Terminated state. This can happen if a process is killed, by the user or by its parent (recall that parent processes can generally kill their children at any time, something the law thankfully does not permit). It may also happen that the system has a policy of killing all the children of a parent process when the parent process dies.

Swapping Processes to Disk

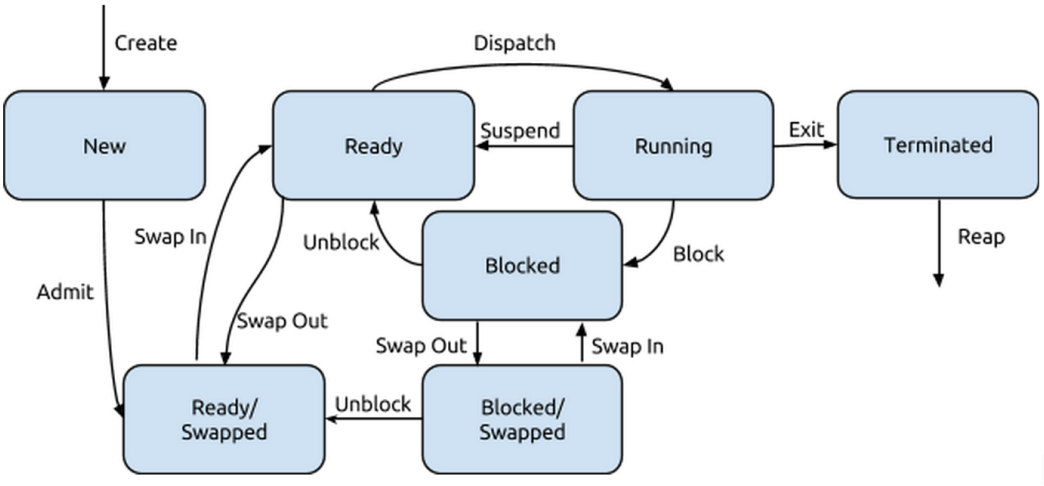
We can expand on the five state model with something else: the idea that a process might be blocked due to its data not being in main memory. We expect that our system has multiple processes in it, but it is quite possible that the user wants to have more processes running than there is space to accommodate. The problem is not the PCBs, which are relatively small (a few thousand bytes), but the stack and heap space allocated to the running program can be very large (on the order of gigabytes).

With no other place to put this memory, the operating system will have to put parts of some processes on disk, and this is what we know as swapping. Thus, when the demands for memory exceed the available memory, some of the processes will be (at least partially) moved to disk storage to make room for other processes. This is a notably expensive operation: writing data to disk, from the perspective of the CPU, takes about seven eternities. Then, if the process is going to need that memory again, the OS will need to load it back in to memory, which will take just as much time as it took to flush it out. So this is something to be done only when necessary.

Because the OS does not want to spend any more time swapping the process in and out of memory than is necessary, (and we need to know if a particular process is in memory or on disk) we need a new state: swapped. Ideally, it will only swap part of a process to disk if that process is blocked. It cannot run anyway, so if it has to choose some data to put on disk, a blocked process's memory is better choice than a ready one's. A process that cannot proceed because of a lack of memory enters that sixth state, swapped, which means it is blocked and not in main memory.

There are two scenarios that may have occurred to you that tell us the swapped state on its own is not sufficient. The first is: what if all processes are ready but there is not enough memory space? Or, in other words, what if we need to swap out part of a process that is ready? The second is: what if the event the blocked process was awaiting has taken place (e.g., the user presses a key) and the process could proceed? How can we tell which processes currently swapped out have had their desired events occur and which have not? The OS would not guess in the second scenario, because swapping is time consuming.

The solution to both problem scenarios is to split the swapped state in two: Ready/Swapped (ready to run, and currently not in memory) and Blocked/Swapped (not ready to run, and currently not in memory). That gives us, finally, the seven-state model, a minor variation of the five-state model:



State diagram for the seven-state model.

The Admit transition is modified to show that by default the new process does not start in main memory. Two new transitions, Swap In and Swap Out, are added to show a process being loaded into main memory and written out to disk respectively. Finally, there is a second Unblock transition, where a Blocked/Swapped process gets whatever it was waiting for and moves to the Ready/Swapped state, because it can now run (but is still on disk).

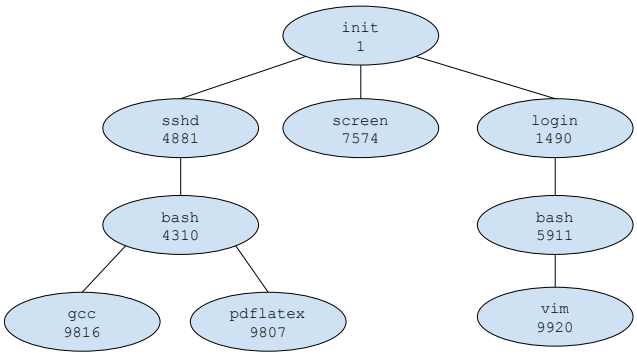
As in the five-state model, there are additional “Exit” transitions that may happen but are not shown. If a process is killed, for example, regardless of whether it is in memory or on disk, it will move to the Terminated state.

5 — Processes in UNIX

The Process in UNIX

Earlier on, we mentioned that in UNIX, a process may create other processes. The creating process is the parent and the newly-created process(es) is (are) its child(ren). Every process has a parent, stretching back to the `init` process (or `launchd`) at the root of the tree.

Each process has a unique identifier in its process control block, and in UNIX we call this the `pid` (process ID). For the most part, users will not need to know or think about the ID of a process except when trying to terminate one that's gotten stuck (`kill -9 24601`). The `init` process always gets a pid of 1. I don't recommend trying to kill `init`. In most cases, `init` will just laugh off your attempt ("'tis but a scratch!") but you might end up rebooting the system or causing a crash.



A tree of processes in a Linux system.

In a UNIX system, we can obtain a list of processes at any time with the `ps` command. The diagram illustrates that each user, when logging in, spawns a `login` process (and an administrator can log people out by killing their `login` process... not that this is necessarily a good way to do it). The user's shell (these days, almost always `bash`, the Bourne Again Shell) is then spawned from `login`. That shell provides the command line interface where the user can enter a command.

When you issue a command, like `ls` or `top` (table of processes), the new process is created and the shell will `wait` on that process to finish (in the case of `ls`) or for the user to tell it to exit (`top`); when it does, control goes back to the shell and you get presented with the prompt again (e.g., `jz@Loki:~/$`). This would, on its face, seem kind of limiting – do I have to log in to the system in a second terminal window to run two things at a time? The answer is no, and there are two ways to get around it.

The first thing we can do is tell the shell we want the task to run in the background. To do that, add to the command the & symbol, like so:

```
gcc fork.c &
```

This will return control almost immediately to the shell (as it will not be waiting for the `gcc` command to finish). You may see some output like `[1] 34429` which is the shell saying the child has been created and it has process ID 34429. When the process is finished, there is another update, looking something like:

```
[1]+ Done gcc fork.c
```

Notably, any console output that the `gcc` command would generate will still appear on the console where the background task was created. Maybe you want that but maybe you want to put the output in a log file, with a command like `cat fork.c > logfile.txt &`. (Telling `gcc` to be silent is a somewhat more complex operation.)

A common example of a command I use involving the &:
`sudo service xyz start &`
This will (with super user permissions - that's the purpose of `sudo`) start up the service `xyz` but return control to the console so I don't have to wait for the `xyz` service to be started to enter my next command. This is good, because the next thing I'd like to do is `tail -f /var/log/xyz/console.log` which will allow me to watch the console log of the `xyz` service as it starts up to see if there are any errors.

The other alternative to get something to run in the background is with the `screen` command. While having something run in the background is nice, it does not work for interactive processes. Suppose you are working on some code in `vi` and you would like to pause that for a minute and write an e-mail (with `pine` or whatever the cool kids use for command line e-mail these days). One approach is to save and exit `vi` and open up `pine`. The other would be to start up each of these in `screen` and switch between them.

Thus instead of just opening `vi fork.c` I can issue the command `screen vi fork.c` and this spawns `screen` and takes me right to editing the file. The key difference is that I can "detach" from this screen and go back to the command line that spawned it. And if I log out, `screen` keeps running with the `vi` inside it. If I have multiple screens running, I can just "reattach" to the one I want to use next. To get a full understanding of `screen`, try the command `man screen` and the user manual will appear to give you some information and instructions about how to use this. Or you can use Google.

Show Me The Code!

The workflow in UNIX is as follows. First, the parent spawns the child process with the `fork` system call. If it is interested in waiting for the child process to finish, it will use the system call `wait`, in which case the parent will be awaiting the completion of the child process. When the child process is finished, it returns a value with the `exit` system call. The parent process will then get this as the return value of the `wait` call and may proceed.

What does `fork` do? It creates a new process; it makes a copy of itself. The parent and child continue execution after the `fork` statement. If `fork` returns a negative number, the `fork` system call failed. If it returns 0, the process that got the 0 back is the child. If it returns a positive value, that is the process ID of the child.

After the `fork`, one of the processes may use the `exec` system call, or one of its variants, to replace its memory space with a new program. There's no rule that says this must happen; a child can continue to be a clone of its parent if it wishes. The `exec` invocation loads the binary file into memory and starts execution [SGG13]. At this point, the programs can go their separate ways, or the parent might want to wait for the child to finish. The parent is then blocked, waiting for the child process to execute.

Let's put this all together in an actual C-code example adapted from [SGG13]:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main( int argc, char** argv ) {
  pid_t pid;
  int childStatus;

  /* fork a child process */
  pid = fork();

  if (pid < 0) {
    /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;

 } else if (pid == 0) {
    /* child process */
    execlp("/bin/ls","ls",NULL);
```

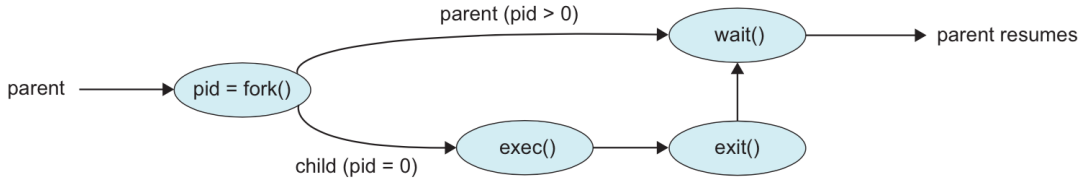
```
  } else {
    /* parent process */
    /* parent will wait for the child to complete */
    wait(&childStatus);
    printf("Child Complete with status: %i \n", childStatus);

  }

  return 0;
}
```

When executed, this code starts up and attempts to spawn a child process. Let us assume that the `fork` command succeeds and we do not enter the error-occurred block. After the fork there are now two processes at the statement `if ( pid < 0 )` . The child process calls `execlp`, replacing itself with the `ls` (list directory contents) command. The parent process will go to the `wait` statement and wait for the child process to complete. The child process runs `ls`, listing the contents of the directory. Then it finishes. The parent process, finally, prints "Child Complete" to the console.

Thus, the output is:

```
jz@Freyja:~/fork$ ./fork
fork   fork.c
Child Complete with status: 0
jz@Freyja:~/fork$
```

Or, to represent this visually:



Process creation with the `fork` system call [SGG13].

What about termination? On the assumption that the process is terminating normally and not being killed, the system call for that is `exit`. If the program itself has no explicit call to `exit`, the `return` statement at the end of `main` will have the same effect.

Use of Fork Design Problem. It's not necessary for a child to replace itself with another one. Let us do an example of how we might write a program where both parts are in the same source file.

There is a task that can be split into parts 'A' and 'B'. You may assume there are implementations for the functions `int execute_A()` and `int execute_B()` that work correctly (and are not shown). If either of the execution functions returns a non-zero value, this indicates an error.

Use `fork()` to create a child process. The child process should call function `execute_B()` and return the result to the parent. The parent process should call `execute_A()` and collect its result. The parent should then collect the result of the child using `wait()` and then produce the console output described in the next paragraph. If no errors occurred, `main` should return 0; otherwise it should return -1.

If an error occurs, it should be reported to the console including the error number (e.g., "Error 7 Occurred."). If more than one error occurs, report both errors. If both functions return zero, it means all is well and the program should print "Completed." to the console.

Use of Fork Design Solution The solution:

```
int main( int argc, char** argv ) {

  pid_t pid;
  int child_result;
  int parent_result;

  pid = fork();

  if ( pid < 0 ) { /* Fork Failed */
    return -1;
  } else if ( pid == 0 ) { /* Child */
    return execute_B();
  } else { /* Parent */
    parent_result = execute_A();
    wait( &child_result );
  }

  if ( child_result == 0 && parent_result == 0 ) {
    printf( "Completed.\n" );
    return 0;
  }

  if ( child_result != 0 ) {
    printf( "Error␣%d␣Occurred.\n", child_result);
  }
  if ( parent_result != 0 ) {
    printf( "Error␣%d␣Occurred.\n", parent_result);
  }

  return -1;
}
```

The Fork Bomb

We will take a slight digression from the general topic of processes to discuss a denial-of-service attack against UNIX systems that is called the "Fork Bomb". The idea behind the attack is to call `fork` repeatedly until the number of processes spawned is too high for the system to manage and it either crashes or is so slow that no useful work can get done. As you can imagine, this is caused by repeatedly calling `fork`. Each time a program does so, there are now two processes running, each of which calls `fork`. There are then 2^n processes after each of n invocations and this exponential growth will soon crash up against the limits of the system.

A system configured to defend against this may impose limits on (1) the total number of processes a user may create; and (2) the rate at which a user may spawn a new process.

Note: do not attempt this on anything other than your personal computer at home. It is a denial of service attack and would certainly count as misuse of resources on any system. Accordingly, trying to do it will very likely result in a ban. Getting banned from using university computer resources is not conducive to completing your degree.

Signals

UNIX systems use signals to indicate events (e.g., the `Ctrl-C` on the console). Signals also are things like exceptions (division by zero, segmentation fault), etc. A signal may be *synchronous* if the signal occurs as a result of the program execution (e.g., dividing by zero); it is *asynchronous* if it comes from outside the process (e.g., the user pressing `Ctrl-C` or one process or thread sending a signal to another). Signals are, in the end, interrupts with a certain integer ID.

By default, the kernel will handle any signal that is sent to a process with the default handler. The behaviour of the default handler may be to ignore the signal, but some signals (segmentation fault) will result in termination of the process.

Here are some of the many signals described in the POSIX.1-1990 standard:

Signal	Comment	Value	Default Action
SIGHUP	Hangup detected	1	Terminate process
SIGINT	Keyboard interrupt (Ctrl-C)	2	Terminate process
SIGQUIT	Quit from keyboard	3	Terminate process, dump debug info
SIGILL	Illegal instruction	4	Terminate process, dump debug info
SIGKILL	Kill signal	9	Terminate process
SIGSEGV	Segmentation fault (invalid memory reference)	11	Terminate process, dump debug info
SIGTERM	Termination signal	15	Terminate process
SIGCHLD	Child stopped or terminated	20,17,18	Ignore
SIGCONT	Continue if stopped	19,18,25	Continue the process if stopped
SIGSTOP	Stop process	18,20,24	Stop process

Alternatively, a process may inform the operating system it is prepared to handle the signal itself (such as doing some cleanup when the Ctrl-C is received instead of just dying). In any event, a signal needs to be handled, even if the handling is to ignore it. Note that the signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

On the command line, the command to send a signal is alsokill followed by a process ID. Normally a command like kill 24601 will send SIGHUP to a process, which will, by default, kill the process. The process has an opportunity to clean things up if it wants to. If the process is still stuck, you can "force" kill the process by sending SIGKILL with the command kill -9 24601. The -9 parameter says to send signal 9 (SIGKILL) rather than the default 1 (SIGHUP). Some users are eager to jump to kill -9 whenever a process is stuck, but it's usually worthwhile to attempt a less severe killing (SIGHUP or Ctrl-C) so that the process can at least try to clean up.

A signal is, interestingly, a form of inter-process communication: you are sending a signal from one process to another, even if it's your terminal doing the sending. When we have more than one process, we might want them to communicate, so we should look into how to make that happen.

6 — Inter-Process Communication

Inter-Process Communication (IPC)

When two or more processes would like to co-ordinate or exchange data the mechanism for doing so is called *inter-process communication*, usually abbreviated as IPC. If a process shares data with another process in the system, the operating system will provide some facilities to make this possible.

The motivations for inter-process communication are fairly obvious and we do not need to waste time with long descriptions of scenarios where IPC is desirable. Every time you send an e-mail, for example, there's IPC going on. In the context of your own program, you could consider breaking a large task into smaller subtasks, allowing multiple users to edit the same data, and system modularity.

Before proceeding, we need to define some idea about what communication is: transfer of data from one process to another. The data being transferred is typically referred to as the *message*. The process sending that message is the *sender* and the process receiving it will be the *receiver*. This terminology is brutally, painfully obvious, but completeness is key.

The processes involved must have some agreement on what data a message should contain, and the way the data is formatted. Though there may be defined standards, e.g., transferring data formatted in XML, for the message content, the processes themselves would have to be aware of the fact that the message is in XML format. How this agreement is reached tends to fall outside the purview of the operating system; perhaps the authors agree in a meeting or the sender publishes its format online and the author of the receiver program writes her code to accommodate that.

Sending and receiving of messages may be either synchronous or asynchronous. If sending is synchronous, the sender sends the message and then is blocked from proceeding until the message is received. If sending is asynchronous, the sender can post the message and then carry on execution. If receiving is synchronous, the receiver is blocked until it receives the message. If receiving is asynchronous, the receiver is notified there is no message available and continues execution. Thus there are four combinations to consider, three of which are common [HZMG15]:

1. **Synchronous send, synchronous receive:** the sender is blocked until the receiver collects the message; the receiver waits for a message and is blocked until it arrives.
2. **Synchronous send, asynchronous receive:** the sender is blocked when the message is sent, but the receiver will continue whether or not a message is available. This is a very uncommon configuration.
3. **Asynchronous send, synchronous receive:** the sender continues execution when the message is sent, but the receiver will wait until the message is received before it can continue. This is the most common configuration; usually the receiver needs a message to continue.
4. **Asynchronous send, asynchronous receive:** the sender can continue as soon as it sends the message, and the receiver will check for a message but will continue whether or not the message is available.

It is also common in the case of asynchronous receive for the receiver to send back another message confirming receipt of the message. When that happens, just reverse the labels: the receiver of the initial message is the sender of the acknowledgement.

A general paradigm for understanding IPC is known as the *producer-consumer* problem. The *producer* creates some information which is later used by the *consumer*. For example, the database may produce the data (some records from the database) to be consumed by the shell to be displayed to the user. This is a general problem and applicable to client-server situations (e.g., web servers sending out web pages to web browsers).

There are three approaches we will consider on how we can accomplish IPC:

1. The file system.
2. Message passing.
3. Shared memory.

All of these methods are quite common and a system can easily implement them all. There is no single option that is optimal in every situation, but each method has some areas of strength and weakness.

File System

One way for two processes to communicate is through the file system. Messages stored in the file system will be persistent and survive a reboot. It can also be used when the sender and receiver know nothing about one another (and the programmer knows nothing about any other IPC mechanisms).

The producer may write to a file in an agreed upon location and the consumer may read from that same location. The operating system is still involved because of its role in file creation and manipulation (as well as permissions for who may read and write a file).

If one file is being used then we have the problem of co-ordination: making sure one process does not overwrite the changes of another. We can get around this, however, by using multiple files with unique IDs. Consider an example from a co-op work term: if the producer is generating XML data, it can write in a file in a designated `import/` directory. The consumer program scans the directory, and when it finds files, reads the file and imports the data contained therein. The imported data is then shown in the program. In this case, since one process writes files and another reads them, there is no possibility that one process overwrites the data of another. As long as the sender chooses distinct file names, it will not overwrite a message if a second message is created before the receiver picks up the first.

Message Passing

Message passing is a service provided by the operating system where the sender will give the message to the OS and ask that it be delivered to a recipient. There are two basic operations: sending and receiving. Messages can be of fixed or variable size.

Our experience with postal mail, or e-mail, suggests that to send a message successfully, the sender needs to indicate where the message should go. And in the simplest case, we will send the message directly to the recipient process. This requires us to know the ID of that process, however, which may be considered a limitation. But let's get into it.

Using Signals. We got onto the topic of inter-process communication by talking about signals, so it makes sense to start with then. Signals are, as previously introduced, an interrupt with a specified ID. They don't contain any "message" within them, so they are somewhat limited. It's a little bit like how pagers work (strange as those are to think about, doctors still use them!). When someone is paged their pager vibrates and they can see only the number who paged them. So it's like a text message, but the body is blank. Nevertheless, the poor sleep-deprived overworked medical resident is alerted and will react appropriately, dealing with the situation before going back to whatever else they were doing before being paged.

As you can imagine, the fact that a signal contains no message is a limitation that means signals can't be used for every single interprocess communication scenario. That is true, but it is sufficient for some of them. When the fire alarm sounds in a building, you don't need an accompanying voice announcement to say "this indicates a fire alarm; please calmly exit the building" – you will figure this out just fine on your own (one hopes!). Some of the

reason it works out, though, is that you have previously been informed that when the fire alarm sounds it means you need to exit the building, and the same is true for signals: you need to know what to listen for and what's supposed to happen if you want to react accordingly.

The appropriate header for including signals is `signal.h` and it contains the definitions that let you write `SIGKILL` instead of having to put an explicit int `9` in your program. Earlier, we introduced a small table of some of the common UNIX signals. They are by no means exhaustive, but unfortunately there is not always 100% agreement between different implementations about what the higher signal numbers are. For this reason it's helpful to use the names, so if you use `SIGABC` then it abstracts away the fact that on one system that's implemented as being `X` and on another it's `Y`.

We already learned how to send a signal from the command line. But there are two functions for sending a signal programmatically:

```
int kill( int pid, int signo );
int raise( int signo );
```

Both functions return 0 if they were successful and -1 if they were unsuccessful (for whatever reason, such as no such process existing). The `raise` function sends the signal to the current process, so a nice little shortcut when you need it.

We need to know the process ID of the recipient. This is fine, but may require a little bit of negotiation for how processes find out about each other. A common convention is for services to register themselves in some way, which might be as simple as putting a file on disk in a specific location that contains the process ID. For example, mysql (a database) server will put its process ID in the file `/var/run/mysqld/mysqld.pid` and in that file is just the number of its process ID (e.g., `1494`). So if you know where to look, you can find the recipient's address. But this could also be communicated in any other way.

You can use `kill` to do some interesting things, like signal all your processes. It depends on the value chosen for your `pid` argument [SR13]:

- `pid > 0` – send the signal to the process with process ID `pid`.
- `pid == 0` – send the signal to all processes in the same process group as the caller.
- `pid == -1` – send the signal to all processes for which the calling process has permission to send a signal, except `init` and the calling process. Broadcast, basically.
- `pid < -1` – send the signal to all processes whose process group ID is equal to the absolute value of `pid`.

In addition to the signals we know, you can also invoke the `kill` function with a 0 argument for the signal. This is called the "null signal". It does not actually send any signal, but can be used to check if the recipient process exists. If not, the `errno` return value of `ESRCH` tells you that it no longer exists [SR13]. This is of limited utility, however, because (1) the process might exit between the time you check for its existence and the time you do something with that information; and (2) process IDs are only relatively unique so a given ID could be reused for a different process, and the recipient is no longer whom you expect it to be...

A signal can be sent to a given process, but that process can only actually deal with it when that process is running. A signal is generated by something, and it is later delivered to the recipient. But during the time between generation and delivery, we say the signal is *pending*. The pending signal is typically delivered at the first opportunity, which might be immediately if the recipient is currently executing [**?**].

Interestingly, for most (but not all) signals, your process can choose to refuse to listen. This is called blocking signals, and can be done to any with with the exception of `SIGKILL` and `SIGSTOP`. When a signal is blocked, it just remains in the pending state until signals of that type are unblocked. Blocking signals is supposed to be temporary, although a badly-behaved program could ignore them indefinitely.

If the same signal is sent more than once when that signal is blocked by the recipient, it might be delivered only once, depending on your particular operating system implementation [SR13].

As introduced earlier, signals have a default action. The action that is taken when the signal is delivered is called the *disposition* of the signal. If you don't explicitly change what happens when the signal arrives, the default (see the table) happens. But we can change it. There are three options (1) ignore it, (2) run a signal handler, and (3) run the default action. The third option is used to undo an earlier change (such as if we said we wanted to ignore the signal but now no longer do). We'll restrict our examination to the "run a signal handler" option in this lecture.

If we decide to register a signal handler, the function is:

```
void (*signal( int signo, void (*handler)(int))) (int);
```

Yikes! That is difficult to read. The good news is that using this in practice is a lot easier. What we actually do is call a function to say "for signal `X`, run function `foo`". There are restrictions on the function signature. It must have a `void` return type and take one parameter of an `int`. It returns a pointer to the old handler (if there was one). You could use that to set it back if you wanted.

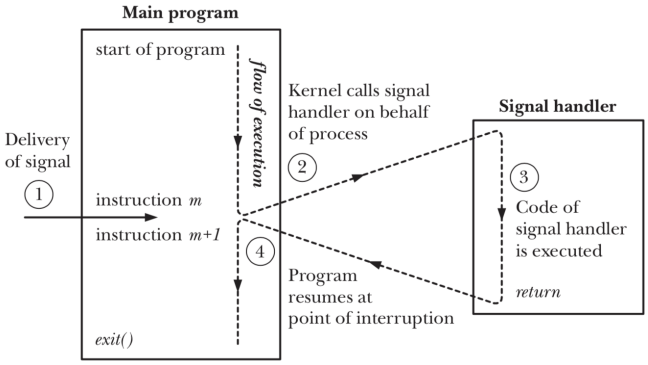
So a sample signal handler would be:

```
void sig_handler( int signal_num ) {
  /* Handle the signal in some way */
}
```

The integer parameter is set to the number of the signal received. This allows us to have one signal handler that handles many signals, if we wish, but differentiate between them at the time of invocation. To register the signal handler, the simplest way is to ignore the return value of `signal`:

```
signal( SIGINT, sig_handler );
```

Alright, with your signal handler set up, then the workflow looks like the diagram below:



Signal delivery, signal handler execution and return [**?**].

The content of your signal handler, however, is restricted. Because the handler deals with an interrupt and runs between two instructions it is important to make sure that the signal handler doesn't mess anything up. If the signal handler runs in the middle of `malloc` and the signal handler itself calls `malloc` it could put the memory management in an invalid state [SR13].

We can only use functions that are *reentrant*. That is to say, a function that can be interrupted during execution, have another complete call to that same function execute, and then resume (and have everything be okay). In particular the big ones to avoid are: `printf`, `malloc`, `free`, and anything that could possibly block the process (e.g., a read of a file!). The subject of reentrancy is a little bit more complicated than can be covered in a few sentences, and future courses should cover reentrancy in much more detail.

There are tables of what functions are safe to invoke from within a signal handler. In general what you are looking for is a designation of *async-signal safe*.

To block a signal, unblock one, or just find out what the current state is, the function is:

```
int sigprocmask( int how, const sigset_t * set, sigset_t * old_set );
```

The first argument is what we would like to do here: if `SIG_BLOCK`, the signals pointed to by `set` are added to the block list; if `SIG_UNBLOCK` then the ones in `set` are removed from the block list; if `SIG_SETMASK` then `set` is assigned to the signal mask (overwrite all current values)[**?**].

The third argument is optional, and if a pointer is provided then upon a change to the signal mask, `old_set` is updated to contain the values from before the change.

There is also the ability to manage signal disposition in a more advanced way using the function `sigaction`, but we will consider this beyond the scope of the course.

There are some helper functions to fill in the mask:

```
int sigemptyset( sigset_t *set ); /* Initialize an empty sigset_t */
int sigaddset( sigset_t *set, int signal ); /* Add specified signal to set */
int sigfillset( sigset_t *set ); /* Add ALL signals to set */
int sigdelset( sigset_t *set, int signal ); /* Remove specified signal from set */
int sigismember( sigset_t *set, int signal ); /* Returns 1 if true, 0 if false */
```

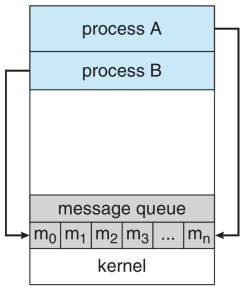
A quick example based on [**?**]:

```
sigset_t set;
sigset_t previous;

sigemptyset( &set ); /* Initialize set */
sigaddset( &set, SIGINT ); /* Add SIGINT to it */

sigprocmask( SIG_BLOCK, &set, &previous ); /* Add SIGINT to the mask */
/* SIGINT is blocked in this section */
sigprocmask( SIG_SETMASK, &previous, NULL ); /* Restore previous mask */
```

Finally, if you want to pause your program for a bit until the call is interrupted by a signal, there is the function `int pause( )`. This function always returns -1 and it suspends your program until the signal handler runs. This can be useful if we really do need to wait for something...

Pass Your Message. Earlier, it was mentioned that signals require you to know the recipient process ID, and also that they contain no message. We can now look at something that overcomes both of these limitations.

To deal with the process ID problem, what we would like is *indirect communication* where the messages are sent to mailboxes (queues). The queue is owned by the operating system, so it is persistent and independent of any particular process. The diagram below shows a message queue for communication between processes A and B:



A view of memory in a message-passing system [SGG13].

UNIX is kind enough to provide us with some message queues that sort of resemble this. The total number of queues, the maximum number of messages, and the maximum size of each message is dependant on the implementation of the operating system. It is unfortunately not possible to give you an exact answer to how many messages can fit. In a real implementation, limits are something you need to consider; you have three choices:

1. Wait for the space to be available (block).
2. Overwrite older messages (sometimes this is what you want).

3. Discard the current message (leave the old ones as they are).

The first step in message-passing is to obtain a *key* that identifies a specific IPC structure (the queue that we will use). Keys are just integer values, so we would like them to be unique (or at least close to it).

One method is to generate the key with the "file to key" function found in `sys/ipc.h` [**?**]:

```
key_t ftok( char *pathname, int proj );
```

The key is generated from the given file name (`pathname`) and the value `proj`. The file does have to exist, because the function uses its inode (file structure on disk that contains the metadata). And the integer argument allows generating multiple IPC objects based off the same file. There is a very small risk of duplicate numbers if we are unlucky, but the risk is small enough that we consider it acceptable.

Another way we can get a key is using the constant `IPC_PRIVATE`. If we give the constant in where a `key_t` is expected then a guaranteed unique key is returned. This method is used when there is a parent and child relationship between the processes that want to communicate.

Regardless of how we generate the key, we use it to get the queue with the function:

```
int msgget( key_t key, int flag );
```

The first parameter is the key that we have previously generated and is straightforward. The `flag` parameter starts with the UNIX permissions and can be modified with additional creation options. The permissions follow the UNIX permission standards, e.g. `0600`.If the queue is being created for the first time, use `IPC_CREAT`. If you want also be sure that this is being created anew, use bitwise-OR to combine `IPC_CREAT` with `IPC_EXCL` so that the call will fail if the queue already exists.

The return value is the queue ID of the queue we will use. Then we can send and receive messages. Alright, we're getting somewhere. But what does a message look like? Unlike in a lot of other contexts, here, the message has a defined structure:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

Wait... Does this mean a message can be only one character at a time? No, all it really means is that whatever message type you want to send has to have the first part be a long value; anything is fine after that. Such as [Hal15b]:

```
struct pirate_msgbuf {
    long mtype;  /* must be positive */
    struct pirate_info {
        char name[30];
        char ship_type;
        int notoriety;
        int cruelty;
        int booty_value;
    } info;
};
```

Right, so let's say that we want to send some data then:

```
int msgsnd( int msqid, const void *ptr, size_t nbytes, int flag );
```

The first parameter is the queue we want to send it to; the second is the message; the third is the data that we want to send. The last parameter is for what happens if the queue is full: normally we just want to wait, so this would be a blocking send, and you can provide 0 as the argument. But if you would prefer, you can use `IPC_NOWAIT`, in which case an attempt to add to the full queue will just return with an error instead of blocking.

And to receive:

```
ssize_t msgrcv( int msqid, void *ptr, size_t nbytes, long type, int flag );
```

The first argument is the queue to receive from as per usual; the second argument is the destination where the message will be copied to; the third argument is the number of bytes. The last parameter, again, can be used to specify that you don't want to wait if there is no message. The fourth parameter, `type`, is used to specify what kind of message you want. It corresponds to the `mtype` field in the message [SR13]:

- `type == 0` – Return the first message on the queue (any type)
- `type > 0` – Return the first message on the queue with the specified type
- `type < 0` – Return the first message on the queue whose type is the smallest value less than or equal to the absolute value of `type`.

When we are finished, we clean up the queue using [SR13]:

```
int msgctl( int msqid, int command, struct msqid_ds * buf );
```

To clean it up, the first argument is the queue ID we got back from `msgget` and the command we provide is `IPC_RMID` and we can give `NULL` as the last parameter. This deletes the queue as well as any data in the queue immediately. This function can do a number of other things, but we will consider the advanced configuration beyond the scope of this course.

Let's put it together in a simple example where the parent process sends a message to the child process:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <unistd.h>

struct msg {
  long mtype;
  int data;
};

int main( int argc, char** argv ) {
    int msgqid = msgget( IPC_PRIVATE, 0666 | IPC_CREAT );

    int pid = fork();
    if ( pid > 0 ) { /* Parent */
        struct msg m;
        m.mtype = 42;
        m.data = 252;
        msgsnd( msgqid, &m, sizeof( struct msg ), 0 );
    } else if ( pid == 0 ) { /* Child */
       struct msg m2;
       msgrcv( msgqid, &m2, sizeof( struct msg ), 42, 0 );
       printf("Received %d!\n", m2.data );
       msgctl( msgqid, IPC_RMID, NULL );
    }
    return 0;
}
```

7 — Sockets

Network Communication

Another form of inter-process communication is of course the idea of network communication. If two processes are not running on the same machine in the same environment, then to get them to communicate we can't use pipes or shared memory... instead, we must communicate over the network. But not only that: we can actually use the network as a method of communication if the two processes are on the same machine (as we will see).

A diagram describing network communication frequently represents the network as just a mysterious cloud or blob, and for the moment we will just pretend that is how it is. There is a whole course on how computer networks function and behave and what the various layers are. For the sake of simplicity we will focus just on how we use the network, and not how the network is implemented.

Sockets

The *socket* API is a standard dating back many years, describing how to communicate over the network in a standard way. The socket is the concept for how to establish a communication channel between two processes. There are really two ways that we can communicate: datagrams and connection streams.

A datagram is a lot like sending a letter in the mail (you know, snail mail): you can mail out letters but they can be delivered in any order, might get lost along the way, and are unidirectional [SR13]. The recipient of a letter can write one back if desired but it may not be. There's no "connection" to be established, it's just message delivery.

The stream is like making a telephone call. When you dial a number, the other side has to be available (i.e., not on the phone or away from their phone) and if they answer then a line of communication is established to allow exchange of data (talking). Then at some point one party or the other will hang up and that's the end of the call. The telephone network has to do quite a bit to get your data from one end to the other in a timely manner, but that's not really your concern at the time that you want to make the phone call.

Much like everything else in UNIX, a socket is handled like a file. It just so happens that when you open a socket, the data to be read or written from the "file" is being routed over the network somehow. To create a socket, we need the `sys/socket.h` header included and the call is [SR13]:

```
int socket( int domain, int type, int protocol )
```

The arguments require some explanation. The `domain` value defines the address format (amongst other things) and is how we would choose between, for example, IPv4 and IPv6 (or other more things...). For the purposes of this course, we'll use IPv4 and the constant for that is defined as `AF_INET` (address family: internet!).

The `type` argument gives what kind of information we are going to be sending. `SOCK_DGRAM` is for datagrams and `SOCK_STREAM` is for a bidirectional byte stream [SR13].

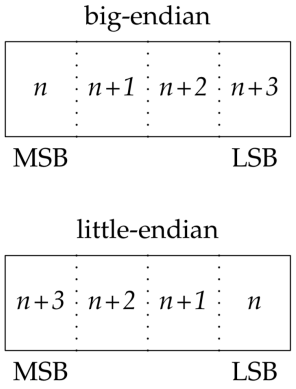
The `protocol` argument is about how the data is to be transported over the connection, and we will just use 0 for the default. If we chose a stream, the default is TCP/IP. Without going into too much detail, TCP/IP is the way a lot of data is transported over the internet. This is a (generally) reliable method of transport that makes sure your data gets to where it needs to go, with all the pieces in the correct order. If we chose a datagram, the default is UDP, which is not a reliable method of communication: data packets might get there or they might not.

Notice that the return type is an integer. Why is that? Well, a file descriptor is really just an integer, as we already saw with the pipe examples. So as long as we remember that's what the integer represents, we will not have any problems.

And just like a file that is opened, a socket that is opened needs to be closed when we are finished with it. For that we can just use `close`. You have probably noticed some asymmetry: there are different calls to open things (`open`, `pipe`, `socket`). That's because when we want to open something we have to say what kind of thing we want to open. When we want to close something, we already know what type it is.

Check the Boot of the Car for your Jumper! When we are going to communicate over the network we have to make sure that we speak the same language. More than that, we have to speak the same dialect (British English, for example, contains phrases and terms that you might not know if you only know Canadian or American English).

Normally this is not something that we need to think about, but when communicating over the network, the system on the other side might have a different idea of how data is organized. Consider, if you will, a four byte (32 bit) integer. There are two possible (reasonable) orders for how its bytes can be stored: smallest to largest or largest to smallest. These are shown below as big-endian and little-endian respectively:



Byte order for a 4-byte (32 bit) integer [SR13].

You might think that little-endian makes no sense. This is because you are a human reading a language left to right in a case where we put lower memory addresses on the left and higher ones on the right. The computer does not care about the convention. But anyway, x86 architecture is little-endian, but others (PowerPC) are not, so we can make no assumptions about what the architecture of the other side is. For this reason, network protocols specify a particular byte ordering so that everyone agrees on the meaning of the material. And this means that we need to translate the values to the big-endian format.

Included in the `arpa/inet.h` header are some functions to help us out. They translate from whatever system byte order you have to the network byte order, and vice versa. Their use is advisable even if you're sure the system you are using is big-endian, because of portability of your code. They come in 32-bit and 16-bit variants depending on what you need.

```
uint32_t htonl( uint32_t hostint32 ) /* Translate 4 byte int to network format */
uint16_t htons( uint16_t hostint16 ) /* Translate 2 byte int to network format */
uint32_t ntohl( uint32_t netint32 )  /* Translate 4 byte int to host format */
uint16_t ntohs( uint16_t netint16 ) /* Translate 2 byte int to host format */
```

Wait, these don't look like integers, or unsigned integers, do they? Well, sometimes we want to be very specific about the size of the integer. As you can imagine, this routine doesn't work if you don't know how big of an integer you're dealing with. And sometimes we will need to be specific about sizes.

Addresses. Right, so keeping this in mind, when we want to call someone, we have to put in their phone number. And if we want someone to call us, we need a phone number and we need to be ready to receive calls. This number has to be translated to the right format, sure, but we're able to do this using those functions above. The structure for a socket address is `struct sockaddr_in`. The structure has three fields, and below is a sample initialization [SR13]:

```
struct sockaddr_in {
  sa_family_t sin_family; /* Address family */
  in_port_t sin_port; /* Port number */
  struct in_addr sin_addr; /* IPv4 Address */
};

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons( 2520 );
addr.sin_addr.s_addr = htonl( INADDR_ANY );
```

As expected, if we want an IPv4 address then we use the matching family type of `AF_INET`. Then there are the IP address and port fields.

You're almost certainly familiar with IPv4 addresses... they take the format of `XXX.XXX.XXX.XXX` where each grouping of `XXX` is a number between 0 and 255. If you want to connect to your router, for example, you might type in `192.168.0.1` in the address bar of your browser. When you type in a name, like `uwaterloo.ca` there is a translation process that occurs that transforms this into an IP address, like `129.97.208.23`. The computer uses the number; names are nice for humans, though. But here we chose a constant value, `INADDR_ANY`. This says, choose the current IP address of the current computer (and it could have several if it has more than one network connection). We aren't trying to use any specific address; we just want one the computer has.

And what about ports? If you want to think about your computer's address as being like a street address, imagine that it's then an apartment building and the port number is which apartment the connection is made with. Different services (processes) are communicating over different ports. No two processes can be using the same port at the same time. By convention, ports with numbers below 1024 considered to be reserved for system services (and requiring superuser access to make use of). For our purposes, we will always choose ports above 1024.

When you log in to a server with `ssh`, for example, the default port for that is 22. This is a "well known" port, in that everyone agrees up front this port is the one for this service. Thus when a server starts up the daemon, it knows to listen for connections on port 22 and when you're ready to connect to that server, the `ssh` client chooses that port by default.

Looking Up the Address. In real life, we probably only rarely use IP addresses directly when trying to connect to services. If you already know it, that's fine, but more likely you use a human-friendly name. For example, you use `ssh username@ecelinux.uwaterloo.ca` and don't need to manually look up the IP address for the server. You can, of course (the command line tool is `nslookup`), but why should you have to? The computer can do this for you.

Looking up hostnames and the like is somewhat complex (and not the focus), so we will just learn one method for doing this. Many examples and older texts use the function `gethostbyname()`, but this is now deprecated and has been replaced with the `getaddrinfo()` function instead. You might still see it in the wild but we should learn the new way.

The function is prototyped in in `netdb.h` and looks like this [Hal15a]:

```
int getaddrinfo(const char *node,     // e.g. "www.example.com" or IP
                const char *service,  // e.g. "http" or port number
                const struct addrinfo *hints,
                struct addrinfo **res);
```

The `node` parameter is the hostname to connect to but can also be an IP address. The `service` parameter can be things like "http" to get the defined port for that protocol, but I really recommend you use explicit port number such as "80" for HTTP. The `hints` parameter is used (optionally) to restrict what kind of connection you want, such as saying you want IPv4, TCP Stream sockets, and letting the function fill in the IP address (see example below). And then there is a pointer to a `struct addrinfo *` (pointer to a pointer); that structure will be updated when the function is done. And the function does have a return value of an int, whereby 0 indicates success.

Okay, so let's try it out [Hal15a]:

```
struct addrinfo hints;
struct addrinfo *serverinfo;  // will point to the results
```

```
memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_INET;     // Choose IPv4
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

int result = getaddrinfo("www.example.com", "2520", &hints, &serverinfo);
if (result != 0) {
  return -1;
}
struct sockaddr_in * sain = (struct sockaddr_in*) serverinfo->ai_addr;
/* Do things with this */

freeaddrinfo( serverinfo );
```

Assuming that all went well, the `serverinfo` pointer is now pointing to a linked list of `struct sockaddr` (the generic form of `sockaddr_in`) which gives us the information we need: the IP address for the server we want to communicate with. The actual info struct is in the linked list node's `ai_addr` attribute and the pointer to the next one is `ai_next`. Most of the time we just need the first result, though. The `struct sockaddr` structure we got out of this can be used in future calls (although we might need to cast the result to the type we need).

If we are interested in getting the structure for the local computer, we can manually initialize the `struct sockaddr_in` as we did before learning about how `getaddrinfo()` works. Or we can call `getaddrinfo()` with `NULL` as the `node` parameter.

As some further notes, it's possible to use `NULL` for the hints if you are willing to accept the defaults (we usually are if we're sure about the type of result we will get). To deallocate the information that has been allocated, the call is `freeaddrinfo()` as shown above.

Client: Connect. Up until now, the tools we've learned about, `socket` and creating the `struct sockaddr` (one way or another) apply to both the client and server side in network communication. Now the paths diverge depending on whether the code we are running is the client or the server.

If we are the client, we'd like to connect to a server. This is the easier workflow. We just call `connect()`. This is done with [SR13]:

```
int connect( int sockfd, struct sockaddr *addr, socklen_t len);
```

The parameters are simple enough: the first argument `sockfd` is the socket file descriptor (the `int` we got back from the call to `socket`). The second parameter is a pointer to the `struct sockaddr` that we have, whether manually created or a returned value from `getaddrinfo`). The last parameter is about the size of the second parameter. If we manually created the structure, we just use `sizeof`; if it was returned from `getaddrinfo()` then there is also an attribute `ai_addrlen` provided. Consider an example using `getaddrinfo` [Hal15a]:

```
struct addrinfo hints;
struct addrinfo *res;
int sockfd;

memset(&hints, 0, sizeof( hints ));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.uwaterloo.ca", "80", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

int status = connect(sockfd, res->ai_addr, res->ai_addrlen);
```

The return value of `connect` determines whether we successfully connected or not. Zero means we were successful, anything else indicates an error. The man pages (i.e., manual pages) describing this function will tell you about specific error codes. For example, a quick search turns up `http://man7.org/linux/man-pages/man2/connect.2.html`. So if you are working with a system call and you get back a nonzero value, it's worth checking the man pages to see what it means. Printing out an error code directly isn't super helpful, but you can compare it against the constants defined in the man page. So if you check and see the status variable equals `ETIMEDOUT`, then you know the connection attempt timed out – you know what went wrong. The specifications don't always associate a specific number with a specific error (e.g., -7 means X) so you have to check against the constants in the implementation you have.

Assuming that you connected successfully, you're now ready to start using the connection. But before we do that, let's see what happens on the server side.

Server: Bind, Listen, and Accept. The overview of what steps the server is going to do is bind, listen, and accept. The bind step is how we choose what port we are going to connect to. The listen step is the part where we wait for connections from a client. Then the last step is accept, that is, establish the connection so we can start talking.

Step one is `bind()`: this is how we associate the socket with whatever port we want to use. When the `ssh` daemon is available for connection, it's because it has bound itself to the port 22 using `bind`.

So a quick example of using `bind` then, done without using `getaddrinfo`:

```
int socketfd = socket( AF_INET, SOCK_STREAM, 0 );
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons( 2520 );
addr.sin_addr.s_addr = htonl( INADDR_ANY );

bind( socketfd, (struct sockaddr*) &addr, sizeof( addr ));
```

With that done, we've acquired the resource of port 2520 for our use. We haven't done anything with it yet, but we've taken it for ourselves. You'll notice also this did not happen on the client side. This is because we don't care (usually) on the client side what the outgoing port number is. So we can just skip that step, unless we have a reason to care what the outgoing port is.

Step two is `listen()`: int this step we wait for incoming connections. This is the simplest step and you just call:

```
int listen(int sockfd, int backlog);
```

We listen on a socket that has been bound with `bind` and we'll allow a backlog up to `backlog` connections (which usually is limited to 20 or so, it depends on your system). If the queue is full the server system will reject additional requests.

So we've chosen a socket (got a phone number), we've said we're ready to listen (our phone is turned on), and then the next step is to `accept()` incoming `connect` requests (press the green icon). The `accept` call looks like this [SR13]:

```
int accept( int sockfd, struct sockaddr *addr, socklen_t *len );
```

The first parameter is, of course, the socket that we are listening to. The second and third parameters are the information about the client. We allocate these, pass them in, and they are updated by the call to `accept`. If we don't care at all about who the client is you can give in `NULL` for the second and third parameters. We don't strictly speaking need those values for communication in both directions, but it might be helpful in many contexts to know who the client is.

The return value is a new file descriptor which describes a new socket. Further communication takes places over that socket (and not the original one). The original socket is still used for accepting connections, and the new one is the socket used for communication with the client.

If `accept` is called and no requests are in the queue, the server is blocked until a request arrives. We simply wait for the connection.

Let's see a quick example of how to put all the pieces together now, skipping the error checking for compactness [Hal15a]:

```
struct sockaddr_in client_addr;
int client_addr_size = sizeof( struct sockaddr_in );
int newsockfd;

int socketfd = socket( AF_INET, SOCK_STREAM, 0 );
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons( 2520 );
```

```
server_addr.sin_addr.s_addr = htonl( INADDR_ANY );

bind( socketfd, (struct sockaddr*) &server_addr, sizeof( server_addr ));
listen( socketfd, 5 );
newsockfd = accept( socketfd, (struct sockaddr*) &client_addr, &client_addr_size );

/* Do something useful */

close( newsockfd );

/* Later when all is done */
close( socketfd );
```

Unless communication is a one-time thing, we probably call `accept` in some sort of loop, constantly accepting new connections and doing something useful with each, before going on to the next.

We could save ourselves some trouble by not caring about the client address:

```
int newsockfd;

int socketfd = socket( AF_INET, SOCK_STREAM, 0 );
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons( 2520 );
server_addr.sin_addr.s_addr = htonl( INADDR_ANY );

bind( socketfd, (struct sockaddr*) &server_addr, sizeof( server_addr ));
listen( socketfd, 5 );
newsockfd = accept( socketfd, NULL, NULL );
/* Do something useful */

close( newsockfd );

/* Later when all is done */
close( socketfd );
```

And then we are finally ready for the client and server to communicate (the client using its original socket file descriptor, and the server using the new file descriptor). As we've seen, there's a lot of setup when we learn about sockets for the first time. It is likely that in a program that does a lot with sockets, some of the lines of boilerplate code will be put into functions which can be invoked with just a few parameters. For example, you might write a client side function like this:

```
int connect_to( const char* host, const char* port );
```

This function does all the initialization, gets the address info, creates the socket, calls `connect`, checks for errors, and then returns the file descriptor for writing. Convenient! But we had to learn a bit about how the magic works before we could use it...

8 — Network Communication

Network Communication, Continued

We now have a good idea about how to establish a connection for communication, but have thus far not actually sent or received any data! Let's assume that all the setup from before is done and we have established a connection. If we want to send datagrams, the workflow is different and we'll come back to that afterwards.

Either side can send or receive data. Figuring out whose turn it is to talk and whose turn it is to listen is the job of client and server themselves. Also what the content is, and how to interpret it is the province of the client and server.

Send. The function for sending data is [SR13]:

```
int send( int sockfd, const void* msg, int length, int flags );
```

The first parameter is the socket file descriptor and this is either the one returned by the call to `socket()` (client-side) or `accept()` (server-side). Then there is the data to send (some arbitrary bytes). The third parameter is the length of whatever we are sending. The last parameter is the flags and for a simple use case we can just give `0` to the function for this.

The return value of the function is the number of bytes sent. If something went wrong, this will be -1 and the `errno` variable will tell you more about what exactly went wrong. Under ideal circumstances, the number of bytes sent equals the length parameter.

Let's consider a modified example from [Hal15a]:

```
char *msg = "Hello world!"
int len = strnlen( msg, 13 );
int sent = send( sockfd, msg, len, 0 );
```

And there you are, the data is sent! Assuming it went well. In real life it might be best to check for -1; unlike something like `malloc()` where the failure case is rare (is the memory allocator really going to be unable to give you an `int` often?), networks are tricky and can fail. Checking is worthwhile...

But wait a minute, we said under ideal circumstances the number of bytes is equal to the length. As you might have guessed, there is a limit to the amount of data that you can send in one chunk. The actual amount you can send in one chunk is reasonably-sized (somewhere around 1KB) but you can't just memorize a number and assume that will always be true across all systems. So if you have a significant chunk of data to send, you'll need to check how much was sent and then you are responsible for sending out the rest.

If the data being sent is large enough that this is a concern, or you are doing something that isn't just a test/demo application, you should track the number of bytes sent and keep calling send, updating the pointer as you advance. The code below is modified from [Hal15a] that sends as many times as necessary until we're down to 0 bytes left to send:

```
int sendall( int socket, char *buf, int *len ) {
  int total = 0;        // how many bytes we've sent
  int bytesleft = *len; // how many we have left to send
  int n;
```

```
  while( total < *len ) {
    n = send( socket, buf + total, bytesleft, 0 );
    if (n == -1) {
      break;
    }
    total += n;
    bytesleft -= n;
  }
  *len = total; // return number actually sent here
  return n == -1 ? -1 : 0; // return -1 on failure, 0 on success
}
```

Receive. And if you'd like to receive data, the call for this is `recv()`; its signature is almost the same but not identical [SR13]:

```
int recv( int sockfd, void * buffer, int length, int flags );
```

The socket file descriptor is the place to receive data from. The buffer parameter is the destination where the data goes, and the length is the maximum size of that buffer. Flags can also be 0 here.

The return value is the number of bytes actually read into the buffer. If you got back -1, then an error occurred and check `errno` for more details. If you got back 0, it means the other side hung up on you: they closed the socket. So no point in waiting anymore!

Knowing that the other side is finished sending data is not necessarily all that easy. That's something to work out with the sender. This is why in movies when characters communicate with walkie-talkies, they say "over" when they have finished their transmission... so the recipient will know they are finished talking. In network communication, we might know in advance the size of data we are supposed to get, or we might be told as part of the (negotiated) protocol, or we might need to wait for the other side to close the connection (then we know they are definitely done).

Suppose we are sending more than just a string. Can we do a fancy thing and write directly to a `struct` by making the buffer location the location of that `struct` and the length the `sizeof` that type? Yes, you can but this requires that the representation you receive over the network to be exactly the same as your struct. A more sensible approach is to serialize your data in some way, and then de-serialize it on the other side.

Serialization is the process of converting the data to some sort of byte-representation so that it can be sent across the network (or stored in a database, or anything really) and then later reconstructed via the deserialization process. This means that no particular data format is needed and systems that don't use the same software or architecture, even, can communicate easily.

In a practical scenario there's no need to write your own (de)serialization routine; there exist libraries like `protobuf-c` that are designed explicitly for this purpose. Pick a good one and use it.

But anyway, this is it! That's how we send and receive data. When we're done, we just call `close()` on the socket and that is the end. We now know how to communicate over the network.

Datagrams

But if you don't want to establish a communication channel and you just want to send a message (that is, calling is for old people and we just want to text people) there are two related functions for that instead [Hal15a]:

```
int sendto( int sockfd, const void* msg, int length, unsigned int flags,
    const struct sockaddr* to, socklen_t tolength )

int recvfrom( int sockfd, void* buffer, int length, unsigned int flags,
    struct sockaddr* from, int* fromlength )
```

Given what we've covered here nothing is really surprising. Because there's no connection established, each send has parameters for where to send the data to and each receive tells you where the data is being received from.

Send still returns the number of bytes sent and may be less than you were expecting if sending a large amount of data.

If you call `connect()` on a datagram socket, incidentally, you can then skip some of this and just use the regular `send` and `recv` operations – the transport is still UDP, but the source and destination don't need to be added every time.

In our next lecture, we will do an in-class exercise related to using sockets. The in-class exercise will provide an opportunity to use the socket code we've learned in a way that is more engaging than just seeing yet another code example in the lecture notes.

cURL

In most situations, however, we don't work with sockets directly when dealing with URLs. Instead we are likely to use cURL (or similar), a network communication and transfer request library. There is an associated command-line tool, but the client-side library is the area of interest because we can build on it in our program and save ourselves a lot of hassle. The curl library can do a lot of things and communicate via a lot of different protocols. It is quite flexible in that way. But it is only for the client-side and isn't meant to be used for server-side operations.

Imagine, if you will, that you want to access a webservice. Lots of things run via web service now and you are probably already somewhat familiar with them. In short, servers have "endpoints" that clients connect to via HTTP, and then the client can get a response. There are numerous examples of services that use this mechanism and they often adhere to some design principles like REST (REpresentational State Transfer). If we wished to communicate, for example, a GET request (to "get" some resource), then we can put together a connection via a socket and write the ""GET / HTTP/1.0\r\n"" into a string and send that message via `send()`. But there's no need to do it all by hand because we can do this very easily with libcurl.

As you might imagine, the information in this section is based on the official libcurl documentation. We don't cover every option or every detail, so it might be worth it to read some of the official documentation if you get stuck: `https://curl.haxx.se/libcurl/c/`.

This example, modified from `https://curl.haxx.se/libcurl/c/https.html` outlines all the key parts we need to use the curl library. We'll go through it and see what each of the parts do:

```
#include <stdio.h>
#include <curl/curl.h>

int main( int argc, char** argv ) {
  CURL *curl;
  CURLcode res;

  curl_global_init(CURL_GLOBAL_DEFAULT);

  curl = curl_easy_init();
  if( curl ) {
    curl_easy_setopt(curl, CURLOPT_URL, "https://example.com/" );
    res = curl_easy_perform(curl);

  if( res != CURLE_OK) {
      fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
    }
    curl_easy_cleanup(curl);
  }

  curl_global_cleanup();
  return 0;
}
```

Starting from the top then: we need to include the header `curl/curl.h`. If you are trying to compile and run a program on your own system and this header is not found, you need to install `libcurl-dev` on the system.

At the beginning of `main` there are two variables declared. The `CURL` structure is our "handle". This needs to be, like many structures, initialized before use and cleaned up when we're done with it. The `CURLcode` type is also a useful structure for finding out how operations went. Unlike in many other scenarios where success is assumed,

remember, in the network we should always check and see what happened.

All the code that uses the network is bracketed in the `curl_global_init()` and `curl_global_cleanup()`. For the initialization, we take the defaults with `CURL_GLOBAL_DEFAULT` which is sufficient. If we forget to call the global initialization function then the later call to `curl_easy_init` will do it for us, but this is dangerous because having it called more than once is an issue. And of course we should not forget to clean up when done.

Each connection is accessed through the handle, and so the handle is initialized with `curl_easy_init()`. If something goes wrong, the handle is set to NULL (hence why it can be used in the if-statement). If everything went well then we can actually get to work. If you are wondering why there is "easy" in the function names, it's not because there is a "hard" interface, but there is a "multi" interface that we will learn about in the future. (Whether you think the multi interface is hard is a separate discussion).

The next thing to do is set options for the connection. The only mandatory option its the URL to use (otherwise we don't know where to connect to, after all). In this case it's a dummy URL but it will do for the purposes of the code example. In the future we will need to set other options to do anything useful with a response, but for now the simple example leaves that out. There are something like 200 options, according to the documentation, so you can configure just about anything you want.

With all setup done, then we can perform the request with `curl_easy_perform`. This is the step that actually sends data over the network and retrieves a response. The result is stored in `res`.

Then we need to check and see if everything worked: `CURLE_OK` is the response code if all went well. Otherwise we get back something else and the function `curl_easy_strerror` interprets an error for us making it a little easier to understand what happened... isn't this a lot better than having to go google what the given value of `errno` is?

A handle can be used multiple times if you need, although you may need to update the options that are set on it to reflect the new things you'd like to happen

If we wanted to re-use a handle but clear all the settings there is `curl_easy_reset`, but that's not the case in this scenario since the example is a one-time thing. Instead, we clean up. When we're done with a given handle we clean it up with `curl_easy_cleanup`. And then of course when all is finished there's the global cleanup. Excellent!

Setting up Callbacks. Almost certainly, however, we want to do something useful with the data we got. Or, we might have some data that we need to send. For each direction, what we want to set up is a *callback*.

In case you haven't been introduced to the concept, it's a lot like what happens when you tell someone "when you're finished, call me back". They were in the middle of something when you called, and when a certain condition is fulfilled, they call you. In the context of using the cURL library, we tell the curl handle what function we would like it to call when the appropriate time comes.

The read callback is used when you are uploading data to the server (sometimes this is a POST operation). The write callback is used when you are receiving data from the server (this can be a GET operation). You may set up a read or write callback (or both) for an operation. There can be different callbacks for different easy handles, of course.

A write function has to have the following signature (according to the official documentation):

```
size_t write_callback( char *ptr, size_t size, size_t nmemb, void *userdata );
```

The name of the function can be anything you like. The `size_t` type represents a size and can be treated like an integer. The `ptr` points to whatever data we received and `nmemb` is the size of that data. `size` is always 1 (the documentation does not explain why this is the case). And the last parameter is user data: we can pass some data to the processing function directly. This is helpful when we have some data that we would need to provide to make processing easier or possible. If, for example, we want to write the requested data to a file with a specified name, we can pass that name in the `userdata` pointer, and then our implementation of the write callback should know how to use it. Finally, the return type is also a `size_t`; the spec requires that the returned size is the number of bytes of the data successfully processed. If it's not equal to the size of `nmemb` then the library interprets that as an error in writing.

A read function is very similar:

```
size_t read_callback( char *buffer, size_t size, size_t nitems, void *inputdata );
```

Again, the function can be called whatever you like. The `buffer` is the area where you are going to put the data to send (and it can't be any bigger than the next parameters specify). `size` here is the size of each data element and `nitems` is the number of items. In practice you will just want to calculate the maximum buffer size by multiplying these two things together. And the return value is the number of bytes successfully put there. We might need to do this in several chunks, though. If you put 0, it signals end-of-file and means no further upload will take place.

And to register the read and write callback respectively, there are two steps. One to register the function, and another to set the data:

```
CURLcode curl_easy_setopt( CURL *handle, CURLOPT_READFUNCTION, read_callback );
CURLcode curl_easy_setopt( CURL *handle, CURLOPT_READDATA, void *pointer );

CURLcode curl_easy_setopt( CURL *handle, CURLOPT_WRITEFUNCTION, write_callback );
CURLcode curl_easy_setopt( CURL *handle, CURLOPT_WRITEDATA, void *pointer );
```

Alright, on to an example, then, modified from `https://curl.haxx.se/libcurl/c/post-callback.html`.

```
#include <stdio.h>
#include <string.h>
#include <curl/curl.h>

const char data[]="Lorem ipsum dolor sit amet, consectetur adipiscing "
  "elit. Sed vel urna neque. Ut quis leo metus. Quisque eleifend, ex at "
  "laoreet rhoncus, odio ipsum semper metus, at tempus ante urna in mauris. "
  "Suspendisse ornare tempor venenatis. Ut dui neque, pellentesque a varius "
  "eget, mattis vitae ligula. Fusce ut pharetra est. Ut ullamcorper mi ac "
  "sollicitudin semper. Praesent sit amet tellus varius, posuere nulla non, "
  "rhoncus ipsum.";

struct data {
  char *readptr;
  size_t sizeleft;
};

size_t read_callback( void *dest, size_t size, size_t nmemb, void *userp ) {
  struct data *d = (struct data *) userp;
  size_t buffer_size = size * nmemb;

  if( d->sizeleft > 0 ) {
    /* copy as much as possible from the source to the destination */
    size_t copy_this_much = d->sizeleft;
    if ( copy_this_much > buffer_size ) {
       copy_this_much = buffer_size;
    }
    memcpy(dest, d->readptr, copy_this_much);

    d->readptr += copy_this_much;
    d->sizeleft -= copy_this_much;
    return copy_this_much;
  }
  return 0; /* no more data left to deliver */
}

int main( int argc, char** argv ) {
  CURL *curl;
  CURLcode res;
  struct data * d = malloc( sizeof( struct data ) );

  d->readptr = data;
  d->sizeleft = strlen( data );

  res = curl_global_init( CURL_GLOBAL_DEFAULT );
  if ( res != CURLE_OK ) {
    fprintf( stderr, "curl_global_init() failed: %s\n", curl_easy_strerror( res ) );
    return 1;
  }

  curl = curl_easy_init();
  if ( curl ) {
```

```
    curl_easy_setopt( curl, CURLOPT_URL, "https://example.com/index.cgi" );

    /* Now specify we want to POST data */
    curl_easy_setopt( curl, CURLOPT_POST, 1L );

    curl_easy_setopt( curl, CURLOPT_READFUNCTION, read_callback );
    curl_easy_setopt (curl, CURLOPT_READDATA, d );

    res = curl_easy_perform(curl);
    /* Check for errors */
    if(res != CURLE_OK) {
      fprintf( stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror( res ) );
    }
    curl_easy_cleanup( curl );
  }
  free( d );
  curl_global_cleanup();
  return 0;
}
```

A brief explanation is in order. Up at the top there is `data`, some amount of sample data and a little structure that is used to reference the data and to keep track of how much data there is to send. This is going to be important in the read callback function. And then there is of course the read callback function itself.

That function conforms, as you would expect, to the function signature as expected. If we are fortunate then it is possible to send the data in one chunk. But in this example we have intentionally made the data large. So we will now figure out how much we can actually fit in there by figuring out the size of the buffer and seeing if the data to be sent is bigger than the available space. Once we've figured out the limit, we can copy the data with `memcpy` to the buffer. Then we update where we are reading from and how much space is left. Then we return the amount copied. When all data has been copied, we return 0.

A large amount of the code of `main` is the same as in the previous example. Some new things of note, are obviously the initialization of the structure `d`. We set the option for the operation to be POST (upload data) since the default is GET (fetch some data), and then we have to register the read callback as well as send the data to be used when the read function runs.

In these examples and our brief overview, we have really only scratched the surface of what the curl library can do. But this is enough to get the flavour of how it works and to begin to do useful work with it.

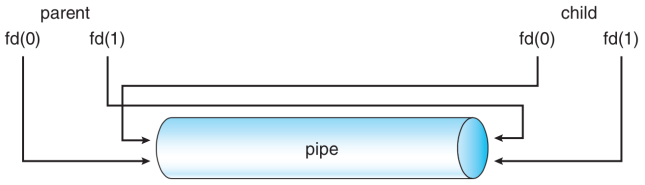
9 — Pipes and Shared Memory

IPC: Same System

In addition to the message-passing mechanisms thus far seen (signals, network), we can also use pipes and shared memory. Pipes share some significant similarity with both message-passing and shared memory, so it makes for a nice way to transition from one to the other.

UNIX Pipes

In UNIX, we can create a *pipe* to set up communication between a producer and consumer. The producer writes in one end of the pipe and the consumer receives it on the other. This is unidirectional, so if bidirectional communication is desired, two pipes must be used (going in different directions). The UNIX method for creating a pipe is `pipe` and it is constructed with the call `pipe( int fileDescriptors[])` where `fileDescriptors[0]` is the read-end and `fileDescriptors[1]` is the write-end [SGG13]. Yes, `fileDescriptors` means that UNIX thinks of a pipe as a file (UNIX thinks *everything* is a file...) even though it is in memory.



A UNIX Pipe [SGG13].

The pipe itself is a block of main memory that is interpreted as a circular queue, and each entry in the queue is fixed in size and usually one character. The sender may place the message into the queue in small chunks, but the receiver gets data one character at a time. Thus, the sender and receiver need to know when the message is finished. This may be through the use of a designated termination character (e.g., the line feed or null character), or the message may begin with an explicit value of the number of characters the message will be [HZMG15].

A UNIX pipe may be stored on disk. When this happens, we call it a *named pipe*. Unless we make it a named pipe, a pipe exists only as long as the processes are communicating. Furthermore, regular pipes depend on file descriptors, so a parent-child relationship is required to get the descriptors from one process to another. The named pipe, however, might be used by any process and will persist even after the creating process has terminated. Another nice bonus of named pipes is that they can be bidirectional, so we do not need two pipes for communication to go in both directions. With that said, communication can only go in one direction at a time; if concurrent communication is required, a second pipe is needed after all.

You may have worked with pipes on the UNIX command line. A command like `cat fork.c | less` creates a pipe that takes the output of the `cat` program and delivers it as input to the program `less` which allows for scrolling and pagination of that data.

UNIX Pipes: Code Example Let's consider an example from [SGG13] that combines the pipe concept with what we've seen before: using `fork` to spawn a new child process and then setting up a communication pipe between

the parent and child. We will send a message “Greetings” from the parent to the child.

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main( int argc, char** argv ) {
  char write_msg[BUFFER_SIZE] = "Greetings";
  char read_msg[BUFFER_SIZE];
  int fd[2];
  pid_t pid;

  if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
  }

  pid = fork();
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }

  if (pid > 0) { /* parent process */
    /* close the unused end of the pipe */
    close(fd[READ_END]);

    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg));

    /* close the write end of the pipe */
    close(fd[WRITE_END]);

  } else { /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);

    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s",read_msg);
     /* close the write end of the pipe */
     close(fd[READ_END]);
  }
  return 0;
}
```

In some of the runs of this program we sometimes get a random character after the “Greetings” message. What gives? Remember that in C a string is NULL terminated. So a `printf` routine with a `%s` qualifier will just keep going until it gets to a 0-byte somewhere. We are supposed to put the null terminator where it belongs, but this may not suffice. When we have a buffer (as we do here), it makes sense to initialize the buffer to 0. So, in this code, we should add a call to `memset` to initialize the `read_msg` variable to 0s.

If we wanted to create a named pipe, the system call for that is `mkfifo` (make first-in-first-out) because sometimes a named pipe is called a FIFO. As it is a file, it can be manipulated with the usual UNIX file system calls: `open`, `read`, `write`, and `close` [SGG13].

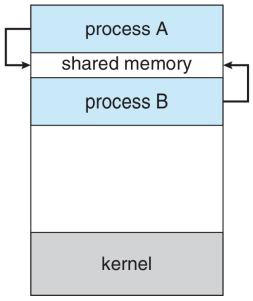
Shared Memory

Conceptually, the idea of shared memory is very simple. A particular region of memory is designated as being shared between multiple processes, all of whom may read and write to that location. To share an area of memory, the OS must be notified.

Normally, a region of memory is associated with exactly one process (its owner) and that process may read and write that location, but other processes may not. If a second process attempts to do so, the operating system will

intervene and that will be an error. If a process wants to designate memory as shared, it needs to tell the operating system it is okay for the other process to have access to that area. Furthermore, the OS needs to know that the memory is referenced by two processes: if the first one terminates and is reaped, the memory may still be in use by the second process, so that previously-shared region should not be considered free as long as the second process is still using it. Once the area of memory is shared, when either process attempts to access it, it is just a normal memory access. The kernel is only involved in the setup and cleanup of that shared area.

This situation is shown in the diagram below.



A view of memory when a block is designated as shared [SGG13].

Note that in the diagram the shared memory area appears in between the memory for processes A and B; this is not necessarily the case (the shared block could be anywhere). It tends to be in the block for the process that creates the shared memory in the first place. This makes sense, because Process A will request the memory from the operating system and then ask the OS to consider a particular block A already owns to be shared.

When a section of memory is shared, there exists the possibility that one process will overwrite another's changes. To prevent this sort of problem, we will need a mechanism for co-ordination... a subject we will return to later.

Suppose we want to share a section of memory. We need to obtain a *key* that identifies a specific memory segment. The way to obtain the key is the same as we previously discussed for message-passing: Either using the constant `IPC_PRIVATE`, or creating the key with the "file to key" function found in `sys/ipc.h` [**?**]:

```
key_t ftok( char *pathname, int proj )
```

The functions we need are in `sys/shm.h`. The workflow for using shared memory is as follows (with details obviously after) [**?**]:

- Create a new shared memory segment – `shmget`.
- Attach the shared memory segment (add the shared segment to the process calling attach) – `shmat`.
- Then the process can use the shared memory.
- Detach, from the shared memory segment (like deallocation, if forgotten then it will happen automatically when a process ends) – `shmdt`.
- Delete the shared memory segment (after all currently attached processes have detached), done by one process only – `shmctl`.

The functions are:

```
int shmget( key_t key, size_t size, int shmflg );
void* shmat( int shmid, const void* shmaddr, int shmflg );
int shmdt( const void* shmaddr );
int shmctl( int shmid, int cmd, struct shmid_ds *buf )
```

To create a shared memory segment, or get a reference to an existing one, we use `shmget`. The first argument is the key, which can be either the result of a `ftok()` call or the constant `IPC_PRIVATE`. The second argument is

obviously the number of bytes the shared memory should be. The third argument is the access permissions that follow the UNIX permission standards, e.g. `0600`.

The flags can also add `IPC_CREAT` (if no segment with the given key exists, create it) and `IPC_EXCL` (if "create" was used, fail if a segment with the given key exists). These are be combined with the permissions, and each other, with bitwise OR operator, like opening a file.

The return value of the function is the integer identifier of the shared memory segment. Given this identifier, it's possible to then attach that segment: add the shared memory to the running process.

When calling `shmat`, the first argument is the ID of the shared memory returned by `shmget`. The second parameter is where to put the memory, but in this case you should always use NULL and let the kernel decide where to put it. If you don't, your program may not work as expected on different machines (or even at different times!) because that memory might be in use. As for flags, we could use `SHM_RDONLY` which would prevent writing (actually an attempt would result in a segmentation fault) [**?**].

The return value of the call to `shmat` is a standard C pointer, containing the address of the shared memory. That's how we know where it is. Now we can read or write (if allowed) the data. At this point we are able to use the shared memory just like any other memory. Nothing special is needed – no read or write calls – just read and assign variables like any other memory.

Of course, a process that wants to attach to the memory has to know the key (or ID) to find it. If we created the segment ourselves, we obviously know where it is. But presumably you want some other process to have it as well. If two processes use the same input values for `ftok()` they will get the same result, so that's one method. Or, if a parent attaches a shared memory segment and then calls `fork()`, the child inherits the shared memory segments, so it's is already set up.

When we are done with a segment we can detach from it with `shmdt`, and it takes as an argument the value returned by `shmat`. It could have been the ID value, in my opinion, but this works as well. If we forget to do this, then it will happen automatically at process termination, but it is good programming practice to deallocate memory when you no longer need it, and similarly to detach from a shared memory segment when we are done.

To delete a shared memory segment, we use `shmctl`. This function can do a lot more than delete it, such as modify properties of the data structure that is used to control shared memory. But we just need to, at a minimum, delete the segment. To do so, call the function with the shared memory ID, and the command is `IPC_RMID` ("remove ID"). We must leave the last argument as `NULL` for this deletion.

Using the `shmctl` function with the deletion command doesn't (immediately) the segment. As long as it's still attached somewhere, the memory can't be cleaned up. In this respect, it's like a file in UNIX – if it's open then it can't be removed until the last process closes it. In Linux it's possible to attach to a still-existing segment if it's been marked for deletion, but this is not consistent behaviour across all UNIX systems and should not be done [**?**].

Consider a minimal (working) example where shared memory is employed to deliver a message from a child process to a parent.

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

int main( int argc, char** argv ) {

    int shmid = shmget( IPC_PRIVATE, 32, IPC_CREAT | 0666  );

    int pid = fork();
    if ( pid > 0 ) { /* Parent */
        waitpid( pid, NULL, 0 );
        void* mem = shmat( shmid, NULL, 0 );
        printf("The message the parent received from the child is %s.\n", (char*) mem );
        shmdt( mem );
        shmctl( shmid, IPC_RMID, NULL );
    } else if ( pid == 0 ) { /* Child */
```

```
        void* mem = shmat( shmid, NULL, 0 );
        memset( mem, 0, 32 );
        sprintf( mem, "Hello World" );
        shmdt( mem );
    }

    return 0;
}
```

Perhaps you have a small security concern: can anyone who knows the key or shared memory ID attach to my shared memory? No – it's only possible to attach to memory segments of your own processes. A process run by user j999doe can only attach to memory shared by another process run by that user. The owner can reassign ownership with the `shmctl()` call, but this is beyond the scope of the course.

Alternatively: `mmap`. An alternative approach for shared memory involves the use of `mmap()`, a function nominally used to map a file into memory. This is neat – it would allow us to map some disk file to a buffer, so when we fetch data from that buffer, the data is read and when we write into the buffer the data is written to the file. It allows us to read and write files as if they were memory accesses. Very convenient. But we can also use this for IPC, hence its introduction here. But first, the functions need to be introduced. As in [SR13]:

```
void* mmap( void* address, size_t length, int protection, int flag, int fd, off_t offset );
int mprotect( void* address, size_t length, int prot );
int msync( void* address, size_t length, int flags );
int munmap( void* address, size_t length );
```

The first argument is the address for where you want the mapped region to go in memory; unless you have a very good reason to do otherwise, let the system choose by supplying `NULL`. The next argument is the number of bytes we want to map. The third argument is the protection rules — read, write execute, and none – that apply (details below). The flag argument indicates the mode that the memory will be mapped in. The second-last argument is the file descriptor of the file we wish to map, and the last argument is the offset from the start of the file where the segment begins.

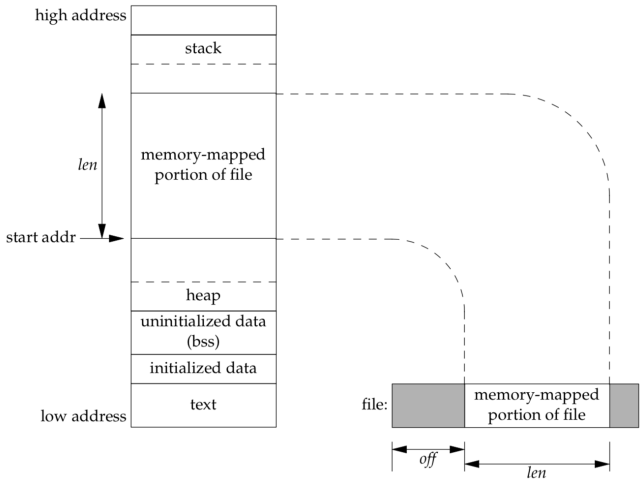
The return value is a pointer to the starting address of the memory area where the file is mapped.

Okay, so about the protection flags: valid values are `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, and `PROT_EXECUTE`, which can be combined using the bitwise OR operator (as always). The read, write, and execute ones look as expected, but whatever flags you choose have to be consistent with how the file was opened with `open`. So you can't use `open` as readonly and then somehow use `mmap` with the write flag. And `PROT_NONE` forbids all access; any attempt to access that memory results in a segmentation fault.

Digression: why would you ever use `PROT_NONE`? What's the point of mapping a segment of the file and then forbidding all access...? A possible use of this is to put "guards" at the start and end of a region of memory. If the process accidentally goes outside the bounds, then a segmentation fault is generated, thus enforcing the ends of the area [**?**].

And lastly, you have two flag options. The first is `MAP_PRIVATE` – modifications are not visible to other processes mapping the same file and are not written out to the underlying file. The other is `MAP_SHARED` – modifications are visible to other processes and are written to the underlying file (although perhaps not instantly – there is a function to make sure we all agree).

As you can imagine, the private option is for when we just want to work with a file ourselves and the shared option is for use in an IPC scenario.



Conceptual diagram of a memory mapped file [SR13].

If we wish to change the protection rules for a section, we use `mprotect`, which takes as arguments the address, the size, and the new access rules.

When a piece of the file is loaded in memory in the shared mode, file changes are not necessarily written out to the underlying file immediately. This is (partly) a performance enhancement but can mean that not everybody agrees on the content of the file because a change has not yet been written to disk. For this, the `msync` function is used: the address and length parameters are self explanatory. For flags we'll just say use `MS_SYNC` which makes this a blocking call. There are other more advanced options but we will not discuss that here. Also, this function does nothing if the file is mapped in private.

Finally, when we're done with a segment, we can unmap it with `munmap`, which takes the same arguments of the address and length. A segment would be unmapped automatically when a process exits, but as always it is polite to unmap it as soon as you know that you are done with it.

Let's do a quick example. Suppose I have a file `example.txt` that contains the text "This is sample text." We are going to overwrite it with "It is now overwritten" in a child process and the parent will see the updated message.

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

int main( int argc, char** argv ) {

    int fd = open( "example.txt", O_RDWR );

    struct stat st;
    stat( "example.txt", &st );
    ssize_t size = st.st_size;
    void* mapped = mmap( NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0 );

    int pid = fork();
    if ( pid > 0 ) { /* Parent */
        waitpid( pid, NULL, 0 );
        printf("The new content of the file is: %s.\n", (char*) mapped);
        munmap( mapped, size );
    } else if ( pid == 0 ) { /* Child */
       memset( mapped, 0, size ); /* Erase what's there */
       sprintf( mapped, "It is now Overwritten");
       /* Ensure data is synchronized */
       msync( mapped, size, MS_SYNC );
       munmap( mapped, size );
    }
```

```
    close( fd );
    return 0;
}
```

The example works acceptably in the sense that we successfully overwrite the data with the new data and the parent process sees the change. But things get weird if we tried to write fewer bytes than the original message was. In general, the mapped area size cannot change. So if we want to write more data than is in the file, we can't make the content of the file larger (and things look strange when we write fewer). In Linux there is the `mremap` call but this is not portable (ie does not work on all systems). But this would be great for something like sorting an array, wouldn't it? The sorted array is the same size as the input and we could share the work...

10 — Threads

Threads

Not very long ago we discussed what makes a process. A process has three major components: an executable program, the data created or needed by the program, and the execution context of the program (files opened, resources allocated, et cetera). A process has at least one *thread*, and can have many.

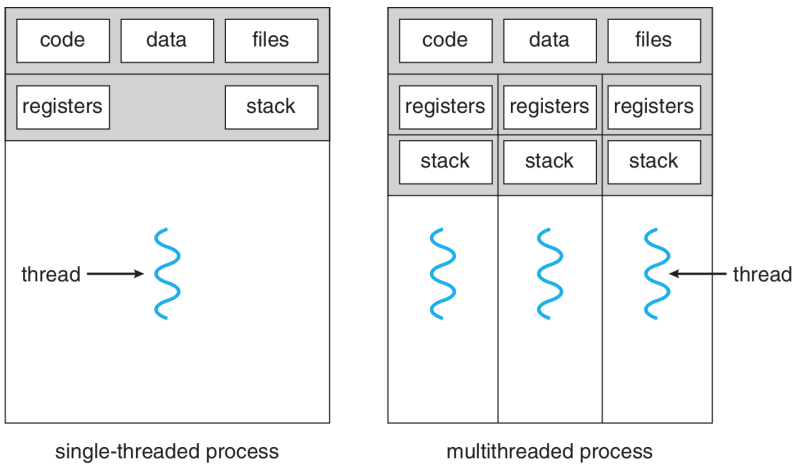
The term "thread" is a short form of *Thread of Execution*. A thread of execution is a sequence of executable commands that can be scheduled to run on the CPU. Threads also have some state (where in the sequence of executable commands the program is) and some local variables. Most programs you have written until now probably had only one thread; that is, your program's code is executed one statement at a time, sequentially in some order.

A multithreaded program is one that uses more than one thread, at least some of the time. When a program is started, it begins with an initial thread (where the `main` function is) and that main thread can create some additional threads if needed. Note that threads can be created and destroyed within a program dynamically: a thread can be created to handle a specific background task, like writing changes to the database, and will terminate when it is done. Or a created thread might be persistent.

In a process that has multiple threads, each thread has its own [Sta14]:

1. Thread execution state (like process state: running, ready, blocked...).
2. Saved thread context when not running.
3. Execution stack.
4. Local variables.
5. Access to the memory and resources of the process (shared with all threads in that process).

Or, to represent this visually:



A single threaded and a multithreaded process compared side-by-side [SGG13].

All the threads of a process share the state and resources of the process. If one thread opens a file, other threads in that process can also access that file.

The way programs are written now, there are few if any that are not in some way multithreaded. One common way of dividing up the program into threads is to separate the user interface from a time-consuming action. Consider a file-transfer program. If the user interface and upload function share a thread, once a file upload has started, the user will not be able to use the UI anymore (and Windows will put the dreaded "(Not Responding)" at the end of its dialog title), even to click the button that cancels the upload. For some reason, users hate that.

We have two options for how to alleviate this problem: when an upload is ready to start, we can call `fork` and create a new process to do the upload, or we can spawn new thread. In either case, the newly created entity will handle the upload of the file. The UI remains responsive, because the UI thread is not waiting for the upload function to complete.

Motivation for Threads

Why choose threads rather than creating a new process? The primary, but not sole, motivation is performance:

1. Creating a new thread is much faster than creating a new process. In fact, thread creation is on the order of ten times faster [TRG+87].
2. Terminating and cleaning up a thread is faster than terminating and cleaning up a process.
3. It takes less time to switch between two threads within the same process (because less data needs to be stored/restored). In Solaris, for example, switching between processes is about five times slower than switching between threads [SGG13].
4. Because threads share the same memory space, for two threads to communicate, they do not have to use any of the IPC mechanisms; they can just communicate directly. This is a bigger item than it might seem like, because inter-task communication can be very expensive.
5. As in the file transfer program, use of threads allows the program to be responsive even when a part of the program is blocked.

This last advantage, background work, is one of four common examples of the uses of threads in a general purpose operating system [Let88]:

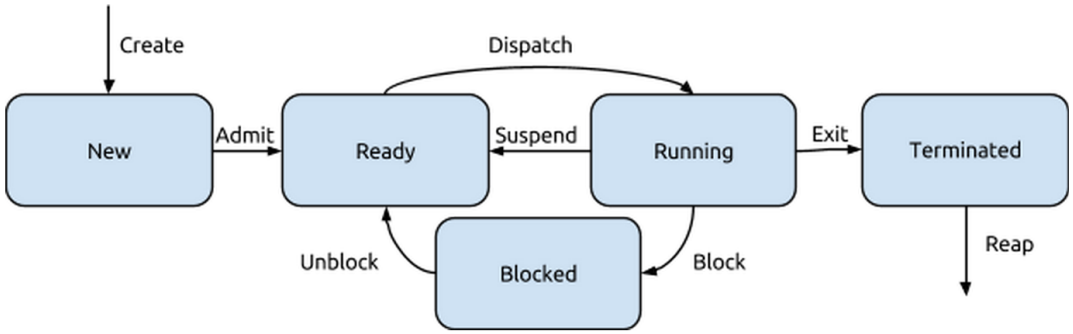
1. **Foreground and Background Work:** as already examined, the ability to run something in the background to keep the program responsive.
2. **Asynchronous processing**: for example, to protect against power failure or a crash, a word processor may write the document data in main memory to disk periodically. This can be done as a background task so it does not disrupt the user's workflow. You've probably experienced this in Microsoft Word, for example, if a document is "recovered".
3. **Speed of Execution:** a multithreaded program can get more done in the same amount of time. Just as the OS can run a different program when the executing program gets blocked (say, on a disk read), if one thread is blocked, another thread may execute. But also, we probably have a computer with many cores, so we can put them all to use.
4. **Modular Structure:** a program that does several different things may be given structure through threads. Threads have specific "jobs" and they each step in and do their job when it is appropriate for them to do so.

There are some drawbacks, however: there is no protection between threads in the same process: so one thread can easily mess with the memory being used by another thread. This once again brings us to the subject of co-ordination, which will follow the discussion of threads.

Also, if any thread encounters an error (such as a division by zero or Segmentation Fault), the whole process might be terminated by the operating system. If the program has multiple processes for different parts, then the other processes will not be affected; if the program has multiple threads and they all share the same process, then any thread encountering an error might bring all of them to a halt.

Thread States

Each individual thread will have its own state. We said earlier that a process may have seven states, but the model for thread state will be the somewhat simpler five-state model. If a process is blocked, we don't really care why, even if the OS does. Hence the five-state model, reproduced once again below:



State diagram for the five-state model.

The transitions work the same way as the state transitions for a process. As with a process, a thread in any state can transition to terminated even though that is not shown on the diagram. When a process is terminated, all its threads are terminated, regardless of what state it is in. The example we started with, the file transfer upload being cancelled, is an example of termination we should consider: thread cancellation.

POSIX Threads

The term `pthread` refers to the POSIX standard (also known as the IEEE 1003.1c standard) that defines thread behaviour in UNIX and UNIX-like systems (Linux, Mac OS X, Solaris...). This is a specification document that says how threads should behave. This standard lets code for one UNIX-like system (e.g., Solaris) run easily on another (e.g., Linux). The POSIX standard for pthreads defines something like 100 function calls, but we need not examine all of them. Let us focus on a few of the important ones and we can see they have some similarity to what we saw with parent and child processes [Tan08]:

- `pthread_create` – Create a new thread. This works a lot like `fork`.
- `pthread_exit` – Terminate the calling thread. This is like `exit` in that it ends execution and returns a value.
- `pthread_join` – Wait for a specific thread to exit. This is like `wait`: the caller cannot proceed until the thread it is waiting for calls `pthread_exit`. Note that it is an error to join a thread that has already been joined.
- `pthread_detach` – If we want to make it so that a thread cannot be joined, then we can make it a "detached" thread with this function.
- `pthread_yield` – Release the CPU and let another thread run. As they all belong to the same program, we expect that threads want to co-operate rather then compete for CPU time and threads can make decisions about when it would be ideal to let some other thread run instead.
- `pthread_attr_init` – Create and initialize a thread's attributes. The attributes contain things like the priority of the thread. ("After you, sir." "Oh no, after you.")

- `pthread_attr_destroy` – Remove a thread's attributes. Free up the memory holding the thread's attributes. This does not terminate the threads.
- `pthread_cancel` – Signal cancellation to a thread; this can be asynchronous or deferred, depending on the thread's attributes.
- `pthread_testcancel` – A thread can check to see if it has been cancelled. If that is the case, this function terminates the calling thread.

This list of functions gives us an overview of the toolkit we have, but we need to elaborate with some examples to fully understand how they work.

Creating a New Thread. When we want to start a new thread, we have to say what that new thread is supposed to do. The function signature for `pthread_create` looks like:

```
pthread_create( pthread_t *thread, const pthread_attr_t * attr, void *(*start_routine)( void * ), void *arg );
```

Where: `thread` is a pointer to a `pthread` identifier and will be assigned a value when the thread is created. The attributes `attr` may contain various characteristics (but you may supply `NULL` if you want the defaults). The third parameter is the function to run, but it requires a little more explanation.The last parameter, `arguments` is the argument passed to the `start_routine`. But that second last one is weird.

The `start_routine` parameter is the name of any function that takes a single untyped pointer and returns an untyped pointer. That is, the function signature has to match those two conditions. The name of the function (and the name of the argument) can be anything you like. See the example below:

```
void* do_something( void* start_params )
```

After the new thread has been created, the process has two threads in it. The OS makes no guarantee about which thread will be executing after the new one is created; this is a matter of scheduling. It could be either of the threads of the process, both of them at the same time, or a different process entirely.

Our experience with C-like languages suggests it is normal to have a single return value from a function, but usually we can have multiple input parameters. It seems limiting to be able to put in just one. There are two ways to get around this: with an array or with structures. In the case of the array, the argument provided to `pthread_create` is just a pointer to the array. This is also, incidentally, how you can get multiple return values out of a function in Java or C# (`public Object[] foo()`), but I don't recommend it as a good programming practice. The other way to do it is to use the `struct`, defining a structure for the parameter type and one for the return type.

The function that is to run in the new thread must expect a pointer to the arguments and then it will need to be cast to the appropriate (actual) type:

```
void* function( void * void_arg ) {
  parameters_t *arguments = (parameters_t*) args;
  /* continue after this */
}
```

This does imply that the caller of the `pthread_create` function has to know what kind of argument is expected in the function being called. That is fairly normal; we do have to know what the arguments mean when we pass them in to any function, but in this case we don't have the "hints" that the types provide.

What about the thread attributes? They can be used to set whether a thread is detached or joinable, scheduling policy, etc. By default, new threads are usually joinable (that is to say, that some other thread can call `pthread_join` on them). As noted before, it is a logical error to attempt multiple joins on the same thread. To prevent a thread from ever being joined, it can be created in the detached state (or the method `pthread_detach` can be called on a joinable thread). Trying to join a detached thread is also a logical error [Bar14]. For virtually all scenarios that we will consider in this course the default values will be fine.

Once we do that, the new thread we created is running. It does whatever its code does, so everything proceeds as expected, until of course the thread gets to the end. Usually, it will terminate with `pthread_exit`. The use of `pthread_exit` is not the only way that a thread may be terminated. Sometimes we want the thread to persist (hang around), but if we want to get a return value from the thread, then we need it to exit.

Returning Values. If a thread has no return values, it can just `return NULL;` which will have the same effect as `pthread_exit` and send `NULL` back to the thread that has joined it. If the function that is called as a task returns normally rather than calling the exit routine, the thread will still be terminated.

Another way a thread might terminate is if the `pthread_cancel` function is called with it as the target. As before, if the termination is deferred rather than asynchronous, the thread is responsible for cleaning up after itself before it stops.

A thread may also be terminated indirectly: if the entire process is terminated or if `main` finishes first (without calling `pthread_exit` itself). Indeed, `main` can use `pthread_exit` as the last thing that it does. Without that, `main` will not wait for other, unjoined threads to finish and they will all get suddenly terminated. If `main` calls `pthread_exit` then it will be blocked until the threads it has spawned have finished [Bar14].

Collecting Returned Values. Like the `wait` system call, the `pthread_join` is how we get a value out of the spawned thread:

```
pthread_join( pthread_t thread, void** retval );
```

The first parameter specifies the thread that you want to join. The second parameter is... wait... two stars? What we are looking for is a pointer to a void pointer. That is, we are going to supply a pointer that the join function will update to be pointing to the value returned by that function. Typically we supply the address of a pointer. This will be hopefully clearer in the example:

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

void * run( void * argument ) {
  char* a = (char*) argument;
  printf("Provided argument is %s!\n", a);
  int * return_val = malloc( sizeof( int ));
  *return_val = 99;
  pthread_exit( return_val );
}

int main( int argc, char** argv ) {
  if (argc != 2) {
      printf("Invalid args.\n");
      return -1;
  }
  pthread_t t;
  void* vr;

  pthread_create( &t, NULL, run, argv[1] );
  pthread_join( t, &vr );
  int* r = (int*) vr;
  printf("The other thread returned %d.\n", *r);
  free( vr );
  pthread_exit( 0 );
}
```

11 — Threads and Concurrency

More on Threads

We got a brief introduction to working with threads and learned how to create a new thread, how one exits (returning a value or not) and how to collect a value from it. Let's take a moment to consider cancellation.

Thread Cancellation. Thread cancellation is exactly what it sounds like: a running thread will be terminated before it has finished its work. Once the user presses the cancel button on the file upload, we want to stop the upload task that was in progress. The thread that we are going to cancel is called the *target* (because we shoot targets, I guess) and there are two ways a thread might get cancelled [SGG13]:

1. **Asynchronous Cancellation:** One thread immediately terminates the target.
2. **Deferred Cancellation:** The target is informed that it is cancelled; the target is responsible for checking regularly if it is terminated, allowing it to clean itself up properly.

The pthread attributes can be used to set the cancellation type before it is created. A thread can declare its own cancellation type through the use of the function:

```
pthread_setcanceltype( int type, int *oldtype )
```

The first parameter is the new state we'd like this thread to take on, which would be one of the constants `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`. The second parameter will be updated to point to what the previous state was (although we might not care).

In deferred cancellation, a thread is responsible for checking if it has been cancelled, and if so, and stopping its activity and cleaning up (closing open files, etc.) before it terminates. It's possible, though generally poor programming practice (and very difficult), to never check for cancellation.

Given that a thread can effectively ignore a cancellation if it is the deferred cancellation type, why would we ever choose that over asynchronous cancellation? Suppose the thread we are cancelling has some resources. If the thread is terminated in a disorderly fashion, the operating system may not reclaim all resources from that thread. Thus a resource may appear to be in use even though it is not, denying that resource to other threads and processes that may want to use it [SGG13].

The pthread command to cancel a thread is `pthread_cancel` and it takes one parameter (the thread identifier). By default, a pthread is set up for deferred cancellation. In the function that runs as a thread, to check if the thread has been cancelled, the function call is `pthread_testcancel` which takes no parameters.

Suppose your background task is to upload a bunch of files, consecutively. It is good programming practice to check `pthread_testcancel` at the start or end of each iteration of the loop, and if cancellation has been signalled, clean up open files and network connections, and then `pthread_exit`. Thus, if the thread has been told to cancel, it will do as it is told within a fairly short period of time.

It is noteworthy that a large number of functions are *cancellation points*; that is, the POSIX specification requires there is an implicit check for cancellation when calling one of those functions. There is an even larger number

of functions that are "potential cancellation points", where the specification says that they could be cancellation points (but maybe aren't). You'll have to check the spec to see if that is the case for a specific function if there is a scenario where unexpected cancellation is a problem.

Now's not a good time! With the presence of cancellation points or asynchronous cancellation, sometimes a thread can be terminated before it has cleaned up some resources. This is undesirable. One way that we can guard against this is to register cleanup handlers for that thread. If, say, our thread allocated some memory, it would be wise to register a cleanup handler that deallocates that memory in case the thread should die unceremoniously. The function signatures are:

```
pthread_cleanup_push( void (*routine)(void*), void *argument ); /* Register cleanup handler, with argument */
pthread_cleanup_pop( int execute ); /* Run if execute is non-zero */
```

To add a cleanup handler, the push function is used. Its two arguments are the function that is supposed to run, and a pointer to the argument that cleanup function will need.

The push function always needs to be paired with the pop function at the same level in your program (where level is defined by the curly braces). You should think of them as being like the opening curly brace at the start of a statement and the closing curly brace at the end; they have to be correctly matched up. The pop function takes one argument: whether it should run or not. If the thread is cancelled, the cleanup function will run; if it continues to the pop function, then you get to choose whether it runs or not.

Consider the following code:

```
void* do_work( void* argument ) {
  struct job * j = malloc( sizeof( struct job ) );
  /* Do something useful with this structure */
  /* Actual work to do not shown */
  free( j );
  pthread_exit( NULL );
```

Suppose that the thread is cancelled during the block operating on `j` and it is set up for asynchronous cancellation. This means that the code will never get to the `free()` call, which means that the memory allocated at the beginning is leaked! We can remedy this with application of a cleanup handler:

```
void cleanup( void* mem ) {
  free( mem );
}

void* do_work( void* argument ) {
  struct job * j = malloc( sizeof( struct job ) );
  pthread_cleanup_push( cleanup, j );
  /* Do something useful with this structure */
  /* Actual work to do not shown */
  free( j );
  pthread_cleanup_pop( 0 ); /* Don't run */
  pthread_exit( NULL );
```

And you may note that you could actually save a line of code by removing the `free()` call and changing the argument to the pop function to be 1: this means the cleanup function runs and it does free the memory allocated. Nice!

Attributes and Using Memory to Pass Data. The earlier example used the return value of a thread. Sometimes, of course, we don't want to do that. One of the advantages of the use of threads is that data can be passed between threads using memory directly. In this case, because there is no return value that we care about, we can use `NULL` in the call to join. This example also shows how to initialize and the attributes, although it doesn't override any of the defaults.

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

int sum; /* Shared Data */

void *runner(void *param);

int main( int argc, char **argv ) {
```

```
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if ( argc != 2 ) {
    fprintf(stderr,"usage: %s <integer value>\n", argv[0]);
    return -1;
  }
  if ( atoi( argv[1] ) < 0 ) {
    fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
    return -1;
  }

  /* set the default attributes */
  pthread_attr_init( &attr );
  /* create the thread */
  pthread_create( &tid, &attr, runner, argv[1] );
  pthread_join( tid, NULL );
  printf( "sum = %d\n", sum );
        pthread_attr_destroy( &attr );
  pthread_exit( NULL );
}

void *runner( void *param ) {
  int upper = atoi( param );
  sum = 0;
  for ( int i = 1; i <= upper; i++ ) {
    sum += i;
  }
  pthread_exit( 0 );
}
```

In this example, both threads are sharing the global variable `sum`. We have some form of co-ordination here because the parent thread will join the newly-spawned thread (i.e., wait until it is finished) before it tries to print out the value. If it did not join the spawned thread, the parent thread would print out the sum early.

Count to 10... Let's do a different take on that program:

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

int sum = 0;

void* runner( void *param ) {
    int upper = atoi( param );
    for (int i = 1; i <= upper; i++ ) {
        sum += i;
    }
    pthread_exit( 0 );
}

int main( int argc, char** argv ) {

    pthread_t tid[3];

    if ( argc != 2 ) {
        printf("An integer value is required as an argument.\n");
        return -1;
    }
    if ( atoi( argv[1]) < 0 ) {
        printf( "%d must be >= 0.\n", atoi(argv[1]) );
    }

    for ( int i = 0; i < 3; ++i ) {
        pthread_create( &tid[i], NULL, runner, argv[1] );
    }
    for ( int j = 0; j < 3; ++j ) {
        pthread_join( tid[j], NULL );
    }
    printf( "sum = %d.\n", sum );

    pthread_exit( 0 );
}
```

What's going wrong here? For very small values of the argument, nothing, but for a large number we get some strange and inconsistent results. Why? There are three threads that are modifying `sum`. And, it turns out, that this is a problem – if threads are trying to write to the same place at the same time, things can go wrong. But what does "at the same time" mean in the context of a program? To understand that, we need to think about how the hardware behaves and how the OS schedules the work.

Concurrency

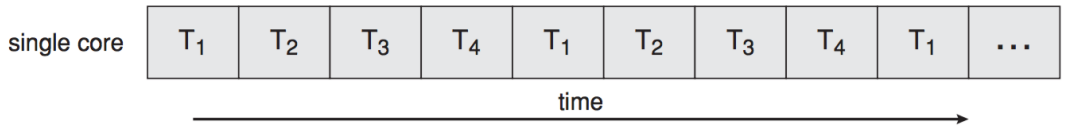
Not that long ago, a typical computer had one processor with one core. It could accordingly do exactly one thing at a time. When we say there is one processor, it's one general purpose processor that executes user processes. There may be additional special-purpose processors in the system (e.g., a RAID controller[2]) but there is only one general purpose processor so we call it a uniprocessor system.

Now, desktops, laptops, and even cell phones are almost certainly using multi-core processors. A quad-core processor may be executing four different instructions from four different threads at the same time. In theory, multiple processors may mean that we can get more work done in the same amount of (wall clock) time, but this is not a guarantee.

Terminology note: we often refer to a logical processing unit as a *core*. The term CPU may refer to a physical chip that contains one or more logical processing units. As far as the operating system is concerned, it does not much matter if a system has four cores in four physical chips or four cores in one chip; either way, there are four units that can execute instructions.

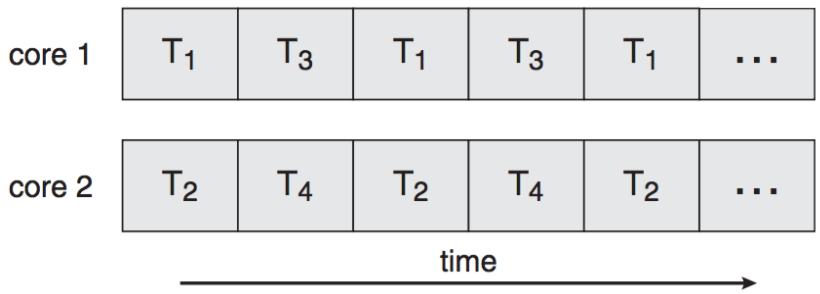
If there is exactly one process with one thread running in the system, then it does not matter how many cores are available: at most one core will be used to execute this task. If there are multiple processes, each process can execute on a different core. But what do we do if there are more processes and threads than available cores? We can hope that the processes get blocked frequently enough and long enough so that all processes get to run, but this is not something we can count on.

Our solution is that the CPU should switch between the different tasks via a procedure we call *time slicing*. So thread 1 would execute for a designated period, such as 20 ms, then thread 2 for 20 ms, then thread 3 for 20 ms, then back to thread 1 for 20 ms. To the user, it seems like threads 1, 2, and 3 are being executed in parallel, because 20 ms is fast enough that the user does not notice the difference.



Execution of different Threads T_1 through T_4 on a single core [SGG13].

Time slicing of execution will still occur, if necessary. Continuing our example, if there are four threads running on a dual-core system, time slicing is necessary to run all those programs.



Execution of different Threads T_1 through T_4 on two cores [SGG13].

[2]The graphics card seems like a more obvious example, but these days there are various programs that can make use of the powerful GPU to do general purpose computation.

Parallelism and Speedup

No doubt it has occurred to you that if there are multiple threads running at the same time, it means a task will get completed faster, right? Well... maybe. It depends a lot on what the task is. There is some overhead involved in splitting a task up and re-combining the results (if necessary), but in most circumstances the overhead is negligible compared to the amount of time working on the task.

If a task can be fully parallelized, it means the task can be split up in such a way that adding a second executing thread would double the speed of execution. Imagine painting an apartment. It would take one person 12 hours to paint the whole apartment and two people could complete the job in 6 hours. The pattern continues: three people can complete the job in 4 hours, four people in 3 hours, et cetera. This is the ideal, but in the real world there will be a limit to how many additional workers you can add and continue to have this speedup characteristic. At some point, the overhead of adding more threads is no longer negligible. In theory, you could hire 720 painters and finish the job in 1 minute, but at some point you cannot physically fit any more painters into the room.

If a task can be partially parallelized, it means the task can be divided, but doubling the workers doesn't result in completing the job in half the time. Two chefs working together in a kitchen might take 75% of the time it would take one chef to cook a meal. Adding the extra worker to the kitchen improved the speed at which food was prepared, but it's not doubled. The chefs can work independently some of the time, but at other times one has to wait for the other; the sauce cannot be put on the chicken until the chicken comes out of the oven.

If a task cannot be parallelized at all, then no amount of extra workers will speed it up. Some tasks can only be done sequentially. You can't cook the steak in one minute by putting it in five ovens (this makes the chef very mad).

Let us consider an example from [HZMG15]: Suppose we have a task that can be executed in 5 s and this task contains a loop that can be parallelized. Let us also say initialization and recombination code in this routine requires 400 ms. So with one processor executing, it would take about 4.6 s to execute the loop. If we split it up and execute on two processors it will take about 2.3 s to execute the loop. Add to that the setup and cleanup time of 0.4 s and we get a total time of 2.7 s. Completing the task in 2.7 s rather than 5 s represents a speedup of about 46%.

A smart fellow by the name of Gene Amdahl came up with a formula for the general case of how much faster a task can be completed based on how many processors we have available. Let us define S as the portion of the application that must be performed serially and N as the number of processing cores available. Amdahl's Law:

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

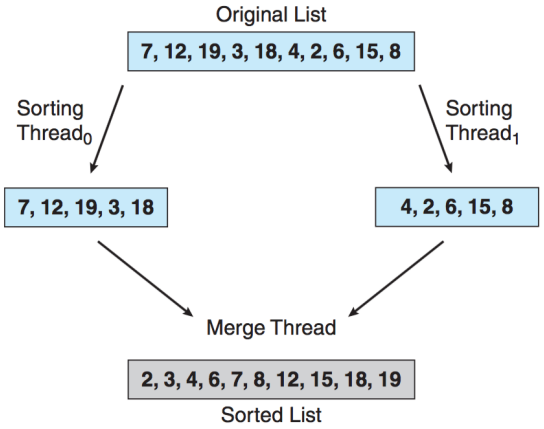
This is a math formula, after all, and you can do things like take the limit as N approaches infinity and you will find the speedup converges to $\frac{1}{S}$. So the limiting factor on how much additional processors help is, of course, the size of the S term in the equation. That squares well with our intuition about how this should work. If the task is completely sequential (cannot be parallelized at all), we cannot make it faster and $\frac{1}{1+0}$ will produce a maximum speedup of 1; or in other words... no speedup at all.

Applying this formula to the example from earlier, we get the following run times:

Processors	**Run Time (s)**
1	5
2	2.7
4	1.55
8	0.975
16	0.6875
32	0.54375
64	0.471875
128	0.4359375

There are two observations from this data immediately. The first is that we get diminishing returns as we add more processors. Going from 1 to 2 processors reduced the runtime dramatically, but going from 64 to 128 reduced the run time only a very small amount. The second is that as we continue to add more processors we are converging on a run time of 0.4 s, which fits our expectations of what would happen with infinite processors. The serial part will take 0.4 s no matter what, and with infinite processors the parallel part would be (effectively) instant. Again, applying the formula, the most we could speed up this code is by a factor of $\frac{5}{0.4} \approx 12.5$. It is not possible to do better than this. In reality we will never be able to equal the limit either, because nobody has infinite processors available, considering that would take an infinite amount of space and an infinite amount of money...

Merge Sort Example. Recall from data structures and algorithms the concept of merge sort. This is a divide-and-conquer algorithm like binary search. Split the array of values up into smaller pieces, sort those, and then merge the smaller pieces together to have sorted data. To get this done, we might have many threads sorting and one thread merging the sorted lists together into a larger, sorted list. Visually, this looks like:



Multithreaded sorting [SGG13].

12 — Concurrency: Synchronization and Atomicity

Synchronization

If a computer only ever did exactly one thing at a time (one process, one thread) we would have no concurrency and therefore no concurrency (co-ordination) problems. We have seen already, however, that in a system with multiple processes or multiple threads we have concurrency: more than one thread or process is making progress. And if the system is multicore, we can have parallelism: more than one thread or process actually executing on a CPU in a given instant. Either or both of these can lead to various problems.

The author of an application does not decide when a thread runs and when a thread switch will occur; these are things the operating system will decide. And the operating system does not usually give much thought to whether it is a convenient or inconvenient time to run a given thread.

The common English usage of the word "synchronization" refers to making two or more things happen at exactly the same time, but what we mean when we talk about synchronization in the computer sense is more general: it means we want some relationship between events, and the relationship can be before, during, after [Dow08]. There are two definitions that we need to consider at this point.

The first is *serialization*: we want there to be some sort of order of events. What we would like is to be certain that Event A takes place before Event B. We have already examined some examples of this, where we have a thread or process that `waits` on another. The merge sort algorithm, if implemented with two threads that sort the sublists and one thread that merges the results, necessarily implies that the merge thread has to wait for the sorting threads to finish. Thus the sorting (event A) must take place before the merging (event B). Or we could phrase it that B must happen after A.

The other thing we often want is *mutual exclusion*: events C and D must not happen at the same time. This is something we've referenced a few times before now, often phrased as the fact that we need some sort of co-ordination. In the discussion of inter-process communication, we said we might have an area of memory that is shared between two processes. If two processes can write to this area, there is some possibility that they both try to access the same place at the same time. If we have mutual exclusion, we are certain that P_1 writing to the shared area (event C) does not happen at the same time as P_2 tries to read it (event D).

Serialization through Messages

Suppose we have two people, Alice and Bob (these are the standard names in computer examples...) who work at the Springfield Nuclear Power Plant in Sector 7G, though they work in separate offices and cannot easily see one another. Alice works the day shift and Bob works the night shift. Suppose also that due to safety regulations, that Alice cannot go home until Bob has arrived and begun working. This is a situation where we would like serialization: the event of Bob's arrival must take place before the event of Alice's departure.

How can we get such behaviour? Well, the simple solution is that Alice stays at her desk and will not leave until she gets a call from Bob. Bob doesn't call until he's arrived at the office. This is a very simple scenario, but message passing is a valid solution for a lot of synchronization problems and it's the sort of thing we do in real life all the time: "Text me when you get here" or "Call me when you've finished".

If we are certain that Bob arrives before Alice departs, we can say that these events are *sequential*, because we know the order of events. At some point during the workday, Alice ate lunch, and at some point before work, Bob ate lunch. But we have no idea who ate lunch first, so we say they ate lunch *concurrently*. The formal, strict definition of concurrent is: two events are concurrent if we cannot tell by looking at the program which will happen first [Dow08].

In common parlance, when we say something happened concurrently, it means they happened at the same time (Alice and Bob both had lunch at 12:00). In this context, concurrency is not the same as saying that they happened at the exact same time; it's saying that we do not know (and cannot guarantee) an order of events. It's possible that Alice had lunch at 12:00 and Bob had lunch at 13:00 or they both ate at 12:00 or Alice had lunch at 13:30 and Bob had lunch at 12:15. We do not know. The order of concurrent events may change on two runs of the program; this is what we call nondeterminism.

In a non-deterministic program, we cannot tell just by looking at the program what the execution order would be. If we have two concurrent events, one in which the program prints "1" to the console and one in which the program prints "2" to the console, these could happen in any order. So we might see the output as "12" or "21". Which one will happen? Without some form of co-ordination, it could be either. This non-determinism makes it difficult to analyze a program. If "12" is the correct output but "21" occurs only very rarely, finding the cause and fixing it might be very painful [Dow08].

This is, incidentally, what is referred to as a "Heisenbug" (a portmanteau of Heisenberg and Bug). This has nothing to do with the "Breaking Bad" TV show, but with Werner Heisenberg (the physicist) and his scientific principle of uncertainty that says: if we precisely know the position of a particle we know nothing about its momentum and vice versa. The Heisenbug is frustrating because the harder we try to track it down, the less likely it is to occur.

Shared Data and Atomic Operations

We noted earlier that the need for co-ordination in inter-process communication arises from the fact that some area of memory is shared. We also know that all the threads of a process share the same data: in the merge sort algorithm the sorting and merging routines both operate on the same data array. Let's examine a simpler example (from [Dow08]) where a shared variable `x` is manipulated by two threads A and B, in pseudocode:

Thread A
```
A1. x = 5
A2. print x
```

Thread B
```
B1. x = 7
```

This is non-deterministic code: there is no co-ordination mechanism here so we cannot say what order these statements will occur. Some possible outcomes are: 5 is printed out and is the final value; 7 is printed out and is the final value, or 5 is printed out and 7 is the final value. Note that there is no way we can print out 7 and get a final value of 5, because we can be certain that statement A1 executes before A2; the problem is we do not know where in relation to the two A-statements that statement B1 will execute.

This example uses more than one thread, but we do not even need multiple threads to have a concurrency problem; just having interrupts in the system is sufficient.

Consider an application that is used to count occurrences of some event (whatever it is). We will store the count in a variable `count` and we will provide some facility for the user to reset the counter (the reset button). Each time we detect the event, we increment `count` with statement of `count++;` which seems like one single statement.

Even a seemingly simple operation like `count++;` is broken down into a series of smaller operations. You've just detected an event. Let's assume the current value of the variable is 4. Thus we want to increment the variable, which will require three steps.

1. Read the current value of `count` (read 4)
2. Add 1 to the value (now it's 5)
3. Write the changed value back to memory (write 5)

Now imagine an interrupt comes at the worst possible time. The interrupt is generated by the reset button: it's supposed to set the value of count to zero.

1. Read the current value of `count` (read 4)
2. Add 1 to the value (now it's 5)
3. INTERRUPT (control goes to the interrupt handler)
4. Write 0 to the variable (write 0)
5. END INTERRUPT (control returns to where it was before the interrupt)
6. Write the changed value back to memory (write 5)

At the end of this execution sequence, the variable `count` shows 5, but it should show 0 (or 1), which is certainly wrong. The user pressed the reset button but the count did not reset! If the reset interrupt had occurred before reading the variable, the count would have been reset, and then an event detected and the count goes up to 1. If the interrupt had occurred after writing the value 5 to the variable we would see the count set to 0 as is expected. If it occurred after the read but before the write, there is an error and the changes the interrupt handler made were lost.

This problem arises because the instruction `count++` is really three things (read, add, write) and can be interrupted at any time. When we are performing an operation that cannot be interrupted, we say it is *atomic*: indivisible. Although since about 1945, we have been able to split the atom (proving that atoms themselves are divisible and that humans are very good at finding ways to make things explode) the use of the word atomic in this context is not about nuclear physics but references the other meaning that stems from Greek word *atomos*, meaning indivisible (or more literally, not-cuttable).

Though there are usually some atomic operations available to us in a given system, we cannot be certain that all operations are indivisible. In fact, the opposite is likely to be true: we can be certain there are some operations that are non-atomic. Therefore we need to make sure at the very least that one operation on the shared variable is finished before the next begins.

A thought: can we do this with serialization? That is, can we make sure that the `count++` operation completes before the `count = 0` operation? That would eliminate the problem of the reset being ignored. Unlike the scenario where Alice waits for Bob to get to work before she leaves, in this case there is no obvious order between the events: the user may press the reset button at any time, even if no event has just occurred. Similarly, the event may be detected even if the user is nowhere around and not going to press the reset button. Our concept of serialization requires that there exists a correct order: first this, then that. Here, where both orders are valid, we need the other approach: mutual exclusion (also called *mutex*).

With mutual exclusion, we do not know or enforce any particular order of events, but what we do care about is that we do not have multiple threads trying to update the variable at the same time. It would mean that the action to reset `count` to zero would have to wait until the `count++` operation was completely finished (or vice versa) before it gets to execute.

Mutual Exclusion through Flags

So we have identified shared data as a potential source of error. A section of code that should be accessed by a maximum of one thread at a time is referred to as a *critical section*. The purpose of mutual exclusion is to ensure that at most one thread is in the critical section at a time. If we ever have more than one thread in the critical section at a time, something has gone terribly wrong. But on the other hand, the critical section is supposed to do something useful, so we cannot solve the problem by not allowing any thread to access it ever.

It is the responsibility of the programmer to identify what critical sections, if any, exist in the program, and to protect them with mutexes. Some analysis tools may exist to identify shared data, but ultimately the best analysis tool is taking a careful look. Critical sections should be as short as possible (but enclose all shared data accesses). The critical section is something that cannot be run in parallel, so it increases the magnitude of the S term in

Amdahl's Law, limiting the speed increase we can get from multiple threads and cores. Besides, it would be impolite to make other threads and processes wait unnecessarily.

Although we frequently say "the" critical section, there are likely to be multiple critical sections in the program, that are protected separately. A thread can be in critical section X while a different thread is in critical section Y and that's fine. For now, we'll consider just one critical section because that is the simplest case, but just remember in the back of your mind that there could be more in the program.

Our first approach then, is to have a variable to indicate if the critical section is currently in use. Suppose we have two threads:

Thread A

```
A1. while ( turn != 0 ) {
A2.    /* Wait for my turn */
A3. }
A4. /* critical section */
A5. turn = 1;
```

Thread B

```
B1. while ( turn != 1 ) {
B2.   /* Wait for my turn */
B3. }
B4. /* critical section */
B5. turn = 0;
```

This scheme enforces strict alternation: first it is the turn of thread A, then the turn of thread B, then back to A, and so on. What if thread A is to run more often than B? If thread B terminates then thread A will be stuck forever because the variable `turn` will always be saying that it is thread B's turn. This solution is obviously not satisfactory. Another approach then:

Thread A

```
A1. while ( busy == true ) {
A2.    /* Wait for my turn */
A3. }
A4. busy = true;
A5. /* critical section */
A6. busy = false;
```

Thread B

```
B1. while ( busy == true ) {
B2.    /* Wait for my turn */
B3. }
B4. busy = true;
B5. /* critical section */
B6. busy = false;
```

The problem with flags is that when we have a statement like `while (busy == true)` followed by `busy = true;` these are two distinct steps: read of `busy` and write of `busy` and a process switch could occur between the read and the write, which is the worst possible timing (as far as we are concerned). If the switch happens at the bad time then threads A and B will both be in the critical section at the same time. This solution is also not satisfactory.

What if instead of using one flag variable, we use an array where each thread writes to its own boolean variable?

Thread A

```
A1. flag[0] = true;
A2. while ( flag[1] ) {
A3.    /* Wait for my turn */
A4. }
A5. /* critical section */
A6. flag[0] = false;
```

Thread B

```
B1. flag[1] = true;
B2. while ( flag[0] ) {
B3.    /* Wait for my turn */
B4. }
B5. /* critical section */
B6. flag[1] = false;
```

Once again, this strategy is defeated by an untimely process switch: if statement A1 sets `flag[0]` to true and there is a switch to thread B, setting `flag[1]` to true, now both processes are stuck. Neither can advance, because each is waiting for the other to set its `flag` variable to false. This is, perhaps, slightly better than two threads in the critical section, but we have two threads that are permanently stuck now. This is also not satisfactory: although we want to forbid two threads from being in the critical section at the same time, keeping all threads out forever is not a good solution either.

The attempts at solution we have attempted so far have all been foiled by an untimely process switch, which will be triggered by an interrupt. This presents a possible solution: disabling interrupts. If interrupts are disabled, then interrupts generated from the user as well as the normal thread switches the scheduler would perform will not occur. Disabling interrupts is a crude solution, however, because during the time in which interrupts are disabled,

the system will be unable to respond to user input or other events (e.g., a fire alarm or detection of an incoming missile!). If an error is encountered in the critical section and the program is terminated, the system is effectively stuck because no other program will be able to run.

It gets worse: if we have multiple processors the disabling interrupts will not be sufficient [Sta14]. But maybe we're on the right track by getting hardware involved: the problem is we would like to read and maybe write a variable in such a way that cannot be interrupted. The hardware designers were aware of the problem and have kindly provided a facility to help us out: the *Test-and-Set* instruction.

Test-and-Set

The Test-and-Set instruction is a special machine instruction that is performed in a single instruction cycle and is therefore not interruptible. It is therefore an atomic read and write. The idea is that the Test-and-Set instruction returns a boolean value. When run, it will examine the flag variable (in this example, `i`) and if it is zero, it will set it to 1 and return true. If `i` is currently set to 1, it will return false. The meaning of the return value is clear: if it is true, it is the current thread's turn to enter the critical section. The Test-and-Set instruction is not actually implemented like this, but a description of its functionality in C is:

```
boolean test_and_set( int* i ) {
  if ( *i == 0 ) {
    *i = 1;
    return true;
  } else {
    return false;
  }
}
```

Why is it described like this? The problem is that English is both verbose and imprecise. If you wanted me to explain what test-and-set does using just English words, it would be a full paragraph and there might be some questions about the meaning of a particular word or phrase. This is why legalese is so dense: because it's an attempt to make English more rigorous.

Now, to make use of the `test_and_set` routine. Let us assume we have an integer variable called `busy` that is initialized at program startup to 0. The implementation is the same for both threads so there is no need to show them side by side.

```
while ( !test_and_set( busy ) ) {
   /* Wait for my turn */
}
/* critical section */
busy = 0;
```

Finally, we have something that will provide mutual exclusion without the risk that the threads will all get stuck because each thinks another is in the critical section. This is good, but can be improved. The `while` loop that is constantly checking the value with the Test-and-Set instruction is an example of *busy-waiting*. A given thread is constantly checking and checking and checking the instruction, and this is a waste of time and effort. Thread A will not get into the critical section while thread B is in there and asking constantly does not make B get the job done any faster, just as a child asking "are we there yet?" does not improve the speed at which he or she gets to his or her destination.

It is less wasteful of resources and effort if the `while` loop contains some instructions saying it should wait a little while before checking again (a sleep instruction). Serialization was achievable through messages; and they can be used to get mutual exclusion. This is the topic to be examined next.

13 — Semaphores

Mutual Exclusion through Messages

The earlier definition of mutual exclusion said only that one thread may be in the critical section at a time. This is the minimum, but there are additional desirable properties that will be used to evaluate any solution [Sta14]:

1. Mutual exclusion must apply (this criterion eliminated most of the flag examples earlier).
2. A thread that halts outside the critical section must not interfere with other threads (the strict alternation routine, even if implemented with Test-and-Set, would fail on this criterion).
3. It must not be possible for a thread requiring access to a critical section to be delayed indefinitely (the situation where all threads get stuck, each thinking another is in the critical section, would fail this criterion).
4. When no thread is in the critical section, a thread that requests access should be allowed to enter right away (no unnecessary waiting).
5. No assumptions are made about what the threads will do or the number of processors in the system (so it should be a general solution, not a special case).
6. A thread remains inside the critical section for a finite time only (this is more of an assumption than a criterion, but our solution must provide a way to indicate the thread has left the critical section).

Recall from earlier the example of the employees Alice and Bob who worked at the Springfield Nuclear Power Plant in Sector 7G. Suppose there is a third employee at the power plant, Charlie, who works on the day shift at the same time as Alice. Safety rules say that at least one of them has to monitor the safety of the reactor at all times and therefore they cannot both take lunch at the same time. If we cannot predict when lunch begins or how long it will last, how can Alice and Charlie co-ordinate to make sure they don't take lunch at the same time?

A possible solution: before Alice gets up from her desk to go for lunch, she calls Charlie. If he does answer, she may proceed. If Charlie does not answer, Alice will know he is not at his desk and she cannot leave at the moment. She can call again, constantly, until she reaches Charlie (busy-waiting), but this ties up a phone line nonstop and is effort intensive for Alice. If she doesn't want to do that, at this point she has two options: one is to simply wait some period of time (perhaps 15 minutes) and call again in the hopes that at that time Charlie will be back from lunch. A better approach would be for Alice to leave a message in Charlie's voice mail box, asking him to call her back when he has finished lunch. Then Alice can go about her work until she gets a call from Charlie and as soon as that happens, she may step out for lunch.

Busy waiting has already been found inadequate as a solution. It wastes CPU time that another process or thread could be putting to productive use. The approach of "wait 15 minutes and try again" might be adequate for Alice as a human, but for the computer it is not ideal. If thread B is in the critical section when A tries to get in, and A then sleeps for 2000 ms before trying again, this at least means A is not wasting CPU cycles while it tries futilely to ask "is it my turn now? Now? Now? Now? Now? Now?", but if B is finished after 20 ms, then thread A waits unnecessarily for 1980 ms.

What we want is something that resembles the call-when-finished semantics of Alice leaving a message and Charlie calling her back. That solution is called the Semaphore.

Semaphore

A semaphore, outside of the context of computing, is a system of signals used for communication. Before ships had radios, when two friendly ships were in visual range, they would communicate with one another through flag semaphores, which is a fancy way of saying each ship had someone holding certain flags in a specific position. Thus the two ships could co-ordinate at a distance, even if the distance was limited to visual range. This worked dramatically better than many alternatives (e.g., shouting).

The computer semaphore was invented in 1965 by Edsger Dijkstra, a brilliant Dutch computer scientist who is sometimes maligned in textbooks as being eccentric or unusual. He described a data structure that can be used to solve synchronization problems via messages in [Dij68]. Although the version we use now is not exactly the same as the original description, even 50+ years later, the core idea is unchanged.

We will begin with the *binary semaphore*: this is a variable that has two values, 0 and 1. It can be initialized to 0 or to 1. The semaphore has two operations: `wait` and `post`. In the original paper, `wait` was called `P` and `signal` was called `V`, but the names in common usage have become a little more descriptive. Mind you, if you can read/write Dutch as I can, the names make some sense: `P` is short for *proberen*, "to test", and `V` is short for *verhogen*, "to raise" or "to increment". But, for historical reasons as much as any other, the traditional lingua franca of computers is English, so the English names have tended to dominate. Furthermore, `post` is also called `signal` in many textbooks.

The `wait` operation on the semaphore is how a program tries to enter the critical section. When `wait` is called, if the semaphore value is 1, set it to 0 and this thread may enter the critical section and continue. If the semaphore is 0, some other thread is in the critical section and the current thread must *wait* its turn. The thread that called `wait` will be blocked by the operating system, just as if it asked for memory or a disk operation. This is sometimes referred to as decrementing the semaphore (because the value changes from 1 to 0).

The `post` operation is how a program sends the message that it is finished with the critical section. When this is called, if the semaphore is 1, do nothing; if the semaphore is 0 and there is a task blocked awaiting that semaphore, that task may be unblocked; else set the semaphore to 1. This is also sometimes called incrementing the semaphore.

If this is still confusing, consider the following analogy. Suppose you like coffee, and going to a particular coffee shop because there you can get your drink exactly the way you like it: half caf, no whip, extra hot, extra foam, two shot, soy milk latte[3]. After this delightful beverage it may be the case that you need to use the washroom. The washroom is locked at such places, so to get in you will need the key, which is available by asking one of the employees. If nobody is currently in the washroom, you will get the key and can proceed. If it is currently occupied, you will have to wait. When the key is returned, if anyone is waiting, the employee will give the key to the first person in line for the washroom; otherwise he or she will put the key away behind the counter.

Observe that the operating system is needed to make this work: if thread A attempts to wait on a semaphore that some other thread already has, it will be blocked and the operating system knows not to schedule it to run until it is unblocked. When thread B is finished and posts to the semaphore it is holding, that will unblock A and allow it to run again.

Note also that the semaphore does not provide any facility to "check" the current value. Thus a thread does not know in advance if it will block when it waits on the semaphore. It can only wait and may be blocked or may proceed directly into the critical section if there is no other thread in there at the moment.

When a thread signals a semaphore, it likewise does not know if any other thread(s) are waiting on that semaphore. There is no facility to check this, either. When thread A signals a semaphore, if another thread B is waiting, B will be unblocked and either thread A or thread B may continue execution (or both, if it is a SMP system), or another unrelated thread may be the one to continue execution. We have no way of knowing.

On the subject of observation, note that nothing about the semaphore as so defined protects against certain "bad" behaviour. Suppose thread C would like to enter the critical section. The programmer of this task is malicious as well as impatient: "my task is FAR too important to wait for those other processes and threads," he says, as he implements his code such that before he waits on the semaphore, he posts to it. Even though A or B might be in

[3] For the record, the author drinks tea, black.

the critical section, the semaphore gets incremented so he is more or less certain that his program will now get to enter the critical section. It's not foolproof: if there are other threads waiting, they might get woken up to proceed instead of C; much depends on the scheduler. Nevertheless, this is really bad: one process can wreak all kinds of havoc by letting another process into the critical section. Though the example here makes the author of thread C a scheming villain (because the example is funnier that way), such a situation may occur without malicious intent if it is simply the result of a programming error.

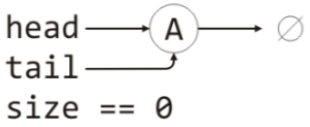
The problem identified in the previous paragraph is usually solved by supplementing the basic binary semaphore. A data structure called a *mutex* (from **mut**ual **ex**clusion) is a binary semaphore with an additional rule enforced: only the thread that has called `wait` may `post` to that semaphore. This adds a small amount of extra bookkeeping to the semaphore, but this is a reasonable price to pay.

Example: Linked List Integrity

We will now examine a situation where a semaphore helps to prevent a synchronization error. This example comes from [HZMG15]. Imagine we have a shared linked list defined as:

```
typedef struct single_node {
  void *element;
  struct single_node *next;
} single_node_t;

typedef struct single_list {
  single_node_t *head;
  single_node_t *tail;
  int size;
} single_list_t;


void single_list_init( single_list_t *list ) {
  list->head = NULL;
  list->tail = NULL;
  list->size = 0;
}

bool push_front( single_list_t *list, void *obj ) {
  single_node_t *tmp = malloc( sizeof( single_node_t ) );

  if ( tmp == NULL ) {
    return false;
  }

  tmp->element = obj;
  tmp->next = list->head;
  list->head = tmp;

  if ( list->size == 0 ) {
     list->tail = tmp;
  }

   ++( list->size );
   return true;
}
```

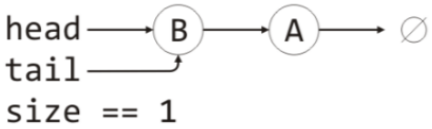
If only one thread can access this data structure, we do not have a problem, but it was a shared linked list. Suppose a thread runs and tries to add an element A to the list using the `push_front` function. Right before the increment of the `size` field takes place there is a process switch. At this point, the new node has been allocated and initialized, the pointers of `head` and `tail` have been updated, but `size` is 0.



The linked list at the time of the thread switch [HZMG15].

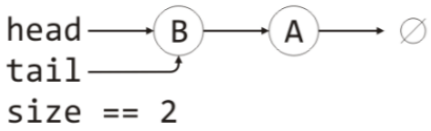
Now, the second thread executes and wants to add B to the linked list. In the conditional statement, `list->size`

== 0 evaluates to true. Thus, the tail pointer is updated.



The linked list after the second thread adds B [HZMG15].

When the first thread gets to run again, it will resume where it left off: it increments the size integer, leaving the final state: head and tail both point to element B, even though there is element A in the list.



The linked list after the first thread resumes [HZMG15].

This is an *inconsistent state*: the linked list has two elements in it but the tail pointer is wrong. An attempt to remove an element from the list will reveal the problem, which can manifest in a few ways, depending on how the removal routine is implemented. If we try to remove the front element we might check that head and tail are equal, and that may give the mistaken impression that B is the last element in the list, so we "lose" A and it becomes a memory leak. Or perhaps the head pointer will be updated but tail will still point to B even after it has been freed, which can result in a segmentation fault or invalid access.

Semaphore Syntax

Binary semaphores are useful, and we can generalize this concept to what is known as a *counting* or *general* semaphore. Instead of having only the values 0 and 1, the setup routine for the counting semaphore allows the choice of an integer value and this is the maximum value. A thread that waits on that semaphore will decrement the integer by 1; a thread that signals on the semaphore will increment the integer by 1. If a thread attempts to wait on a semaphore and the decrement operation makes the integer value negative, the calling thread is blocked. If, however, the semaphore is, for example, initialized with 5 and the current value is 2, a thread that waits on that general semaphore will not be blocked.

In UNIX, the semaphores are always general. So, the functions are:

```
sem_init( sem_t* semaphore, int shared, int initial_value);
sem_destroy( sem_t* semaphore )
sem_wait( sem_t* semaphore )
sem_post( sem_t* semaphore )
```

Of these functions, the only one where the parameters are not obvious is the initialization routine. The parameter shared will be set to either 0 or 1: 1 if the semaphore is to be shared between processes (e.g., using shared memory), 0 otherwise. I'll also take a moment also to point out the importance of getting the initial value correct. If we choose the wrong initial value then our program might get stuck or we might not get the mutual exclusion behaviour we are supposed to have.

Applying the Semaphore to the Linked List

Now that we have the appropriate syntax we can apply it to the linked list example from this section. We will add to the linked list structure (struct single_list) a semaphore: sem_t sem;. In the initialization routine, we need to call the initialization method: sem_init(&(list->sem), 0, 1);

Finally, the semaphore_wait and semaphore_signal operations need to be added to push_front at the start and end of the critical section, respectively. Recall from earlier that we want the critical section to be as small as it can be. Putting it all together:

```
typedef struct single_node {
  void *element;
  struct single_node *next;
} single_node_t;

typedef struct single_list {
  single_node_t *head;
  single_node_t *tail;
  int size;
  sem_t sem;
} single_list_t;

void single_list_init( single_list_t *list ) {
  list->head = NULL;
  list->tail = NULL;
  list->size = 0;

  sem_init( &( list->sem ), 0, 1 );
}

bool push_front( single_list_t *list, void *obj ) {
  single_node_t *tmp = malloc( sizeof( single_node_t ) );

  if ( tmp == NULL ) {
    return false;
  }

  tmp->element = obj;

  sem_wait( &( list->sem ) ); {
    tmp->next = list->head;
    list->head = tmp;

    if ( list->size == 0 ) {
       list->tail = tmp;
    }
    ++( list->size );

  } sem_post( &( list->sem ) );

  return true;
}
```

Strictly speaking, the braces ({ }) to enclose the critical region (between the wait and signal operations) are not necessary. This is just a use of C syntax to make it more obvious what the critical region is and to make it harder to make a mistake.

The critical section here just encloses the modification of the shared linked list. In theory one might put the wait and signal operations at the start and end of the entire function, respectively. This is, however, suboptimal: it forces unnecessary waiting. In this specific example, including the call to `malloc` is especially bad; the memory allocation itself can block if insufficient memory is available. Thus the process currently in the critical section is blocked and that means no other thread can enter the critical section. This might result in the system getting totally stuck [HZMG15].

14 — Synchronization Patterns

Synchronization Patterns

There are a number of common synchronization patterns that occur frequently and we can use semaphores to solve them. These synchronization patterns are ways of co-ordinating two threads or processes. We have already examined serialization and mutual exclusion, but there are many more. Throughout this section we will use pseudocode and something like "Statement A1" could be any valid statement in the program; what exactly it does is not important as the semaphore is a general solution. The examples in this section come from [Dow08].

Signalling

Recall from earlier the example with Alice and Bob at the power plant. This was signalling: Bob sends a signal to Alice to indicate that he has arrived to work and Alice may go home. Signalling can be used in general as a way of indicating that something has happened. Suppose we have a semaphore named `sem`, initialized to 0.

Thread A
```
1. Statement A1
2. post( sem )
```

Thread B
```
1. wait( sem )
2. Statement B2
```

If B gets to the `wait` statement first, it will be blocked (as the semaphore is 0) and cannot proceed until someone posts on that semaphore. When A does call `post`, then B may proceed. If instead A gets to the `post` statement first, it will post and the semaphore value will be 1. Then, when B gets to the `wait` statement, it can proceed without delay. Regardless of the actual order that the threads run, we are certain that statement A1 will execute before statement B2.

Note that this is a situation where it makes sense for a thread to post a semaphore even if it is not the thread that waited on that semaphore. Thus, sometimes, the semaphore is appropriate and the mutex structure is not necessary in every circumstance.

Rendezvous

The rendezvous is an expansion of the signalling pattern so that it works both ways: two threads should be at the same point before either of them may proceed (they "meet up"). Given the following code:

Thread A
```
1. Statement A1
2. Statement A2
```

Thread B
```
1. Statement B1
2. Statement B2
```

The desirable property is that A1 should take place before B2 and that B1 should take place before A2. As each thread must wait for the other, two semaphores will be needed: one to indicate that A has arrived and one for B. We will assign them the names `aArrived` and `bArrived` and initialize both to 0. A first attempt at a solution:

Thread A
```
1. Statement A1
2. wait( bArrived )
3. post( aArrived )
4. Statement A2
```

Thread B
```
1. Statement B1
2. wait( aArrived )
3. post( bArrived )
4. Statement B2
```

The problem here should be obvious: thread A gets to the `wait` statement and will wait until B posts its arrival before it can proceed. Thread B gets to its `wait` statement and will wait until A posts its arrival before it will proceed. Unfortunately, each thread is waiting for the other to post and neither of them can get to the actual `post` statement because they are both blocked.

Neither thread can proceed. The situation can never be resolved, because there is no external force (e.g., a hardware interrupt that means the data has been read from a device) that would cause one or the other to be unblocked. This is a situation called *deadlock*, and it is a subject that will receive a great deal of examination later on. For now, an informal definition is: all threads are permanently stuck. Obviously, this is undesirable.

What if instead, the threads reverse the order and post first before waiting?

Thread A
```
1. Statement A1
2. post( aArrived )
3. wait( bArrived )
4. Statement A2
```

Thread B
```
1. Statement B1
2. post( bArrived )
3. wait( aArrived )
4. Statement B2
```

This solution works: if A gets to the rendezvous point first, it posts its arrival and waits for B. If B gets there first, it posts its arrival and waits for A. Whichever gets there last will post and unblock the other, before it calls `wait` and will be able to proceed directly because the first thread to arrive already posted.

A variation on this can also work where only one thread posts first and the other thread posts second. This is shown below:

Thread A
```
1. Statement A1
2. wait( bArrived )
3. post( aArrived )
4. Statement A2
```

Thread B
```
1. Statement B1
2. post( bArrived )
3. wait( aArrived )
4. Statement B2
```

While this solution will not result in deadlock, it is somewhat less efficient than the previous: it may require an extra switch between processes. If A arrives at the `wait` statement, it waits; then B posts and must then wait for A's post (it cannot proceed right away even though it got there second). After A gets to run again it may proceed to post and unblock B. For the most part, we are usually satisfied as long as we are certain that deadlock will not occur that a given solution is acceptable. Nevertheless, the previous solution is provably better and is what programmers want to implement.

Mutual Exclusion

We saw previously the motivation and concept of mutual exclusion through messages in the linked list example. The general form in pseudocode is of course:

Thread A
```
1. wait( mutex )
2. critical section
3. post( mutex )
```

Thread B
```
1. wait( mutex )
2. critical section
3. post( mutex )
```

The mutex semaphore is originally initialized to 1 (unlike the previous examples where it started at 0), so whichever thread gets to the `wait` statement first will proceed immediately and not be blocked at all. If the semaphore were initialized to 0 then neither thread could ever get to the `post` statement or ever get into the critical section (deadlock).

Note that both threads A and B are identical here, which was not always the case in previous examples. This is a *symmetric* solution. It is easier to make predictions about the behaviour of the threads when they all do the same thing. If the different threads have different sections of code, they are *asymmetric*. The symmetric solutions very often scale well: we could have arbitrarily many threads executing in that same pattern, as long as they all wait on the semaphore before entering the critical section, we can be sure mutual exclusion is enforced.

Multiplex

In addition to the binary semaphore, we also discussed the general semaphore. If the general semaphore is initialized to n, then at most n threads can be in the critical section at a time. This pattern is more common than it might appear at first glance. Restaurants have a certain number of tables and seats. If more people wish to dine than there are seats available, those customers must wait until some seats become available (other customers leave). Restaurants generally want to pack in as many seats as possible, but fire safety regulations set a maximum occupancy for a given space.

In a computer related example, suppose that the system has a problem that when too many concurrent database requests are happening. The queries become slow and eventually time out. A potential solution is to protect all database accesses with a binary semaphore, so only one database query can run at any time. Analysis may reveal that this is too restrictive a policy; perhaps we can execute 5 queries concurrently without any slowdown. Then initialize the semaphore with a value of 5, allowing at most 5 threads into the critical section at any time.

Now, to represent this in pseudocode. This is a symmetric solution, so it will work for arbitrarily many threads (and showing A and B side by side is not necessary).

Thread *K*

```
1. wait( mutex )
2. critical section
3. post( mutex )
```

This looks exactly like the solution for mutual exclusion, as it should. The only difference is how many threads can enter the critical section at a time (1 vs. many).

Barrier

The barrier pattern is a generalization of the rendezvous pattern; a way of having more than two threads meet up at the same point before any can proceed. Given n threads, each of which knows that the total number of threads is n, when the first $n - 1$ threads arrive, they should wait until the nth arrives. As a solution we might consider a variable to keep track of the number of threads that have reached the appropriate point. Because this variable is shared data, modification of it should be in a critical section. Thus we will have a semaphore, initialized to 1, called `mutex` to protect that counter. Then we will have a second semaphore, `barrier` that will be the place where threads wait until the nth thread arrives.

Thread *K*

```
1. wait( mutex )
2. count++
3. post( mutex )
4. if count == n
5.     post( barrier )
6. end if
7. wait( barrier )
```

When the nth thread arrives, it unlocks the barrier (posts it) and then may proceed. Unfortunately, this is not a solution, because it will lead to some threads being permanently stuck. If there is more than one thread waiting at the barrier, the first thread will be unblocked when the nth thread posts on it. However, after that, there are no other `post` statements and therefore the other threads waiting are stuck forever, waiting for a post that will never come.

Perhaps the idea occurred to you that the nth thread to arrive should post n times:

Thread *K*

```
1. wait( mutex )
2. count++
3. post( mutex )
4. if count == n
5.     for i from 1 to  n
6.         post( barrier )
7.     end for
8. end if
9. wait( barrier )
```

This is a solution that allows all of the n threads to proceed (none get stuck), but it is less than ideal. The thread that runs last is very likely the lowest priority thread (if it were high priority it would likely have run first) and therefore when it posts on the semaphore, the thread that has just been unblocked will be the next to run. Then the system switches back, at some later time, to the thread currently unblocking all the others. Thus, in the worst case, there are $2n$ process switches, when it could be accomplished with just n [HZMG15]. Have each thread unblock the next:

Thread *K*
```
1. wait( mutex )
2. count++
3. post( mutex )
4. if count == n
5.     post( barrier )
6. end if
7. wait( barrier )
8. post( barrier )
```

This pattern of a wait followed immediately by a post is a common pattern called a *turnstile*. The analogy should be familiar to anyone who has travelled by subway (e.g., in Toronto or NYC): a turnstile allows one person at a time to go through. A turnstile pattern allows one thread at a time to proceed through, but can be locked to bar all threads from proceeding. Initially the turnstile in the above example is locked, and the nth thread unlocks it and permits all n threads to go through.

Alert readers may have noticed something that causes some minor distress: on line 4 in this solution (and the previous, suboptimal one), we are reading the value of `count`, a shared variable, without the protection of a semaphore. Is this dangerous? Yes, but the alternative is, in this specific instance, worse. Consider this instead:

Thread *K*
```
1. wait( mutex )
2. count++
3. if count == n
4.     post( barrier )
5. end if
6. wait( barrier )
7. post( barrier )
8. post( mutex )
```

The problem here is deadlock once again. The first thread waits on `mutex` and then goes to wait on the `barrier` semaphore. At this point, the first thread is blocked. When a second thread comes along, it will wait on `mutex` but can get no further because the first thread has not posted on it. The counter will be 1, but cannot get past 1. The condition of `count` equalling n can never be true. Thus, all the threads are stuck. This is a common source of deadlock: blocking on a semaphore while inside a critical region.

The Reusable Barrier

The barrier solution we have is good (as long as we can live with the read of the `count` variable outside of a mutual exclusion protection), but the way it is implemented now, `count` can increase but never decrease. Once the barrier is open, it can never be closed again. Programs very often do the same thing repeatedly, so a one-time use barrier is not ideal; it would be better to have a reusable barrier. One way to deal with this is to decrement `count` after

the rendezvous has taken place. The line labelled "critical point" is the section of code that must wait until all the threads have rendezvoused[4].

Thread *K*

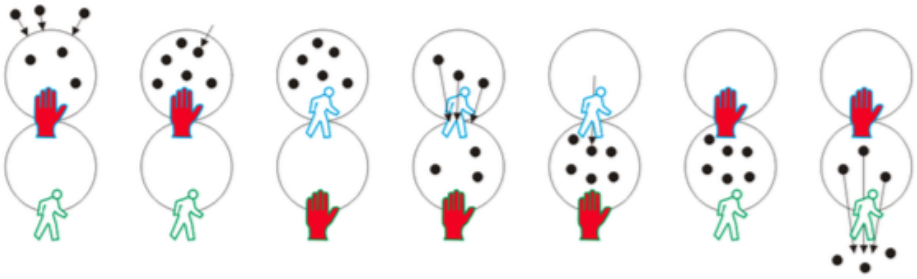
```
 1. wait( mutex )
 2. count++
 3. post( mutex )
 4. if count == n
 5.     post( turnstile )
 6. end if
 7. wait( turnstile )
 8. post( turnstile )
 9. [critical point]
10. wait( mutex )
11. count--
12. post( mutex )
13. if count == 0
14.     wait( turnstile )
15. end if
```

There are two problems with the above implementation. Suppose thread $n-1$ is about to execute line 4 (the comparison of `count`) and then there is a process switch and the nth thread comes to this point. Both of them will find that `count` is equal to n and therefore both threads will post the turnstile. The same problem occurs on line 13: more than one thread may wait on the turnstile, resulting in deadlock. Let us fix that:

Thread *K*

```
 1. wait( mutex )
 2. count++
 3. if count == n
 4.     post( turnstile )
 5. end if
 6. post( mutex )
 7. wait( turnstile )
 8. post( turnstile )
 9. [critical point]
10. wait( mutex )
11. count--
12. if count == 0
13.     wait( turnstile )
14. end if
15. post( mutex )
```

This solves the problem previously identified by putting the checks of `count` inside the critical section controlled by `mutex`. Unfortunately, there is another problem that can occur here: suppose one particular thread gets through the second mutex but is running in a loop and gets back through the first mutex again. This would be like one thread being one "lap" ahead of the others. We can prevent this by having two turnstiles: first all threads wait at the first turnstile until the last gets there and lets them through. Then all threads wait at a second turnstile until the last gets there and lets them all through again. The form of the solution represented visually, is:



A visual representation of the group rendezvous with two turnstiles [HZMG15].

[4] In the immortal words of Bill Watterson, author of *Calvin and Hobbes*: "verbing weirds language."

From left to right, the seven steps of the diagram are [HZMG15]:

1. The threads arrive at the rendezvous.
2. The nth thread arrives at the rendezvous.
3. That last thread unlocks the first turnstile and locks the second turnstile.
4. The threads pass through the first turnstile.
5. The nth thread passes through the first turnstile; it may not have been the last to arrive at the first turnstile.
6. The thread that just arrived locks the first turnstile and unlocks the second.
7. The threads leave the rendezvous.

Thread *K*

```
 1. wait( mutex )
 2. count++
 3. if count == n
 4.     wait( turnstile2 )
 5.     post( turnstile1 )
 6. end if
 7. post( mutex )
 8. wait( turnstile1 )
 9. post( turnstile1 )
10. [critical point]
11. wait( mutex )
12. count--
13. if count == 0
14.     wait( turnstile1 )
15.     post( turnstile2 )
16. end if
17. post( mutex )
18. wait( turnstile2 )
19. post( turnstile2 )
```

This solution can also be called a *two-phase barrier* because all threads have to wait twice: once at each turnstile.

15 — The Producer-Consumer Problem

Classical Synchronization Problems: Producer-Consumer

Various operating systems textbooks provide a few "classical problems": some scenarios that are phrased in real-world terms but meant to be an analogy for a problem that operating systems will deal with. These standard or classic problems are used to test any newly-proposed synchronization or coordination scheme. The solutions make use of semaphores as the basis for mutual exclusion. We are going to examine three of them in-depth: the producer-consumer problem, the readers-writers problem, and the dining philosophers problem.

The most common synchronization problem is the producer-consumer problem, also sometimes called the bounded-buffer-problem. Two processes share a common buffer that is of fixed size. One process is the producer: it generates data and puts it in the buffer. The other is the consumer: it takes data out of the buffer. This problem can be generalized to have p producers and c consumers, but for the sake of keeping the explanation simple, for now we will have just one of each [Tan08].

There are a couple of rules to be aware of. It is not possible to write into a buffer that is already full; if the buffer has capacity N and there are currently N items in it, the producer cannot write into the buffer and must wait until there is space. It is similarly not possible to read from an empty buffer; if the buffer has zero elements in it, the consumer cannot read from the buffer and must wait until there is something in there.

To keep track of the number of items in the buffer, we will have some variable `count`. This is a variable shared between more than one thread, and therefore access to this should be controlled with mutual exclusion. Let us assume the maximum number of elements in the buffer is defined as `BUFFER_SIZE`.

If busy-waiting is permitted, that is, we do not care if we are wasting CPU time, we can get away with one mutex, which we can call `mutex`. Each of the producer and consumer threads very likely run in an infinite loop on their own, but the code below is the sufficient to explain one iteration.

Producer

```
 1. [produce item]
 2. added = false
 3. while added is false
 4.    wait( mutex )
 5.    if count < BUFFER_SIZE
 6.        [add item to buffer]
 7.        count++
 8.        added = true
 9.    end if
10.    post( mutex )
11. end while
```

Consumer

```
 1. removed = false
 2. while removed is false
 3.    wait( mutex )
 4.    if count > 0
 5.        [remove item from buffer]
 6.        count--
 7.        removed = true
 8.    end if
 9.    post( mutex )
10. end while
11. [consume item]
```

While this accomplishes what we want, it is inefficient. Let's add a new rule that says we want to avoid busy-waiting. Thus, when the producer is waiting for space it will be blocked and just as the consumer will be when the consumer is waiting for an element. To accomplish this, we will need two general semaphores, each with maximum value of `BUFFER_SIZE`. The first is called `items`: it starts at 0 and represents how many spaces in the buffer are full. The second is the mirror image `spaces`; it starts at `BUFFER_SIZE` and represents the number of spaces in the buffer that are currently empty.

Producer

```
1. [produce item]
2. wait( spaces )
3. [add item to buffer]
4. post( items )
```

Consumer

```
1. wait( items )
2. [remove item from buffer]
3. post( spaces )
4. [consume item]
```

The producer can continue to produce items until the buffer is full and the consumer can continue to consume items until the buffer is empty. This solution works okay, given two assumptions: (1) that the actions of adding an item to the buffer and removing an item from the buffer add to and remove from the "next" space; and (2) that there is exactly one producer and one consumer in the system. If we have two producers, for example, they might be trying to write into the same space at the same time, and this would be a problem.

To generalize this solution to allow multiple producers and multiple consumers, what we need to do is add another binary semaphore, `mutex` (initialized to 1), effectively combining the previous solution with the one before it:

Producer

```
1. [produce item]
2. wait( spaces )
3. wait( mutex )
4. [add item to buffer]
5. post( mutex )
6. post( items )
```

Consumer

```
1. wait( items )
2. wait( mutex )
3. [remove item from buffer]
4. post( mutex )
5. post( spaces )
6. [consume item]
```

This situation should be setting off some alarm bells in your mind. In the synchronization patterns examined earlier, we mentioned the possibility of deadlock: all threads getting stuck. The hint that we might have a problem is one `wait` statement inside another. Unfortunately, seeing this pattern is not necessarily a guarantee that deadlock is going to happen (that would be too easy). This is, however, a sign that we need to analyze the code to determine if there is a problem.

Reading through the psuedocode above, you should be able to reason that this solution will not get stuck. You may choose a strategy along the lines of "proof by contradiction" and try to come up with a scenario that leads to deadlock. If you are unable to find one, then you may have a suitable solution (though it might be best to have someone else check to be sure). This is not a substitute for a formal mathematical proof, but the logic in your analysis should be convincing. Consider an alternate solution:

Producer

```
1. [produce item]
2. wait( mutex )
3. wait( spaces )
4. [add item to buffer]
5. post( items )
6. post( mutex )
```

Consumer

```
1. wait( mutex )
2. wait( items )
3. [remove item from buffer]
4. post( spaces )
5. post( mutex )
6. [consume item]
```

This solution is very much like the one we are certain works, except we have swapped the order of the `wait` statements. As before, we need to analyze this code to determine if there is a problem. This solution does have the deadlock problem. Imagine at the start of execution, when the buffer is empty, the consumer thread runs first. It will wait on `mutex`, be allowed to proceed, and then will be blocked on `items` because the buffer is initially empty. The thread is blocked. When the producer thread runs, it waits on `mutex` and cannot proceed because the consumer thread is in the critical section there. So the producer is blocked and can never produce any items. Thus, we have deadlock. This situation could occur any time the buffer is empty.

If the above pseudocode were implemented it is not a certainty that there will be a deadlock every time. In fact, the code will probably work fine most of the time. Once, however, we have found one scenario that can lead to deadlock, there is no need to look for other failure cases; we can write off this solution and replace it with a better one.

But let's get to doing an actual example! We will take some time to analyze this solution and understand how we got from the psuedocode above to the actual code below.

Producer-Consumer Example

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

#define BUFFER_SIZE 20
sem_t spaces;
sem_t items;
int counter = 0;
int* buffer;

int produce() {
  ++counter;
  return counter;
}

void consume( int value ) {
  printf("Consumed %d.\n", value);
}

void* producer( void* arg ) {
  int pindex = 0;
  while( counter < 10000 ) {
    int v = produce();
    sem_wait( &spaces );
    buffer[pindex] = v;
    pindex = (pindex + 1) % BUFFER_SIZE;
    sem_post( &items );
  }
  pthread_exit( NULL );
}

void* consumer( void* arg ) {
  int cindex = 0;
  int ctotal = 0;
  while( ctotal < 10000 ) {
    sem_wait( &items );
    int temp = buffer[cindex];
    buffer[cindex] = -1;
    cindex = (cindex + 1) % BUFFER_SIZE;
    sem_post( &spaces );
    consume( temp );
    ++ctotal;
  }
  pthread_exit( NULL );
}

int main( int argc, char** argv ) {
  buffer = malloc( BUFFER_SIZE * sizeof( int ) );
  for ( int i = 0; i < BUFFER_SIZE; i++ ) {
    buffer[i] = -1;
  }
  sem_init( &spaces, 0, BUFFER_SIZE );
  sem_init( &items, 0, 0 );

  pthread_t prod;
  pthread_t con;

  pthread_create( &prod, NULL, producer, NULL );
  pthread_create( &con, NULL, consumer, NULL );
  pthread_join( prod, NULL );
  pthread_join( con, NULL );

  free( buffer );
  sem_destroy( &spaces );
  sem_destroy( &items );
  pthread_exit( 0 );
}
```

Mutex Syntax

Before we go on to the the next code example, we should take a moment to learn about the syntax of the pthread mutex. While it is possible, of course, to use a semaphore as a mutex, frequently we will use the more specialized tool for this task.

The structure representing the mutex is of type `pthread_mutex_t`. We don't care about the internals or what the struct is made of; it is either locked or unlocked and that's all that matters to us.

```
pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes )
pthread_mutex_lock( pthread_mutex_t *mutex )
pthread_mutex_trylock( pthread_mutex_t *mutex ) /* Returns 0 on success */
pthread_mutex_unlock( pthread_mutex_t *mutex )
pthread_mutex_destroy( pthread_mutex_t *mutex )
```

The first function of note is `pthread_mutex_init` which is used to create a new mutex variable and returns it, with type `pthread_mutex_t`. It takes an optional parameter, the attributes (the details of which are not important at the moment, but relate mostly to priorities). We can initialize it using `NULL` and that is sufficient. There is also a syntactic shortcut to do static initialization if you do not want to set attributes [Bar14]:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

When created, by default, the mutex is unlocked. There are three methods related to using the mutex; two to lock it and one to unlock it, all of which take as a parameter the mutex to (un)lock. The unlock method, `pthread_mutex_unlock` is self-explanatory. As expected, attempting to unlock a mutex that is not currently locked is an error, but it is also an error if one thread attempts to unlock a mutex owned by another thread [Bar14].

The two kinds of lock are `pthread_mutex_lock`, which is blocking, and `pthread_mutex_trylock`, which is nonblocking. The lock function works as you would expect: if the mutex is currently locked, the calling function is blocked until its turn to enter the critical section; if the mutex is unlocked then it changes to being locked and the current thread enters the critical section. Trylock is more complicated and not necessary for understanding the producer-consumer example, but will come up again soon when we look at another classical synchronization problem.

To destroy a mutex, there is a method `pthread_mutex_destroy`. As expected, it cleans up a mutex and should be used when finished with it. If attributes were created with `pthread_mutexattr_init` they should be destroyed with `pthread_mutexattr_destroy`.

An attempt to destroy the mutex may fail if the mutex is currently locked. The specification says that destroying an unlocked mutex is okay, but attempting to destroy a locked one results in undefined behaviour. Undefined behaviour is, in the words of the internet, the worst thing ever: it means code might work some of the time or on some systems, but not others, or could work fine for a while and then break suddenly later when something else is changed[5].

Parallelizing the Producer-Consumer Solution

Now suppose that we wanted to have ten producers and ten consumers. How do we get there from here?

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

#define BUFFER_SIZE 100
int buffer[BUFFER_SIZE];
int pindex = 0;
int cindex = 0;
sem_t spaces;
sem_t items;
```

[5] Sadly, the specifications for C and POSIX and many other things are riddled with these "undefined behaviour" situations and it causes programmers everywhere a great deal of stress and difficulty. Another example: reading from an uninitialized variable in C produces undefined behaviour too.

```
pthread_mutex_t mutex;

int produce( int id ) {
  int r = rand();
  printf("Producer %d produced %d.\n", id, r);
  return r;
}

void consume( int id, int number ) {
  printf("Consumer %d consumed %d.\n", id, number);
}

void* producer( void* arg ) {
  int* id = (int*) arg;
  for(int i = 0; i < 10000; ++i) {
    int num = produce(*id);
    sem_wait( &spaces );
    pthread_mutex_lock( &mutex );
    buffer[pindex] = num;
    pindex = (pindex + 1) % BUFFER_SIZE;
    pthread_mutex_unlock( &mutex );
    sem_post( &items );
  }
  free( arg );
  pthread_exit( NULL );
}

void* consumer( void* arg ) {
  int* id = (int*) arg;
  for(int i = 0; i < 10000; ++i) {
    sem_wait( &items );
    pthread_mutex_lock( &mutex );
    int num = buffer[cindex];
    buffer[cindex] = -1;
    cindex = (cindex + 1) % BUFFER_SIZE;
    pthread_mutex_unlock( &mutex );
    sem_post( &spaces );
    consume( *id, num );
  }
  free( id );
  pthread_exit( NULL );
}

int main( int argc, char** argv ) {
  sem_init( &spaces, 0, BUFFER_SIZE );
  sem_init( &items, 0, 0 );
  pthread_mutex_init( &mutex, NULL );

  pthread_t threads[20];

  for( int i = 0; i < 10; i++ ) {
    int* id = malloc(sizeof(int));
    *id = i;
    pthread_create(&threads[i], NULL, producer, id);
  }
  for( int j = 10; j < 20; j++ ) {
    int* jd = malloc(sizeof(int));
    *jd = j-10;
    pthread_create(&threads[j], NULL, consumer, jd);
  }
  for( int k = 0; k < 20; k++ ){
    pthread_join(threads[k], NULL);
  }
  sem_destroy( &spaces );
  sem_destroy( &items );
  pthread_mutex_destroy( &mutex );
  pthread_exit( 0 );
}
```

16 — The Readers-Writers Problem

The Readers-Writers Problem

This problem is about concurrent reading and modification of a data structure or record by more than one thread. A writer will modify the data; a reader will read it only without modification. Unlike the producer-consumer problem, some concurrency is allowed:

1. Any number of readers may be in the critical section simultaneously.
2. Only one writer may be in the critical section (and when it is, no readers are allowed).

Or, to sum that up, a writer cannot enter the critical section while any other thread (whether reader or writer) is there. While a writer is in the critical section, neither readers nor writers may enter the critical section [Dow08]. This is very often how file systems work: a file may be read concurrently by any number of threads, but only one thread may write to it at a time (and to prevent reading of inconsistent data, no thread may read during the write).

This is similar to, but distinct from, the general mutual exclusion problem and the producer-consumer problem. In the readers-writers problem, readers do not modify the data (consumers do take things out of the buffer, modifying it). If any thread could read or write the shared data structure, we would have to use the general mutual exclusion solution. Although the general mutual exclusion routine would work in that it would prevent errors, it is a serious performance reduction versus allowing multiple readers concurrently [Sta14]. Thus, this situation is worth examining in its own right.

Let us keep track of the number of readers at any given time with a variable `readers`. We will need a way of protecting this variable from concurrent modifications, so there will have to be a binary semaphore `mutex`. We will also need one further semaphore, `roomEmpty`, as a way of indicating that the room is empty. A writer has to wait for the room to be empty (i.e., `wait` on the `roomEmpty` semaphore) before it can enter. The solution comes from [Dow08]:

Writer

```
1. wait( roomEmpty )
2. [write data]
3. post( roomEmpty )
```

Reader

```
 1. wait( mutex )
 2. readers++
 3. if readers == 1
 4.     wait( roomEmpty )
 5. end if
 6. post( mutex )
 7. [read data]
 8. wait( mutex )
 9. readers--
10. if readers == 0
11.     post( roomEmpty )
12. end if
13. post( mutex )
```

The code for the writer is much simpler than that of the readers. The writer may only enter into the critical section if the room is empty. When it has finished, it indicates that the room is empty. The writer can be certain that

when it exits the critical section that there are no other threads in the room, because no thread may enter the room while the writer was there.

The reader code is somewhat more complicated. The first reader that arrives encounters the situation that the room is empty, so it "locks" the room (waiting on the `roomEmpty` semaphore), and that will prevent writers from entering the room. Additional readers do not check if the room is empty; they just proceed to enter. When the last reader leaves the room, it signals that the room is empty ("unlocking it" to allow a writer in). This pattern is sometimes called the *light switch*, as in [HZMG15]: the first one into the room turns on the lights and the last one out turns them off again.

The reader code has that situation that makes us concerned about the possibility of deadlock: a wait on `roomEmpty` inside a critical section controlled by `mutex`. With a bit of reasoning, we can convince ourselves that there is no risk: the only situation in which a thread waits on `roomEmpty` is that a writer is currently in its critical section. No other reader thread can get the `mutex` lock. As long as the write operation takes finite time, eventually the writer will post the `roomEmpty` semaphore and the threads can continue. Deadlock is not a problem.

There is, however, a second problem that we need to be concerned about. Suppose some readers are in the room, and a writer arrives. The writer must wait until all the readers have left the room. When each of the readers is finished, it exits the room. In the meantime, more readers arrive and enter the room. So even though each reader is in the room for only a finite amount of time, there is never a moment when the room has no readers in it. This undesirable situation is not deadlock, because the reader threads are not stuck, but the writer (and any subsequent writers) is (are) going to wait forever. This is a situation called *starvation*: a thread never gets a chance to run.

Recall criterion 3 of the list of properties we want in any mutual exclusion solution: it must not be possible for a thread requiring access to the critical section to be delayed indefinitely. This problem is just as bad as deadlock in that if it is discovered, it eliminates a proposed solution as an acceptable option, even though starvation might only be an unlikely event. We must therefore improve on this solution such that there is no longer the possibility that a writer starves.

Conceptually, the solution that accomplishes the goal looks something like this: when a writer arrives, any readers currently reading should be permitted to finish their read, but no new readers should be allowed to start reading. Thus, eventually, all the readers currently in the critical section will finish, the writer will get a turn, because the room is empty, and when the writer is done, all the readers that arrived after the writer will be able to enter.

A new binary semaphore is needed here, called `turnstile`.

Writer

```
 1. wait( turnstile )
 2. wait( roomEmpty )
 3. [write data]
 4. post( turnstile )
 5. post( roomEmpty )
```

Reader

```
 1. wait( turnstile )
 2. post( turnstile )
 3. wait( mutex )
 4. readers++
 5. if readers == 1
 6.     wait( roomEmpty )
 7. end if
 8. post( mutex )
 9. [read data]
10. wait( mutex )
11. readers--
12. if readers == 0
13.     post( roomEmpty )
14. end if
15. post( mutex )
```

As before, the writer code is simpler, so we will examine it first. When the writer arrives, it will wait on the turnstile. If it is not the first writer, subsequent writers will queue up there, but if proceeds then it will wait for the room to be empty. Because the writer has locked the turnstile, no new readers can enter. There may be an arbitrary number of readers currently in the room, but each is there for only a finite amount of time. These existing-readers will be allowed to finish and leave the room. Then the writer gets a turn. When the writer is done, it posts the turnstile, which might unblock a reader or another writer.

Readers first get to the turnstile, and if they find it is locked, a writer is in its critical section. Thus, readers will queue at the turnstile if necessary, otherwise proceed. After that, the code is the same as we saw before: keep track of the number of readers and post if the room is empty or not empty.

Does this solution satisfy our goals of avoidance of deadlock and starvation? Starvation is fairly easy to assess: the first attempt at the solution had one scenario leading to starvation and this solution addresses it. Problem solved. You should be able to convince yourself that the solution as described cannot starve the writers or readers.

On to deadlock: the reader code is minimally changed from before; we have identified the turnstile code as not being a problem on its own (though its interactions with other threads need to be examined). The more dangerous block of code is on the side of the writer, because it has that pattern: two waits. If the writer is blocked on the `roomEmpty` semaphore, no readers or writers could advance past the turnstile and no writers. If the writer is blocked on that semaphore, it means there are readers in the room, and the readers will individually finish and leave (their progress is not impeded). Given that, the room will eventually become empty and the writer will be unblocked.

Note that this solution does not give writers any particular priority: when a writer exits it posts `turnstile` and that may unblock a reader or a writer. If it unblocks a reader, a whole bunch of readers may enter before the next writer is unblocked and locks the turnstile again. That may or may not be desirable, depending on the application. In any event, it does mean it is possible for readers to proceed even if a writer is queued. If there is a need to give writers priority, there are several techniques for doing so [Dow08].

Let's modify the solution so that writers have priority over readers. Giving writers priority may potentially cause readers to starve, but you may ignore this. We will probably want to break up the `roomEmpty` semaphore into `noReaders` and `noWriters`. A reader in the critical section should hold the `noReaders` semaphore and a writer should hold `noWriters` and `noReaders`.

Writer

```
 1. wait( writeMutex )
 2. writers++
 3. if writers == 1
 4.     wait( noReaders )
 5. end if
 6. post( writeMutex )
 7. wait ( noWriters )
 8. [write data]
 9. post( noWriters )
10. wait( writeMutex )
11. writers--
12. if writers == 0
13.    post( noReaders )
14. end if
15. post( writeMutex )
```

Reader

```
 1. wait( noReaders )
 2. wait( readMutex )
 3. readers++
 4. if readers == 1
 5.    wait( noWriters )
 6. end if
 7. post( readMutex )
 8. post( noReaders )
 9. [read data]
10. wait( readMutex )
11. readers--
12. if readers == 0
13.     post( noWriters )
14. end if
15. post( readMutex )
```

Yikes! The complexity for the writer increased dramatically. The reader is not all that different than it was before, however, and the writer now is to some extent the mirror image of the reader.

Readers Writers Example

Using the pseudocode as above, we can implement the readers-writers behaviour in a given program using only semaphores (optionally using a mutex to replace a semaphore). But in the pthread library there is support for readers-writers lock types, meaning we don't have to reinvent the wheel. We can probably well imagine the actual complexity of the previous solution, so it's our hope that the readers-writers lock would make our code simpler to read and harder to make mistakes.

The type for the lock is `pthread_rwlock_t`. It is analogous, obviously, to the mutex type `pthread_mutex_t`. Let's consider the functions that we have:

```
pthread_rwlock_init( pthread_rwlock_t * rwlock, pthread_rwlockattr_t * attr )
```

```
pthread_rwlock_rdlock( pthread_rwlock_t * rwlock )
pthread_rwlock_tryrdlock( pthread_rwlock_t * rwlock )
pthread_rwlock_wrlock( pthread_rwlock_t * rwlock )
pthread_rwlock_trywrlock( pthread_rwlock_t  *  rwlock )
pthread_rwlock_unlock( pthread_rwlock_t * rwlock )
pthread_rwlock_destroy( pthread_rwlock_t * rwlock )
```

In general our syntax very much resembles that of the mutex (attribute initialization and destruction not shown but they do exist). There are some small noteworthy differences, other than obviously the different type of the structure passed. Whereas before we had functions for lock and trylock, we now have those split into readlock and writelock (each of which has its own trylock function). As before, we will return to the subject of how trylock works soon.

In theory, the same thread may lock the same rwlock n times; just remember to unlock it n times as well.

And speaking of unlock, there's no specifying whether you are releasing a read or write lock. This is because it is unnecessary; the implementation unlocks whatever type the calling thread was holding. Much like `close()`, if we can figure out what we're closing we don't need the caller of the function to specify what to do.

As for whether readers or writers get priority, the specification says this is implementation defined. If possible, for threads of equal priority, a writer takes precedence over a reader. But your system may vary.

Consider the following example of the simple readers-writers (without writer priority and with risk of starvation) using the "old" way:

```
int readers;
pthread_mutex_t mutex;
sem_t roomEmpty;

void init( ) {
  readers = 0;
  pthread_mutex_init( &mutex, NULL );
  sem_init( &roomEmpty, 0, 1 );
}

void cleanup( ) {
  pthread_mutex_destroy( &mutex );
  sem_destroy( &roomEmpty );
}
```

```
void* writer( void* arg ) {
  sem_wait( &roomEmpty );
  write_data( arg );
  sem_post( &roomEmpty );
}

void* reader( void* read ) {
  pthread_mutex_lock( &mutex );
  readers++;
  if ( readers == 1 ) {
    sem_wait( &roomEmpty );
  }
  pthread_mutex_unlock( &mutex );
  read_data( arg );
  pthread_mutex_lock( &mutex );
  readers--;
  if ( readers == 0 ) {
    sem_post( &roomEmpty );
  }
  pthread_mutex_unlock( &mutex );
}
```

Now see it as it would be with the use of a rwlock!

```
pthread_rwlock_t rwlock;

void init( ) {
  pthread_rwlock_init( &rwlock, NULL );
}

void cleanup( ) {
  pthread_rwlock_destroy( &rwlock );
}
```

```
void* writer( void* arg ) {
  pthread_rwlock_wrlock( &rwlock );
  write_data( arg );
  pthread_rwlock_unlock( &rwlock );
}

void* reader( void* read ) {
  pthread_rwlock_rdlock( &rwlock );
  read_data( arg );
  pthread_rwlock_unlock( &rwlock );
}
```

Seek and Destroy: the Search-Insert-Delete Problem

This is an extension of the readers-writers problem called the search-insert-delete problem. Instead of two types of thread, reader and writer, there are three types of thread: searchers, inserters, deleters. They operate on a shared linked list of data.

Searchers merely examine the list; hence they can execute concurrently with each other. Searcher threads must call `void search( void* target )` where the argument to the searcher thread is the element to be found. These most closely resemble readers in the readers-writers problem.

Inserters add new items to the end of the list; only one insertion may take place at a time. However, one insert can proceed in parallel with any number of searches. Inserter threads call `node* find_insert_loc()` to find where to do the insertion; then `void insert( void* to_insert, node* after )` where the arguments are the location and the element to be inserted. Assume `insert` is written so the insertion can be done in parallel with the searches. Inserters resemble readers with an additional rule that only one of them can manipulate the list at a time.

Deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and when the deleter is accessing the list, no inserters and no searchers may be accessing the list. Deleter threads call `void delete( void* to_delete )` where the argument to the deleter thread is the element to be deleted. These most closely resemble writers in the readers-writers problem.

It turns out we don't need to modify things too much to allow for this third kind of thread. We need to keep track of when there are "no inserters" and "no searchers" (some hints for our semaphores) and another mutex to go around the actual insertion... See the code implementation below.

```
pthread_mutex_t searcher_mutex;
pthread_mutex_t inserter_mutex;
pthread_mutex_t perform_insert;
sem_t no_searchers;
sem_t no_inserters;
int searchers;
int inserters;


void init( ) {
  pthread_mutex_init( &searcher_mutex, NULL );
  pthread_mutex_init( &inserter_mutex, NULL );
  pthread_mutex_init( &perform_insert, NULL );
  sem_init( &no_inserters, 0, 1 );
  sem_init( &no_searchers, 0, 1 );
  searchers = 0;
  inserters = 0;
}

void* searcher_thread( void *target ) {
  pthread_mutex_lock( &searcher_mutex );
  searchers++;
  if ( searchers == 1 ) {
    sem_wait( &no_searchers );
  }
  pthread_mutex_unlock( &searcher_mutex );

  search( target );

  pthread_mutex_lock( &searcher_mutex );
  searchers--;
  if ( searchers == 0 ) {
    sem_post( &no_searchers );
  }
  pthread_mutex_unlock( &searcher_mutex );
}
```

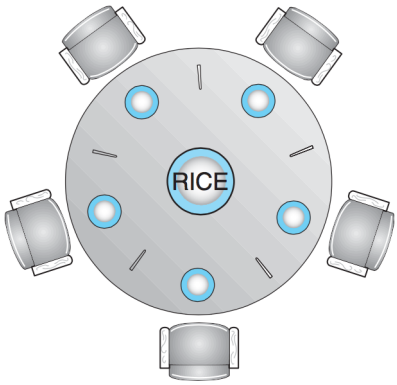
```
void* deleter_thread( void* to_delete ) {
  sem_wait( &no_searchers );
  sem_wait( &no_inserters );

  delete( to_delete );

  sem_post( &no_inserters );
  sem_post( &no_searchers );
}


void* inserter_thread( void *to_insert ) {
  pthread_mutex_lock( &inserter_mutex );
  inserters++;
  if ( inserters == 1 ) {
    sem_wait( &no_inserters );
  }
  pthread_mutex_unlock( &inserter_mutex );

  node * insert_after = find_insert_location( );
  pthread_mutex_lock( &perform_insert );
  insert( to_insert, insert_after ); /* Can update its
      position if needed */
  pthread_mutex_unlock( &perform_insert );

  pthread_mutex_lock( &inserter_mutex );
  inserters--;
  if ( inserters == 0 ) {
    sem_post( &no_inserters );
  }
  pthread_mutex_unlock( &inserter_mutex );
}
```

Question: could you implement this with a `pthread_rwlock_t` despite there being three kinds of thread?

17 — Deadlock

The Dining Philosophers Problem

The dining philosophers problem was also proposed by Dijkstra in 1965. The problem can be formulated with n philosophers, but problem is typically described as involving five philosophers. The description that follows is from [SGG13]. These five smart individuals spend their lives thinking, but every so often, they need to eat. They share a table, each having his or her own chair. In the centre of the table is a bowl of rice, and the table is laid with five single chopsticks. See the diagram below.



The situation of the dining philosophers [SGG13].

When a philosopher wishes to eat, she sits down at her designated chair, and attempts to pick up the two chopsticks that are nearest (one on the left, and one on the right). Philosophers are polite and therefore do not grab a chopstick out of the hands of a colleague. When a philosopher has both chopsticks, she may eat rice, and when she is finished, she puts down the chopsticks and goes back to thinking.

Some textbooks formulate this problem as philosophers needing two forks to eat rather than chopsticks. It is, of course, much easier to imagine having difficulty eating with only one chopstick than having difficulty with only one fork. Of course, the scenario is a little bit silly; we don't study it because it is supposed to be a true to life model of how real philosophers behave. The scenario is just a convenient and memorable example of a whole class of problems.

Suppose then that semaphores are the method for managing things. Because only one person can be in possession of a chopstick at a time, each chopstick may be represented by a binary semaphore. As a philosopher needs the chopstick to his left and right to eat, when the philosopher sits down he attempts to acquire the left chopstick, then the right, eats, and puts the chopsticks down. This works fine, until all philosophers sit down at the same time. Each grabs the chopstick to his or her left. None of them are able to acquire the chopstick to his or her right (because someone has already picked it up). None of the philosophers can eat; they are all stuck. This is deadlock.

This example makes it more clear why we call a situation where a thread never gets to run "starvation". If a philosopher is never able to get both chopsticks, that philosopher will never be able to eat, and though I am not an expert on biology, I have it on good authority that people who do not eat anything end up eventually starving to death. Even philosophers.

One thing that would guarantee that this problem does not occur is to protect the table with a binary semaphore. This would allow exactly one philosopher at a time to eat, but at the very least, deadlock and starvation would be avoided. Although this works, it is a suboptimal solution. There are five seats and five chopsticks and yet only one person is eating at a time. We can get better concurrency and use of the resources.

Next idea: what if we limit the number of philosophers at the table concurrently to four? The pigeonhole principle applies here: if there are k pigeonholes and more than k pigeons, at least one pigeonhole must have at least two pigeons. Thus, at least one of the four philosophers can get two chopsticks [HZMG15]. Implementing the solution is easy; we have a general semaphore with a maximum and initial value of 4.

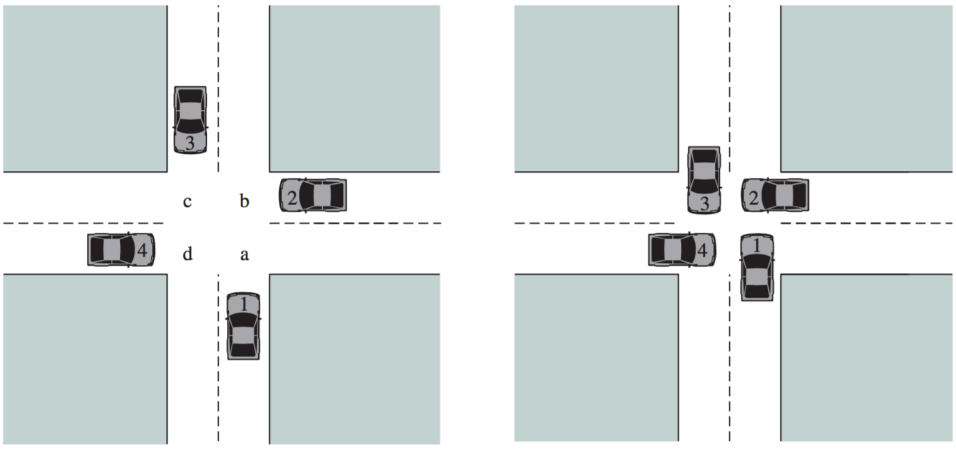
Another idea: the problem above occurs because every philosopher tries to pick up the left chopstick first. If some of them try to pick up the left and some pick up the right first, then deadlock will not happen, either [SGG13].

The dining philosophers problem is a good springboard from which to launch into a much deeper discussion of deadlock, and starvation.

Deadlock

We have already introduced the subject of deadlock and gave an informal definition as all processes being "stuck" (unable to proceed). A more formal definition is given in [Sta14]: "the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other". There is emphasis on permanent. It may be possible for all processes to be stuck temporarily, because one is waiting for some event (e.g., a read from disk), but this situation will resolve itself and is not deadlock. A set of processes is truly deadlocked when each process in the set is blocked on some event that can only be triggered by another blocked process in the set. In this case it is permanent, because none of the events can take place.

A deadlock involves some conflicting needs for resources by two or more processes. Consider a traffic deadlock. Suppose four cars arrive at a four-way stop intersection at the same time. We can divide the intersection into four quadrants, labelled, a, b, c, and d. To drive straight through the intersection, any one vehicle will need at least two of the quadrants (the two directly in front of it).



An illustration of deadlock involving cars [Sta14].

According to the Highway Traffic Act, whichever vehicle arrives at the stop sign first has the right of way. In the event that two vehicles arrive at the same time, a vehicle yields the right of way to the vehicle on its right. As long as three or fewer vehicles come to the stop sign at the same time, this works. If all four vehicles arrive at the same time, we have the potential for a problem. It is not a deadlock yet, because none of the processes are stuck yet, but it could happen. If all the drivers believe they should go first, we get the situation on the right, and we actually do have deadlock. This is very much like the dining philosophers problem; deadlock occurs if everyone tries to do the same thing at the same time.

Of course, for deadlock to occur, we do not have to have symmetric processes trying to do the same thing at

the same time. Given two semaphores, `a` and `b`, and two processes, we can have the following code that will sometimes, but not always lead to deadlock. If thread P locks `a` and then there is a process switch, and `b` is locked by Q, both threads will be stuck. Each has one resource the other needs, but they are both blocked and waiting for the other. It is obvious when we compare threads P and Q like this side-by-side, but in reality the problem is not usually that easy to see.

Thread P

```
1. wait( a )
2. wait( b )
3. [critical section]
4. post( a )
5. post( b )
```

Thread Q

```
1. wait( b )
2. wait( a )
3. [critical section]
4. post( b )
5. post( a )
```

Reusable and Consumable Resources

So, deadlock takes place when two processes or threads are competing for resources. We can generally classify a resource as either *reusable* or *consumable*. A reusable resource can be used by one process at a time, and is not depleted by that use. A process may lock the resource, make use of it, then release it such that other processes may acquire it. Processors, memory, files, and semaphores are all examples of reusable resources. If process P gets resource A and then releases it, process Q can acquire it. Thus, the example above involving P and Q is a deadlock involving reusable resources.

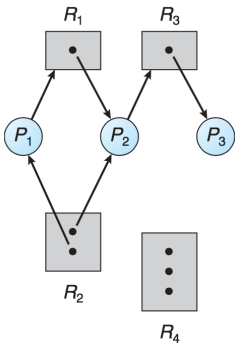
A consumable resource is one that is created and destroyed upon consumption. If the user presses the "Z" key on the keyboard, this generates an interrupt and produces the "Z" character in a buffer. A process that takes input will then consume that character (e.g., it goes into the `vi` editor window) and it is unavailable to other processes. Other things that are consumable resources: interrupts, posts, and messages. Deadlock is somewhat less likely when dealing with consumable resources, but in theory, all processes could be waiting to receive a message, if it is a blocking receive, and no process can send a message (because they are all waiting for some other process to send first).

Conditions for Deadlock

When a disaster happens, it is typically a result of a chain of things going wrong. If any one of those things did not happen, the disaster would be averted. This is referred to as "breaking the chain". There are four conditions for deadlock:

1. **Mutual Exclusion**: A resource belongs to, at most, one process at a time.
2. **Hold-and-Wait**: A process that is currently holding some resources may request additional resources and may be forced to wait for them.
3. **No Preemption**: A resource cannot be "taken" from the process that holds it; only the process currently holding that resource may release it.
4. **Circular-Wait**: A cycle in the resource allocation graph.

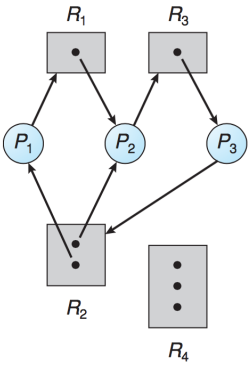
If the first three conditions are true, deadlock is possible, but deadlock will only happen if the fourth condition is fulfilled. But what is a resource allocation graph? It is a directed graph that tells us the state of the system by representing the processes, the resources, and which resources are held by which processes. Consider the example below:



A sample resource allocation graph [SGG13].

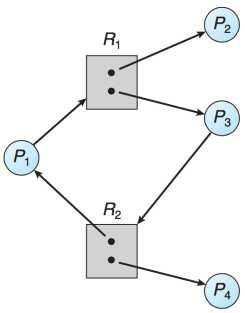
A process is represented by a circle labelled P_n. A resource is a box labelled R_n and contains one or more black circles, representing how many of that resource are available. A binary semaphore would therefore have one black circle; a general semaphore will have 1 or more. A directed edge from P_i to R_j is a request: a process requests that resource. A directed edge from R_j to P_i represents assignment - the process holds that resource. When a request is made, a request edge is inserted into the graph; if the request is fulfilled it is instantly turned into an assignment edge. When a resource is released, the assignment edge is deleted [SGG13].

If there are no cycles in the graph, then we can be certain that no process in the system is deadlocked. If a cycle exists, then some process may be deadlocked:



A sample resource allocation graph with a deadlock [SGG13].

But the presence of a cycle, on its own, is not necessarily certain that there is a deadlock:



A sample resource allocation graph with a cycle, but no deadlock [SGG13].

In this case, P_4 and P_2 are not part of the cycle and when they release their held resources, the other two processes will be able to proceed.

Dealing with Deadlock

There are four basic approaches to dealing with deadlock, each of which we will examine in turn.

1. Ignore it.
2. Deadlock prevention.
3. Deadlock avoidance.
4. Deadlock detection.

Deadlock Option 1: Ignore it

This option is certainly convenient: we simply pretend that deadlock can never happen, or if it does happen, it is someone else's fault. That said, this is the approach taken in Microsoft Windows, so it is a valid option. If two processes get deadlocked, the user may simply see no progress (or the "not responding" dialog) and have to open task manager and start killing processes. But the operating system designers can plausibly blame the problem on the program authors. Since we are the program authors we can't really pass the buck here. You can try to blame the users, but they'll tell you that your system shouldn't allow them to make mistakes in the first place. Well then.

Setting aside the "do-nothing" option, let us move on to some approaches that actually deal with the problem.

Deadlock Option 2: Deadlock Prevention

This approach is a way of preventing a deadlock from being possible. The first three conditions for deadlock (mutual exclusion, hold and wait, and no preemption) are all necessary for deadlock to be possible. If we eliminate one of these three pillars, deadlock is not possible and it is prevented from happening.

Mutual Exclusion. This pillar cannot, generally speaking, be disallowed. The purpose of mutual exclusion is to prevent errors like inconsistent state or crashes. Getting rid of mutual exclusion to rule out the possibility of deadlock is a cure that is worse than the disease. It is therefore not acceptable as a solution.

When it comes to your specific program, though, you might be able to make it so that mutual exclusion is not necessary. If you can give each thread, for example, its own copy of the data, or you can find an algorithm that no longer requires locking a given resource, then by all means, use it! Locking and other mutual exclusion constructs are things we use because we must, not because we like to.

Hold and Wait. To prevent the hold-and-wait condition, we must guarantee that when a process requests a resource, it does not have any other resource. In a practical sense this does not mean that things can be requested only one at a time; that would be like telling the philosophers that when they have one chopstick already, they cannot request another.

The operating system could operate under these rules, but we know that our usual commercial operating systems don't do this. We could imagine a world where things are different, but that's not very helpful when it comes to implementing the program we're trying to write. Even so, the idea of requesting everything up front is obviously not possible (at the time of opening a document you are not asked if you are going to use the printer sometime later, just in case), nor is giving up all resources before asking for more possible. A resource that cannot be easily released is memory, as released memory may be collected and reassigned by the operating system. Therefore we cannot release all resources, and releasing only some of the resources means that we cannot categorically rule out deadlock; we can only make it less likely to occur [HZMG15].

Another idea that might work is *two-phase locking*. A process attempts to lock a group of resources at once, and if it does not get everything it needs, it releases the locks it received and tries again. Thus a process does not wait while holding resources. If a philosopher picks up a chopstick but is unable to acquire a second, she puts down the chopstick she has picked up and tries again. Although she may not be the one to eat next, at least not all philosophers are stuck holding one chopstick.

Two phase locking is not applicable to our current model for semaphores, where there is no way to know the value of the semaphore and the operating system will block a process on a `wait` if some other thread is in the critical region. After the process is blocked on the semaphore, a second process will run, and the first process does not get the opportunity to release the resources it holds. If there exist requests for resources and mutual exclusion areas that return, perhaps, `true` if the resource is acquired and `false` if it is unavailable, this can work. Then the program is responsible for checking if any of the requests returned `false` and releasing any resources where the request returned `true`.

Fortunately, we know of some routines that do just this: the `trylock` functions that were mentioned earlier but not expanded upon:

```
int pthread_mutex_trylock( pthread_mutex_t * mutex )
int pthread_rwlock_tryrdlock( pthread_rwlock_t * rwlock )
int pthread_rwlock_trywrlock( pthread_rwlock_t  * rwlock )
```

These functions return an integer and it's extremely important to check and see if the return code is 0, because that is the only way to know if the lock was acquired. The call is non-blocking so the code will carry on regardless. Consider below a code description of the dining philosophers if they used two-phase locking via the trylock routines. Assume that the mutex variables have been initialized appropriately. It should be possible to reason about this solution and demonstrate that (1) a philosopher can only eat if they have both chopsticks, and (2) deadlock does not occur.

```
int locked_both = 0;
while( locked_both == 0 ) {
  int locked1 = pthread_mutex_trylock( chopstick1 );
  int locked2 = pthread_mutex_trylock( chopstick2 );
  if (locked1 != 0 && locked2 == 0) {
    pthread_mutex_unlock( chopstick2 );
  } else if (locked1 == 0 && locked2 != 0 ) {
    pthread_mutex_unlock( chopstick1 );
  } else if (locked1 != 0 && locked2 != 0 ) {
    /* Do nothing */
  } else {
    locked_both = 1;
  }
}
eat( );

pthread_mutex_unlock( chopstick1 );
pthread_mutex_unlock( chopstick2 );
```

There does exist a "try" function for the semaphore as well:

```
int sem_trywait( sem_t * sem );
```

If we would have been blocked at the call to `sem_trywait`, it returns -1 and `errno` is set to `EAGAIN`.

The idea of a process releasing resources that it is holding is the basis for the next solution: knocking down the "no preemption" pillar.

No Preemption. If we violate this condition, it means that we do have preemption: forcible removal of resources from a process, by the operating system. Suppose a process P_1 holds R_1 and R_2 and wants to get R_3, but R_3 is unavailable. P_1 will be blocked by the operating system. If another process P_2 comes by and requests R_1 and R_2, the resources R_1 and R_2 are taken away from P_1. The resources are added to the list of things that P_1 is waiting for (so it is waiting for all three now). In the meantime, P_2 can use them and continue. P_1 will be unblocked when all three resources are once again available for it to acquire [SGG13].

This isn't really something we can do as program designers, because only the operating system really has the power to carry out. It's included for completeness, but is not anything we can actually implement ourselves.

For preemption to work, however, the resource must be a resource of a type where the state can be saved and restored (e.g., the CPU with its registers). This is not applicable to all resources; if a printer is in use by P_1 it cannot be preempted and given to P_2, otherwise the printout will be a jumble. Thus, preemption is also not sufficient to prevent deadlock from ever transpiring, it once again only makes it less likely.

18 — Deadlock Avoidance

Dealing with Deadlock

Deadlock Option 3: Deadlock Avoidance

In examining deadlock prevention, we were attempting to rule deadlock out categorically by eliminating one of the three preconditions for deadlock to be possible. If successful, then we can be sure that deadlock does not occur. Unfortunately, eliminating the pillars came with some conditions and the best we could accomplish was merely making deadlock less likely and not resolving the problem in its entirety. Thus, we are forced to live with a system where a deadlock is possible. However, deadlock being possible is not the same thing as deadlock being inevitable; we can take steps to avoid it if there is a danger of it actually happening. The basic strategy is: do not allow a cycle in the resource allocation graph.

In the dining philosophers problem, one strategy to prevent the deadlock was limiting the number of concurrently-eating philosophers to four, even though the table has five seats. This was a way of preventing a cycle in the resource allocation graph: with only four philosophers and five chopsticks, there were insufficient requests to complete a cycle. This solution is suitable, but not necessarily generalizable to all deadlock situations.

Ordering of Resources

A general strategy to prevent a cycle from forming is to impose ordering on resource requests. Resources are given some order and requests must follow that order. This idea may have occurred seemed obvious from this example:

Thread P

```
1. wait( a )
2. wait( b )
3. [critical section]
4. post( a )
5. post( b )
```

Thread Q

```
1. wait( b )
2. wait( a )
3. [critical section]
4. post( b )
5. post( a )
```

Thread Q requests `b` first and then `a`, while P does the reverse. The deadlock would not take place if both threads requested these two resources in the same order, whether `a` then `b` or `b` then `a`. Of course, when they have names like this, a natural ordering (alphabetical, or perhaps reverse alphabetical) is obvious.

To generalize and formalize this principle, if the set of all resources in the system is $R = \{R_0, R_1, R_2, ...R_m\}$, we assign to each resource R_k a unique integer value. Let us define this function as $f(R_i)$, that maps a resource to an integer value. This integer value is used to compare two resources: if a process has been assigned resource R_i, that process may request R_j only if $f(R_j) > f(R_i)$. Note that this is a strictly greater-than relationship; if the process needs more than one of R_i then the request for all of these must be made at once (in a single request). To get R_i when already in possession of a resource R_j where $f(R_j) > f(R_i)$, the process must release any resources R_k where $f(R_k) \geq f(R_i)$. If these two protocols are followed, then a circular-wait condition cannot hold [SGG13].

In the dining philosophers problem, we assign each of the five chopsticks a number from 0 to 4. Each philosopher must then request them in ascending order. The first philosopher requests chopstick 0, on her left, and then chopstick 1, on her right. The second requests chopstick 1 and then chopstick 2. This continues until the last

philosopher who would previously have requested chopstick 4 and then 0, but under the new rules, this is forbidden. This philosopher must instead request 0 on his right, and then 4, on his left. This last philosopher will be blocked when trying to acquire chopstick 0 and it means chopstick 4 will be available for the second-to-last philosopher. Thus, deadlock is avoided.

In [SGG13] is a proof that ordering the resources prevents deadlock. The approach is proof by contradiction: assume a circular wait is present. Let the set of processes in the circular wait be $\{P_0, P_1, ...P_n\}$ and the set of resources be $\{R_0, R_1, ...R_n\}$. The cycle is formed as: P_i is waiting for resource R_i and that resource is held by P_{i+1}. The exception is the case of P_n, which is waiting for resource R_n that is held by P_0 (completing the cycle by wrapping around). Since Process P_{i+1} holds resource R_i while requesting R_{i+1}, this means $f(R_i) < f(R_{i+1})$ for all i. But this means that $f(R_0) < f(R_1) < ... < f(R_n) < f(R_0)$. It cannot be the case that $f(R_0) < f(R_0)$: a contradiction, meaning a circular wait cannot occur.

In development this is usually enforced just by coding convention and code review. If you say that mutexes must always be acquired in alphabetical order (or their order in some file), if everyone sticks to that there will be no issue. Sometimes it's not that simple, though, because one mutex can be pointed to by two pointers... which name is the correct one?

Stay Alert, Stay Safe[6]

Suppose that instead of ordering the resources, each process will need to give the operating system some additional information about what resources might be requested. Processes need to say in advance of execution what is the maximum number of resources of each type they might conceivably need. In a system with a tape drive and printer, perhaps process A needs the tape drive first, then the printer, and process B needs the printer and then the tape drive. With this knowledge, the system can make more intelligent decisions about when to run a process or make it wait, to avoid getting into a deadlock [SGG13].

We say a state is *safe* if there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum resources immediately [Tan08]. Hence why we needed to know in advance the maximum resources that could be required by the process.

More formally, there must exist a *safe sequence*: a sequence of processes $< P_1, P_2, ...P_n$ is a safe sequence in the current allocation state if, for each P_i the resource requests that P_i can still make can be satisfied by the currently available resources plus resources held by P_j where $j < i$. If a resource P_i needs is not currently available, P_i can wait until all P_j have finished and releases its resources. When P_i terminates, P_{i+1} can obtain its needed resources and continue [SGG13].

Any state that is not safe is considered *unsafe*. If the system is in a safe state, then there is no deadlock. Being in an unsafe state does not mean that there is a deadlock, but it means a deadlock is possible. The analysis we do is the worst case scenario: every process immediately requests the maximum resources it could ever use. Perhaps the processes do not make those requests in reality.

Consider an example from [Tan08] in which there are three processes, A, B, and C. Assume there is only one resource, and a maximum of 10 instances exist. Suppose A has 3 resources but may request up to 9, B has 2 and may request up to 4, C has 2 and may request up to 7. There are 3 resources currently free. How do we determine that the state described below is safe?

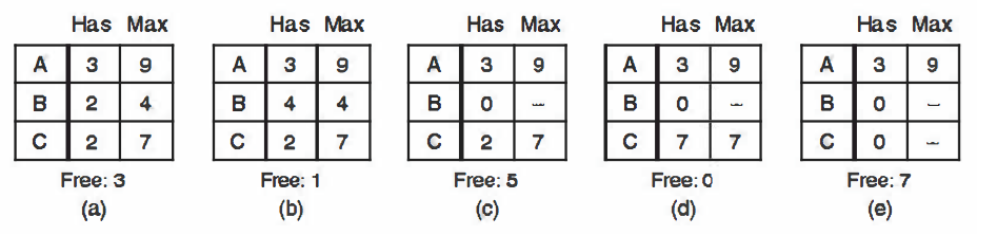
	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

A diagram representing the resource allocation state of the system [Tan08].

Recall that for a state to be safe we need only one path that allows all processes to complete. Multiple solutions may exist, and there may be paths that lead to deadlock. But we need only one solution, such as:

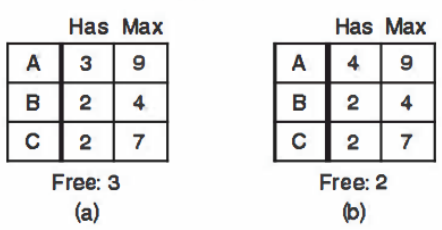
[6] If this reference makes no sense to you, it's because I'm old: https://www.youtube.com/watch?v=cgvv4wnVlFU



A diagram demonstrating the state in (a) is safe [Tan08].

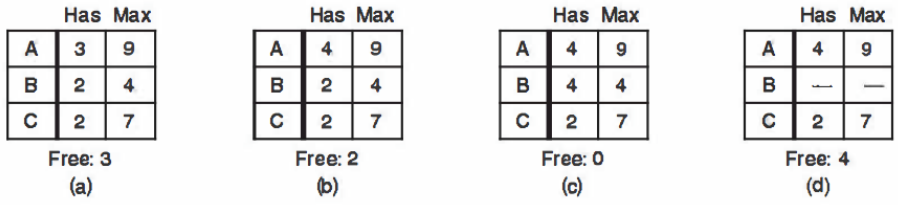
The part of the diagram labelled (a) is the initial state of the system. Imagine that process B is allowed to run exclusively, until it gets two more resources, so it now has 4 resources and there is one free, as diagram (b) depicts. B is allowed to run to completion, and it therefore finishes and releases its resources to the free pool, shown in (c). Then process C is scheduled and gets its full complement of resources, consuming all the elements in the free pool, as in (d). C runs exclusively until completion and it releases its resources, shown in (e). Finally, A is able to get all the resources and run to completion, too. Thus, we proved that the state is safe: there is a way the system can avoid a deadlock, and run all processes to completion.

Suppose, however, A requests and gets another resource. In that case, the initial condition has changed so that A has 4 resources and there are 2 free resources. Or, in the diagram below, the state changes from (a) to (b).



A diagram representing the updated resource allocation state of the system [Tan08].

Given this new initial state (b), we need to evaluate if it is safe.



A diagram demonstrating the state of [Tan08].

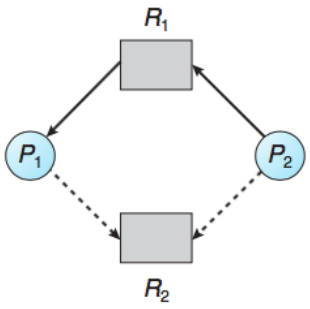
Let us start by attempting to run A - there are 3 free resources and A would need 6 to continue (because a process immediately takes its full resources when it runs), so it cannot run to completion. The same is true for C - it needs 5 additional resources when 3 are available. B, however, can continue: it needs 2 resources when 3 are available. If we attempt to run B exclusively, it gets its full resources, shown in part (c), and then will terminate and release its resources, as in (d). At this point, however, there is no way to continue; neither A nor C can get the full complement of resources it might need because there are only 4 free resources. We are out of options; there is no path that guarantees all processes run to completion. Thus, the state is unsafe.

Remember that just because the state is unsafe, does not mean that deadlock is present or certain. The analysis is worst-case: C may never need all 7 of the resource and may run to completion with its current 2. However, in an unsafe state, we cannot be certain that deadlock will not occur.

Resource-Allocation-Graph Algorithm. The fourth condition for deadlock is modelled, typically as being a resource allocation graph with a cycle in it. Idea: let us use that idea to avoid deadlock by having the operating

system maintain a resource allocation graph. This is applicable if there is only one instance of each resource and still requires that all the resources that a process will require must be declared in advance. However, this condition does not have to be strictly adhered to, if the system allows additional requests only when none of the process's requests have yet been made.

The model for resource allocation graphs earlier had two kinds of edges: one representing requests (a process requests a resource) and one representing allocation (a resource currently belongs to a process). We will require a new type of edge in the graph: a *claim* edge, as a way of indicating a process may request a resource at some time in the future, and it is drawn with a dashed line. When the process actually makes the request for the resource, a claim edge is converted to a request edge, and upon release the assignment edge reverts to a claim edge [SGG13].



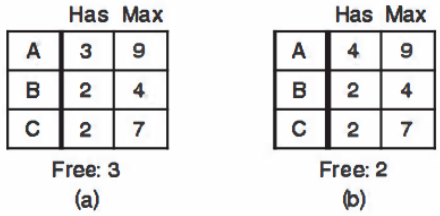
A resource allocation graph showing claim edges [SGG13].

A resource request will only be granted if converting the request edge to an assignment edge will not result in a cycle in the graph. There are graph algorithms in code that can be used to detect a cycle (and you may have studied them in a data structures and algorithms class). If no cycle is found, then allocation of the resource will not move the system into an unsafe state. If a cycle is found, the request should not be granted, as it risks a deadlock.

As mentioned, this is applicable only if all resource requests are known in advance and there is only one instance of each resource.

The Banker's Algorithm. The banker's algorithm is more general: it allows for resources with multiple instances. It received this name because it is hypothetically an algorithm that a small town banker might follow if he or she were trying to prevent allocating the cash on hand in such away that he or she could no longer satisfy customers[7]. Banks typically lend out more money than they have on hand on the (usually-correct) theory that not everyone will come asking for all their deposits all at once[8].

In fact, the analysis we did earlier to determine if a state is safe or unsafe, is the foundation of the banker's algorithm. Recall this diagram from earlier:



A diagram representing the updated resource allocation state of the system [Tan08].

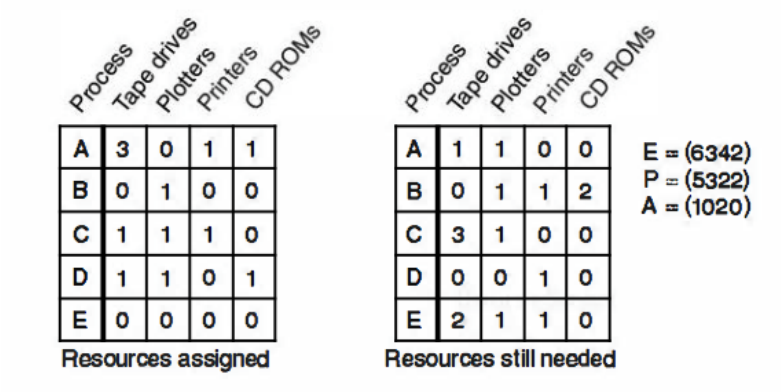
Granting the request from process A, the transition from (a) to (b), moved the system from a safe state to an unsafe state. The operating system, when it receives a resource request, will evaluate the new state to see if it would transition the system to an unsafe state. If it would result in that transition, the request will be denied or

[7] Although the financial crisis of 2008 might convince you that the banker's algorithm is "do whatever makes you rich and when you run out of money, the government will bail you out", this is not the algorithm we are studying here.

[8] Not to digress too much to the subject of banking, but if it IS the case that everyone asks for their money all at once, this is called a "run" on the bank and is generally considered a disaster. Central banks step in with more money to fix this situation.

A will be blocked until the request can be fulfilled without putting the system in an unsafe state. Holding to this condition means deadlock will be avoided.

The banker's algorithm can accommodate multiple resources, as shown in the diagram below.



The banker's algorithm with multiple resources [Tan08].

The left matrix shows the current state of assigned resources and the matrix on the right shows the maximum number of resources that a process might need to complete execution. At the far right of the diagram is a series of vectors: E for the existing resources, P for the resources some process currently possesses, and A for the available instances of that resource.

The informal approach for checking if a state is safe is described by [Tan08]:

1. Look for a row in the matrix, r, where the unmet resource needs are less than or equal to the available resources in A. If no such row exists, the system state is unsafe.
2. Assume the process from r gets all the resources it needs. Mark that process as terminated and put all its resources into A, the available pool.
3. Repeat steps 1 and 2 until either: (i) all processes are marked terminated and the initial state was safe; or (ii) no process remains whose needs can be met and the initial state is unsafe.

If more than one process may be chosen in step 1, it does not matter which we choose: the pool of available resources will either stay the same or get larger; it cannot shrink.

To determine if granting a resource will make the system unsafe, the operating system should simply perform a what-if calculation. Assume the resource is granted and do the safe state calculation given that new state. If the result is that the state is unsafe, the request should be deferred or denied.

A formal, mathematical description of the banker's algorithm is defined in [SGG13].

As great as the banker's algorithm is in theory, in practice it is utterly useless, as highlighted by [Tan08]. Processes rarely know in advance what their maximum resource needs will be. Also, the number of processes is not fixed, but varies as users log in and launch and close programs and log out. Finally, a resource that was thought to be available can suddenly vanish (a peripheral device is disconnected, the network goes down, a printer breaks...). Thus in practice, the banker's algorithm can almost never be used.

19 — Deadlock Detection & Recovery

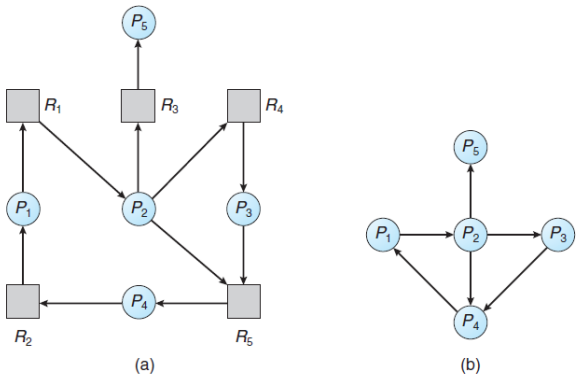
Deadlock Detection

Thus far we have examined several ways to prevent or avoid deadlock, but all of those solutions have come with significant drawbacks or limitations. Avoidance analyses are also conservative; they will prevent a request from taking place if there is even a small chance it could lead to a deadlock. If we cannot stop deadlock from happening or cannot live with the performance reduction that avoidance mandates, then perhaps the next best thing is to let all resource requests proceed and then determine later if a deadlock exists, and if so, do something about it.

The truth is that the operating system is in the best position to detect if there is a deadlock, but commercial operating systems don't do that. We can do it ourselves in our program using some of the following techniques but we almost certainly won't have the full picture. This is because we cannot see what's going on inside other processes or inside the kernel. But we will do our best under the circumstances, if we need to.

Most programs realistically do not and cannot do this; however, database servers are a (frequently used, in this course anyway) potential example of a program that does check whether there is a deadlock and takes action to resolve it.

The basic strategy for deadlock detection is like the deadlock avoidance strategy in that it relies on a model of the resource allocation and requests. If resources have only a single instance, we may reduce the graph to a simplified version called the *wait-for* graph. This removes the resource boxes from the diagram and indicates that a process P_i is waiting for process P_j rather than for a resource R_k that happens to be held by P_j. An edge $P_i \rightarrow P_j$ exists in the wait-for graph if and only if the resource allocation graph has a request $P_i \rightarrow R_k$ and an assignment edge $R_k \rightarrow P_j$ [SGG13]. Consider the example below:



(a) A resource allocation graph and (b) its corresponding wait-for graph [SGG13].

Given the wait-for graph, it is trivial for humans to look at this and determine if there is a cycle, but for the computer it takes slightly more work. We must execute an algorithm to determine if there is a cycle. A cycle exists in the wait for graph if and only if a deadlock exists in the system. Such cycle detection algorithms tend to have runtime characteristics of $\Theta(n^2)$ where n is the number of nodes in the graph. Though not a formal proof and probably not acceptable to write on a data structures and algorithms examination, the premise of the algorithm is: for each node n in the graph, examine each possible path from that node. If a node is reached from which no

further path is available, examine the next path. If node n is reached on the current path, a cycle is detected and the algorithm terminates.

General Deadlock Detection Algorithm

We will use the general deadlock detection algorithm from [Tan08] that allows for multiple resources of each type. In this algorithm, there are n processes numbered P_1 through P_n and m resources. Resources are represented by two vectors: E, the existing resource vector – the total number of instances of each resource; and A, the available resource vector – how many instances of each resource are currently available (not assigned to a process). If resource i has two instances total and one is currently assigned to a process, E_i is 2 and A_i is 1.

We need two matrices to represent the current situation of the system. The first is C, the current allocation; it contains data about what resources are currently assigned to each process. Thus, row i of C shows how many of each resource P_i has. The second matrix is R, the request matrix. Row i of this matrix shows how many of each resource P_i wants. Thus, C_{ij} shows the number of instances of resource j that P_i has and R_{ij} shows the number of instances of resource j that P_i wants. Or, to show what those look like:

Resources in Existence

$$[E_1, E_2, ..., E_m]$$

Resources Available

$$[A_1, A_2, ..., A_m]$$

Current Allocations

$$\begin{bmatrix} C_{11} & C_{12} & ... & C_{1m} \\ C_{21} & C_{22} & ... & C_{2m} \\ \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & ... & C_{nm} \end{bmatrix}$$

Requests

$$\begin{bmatrix} R_{11} & R_{12} & ... & R_{1m} \\ R_{21} & R_{22} & ... & R_{2m} \\ \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & ... & R_{nm} \end{bmatrix}$$

Note that at all times a resource is considered either allocated or available. This means this mathematical relationship always holds:

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j.$$

There is one more bit of setup before we are ready to run the algorithm. The key idea is comparison of vectors, so let us define for notational convenience, the idea of "less than" for two vectors. We will say that for two vectors A and B of length m, $A \leq B$ means that $A_i \leq B_i$ for all i from 1 to m.

At last, the algorithm. The starting condition is that all processes are unmarked and the vectors and matrices described above are populated. The algorithm will go through all processes and determine if they can complete, under worst-case conditions (keeping all resources until termination), and if they can, marks them. At the end, any processes that are not marked are deadlocked.

1. Search for an unmarked process whose requests can all be satisfied with the available resources in A. Mathematically: find a process P_i such that row $R_i \leq A$.
2. If a process is found, add the allocated resources of that process to the available vector and mark the process. Mathematically: $A = A + C_i$. Go back to step 1.
3. If no process was found in the search of step 1, the algorithm terminates.

This approach is similar to the banker's algorithm. Step one looks for a process that can run to completion, and we are certain it will be able to do so because the currently available resources equal or exceed its needs. That process can finish, and when it does so, its currently-held resources are released and available for another process to acquire. Step two reflects this by adding the resources it holds to the available set. Then another process is selected. At the end, either all processes can finish and there is no deadlock, or there is a set of processes (at least two) that are deadlocked.

This algorithm has a runtime performance characteristic of $\Theta(m \times n^2)$.

Deadlock Detection Algorithm Example. Let's now use this algorithm on an example also from [Tan08]. Let us assume we have 4 types of resources and 3 processes. The names and types of the resources do not matter - they can be anything - all that matters are the numbers. Each process has some resources currently allocated to it, and each process has some requests outstanding. Thus, the initial state of the system is:

$$E = [4, 2, 3, 1] \qquad A = [2, 1, 0, 0]$$

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \qquad R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Now, carry out the algorithm. The first process cannot proceed because its request R_{14} cannot be met; there are no instances of resource 4 available. Process 2 cannot proceed either, because it needs resource 3 and there are none of those available either. Process 3 can run and will eventually return its resources, changing the available vector to $A = [2, 2, 2, 0]$. Mark process 3.

Process 1 still cannot run because its need for resource 4 still cannot be met. Process 2, however, can, and it will do so, returning all its resources. The available vector is changed to $A = [4, 2, 2, 1]$. Mark process 2.

Process 1 can now run. Return its resources: $A = [4, 2, 3, 1]$. Mark process 1.

There are no remaining unmarked processes in the system, and therefore no deadlock. As a sanity check, compare vector E and the final values for A – they should be the same if there is no deadlock.

The Assumption Algorithm

One other way we could detect deadlock is just by assuming it. If we have started some operation and it is taking too long, we can assume that a deadlock occurred and decide to take action. Obviously, it is possible that the operation is just taking incredibly long and is not stuck; however, user patience is limited.

This does mean we might declare a deadlock when there really is none. This is a tradeoff we may have to accept due to not having full information about the state of all threads/processes.

One way to implement this is via a watchdog timer. When starting something that might get deadlocked, you also begin a timer. If the task finishes first, then cancel the timer; if the timer fires, then deadlock is assumed and we take a resolution action.

When to Detect Deadlock

The runtime characteristic of the simple deadlock detection algorithm was identified as $\Theta(n^2)$, and the runtime characteristic of the general algorithm was shown as $\Theta(m \times n^2)$ where n is the number of processes and m is the number of resources in the system. This means that the deadlock detection routine is expensive to execute.

This prompts a question: how often should the deadlock detection algorithm be run? One strategy is to run it every time a resource is requested. Running the algorithm might be rather expensive, so perhaps this is too often. An obvious optimization: it should only run every time a resource request cannot be granted (that is, a process gets blocked). Another idea: run it periodically instead.

When to run the deadlock detection algorithm depends on how often we expect deadlock to occur, and how severe a problem it is when deadlock occurs. If deadlock happens a lot, checking for deadlock often will make sense. If the consequences of a deadlock are severe, it makes sense to check frequently to identify the problem as soon as possible.

Several sources including [Tan08] suggest running the deadlock detection algorithm when CPU utilization is low. This is not only because it would be a bad idea to run an algorithm that is time consuming and computationally expensive while the system is busy. When a deadlock is present in the system, many processes are stuck and cannot proceed, so a drop in CPU usage may be an indication that many processes are deadlocked.

Deadlock Recovery

Once a deadlock has been detected, a system can recover from that deadlock by "breaking" the deadlock. These are called recovery strategies and they are ways the system may automatically deal with the problem. It is possible to have a manual form of deadlock recovery, where an operator is notified and that person is responsible for sorting out the problem, but the manual method needs no further discussion.

Recovery Strategies

There are several strategies that we could apply, which we will refer to in humorous terms. All are valid solutions of various complexity. Unfortunately, none of the solutions are particularly pleasant. They can result in data loss, delays in completion of programs, or some other problems. Ideally we would like to break the deadlock with as little disruption as possible, so the strategies that rely on selection of one or more victims should choose carefully. These strategies are not mutually exclusive; a system may implement all of them.

Robbery

The strategy of "robbery" is just a humorous way of saying preemption. This is virtually identical to the discussion of knocking down the pillar of deadlock for deadlock prevention; the only difference is that recovery is run only when deadlock is detected. Suppose a process P_1 has a resource R_1 and needs R_2, while a process P_2 has R_2 and needs R_1. We may block process P_2 and take away R_2 from P_2 and allow P_1 to have it. After that resource becomes available again, it is returned to P_2. In the general case, the system will take resources from ("rob") some process(es) and give those resources to other processes until the deadlock cycle is broken. To do so, the system needs to choose a victim to rob; a subject we will examine shortly.

The resource should be an appropriate type to be preempted: it must be possible to save and restore the state. Preemption of a printer is not realistic, nor is memory. However, other resources, like a processor or access to the network, may be. If the state of the resource cannot be saved and restored, then preemption is not a good strategy.

Mass Murder

If a deadlock is detected, we may choose to terminate (kill) all the processes involved in the deadlock. This solution is surprisingly common. It is one way to be certain that the deadlock cycle is broken. The resources that these processes were holding will become available. There is no need to determine a victim: kill 'em all and let root sort them out.

This solution, while easy to implement, may not solve the problem, however, as the circumstances that caused the deadlock may occur again if all the processes are restarted. If the deadlock were an unlikely situation caused by "unlucky" timing then it will probably not recur, or at least, will not recur for some time.

If processes P_1 and P_2 are deadlocked, however, is it really necessary to kill both of them? If we killed only one, the other could proceed...

Murder

Perhaps instead of killing all processes involved in a deadlock, instead, we might kill processes selectively. Like preemption, selecting which process is the victim is important. When the victim is killed, its resources are added to the available set and this will hopefully allow other processes to proceed. At this point, the deadlock detection algorithm must run again to determine if a deadlock still exists; if not the problem is dealt with, but if so, this strategy needs to be repeated (select a new victim, kill it, free up its resources, and determine if deadlock is still present) again and again until the logjam is broken.

Time Travel

Time travel is just a clever way of saying *rollback*: returning the state of a process to a saved state from an earlier time. To do so, of course, there must be a saved state that was created in advance of a problem (otherwise there

is no state to roll back to...). The saved state is called a *checkpoint* and the act of creating and saving a checkpoint is called *checkpointing*[9]. Checkpoints may be created periodically or before beginning a particular operation that requires a lot of resources.

A checkpoint contains the memory image, including the call stack, and resource state of a process. It is written to disk and will usually persist as long as the process continues to execute.

You might already be somewhat familiar with the concept of rollbacks if you have used version control software like Subversion (`svn`) or Git (`git`). A previous state of the source code is saved and if a developer commits a change that is detrimental to the software, that change can easily be undone by reverting the state of the source files to an earlier version. Rollbacks are also common in databases; if an attempted modification of a database record fails for some reason, such as data being too long for a field, then the state of the database is restored to what it was before the attempted change.

Unfortunately, rollback does not always succeed. Sometimes moving the process back to an earlier state just moves it a few steps back on the same road that leads to the deadlock. Or a different ordering of events and resource requests may avoid the deadlock entirely. Rollback may be attempted a few times before giving up and trying another strategy.

Armageddon

Armageddon - the end of the world. If a deadlock has occurred, sometimes the best thing to do is reboot the system. This has a side effect of killing all processes, whether they are stuck or not, but is sometimes the best way to make sure that the system is in a valid state. NASA's Spirit rover, one of the missions to explore the red planet, relies on this strategy if it detects a deadlock. Like killing all affected processes, it is easy to implement, but is most disruptive.

Victim Selection

If we have to choose a victim process for one of the strategies, e.g., termination, then we need a strategy for which process to choose. We could choose randomly: kill a process and hope that that was enough. Sometimes this will work, because it breaks the deadlock and the other processes can all proceed. In general, however, making an informed decision is better.

Note that it is, strictly speaking, not necessary to choose one of the processes involved in the deadlock. For example, if P_1 and P_2 are deadlocked, a process P_3 may have an instance of a resource that P_1 needs, and killing P_3 will allow both P_1 and P_2 to proceed.

We can think of this as kind of an optimization problem. Define a cost function for choosing each process, evaluate the cost function, and then choose the lowest cost. Some factors to consider in selection of processes [SGG13]:

1. The priority of the process.
2. How long the process has been executing.
3. How long is remaining in execution, if known.
4. What resources the process has (number and type).
5. Future resource requests, if known.
6. Whether the process is user-interactive or in the background.
7. How many times, if any, the process has been selected as a victim.

These sorts of selection routines tend to favour older processes rather than younger ones. This is not because older processes vote in higher numbers, but because it tends to be more expensive to restart an older process. A process that has been running for a long time, if it is restarted, has to do a lot more work to get to the point

[9] Once again, verbing weirds language.

where it was terminated than a younger process. Another reason: if the oldest process were constantly the one selected, that process itself might never get to run to completion (starvation) because it is constantly killed before it finishes. A third reason is somewhat more subtle: if the killing process is very aggressive then perhaps no tasks run to completion because each process, shortly after becoming the oldest, is claimed by the deadlock recovery process. Therefore, young processes tend to be the ones selected.

The final element in the list, keeping track of how many times a process has been victimized, is also there to prevent starvation. The selection process very likely produces the same or similar results each time it is run, so it may happen that the same process is selected over and over again. It may be advisable to take the number of rollbacks or terminations into account so that no process in particular is starved.

Miscarriages of Justice The deadlock detection algorithm we have chosen tends to be conservative in that it will err on the side of saying that there is a deadlock. This is because the worst case is assumed: that processes take resources and keep them until the end of their execution. In practice, however, processes will release resources (or at least they should!) so we might detect a deadlock when there is none. We may also, then, kill an innocent process in a system that is not actually deadlocked. Oops!

It turns out that our deadlock detection algorithms do not have to be perfect if we have chosen an appropriate recovery strategy. If killing the process and restarting it does not have unexpected side effects (like asking the user the same question a second time), then being selected does not have an impact on the correctness of the program; just on how long it takes to execute. Suppose the process is a compiler. It reads the source files, processes that input, and produces the binary output file. If partway through, this task is terminated and has to start again, it takes slightly longer for the compile to finish, but there is no impact on the correctness of the binary.

Livelock

Usually we like to think that our recovery strategy is, well, effective, and means that (at least the surviving) threads become able to continue. Unfortunately, no; sometimes the action we have taken to resolve deadlock just results in ending up at the same impasse again soon thereafter. This was mentioned when talking about rollback as a strategy, but could happen regardless of the actions we take.

Deadlock recovery strategies are not the only way to end up in livelock, mind you. Imagine if the Dining Philosophers all used two-phase locking. Imagine each of them doing the same steps at the same time in perfect synchronicity: pick up a chopstick, fail to get the other chopstick, put down the first, go back to step one. They would all perform the same set of actions repeatedly, but would never get to the critical step: eat. Also, their perfect synchronicity would probably make them look like they were in a music video.

In real life, it is unlikely that n threads executing on a general purpose operating system would achieve this sort of perfect lock-step. Random fluctuations in timing or interrupts or just the scheduler doing its job would result in someone eventually being lucky enough to acquire both chopsticks and eat. However, as execution of tasks takes on longer timeframes (e.g., jobs that run for hours, not microseconds), deadlock may repeatedly occur, be detected, a recovery strategy runs, and we do the same things that lead to the same problem again.

One problem with livelock is that it is harder to detect than deadlock. In deadlock, threads are known to be blocked and analysis tools (e.g., a debugger) can tell you at what statement those threads are blocked. With livelock, it seems like the threads are running and working, but somehow no work is getting done. For this reason, a well-designed recovery strategies may have some sort of limit on the number of retries before concluding this won't work and either going to a different strategy or involving an operator.

20 — Memory

Main Memory

In executing a program, the CPU fetches instructions from memory, and decodes the instruction. It may be that the instruction requires fetching of operands from memory. After the operation is completed, a result may be stored back in memory. So a single simple instruction like an addition could easily result in four memory accesses. Executing a program therefore means spending a lot of time interacting with memory.

Like the CPU, main memory is a resource that needs to be shared between multiple processes. The way programs are written, application developers behave as if (1) main memory is unlimited, and (2) all of main memory is at the program's disposal. Simple logic tells us that application developers are wrong: an infinite amount of data storage would require an infinite amount of physical space. Memory space is limited to the physical amount of RAM in the machine, which is a function of how much money you spent when purchasing it. Even so, why is it that program developers pretend that memory is infinite and unshared when it is not?

Certainly compared to the early days of computing, the amount of memory available is huge. The Commodore 64, introduced in 1982, had a whopping 64 KB of memory[10]. A 4-byte (32-bit) integer was a significant fraction of memory, and application developers had to scrimp and save to avoid wasting even a single integer's worth of memory. This historical reason is why languages like C and Java support types like `short` even though you have probably never used them outside of a programming exercise or examination. In the meantime, memory has jumped up to 8 or 16 GB. Remember that 1 GB is 1024 MB and 1 MB is 1024 KB. Now think about the fact that we still use 32-bit integers. If a thousand integers were wasted unnecessarily, would anyone notice or care? This makes the problem better by "kicking the can down the road" – we can use a lot more memory before we are in danger of running out, but it's still possible to run out. Furthermore, even though memory might be big enough for every process to have its own area, that would not work if every developer assumes memory is unshared. So we still have not solved the mystery of why application developers can be oblivious to the realities of main memory.

The answer is that most modern operating systems manage the shared resource of memory for them. This was not always the case, and applications used to be responsible for managing all of memory. It was also not so long ago that there were various third party programs to let the user do some memory management, too. Around the time of the last versions of MS-DOS and Windows 95, there were products like QEMM 8 that you could use to move programs around in memory. But you're not here to hear old war stories about moving parts of Windows into high memory so that TIE Fighter would run. One of the major objectives of the operating system is to manage shared resources, and that is exactly what main memory is.

No Memory Management

The simplest way to manage memory is, well, not to manage memory at all. Early mainframe computers and even personal computers into the 1980s had no memory management strategy. Programs would just operate directly on memory addresses. Memory is viewed as a linear array with addresses starting at 0 and going up to some maximum limit (e.g., 65535) depending on the physical hardware of the machine. The section of memory that is program-accessible depended a lot on the operating system, if any, and other things needed (e.g., the BASIC compiler). So to write a program, we need to know the "start" address (the first free location after the OS, drivers, compiler and all that) and the "end" address, the last available address of memory. These would differ from machine to machine, making it that much harder to write a program that ran on different computers.

[10] And now you know why it was called the Commodore **64**.

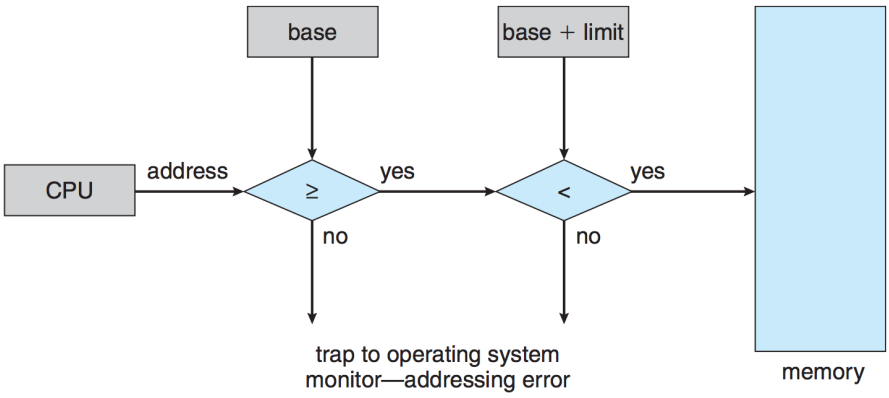
A program executed an instruction directly on a memory address, such as writing 385 into memory location 1024. Suppose you wanted to have two programs running at the same time. Immediately, a problem springs to mind: if the first program writes to address 1024, and the second program writes to address 1024, the second program overwrote the first program's changes and it will probably result in errors or a crash. Alternatively, if programs are aware of one another, the first program can use memory locations less than, say, 2048 and the second uses memory locations above 2048. This level of co-ordination gets more and more difficult as more and more programs are introduced to the system and is next to impossible if we do not control (have the source code to) all the programs that are to execute concurrently.

In theory, there is a solution: on every process switch, save the entire contents of memory to disk, and restore the memory contents of the next process to run. This kind of swapping is, to say the least, incredibly expensive – imagine swapping out several gigabytes of memory on every process switch – but the problem is avoided because only one process is ever in memory at a time.

Aside from the inefficiency, there is another problem: there is no protection for the operating system, either. The operating system is typically placed in either low memory (the start of addresses) or high memory (from the end of addresses), or in some cases, a bit of both. An errant memory access might result in overwriting a part of the OS in memory, which can not only lead to crashes, but could also result in corrupting important files on disk.

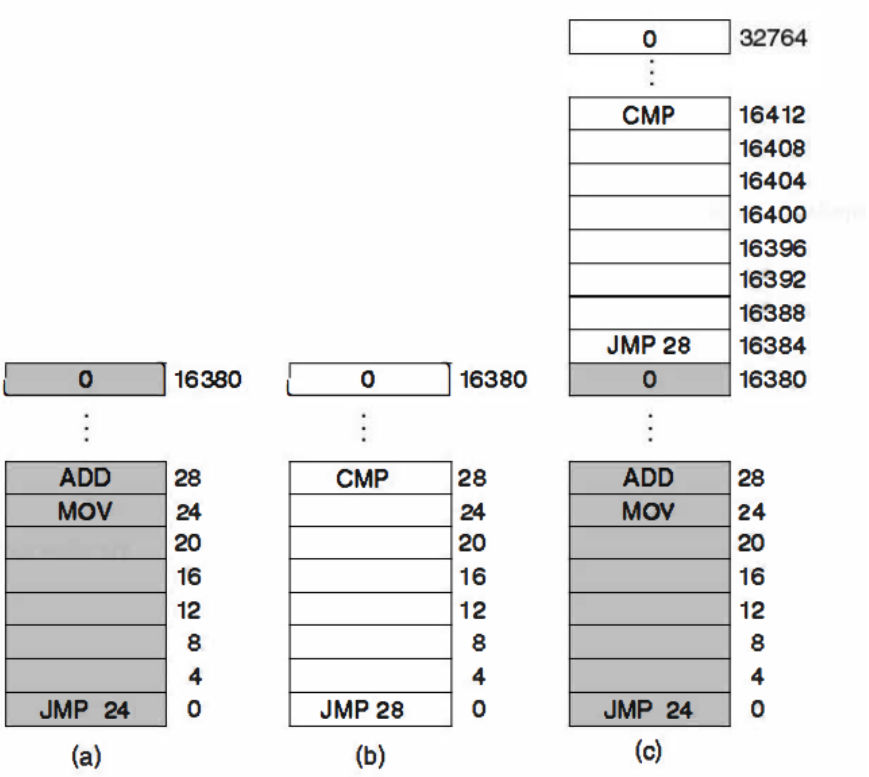
We can attempt to solve the problem of protection by keeping track of some additional information. The IBM 360 solved this problem by dividing memory into 2 KB blocks and each was assigned a 4-bit protection key, held in special registers in the CPU. The Program Status Word (PSW) also contained a 4 bit key. The 360 hardware would then identify as an error an attempt to access memory with a protection code different from the PSW key. And the operating system itself was the only software allowed to change the protection keys. Thus, no program could interfere with another or with the operating system [Tan08].

We can generalize this solution by having two values maintained: the *base* and *limit* addresses. These define the start and end addresses of the program's memory. Every memory access is then compared to the base address as well as the [base + limit] address. If an attempted memory access falls outside that acceptable range, this is an error. As this operation is likely to be executed approximately infinity times, to make the operation as fast as possible, the base and limit variables are usually registers and this comparison is done using hardware. The flow chart below describes the operation (keeping in mind that both comparisons can be done in parallel).



Hardware address protection with base and limit registers [SGG13].

This, unfortunately, does not solve the problem. Imagine we have two programs numbered simply 1 and 2, each 16 KB in size. Suppose then we will load them into memory in different consecutive areas, as in the figure below:



(a) Program 1. (b) Program 2. (c) Programs 1 and 2 loaded into memory consecutively. [Tan08].

Program 1 will execute as expected. The problem is immediately obvious when Program 2 runs. The instruction at address 16384 is executed: `JMP 28` takes execution to memory address 28 (an `ADD` instruction and not the expected address of 16412 and the `CMP` comparison). The problem is that both programs reference absolute physical locations.

The IBM 360's stopgap solution to this was to do static relocation: if a program was being loaded to a base address 16384 the constant 16384 was added to every program address during the load process. While slow, if every address is updated correctly, the program works [Tan08].

This is, unfortunately, not as easy as it sounds. A command like `JMP 28` must be relocated, but the 28 in a command like `ADD R1,28` (add 28 to register R1 and store the result in R1) is a constant and should not be changed. How do we know which is an address and which is a constant? It gets worse: in C a pointer contains an address, but addresses are just numbers, so in theory we could just dereference any integer variable and it would take us to a memory address (whether it's valid or not is not the point here). That makes it even harder to know if a number is just a number or an address.

When we are writing a program, unless it's in assembly, we do not usually refer to variables by their memory locations. The command we write looks something like `x = 5;` and although we know that variable `x` is stored in memory, the question arises: when is the variable assigned a location in memory? There are three obvious times to do it [SGG13]:

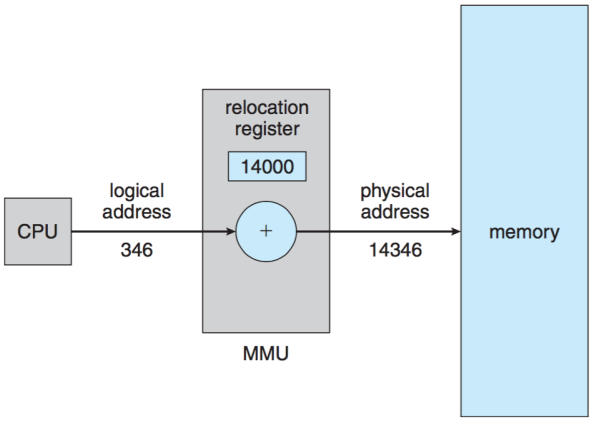
1. **Compile time**: the solution we saw first with commands like `JMP 28`. If we are certain where the process will be loaded into memory, at compile time we can convert those variables to address locations. This is what happens in assembly, and in the MS-DOS .COM format (like `command.com`).
2. **Load time**: the IBM 360 solution; at the time when the code is to be loaded into memory, the addresses are updated. This requires that the compiler indicate what numbers are addresses and should be updated when the program is loaded into memory.
3. **Execution time**: if programs can move around in memory during execution (something we have not yet examined), then we need to do the binding at run-time. For this to work, though, we will need help from the hardware developers...

Address Space

It is clear that having no memory management system leaves us with a number of problems in memory. What we would like to do is introduce an abstraction; a layer of indirection. We do this with a concept called *address space*. An address space is a set of addresses that a process can use; each process has its own address space, independent of other processes' address spaces (except when we create shared memory).

Telephone numbers in Canada and the USA take the form of NNN-NNNN, a seven digit number. In theory, any number in the range 000-0000 to 999-9999 could be issued, but in practice certain numbers are reserved (like the 000 or 555 prefixes). Given the number of telephones in the countries, seven digits could not possibly be enough (10 million numbers for a population around 350 million?!). In fact, many readers probably looked at this and thought it was wrong that phone numbers are seven digits; phone numbers are ten digits! Those three additional digits are the area code, after all, and they relate to a geographic area. The number 416-555-1234 is identifiable by its area code as being located in Toronto (or at least a cell phone registered there), and the number 212-555-1234 is in New York City. Although ten digit dialing is mandatory in Toronto (and presumably NYC), if you live in a district where ten digit dialing is not mandatory, you can dial 555-1234 and it will connect you to the number 555-1234 in your local area code. This is the idea we want to apply to memory: let each process have its own area code. So process can write to location 1024 and process 2 can write to location 1024 and these are two distinct locations, perhaps 21024 and 91024 respectively.

Now, instead of altering the addresses in memory, we will effectively prefix every memory access with an area code. The address that is generated by the CPU, e.g., the 28 in `JMP 28`, is the *logical address*. We then add the area code to it to produce the *physical address* (the actual location in memory and the address that it sent over the bus). In practice, to speed this up, it is done via some hardware, and the "area code" is a register called the *relocation register* as below:



Dynamic translation of logical addresses to physical addresses with a relocation register [SGG13].

The process itself does not know the physical address (14346 in the above example); it knows only the logical address (346). This is a run-time mapping of variables to memory. We get some protection between processes, though we would get more protection if we brought back the limit register and compared the physical address to the base and [base + limit] values again.

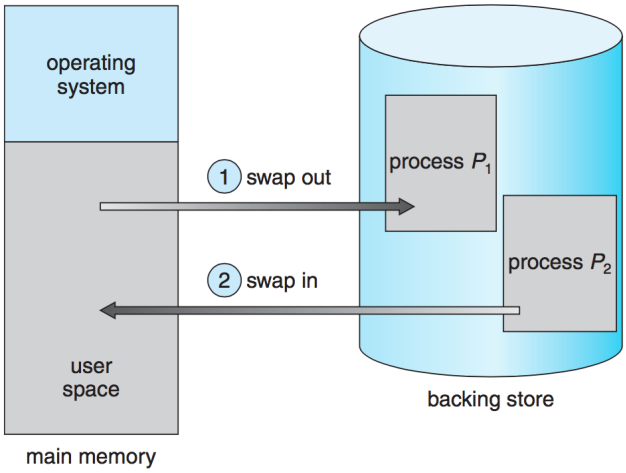
This scheme also gives us something new: we can relocate a process in memory if we change the relocation register's value accordingly. A process that is currently loaded into memory with a relocation register value of 14000 can easily be moved to another location. Copy all the memory from relocation register to the limit to a new location, such as 90000, and then update the relocation register to the new starting location (90000). After that, the old location of the process's memory can be marked as available or used by another process.

These benefits do not come for free. Every memory access now includes an addition (or two if the limit register comes into play). Comparisons are pretty quick for the CPU, but addition can be quite a bit slower, because of carry propagation time[11]. So every memory access has a penalty associated with it to do the addition of the relocation register value to the issued CPU address.

[11] If you are the sort of person who is really only interested in software and you have been wondering why the program has made you learn

Swapping

To run, a process must be in main memory. Given enough processes, or processes sufficiently demanding on the memory of the system, it will not be possible to keep all of them in memory at the same time. Processes that are blocked may be taking up space in memory and it might be logical to make room for processes that are ready to run by moving blocked processes out of memory. The process of moving a process from memory to disk or vice-versa is called *swapping*.



Swapping processes (1) from memory to disk and (2) from disk to memory [SGG13].

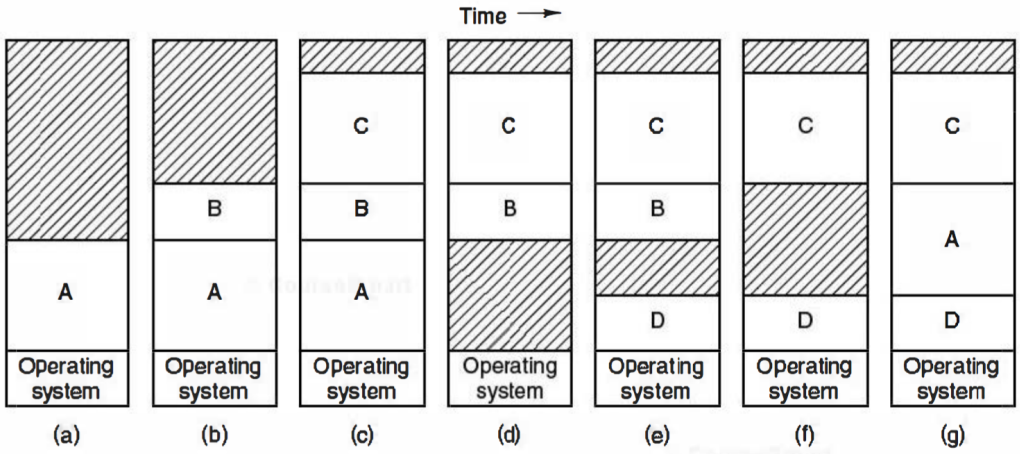
Unfortunately, swapping a process to disk is very painful. If the process is using 1 GB of memory, to swap a process out to disk, we need to write 1 GB of memory to disk. To load that process back later, it means reading another 1 GB from disk and putting it into main memory. If 1 GB strikes you as ridiculous in size, according to the Mac OS X system utilities, with five PDF documents (whose combined file size on disk is 80.6 MB) open, the "Preview" application is consuming 2.05 GB as I write this. So, swapping is something we would like to do as little as possible, but it will be necessary eventually.

Modern operating systems do not perform this kind of swapping because it is simply too slow. Too much time would be wasted swapping processes to and from disk. A modified form of swapping is used, but this is a subject we will return to later on in the examination of memory [SGG13].

When swapping a process back in from disk it is not necessary to put it back in exactly the same place as it originally was. This works because the relocation register will be updated with the new location of the process when it is moved back to memory. See the diagram below showing the state of memory after seven swap operations.

---

all sorts of details about hardware, this is a good example of why. You will have noticed in this section that the hardware developers have bailed the software developers out of various problems by taking operations that would be painfully slow and doing them a lot faster. In the case of the CPU addition carry propagation problem, if you don't understand the hardware, the software you write will be slow or problematic and you will not know why.



A view of memory over time, swapping processes in and out as needed [Tan08].

In (a), the only process in memory is A. In (b), process B is added and in (c) process C starts and is loaded into memory. When process D would like to run, there is insufficient free space for it, so a process will need to be swapped out. In (d), process A is chosen by the OS and is swapped out so that in (e) process D may be in memory. If A is ready to run again, space must be made for it, so B is swapped out in (f) and then when A is loaded the state of memory is as shown in (g).

Thus far we have considered process memory as a large, fixed-sized block. A process gets a big section of memory and operates in that area. As you know from previous programming experience, use of the `new` keyword in some languages, or `malloc()` in C, will result in dynamic memory allocation. We will have to deal with this next.

21 — Dynamic Memory Allocation

Dynamic Memory Allocation

By now you must surely be familiar with dynamic memory allocation from the perspective of the application developer. To create a new instance of an object in Java, for example, you use the `new` keyword and the runtime will come and garbage collect it when it is no longer needed. In C++ we have the `new` and `delete` operators to allocate and deallocate memory. The `new` and `delete` operators invoke the constructor and destructor, respectively. C works on memory at a lower level: to allocate a block of memory in C, there is `malloc()` and when finished, you return it with `free()`. This level is a lot closer to the way the operating system thinks about memory: just tell me how much you need and tell me when you are finished with it.

If we generalize this interface, we get two signatures [HZMG15]:

`void *allocate_memory( int size )`
Allocate a block of `size` bytes of memory; return a pointer to the address of the first byte.

`void deallocate_memory( void *mem_block )`
Return the allocated block of memory at the address `mem_block` to the pool of free memory.

This should square nicely with your experience of using `malloc()` and `free()` in C. To allocate an integer, you call `malloc( sizeof( int ) )`. This creates, somewhere in memory, a new integer and returns its address, which can be stored in a pointer (presumably an integer pointer, but you can store it in a void pointer too). To be sure to ask for the correct amount of memory, we have `sizeof` which works out the size of its argument (integer) and then the size of an integer, say, 4 bytes, is supplied to `malloc()`, so 4 bytes are allocated.

When you `free()` that pointer, all that happens is that the memory is marked as available, which is why you can sometimes get away with dereferencing a pointer after it has been freed. Sometimes it takes a while for that memory to be reclaimed or reused so the old value just happens to still be there in memory. Note that `free()` does not specify how much memory is being returned. This means two things: (1) that the operating system is keeping track of each allocated block's size, and (2) that it is not possible to return part of a block.

With the preliminaries about memory allocation out of the way, now it is time to turn our attention to fulfilling the memory allocation requests that we receive. As we will see, this is not a trivial problem. The operating system will try to find some free memory to meet the request. Although running out of memory is a rare thing given the size of main memory in a modern computer, there is still the possibility that some request may not be fulfilled because no block meeting that need is available.

Fixed Block Sizes

One possibility for how to allocate memory is in fixed block sizes. All blocks of memory allocated are the same size. This does not mean that requests are not of varying size, it just means that all blocks allocated are the same size. If a request comes in for 1 byte, 1 block is allocated. If a request comes in that is, say, 1.5 blocks, 2 blocks are allocated.

It is immediately obvious when we look at this that some memory is "wasted". If 1.5 blocks are requested and 2 blocks are allocated and returned, we are using up an extra 0.5 blocks. This space cannot be used for anything

useful (as it shows as allocated). This is a problem called *internal fragmentation* – unused memory that is internal to a partition. This is obviously going to occur often when fixed block sizes are used, and the bigger each block is, the more memory will be wasted in internal fragmentation.

One Size of Blocks. Suppose the system has only one size of blocks, perhaps, 1 KB. To implement this strategy, divide up memory into blocks of this fixed size and maintain a linked list of addresses of all currently available blocks. When a block is allocated, remove its corresponding node from the linked list; when a block is freed, put a node with that address into the linked list. If the list is empty, a memory request cannot be satisfied, and null will be returned. This is definitely fast as we can allocate memory in $\Theta(1)$ time [HZMG15].

Fixed Block Sizes, Multiple Size Options. Recognizing that some memory allocation requests are bigger than others, it might make sense to have several different block sizes; perhaps 1 KB, 2 KB, and 4 KB. These can generally be allocated and deallocated in $\Theta(1)$ time if we have one linked list for each different size of block [HZMG15].

Unfortunately, fixed block sizes suffer from a lot of internal fragmentation. This may be suitable for embedded systems where simplicity and speed of operations are more important than worrying about wasting memory. It is obvious from working with languages like C that this is not how `malloc()` works: 1 KB of memory is not allocated to store a 4-byte integer. What we need instead is a variable block size.

Variable Block Sizes

To a certain extent, variable block sizes are not that different from fixed block sizes; we just take the size of blocks down to the smallest they can be. In a typical system with byte-addressable memory, in a way, the smallest block is one byte. Now we have a different problem: keeping track of what is allocated and what is free.

Bitmaps. It is possible to divide memory into M units of n bits, and then to create a bit array of size M storing the status of each of those units. If a bit m in M is 0, it means that unit is unallocated; if it is 1 then that unit is allocated. How much memory is lost to this overhead? $100/(n+1)$% of the memory is used. If a unit is 4 bytes, the bitmap is about 3% of memory; if it is 16 bytes the bitmap takes about 0.8% of memory. Finding a block of k bytes requires searching the bitmap for a run of $\frac{8k}{n}$ zeros [HZMG15].

Linked Lists. The other approach, as in the case of fixed size blocks, is to use linked lists. The information of the linked list can be stored separately from all memory allocation or as part of the block of memory. Either approach is workable.

After startup, the linked list contains one entry, as all available memory is in one contiguous block. When a memory request is allocated, for example, to allocate 128 bytes, the block is divided up. Suppose we allocate the first 128 bytes. A new entry is placed in the list, at 128 bytes. The node that is added contains the start address, the length of the block, and a bit indicating it is allocated. The unallocated block's node will contain the updated entry: smaller size, new start address, and the bit indicating it is unallocated. When a block is deallocated, we simply find that block in the linked list and set the bit to zero to indicate it is now available again.

In a typical system there may be a lot of allocation and deallocation of memory. This will probably lead to breaking memory up into smaller pieces. We may end up with a situation where the free blocks are small and spread out, as in the figure below:

Allocated blocks in memory after some time; the "checkerboard" situation [HZMG15].

If this happens, it may be that there is a contiguous block of free memory available of size N, but this request cannot be fulfilled because the memory is logically split up into smaller pieces. To solve this, we need a way to recombine the split blocks, commonly called *coalescence*. See the updated figure below:

The "checkerboard" situation with the adjacent free blocks coalesced [HZMG15].

Coalescence. Coalescence is just the process of merging two (or more) adjacent free blocks into one larger block. It also makes sense that dividing memory should be a reversible operation. This solves the problem of a block of N contiguous bytes being unable to be allocated. Coalescence can be done periodically or whenever a block of memory is freed.

As pointed out in [HZMG15], coalescence makes it a good idea to maintain the memory blocks in a doubly-linked list. Recall a linked list has "next" pointers connecting the nodes and a doubly-linked list has "next" and "previous" pointers, to make it easier to traverse the list in both directions. When a block is freed, it may be in the middle of two free blocks, so it is convenient to have previous and next pointers so the adjacent sections can be merged efficiently.

Even with coalescence, we may have the problem that N free bytes exist in the system but spread out over many little pieces, so the request for N cannot be satisfied. When free memory is spread into little tiny fragments, this situation is called *external fragmentation*. It is analogous to internal fragmentation in that there are little bits of space that cannot be used for anything useful, except of course that they are not inside any block (hence external).

External Fragmentation. One way to reduce external fragmentation is to increase internal fragmentation. If a request for N bytes comes in and there is a block of $N + k$ available, where k is very small (and unlikely to be allocated on its own), it makes sense to allocate the whole $N + k$ block for the request and just accept that k bytes are lost to internal fragmentation. For example, if a free block contains 128 bytes and the request is for 120 bytes, it may not be worth the hassle and overhead to split this block into 120 and 8, as it is unlikely the 8 bytes will be filled anyway. Some systems round up memory allocations to the nearest power of 2 (e.g., a request for 28 bytes gets moved up to 32). Of course, this does not really help with satisfying the request for N bytes of memory; it just keeps external fragmentation down.

Another idea is *compaction*, which can also be thought of as *relocation*. The goal is simply to move the allocated sections of memory next to one another in main memory, allowing for a large contiguous block of free space. This is a very expensive operation; to do this successfully, the Java runtime, for example, must "stop the world" (halt all program execution) while it reorganizes memory. This tends to make Java unsuitable for use in writing a real-time operating system. But even if we are willing to pay the cost, it might not be possible to do.

In previous discussions of memory management from the perspective of the application developer, languages with garbage collection like Java or C# may do memory compaction as needed when the garbage collector runs. This can work in such languages, because variables are references and unless you are writing an `unsafe` block in C#, references can be moved around in memory at the garbage collector or runtime's convenience; all it needs to do is update every reference. This is not the case in languages like C where we operate directly on memory addresses, and thanks to things like pointer arithmetic and using integer variables as addresses, there is no reliable way to update all references.

The final way we can try to prevent or deal with external fragmentation is through different allocation strategies; that is, how to fit a memory request to a block of free memory. We will examine those strategies now.

Variable Allocation Strategies

Given a memory request of N, where do we allocate the memory? If there is no block of at least size N, the request cannot be satisfied. If there is only one, the decision is easy. As long as memory has two free blocks of sufficient size (N or more) that cannot be coalesced, a memory allocation request will require making a decision about which of those blocks to split to meet the allocation request. There are five strategies we will examine [HZMG15]:

1. First fit.
2. Next fit.
3. Best fit.
4. Worst fit.

5. Quick fit.

As a performance optimization, we could have two linked lists: one for allocated memory and one for unallocated memory. That way to find a free block we do not have to look through the allocated blocks.

First fit. The strategy of first fit is to start looking at the beginning of memory, and check each block. If the block is of sufficient size, split it to allocate the memory, and return the balance to the unallocated memory list. This algorithm has a runtime of $O(n)$ where n is the number of blocks. This algorithm is simple to implement.

Next fit. This strategy is a modification of the first-fit algorithm. Instead of starting at the beginning of memory and finding the first block that meets the request, keep track of where the last block was allocated, and then start the next search after that. This prevents the situation where there are a lot of small unallocated blocks (external fragmentation) all concentrated at the start of memory [HZMG15]. The runtime is still $O(n)$, as with first fit.

Best fit. Instead of just walking through the list and splitting up the first block equal to or larger than N, we could instead try to make a more intelligent decision. Considering all blocks, we choose the smallest block that is at least as big as N. This produces the smallest remaining unallocated space at the end.

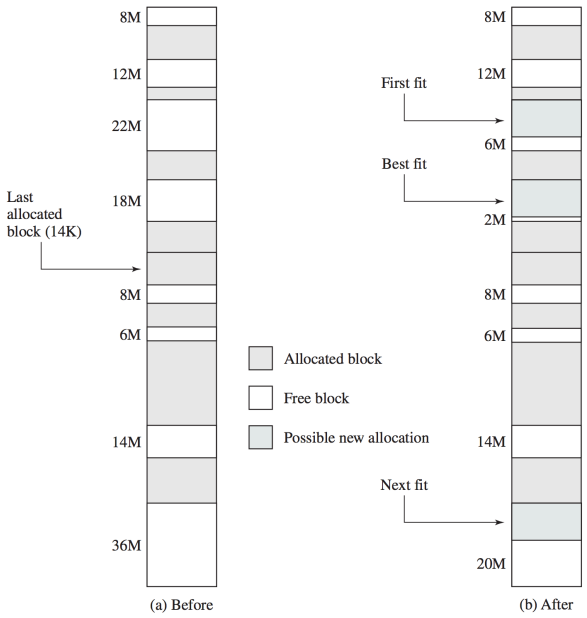
This would require either (1) checking every available block ($\Theta(n)$ runtime); or (2) keeping the blocks sorted by increasing size ($O(n)$ runtime). If we use an AVL tree or red-black tree, then we can get best fit to run in $\Theta(\ln(n))$, possibly better runtime than first fit [HZMG15].

Worst fit. The problem with best fit is that the leftover bits of memory are likely to be too small to be useful. Rather than trying to find the smallest block that is of size N or greater, choose the largest block of free memory. When the block is split, the remaining free block is, hopefully, large enough to be useful.

As with best fit, we must either (1) check each available block; or (2) keep the block sorted by size, though decreasing size this time. A max heap is appropriate, or a binomial or Fibonacci heap could also be appropriate [HZMG15].

Quick fit. Though not a solution on its own, quick fit is an optimization. If memory requests of a certain size are known to be common, e.g., requests for 1 MB, it might be ideal to keep a separate list of blocks that are of perhaps 1-1.1 MB in size, so that if the request for 1 MB does come in, it can be satisfied immediately and quickly.

Example: 16 MB Allocation. The diagram below illustrates where in memory a request of 16 MB may be placed:



An example of where the first, best, and next fit algorithms would place an allocation [Sta14].

The worst fit algorithm is not shown, but it would overlap with the placement indicated for next fit, because the largest block of free space before the allocation is is 36M.

Choosing a Strategy

According to [SGG13], simulations show that worst fit performs, well, worst in terms of time required to fulfill an allocation request and that it results in the most wasted space. The performance problems of worst fit can be fixed, of course, by keeping the memory blocks in a max heap, but that still does not address the wasted space problem. First (next) and best fit are about equal in how well they utilize memory, but first fit tends to be faster. Despite this, even with optimization, given x allocated blocks, another $0.5x$ blocks may be lost to fragmentation.

This is supported by [Sta14], whose analysis also indicates that first fit is the fastest and best algorithm. The next fit algorithm tends to do allocations at the end of memory, so the largest block of free memory (typically at the end) is quickly broken up. On the other hand, first fit tends to litter the beginning of memory with small fragments. Best fit tends to produce free blocks that are too small to be useful.

Advanced Strategy: Binary Buddy

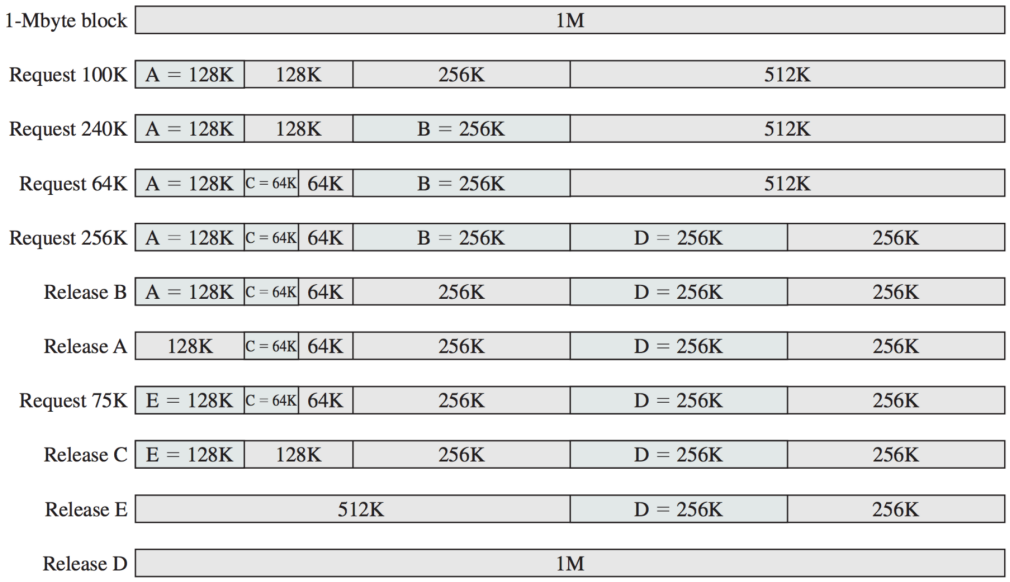
Now let us examine a compromise between fixed and variable allocation, as laid out in [Sta14]. There is some internal fragmentation, but it is a trade-off against how much external fragmentation we are willing to accept.

In a buddy system, memory blocks are available in powers of 2. More formally, a block is of size 2^K, where $L \leq K \leq U$ and 2^L is the smallest block size that can be allocated and 2^U is the largest block size that can be allocated (usually the full size of memory).

Initially, memory is treated as a single block of size 2^U. If a request of size n occurs such that $2^{U-1} < n \leq 2^U$, then the entire block is allocated. Otherwise, the block is split into two "buddies", of size 2^{U-1}. If $2^{U-2} < n \leq 2^{U-1}$, allocate one of the blocks of 2^{U-1} to the request. Otherwise, subdivide again. Repeat until the smallest block greater than or equal to n is allocated.

In subsequent allocations, we look through the data structure, typically a tree, to find either (1) a block of appropriate size; or (2) a block that can be subdivided to meet the allocation. Whenever a pair of buddies (two blocks of equal size, split from the same "parent") in the list are both free, they can be coalesced.

Consider the example below where a 1 MB block is allocated using the Binary Buddy system.



An example of the Binary Buddy system for memory allocation [Sta14].

22 — Memory: Segmentation and Paging

Memory Segmentation

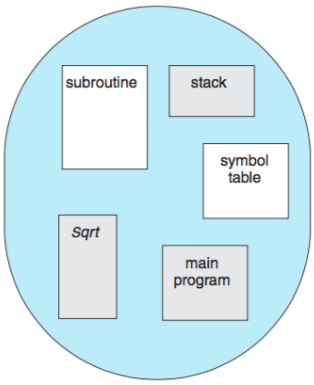
Though you've been repeatedly told that memory is a linear array of bytes, you have also likely been told that there's the stack and the heap, libraries and instructions. Both of these are true; they are simply views of memory at two different levels of abstraction. Each of the elements such as the stack, the heap, the standard C library, et cetera, are known as *segments*.

Programmers do not necessarily give much thought to whether variables are allocated on the stack or the heap, or where program instructions appear in memory. In many cases it does not matter, though C programmers are well advised to know the difference.

A full program has a collection of segments, each of which may be different lengths. Normally the compiler is responsible for constructing the various segments; perhaps one for each of the following [SGG13]:

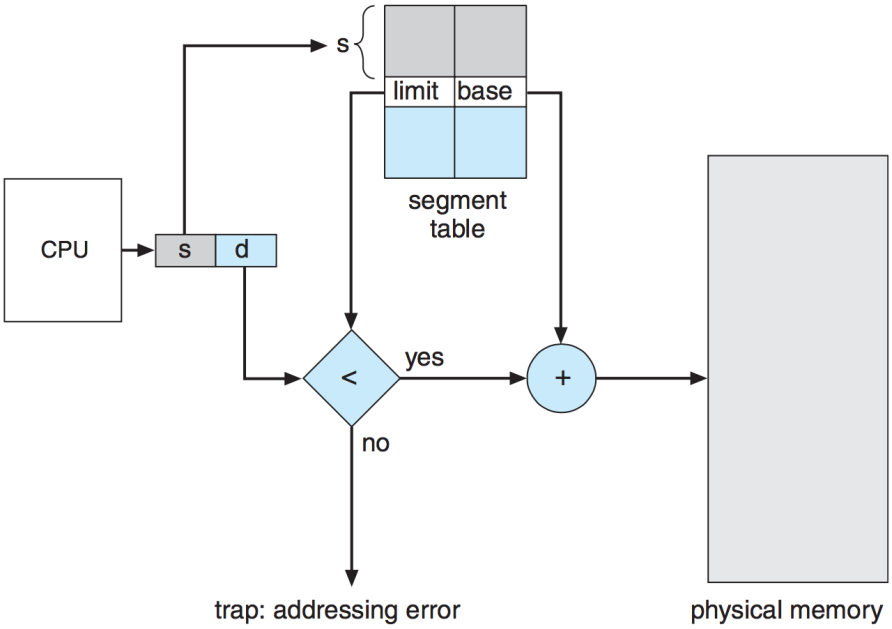
1. The code (instructions).
2. Global variables.
3. The heap.
4. The stack (one per thread).
5. The standard C library.

From the programmer's perspective, memory may simply be various blocks, as below:



One way programmers might look at memory [SGG13].

Rather than thinking about memory as just a pure address, we can think of it as a tuple: *<segment, offset>*. Given that, we need an implementation to map these tuples into memory addresses. The mapping has a segment table; each entry in the table contains two values: the base (starting address of the segment) and the limit (the length of the segment). So there will be some addition involved as well as a comparison to see if the address lies within that range. As is typically the case, memory accesses are such a common operation that we will need another rescue from the hardware folks to make this not painfully slow.



Segmentation hardware [SGG13].

With segmentation, memory need no longer be contiguous; we can allocate different parts of the program in different segments; different segments can be located in different areas of memory.

Paging

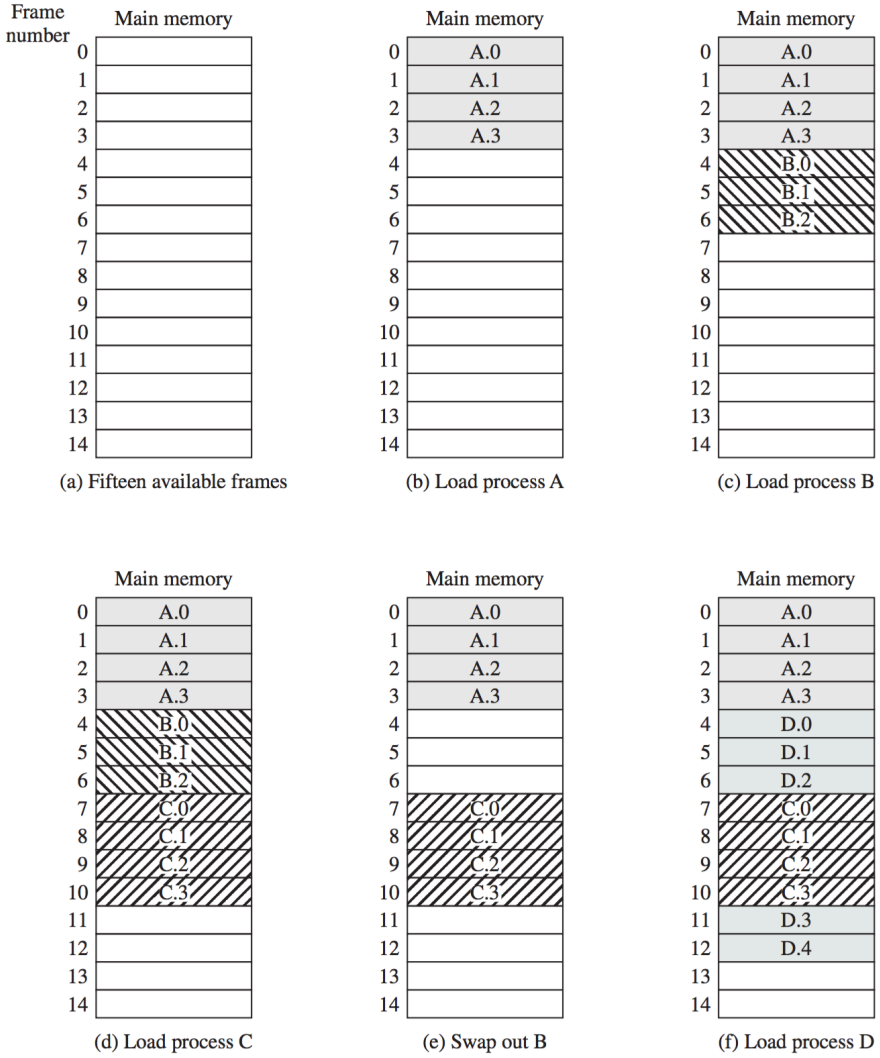
Fixed and variable sized partitions suffer from fragmentation, whether external or internal. Let us divide memory up into small, fixed-size chunks of equal size, called *frames*. Each process's memory is divided up into chunks the same size as a frame, called *pages*. Then a page can be assigned to a frame. A frame may be empty or may have exactly one page in it.

Imagine, as an analogy, a simple picture frame. The frame may be empty or it may contain a picture. If the picture frame is empty, all that is necessary is to put a picture in it. To put in a different picture, you would first need to take out the picture that is already there. Taking out the picture to empty the frame is allowed, too. A picture is always aligned so that it is completely in one frame; not half in and half out. Now expand this scheme by having a very long row of picture frames, each of which can contain one picture at a time, at most, and a picture can be in at most one frame at a time.

When a process starts, it is loaded into memory, and has some initial memory requirements (e.g., the stack and global variables), so it will require some number of pages. The number of pages can and will change over time as memory is allocated and freed. A process may also be swapped out to disk, but to run it will need to be swapped back in. Either way, a process will take up a certain number of pages in memory at any given time.

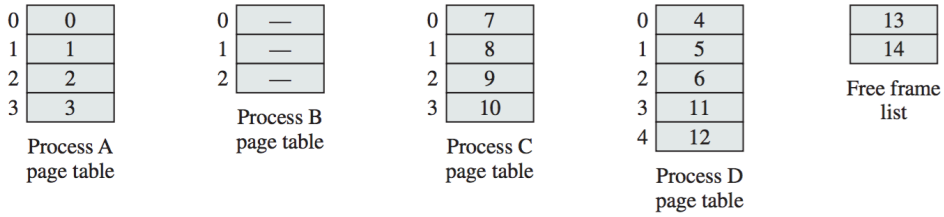
Pages provide the benefit of separating the logical address from the physical address: programmers may pretend the address space of the computer is 2^{64} bytes rather than however many GB of memory are in the physical machine.

Consider the diagram below, in which there are 15 free frames initially. Then we load process A, which consists of 4 frames; then process B with 3 frames, C with another 3 frames. Eventually, B is swapped out (put back on disk) and D is loaded. Process D has five frames, but the frames do not have to be contiguous (and although they are shown in order, they do not necessarily have to be).



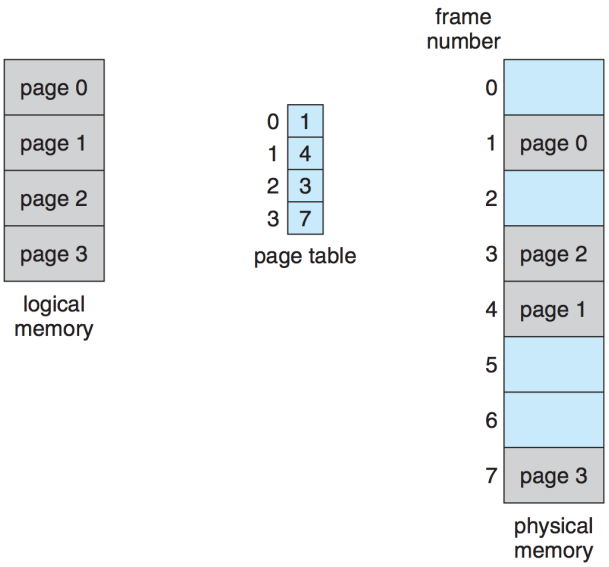
Assignment of process pages into free frames [Sta14].

Now that we have multiple segments for each process and they are no longer contiguous, it is insufficient to have a base address and a limit. Now instead, each process needs a page table, to keep track of which pages are located where in memory. A list of free frames is also necessary. See the diagram below:



The page tables as of (f) in the previous diagram [Sta14].

The page table is used to map logical memory to physical memory, as in the diagram below:



Mapping of logical memory to physical memory via the page table [SGG13].

For convenience, page size is usually a power of 2 (and the actual value is determined by hardware). The selection of a power of 2 makes translating a logical address into a tuple of the page number and offset easy. If the logical address space has size 2^m and the page size is 2^n bytes, n obviously smaller than m, then the high order $m - n$ bits of the logical address are the page number and the lower n bits are the page offset [SGG13].

External fragmentation is eliminated as a problem in this scheme, because pages are all the same size. That also means that compaction is not an issue. Compaction, when it's possible, is painful enough in memory; it is excruciating to do on disk. It is therefore desirable to avoid it entirely. We accept some internal fragmentation because a process gets a whole page at a time.

But how much internal fragmentation do we have to live with? Not very much. If the memory required aligns perfectly with a multiple of the page size, then no memory is wasted. If a new memory allocation comes in, then a new page is allocated and added to the logical memory space of the process. The last frame, however, may not be completely full. In the worst case scenario, a full page less one byte is wasted. However, internal fragmentation of one page is not very much in the grand scheme of things.

How big should page sizes be? If they are smaller, then less memory is wasted in internal fragmentation. However, having a large number of pages introduces a lot of overhead. The size of pages has tended to grow along with the size of main memory in computers [SGG13]. The key factor is actually disk: the disk operates on a certain block size and it is most efficient for the size of a page to be equal to a disk read/write size. That way when a page is to be swapped into or out of memory, it can be done in a single disk read or write. In a typical modern system, pages are 4 KB, but they can be bigger.

Now we finally have a good answer to why the application developer can treat memory as if it is infinitely large and unshared. The program is scattered across physical memory, but appears to the application developer and running application as if it is all contiguous.

We also get protection in this scheme: a program cannot access any address outside of its memory space. There is simply no way to make a memory request outside of the logical memory space. No matter what address is generated, it could only be inside the page table, and the page table has only entries of that process.

The operating system, however, can manage memory of all processes, so it will need another scheme. The OS will operate on the *frame table*, a listing of all the frames, indicating which page of which process a frame currently holds, if any.

Shared Pages

Another great advantage of paging is the possibility of sharing of common code. Users very often have multiple programs open; and sometimes they are duplicates (e.g., notepad, Microsoft Word, et cetera). In a multiuser system, different users may have some of the same program open (e.g., Skype, Firefox, et cetera). We could reduce memory consumption if common parts of this program are shared between all instances of that program.

Example: consider the text editor `vi`. Imagine there are 5 users on the system, each of whom wants to use `vi`. Let's say the program itself uses 10 pages (made up number) on its own, and then some variable number of pages based on what file is being edited. Without sharing, each copy of `vi` that runs will consume 10 pages, so 50 pages are being used for the executable. If we can share those 10 pages, we have saved 40 pages worth of memory space.

Other programs and code can easily be shared, such as compilers, libraries, and operating system utilities. In fact, any code can be shared as long as it is *reentrant* (also sometimes called pure or stateless). This is code that does not change when it is executed. That means there is no state maintained by the code. Any function that accesses a global or static variable is non-reentrant, such as [HZMG15]:

```
int tmp;
void swap( int *x, int *y ) {
  tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Page Table Structure

In the simplest form, the page table is just a standard table. This structure is simple, but page tables can be very large. If the system is 32-bit, and page sizes are 4 KB (2^{12}), then the page table has $2^{32}/2^{12} = 2^{20}$ pages, or about 1 million entries. Given that page tables themselves can be quite large, we will examine three strategies for how to structure the page table, other than the simple table structure:

1. Hierarchical paging.
2. Hashed page tables.
3. Inverted page tables.

Hierarchical Paging. Rather than have one big table, we have multiple levels in the page table; this means the page table can be broken up and need not be contiguous in memory. Suppose we have a two level system. If the page number is p, the first k bits indicate the *outer page*. The outer page then contains some information about where the *inner pages* are. The remaining $p - k$ bits identify the inner page. After the inner page is identified, the displacement d is then calculated from the inner page [SGG13].

Hashed Page Tables. Instead of the page table being an array of entries, turn it into a hash table. There is a hash function to assign pages to "buckets" and each bucket is implemented as a linked list. Then each element of the list is examined to find the matching page.

Inverted Page Tables. For 32-bit virtual addresses, a multilevel page table can work. But with 64-bit computers, with 4 KB pages, the page table requires 2^{52} entries, and if an entry is 8 bytes, then the table is over 30 million gigabytes (30 PB). Using this much memory for the page table is unrealistic. Instead, we can have an inverted page table: there is one entry per frame, rather than one entry per page. The entry keeps track of the process and page number. This saves a huge amount of space (1 GB of ram with a 4 KB page size means the page table requires only 2^{18} entries). The drawback is that it is no longer possible find a physical page by looking at the address; instead, we must search the entire inverted page table. Slow as this operation is, we can make it faster via hardware... [Tan08].

Paging: Hardware Support

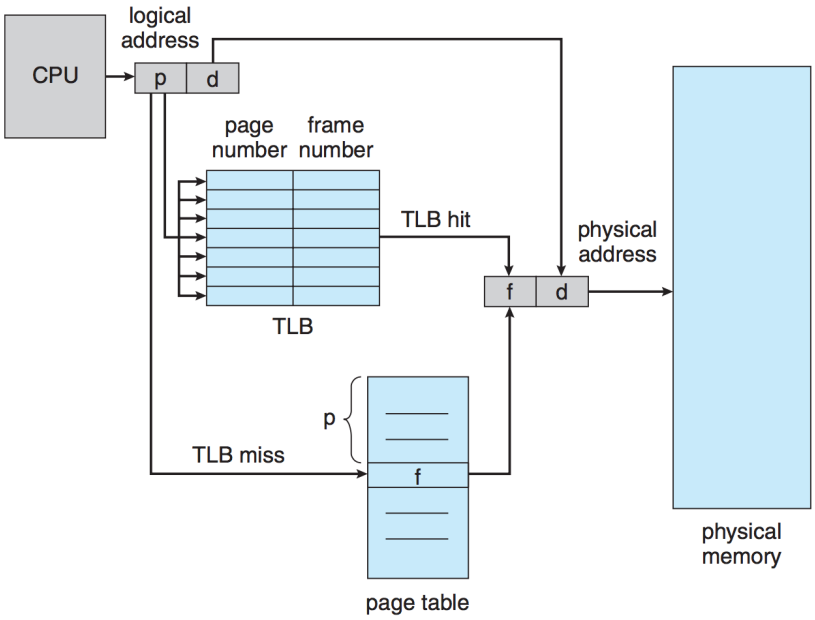
As we have repeatedly discussed, memory accesses are very frequent and often require additions and comparisons. After all, an operation as simple as adding two numbers requires fetching the add instruction, fetching the operands, and storing the result. To prevent abysmal performance, modern computers have hardware support, because hardware is much, much faster than doing these operations in software.

The simplest implementation of the page table is to use a set of dedicated registers. Registers are the fastest form of storage. When a process switch takes place, these registers, just as all other registers, are replaced with those of the process to run. The PDP-11 was an example of a system that had this architecture. Addresses were 16 bits and the page size was 8 KB. Yes, it was a very long time ago. The page table was therefore 8 entries and kept in fast registers. This might work if the number of entries in the page table is small (something like 256 entries). The page table can easily be something like 1 million entries, so it would be a little bit expensive to have that many registers. Instead, the page table is kept in main memory and a single register is used to point to the page table [SGG13].

Unfortunately, this solution comes with a big catch. To access a page from memory, we need to first figure out where it is, so that requires accessing the page table in main memory. Then after retrieving that, we can look in the page table to find the frame where the desired page is stored. Then we can access that page. So two memory accesses are required for every read or write operation. Remember that as far as the CPU is concerned, main memory already moves at a snail's pace. Doubling the amount of time it takes to do a read or write means it takes roughly forever. Thus, we will need to find a way to speed this up.

Surprise surprise, the solution is hardware: a fast cache called the *translation lookaside buffer* (TLB). You can think of the TLB as a key-value pair (think HashMap). The key is the logical address and the value is the physical address. To make the search fast, the comparison is done on all items simultaneously. To prevent this from being extremely expensive, the size of the TLB is limited; it's usually something around 32 to 1024 entries in size. Systems have evolved from having no TLBs to having multiple levels, over time [SGG13].

When a memory read or write is issued, the page number is checked against the TLB. If it is found in the TLB then the frame number is immediately known. If the page number is not found in the TLB, this is what we call a *TLB miss* and we must look in the full page table, which unfortunately is slower because it requires reading from memory. See the diagram below:



Mapping of logical memory to physical memory via the page table [SGG13].

The TLB idea is a specific instance of the strategy of caching. Much earlier, when talking about the basics of computer hardware, we mentioned that memory comes at different levels and and different speeds. Caching is a

critical idea in computers and operating systems. In fact, caching is such an important topic, that it will be the next topic examined.

23 — Caching

Caching

Caching is very... hit and miss.

Caching is very important in computing, and not just memory. We examine the idea of caching in the context of memory, but it is applicable any time there is a large resource that is divided into pieces, some of which are used more often than others. Caching provides a significant benefit in some circumstances and not useful in others (hence "hit and miss"). The goal of caching is to speed up operations. It is desirable to read information from cache, when possible, because it takes less time to get data from cache to the CPU than from main memory to the CPU. CPUs are a lot faster than memory and it is best if we do not keep them waiting.

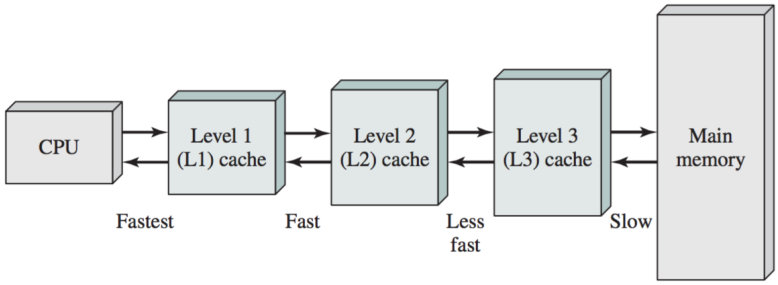
Caches do not have to operate on pages; they can operate on anything, but they are typically blocks of a given size. An entry in a cache is often called a *line*. We will assume for the balance of this discussion that a cache line maps nicely to a page.

As discussed, the CPU generates a memory address for a read or write operation. The address will be mapped to a page. Ideally, the page is found in the cache, because that would be faster. If the requested page is, in fact, in the cache, we call that a cache *hit*. If the page is not found in the cache, it is considered a cache *miss*. In case of a miss, we must load the page from memory, a comparatively slow operation. A page miss is also called a *page fault*. The percentage of the time that a page is found in the cache is called the *hit ratio*, because it is how often we have a cache hit. We can calculate the effective access time if we have a good estimate of the hit ratio (which is not overly difficult to obtain) and some measurements of how long it takes to load data from the cache and how long from memory. The effective access time is therefore computed as:

$$\text{Effective Access Time} = h \times t_c + (1 - h) \times t_m$$

Where h is the hit ratio, t_c is the time required to load a page from cache, and t_m is the time to load a page from memory. Of course, we would like the hit ratio to be as high as possible.

Caches have limited size, because faster caches are more expensive. With infinite money we might put everything in registers, but that is rather unrealistic. Caches for memory are very often multileveled; Intel 64-bit CPUs tend to have L1, L2, and L3 caches. L1 is the smallest and L3 is the largest. Obviously, the effective access time formula needs to be updated and expanded, when we have multiple levels of cache with different access times and hit rates. See the diagram below:



Three levels of cache between the CPU and main memory [Sta14].

If we have a miss in the L1 cache, the L2 cache is checked. If the L2 cache contains the desired page, it will be copied to the L1 cache and sent to the CPU. If it is not in L2, then L3 is checked. If it is not there either, it is in main memory and will be retrieved from there and copied to the in-between levels on its way to the CPU. Because caches have limited size, we have to manage this storage carefully.

Page Replacement Algorithms

Whenever a page fault occurs, the operating system needs to choose which page to *evict* from (kick out of) the cache to make space for the new one. This assumes that the cache is full, which it likely is except at system startup. We could, of course, just select a random page, but we should do this task more intelligently, if we can.

To make an intelligent decision about what sort of strategy to choose, we need to know a few things about how data is accessed in a system. A few observations from [HZMG15]:

1. A rule of thumb in software engineering is that 10% of the source code will be executed 90% of the time. This is a variant on the Pareto Principle, also known as the 80/20 rule[12]. This may seem sensible to you given that code has a lot of handling of special cases and rarely used operations.
2. The principle of *temporal locality*: a memory location that has been recently accessed is likely to be accessed again in the future.
3. The principle of *spatial locality*: a memory location near one that has recently been accessed is likely to be accessed again in the future.

An example of code that would involve both spatial and temporal locality might be a function that sums up all the values of an array. The sum variable is accessed repeatedly, and the fact that it was recently accessed means it is likely to be accessed again soon. The array being accessed at index i now means it is likely that the array at index $i + 1$ is likely to be accessed soon.

If a page has been altered in cache, then that change has to be written to main memory at some point. It can be done immediately when the page is changed, or it can be done when the page is evicted from the cache. The second option means fewer main memory accesses, if a page is written to multiple times before it is sent to main memory. If that memory is shared, however, between multiple processors or an I/O device, then the delay in updating main memory may be intolerable. If a page has not been modified in cache, it can simply be overwritten. If all other factors are equal, we should replace a page that has not been modified, as the work to write it out to memory need not be done.

The Optimal Algorithm. The optimal page replacement algorithm is fairly simple: replace the page that will be used most distantly in the future. For each page, make a determination about how many instructions in the future that page will be accessed. The selected page is the one with the highest value.

Unfortunately, there is a glaring flaw in this algorithm: it is impossible to implement. It requires clairvoyance (seeing into the future), and at least as far as I know, nobody has invented a way to do so reliably[13]. The program and operating system have no real way of knowing which pages will be used in the future.

As it is unimplementable, it is mostly a benchmark against which other algorithms can be compared. If we know that a given algorithm is, say 1% less efficient than the hypothetical optimal algorithm, then no matter how much we improve that algorithm, the best performance increase we can get is 1% [Tan08].

Not-Recently-Used. Operating systems may collect page usage statistics. If so, computers may have two status bits associated with each page, called R and M. The R bit is set when a page is referenced (either read or written)

[12]Note that the 80/20 rule is not always applicable. For example, studying 20% of the course material is unlikely to earn you 80% of the marks on the exam.

[13]Recall the joke "Why do you never see the newspaper headline that a psychic has won the lottery?"

and the M bit is set when the page is written to (modified). Once a bit is set, it remains so until the OS changes its value. The R and M bits can used to build a paging algorithm as follows [Tan08]:

Initially, R and M are 0. Periodically, the R bit is cleared to distinguish pages that have not been recently referenced. This may happen every clock interrupt, for example. When a replacement needs to take place, the operating system will examine all the pages and sort them into four buckets based on the R and M bits, in ascending order of precedence:

1. Not referenced, not modified.
2. Not referenced, modified.
3. Referenced, not modified.
4. Referenced, modified.

The OS will prefer to remove a page from the lowest-numbered class, when possible. This NRU algorithm is fairly easy to understand and may provide adequate performance.

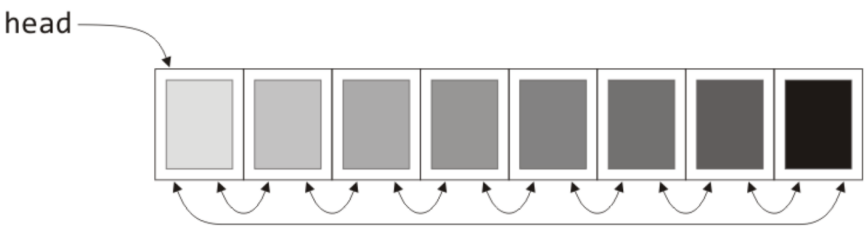
First-In-First-Out. First-In-First-Out is quite easy to understand and implement; the idea is some sort of temporal locality. If there are N frames, keep a counter that points to the frame that is to be replaced (the counter ranges from 0 to $N - 1$). Whenever a page needs to be replaced, replace the page at the counter index and increment the counter, wrapping around to 0 where necessary.

Some of the time this strategy works: if a page is not going to be referenced again (or at least, for a very long time), it is a good choice to get rid of. Often times, however, the same page is referenced repeatedly, and this algorithm does not take that into account. If a page is referenced often, we want to keep it in memory, not evict it just because it has been in the cache the longest. So FIFO is probably not the best choice.

A Second Chance (the Clock Algorithm). To improve on the FIFO algorithm, suppose we give pages a "second chance" based on whether or not the R bit is set. If the oldest page has not been used recently (the R bit is not set), then the oldest page is removed. If the oldest page has recently been referenced, the R bit is cleared. The search then goes backwards (to the next-oldest page) and repeats the procedure. This happens until a page is removed. Because the algorithm clears the R bit as it goes, even if all pages have recently been referenced, eventually a page will be selected and evicted. Thus the algorithm will eventually terminate. The index is updated whenever this happens so a page that got a second chance is not the "oldest" anymore. In some textbooks this is called the Clock replacement algorithm because we can think of the cache as a circular buffer and the current oldest page as being pointed to by the hand of the clock [Tan08].

This addresses the problem of a page that is frequently used eventually becoming the oldest and being evicted, only to be brought back into memory again immediately afterwards.

Least Recently Used (LRU). The least recently used (LRU) algorithm means the page that is to be replaced is the one that has been accessed most distantly in the past. You might consider time stamps and searching a list, but because there are only two operations, it need not be that complex. When a page in the cache is accessed, move that page to the back of the list. When a page is not found in cache, the page at the front of the list is removed and the new page is put at the back of the list. This requires nothing more than a cyclic doubly-linked list. Both operations can be performed in $\Theta(1)$ time [HZMG15].



A cyclic, doubly-linked list in which the `head` pointer indicates the least recently used page [HZMG15].

Not Frequently Used (NFU). The not frequently used (NFU) algorithm is similar to the LRU approach, but can be implemented in software rather than relying on hardware support that may not be present. Each page gets an associated software counter, which starts at 0. Whenever the R bit would have been updated to 1, 1 is added to the counter. When a page is to be replaced, the page with the lowest counter value is the one that is replaced [Tan08].

It may occur to you that this solution has a problem: like an elephant, it never forgets and counters can never decrease. A page that was accessed very frequently at the start of the program will accumulate a high counter value early on, and therefore might never be evicted, even though it is not needed again. What we need is a way for the count to decline over time.

The solution is called *aging*: counters are shifted to the right by 1 bit before the 1 is added; and instead of adding 1 (which increments the rightmost bit), set the leftmost bit to 1. Now we have a bit array of byte size (no need for a 4-byte integer, most likely). Thus, a page that has not been referenced in a while has its value decrease over time. The page replacement algorithm still evicts the lowest value page when it is time to replace a page. We lose a certain amount of precision compared to the LRU approach: if pages a and b both have patterns of `00000001` we know they were both last accessed 8 cycles ago, but all history before that point is lost. Which page between a and b was least recently used is unknown [Tan08].

Pre-Paging. Thus far all of our strategies for putting things in the cache have been "on-demand": it is only when a page is needed and not found in the cache that a page is loaded into the cache. Suppose instead that the operating system takes some steps to guess about what pages might be needed next, based on the principle of temporal or spatial locality. This might reduce the amount of time spent waiting for a page in the future.

Predicting which pages are likely to be used in the near future is, indeed, a matter of clairvoyance. There exist various techniques to determine the likely pages to be used frequently (called the "working set"), but they are complex and will not be examined further right now.

A situation in which pre-paging might be useful is when a program is started or swapped into memory. Technically, no pages of the process need to be loaded into the cache to start with; we can simply suffer through a whole bunch of page faults. This is rather slow; it might be better to load multiple pages into the cache to start with, though this will of course require some guesses about what pages will be needed [Tan08].

Choosing an Algorithm

Consider the following table that gives a quick overview of the algorithms we have discussed:

Algorithm	**Comment**
Optimal	Impossible to implement, but a benchmark to compare against
Not Recently Used	Not very good performance
First In First Out	Highly suboptimal
Second Chance	Much better than FIFO, but just adequate
Least Recently Used	Best performer, difficult to implement?
Not Frequently Used	Approximation of LRU
NFU + Aging	Better approximation of LRU

If hardware is available to support it, the LRU algorithm is the best. If not, the NFU + Aging scheme is the next best thing we can implement. Unless there is a specific reason to choose one of the other algorithms, LRU is very likely to be the algorithm selected.

Local and Global Algorithms

When a process switch occurs, we could dump the entire cache to make way for the next process to run, but this is most likely unnecessary work. If a process P_1 is suspended now, P_2 may run for a while and then P_1 runs again; it may be that when P_1 starts again, some of its pages are still in the cache and do not need to be loaded again. If P_2 did replace all the cache lines, then P_1 will have to load them all in again, but that is the worst case scenario. So we may have multiple processes with pages in the cache at a given time.

Another important consideration in the page replacement algorithm is whether that algorithm should care about which process the page belongs to or not. Suppose we are using the LRU algorithm. If process P_1 has a page fault, do we replace the least recently used page in all the cache (global replacement)? Or do we replace the least recently used page in the cache that belongs to P_1 (local replacement)?

Of course, different levels of cache might have different strategies: if the L1 cache is 16 KB and pages are 4 KB, there can be 4 pages in the L1 cache, so global replacement probably makes sense there. If L2 cache is 256 KB there are 64 pages and maybe local replacement makes sense there, but even 64 pages is fairly small. When caches are somewhat larger, however, things may be a bit more interesting.

Local algorithms give each process some (roughly) fixed number of pages in the cache. Global algorithms dynamically allocate cache space to different programs based on their needs, so the number of pages in the cache for each process can vary over time.

According to [Tan08], global algorithms work better, especially when processes' memory needs change over time. If the local algorithm is used, we may be wasting some of the space in the cache for a process that does not really need it.

Suppose we have a sufficiently large cache, and a dynamic allocation (global algorithm) with some intelligence to keep any process from having too many or too few pages in the cache. To manage the complexity of how much of the cache should be allocated to a particular process, we might wish to keep track of the number of page faults with a measure called the *page fault frequency* or PFF. In simplest terms, the PFF is the guide for telling whether a given process has too few, too many, approximately the right number of pages.

If the PFF is above some upper threshold, that indicates that more cache space is needed for that process. If the PFF is below some lower threshold, it suggests that the process has too much cache space allocated and could stand to part with some.

PFF relies on an assumption: that the page replacement algorithm has the property that fewer page faults will occur if a process has more pages assigned. This is the case for the LRU algorithm, but not necessarily true for the FIFO approach [HZMG15].

24 — Virtual Memory

Virtual Memory

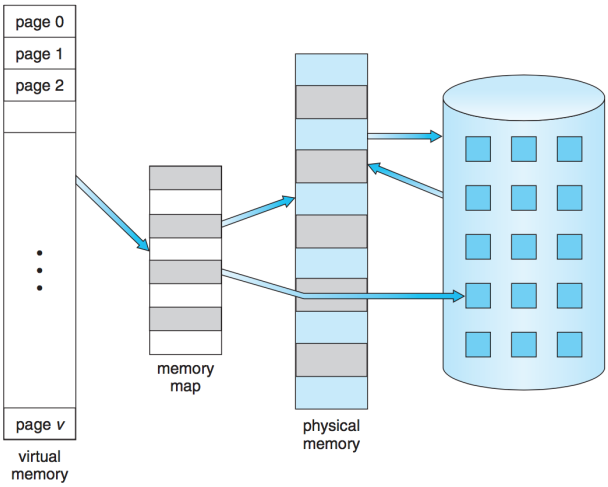
Even with paging and moving pages into and out of memory, there is a limit on what we can do with the system. Specifically, a program that requires more memory than the machine's physical memory cannot run. Maybe that seems ridiculous in the modern era of 4, 8, and 16 GB of memory, but server or supercomputer systems working on large datasets may require more memory than is available in the machine. Furthermore, when we have many programs running and a multiprocessor systems, we could have a situation where the sum of memory requirements exceeds the available memory. It is less than ideal to have a processor waiting for something to do because a process that is otherwise ready to run cannot proceed because there are insufficient free frames for it to run.

The problem is: a process must be entirely in memory or entirely on disk. In a lot of cases, the entire program is not needed at any given time. Code used to handle unusual situations (error handling, etc.) may not be needed except occasionally. Startup code is needed at the beginning of the program, but then never again. Data structures and collections may be declared to be very large even when it is not actually needed (e.g., the `ArrayList` in Java defaults to 16 elements, even if you might only need 4, wasting a bit of space).

If we could execute programs that are only partly in memory, there would be three major benefits [SGG13]:

1. A program is no longer constrained by the size of physical memory; programmers can use the entire virtual space without worrying about whether it fits.
2. Each program could use up less physical memory, allowing more processes to execute concurrently.
3. Less I/O is needed to swap user programs in or out.

Good news, everyone! We have already discussed many of the key ideas to making such a system work when we examined caching. The principle is really the same: main memory can be viewed as yet another level of cache and the disk is the last stop where the data can be. If a page is referenced and not currently in main memory, it is a page fault, and the page is loaded from disk into main memory. A page might need to be evicted from main memory to make way for it; thus a page replacement algorithm is needed to select the "victim" page and write it out to disk.



Virtual memory exceeding the size of physical memory, with some pages on disk [SGG13].

The typical approach is also like that of the cache, which is to use demand paging. A page is loaded into memory only if it is referenced or needed, thus preventing unnecessary disk accesses. This is also called the "lazy" approach in [SGG13], though lazy is typically an insult and in this case it is not necessarily bad. Clearly, we would like to involve disk as little as possible, because disk is, from the perspective of the CPU, extremely slow.

The fact that disk is so slow means we can get into a troublesome state called *thrashing*: the operating system is spending most or all of its time swapping pages in and out of memory and very little actual work can get done.

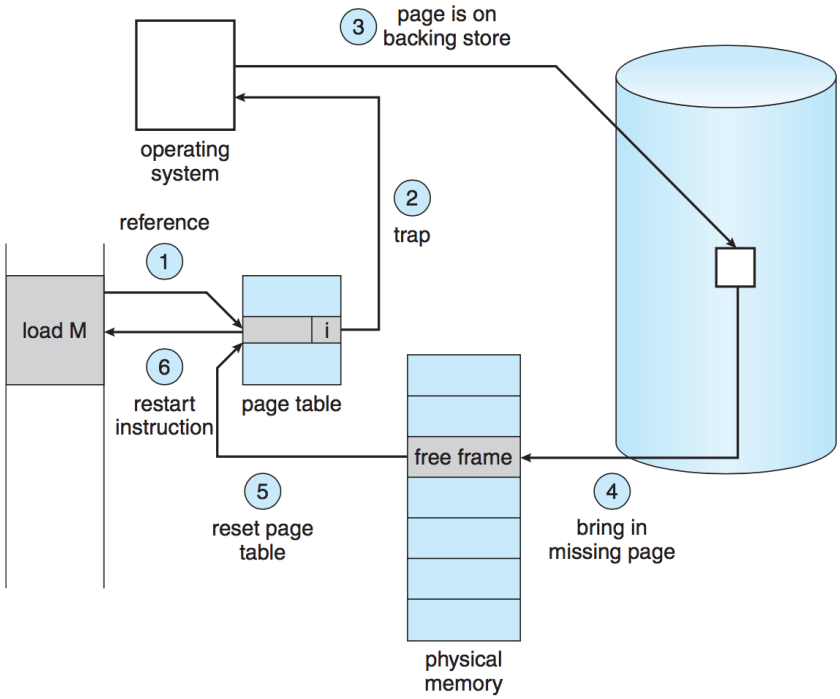
Apparently, if the NeXT operating system, NeXTStep[14] were booted on a machine with only 2 MB of RAM instead of the expected 4 MB, the steady state of the system would be a constant level of swapping [HZMG15]. This was particularly bad because the most common cause of failure in the NeXT boxes was hard disk drive. But they did have cool magnesium cases, so after they died they at least made impressive conversation pieces and doorstops.

With virtual memory, each memory reference is a six step process [SGG13]:

1. Check if the memory reference is valid or invalid (just as we have done before).
2. If the reference is invalid, terminate the program (segmentation fault). If it was valid, but the page referenced is not in memory, we will need to retrieve it.
3. Find a free frame (or make one by evicting some other page).
4. Request a disk read (and possibly write) to bring in the new page.
5. When the disk read is complete, update the records to show the new page is in memory.
6. Restart the instruction that referenced the page that needed to be brought into memory.

Or, to view this visually:

[14] One of the three parents of Mac OS X: the classic Mac OS, BSD [UNIX], and NeXTStep.



Handling a page fault [SGG13].

Note that between steps 4 and 5, a significant amount of time will take place while the disk performs a read of the desired page and possibly a write of the page to be evicted if it was modified. While the slow disk operations are going on, the process is blocked on that I/O operation and, in the meantime, the processor can and should be working on something else.

The key requirement in the described workflow is the ability to restart any instruction following the page fault. We save the state of the process, including all the registers and instruction pointer and so on, when the page fault occurs, so we are able to restart the process exactly where it was. The difference is that after the restart, the page needed is in memory and is accessible. A page fault could occur on any memory reference, including fetching the next instruction. If it happens at that time, the fetch operation is done again. If a page fault happens when doing an operation that required fetching an operand, then we fetch and decode the instruction again and then fetch the operand. So a little bit of work may be repeated [SGG13].

Consider the `ADD` instruction that adds `A` to `B` and stores the result in `C`. First, we must fetch and decode the instruction. That tells us about the two operands, which must be retrieved themselves. Then we can add the two operands, and store the result in the target location. If a page fault occurs when trying to write to `C`, because that page is not currently in memory, we will restart the instruction. That means back to step one: fetch the `ADD` instruction again, then get the operands, then perform the addition, and finally write it into the destination location [SGG13].

As we can see, some work is repeated here: fetching and decoding the instruction, as well as taking the operands and doing the addition. This is not a big deal, because the time it takes the CPU to do such an operation is minuscule. CPUs are very good at executing instructions and doing this one a second time is not a big task.

While fetching the page containing `C` from disk, the page that contains `A` or `B` could get swapped out, meaning that the second run of the instruction will also produce a page fault. This is unlikely if using a sane replacement algorithm, because the page with `A` and `B` having just been referenced, it is a poor candidate for eviction. But a random page replacement algorithm could result in that behaviour on occasion. While hypothetically possible, it is very unlikely that a system would get stuck on the same instruction forever if the page containing `A` were constantly replaced in memory and cache by the page containing `C` and vice versa. But a system vulnerable to this problem would have some very significant design issues, to say the least.

The question is, can every instruction be restarted without affecting the outcome? The answer is no; an example is the situation that occurs when an instruction modifies more than one memory location. If we are moving a block

of n bytes, it is possible those bytes will straddle a page boundary[15], either at the source or destination. We would like to avoid this situation, because the move operation may not be easily restarted if the source and destination overlap (i.e. the source is modified). One is for the CPU to try to access the start and end addresses before the move begins; if one of the pages needed is not in memory, the page fault is triggered before any data is changed, so we can be sure the move will succeed when it actually starts. Another solution is temporary registers to hold overwritten location; if a page fault occurs, then the temporary data is restored so the instruction may be restarted without affecting the operation's correctness [SGG13].

Virtual Memory Performance

As we have already seen, finding something in the cache is significantly faster than having to go to main memory. Retrieving something from disk is dramatically slower, but computing how long it takes to retrieve a given page will follow the same principle. Recall from earlier the effective access time formula:

$$\text{Effective Access Time} = h \times t_c + (1 - h) \times t_m$$

Where h is the hit rate, t_c the time to retrieve a page from cache, and t_m is the time to retrieve it from memory. If we replace the terms t_c and t_m with t_m and t_d (time to retrieve it from disk) respectively, and redefine h as p, the chance that a page is in memory, we can get an idea of the effective access time in virtual memory:

$$\text{Effective Access Time} = p \times t_m + (1 - p) \times t_d$$

And just while we're at it, we can combine the caching and disk read formulae to get the true effective access time for a system where there is only one level of cache:

$$\text{Effective Access Time} = h \times t_c + (1 - h)(p \times t_m + (1 - p) \times t_d)$$

This is good, but what is t_d? This is a measurable quantity so it is possible, of course, to just measure it[16]. We expect p to be large if our paging algorithm is any good. But what needs to take place to handle a page fault (miss) is [SGG13]:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Identify this interrupt as a page fault.
4. Check that the page reference was legal.
   (a) If so, determine the location of the page on disk.
   (b) If not, terminate the requesting program. Steps end here.
5. Figure out where to place the page in memory (use our replacement algorithm).
6. Is the frame we have selected currently filled with a page that has been modified?
   (a) If so, schedule a disk write to flush that page out to disk. The disk write request is placed in a queue.
   (b) If not, go to step 11.
7. Wait for the disk write to be executed. The CPU can do something else in the meantime, of course.
8. Receive an interrupt when the disk write has completed.
9. Save the registers and state of the other process if the CPU did something else.

[15] Some CPUs and operating systems have boundary issues and take this kind of thing a bit more seriously, requiring alignment of data structures to byte and block boundaries... others don't seem to mind at all.

[16] One of my favourite engineering sayings is "Don't guess; measure."

10. Update the page tables to reflect the flush of the replaced page to disk. Mark the destination frame as free.
11. Issue a disk read request to transfer the page to the free frame.
12. As before, while waiting, let the CPU do something else.
13. Receive an interrupt when the disk has completed the I/O request.
14. Save the registers and state of the other process if the CPU did something else.
15. Update the page tables to reflect the newly read page.
16. Restore the state of and resume execution of the process that encountered the page fault, restarting the instruction that was interrupted.

The slow step in all of this, is obviously, the amount of time it takes to load the page from disk. According to [SGG13], restarting the process and managing memory and such take something like 1 to 100 μs. A typical hard drive in their example has a latency of 3 ms, seek time (moving the read head of the disk to the location of the page) is around 5 ms, and a transfer time of 0.05 ms. So the latency plus seek time is the limiting component, and it's several orders of magnitude larger than any of the other costs in the system. And this is for servicing a request; don't forget that several requests may be queued, making the time even longer.

Thus the disk read term t_d dominates the effective access time equation. If memory access takes 200 ns and a disk read 8 ms, we can roughly estimate the access time in nanoseconds as $(1 - p) \times 8\ 000\ 000$.

If the page fault rate is high, performance is awful. If performance of the computer is to be reasonable, say around 10%, the page fault rate has to be very, very low. On the order of 10^{-6}.

And now you also know why solid state drives, SSDs, are such a huge performance increase for your computer. Instead of spending time measured in the milliseconds to find a page on disk, SSDs produce the data much quicker, with times that look more like memory reads.

We have not yet covered file systems, but files tend to come with a bunch of overhead for file creation and management. To avoid this, the system usually has a "swap file" which is just one giant file or partition of the hard drive. The system can get better performance by just dealing with the swap file as one big file (block) and not tiny individual files [SGG13].

Copy-on-Write

Recall that in UNIX we create a new process with a call to `fork()` and that this creates an exact duplicate of the parent process. That process may replace itself with the `exec()` system call immediately. Thus it seems like it might be a waste of time to copy the memory space of the parent if the child is just going to throw it away immediately. What UNIX systems do is use the *copy-on-write* technique: the parent and child share the same pages, but they are marked as copy-on-write [SGG13].

If there is immediately an `exec()` invocation then the new memory pages for the child are brought in and the unnecessary work is avoided. But suppose the child is to remain a clone of the parent.

Initially, all pages are shared but marked. As soon as either the parent or child attempts to modify a page, a copy of the page is made and the modification is then applied to the new copy of the page. All unmodified pages remain shared, but only the pages that are actually modified will be copied, so there is no unnecessary copying.

Some versions of UNIX, notably Solaris and Linux, have a system call `vfork()` which skips the copy-on-write procedure. The parent process is suspended and the child process uses the memory space of the parent [SGG13]. So any alterations the child makes will be visible to the parent when it resumes. Thus this is potentially dangerous, as the child can wreak a whole bunch of havoc. This is efficient if the intention is to immediately `exec()` and get a new memory space.

25 — Virtual Memory II

Virtual Memory, Continued

We will continue with our discussion of virtual memory by looking at a few advanced considerations in virtual memory.

Allocation of Frames

In a simple system where we do not do anything advanced, if there are n frames free in the system, we will demand page all of them. So the initial state is that all frames are empty, and as needed, pages are read into those frames. Once all n frames are filled with pages, page $n + 1$ must replace a page already in a frame (because there is no more space). When a process terminates, all its frames are marked as free. In theory, one process could fill all the frames in the system. This is as simple as it can be; from there we can build on it.

We might reserve a few pages to be free at all times for performance considerations. When we want to move a page into a frame, if all of the frames are full, we select a victim and write that victim out to disk if necessary. If we keep a few frames free, the newly-read page can be read into one of the free frames, and we can write the old page out to disk at a convenient time. The read does not need to wait for the write (flush of the old page) and therefore the user process gets to continue a bit sooner than it otherwise would.

Assuming, as we did with cache, that we might want to allocate different numbers of frames to different processes (and not necessarily let one run wild), we are constrained in the number of pages we can allocate. The maximum number of frames a process could have is the maximum number of frames in the system (obviously), but the minimum is more interesting.

The motivation behind allocating at least a minimum number of frames is ostensibly performance. As long as our page replacement algorithm is sane (i.e., not FIFO), adding more frames reduces the page fault rate. As we demonstrated earlier, a page fault is a huge performance decrease.

The absolute minimum number of frames is determined by the architecture of the system. Imagine a machine where a memory reference instruction may contain one memory address. In the worst case, the instruction and the address are in different pages, so we will need two frames to be able to complete this instruction. If the max frames for this process were 1, the instruction could never be completed. The IBM 370 `MVC` instruction is an extreme example: it moves data from one storage location to another. It takes six bytes, and the instruction itself can straddle two pages. That requires six frames. The worst case scenario is when the `MVC` instruction is the operand of an `EXECUTE` instruction that also straddles a page boundary. So eight frames are needed [SGG13].

In theory, the problem could be infinitely bad if the computer architecture allows referencing of an indirect address. The address being referenced could be on another page. That instruction could then reference another indirect address, and it's turtles all the way down. If that is the case, the entire virtual memory must be in main memory (so, not possible). The standard solution is to limit the levels of indirection to some manageable value, like 16. Rather like recursion leading to a stack overflow, it is possible that a stack overflow prevents a program that is correct from running, but more likely the program is in an infinite loop and it is better off terminated. If we limited the levels of indirection to 16, then the worst case scenario is a requirement of 17 pages [SGG13].

Assuming we do not allocate every process the minimum or maximum, there are a few allocation algorithms we

might follow. We already got a glimpse at this when we talked about local vs. global cache replacement.

If there are m frames in the system and the operating system reserves k of them for its own use, there are $m - k$ frames available for processes.

Equal Allocation. If there are n processes in the system, if we allocate them equally, each process gets $(m - k)/n$ frames. If this division produces a remainder, the leftover frames can be kept as a pool of free frames for performance purposes as above. There is an obvious flaw in this plan: why does a text editing program get the same amount of frames as a web browser and the same amount of frames as a game (which tends to be VERY demanding on memory).

Proportional Allocation. So how about proportional allocation: each process should get a share of the frames based on its needs. The strategy suggested in [SGG13] to do frame allocation runs something like this. Let the virtual memory size of a process p_i be defined as s_i. Thus $S = \Sigma s_i$. If the total number of frames in the system is, once again, m, and the operating system reserves k for itself, then we allocate a_i frames to a process p_i according to the following formula: $a_i = s_i/S \times (m - k)$.

This value of a_i is only an estimate. It may not divide evenly, and we can only allocate an integer number of frames to each process. So we will have to raise or lower a_i a bit to make it an integer. If a_i is below the minimum number of frames, then it needs to get bumped up to that minimum. We also must respect the limits of the system: the sum of all a_i values may not exceed the total frames of memory (minus the OS reserve), so a few of the larger processes may need to have their allocations nudged down.

Note that with proportional allocation, as with equal allocation, there is no regard given to the priorities of the processes. Normally, we would want to give more frames to higher priority processes so their page fault rates are lower and they can therefore execute more quickly. So we could modify the a_i values according to process priority, as well [SGG13]. The subject of priority is something we have not yet examined much yet, but will do so soon when we get to talking about scheduling; the next major topic. But, first, let's finish what we are doing with memory.

Local and Global Replacement. We already saw the idea of local replacement and global replacement in the cache. To refresh your memory, if we have chosen the LRU algorithm, in a global replacement policy, the least recently used page anywhere in the cache (memory) will be replaced. If we chose a local page replacement policy, it is the least recently used page that belongs to the current process.

Local is not affected much by our allocation concerns, so we can more or less leave that alone.

Suppose we choose global allocation. We will see the number of pages allocated to each change over time. If we assigned a colour to each process, it might be a bit like watching a little simulation of fictional countries as they fight over land, gaining and losing territory. Over time, a global replacement algorithm means processes that run more often will accumulate more pages in memory as they will acquire more pages than get taken from them.

It might make sense to watch to see how many frames each process has. It may be desirable to prevent a process from falling below the minimum number of pages, or to swap completely to disk a process that no longer meets that threshold.

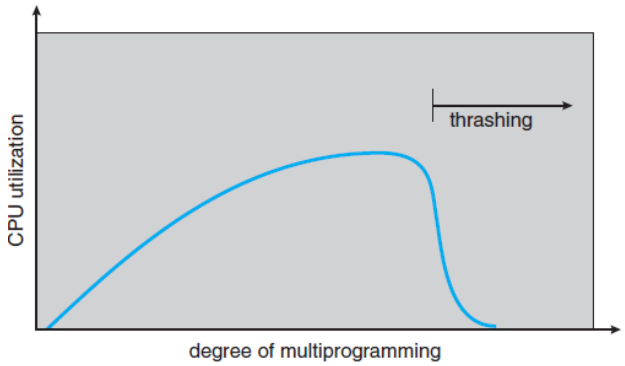
Thrashing

Thrashing received a little bit of mention in the introduction to virtual memory, but it deserves some further consideration. The quick definition of thrashing still applies: the operating system is spending so much time moving pages into and out of memory that no useful work gets done. Aside from intentionally depriving the system of RAM (as in the NeXTStep scenario), how can we get into this state, and how can we get out of it?

Let's consider a simple example from [SGG13]. In simple operating systems, the logic that controlled how many processes to run at a time would rely just on the CPU utilization. If CPU utilization is low, the CPU needs more work to do! Assign it more work by starting or bringing more processes into memory. The global page replacement policy is used here, so when a process gets a page fault, it takes a frame from another process. Under most circumstances, this works just fine.

This situation is not obviously wrong until one process starts to have a lot of page faults. This is not unreasonable; a compiler might be finished with reading and parsing the input files and moving to generation of binary code, for example, which requires a whole bunch of new instructions pages, plus the pages for the output. When this process does so, it starts taking pages from other processes. The victim processes need the pages they had, so when they get a turn to run, they too start generating page faults. So more and more requests are queued up for memory writes and reads, so the CPU is not very busy. Here's the fatal mistake: seeing that the CPU is not very busy, the OS schedules **more** programs to run.

A new process getting started will need at least the minimum number of pages to get started. These have to come from somewhere, so they will necessarily come from the pages currently belonging to other processes. This causes more page faults and more time spent paging and lower CPU utilization... prompting the OS to start more processes. No more work is getting done, because the system spends all its time moving pages into and out of memory, thrashing all around, acting like a maniac[17].



Thrashing [SGG13].

The solution is simple, actually; to increase CPU utilization we need to stop the thrashing which means we need fewer programs in memory at a time. What this really tells us is that CPU usage alone is not a sufficient indicator of whether more or fewer processes need to be running right now. It also matters *why* the CPU utilization is low. Another reason that might cause low CPU usage is, as you may recall, deadlock.

What if we stop using a global replacement policy and instead force local replacement? This limits some of the damage; if Process P_1 is thrashing,it cannot steal pages from P_2 and we do not get a cascade where all processes are constantly fighting over who has how many pages. But, if P_1 is spending all its time paging to and from disk, any other process that wants to use the disk (whether to swap or to interact with files) is going to have to spend more time waiting for the disk to be free.

We could decide whether we need to start or suspend programs not just based on CPU usage, but also on the number of page faults that occur in a given period of time. If there are too many page faults, it indicates we have too much going on. That is a reactive solution, however, and might only engage after a fair amount of time and effort has been wasted in thrashing. It would be better to be proactive and deal with this before thrashing has started.

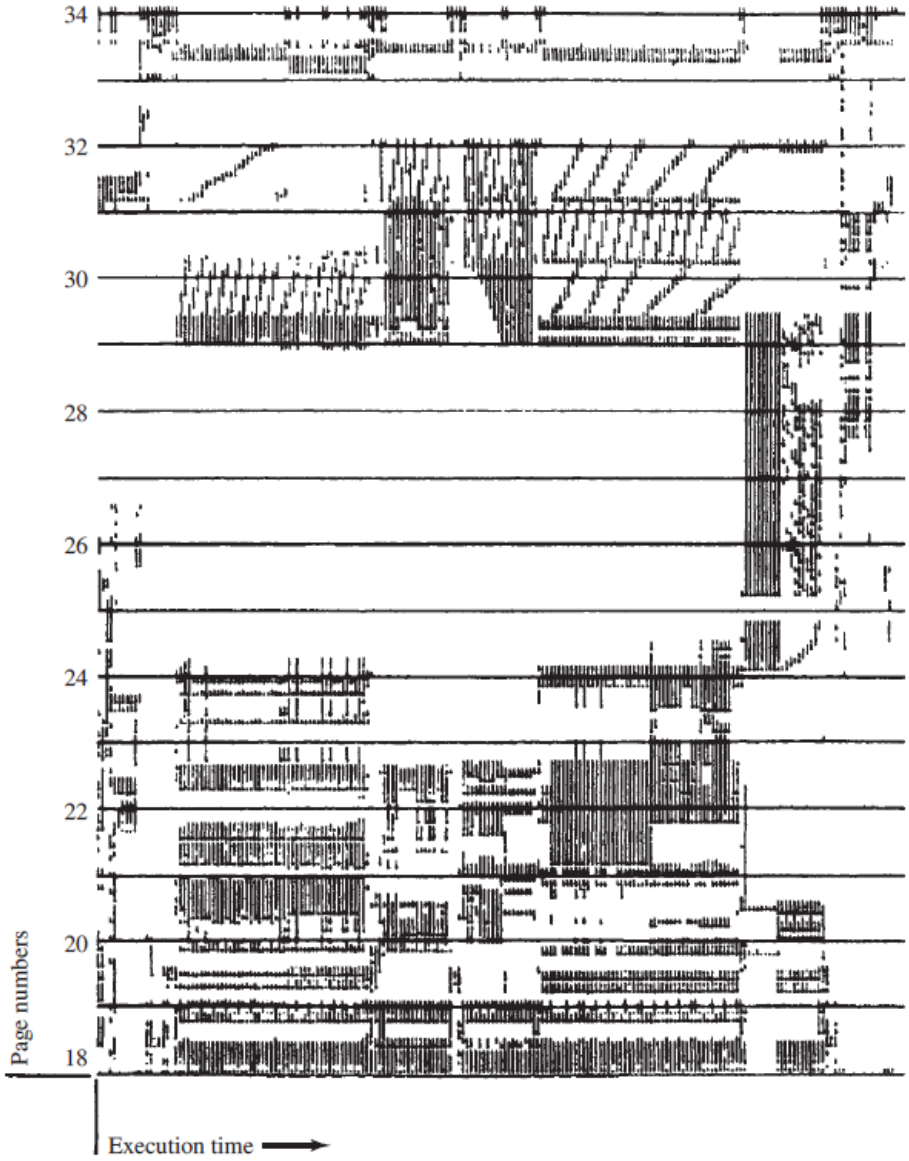
As is typical, a bit of clairvoyance would help here: how many pages is the process going to need and when will it need them? Unfortunately, there is no good way to know or to figure this out. So we will have to rely on (educated) guessing. You may hate the idea of guessing. As a general engineering practice, "let's do an experiment and find out" is good, provided you can analyze the result and if things go horribly wrong, no damage is done. Those last conditions make it kind of infeasible to do medical experiments just to see what would happen, or for NASA to play around with satellites in orbit. So in those cases, a simulation is probably the better approach. Even so, sometimes we don't have (or can't get) the data we need and we will therefore need to "make our best guess". But we are digressing.

What do we know about process memory accesses? They tend to obey the principle of locality, both temporal and spatial. Recall this from the discussion about caching. We could think of different parts of the program as different localities, as if they were areas on a map. And localities may overlap sometimes. So what we would like

[17] Whiplash! ... If this joke means nothing to you, it's also because I'm old.

to do, then, is give a process enough frames for its locality. If we do so, it can operate in this little area without encountering (too many) page faults. If we give insufficient pages, the process will be thrashing. Eventually it will leave the locality and that will result in some more page faults, but this is generally unavoidable [SGG13].

Before coming up with a solution that relies on locality, it is a good idea to check that the principle of locality holds. Fortunately we do not have to verify this for ourselves; someone else has already done so:



Graph showing the locality of memory accesses [**?**].

The take-away from this graph is that yes, locality is real. Given that, we can now move on and use it.

The Working-Set Model

A potential solution is the working-set model as described in [SGG13]. We retain the last n pages in memory as they represent the locale of the program. Assuming that most memory accesses are local, the most recent n pages will be the most frequently used. In the textbook descriptions, this n is usually called Δ,the working set window. Pages that have been recently used are in the working set. If a page has not been accessed recently, it will drop out of the working set after Δ time units since its last reference.

Suppose the window is defined as being ten accesses. Any page that was accessed in the last ten requests will then be considered part of the working set. If the next ten memory accesses are all in page k, then after those further

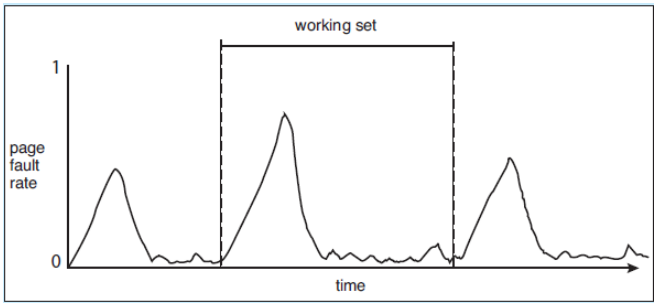
ten accesses, the working set will contain only k. So the size of the working set will change over time and can be anywhere from 1 to Δ pages.

If Δ is too small, the working set will not encompass the entire locale. If it is too large, it will cover multiple locales. Underestimating the size of the locale is bad, because it will mean more page faults being caused. Overestimating is also bad, because it means fewer processes will be allowed to run.

If the working set of every process is summed up, we will get the total number of frames each process would "like" to have. If this sum exceeds the $(m - k)$ available frames in the system, at least one process is going to be unhappy because it does not have as many frames as it will need. And like unhappy workers who go on strike, unhappy process start thrashing.

Once a value of Δ is determined, the OS will monitor the working set of each process and use that to figure out if the system is currently overloaded. If it is, the OS will pick a victim and suspend it to prevent thrashing. If the system is underloaded (frame supply exceeds demand), more processes can be started to run.

Now we should examine how this solution works. The page fault rate of a process tends to vary over time. At the beginning, there will be a bunch of page faults as the program starts up. Then once established in its first locale, the rate will drop. When it come time to move to a new locale, the page fault rate rises until the program is "settled" in that new locale. See the graphic below:



Here's an analogy. Imagine that you have moved to a new city for a co-op term. When you have just moved, you will frequently rely on Google Maps (or whatever other map program you like... or even, dare I say it, the paper ones?!) to find what you are looking for and how to get there. You need to buy groceries, so you ask Google for directions to the nearest grocery store of choice. Once you've been there, you know the way so you don't have to ask again. Not knowing where the grocery store is equals a page fault, and asking Google is like asking the operating system to bring in that page from disk. Once you know the way to the grocery store, it is part of your working set and you do not have to ask again... until your next co-op term when you move somewhere else.

26 — Uniprocessor Scheduling

Uniprocessor Scheduling

As you may imagine, processor scheduling can be very complex when we have multiple processes and multiple processors. Accordingly, to keep things simple, we will start with single processor systems (uniprocessor scheduling) and then build on that to look into multiprocessor scheduling.

Scheduling is easy to define: given that we have multiple processes in the system, we are going to have to make some decisions about when different programs run. There are four types of scheduling, but one of the four is I/O scheduling which we will come back to later on. The other three bear more examination [Sta14]:

1. **Long-Term Scheduling**
2. **Medium-Term Scheduling**
3. **Short Term Scheduling**

A student-relevant analogy to explain the different levels: Long-term scheduling is deciding what courses to take in your degree (e.g., do I want to take ECON 102 as a non-technical elective?). Medium-term scheduling is about deciding what courses you are taking in a given term (do I take ECON 102 in 2B or 4A?). Short term scheduling is deciding what course you are going to study right now (I'll study ECE 254 because the final exam is tomorrow!)

Long-Term Scheduling. The long term scheduler is responsible for determining which programs are going to run at all. It controls how many jobs the system is accepting to do at once. It controls the transition from the "new" state into the ready to run state (where it falls in the purview of the medium or short term scheduler). When a process exits, the long term scheduler might decide to admit a new one. Ah, but which one! The first? Based on some criteria?

Long term scheduling does not tend to happen too much on desktop computer systems. The user controls how many processes are running at a given time. If you choose to open 50 programs, performance may suffer, but the OS will not attempt to stop you.

Mind you, in rare occasions, there is some form of long term scheduling. If you are trying to log into a server, because you want to play Diablo III[18], you may receive an error saying the server is busy/full. This is the server telling you that it will not start a new game for you (a new process) because it has reached a limit of how many processes (games) are currently permitted. Only when some people log out will you be allowed to log in.

Mobile operating systems, notably Android, can be somewhat more aggressive. If memory is running low, Android can suspend and kill processes to make way for new ones.

Medium-Term Scheduling. Medium term scheduling is a lot more interesting than long term scheduling, and it revolves around swapping processes to and from disk. A process that is swapped to disk is not going to run in the immediate future, but it is at least likely to run before too long. Swapping a process to disk takes it out of the realm of the short term scheduler; swapping it in to memory returns it to that domain.

[18] Barbarian, for the curious.

Short-Term Scheduling. The short term scheduler, also sometimes called the dispatcher, is where the real action is. The medium and long term scheduler are all about someday and sometime and whenever. The short term scheduler is about "what are we going to do *right now*". The short term scheduler is going to run a lot, so it is very important. The short term scheduler will often run after certain things occur.

If we have co-operative multitasking, short term scheduling will only take place in one of two circumstances: the currently executing process yields the CPU or the currently executing process terminates (voluntarily or with an error). This is not how most operating systems work these days (though Mac OS 9 did follow this scheme, fortunately, its successor of Mac OS X does not). If the process does yield or terminate, then the short term scheduler will run.

For reasons that are presumably obvious (hint: some people are jerks), co-operative multitasking is problematic. What we will discuss from here on out is pre-emptive multitasking; the operating system, and not the running process, is responsible for deciding when it's time to switch processes. Some pre-emptive systems still have the concept of yield, and that is still an occasion to run the scheduler. Similarly, in pre-emptive systems, processes still terminate.

The dispatcher will certainly run when a process becomes blocked, such as on an I/O operation. If a process requested a write to the network and is blocked, now the short term scheduler needs to decide what process runs next. If it gets blocked on a semaphore or mutex, that is also an occasion to switch to another process. Page faults are also a great occasion to find something else to do.

Another time to make a scheduling decision is after handling an interrupt, whether from an I/O device or otherwise. After the interrupt is handled, we can return to execution exactly where we left off, or we can go somewhere else (the original process is suspended already so why not leave it in that state?).

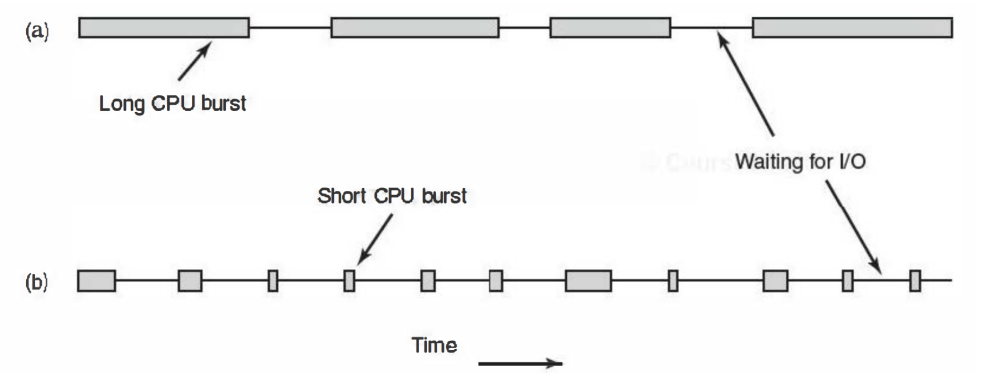
System calls like `fork` and even signalling on a semaphore may also provide good opportunities to switch from one process to another. Again, by invoking the operating system through the system call, the calling process is suspended (the trap handler runs and the OS takes over). So it is necessary to decide what process executes next. We acknowledged this in the discussion of `fork` by saying it was not known if the parent or child would execute next. In the case of semaphores, we do not even know which of the processes waiting on the semaphore will be the one to receive the signal, and even then, which of the signalling process and waiting process will resume.

Finally, there is also time slicing. If time slices are defined as t units, every t time units, there will be an interrupt generated by the clock. The whole purpose of the interrupt handler in that case is to run the short term scheduler to choose a process to run, so that different processes run (seemingly-)concurrently.

Process Behaviour: Rate Limiting Step

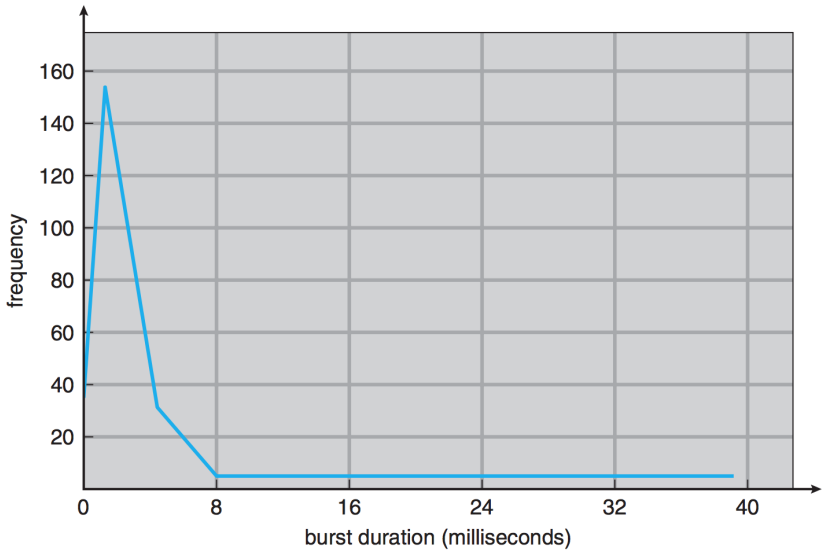
Processes tend to alternate periods of computing with input/output requests. These tend to alternate, so for a while the CPU does a lot of work, called a *CPU Burst*, and then it needs to do some I/O, and the period where it is waiting is called the *I/O Burst*. After the I/O is completed, the CPU can go at it again. How much of each a process does allows for classification of the process into one of two categories.

Processes that spend most of their time computing are called *CPU-Bound* and the limiting factor in their execution is how fast the CPU executes the code. Complex mathematical equations, for example, require a lot of CPU time. They can speed up significantly if you get a faster CPU. The alternative is a process that spends most of its time waiting for I/O, which we call *I/O-Bound*. In this case, having a faster CPU will not make the difference in how long it takes to execute the program. See the diagram below:



CPU usage diagram showing (a) a CPU-Bound process and (b) an I/O-Bound process [Tan08].

Studies have been done to examine the durations of CPU bursts. An I/O-Bound program will tend to have short CPU bursts, of course. Over time, CPUs have gotten faster at a rate much higher than the rate of I/O speedup. This is probably not surprising to you: a new CPU comes out every few months and they get faster. I/O standards like Serial-ATA and USB and such change very slowly, over the course of years. So it makes sense that over time programs tend towards being I/O-Bound. The results of the study are shown below:



CPU-Burst histogram [SGG13].

But what does this have to do with scheduling? There are two applications. The long term scheduler may attempt to keep a balance between CPU- and I/O-Bound tasks to get the best resource utilization. This requires that the long term scheduler have some idea about which processes are which.

Another is that if the disk is a common “pinch point”, when the disk has nothing to do, the short term scheduler should immediately schedule a process that is likely to issue a disk request, so the disk is kept as busy as possible.

Scheduling Criteria

The goals of scheduling depend significantly on the objectives of the system. If the system is supposed to respond to events within a certain period of time (real time system), this matters a lot to scheduling. Perhaps the goal is for the CPU to be used maximally (as in a supercomputer). Or maybe the most important thing is for users to feel like the system answers them quickly when they issue a command.

As usual when making a decision, we could just decide randomly. You may safely assume that just picking at random is not going to do well. So we should make an intelligent decision and we therefore need evaluation criteria.

We will examine the following scheduling criteria [Sta14]:

1. Turnaround time.
2. Response time.
3. Deadlines.
4. Predictability.
5. Throughput.
6. Processor utilization.
7. Fairness.
8. Enforcing priorities.
9. Balancing resources.

Turnaround Time. The turnaround time is the amount of time between starting a process and when it finishes. This is the execution time plus the amount of time waiting for resources (processor, I/O, etc). So, wall-clock time matters here. This is important to users; all things being equal the users would like their task to complete as fast as possible. If I ask the system to convert a file from FLAC format to mp3 format so I can import it to my mp3 player, I care about the turnaround time: how long will it take to convert this file?

Response Time. Assuming a process is not a background process (daemon), this is the time between putting in a request and getting some answers back. A process can often start producing output while still doing the request. If I am searching for a file on disk, while the search continues to run, any results found so far can be shown in the results list. Response time is very important to users; if they click a button and it takes a long time to acknowledge the command, users get frustrated and sometimes issue the command more than once. Like the time I accidentally bought two train tickets via an Android app[19].

Deadlines. This is more of a concern for real-time systems that have specific deadlines. If I am watching a movie with a Blu-Ray player, the player needs to read data from the disk, decrypt and decode it, and display it to the screen within a certain period of time, otherwise the video quality is degraded. Meeting the deadlines may be the most important thing in the system; much depends on the application.

Predictability. It is desirable if a given job runs in a fairly consistent amount of time. If a task normally takes x minutes and right now it is taking $2x$ minutes, somewhere between x and $2x$ minutes, the user may fear the process is stuck and might terminate it. Users are very impatient.

Throughput. Scheduling policy should try to maximize the number of processes that complete in a given amount of time. This is our way of figuring out how much work is being accomplished. Much will depend here on the nature of the processes, but we would always like to get more done in the same amount of time. If I am in line to renew my drivers' license, if the Service Ontario office has a high throughput, more people will get their requests finished in the day. If it has a low throughput, I may be waiting there all day. Which would never happen. Oh no.

Processor Utilization. This is, obviously, how much of the time the CPU is busy. As already mentioned, supercomputers are really expensive to build and maintain. Processing time on supercomputers is sold (leased?) to various organizations like research labs and universities. Any time the supercomputer is not in use is wasted. So we may give a lot of priority to keeping it busy.

Fairness. Who could be against fairness? Humans have a lot of inbuilt notions about fairness, but we are not necessarily expecting a perfectly fair routine for scheduling. What we do expect, however, is that processes should get at least a basic level of fairness, e.g., no starvation. There is a story, possibly apocryphal, about a system being shut down in 1972 that had a yet-unexecuted batch job that had been submitted some five years earlier. I think we can safely call that unfair.

[19] Fortunately, this was a single ride ticket, so it was only a 5 Euro mistake and not a monthly ticket which could have easily been a 100+ Euro mistake.

Priorities. Processes can be assigned priorities: a way of saying that this process is more or less important compared to others. Scheduling should respect this, within reasonable limits. The highest priority process should probably run most often, but it should not always be the one selected, as that would violate the principle of fairness.

Balancing Resources. We will get a better outcome if we balance resource usage. The example mentioned above applies: we would like to choose some CPU-Bound and I/O-Bound processes so both the CPU and I/O are kept busy. If we have more information about what sort of I/O the processes will need, so much the better: we can choose a process that will use the printer if run, when we know the printer is idle, and avoid choosing a process that will use the network if we know the network is busy.

Scheduling Algorithm Goals

The priorities of these different goals depend significantly on the kind of system it is. The following list gives some idea about what is important in different kinds of systems [Tan08]:

All Systems.

- Fairness
- Priorities
- Balancing Resources

Batch Systems.

- Throughput
- Turnaround time
- CPU Utilization

Interactive Systems.

- Response time.
- Predictability.

The (Ab)use of Priorities

Okay, so we've said that processes have priorities and we've repeatedly referenced priority as being an important consideration in various operations like scheduling and page allocation and so on. We have not yet actually talked much about priorities.

Each process's priority is typically an integer. Whether higher numbers are higher priority or lower numbers are higher priority is a question of system design. As is typical, if there are two ways to do something, some people will choose one way and some will choose the other. In UNIX, a lower number is higher priority; Windows is the opposite. Picking a convention and sticking to it would be nice, but there we are.

With a priority value assigned, it can be used to make decisions about resources. If P_1 wants resource R_1 and P_2 wants that same resource, where the priority of P_1 is greater than that of P_2, choose to assign the resource to P_1.

The operating system or the program author may be responsible for assigning a priority to each of the processes. These priorities may change over time with various criteria. System administrators can change priorities, usually.

Users may have a say in priority, in at least a limited way. In Windows, for example, as a user it may be possible to set a task to a higher or lower priority. Giving this to users was probably a bad idea, because users often do it

wrong. If you have a long, CPU-Bound task, the right thing to do is to give it a low priority and not a high one. You might expect that the high priority will get the task done faster, but it kills the performance of the system and makes users unhappy (even though that's what they wanted).

In some systems, the highest priority non-blocked process will always run. This is a great way of making sure that higher priority processes have right-of-way, but a terrible way of ensuring fairness and preventing starvation. We could easily have a situation where low priority processes never get a chance to run, because there are too many processes of middle or high priority. So, scheduling is not necessarily as easy as just finding the highest priority thing to do...

27 — Scheduling Algorithms

Scheduling Algorithms

Earlier we saw one example of a simple scheduling algorithm: always choose the highest priority (non-blocked) task, and execute it. There are many other choices we could make. We will examine the following options:

1. Highest Priority, Period
2. First-Come, First-Served
3. Round Robin
4. Shortest Process Next
5. Shortest Job First
6. Smallest Remaining Time
7. Highest Response Ratio Next
8. Multilevel Queue (Feedback)
9. Guaranteed Scheduling
10. Lottery

The OS may maintain data about each process to be used in making decisions about scheduling [Sta14]:

1. The time spent waiting to run.
2. The time spent executing.
3. The total time of execution.

The first two criteria in the list can easily be measured, but the third one must be estimated or supplied by the user. Old batch systems did sometimes ask users to provide an estimate of how long they thought a task was likely to take. If the user underestimated, then execution would be halted at the time they said, even if his or her operation was unfinished. But the higher the estimate, the lower the priority the task received; if too high the user task might never be scheduled to run at all. Operating systems do not ask the users in desktop systems to estimate how long a process will run, but supercomputers and other batch jobs may still consider this as a criterion.

Highest Priority, Period

Implementing this is not difficult; we could have some priority queues (one for each priority level). If a task is not blocked, put it in its appropriate priority queue. If the process's priority is changed (manually or otherwise), move it to its new home. We might have a priority heap, or just one big linked list or array that we keep sorted by priority. Many options.

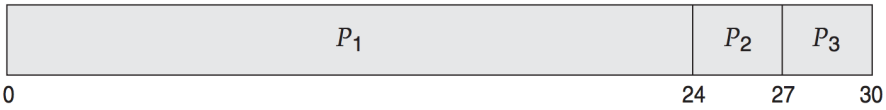
The flaw in this has already been identified: it is vulnerable to starvation. A process of relatively low priority may never get the chance to run, because there is always something better to do right now. Humans presumably encounter situations like this all the time. Software projects may have bug reports open for years on end because they are never important enough to be addressed. In my personal life, there are books I'd like to read (the example "Der Deutsch-Französische Krieg 1870/71" springs to mind), but I never get to them because something else always comes up... work to do, another book to read, a social event...

In some systems this is a desirable property. It does not fulfill all short term scheduling criteria, notably response time and fairness. It may be suitable for life-and-safety-critical systems, such as the process to control a robot arm, to prevent a situation where the robot arm goes through the wall and the building falls down and you're dead.

First-Come, First-Served

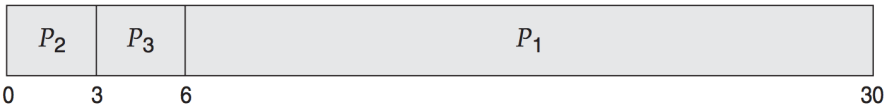
This is an obvious algorithm that is simple to implement. Whichever process requests the CPU first gets the CPU first. Just imagine a queue of processes in which all processes are equal. A process enters the queue at the back and whichever process is at the front will be dequeued and get to run. If the current process finishes or is blocked for some reason, the next ready process is selected. This is actually a simplification of the highest priority, period scheme, because it ignores priority altogether. All processes will get a chance to run eventually, so low priority processes will not starve.

FCFS can result in some undesirable outcomes. The average waiting time for processes might vary wildly. Consider if we have three processes, P_1, P_2, and P_3. Let us say that P_1 needs 24 units of CPU time; P_2 and P_3 require 3 units each.



First-Come, First-Served Scheduling for P_1, P_2, and P_3 [SGG13].

The total time needed to complete all processes is 30 units. The average completion time, however, is $(24 + 27 + 30)/3 = 27$ units. Nothing wrong with this. Suppose, however, the processes arrived in a slightly different order:



P_1, P_2, and P_3 arriving in a different order [SGG13].

If P_1 arrived last instead, then the picture is somewhat different. It still takes 30 units of time to complete all processes, but the average completion time is $(3 + 6 + 30)/3) = 13$ units. Now this might make a big difference to the user: seeing processes 2 and 3 completed sooner will probably be viewed positively. Still, we consider it an undesirable thing that we have such a dramatic difference in how long P_2's turnaround time is.

FCFS as a scheduling algorithm also tends to favour CPU-Bound processes rather than I/O-Bound processes. When a CPU-Bound process is running, the I/O bound processes must wait in the queue like everybody else. This might lead to inefficient use of the resources; disk is slow so we would like to keep it busy at all times. If a disk write is completed and there is a process that would like to read something from disk, it would be ideal to issue that read straight away so the disk is constantly in use. With FCFS, however, the I/O devices are likely to suffer idle periods [Sta14].

The FCFS algorithm as it is generally described, is non-preemptive; a process that gets selected from the front of the queue runs until there is a reason to make the swap. So in theory, one process could monopolize the CPU (remember that some people are jerks). If we modify FCFS with periodic preemption, then we get a new system, Round Robin.

Round Robin

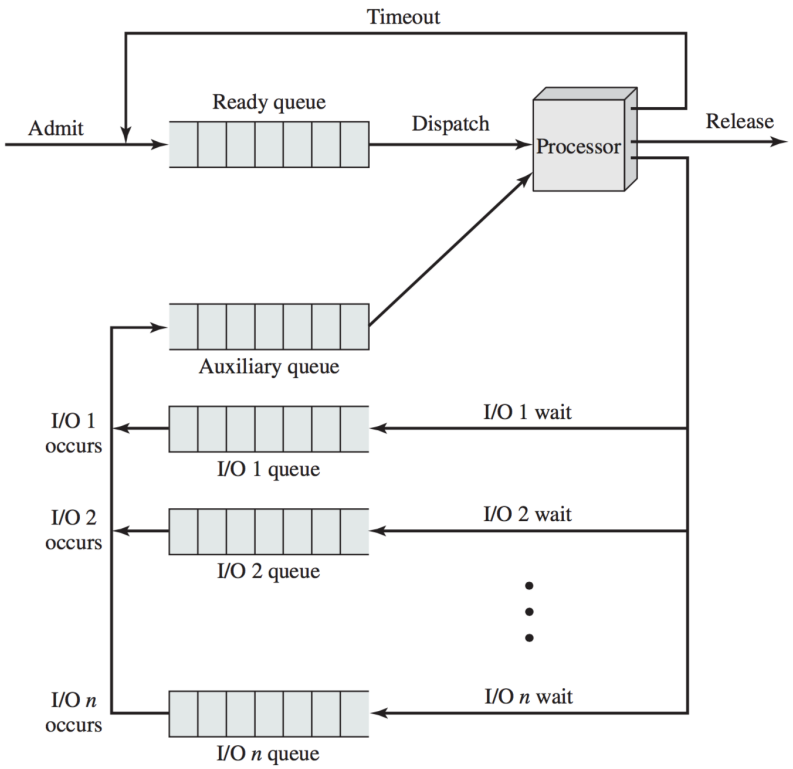
The idea of time slicing has already been introduced: every t units of time, a timer generates an interrupt that is the prompt to run the short-term scheduler. Time slicing itself can be combined with many of the strategies for choosing the next process, but when it is combined with FCFS we get Round Robin.

The principal issue is: how big should t be? If t is too long, then processes may seem to be unresponsive while some other process has the CPU, or short processes may have to wait quite a while for their turn. If t is too short, the system spends a lot of time handling the clock interrupt and running the scheduling algorithm and not a lot of time actually executing the processes [Sta14].

On the principle of "don't guess; measure" we could make a decision about the size of t based on the patterns of the system. If we know that the typical process tends to run for r units of time before getting blocked on I/O or something, then it would be logical to choose t such that it is slightly larger than r. If t is smaller than r, then processes will frequently be interrupted by the time slice. Processes that are going to use a lot of CPU will be split up over multiple time slices anyways, but it's frustrating if the process would take 1.1 time slices. If t is larger than r, many processes will not run up against that time slice limit and will hopefully accomplish a useful chunk of work before getting blocked for I/O or some other reason.

Thinking about this further, Round Robin tends to favour CPU-Bound processes. An I/O-Bound process spends a lot less time using the CPU. It runs for a short time, gets blocked on I/O, then when the I/O is finished, it goes back in the ready queue. So CPU-Bound processes are getting more of the CPU time.

Round Robin can be improved to Virtual Round Robin to address this unfairness. It works like Round Robin, but a process that gets unblocked after I/O gets higher priority. Instead of just rejoining the general queue, there is an auxiliary queue for processes that were previously blocked on I/O. When the scheduler is choosing a process to run, it takes them from the auxiliary queue if possible. If a process simply ran up against the time limit, it goes into the regular ready queue instead. If dispatched from the auxiliary queue, it runs for up to the as-yet-unused fraction of a slice [HS91].



Queueing diagram for the Virtual Round Robin scheduling approach [Sta14].

Shortest Process Next

If some information is available about the total length of execution then we may wish to give priority to short processes. We let short processes go first to get them over with quickly. The reason is made clear by the second example of FCFS from earlier; though it still took 30 units of time, total, the average time to completion of a task was a lot lower when the shorter processes of P_2 and P_3 ran before P_1. This means we will get faster turnaround times and better responsiveness, but longer processes may be waiting an unpredictable amount of time.

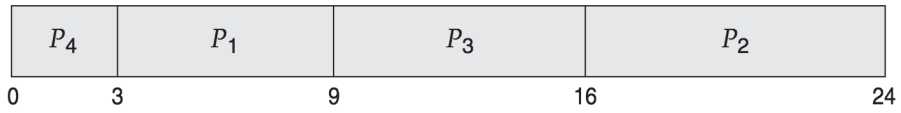
This is the sort of thing that used to happen in batch job processing on mainframes. The programmer was asked to give an estimate of the amount of time the program (e.g., compile) would run. If the programmer's estimate was too low, the program execution would be terminated early. If the programmer's estimate was too high, the job may never be scheduled to run, or at least, have to wait a very long time.

This might work, but you have noticed that the OS does not ask you, when you start a text editor, roughly how long you expect to be. Your boss on co-op may not be so accommodating. This is also a global assessment: how long the whole process takes. It might be better to worry about the length of CPU bursts and make decisions somewhat more locally...

Shortest Job First

This strategy, unfortunately, is not given quite the right name, but it's the common name for the scheme. We should call it "shortest next CPU burst", but then again, the other sciences are not so good at naming things either... the Red Panda is not a Panda at all.

Right, naming aside, the goal is to choose the process that is likely to have the smallest CPU burst. If two are the same, then FCFS can break the tie (or just choose randomly). If we have 4 processes, P_1 through P_4, whose predicted burst times are 6, 8, 7, and 3 respectively, we should schedule then such that the order would be P_4, P_1, P_3, P_2:



Scheduling the jobs with the shortest CPU burst first [SGG13].

The average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ units of time. This is significantly better than the FCFS scheduling. In fact, the algorithm is provably optimal in giving the minimum average waiting time for processes. Moving a short process up means it finishes faster, and that decreases its waiting time, while moving longer processes back increases their waiting time. Overall, this scheme is a net positive because the decreases outweigh the increases [SGG13].

The problem is predicting the CPU burst times. The best thing we may be able to do is gather information about the past and use that to guess about the future. The formula in [Sta14]:

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^{n} T_i$$

where: T_i is the burst time for the ith instance of this process; S_i is the predicted value for the ith instance; and S_1 is a guess at the first value (not predicted, because we have no data to go on at that point). Instead of picturing it as a sum each time, we can modify the equation to just update the value:

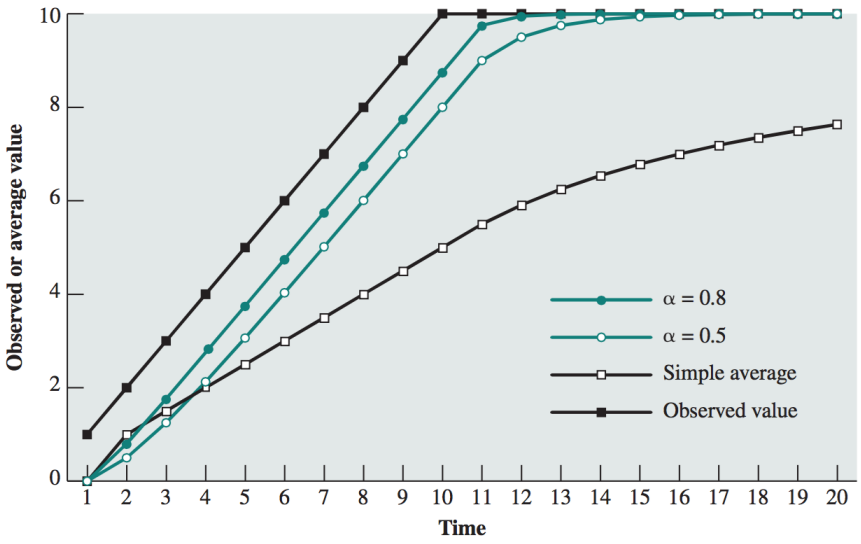
$$S_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

This routine, as you may have guessed, gives each term in the summation equal weight of $\frac{1}{n}$. What we would like to do is give greater weight to the more recent values. The strategy for doing so, as in [Sta14] is exponential

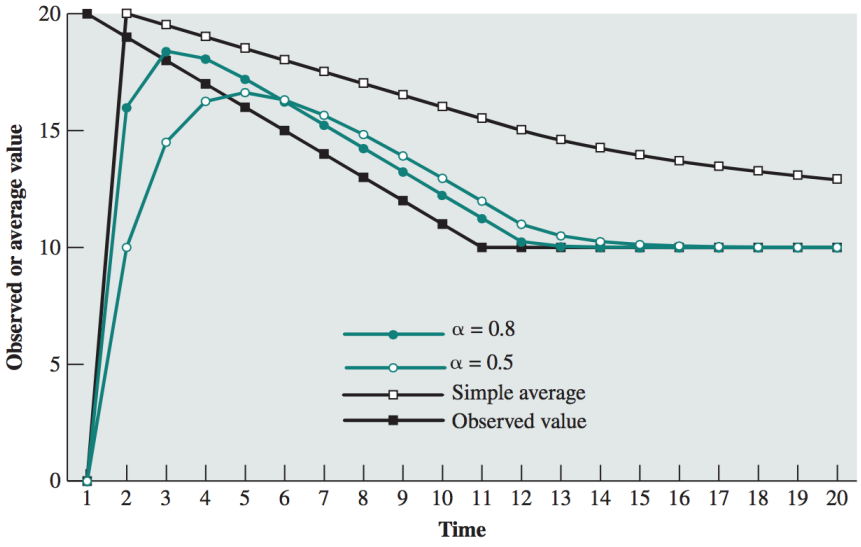
averaging. We define a weighting factor α, somewhere between 0 and 1, that determines how much weight the observations are given:

$$S_{n+1} = \alpha T_n + (1 - \alpha) S_n$$

Thus, the older the observation, the less it is counted in the average. The larger the value of α, the more the recent observations matter. The diagram below compares simple averaging (no use of α to age the data) against the values of α of 0.8 and 0.5. The case of $\alpha = 0.8$ assigns significantly more weight to the most recent 4 events than the others.



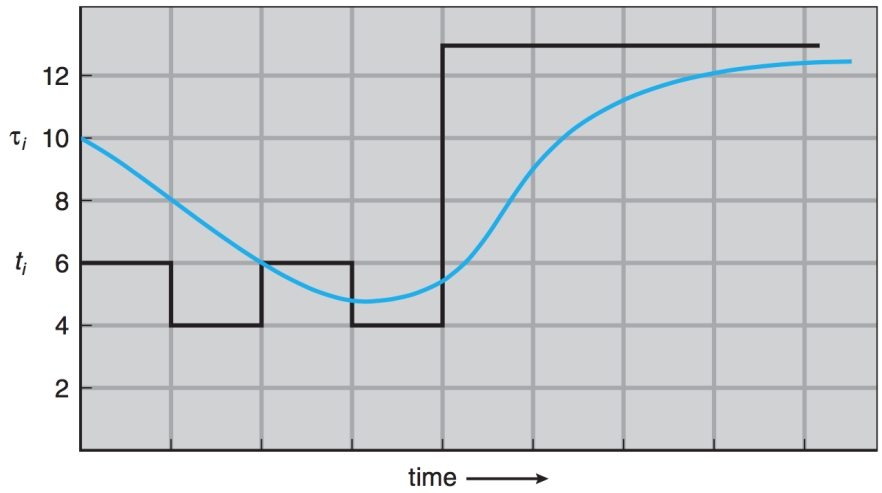
(a) Increasing function



(b) Decreasing function

Use of Exponential Averaging [Sta14].

We might give an estimate of $S_1 = 0$ to start with, which gives new processes priority and a chance to get started. After they have started and run a bit, we will have some data. But first we have to give them a chance.



Predictions (τ_i) and actual CPU burst (t_i) example [SGG13].

There is still a chance that longer processes will starve; if there is a constant stream of shorter processes, they will continue to get scheduled ahead of a long one, causing the long one to wait unreasonably long...

Shortest Remaining Time

Shortest remaining time is a modification of the preceding strategy, which allows for some additional preemption. When a new process is scheduled or an old one becomes unblocked, the scheduler will evaluate if it has a shorter predicted running time than the currently-executing process. If so, then the new (or unblocked) process will displace the currently-executing one and start running right away.

As with Shortest Job First, there is a chance that long processes will starve because of a steady stream of shorter processes. If we choose S_1 to be zero for new processes, it means they will always preempt the running process. This may or may not be desirable.

One advantage we have in choosing SRT is that it means we no longer need to have time slicing. Instead of interrupting the running process every t units of time, the other interrupts (users launching programs, hardware operations completed, etc) will be the prompts to run the scheduler. Thus, the system does not spend any time handling the clock interrupts, which will be a performance increase. Handling the clock interrupt is not expensive, but even an inexpensive operation, done a million times, will eventually add up... [Sta14]

Batch and Interactive

So far the scheduling routines we have looked at are more suitable to batch processing systems (mainframes, supercomputers, etc.) than to interactive desktop systems. The remaining scheduling algorithms in the list will look a lot more like what we expect to see on our laptops and phones...

28 — Scheduling: Idling, Priorities, Multiprocessor

Scheduling Algorithms, Continued

Carrying on from last time, we will examine some more scheduling algorithms.

Highest Response Ratio Next

We will introduce a new measure: normalized turnaround time. This is the ratio of the turnaround time (the time waiting plus the amount of time taken to execute) to the service time (the time it takes to execute). The relative amount of time waiting is somewhat more important; we can tolerate longer processes waiting a comparatively longer period of time. The goal, then, of the HRRN strategy, is to minimize not only the normalized turnaround time for each process, but to minimize the average over all processes [Sta14].

The way to calculate the response ratio, R, is by the following formula: $\frac{w+s}{s}$ where w is the waiting time and s is the service time. The service time is, as always, a guess. When it is time to select the next process to run, choose the process with the highest R value. A new process will have a value of 1.0 at the beginning, because it has spent no time waiting (yet). Thus it is not that likely to get selected.

Jobs with a small s, i.e., short jobs, are likely to get scheduled quickly, which is still a positive. In spite of this, the HRRN approach introduces something important we have not yet had: factoring in the age of the process. The term w indicates how much time a process has spent waiting. Thus, a process that has spent a long time waiting will rise in priority, over time, until it gets a turn. So processes will not starve, because even a process that is expected to have a very long s will eventually have a high enough R due to the growth of w.

We still need to have some way of estimating s, which may or may not be simple guessing.

Multilevel Queue (Feedback)

For the most part, until now, we have treated processes more or less equally (except when we have taken the highest priority process). While it might seem very fair, it may not be ideal for a situation where processes behave differently. A desktop or laptop has many processes running, some of which are visible to the user (foreground, or interactive processes) and some of which are not (background and batch processes). We could then apply different scheduling algorithms to different types of process.

The multilevel queue takes the ready queue and breaks it up into several queues. A process can be in one (but only one) of the queues and it is assigned to the queue based on some attribute of the process (priority, memory needs, foreground/background, et cetera). The foreground queue, for example, could be scheduled by Round Robin, and the background by First-Come, First-Served [SGG13].

When there are multiple queues, we also need a way of choosing which of the queues to take from next. The policy we choose depends on goals. We might say some queues have absolute priority over others (e.g., as in the earlier highest priority, period option), or we might have time slicing amongst the queues. This could be balanced evenly

(rotate through each) or give more time slices to some queues at the expense of others.

An example of this was the CTSS (Compatible TimeSharing System) that ran on the IBM 7094. The CTSS designers decided that it was ideal to give CPU-Bound processes longer blocks of time to execute so they would not have to spend so much time swapping. They set up multiple queues. In the highest priority class, a process got 1 time slice; in the next one down, a process got 2 time slices; the third class meant 4 time slices, and so on. If a process ran up against the limit of a time slice (e.g., used the full 2 time slices), it was moved down a class. So it got a lower priority, but when it did get selected to run, it was able to run with a lower chance of being interrupted [Tan08].

Like a few schemes, we have seen so far, this is a ratchet: a process can move down in the priority list, but there does not appear to be a way for it to move up. So a process that needed a lot of CPU early on was going to be punished "forever". The designers of the system assumed that if the user pressed the Enter key, it might be a sign the process was likely to become interactive (and therefore should move up in priority). Some genius user (there's always one), figured out that by pressing the enter button repeatedly, his long running processes would finish faster. This was a bit unfair; his processes got priority over the others. But things really broke down when this individual decided to be nice: he told all his friends. And suddenly everyone was doing it and the benefit of the system was lost [Tan08].

This scheduling algorithm may also be referred to as *feedback*. We do not have any information in advance about how long various processes will be. Instead of being concerned with how much CPU time will be used, which requires clairvoyance or guessing, we assign priority based on the amount of CPU time assigned so far. A process that has used a lot of CPU so far gets lower priority.

Guaranteed Scheduling

And now for something completely different. The idea behind guaranteed scheduling is to promise the users something and then fulfill that promise. We could promise that if there are n users, each gets an equal share ($1/n$) of the CPU time. Or with m processes, each process gets $1/m$ of the CPU time.

To make this happen, the system must keep track of how much CPU time each process has received since its creation. It then considers the how this value compares to the ideal (time since creation divided by n). If a process has a value of 0.5, it means it has had only half the CPU it "should" have received. If it has a value of 2.0, it has had double. So the scheduling algorithm is then to run the process with the lowest score, trying to keep all values as close to 1.0 as we can [Tan08].

Lottery

The lottery is a system to give predictable results with a simple implementation. The premise is that every process gets some number of "lottery tickets" for each resource (e.g., CPU). When a decision has to be made, a lottery ticket is selected at random. The process that holds that ticket gets that resource. This system provides some clarity; if a process has priority p, what does that mean? If a process has a fraction f of the total tickets, then we can expect that process to get about f% of the resource. When a process is created or terminates this may increase or decrease the number of tickets, or result in their redistribution [Tan08].

More important processes are given more tickets and have, therefore, a higher chance of winning. If there are 100 tickets outstanding, if a process has 25 of them, it has a 25% chance of winning any given draw. To increase a process's chance of winning, give it more tickets. To decrease it, give it fewer. Unlike in the real lottery, though, there is always a winner. There are no "unpurchased" tickets and we can choose only a ticket that someone is holding.

Co-operating processes may be permitted to exchange tickets. A client that sends a request to a server might then give all its tickets to the server, increasing the chance the server gets the resources to run next [Tan08].

This is a lot less overhead than guaranteed scheduling. We do not have to keep track of how much of the resource a process has received. Assuming that the lottery system is sufficiently random, over time the resource allocation will tend towards the proportions of the tickets each process holds. If process A has 20% of the tickets, B has 30%, and C has 50%, then the CPU will be given to the processes in approximately a 20:30:50 ratio, as expected. Of course, random number generation is a small struggle for computers, though the complexity of this is not

something we want to examine here.

The Idle Task

Sometimes our scheduling algorithm cannot produce a new process to run next because there is, quite frankly, nothing to do. The actual implementation of the idle thread may vary across different systems. In some cases it is just repeatedly invoking the scheduler; in others it does a bit of useless addition; or it might just be a whole bunch of `NOP` instructions. With truly nothing to do, the CPU can be told to halt or switch to a lower power state. Whatever it actually "does", the idle thread never has any dependencies on anything else and is always ready to run.

Since the idle task does not necessarily do much, why have it? It prevents having special cases in the scheduler, first of all. It also provides some accounting information about how much of the time the CPU is not doing anything. In fact, a lot of the time on the desktop or laptop, task manager will tell you that "System Idle Process" is taking up a large percentage of the CPU. Because you are more intelligent than a certain technology journalist who may or may not be named John Dvorak, you will recognize that this just means the CPU is not doing anything; it does not mean that some mysterious system process is using up all your CPU's time.

Saving power by shutting down (parts of) the processor seems like a nice savings of energy (and potentially increases battery life). On the other hand, time when the CPU is doing nothing might potentially be put to use. There are usually some accounting and housekeeping tasks that the CPU can be doing when it has nothing else. For example, the OS could collect statistical data, or defragment the hard drive (something we will look at).

Bumping the Priority

Sometimes we get into a situation called a *priority inversion*. This is what happens when a high priority process is waiting for a low priority process. Suppose that P_1 is high priority and is blocked on a semaphore, while P_2 is in its critical section. As P_2 is low priority, it might be a long time before P_2 is selected again to run and can finish and exit the critical section. So P_1 cannot run, because it is blocked, and it could be blocked for a very long time. In the meantime, other processes with lower priority than P_1 (but higher than P_2) carry on execution. Having P_1 waiting for the lower priority processes is rather undesirable.

The solution is *priority inheritance*. The right thing to do is to bump up the priority of P_2, temporarily, to be equal to that of P_1, so that P_1 can be unblocked as quickly as possible. To generalize, a lower priority process should inherit the higher priority if a higher priority process is waiting for a resource the lower priority process holds. So P_2 will get selected, will execute and exit the critical section. Its priority then falls down to normal, meaning P_1 will be selected and may continue.

A famous case of priority inversion took place on the Mars Pathfinder rover. In short, a low priority task would acquire a semaphore, locking the information bus. A high priority task also needed that bus. There was, finally, a medium priority task for communication, which ran for a long time. If the low priority task had the bus, the high priority task was blocked; the medium priority task would be selected to run. While that happened, the low priority task could not finish and release the semaphore. Accordingly, the high priority task was stuck waiting. After a specified period of time, the system assumed the high priority task was stuck and its deadlock resolution strategy was armageddon: total system reboot (resulting in the loss of some data).

The Pathfinder solution was to enable priority inheritance on the information bus semaphore. This meant that the low priority task would get a higher priority than the communication task and would run to completion; then the high priority task would be able to run and the system would not assume a deadlock had occurred. It was, to a certain extent, fortunate that this option was built into the system at all and needed only to be turned on. Doing a major software update on a system that is located *on another planet* is not quite the same as installing Windows updates on Patch Tuesday.

Multiprocessor Scheduling

If you thought scheduling for a single processor was complicated enough, well, things are about to get exponentially harder. When we have more than one processor working on things at a time, then the complexity increases dramatically.

We can classify multiprocessor systems into three major buckets [Sta14]:

1. **Distributed**. We have a collection of relatively autonomous systems who interact. For more about this, take the class distributed systems.
2. **Functionally Specialized**. The system has lots of specialized chips working on their specific area (but we'll come back to this when we talk about I/O scheduling).
3. **Tightly Coupled**. A set of processors that share a common main memory and are under the control of the operating system. This is the kind we're most familiar with and going to examine here.

Then we have to worry about the interactions of various processes. Specifically, how often they plan to interact. See this table (from [Sta14] again) that provides an overview of the granularities:

Grain Size	**Description**	**Interval (Instructions)**
Fine	Single instruction stream	< 20
Medium	Single application	$20 - 200$
Coarse	Multiple processes	$200 - 2000$
Very Coarse	Distributed computing	$2000 - 1\text{M}$
Independent	Unrelated processes	N/A

To sum it all up, the finer-grained the parallelism, the more care and attention needs to be given to how we are going to schedule a process in a multiprocessor system. If the processes are totally independent, then there is not too much to worry about; if we are taking a single process's thread and doing different instructions on different CPUs, then we have to be very careful to make sure that the execution is correct.

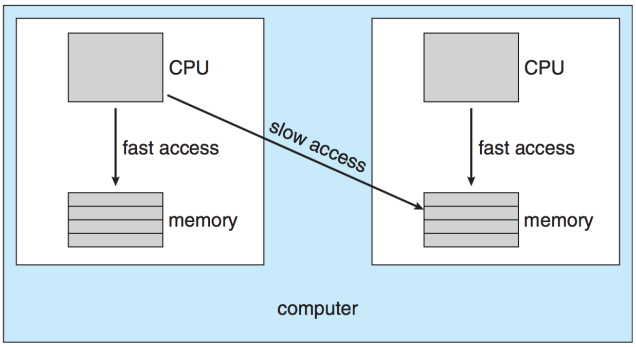
If we choose asymmetric multiprocessing, we have a boss processor and this one alone is responsible for assigning work and managing the kernel data structures (e.g., process control blocks). If instead, the system uses symmetric multiprocessing, each processor is responsible for scheduling itself. We will need to make use of mutual exclusion and other synchronization techniques in the kernel to prevent errors in managing the execution of processes. We do not want to have two processors trying to dequeue from the ready queue at the same time, after all.

Processor Affinity

Let us imagine that every processor has its own cache (e.g., the L1, L2, & L3 caches). If that is the case, then we want to have *processor affinity*. After some period of time of executing on this processor, a process will have a bunch of its data in the cache of that processor. If the process begins executing on another processor, all the data is in the "wrong" cache and there will be a lot more cache misses (which will slow down execution). Ideally, then we will keep executing on the same processor, wherever possible. This desire to stick with a certain processor is called processor affinity.

If the OS is just going to make an effort but not guarantee that a process runs on a given processor, that is called *soft affinity*. A process can move from one processor to another, but will not do so if it can avoid it. The alternative is *hard affinity*: a process will only run on a specified processor (or set of processors). Linux, for example, has both soft and hard affinity [SGG13].

Another motivation why we might want to lock a process to a particular processor occurs when memory accesses are non-uniform. For the most part we assume that any memory read takes as much time as any other, and if we have one bus connecting the CPU to all of main memory, that is a safe assumption. If the CPU can access some parts of memory faster than others, the system has *non-uniform memory access* (NUMA). See the diagram below:



A system with Non-Uniform Memory Access (NUMA) times [SGG13].

If we have this situation, then our choice of processor should be based on where the memory of the process is located. The memory allocation routine should also pay attention to where to allocate memory requests, preferring to keep the program code together in memory. If there is data in one of the other blocks of memory, it does not mean game over, but it means slower execution.

Load Balancing

It is presumably obvious that if we have 4 processors, it is less than ideal to have one processor at 100% utilization and 3 processors sitting around doing nothing. We want to keep the workload balanced between all the different systems. The process for this is *load balancing*.

Load balancing is typically necessary only where each processor has its own private queue of processes to run (the "grocery store queue" model); if there is a common ready queue (the "bank queue" model) then load balancing will tend to happen all on its own, as a processor with nothing to do will simply take the next process from the queue. But in most of the modern operating systems we are familiar with, each processor does have a private queue, so we need to do load balancing [SGG13].

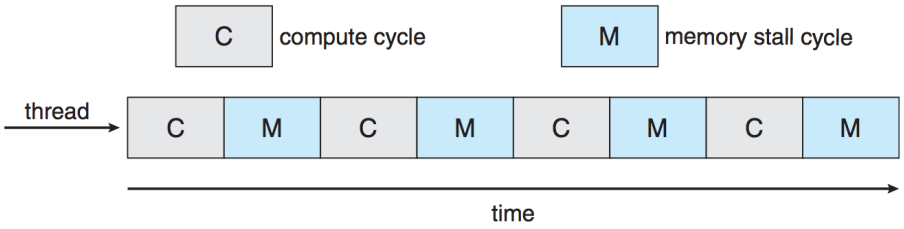
There are two, non-exclusive approaches to redistributing the load: *push* and *pull* migration. It is called migration because a process migrates from one processor to another (it moves homes). If there is push migration, a task periodically checks how busy each processor is and then moves processes around to balance things out (to within some tolerance). Pull migration is when a processor with nothing to do "steals" a process from the queue of a busy processor. The Linux and FreeBSD schedulers, for example, use both [SGG13].

Load balancing, as you can imagine, sometimes conflicts with processor affinity. If a process has a hard affinity for a processor, it cannot be migrated from one processor to another. If there is a soft affinity, it can be moved, but it is presumably not our first choice and we will move that process only if we have no other option. Even then, we might consider what to do: should we always move a process despite the fact that it means a whole bunch of cache misses? Should we never do so and leave processors idle? Perhaps the best thing to do is to put a certain "penalty" on moving and only move a process from one queue to another if it would be worthwhile (i.e., the imbalance is sufficiently large).

Multicore Processors

Before the early 2000s, the only way to get multiple processors in the system was to have multiple physical chips. But if you open up your laptop you are likely to find one physical chip. What gives? *Multicore processors*. As far as the operating system is concerned, a quad-core chip is made of four logical processors, but it's all in one package and this can be faster and more convenient.

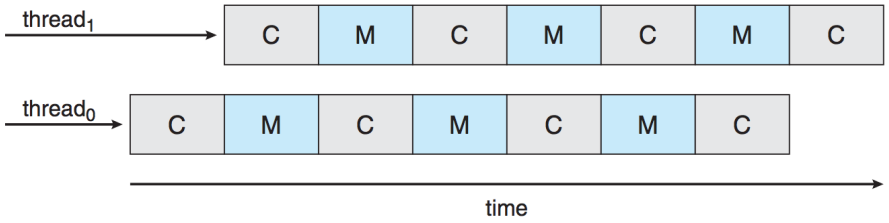
When a processor issues a memory read or write that is a cache miss (so the read has to go to memory), the CPU core can spend a lot of time (perhaps as much as 50%) of its time waiting for that read to take place. We might refer to periods of time where there is computation as a compute cycle, and time spent waiting for memory as a *memory stall*. These tend to alternate, though the length of time for each will vary significantly. It's all a question of how often memory is accessed and how many cache misses there are. See the diagram below:



Alternating compute and memory stall cycles [SGG13].

During a memory stall, the processor core may have nothing to do. As you might learn about if you take a course in processor design or programming for performance, you can sometimes move instructions around so that the memory read goes out "early" and a few other instructions can be executed in the meantime.

To offset this problem, the solution was originally called *hyperthreading*: two threads are assigned to each core; if one thread does a memory access or stalls, the code can switch to another thread with a limited penalty. The first CPU I had with hyperthreading was a Pentium IV (it was 2003), and it had one physical core but it would seemingly work on two threads at a time. See below:



Hyper threading in a single CPU core [SGG13].

If we have coarse-grained multithreading, a thread will execute for a while and when there is a memory stall (or some other reason why a process gets blocked) then the processor will swap to another thread or process, flushing the instruction pipeline. That is expensive. If we have fine-grained multithreading, it looks more like the diagram above where we have alternation between two threads that are in the pipeline at the same time. The cost of switching between these two threads is small [SGG13].

So now we have two different levels of scheduling: assigning a process or thread to a processor (the job of the operating system) and deciding when to swap between the two threads in the core (typically the job of the hardware).

29 — Real-Time Scheduling, Windows & UNIX

Real-Time Scheduling

Long ago, this course was "real time operating systems" and there was a small amount of emphasis given to the subject of real time systems. A tiny bit of this is preserved in this section about real-time scheduling, that is to say, scheduling for real-time systems. But what is a real-time system? It is one that is supposed to respond to events within a certain amount of real (wall-clock) time. There are deadlines, and there are consequences for missing deadlines. Furthermore, fast is not as important as predictable. If real time systems are of interest to you, you may wish to take some of the later embedded systems courses, or the famous CS trains course[20].

We say that a task is *hard real-time* if it has a deadline that must be met to prevent an error, prevent some damage to the system, or for the answer to make sense. If a task is attempting to calculate the position of an incoming missile, a late answer is no good. A *soft real-time* task has a deadline that is not, strictly speaking, mandatory; missing the deadline degrades the quality of the response, but it is not useless [Sta14].

If a task is hard real-time, there are two scenarios in which it might not complete before its deadline. The first is that it is scheduled too late; like an assignment that will take two hours to complete being started one hour before the deadline. If that is the case, the system will likely reject the request to start the task, or perhaps never schedule the task to run at all. Why waste computation time on a task that will not finish in time? The second scenario is that at the time of starting, completion was possible, but for whatever reason (e.g., other tasks with higher priority have occurred) it is no longer possible to meet the deadline. In that case, execution of the task may be terminated partway through so that no additional effort is wasted on a task that cannot be completed.

As a general note, most of the operating systems you are familiar with (standard Desktop/Server Linux, Mac OS, Windows) are not very suitable to real time systems. They make few guarantees, if any, about service. When there are consequences for missing deadlines, this kind of thing matters. This was, as we already discussed, a reason why Java is not a very good choice for a real-time system: the garbage collector runs whenever it pleases and can "stop the world" (halt all execution) until that is finished. And how long is that going to take? Nobody knows.

In Windows you can set a high priority (e.g. level 31) to a process and it calls the priority "Realtime", but please don't be fooled: this does not mean it guarantees it will make a particular deadline.

Timeline Scheduling

If a process or task recurs at regular intervals, it is *periodic* – repeats at a fixed period. Periodic tasks are very common: check a sensor, decode and display a frame of video to a screen, keep the wifi connection alive, etc.

Consider a periodic task to have two attributes: τ_k, the period (how often the task occurs) and c_k, the computation time (how long, in the worst case, the task might execute). In real-time systems we are usually pessimists and care almost exclusively about the worst case scenario. We can calculate the processor utilization of periodic tasks according to the following formula, to get the long-term average of processor utilization U [HZMG15]:

[20] Abandon sanity, all ye who enter here.

$$U = \sum_{k=1}^{n} \frac{c_k}{\tau_k}$$

If $U > 1$, it means the system is overloaded: there are too many periodic tasks and we cannot guarantee that the system can execute all tasks and meet the deadlines. Otherwise, we will be able to devise a schedule of some sort that makes it all work.

If the only tasks in the system are periodic ones, then we can create a fixed schedule for them. This is what the university administrators do when they create the schedule of classes for a given term. Every Monday, for example, from 13:30-14:50, course ECE 254 (a "process", if you will) takes place in classroom EIT 1015 (a resource). There is no way to have two classes in the same lecture hall at the same time, so if there are more requests for room reservations than rooms and time slots available, it means some requests cannot be accommodated.

A world in which all the tasks are periodic and behave nicely is, well, a very orderly world (and that has its appeal). Unfortunately, the real world is not so accommodating most of the time. So we will need to deal with tasks that are not periodic, which we can categorize as *aperiodic* or *sporadic*.

Aperiodic tasks are ones that respond to events that occur irregularly. There is no minimum time interval between two events; therefore it would be very difficult to make a guarantee that we will finish them before the next one occurs, so they are rarely hard real-time (hard deadlines). Tasks like this should be scheduled in such a way that that they do not prevent another task with a hard deadline from missing its deadline. As an example, if we expect an average arrival rate of 3 requests per second, there is still a 1.2% chance that eight or more requests appear within 1 second. If so, there is not much we can do to accommodate them all (most likely) [HZMG15].

Sporadic tasks are aperiodic, but they require meeting deadlines. To make such a promise we need a guarantee that there is a minimum time τ_k between occurrences of the event. Sporadic tasks can overload the system if there are too many of them, and if that is the case, we must make decisions about what tasks to schedule and which ones will miss their deadlines. If we know a task will not make its deadline, we will likely not even bother to schedule it (why waste the time on a task where the answer will be irrelevant?) [HZMG15].

So, these aperiodic and sporadic tasks really mess with timeline scheduling. This unpredictability makes it hard to create a simple timetable and follow it, just as we would expect when it comes to assigning classes and rooms.

If we have pre-emptive scheduling, then we can examine two optimal alternatives. They are called optimal because they will ensure a schedule where all tasks meet their deadlines, not because they are the ideal algorithms.

Earliest Deadline First

The earliest deadline first algorithm is, presumably, very familiar to students. If there is an assignment due today, an assignment due next Tuesday, and an exam next month, then you may choose to schedule these things by their deadlines: do the assignment due today first. After completing an assignment, decide what to do next (probably the new assignment, but perhaps a new task has arrived in the meantime?) and get on with that.

The principle is the same for the computer. Choose the task with the soonest deadline; if there is a tie, then random selection between the two will be sufficient (or other criteria may be used, if desired). If there exists some way to schedule all the tasks such that all deadlines are met, this algorithm will find it. If a task is executing because its deadline is the earliest and another task arrives with a sooner deadline, then preemption means the currently executing task should be suspended and the new task scheduled. This might mean a periodic task being preempted by an aperiodic or sporadic task [HZMG15].

Least Slack First

A similar algorithm to earliest deadline first, is least slack first. The definition of *slack* is how long a task can wait before it must be scheduled to meet its deadline. If a task will take 10 ms to execute and its deadline is 50 ms away, then there are (50 - 10) = 40 ms of slack time remaining. We have to start the task before 40 ms are expired if we want to be sure that it will finish. This does not mean, however, that we necessarily want to wait 40 ms before starting the task (even though many students tend to operate on this basis). All things being equal, we prefer tasks

to start and finish as soon as possible. It does, however, give us an indication of what tasks are in most danger of missing their deadlines and should therefore have priority.

Commercial OS Scheduling Algorithms

In this section we will examine how real commercial operating systems schedule their processes and threads. We will examine UNIX, Windows, and finally, Linux scheduling. We will see what approaches are used and what is interesting or novel about them.

Traditional UNIX

The traditional UNIX scheduling is really ancient; as in System V R3 and BSD 4.3. It was replaced in SVR4 (which had some real-time support). The information about the traditional UNIX scheduler comes primarily from [Sta14].

The routine is a multilevel feedback system using Round Robin within each of the priority queues. Time slicing is implemented and the default time slice is a (very long) 1 second. So if a process does not block or complete within 1 s, then it will be preempted. Priority is based on the process type as well as the execution history.

Processor utilization for a process j is calculated for an interval i by the following formula:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

And the priority is for process j at interval i is calculated by the formula:

$$P_j(i) = B_j + \frac{CPU_j}{2} + N_j$$

where B_j is the base priority of process j and N_j is the "nice" value of process j. The "nice" value is a UNIX way to allow a user to voluntarily reduce the priority of a process to be "nice" to other users (but honestly, who uses this?) [Tan08]. Actually, the answer to that question is: system administrators. An admin can "re-nice" a process and make it somewhat nicer than it would otherwise be.

The CPU and N components of the equation are restricted to prevent a process from migrating outside of its assigned category. A process is assigned to a given category based on what kind of process it is. To put it in simple terms, the OS puts its own needs first and tries to make the best use of resources it can. From highest to lowest priority, the categories are:

1. Swapper (move processes to and from disk)
2. Block I/O device control (e.g., disk)
3. File manipulation
4. Character I/O device control (e.g., keyboard)
5. User processes

Yes, unfortunately, user processes get piled at the bottom of the list. The use of the hierarchy should provide for efficient use of I/O devices and tends to penalize processor-bound processes at the expense of I/O bound processes. This is beneficial, because CPU-bound processes should be able to carry on executing when an I/O-bound process waits for the I/O operation to complete, but when an I/O operation is finished, we would like to start the next I/O operation so the I/O device is not waiting. This strategy is reasonably effective for a general-purpose, time-sharing operating system.

Windows

Windows schedules threads using a priority-based, preemptive scheduling algorithm, ensuring that the highest priority thread runs. The official name for the selection routine is the *dispatcher*. A thread will run until it is preempted, blocks, terminates, or until its time slice expires. If a higher priority thread is unblocked, it will preempt a lower priority thread. Windows has 32 different priority levels, the regular (priority 1 to 15) and real-time classes (16 to 31). A memory management task runs at priority 0. The dispatcher maintains a queue for each of the scheduling priorities and goes through them from highest to lowest (yes, in Windows, higher numbers are higher priorities) until it finds something to do (and if there is nothing else currently ready, the System Idle Process will "run") [SGG13].

There are six priority classes you can set a process to via Task Manager, from highest to lowest:

1. Realtime
2. High
3. Above Normal
4. Normal
5. Below Normal
6. Low

A process is usually in the Normal class. Within that class there are some more relative priorities. Unless it is in the real-time category, the relative priority can change. The details are summarized in the table below:

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Windows thread priorities [SGG13].

If a process reaches the end of a time slice, the thread is interrupted, and unless it is in the real-time category, its priority will be lowered, to a minimum of a of the base priority of each class. When a process that was blocked on something (e.g., a wait or I/O), its priority is temporarily boosted (unless it is real-time, in which case it cannot be boosted). The amount of the boost depends on what the event was: a process that was waiting for keyboard input gets a bigger boost than one that was waiting for a disk operation, for example [SGG13].

The operating system also gives a priority to whatever process is running in the selected foreground window. This is different from foreground vs. background processes as discussed earlier, because the definition of a foreground process was one that was user-interactive. Here, the distinction is which of the user-interactive processes is currently "on top" in the UI. Not only does this process get a priority boost, but it also gets longer time slices. This can be disabled, if so desired, but it really highlights the different heritages of Windows and UNIX. UNIX was originally a time-sharing system with multiple users and lots of processes; Windows was originally a single-user desktop operating system doing one or maybe a few things at a time.

30 — Scheduling in Linux

Commercial OS Scheduling, Continued

We will now continue the discussion of commercial (real-world) operating system scheduling with a much more in-depth examination of Linux.

Linux has two scheduling modes: Real-Time and Non-Real-Time (or perhaps we should call that the "normal" one). It is not necessary to use the real-time scheduler, strictly speaking, and if the real-time scheduler is used, the system can still have non-real-time threads which will be scheduled according to the normal scheduler routine.

Linux Real-Time Scheduler

The Linux scheduler operates based on *scheduling classes*, which are very much like the categories above. There are three classes into which priorities can be assigned [Sta14]:

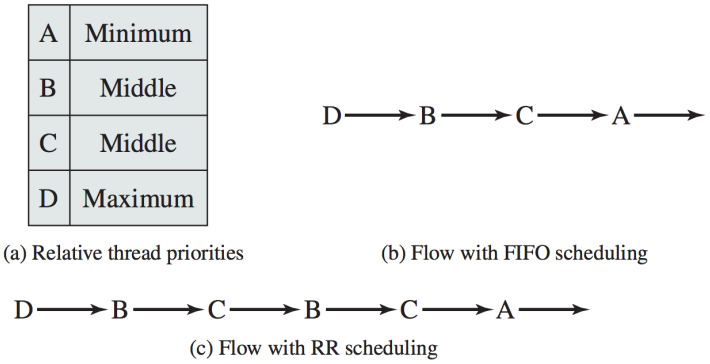
- `SCHED_FIFO`: First-In, First-Out Real-Time threads
- `SCHED_RR`: Round-Robin Real-Time threads
- `SCHED_OTHER`: Other (non-real-time) threads.

In each class, threads may have different priorities relative to one another. Lower numbers indicate higher priorities. Real-time priorities are in the range [0-99] and the other priorities are [100-139].

The rules for `SCHED_FIFO` are as follows [Sta14]:

1. The system will only interrupt a FIFO thread if one of the following is true:
   (a) Another FIFO thread of higher priority becomes ready.
   (b) The current FIFO thread gets blocked (e.g., on I/O).
   (c) The current FIFO thread yields the CPU with `sched_yield`.
2. If a FIFO thread is interrupted, it is placed in the queue associated with its priority.
3. If a FIFO thread becomes ready and that thread has higher priority than the currently-executing thread, the currently-executing thread is preempted in favour of the highest priority ready FIFO thread. If two or more threads are at the highest priority, the one that has been waiting the longest is chosen.

The policy is the same for Round-Robin real-time scheduling, except time slicing is implemented. So if a Round-Robin thread has executed for a full time slice it is suspended and the scheduler will select a real-time thread of equal or higher priority (which could certainly be the same thread, but is not necessarily). The difference is illustrated in the diagram below:



Real-Time scheduling in Linux comparing FIFO to Round-Robin (RR) [Sta14].

One of the threads in the `SCHED_OTHER` category can execute only if there are no threads in the Round-Robin or FIFO queues that are ready at the moment.

Linux Non-Real-Time Schedulers

In Linux 2.4 and earlier (shockingly late, now that I think of it), the Linux kernel used something like the traditional algorithm. Then they introduced a scheduling algorithm that was commonly called the $O(1)$ scheduler, because it executed in constant time ($O(1)$) under all circumstances. This was a big improvement over the previous scheduling algorithm which ran in $O(n)$ time. It also worked a lot better for SMP systems, because it introduced processor affinity and load balancing. Since version 2.6.23 of the kernel, however, a new scheduling algorithm has replaced the $O(1)$ scheduler; it is called the *Completely Fair Scheduler* (CFS).

Let us start by looking at the $O(1)$ scheduler. The traditional UNIX scheduler fell down on a couple of fronts: it was not very good at handling very large numbers of processes; it was an $O(n)$ algorithm, so its performance got worse as more processes appeared in the system. It also had significant difficulty with SMP systems due to its design, notably [Sta14]:

1. A single run queue;
2. A single run queue lock; and
3. An inability to pre-empt running processes.

The single run queue means a task can and will be scheduled on any processor (good for load balancing), but there is no implementation of processor affinity. Thus, a task running on CPU-0 could be easily reassigned to CPU-1 resulting in lots of cache misses.

The single run queue lock means there is one mutual exclusion construct protecting manipulation of the run queue. Thus, when one processor wants to modify it (enqueueing or dequeueing a task, for example), all other processors have to wait until it is unlocked (which can take non-trivial time as an $O(n)$ operation for sufficiently large values of n). Thus, processors may be waiting for something to do.

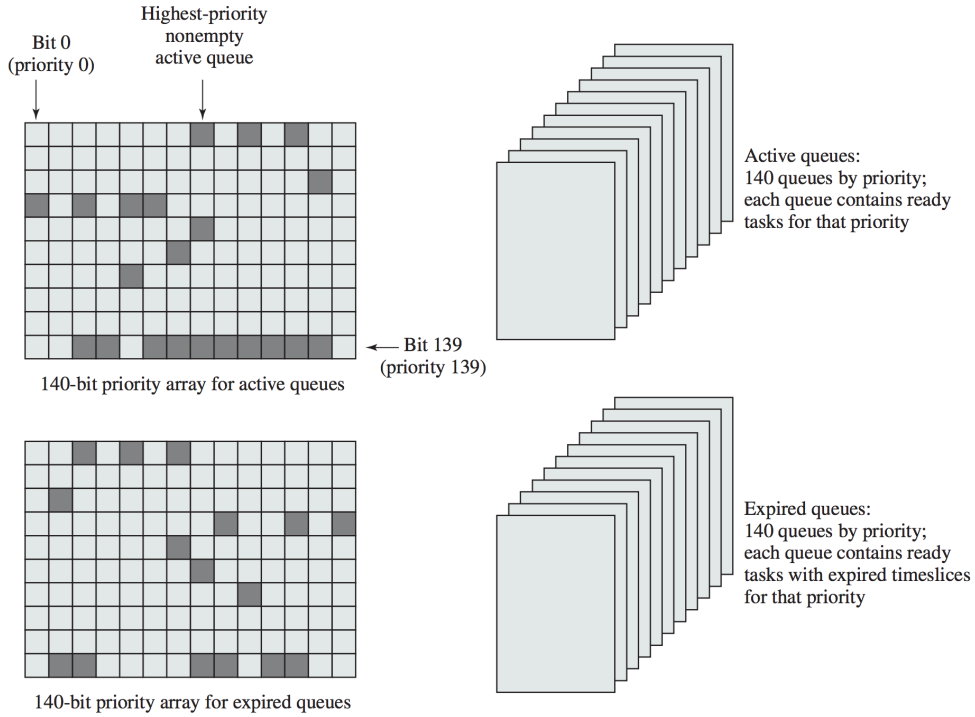
Finally, pre-emption was not possible; lower priority tasks would continue to execute while higher priority tasks were waiting. Only something getting blocked, a time slice expiration, or an interrupt might cause the scheduler to re-evaluate what process should run next.

So now that we know the problem with the traditional scheduler, we can see how the $O(1)$ scheduler is designed to address these problems. The kernel would maintain two data structures for the processor in the system [Sta14]:

```
struct prio_array {
  int nr_active; /* number of tasks in this array */
  unsigned long bitmap[BITMAP_SIZE]; /* priority bitmap */
  struct list_head queue[MAX_PRIO]; /* priority queues */
}
```

There is one queue for each priority level, thus `MAX_PRIO` (140) is both the highest priority and the number of queues. The bitmap array is of a size to provide one bit per priority level, so with 140 levels and 32 bit words, `BITMAP_SIZE` is 5. The purpose of the bitmap is to indicate which queues are empty. There is an active queue structure as well as an expired queue structure.

Initially, there are no tasks in any queues and all the bits in the bitmap are zero. If a process is created and enters the ready queue, it is put in the queue corresponding to its priority value. If that queue was previously empty, then its bit in the bitmap is set to 1 to indicate that queue is no longer empty. See the diagram below:

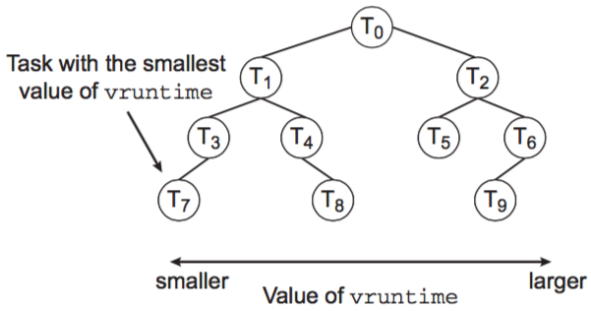


Linux $O(1)$ scheduler internal structures [Sta14].

If a process does not complete its full time slice before it is preempted, then it goes back in the ready queue. If it does run to the end of the time slice, it is placed in the expired queue instead. All scheduling takes place from the active queues. The highest priority queue is chosen; if there are multiple tasks in that queue, they are scheduled in Round-Robin fashion. This continues until the active queue structure is empty. When that happens, the active and expired queues change places, and execution (scheduling) continues [Sta14].

Part of the difficulty with the $O(1)$ scheduler is that it does not provide very good performance for interactive processes, notably the ones you work with on your desktop computer. Given that the Linux folks always claim that this year or next year is "the year of the Linux desktop" (... still waiting for that) a new scheduler was needed. Hence, the relatively rapid replacement with the Completely Fair Scheduler.

The CFS, written by Ingo Molnár, is not $O(1)$, unfortunately. It uses a red-black tree to model the ready queue, where processes are inserted based on a linear ordering of execution time. The leftmost node in this tree is therefore the task that has spent the least amount of time executing and that is what will be scheduled next. Because a red-black tree remains balanced, the time to access the leftmost node will be $O(\ln(n))$, though caching could be used to make access to the next task faster. If a task gets blocked it will not end up in the queue again, but if it reaches the end of a timeslice or gets preempted, then it will be inserted into the tree with its updated execution time, which is very likely not the same place it was taken from (which might require rebalancing the tree (a $O(\ln(n))$ operation) [HZMG15].



The Completely Fair Scheduler's red-black tree structure of ready tasks [SGG13].

Rather than using a strict rule, the CFS scheduler assigns a proportion of CPU processing time to each task based on the nice value. A nice value may be in the range -20 to +19 (lower priority is still higher priority). The CFS does not use a particular length of time slice, but instead has a *target latency* which is an interval of time in which all ready tasks should get to run at least once. The CPU time is then doled out based on the targeted latency. There are usually default and minimum values, but targeted latency can increase if there is a big increase in the number of tasks to be executed [SGG13].

The linear ordering of execution time, called `vruntime` in the earlier diagram, is also called the *virtual run time*. This is a way of keeping track of how much time a task has been executing. As with a lot of history keeping, there is a decay factor so that more recent history is more highly weighed in the calculation. Higher priority processes' history decays faster; lower priority processes' history decays more slowly. For tasks at a normal priority (nice value of zero), the virtual run time equals the physical run time. If the physical run time is, say, 50 ms, a process with a nice value of 0 will have a virtual runtime of 50. If the process has a positive nice value, the virtual runtime will be larger than 50; if a negative nice value, the virtual runtime will be less than 50 [SGG13].

Tasks that spend a lot of time using the CPU will, under this system, normally get a lower priority than a task that spends a lot of time waiting for I/O (e.g., sleeping). So a process that is user-interactive and waiting for user input will get to execute fairly quickly, making the system seem responsive to the user. Which users, of course, like.

Another thing that is noteworthy in the CFS is the addition of group scheduling: we may designate a number of processes as belonging to a group. This is useful when a process spawns lots of threads or lots of new processes. Instead of treating every thread or process totally equally, a multithreaded program's threads can all be pooled together so that the group is equal to other processes. Within the group, the scheduler will try to treat the threads or processes fairly, too.

A Decade of Wasted Cores

In 2016, researchers published a paper, exposing serious problems in the Linux scheduler, with the dramatic title: "The Linux Scheduler: a Decade of Wasted Cores" [LLF+16]. The authors found four significant bugs in Linux multicore scheduling such that there were threads waiting to run even when cores were sitting idle. Performance degradation is in the range of 13-24%, but may be as much as 138 times when looking at some corner cases.

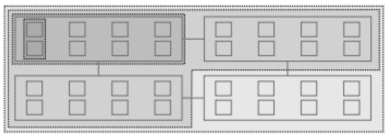
There are four different problems but they all cause the same behaviour: cores are left idle for a long time when runnable threads are waiting to execute. If it is brief, it is not a problem; but if it goes on for longer then it will be more of an issue. Suppose there are 4 CPUs, each of which is busy, and there is one thread waiting in the queue for CPU 0. If the thread in CPU 3 terminates, it may take a moment for the thread waiting on CPU 0 to move there; moving it takes some "effort" on the part of the scheduler (notice this situation, decide to do something about it, actually carry out the move) and potentially results in a few more cache misses. It may be better to leave it alone, But if that thread is waiting an unreasonably long time (in the few hundred milliseconds) then it is a problem.

Recall from earlier the completely fair scheduler we have discussed. There will be multiple run queues, one for each core. The simplest case for load balancing means two CPUs. If CPU 0 has one low priority thread and CPU 1 has three high priority threads, some sort of balancing will be needed, otherwise the high priority threads will run less than the low priority thread. Linux will periodically try to keep the queues balanced.

Unfortunately, load balancing is expensive and will run periodically but not often. But a completely idle core will result in emergency load balancing. There's a problem and we need to do something about it! And you might

imagine that load balancing is just look at how busy each of the cores is and move things from the most busy to the least busy core (... which is what most people do in the ECE 459 load balancing assignment!). That oversimplifies the solution because it does not consider cache locality or non-uniform memory access.

Thus, above the level of each core is a larger unit, a scheduling domain. Scheduling domains are configured by what hardware they have in common (e.g., level 2 cache). See the image below:



A machine with 32 cores, 4 groups, and SMT-sharing amongst pairs of cores [LLF+16].

In this image we have multiple levels: three groups are reachable from the first core (CPU 0) in one hop and the rest reachable in two hops. The scheduler will avoid duplicating work by making sure that one core is responsible for load balancing within that schedule domain. This is the lowest numbered idle core, or the lowest number overall if all are busy. The only way a core can get woken up is for another one to wake it up, and so a core that is busy and notices a lazy one sleeping nearby will wake it up and tell it to do load balancing [Coy16].

So what are the four bugs that caused this problem? The summary of these bugs from [Coy16]: (1) the group imbalance bug, (2) the scheduling group construction bug, (3) the overload on wakeup bug, and (4) the missing scheduling domains bug.

The Group Imbalance Bug Cores would attempt to steal work from other cores if the average load of the victim scheduling group is higher than the average load of the one doing the stealing. But averages can be misleading! The fix is to use the minimum load of the group, meaning the load of the least loaded core of the group. This means cores will steal more often, but this is better than leaving them idle. This can result in a 13% decrease in the runtime of `make`.

Scheduling Group Construction The Linux `taskset` command allows applications to be pinned to specific cores. If the groups are two hops apart, the load balancing thread might not steal them... This problem arises because all groups are constructed from the perspective of core 0. If, therefore, the load balancing is running on core 31 it might not steal from a neighbouring core because it thinks it is too far away because it is two hops from core 0.

Overload on Wakeup We have already discussed the idea of processor affinity, but sometimes, too much of a good thing is a problem. If a thread goes to sleep on group 1, when it and it gets unblocked later by some other thread, the scheduler will try to put it on one of the cores in group 1... even if other groups are not busy. This will reduce the number of cache misses, sure, but it means sometimes a thread gets in a queue that's busy rather than one that's free.

Missing Scheduling Domains The last bug appears to have been caused by an error during refactoring. When a core was removed and re-added a step was skipped after the refactoring changes which could cause all threads of an application to run on a single core instead of all of them.

In conclusion: scheduling is by no means a solved problem. A simple scheduling algorithm that worked reasonably well in a single core environment was not adequate to the multiple core world. Averages can be misleading and optimizations sometimes do more harm than good.

Bibliography

[Bar14] Blaise Barney. POSIX Threads Programming, 2014. Online; accessed 1-March-2015. URL: https://computing.llnl.gov/tutorials/pthreads/.

[Coy16] Adrian Coyler. The linux scheduler: a decade of wasted cores, 2016. Online; accessed 24-April-2017. URL: https://blog.acolyer.org/2016/04/26/the-linux-scheduler-a-decade-of-wasted-cores/.

[Dij68] Edsger Dijkstra. The Structure of the 'THE' Multiprogramming System. In *Communications of the ACM*, 1968.

[Dow08] Allen B. Downey. *The Little Book of Semaphores (2nd Edition)*. Green Tea Press, 2008.

[Hal15a] Brian "Beej Jorgensen" Hall. Beej's guide to network programming: Using internet sockets, version 3.0.21, 2015. Online: accessed 5-July-2018. URL: https://beej.us/guide/bgnet/html/single/bgnet.html.

[Hal15b] Brian "Beej Jorgensen" Hall. Beej's guide to unix ipc, 2015. Online; accessed 21-August-2019. URL: https://beej.us/guide/bgipc/html/multi/index.html.

[HS91] S. Haldar and D. Subramanian. Fairness in Processor Scheduling in Time Sharing Systems. Operating Systems Review, 1991.

[HZMG15] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2015. Online; version 0.15.08.17.

[Let88] Gordon Letwin. *Inside OS/2*. Microsoft Press, 1988.

[LLF+16] Jean-Pierre Lozi, Baptiste Lepers, Justin R. Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 1:1–1:16, 2016. URL: http://doi.acm.org/10.1145/2901318.2901326.

[Sal] Peter Jay Salzman. Reading And Writing Files in C. Online; accessed 4-June-2018. URL: http://www.dirac.org/linux/programming/tutorials/files_in_C/.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[SR13] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment, Third Edition*. Addison-Wesley, 2013.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.

[TRG+87] Avadis Tevanian, Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young. Mach threads and the UNIX kernel: The battle for control. In *Proceedings, Summer 1987 USENIX Conference*, 1987.