Contents

1 — Introduction, Operating Systems, Security 2

2 — Review of Processes and Threads 8

3 — Sunrise to Sunset 14

4 — Review of Concurrency Control 19

5 — Concurrency Control Implementation 24

6 — Memory 29

7 — Dynamic Memory Allocation 35

8 — Memory: Segmentation and Paging 40

9 — Caching 47

10 — Virtual Memory 52

11 — Virtual Memory II 57

12 — Uniprocessor Scheduling 63

13 — Scheduling Algorithms 69

14 — Scheduling: Idling, Priorities, Multiprocessor 76

15 — Real Time Scheduling 82

16 — Real Time Scheduling Algorithms 87

17 — Scheduling Algorithm Evaluation; UNIX & Windows 93

18 — Scheduling in Linux 98

19 — Input/Output Devices & Drivers 103

20 — More About Input/Output Devices 109

21 — Disk Scheduling 114

22 — File System Implementation 120

23 — File Allocation Methods 127

24 — Reliability: RAID 134

25 — Reliability: Fail-Soft Operation 139

26 — Virtualization and Containers 144

1 — Introduction, Operating Systems, Security

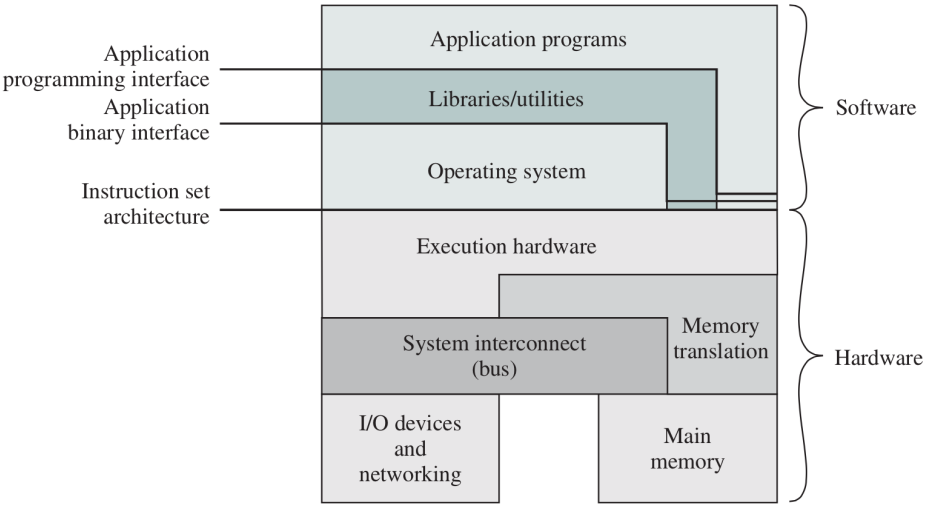
About the Course

We'll start by reviewing the highlights of the class syllabus. Please read it carefully (it is available in Learn under Content → Overview). It contains a lot of important information about the class including: the lecture topics, the grading scheme, contact information for the course staff, and university policies.

I C what you did there. This course assumes that you have enough knowledge of C that we don't have to spend some time going over it. In fact, we went over it in ECE 252. If you're still not entirely comfortable with C as a language, I would recommend going back to revisit the content from the first lecture – and maybe some assignments or similar!

Introduction to Operating Systems

> *Operating systems are those programs that interface the machine with the applications programs. The main function of these systems is to dynamically allocate the shared system resources to the executing programs.*
>
> *But the interface with adjacent levels continues to shift with time. Functions that were originally part of the operating system have migrated to the hardware. On the other side, programmed functions extraneous to the problems being solved by the application programs are included in the operating system.*
>
> - What Can Be Automated?: The Computer Science and Engineering Research Study, MIT Press, 1980



Structural diagram of a modern computer [Sta18].

An operating system (often abbreviated OS) is a piece of software that sits between the hardware of a computer and the applications (programs) that run on that computer. The OS does many different things and often has many (occasionally-conflicting) goals. It is responsible for seeing to it that other programs can operate efficiently, providing an environment for other programs, and collecting and reporting data about what is and has been happening. It also needs to enforce policies that are defined by system administrators.

An operating system is also responsible for resource allocation. In the real world, the resources we have to work with, such as CPU time or memory space, are limited. The OS decides how to allocate these resources, keeps track of who currently owns what, and, in the event of conflicting requests, determines who gets the resource.

The OS usually enables useful programs like Photoshop or Microsoft Word to run. Any computer has various pieces of hardware, such as the CPU, memory, input/output devices (such as monitors, keyboards, modems). The OS is responsible for abstracting away the details of this, so that the authors of programs do not have to worry about the specifics of the hardware. Imagine how painful it would be to write even a simple program, like the Hello World example, if we had to write our program differently for every combination of hardware.

In most cases there will be multiple programs running on the computer. This implies the sharing of various resources. When this is the case, there is the potential for conflicts to arise. An operating system creates and enforces rules to make sure all the programs get along and play fairly. Of course, not all interaction between programs is competitive; sometimes they want to co-operate, and the OS helps them do that, too.

Another goal may be to use the computer hardware efficiently. This is not usually an issue with personal laptops, but imagine a supercomputer. A supercomputer used to do extremely complex computations is expensive to build and maintain. Any moment when the supercomputer is not doing useful work is a costly waste, so an operating system for such a computer would try to maximize CPU usage. There is, after all, only so much CPU time in supercomputers and there are many programs eager to run (weather simulations, particle physics simulations...).

Operating systems tend to be large and do a lot of things. We expect now that an OS comes with a web browser, an e-mail client, some method for editing text, et cetera. These things, while important and useful, are not what we are going to focus on. The part of the operating system that we will study is what we call the *Kernel* - it is the "core"; the portion of the OS that is always present in main memory and the central part that makes it all work.

Operating systems will evolve over time. There will be new hardware released, new types of hardware, new services added, and bug fixes. Evolution is constrained by a need to maintain compatibility for programs. If the user upgrades his or her desktop OS and a program breaks, even if it's the program author's fault, the user blames the OS vendor. If you look at Microsoft Windows, you can see their strict devotion to not breaking binary compatibility (at least, as much as they reasonably can). Linus Torvalds, yes, the person Linux is named after, gets unreasonably angry with people if they submit a kernel change that might break user-space programs.

How obsessive are OS designers? Consider this example, admittedly from more than 25 years ago, about Windows 95 (yes, as in 1995)... from [Spo00]:

> Windows 95? No problem. Nice new 32 bit API, but it still ran old 16 bit software perfectly. Microsoft obsessed about this, spending a big chunk of change testing every old program they could find with Windows 95. Jon Ross, who wrote the original version of SimCity for Windows 3.x, told me that he accidentally left a bug in SimCity where he read memory that he had just freed. Yep. It worked fine on Windows 3.x, because the memory never went anywhere. Here's the amazing part: On beta versions of Windows 95, SimCity wasn't working in testing. Microsoft tracked down the bug and added specific code to Windows 95 that looks for SimCity. If it finds SimCity running, it runs the memory allocator in a special mode that doesn't free memory right away. That's the kind of obsession with backward compatibility that made people willing to upgrade to Windows 95.

As we proceed we will focus on UNIX-like systems with some mention of Microsoft Windows where appropriate. This is for practical reasons: there is only so much time and UNIX-like systems represent a very large percentage of the operating systems out there including Linux, macOS/Mac OS X, Android, and more.

Time, What is Time?

And then there's the real-time part of the puzzle. What makes a system real-time versus non-real-time? What it comes down to is whether or not wall-clock deadlines actually matter. Real life is full of systems, many of them embedded systems, that operate on the basis of meaningful deadlines. Some common examples of real-time systems include aviation, anti-lock braking systems, industrial machinery, video conferencing, and satellite launches.

When we talk about real-time systems, they deal with "tasks", which really are just things that need doing. Tasks are subdivided into the *hard real-time* and *soft real-time* categories. If a task is hard real-time, then a late answer is useless. If we're trying to launch a satellite into orbit, if the we don't have the calculation of when to fire the thrusters aren't ready in time, the calculation is not useful and would need to be repeated. If a task is soft real-time, then a late answer is of diminished quality but could still potentially be used: if decoding a frame of video takes longer than expected, the video quality (and viewing experience) is worse, but it's not totally ruined.

There will be much more about real-time systems later on. In most topics, we'll first talk about operating systems in general, and then we'll build on that to see what new complexity or differences the new constraint of real-time adds.

Security Now

When I taught a previous operating systems course, ECE 254, after the introduction, we spent some time talking about the history of operating systems and how we got to the modern generations of them. While some understanding of the history of a thing might be helpful in explaining some design decisions, the discussion was too vague and high-level to be useful. Nice trivia, maybe. So instead we are going to talk about security (but mostly, actually, protection).

In a lot of textbooks, security and protection are left to the end. This strikes me as being a problem: security is something you want to bake into your design and something you want to have in your mind constantly. It doesn't work to try to bolt it on afterwards. As we go through the topics of the course, we aren't going to stop at every opportunity to consider the security implications or risks of each implementation or decision... but it would still be a good idea to think about security in each situation.

An operating system is designed to support multiple users concurrently, each of whom are likely running multiple programs concurrently too. There are also the operating system's processes itself, which are not really under the user's control either. Even if you are the only user on your system, the operating system will enforce certain rules so that malicious programs (or malicious websites) can't steal your personal data or sabotage the system.

Real-time operating systems are less likely than others to support the idea of multiple users concurrently; industrial machinery to assemble a tank doesn't really have user accounts and configuration the way that a server running Ubuntu might. Still, even that kind of system can have multiple processes concurrently – user processes or OS processes – so what we are discussing here is still relevant.

Typically, OS designers create some policies and also policy tools. Some policies are just a part of the operating system and cannot be changed – e.g., a file must have the "execute" permission to run. Others are configurable by system administrators – e.g., may non-administrator users install new programs on the system? Security policies do have some tradeoff with usability, in that it can be frustrating for users who are denied some operation and must instead ask administrators to do it for them. But you also can't be too lax about this, because you most certainly do NOT want to find your company's name on TV having to report a data breach in which user personal data was stolen.

OS designers must see to it that the security policies (whether configured or not) are enforced consistently. This helps to ensure that sensitive data is protected, not corrupted, and only accessed by those who should have access. Operating systems that do not provide proper protection and security will inevitably be exploited by malicious users and it will cease to be used.

Whatever the specifics about policies, there are three desirable properties that go by the acronym "CIA" – Confidentiality, Integrity, Availability. Confidentiality is that information should only be access by those who are authorized

to see it. Integrity is that information should be consistent and correct. Availability is that information or services should be available when they are needed.

Protection vs. Security It's important to draw a quick distinction between *protection* and *security* as in [SGG13]. Protection is about "internal threats" – things like making sure that user `morgan` cannot access the private files of user `taylor`. Security is more about "external threats" – things like making sure that evil hackers don't gain access to the system. Maybe we take those in order.

Protection

Truthfully, most of our discussion in this course will be about protection – how does the design of the operating system ensure that the rules are followed. Following the rules is important to actually have a functioning system. Without it, anarchy results. As much as we'd all like to live in a world where everyone is nice and nobody does anything wrong, but, uh, have you seen the news? But even if you did have a system where all the legitimate users lacked malicious intent, it would be possible for someone to disrupt the experience of others by accident. That sometimes happens on the ECE-operated servers, where someone unintentionally writes a program that exhausts shared memory or gets in an infinite loop and uses excessive CPU time.

What are the goals, then, of protection? Ultimately, it's about enforcing the policies about responsible resource usage [SGG13]. What is responsible and reasonable for a given system may be different when compared to a different system. Some servers are dedicated to running exactly one service and there's nothing wrong with letting that service take all available CPU time, memory, disk space... Other servers, like the ECE Ubuntu systems, are meant to be shared amongst hundreds and hundreds of students, any of whom could be working on different courses at any time... so resources there need to be shared.

An obvious case of access control that's important to consider is permissions on files. There are many files in a shared system, but not all of them are yours. Some of them belong to you, yes, and some belong to other users, and yet others belong to the operating system itself. You wouldn't want to do your assignment on a shared server that had no enforcement of ownership: it would be all too easy for someone to copy your code... or delete it to sabotage you.

Another example: the operating system also enforces logical walls between different processes. As we go through the course we'll see a few different cases. For one, the memory of a given program is not accessible with other programs.

These rules take some effort to enforce: requests and actions have to be checked, by the operating system (of course), to make sure that they are valid, where valid means in compliance with the rules.

With that said, there are exceptions. You can make a file "public" so that you would intentionally allow others to access it. You can also ask your program to used shared memory such that another program can share the same section (and we'll talk about that). Also, system administrators can generally override policies and do things that normal users can't... including reading other people's files. You can see why we need a lot of trust in administrators.

The usual protection rules that we see above – limiting access to data based on rules – are the most common ones, but they aren't the only ones. We could have rules that terminate processes if they use too much CPU time or memory, but those are fairly rare in the real world. While there are probably no good reasons for users to read one another's data, there are frequently good reasons for using a lot of resources.

Let's imagine for a minute that I want to edit a lecture video. It makes sense, then, that editing the video uses a very large amount of memory as I'm putting together all the different pieces of content. When I'm done editing and I want to render the video (create the final video output), that's a CPU-intensive task and it may max out all the CPUs for quite a while. That's a perfectly legitimate use of the system – I'm not doing anything malicious and it's for a valid work-related purpose. But it would maybe set off alarms if we just considered the memory and/or CPU usage in isolation.

That does mean that people can exploit the lack of strictness and make excessive use of system resources. When we talk about scheduling algorithms, we'll see how a real system will do its best to make sure that even if a user is requesting excessive CPU time, that it won't impact others excessively. If that's not enough, there's always the option to escalate to system administrators who can do something about it.

So this brief introduction should hopefully make clear the importance of protection when we consider how file systems, shared memory, and others work to prevent internal problems... so let's also consider external factors.

Security

Although we won't talk about security in as much detail as protection, I want to at least go over a few different ways in which attackers could try to exploit the system. We need this in mind when understanding the design of an operating system because the presence of these threats will cause us to change how we design systems. Just think about how different air travel would be if there were actually no need for security.

Some attacks worth considering below [SGG13]:

- **Breach of Confidentiality.** This is when some external actor gains access to information they should not be able to have, such as users' personal details (name, address, etc) or private information (health records, financial records). Getting this data is pretty lucrative for attackers, because it can be used for the attacker's financial gain. It's also terrible for the company, because a leak of private information makes regulatory authorities very mad.
- **Breach of Integrity.** This is when an attacker is able to corrupt or otherwise alter data that they should not have access to. This can be totally destructive – erase all users from the database – or more subtle, where the attacker increases the payroll pay going to a certain account.
- **Theft of Service.** This is what happens when an attacker is able to make use of some resources that they should not be able to. This might be getting some paid software-as-a-service without paying, or using the company servers to mine cryptocurrency, for example.
- **Breach of Availability and Denial of Service.** This is what happens when an attacker is able to prevent a service from working as intended. This might be by trying to overwhelm a service with too many requests, or it could be deliberately crashing a server that is vulnerable in some way.

These categories are not exhaustive, but give an idea of the kind of bad behaviour that we should be worried about. If we want to have a secure system, good design of the operating system is necessary but not sufficient. If a house has locks but the owners don't use them, the locks are not very effective. Configuration and policies are important too. But there's also the human factor.

Most likely the weakest link in any system is the human! A perfectly secure system technically can easily be defeated by *phishing*, an attacker tricking a legitimate user into handing over their credentials or doing something they should otherwise not be doing ("yes, I'll tell you the payroll information..."). That sort of thing is harder to design to solve, but should not be overlooked either.

The categories discussed above are just types of problem that attackers can cause. Some of them used examples where I gave a specific type of attack, but here's a few ideas, not exhaustive (obviously) that come to mind:

- **Excessive Requests** – overload the system with demands, whether for CPU, memory, responses, etc.
- **Malformed Requests** – send intentionally malformed requests that can cause the system to behave in an unexpected way, crash, or return information it should not be sending.
- **Back Door** – sneaking some code into a program that allows (normally unauthorized) access to a system
- **Intercepting Messages** – observing the communication between systems and intercepting messages to change them or gain information. This is also called a Man-in-the-Middle Attack.
- **Trojan Horse** – tricking someone into installing something that contains a hidden payload that can do one of the things above...

Security is, and always has been, a sort of arms race. When an exploit or other vulnerability is discovered, operating system developers need to find a way to guard against it and implement a patch for it (and roll it out). Sometimes the security problem is just the result of a bug, in which case patching it just ensures the system has its intended

behaviour. If it's a design problem, then changing the design might actually be painful because it will cause existing software to break. It gets worse, I suppose, because if a bug has been around for long enough, programs may actually rely on the behaviour of that bug. So patching it breaks them too!

Now we have a dilemma! What do we do? Break existing software in the name of security or preserve the user-land programs' behaviour at the cost of security? There are no one-size-fits all answers to this question. That would be too easy.

This is not a course in security, so we can't spend too much time talking about it and will cut it off here, even if it can be fascinating. The goal is that we should just be thinking about security (and protection) when considering a design decision. And that consideration should take place early on in the process and not be simply an afterthought!

2 — Review of Processes and Threads

Past is Prologue

In prerequisite courses (should you have taken them as per the standard program!), we covered three important things that are relevant to the operating system: processes, threads, and hardware. Here, we're going to take some time to review these things and make sure we're on solid ground before moving on. Does this look and sound exactly like what was covered in ECE 252? Yes. But review is necessary.

The Process and the Thread

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.

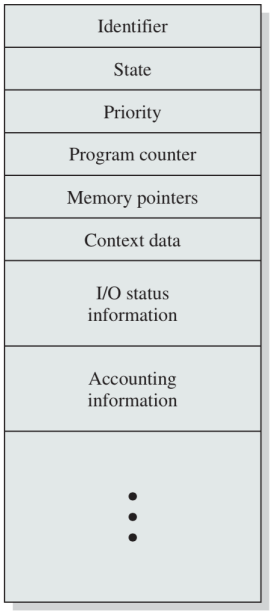
The Process Control Block. The operating system's data structure for managing processes is the *Process Control Block* (PCB). This is a data structure containing what the OS needs to know about the program. It is created and updated by the OS for each running process and can be thrown away when the program 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 [Sta18]:

- **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.

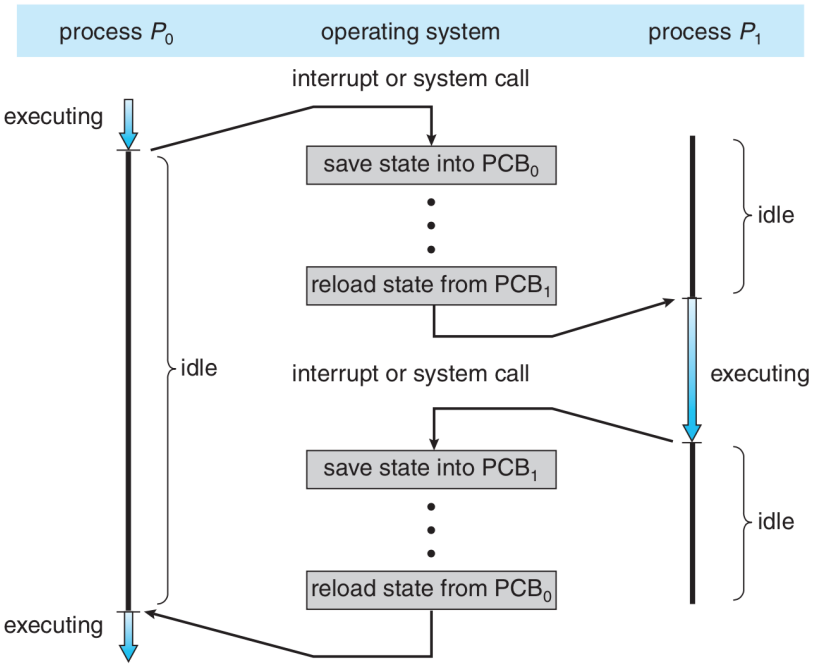
- **Accounting Information.** Some data about this process's use of resources. This is optional (but common).

To represent this visually:



A simplified Process Control Block [Sta18].

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. 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.

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*.

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.

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

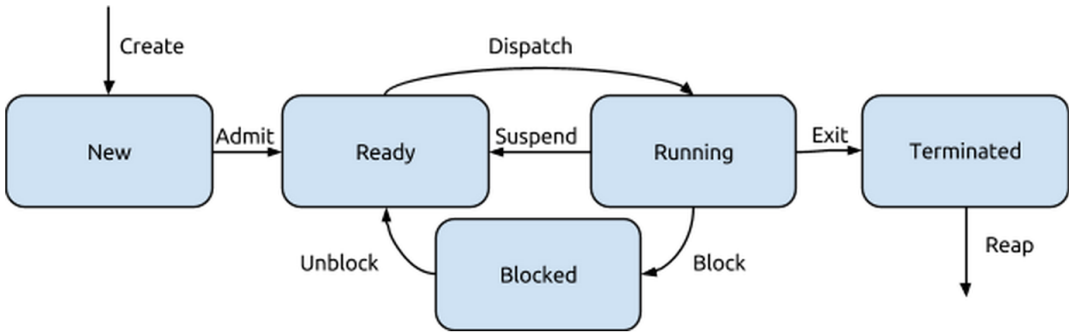
[1] Security advice: don't click on links you receive by e-mail.

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.

Process States. The diagram below shows the five-state model:



State diagram for the five-state model.

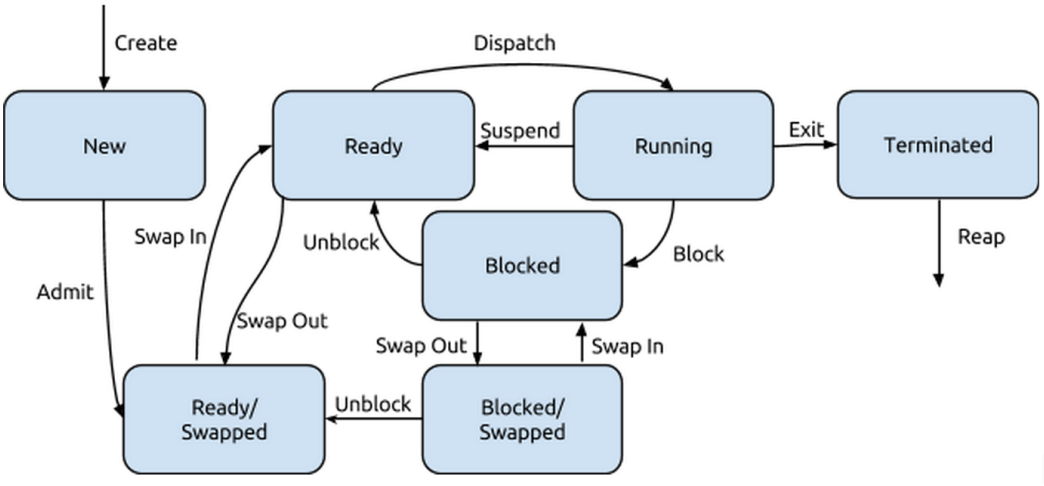
There are now eight transitions, most of which are similar to what we have seen before:

- **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.

Remember that this model works for a thread, but the process has two additional ones:

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.

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.

At this point I assume you remember how to use `fork()` and related functions like `wait()` and there's no need to recap it here. If you are uncertain about it, please check the ECE 252 notes!

And the Thread

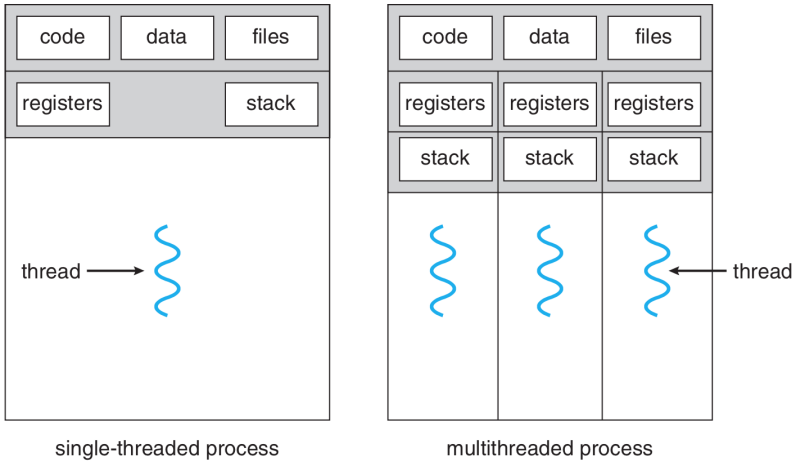
The term "thread" is a short form of *Thread of Execution*. A thread of execution is a sequence of executable commands that can be scheduled to run on the CPU. Threads also have some state (where in the sequence of executable commands the program is) and some local variables. Most programs you have written 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 [Sta18]:

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].

As you know, the primary motivation for threads over processes is performance. They are much faster to create and clean up than processes, and there's no overhead of establishing shared-memory communication. But of course, there are risks, like any one thread crashing the whole program.

Like with processes, I'll assume you remember how the various pthread functions work from ECE 252 – if not, please go back and look at that – it will save you a lot of headache...

3 — Sunrise to Sunset

Here When You Need Me

We already learned about the lifecycle of a process, but the operating system itself has a lifecycle as well. It needs to be started, it has a period where it executes, and will eventually shut down (for some reason or another). While it's running, it offers a number of services and a suite of functionality that is intended to make it easy and safe for user programs to execute.

Starting Up

We will imagine the operating system is already installed on disk and not worry too much about systems without permanent storage (even though they did exist in the past). When you power on the computer, the CPU execution at a predetermined location. This is in the BIOS (Basic Input-Output System), which is firmware, when it's an older computer. Newer computers use UEFI (Unified Extensible Firmware Interface) that stores the boot information on disk rather than in the firmware itself. I don't want to get too bogged down in the details about how this works, but however it's implemented, it goes to the configured boot device (usually a hard drive) and starts execution of the boot loader.

What's a boot loader? It's a small piece of code whose purpose is to start up the operating system. This may be a multi-stage process involving loading a progressively-larger boot program from disk, but in the end, the goal is to load the kernel and begin its execution. The boot loader program may do other things, like verify the state of the machine and if the diagnostics are okay, proceed; it can also initialize things like the CPU registers, device controllers, wipe memory... [SGG13].

The boot loader itself has at least some initialization code located in the first block of the hard drive that is the boot device. If the system has more than one drive, not all of them are necessarily configured for booting the system. Only if the first block of the drive contains this information can it be used for booting. The actual boot loader might be bigger. No matter how many steps it takes, eventually control is transferred to the kernel itself and then it's running.

Once the kernel is running, it can start the relevant operating system services, including the things that users will eventually interact with, like the login screen or the remote login server. Some services and utilities are always started when the OS begins running; others are configurable so that an administrator can choose to have them start on system boot up. Although they are started by the OS without any direct user interaction, most processes are still user-level programs that operate like normal and must interact with the OS through the system call interface. Speaking of...

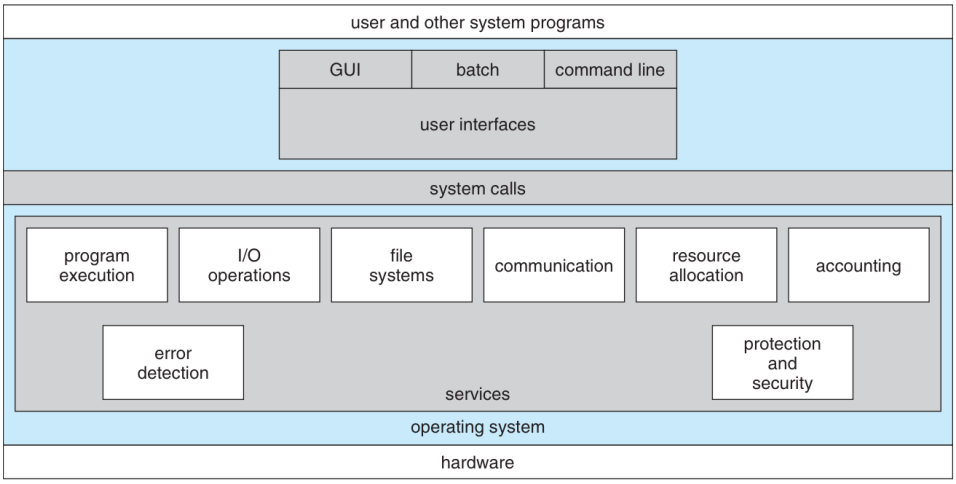
What Can You Do For Me?

The main purpose of the operating system is not to run itself, but is there to make things work for the user-level programs that are supposed to run. Our previous experience with systems programming has taught us a lot about the services that the operating system has to offer. Some examples of things we already asked the OS to do for us:

- Process and thread creation, termination
- Inter-process communication
- Concurrency control
- Memory allocation

The operating system does a lot more than this, and the visibility of that functionality varies. The OS must check permissions whenever you want to open a file, but you probably do not notice or think about it unless somehow permission is denied. There is scheduling, which we had to think about when it comes to concurrency and synchronization, but it was treated as external forces and nothing we could control. And then there are yet other things that the operating system does independently and we may or may not have any way of interacting with them, like the accounting and record keeping the system does.

A visual representation of that from [SGG13]:



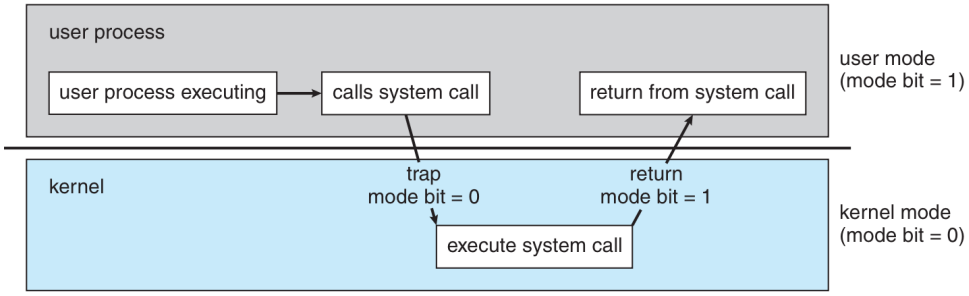
At steady-state, the operating system runs whatever background tasks it needs to do, but the interesting things happen as the result of a user-level program asking for the OS to do something. And you know how that works...

Remember, it's a Trap. Previously, we covered this from the point of view of a user program that wants to activate the kernel. We said that it operates on interrupts and the interesting thing is the intentional use of the trap: this is how a user program gets the operating system's attention. When a user program is running, the operating system is not; we might even say it is "sleeping". If the program running needs the operating system to do something, it needs to wake up the OS: interrupt its sleep. When the trap occurs, the interrupt handler (part of the OS) is going to run to deal with the request.

We saw the concept of user mode vs. supervisor mode instructions: some instructions are not available in user mode. Supervisor mode, also called kernel mode, allows all instructions and operations. Even something seemingly simple like reading from disk or writing to console output requires privileged instructions. These are common operations, but they involve the operating system every time.

Modern processors keep track of what mode they are in with the mode bit. This was not the case for some older processors and some current processors have more than two modes, but we will restrict ourselves to dual-mode operation with a mode bit. Thus we can see at a glance which mode the system is in. At boot up, the computer starts up in kernel mode as the operating system is started and loaded. User programs are always started in user mode. When a trap or interrupt occurs, and the operating system takes over, the mode bit is set to kernel mode; when it is finished the system goes back to user mode before the user program resumes [SGG13].

Suppose a text editor wants to output data to a printer. Management of I/O devices like printers is the job of the OS, so to send the data, the text editor must ask the OS to step in, as in the diagram below:



Transition from user to supervisor (kernel) mode [SGG13].

So to print out the data, the program will prepare the data for printing. Then it calls the system call. You may think of this as being just like a normal function call, except it involves the operating system. This triggers the operating system (with a trap). The operating system responds and executes the system call and dispatches that data to the printer. When this job is done, operation goes back to user mode and the program returns from the system call.

Motivation for Dual Mode Operation. Why do we have user and supervisor modes, anyway? As Uncle Ben told Spiderman, "with great power comes great responsibility". Many of the reasons are the same as why we have user accounts and administrator accounts: we want to protect the system and its integrity against errant and malicious users.

An example: multiple programs might be trying to use the same I/O device at once. If Program 1 tries to read from disk, it will take time for that request to be serviced. During that time, if Program 2 wants to read from the same disk, the operating system will force Program 2 to wait its turn. Without the OS to enforce this, it would be up to the author(s) of Program 2 to check if the disk is currently in use and to wait patiently for it to become available. That may work if everybody plays nicely, but without someone to enforce the rules, sooner or later there will be a program that does something nasty, like cancel another program's read request and perform its read first.

This doesn't come for free, of course: there is a definite performance trade-off. Switching from user mode to kernel mode requires some instructions and some time. It would be faster if everything ran in kernel mode because we would spend no time switching. Despite this, the performance hit for the mode switch is judged worthwhile for the security and integrity benefits it provides.

Policy and Mechanism. On this subject of the motivation for dual mode operation, let's talk a little bit about policy and mechanism. In short, *policy* tells us what will be done and *mechanism* is how the policy is carried out; ideally these are separated to some degree to provide flexibility [SGG13].

Some policies are configurable, such as how much do priorities of processes matter; others are not, such as whether processes can read from files for which they do not have the read-permission. What is configurable policy is a question of operating system design. Actually applying the policy is involves both design and implementation. For the most part, though, operating systems tend to err on the side of having fewer configuration options. Is that because the authors know best what the user (system administrator) wants and needs, because they merely think they do? That's a lengthy debate, to be sure, but outside the scope of this course.

From the point of view of the user program, policy is something that we have to contend with but may not have any say in it, though you could hypothetically require that the application be granted certain permissions before running. If the policy is that it's not possible to access the memory of other processes, well, that's policy and we have to write a program with that in mind. Having to follow the rules may be less convenient, but it's not optional.

The operating system code itself has no such restrictions: it is free to ignore policy if it wants and freely read and write in memory. This level of access requires that the operating system be, to some great extent, trusted by the application authors. If you didn't trust the operating system, would you feel comfortable logging in to online banking on that system? Are you sure?

The operating system also has to be responsible for the mechanism: enforcement of the policy. We will see in later topics how to enforce certain policies, such as fairness in terms of CPU time, or terminating a thread if it tries to read memory that belongs to some other process.

Switching Processes or Threads

We discussed some details about how a process switch occurs and reviewed it recently, but there wasn't much detail on when does a switch of processes happen? From the point of view of the application program we say that the process switch could happen at any time and we tried to imagine that it would happen at the most inconvenient time to cause an issue. It could happen at any time from the point of view of the user program, but the operating system gets to choose (to some extent) when it happens. The operating system, fortunately, does get to choose which process will run next, but that's not a trivial decision, to the point where scheduling is one of the major topics of this course.

Realistically, processes always get paused as a result of an interrupt, and that may or may not result in a thread switch. Some of them are intentional, voluntary invocations of system calls which result in the trap interrupt. Other interrupts, regardless of source, lead to the same thing. The interrupt is handled and while that's the case, the operating system is running and the thread that was executing before is suspended.

The first situation where a process (thread) switch must occur is when the currently-executing process (thread) gets blocked. It's the user program that chose to wait for network data or reading from a file. Whatever the reason that the current process becomes blocked, that means it cannot continue executing.

Similarly, another reason that process switch must occur is the termination (voluntary or otherwise) of the currently-executing process or thread. If it was a call to voluntarily exit, that was a system call. If it occurred as a result of something going wrong (e.g., division by zero), that's an exception and the exception is an interrupt.

That covers the mandatory switches but there are those where the operating system chooses to switch. Whatever interrupt has been handled, whether a system call trap, the currently-executing thread is suspended and the operating system is running. After dealing with the request (whether or not a final answer is ready), it's equally valid to restore a different process to run next rather than the one that was just suspended. This is, again, a scheduling decision. But this category covers a lot of situations, including creating a new process.

When we get to scheduling, we'll also consider things like wanting to prevent monopolization of the CPU through various mechanisms that force threads to take turns. One possible way to do so is a timer interrupt, where an interrupt is generated after a period of time, and that's a prompt to consider if a switch should occur.

Keeping Track

At steady state, the operating system will have to keep track of the various resources in some sort of tables or other data structures such as memory tables, I/O tables, file tables, and process tables [Sta18]. We will get into the details of how memory, I/O, and files are managed in later topics. But they are pretty much exactly what they sound like. But let's cover these at a high level now.

Memory tables track the state of memory, both the memory that's free and what is in use. The operating system needs memory for itself to do its job, such as containing the process control blocks, buffers, etc. And unlike a user process that can ask the OS to allocate and deallocate memory for it, the OS has to do this the "hard" way. But in addition to that, the OS needs to keep track of some attributes of memory, including the protection rules and whether any sections of memory are shared [Sta18].

Memory tables may need to be updated every time memory is allocated or deallocated and whenever changes take place on shared memory. Process creation and destruction also result in significant changes to the memory being managed as well as the process. Yes, a new process control block needs to be created, but also the memory space for the process itself is required – the program has some global variables, stack, and heap.

I/O tables are used for keeping track of the status of the various I/O devices attached to the system. I/O devices may be shared or assigned to a specific process. But more importantly, when there are I/O operations in progress, it's necessary to keep track of what the operation is and where the data is coming from and going to.

And then there are file tables: the operating system keeps track of which files are open overall, and which ones a given process is using. Remember that in UNIX, the abstraction is that everything is a file, so files refer to not only actual files on disk but also other things like sockets or pipes.

Shutting Down

Shutting down the operating system is relatively straightforward procedure. To do so, it will need to notify all running processes that they should exit. Sending them a signal that asks them to exit should be sufficient, and ideally the program authors have implemented something that makes it a graceful shutdown rather than just dying unceremoniously.

As you may know, asking the programs to shut down politely may not actually get them to terminate. This may be because they intentionally or unintentionally ignore the polite request to terminate. If you're editing a document, the editor might have a little pop up window asking if you'd like to save changes, and will wait an indefinite period for the user to decide. If the OS shutdown has been called, at some point you may decide to forcibly terminate the processes even if it means the user loses some work. It's an interesting design decision to think about how long you are willing to wait.

The other consideration is: can every user, even those who are not administrators, request a shutdown of the system? For a desktop system then it is probably fine – if I'm done with the computer and want to turn it off to save power that's sensible. But if it's a multi-user server, other people would be very annoyed if I turned it off while they are using it!

Once the user programs are all terminated, the operating system can terminate its own internal services and then signal to the hardware to switch the machine off, or, alternatively, to restart. And if we choose restart, then it's back to the beginning.

4 — Review of Concurrency Control

Concurrency Control

When discussing concurrency and synchronization in previous courses we talked about two important concepts: the semaphore and the mutex. We used them for multiple scenarios including mutual exclusion (preventing race conditions), making one thread wait for another, or managing space in a buffer with the producer-consumer problem. We'll review the basic theory and usage functions for our four favourite mutual exclusion constructs: the mutex, the semaphore, the readers-writers lock, and the condition variable.

Yes, you can, to some extent, get the program behaviour that you want with *just* the semaphore. The other sorts of concurrency control constructs will make it easier to accomplish the goals and harder to make a mistake.

The purpose of the concurrency control constructs is to ensure the correctness of the program. We use them to prevent race conditions, but also to ensure that the program executes in the order that we intend. We said it's a race condition where you have unsynchronized concurrent accesses to the same memory location, where at least one of those accesses is a write. Those situations will fall into one of the following categories, which you may also see referred to sometimes as "hazards":

		Second Access	
		Read	**Write**
First Access	**Read**	No Dependency Read After Read (RAR)	Anti-dependency Write After Read (WAR)
	Write	True Dependency Read After Write (RAW)	Output Dependency Write After Write (WAW)

One of them is not real, though: there's no read-after-read dependency, because it doesn't meet the definition we just covered: in read-after-read, none of the accesses are a write so there's no issue. And indeed, if a global variable is never changed then concurrent accesses to it are not a problem. It appears in the table for completeness; otherwise there might be a question about why that box is missing or blank.

The other situations are all easy to imagine. The true dependency is likely the first one we think of when imagining a dependency: writing the result of a previous stage of the calculation must go first because the value will be read as the input to the next stage. The output dependency is also common: we don't want the older version of the weather forecast to overwrite the more up-to-date one. The anti-dependency is perhaps the least likely one to imagine when we talk about hazards, but it is what happens when we cannot overwrite thee contents of some message buffer until that data has been consumed. You can frame other scenarios of program coordination as one or another of these dependencies.

It's important to remember that all of the forms of concurrency control that we talked about in the previous course are what is called *advisory*. As I like to say in the analogy, they are like seatbelts in a car: they only work if you use them correctly. For example, nothing prevents the program author from accessing shared memory without holding the appropriate lock, and nothing compels them to unlock it when they're finished. They are supposed to. The program doesn't work correctly if they don't. But nothing forces them to. Other programming languages (like Rust!) make it harder to forget to unlock things and forbid access to things protected by a Mutex. But this is C, so, the onus is on the application developer.

The car analogy works here too: the OS is like the manufacturer: it provides seatbelts in the vehicle (concurrency control constructs) and even explains in the owner's manual how they work, but the manufacturer can't force everyone to put them on. The operating system does not second guess the application developers' choices, even if

it leads to wrong outcomes or concurrency issues.

The Mutex

Although we spent a lot of time talking about the semaphore, in many scenarios, all we really need is the mutex functionality to prevent unwanted concurrent access to the same memory location(s). Whereas semaphores talk about wait and signal/post, most commonly in discussions we talk about locking and unlocking the mutex.

The mutex has two states, and only two states, locked and unlocked. There's no need to specify an initial value or anything to that effect.

```
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. 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 [Bar14]. Why? This is a binary sort of construct (locked or unlocked), so call to unlock does not mean that two threads that try to lock it will proceed; in this sense it is not like a semaphore.

The intended behaviour is that the thread that locked the mutex is the same the one to unlock it later on. There are scenarios where it's okay to send a locked lock over to another thread and have it take some action and then unlock it, but that's rare and harder to get right.

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.

To destroy a mutex, use `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.

The Semaphore

You remember, I'm sure, that in UNIX the kind of semaphore 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 as the initial value. 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.

In the previous course, the introduction to the semaphore allowed for the possibility that the mutex could have a maximum value. UNIX semaphores do not give you this option, so if you call `post` 300 times on the semaphore whose initial value was 5, it doesn't matter if those 300 post calls result in a final value of 305.

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. Remember that the increment and decrement of the counter are unconditional. Things are a bit different in the mutex, but we didn't need to know so much about that to use it.

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. The closest we get to that is the trylock behaviour, which will acquire the lock if it can, but it still doesn't really tell you the current value. We learned previously that the reason is that the attempt to "check" the value might return you the current value, sure, but by the time you get that answer back and decide to do something with it, the value could have changed, making the answer useless.

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 multiprocessor system), or another unrelated thread may be the one to continue execution. That's an operating system (scheduling) decision.

Another possible misconception around semaphores relates to posting on the value when the value is negative. If the semaphore's value is currently -3 and there are 3 threads waiting, a call to `post` will both increment the value to -2 and unblock one of the threads waiting. Sometimes there is an expectation that the counter must get back to 0 before any waiting thread is allowed to proceed, but it's easy to show that that kind of semantics would result in a deadlock in situations where multiple threads must wait their turn before proceeding.

Unlike the mutex, there's no requirement for the semaphore to be in a certain state before it is destroyed. The previous course talked about the reusable barrier as generally desirable over the non-resuable versions since it meant there was less overhead of destroying and recreating the semaphores. But efficiency (real or imagined!) was the only motivation there; there was no logical error in choosing the other route.

We used the semaphore in all sorts of synchronization problems. It could be used in place of a mutex, sure, but it can also be used to arrange a rendezvous, allow limited concurrency with multiplex, and we could even use its internal value indirectly in the producer-consumer problem. As fun as it was to imagine scenarios involving making pizza and the Rebellion's attempt to escape Lord Vader on Hoth, we won't be examining all the ways we could use it any further.

The Readers-Writers Lock

Unlike the producer-consumer problem, some concurrency is allowed, recognizing the fact that there is no read-after-read dependency:

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 [Sta18].

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).

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.

And we also saw an example of it:

```
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 );
}
```

We also examined situations where we really did need to enforce writer priority would require us to build our own complicated solution with semaphores and a mutex. There's nothing wrong with that, but it doesn't need to be re-examined here since it's not a system-offered concurrency control mechanism. Or in other words, we won't be examining the details of that in the next topic, since it's just built upon the semaphore and mutex types.

The Condition Variable

The condition variable looks a lot like a semaphore, but it has a few interesting differences. We have the option, when an event occurs, to signal either one thread waiting for that event to occur, or to broadcast (signal) to all threads waiting for the event [HZMG20].

Consider the condition variable functions:

```
pthread_cond_init( pthread_cond_t *cv, pthread_condattr_t *attributes );
pthread_cond_wait( pthread_cond_t *cv, pthread_mutex_t *mutex );
pthread_cond_signal( pthread_cond_t *cv );
pthread_cond_broadcast( pthread_cond_t *cv );
pthread_cond_destroy( pthread_cond_t *cv );
```

To initialize a `pthread_cond_t` (condition variable type), the function is `pthread_cond_init` and to destroy them, `pthread_cond_destroy`. As with threads and a mutex, we can initialize them with attributes, and there are functions to create and destroy the attribute structures, too.

Condition variables are always used in conjunction with a mutex. To wait on a condition variable, the function `pthread_cond_wait` takes two parameters: the condition variable and the mutex. This routine should be called only while the mutex is locked. It will automatically release the mutex while it waits for the condition; when the condition is true then the mutex will be automatically locked again so the thread may proceed. The programmer then unlocks the mutex when the thread is finished with it [Bar14]. Obviously, failing to lock and unlock the mutex before and after using the condition variable, respectively, can result in problems.

One possible scenario where it makes sense to use the condition variable is one in which a thread is partway through some thing and wants to wait until a condition is fulfilled before proceeding. It requires unlocking the mutex and letting other threads proceed, but the condition variable implementation means there's no need exit from this function and restart it and try again – we can just stop now and continue when it's time.

In addition to the expected `pthread_cond_signal` function that signals a provided condition variable, there is also `pthread_cond_broadcast` that signals all threads waiting on that condition variable. It's this "broadcast" idea that makes the condition variable more interesting than the simple "signalling" pattern we covered much earlier on.

It should be noted that if a thread signals a condition variable that an event has occurred, but no thread is waiting for that event, the event is "lost". Because an event that takes place when no thread is listening is simply lost, it is (almost always) a logical error to signal or broadcast on a condition variable before some thread is waiting on it. This is sometimes called the "lost wakeup problem", because threads don't get woken up if they weren't waiting for this. Maybe it's not as dire as that makes it sound, however, because sometimes an event is broadcast but nobody is interested and that's fine.

The broadcast does wake up many different threads, in the sense that they are no longer waiting for the condition variable, but it does not mean they all start running immediately. They each have to wait their turn for the mutex to continue for real. So if you're worried that a broadcast results in many threads immediately running... no need.

The condition variable with broadcast can be used to replace some of the synchronization constructs we've seen already. Consider the barrier pattern from the previous course. There are n threads and we wait for the last one to arrive. Then the last thread signals to unlock the barrier and then each thread calls post to unblock the next thread until all of them are through. This is a lot of calls and maybe it would be better to make it a broadcast instead.

The Implementation

Having seen and worked with these before, we should have a pretty good understanding of what it's like to be a user program making use of these constructs. But how do they really work? The answer is that it's both simpler and more complicated than it might seem.

5 — Concurrency Control Implementation

Concurrency Control Construct Implementation

In the concurrency course and the content just reviewed, we discussed at length the need for concurrency-control constructs and their purpose in ensuring the correct execution of the user program. Still, the introduction to the subject covered ideas about how concurrency control might actually be achieved. There were some solutions that did not work: strict alternation, using flags to indicate whose turn it is next, and more. They all failed for various reasons, usually because the resulting scheme did not provide mutual exclusion, or was vulnerable to deadlock or starvation. Inclusive or.

One possible solution that works in an embedded system or very simple OS is disabling interrupts. This is pretty crude, but it does get the job done – if there are no interrupts, then we don't have process switches as a result of hardware operations completing, or the timer interrupt for switching between threads, for example. That does permit certain bad behaviour, though, where one thread can disable interrupts and then just never re-enable them, guaranteeing that it can monopolize the CPU time. Or maybe it just accidentally exits before re-enabling interrupts and the whole system is stuck. And, of course, if there's an emergency situation, the system can't respond to that either. So disabling interrupts is dangerous even if it might sometimes be effective.

But as we saw, even if every process behaves in a very thoughtful and considerate way, if we have multiple processors, then disabling interrupts will not be sufficient [Sta18]. It is pretty easy to establish a scenario where Thread A executes on CPU 2 and Thread B executes on CPU 5 and each of them ends up in the critical section at the same time. Where we landed was that for the mutex (binary semaphore) we could use 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;
  }
}
```

Because it is done in the single machine instruction, this works no matter how many CPUs are executing the same code in different threads concurrently (and there's some hardware magic around caches and cache coherence, but that's for another course). An example of the code that uses the `test_and_set` routine:

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

So, no matter how many threads are executing the code above concurrently, only one will succeed in actually setting the value to 1 and will break out of the while-loop. And when the thread is finished, it sets the variable back to zero which should allow the next spinning thread to continue.

You will notice here that the assignment of `busy = 0` doesn't use something like a test-and-clear instruction or similar. If you wanted to use one it would make sense, but you don't really need it: if only one thread is ever in that critical section at a time, only that thread will be modifying the variable `busy`. Because the other access are atomic they will either succeed or fail in one step – and either the value is 0 or 1 (it won't be somewhere in between) – so this is grudgingly okay. That doesn't work for all scenarios, of course, and an analysis tool like Helgrind might call this out as being a possible race condition. So probably we shouldn't do that in reality.

Compare-and-Swap. That works okay for the mutex scenario, but doesn't work for the general, counting, semaphore where the values can be things other than 0 or 1 (locked and unlocked). For that we have the compare-and-swap, or sometimes it's called compare-and-exchange, instruction. Like test-and-set, it is implemented as a hardware instruction that is completed in one cycle and is uninterruptible. As before, it is not implemented like this, but a precise C-language definition looks like this:

```
int compare_and_swap( int * value, int old_value, int new_value ) {
  if ( *value == old_value ) {
    *value = new_value;
  }
  return *value;
}
```

And to make use of it in trying to decrement a semaphore:

```
int old = 1;
while (true) {
  int actual = compare_and_swap( sem, old, old - 1 );
  if ( actual == old ) {
    old = old - 1;
    break;
  } else {
    old = actual;
  }
}
/* WAIT FOR OUR TURN */
/* critical section */
while (true) {
  int actual = compare_and_swap( sem, old, old + 1 );
  if ( actual == old ) {
    break;
  } else {
    old = actual;
  }
}
```

The CAS routine will change the value of the integer `value` from `old_value` to `new_value` if it succeeds and make no change if it did not succeed. Why might it fail? It might fail if some other thread has modified the value in the meantime. That's why we get the actual value back as the return statement of the function: so we can update the old value in the current thread. If we wanted to change it from 1 to 0 and we find it's already 0, then we just need to try again changing it from 0 to -1. That might also fail, but we will eventually succeed, even if it takes an arbitrary amount of time. It might be possible for a thread to be so unlucky it never gets a turn, but let's just say that this does not happen. And we could prevent that risk entirely if we use an alternative approach below.

The initial guess for `old` can certainly be wrong; the initial attempt to set it will fail and we'll get the correct value. Take note also the need to do the compare-and-swap operation to increment the value as well – otherwise we'll have race conditions there. That wasn't a concern for the test-and-set approach because in that case because the only thread trying to change it from 1 to 0 was the one that got in, but this still relies on busy-waiting.

Alternative: if there is appropriate hardware support, you could use a simpler version of this where you just attempt to do an atomic increment or atomic decrement. That prevents the scenario where multiple attempts are necessary to get the value. Just make sure to choose the correct atomic primitive that returns the *new* value, since that will determine whether the calling thread should get blocked or not.

Regardless of whether we use the atomic decrement approach or the compare-and-swap, there is the big `WAIT FOR`

`OUR TURN` comment there. That's covering up the fact that unlike the test-and-set approach, succeeding at the compare-and-swap routine execution doesn't mean it's actually the current thread's turn – just that it successfully updated the counter value. We actually need to wait until we know for sure that this thread is permitted to proceed.

How do we wait for it? That part's the job of the operating system. Where we left things off in ECE 252 was that sometimes we don't get the result that we want and if it's not our turn, then the operating system blocks the process. No real details were provided as to how that happens. That sort of hand-waving of OS magic really just put off handling this scenario to later, except later is now.

Blocking and Unblocking

So the caller has tried to lock the mutex or wait on the semaphore; each of those was a system call. Inside the system call it uses one of the methods above – test-and-set, compare-and-swap, atomic operation – to do the assessment of whether the thread can enter. But it happens in the system call and most likely inside the kernel code. It doesn't have to be in the kernel, but it does have to report a negative result to the kernel, at the very least.

If the caller used a try-lock or try-wait sort of call, then they don't get blocked no matter the outcome; if they would have been blocked, simply send back a response that says "no" (usually, a non-zero return value) and maybe increment the counter again (depending on how the call to try-wait worked). Then the implementation is done and you can skip the remainder of this section.

If the thread is supposed to be blocked, that's not a very complex operation from the point of view of the operating system. Change the status of the calling thread to be blocked (so that it is not ready to run), and then choose another thread to continue execution. Which one? Scheduling decision, as you know, except scheduling is *also* something that we hand-waved away in the past and are about to find ourselves forced to revisit. Anyway, back to blocked threads.

Marking a thread as blocked or moving it to a "blocked" queue is insufficient, though. We will need some way of knowing that this thread is blocked on the particular semaphore or mutex it was accessing – hence, an implementation of blocking and unblocking will require, perhaps, that the concurrency control construct has an associated queue of the threads waiting for it. Or maybe when a thread is blocked there's somewhere in its PCB that says what it's waiting for. At that point the thread can just wait until its turn arrives.

Eventually, whichever thread did lock the mutex will want to unlock it or some thread will post on a semaphore. If it's a counting semaphore, we always increase its internal counter by 1 and will unblock a waiting thread. For the mutex, we'll only change the internal counter from 0 to 1 if there is no thread waiting; otherwise the only thing to do is unblock a waiting thread. That particular implementation is not the only way to go about it, but it's important not to just set the mutex back to unlocked and forget about whatever threads are waiting – or worse, set it back to unlocked AND unblock one of the threads and allow it to proceed. Either one of those outcomes would ruin the mutual exclusion behaviour we want.

This prompts immediately the question of what thread should be unblocked when an unlock or post event occurs. Again, that's a scheduling decision, but you could choose a simple and "fair" approach of just taking the first thread in the queue. This might not be optimal, because it ignores things like thread priorities, but it does work.

The first-come-first-served approach does prevent the possibility of starvation, as each thread will eventually get a turn. A more advanced system could consider thread priorities, to be sure, but cannot be based solely on thread priority, because that would introduce the risk that threads with lower priorities never get a chance. A really advanced system could even consider things like whether a deadlock is possible or more likely. Just keep in mind that even commercial operating systems don't do this, so the simple approach to dealing with deadlock of ignoring it is the most popular one.

The thread that's unblocked is marked as ready to run again (moved to the ready queue, perhaps?). And at this point it is again a scheduling decision as to which thread continues execution. But we knew that from doing all this from the application developer point of view.

When a thread is unblocked after waiting for the mutex or semaphore, it resumes its execution at the return of the system call and will proceed as expected. So it turns out, the actual implementation of blocking and unblocking of

the various threads to have concurrency control is rather straightforward for both the semaphore and the mutex scenarios.

Readers-Writers Locks

As you know, the readers-writers lock allows for multiple read-threads to execute concurrently, but only one writer and when the writer is in the critical section, no readers. We built up the concept of the readers-writers problem using just the basic semaphore and mutex constructs, but it's probably more efficient to just have a self-contained construct that meets the goal.

A simple counter showing the number of threads in the critical section would not really get the job done. Why not? Well, if the counter is currently 1, is that a reader or a writer thread? We don't know. So let's have two counters, shall we? One for readers and one for writers. But things get interesting depending on whether you care about writer priority.

Let's imagine a scenario where we do not care about priority for writers. So if a writer wants to enter and there's a writer or n readers, the writer is blocked; otherwise it can proceed. If a reader wants to enter and there's a writer, then the reader is blocked, but if there are readers then there's no other issue.

At first we might consider two counters; one for readers and one for writers. This may not be ideal, though, because if the readers counters is 7, is that seven readers currently in the room or seven readers waiting to enter? It's hard to know and we'll need some more accounting.

Since we talked about the light bulb analogy for the way of understanding the reader-writer lock, then maybe the best thing is to have a boolean variable that says whether the lights are on. Then a counter could be used to keep track of the number of readers currently in the room. The reader-writers lock can have associated reader and writer queues.

If a reader wants to enter the room, try to change the light switch from 0 to 1 with a test-and-set instruction. If this thread succeeds, it is the first reader and may proceed, increment the reader counter, and will not get blocked. If the thread fails, maybe there's a writer in the room? The way to know that would be to consider the state of the reader counter: if the counter is zero then there's a writer and the reader should get blocked. Otherwise, increment the counter of readers and proceed without getting blocked.

When a reader is done, decrement the count of readers. If the reader count falls to zero when this reader is done, unblock a writer if one is waiting, or otherwise set the light switch to 0.

If a writer wants to enter the room, try to change the light switch from 0 to 1. If the writer succeeds, it can proceed. If it fails, block the writer. When the writer is done, unblock either all waiting readers or a writer. Note that it's *all* the readers, because any number of them might be waiting at this point.

As we can see, there's some ability to give priority to writers based on what happens when a writer exits, if we always prioritize writers. That might be slightly impractical since there is a risk for readers to starve in that scenario, but readers-writers locks work best in scenarios where writes are less common than reads so it's unlikely to happen in practice.

With that said, the thing we actually wanted to see for writers to have priority is that when a writer is waiting, no new readers are allowed to enter the room. How would we go about that? It's easy enough: when a reader wants to enter the room, take a look at the state of the waiting queue for writers. If there is one, even if we would normally let the reader proceed, block the reader thread.

You've no doubt noticed that things have become complicated here, because the logic requires us to try to change the light switch and also to maybe modify the counter and then decide based on that. This is not atomic, so we'll need some other things that provide atomicity guarantees. So it might actually make sense to use a mutex before modifying flag and counter, but just don't forget to unlock that mutex internally before blocking the thread.

In the previous course, we discussed the idea of the spinlock and the variant of it, the readers-writers spinlock. We discussed the implementation as looking something like this:

Counter	Flag	Interpretation
0	1	The spinlock is released and available.
0	0	The spinlock has been acquired for writing.
n $(n > 0)$	0	The spin lock has been acquired for reading by n threads.
n $(n > 0)$	1	Invalid state.

I think we could say that this validates the idea of how to implement we've discussed is plausible. In this case, flag being 1 indicates the lights are off and flag being 0 means the lights are on. This, like the side of the road we drive on, is one of those choices where you could choose either convention and it would be fine. It's not what I would choose (or did choose in the earlier example) but it doesn't mean this is wrong.

Condition Variables

Condition variables have a lot of similarities, but their semantics are different. Remember that we can have the lost-wakeup problem, but also that the condition variable is always paired with a mutex.

When a thread waits on a condition variable, it must be holding the mutex it has previously acquired to call the wait function. This releases the mutex, or unblocks some thread that was waiting for it. But the current thread always will get blocked, unconditionally, and will be waiting in the condition variable's queue.

If a thread signals on the condition variable, if any one thread is waiting, it is removed from the blocked queue of the condition variable, but it goes into the blocked queue for the mutex. If a thread broadcasts on the condition variable, then all the threads waiting in the condition variable queue are moved to be waiting for the mutex. Whether it's one thread or many, waiting on the condition to be fulfilled doesn't mean that they execute immediately. That would actually be bad.

Notice that no manipulation of the internal counter takes place as there is no internal counter to speak of: the behaviour of the conditional variable as described has no need for one. In a condition variable scenario, if no thread was waiting when a signal or broadcast takes place then no thread gets woken up. Hence, the lost-wakeup problem: if things happen in the wrong order somehow, then the signal is not received by any thread and a thread could be waiting forever. That might not be what you want. This isn't to say that condition variables are useless, but they fulfill a different purpose than just semaphore with broadcast.

So, the functionality as described here does not require a counter, and in fact, it would be incorrect to have one. In a semaphore, if the call to post takes place first it increments the counter and the thread that calls wait doesn't get blocked. So we cannot just use the semaphore implementation as the basis for the condition variable implementation. But that's okay, this is simpler anyway.

6 — 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[2]. 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.

[2] And now you know why it was called the Commodore **64**.

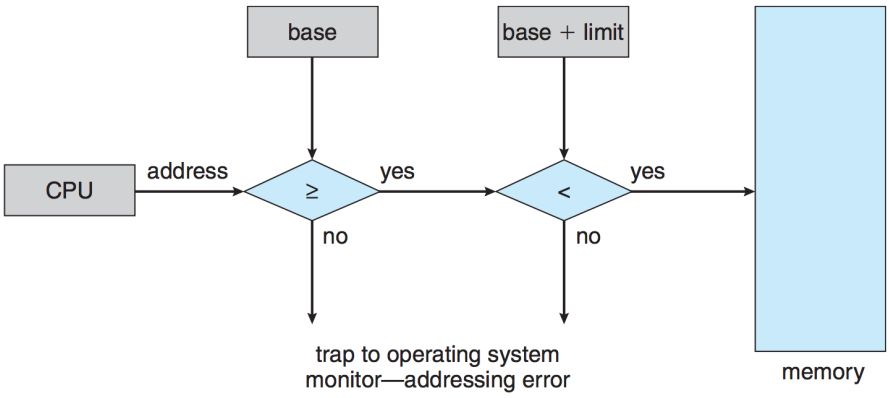
A program executed an instruction directly on a memory address, such as writing 385 into memory location 1024. Suppose you wanted to have two programs running at the same time. Immediately, a problem springs to mind: if the first program writes to address 1024, and the second program writes to address 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.

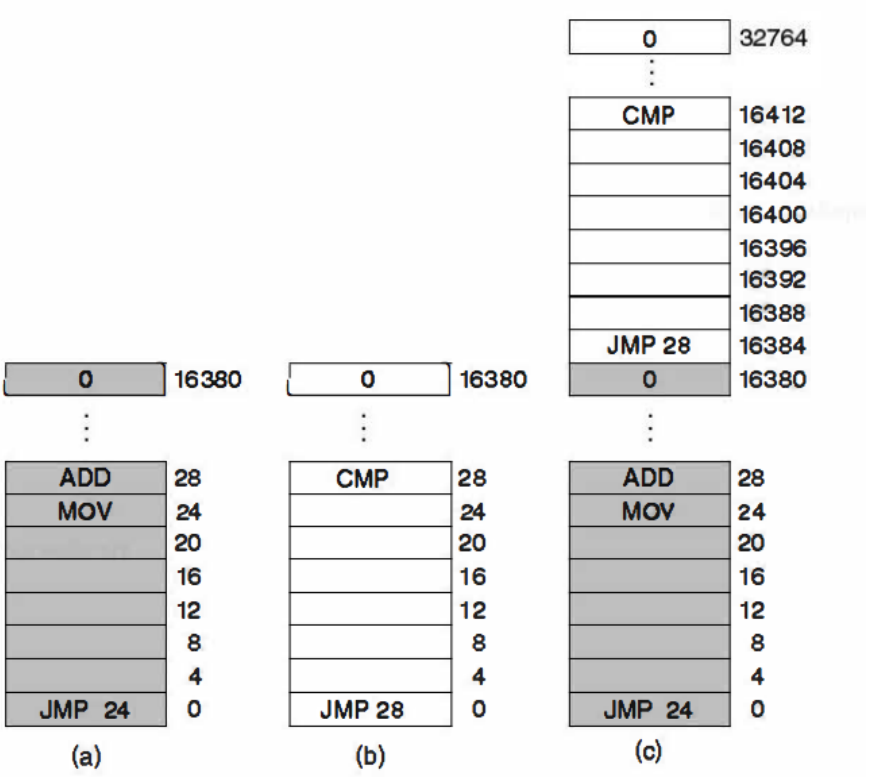
We can attempt to solve the problem of protection by keeping track of some additional information. The IBM 360 solved this problem by dividing memory into 2 KB blocks and each was assigned a 4-bit protection key, held in special registers in the CPU. The Program Status Word (PSW) also contained a 4 bit 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]:

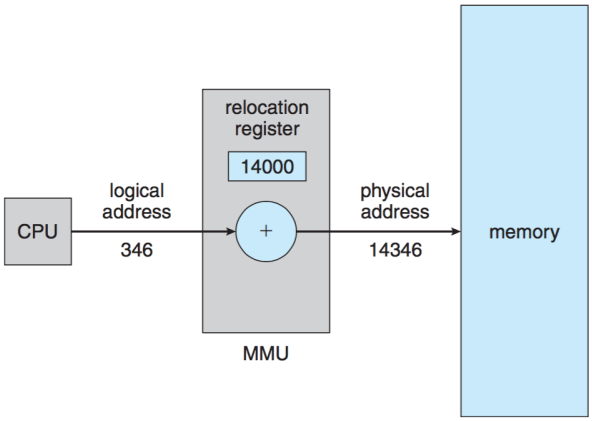
1. **Compile time**: the solution we saw first with commands like `JMP 28`. If we are certain where the process will be loaded into memory, at compile time we can convert those variables to address locations. This is what happens in assembly, and in the MS-DOS .COM format (like `command.com`).
2. **Load 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.

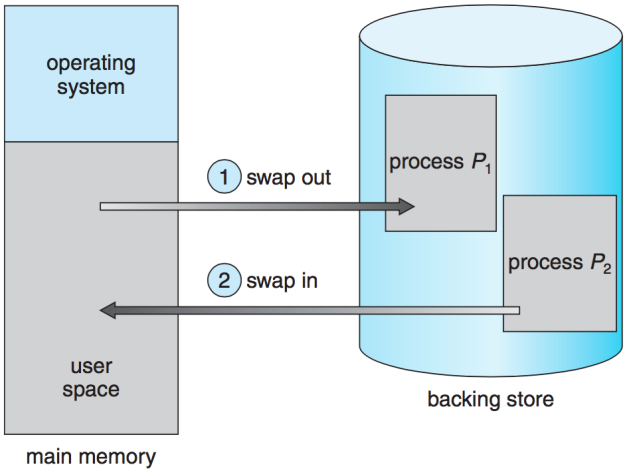
This scheme also gives us something new: we can relocate a process in memory if we change the relocation register's value accordingly. A process that is currently loaded into memory with a relocation register value of 14000 can easily be moved to another location. Copy all the memory from relocation 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[3]. So every memory access has a penalty associated with it to do the addition of the relocation register value to the issued CPU address.

[3] 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].

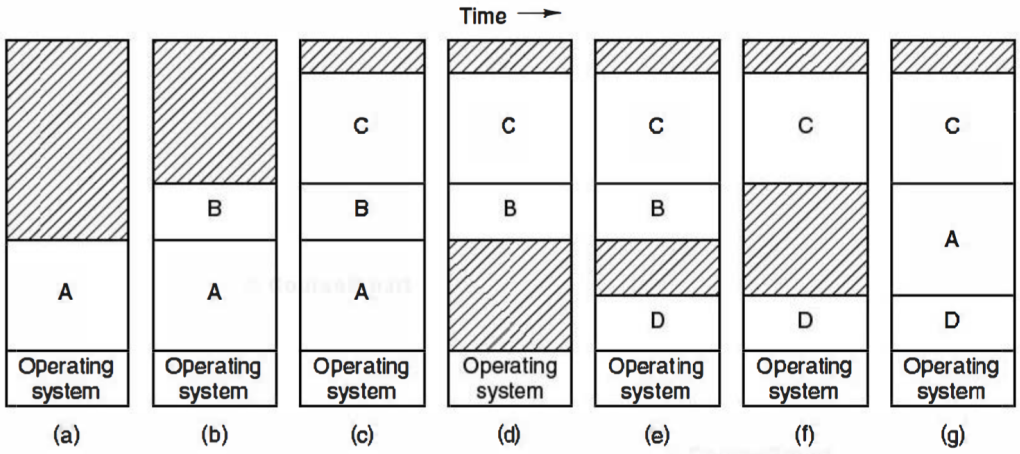
Unfortunately, swapping a process to disk is very painful. If the process is using 1 GB of memory, to swap a process out to disk, we need to write 1 GB of memory to disk. To load that process back later, it means reading another 1 GB from disk and putting it into main memory. If 1 GB strikes you as ridiculous 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.

7 — 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.

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 [HZMG20].

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 [HZMG20].

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 [HZMG20].

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 [HZMG20].

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 [HZMG20].

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 [HZMG20], 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 [HZMG20]:

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 [HZMG20]. 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.

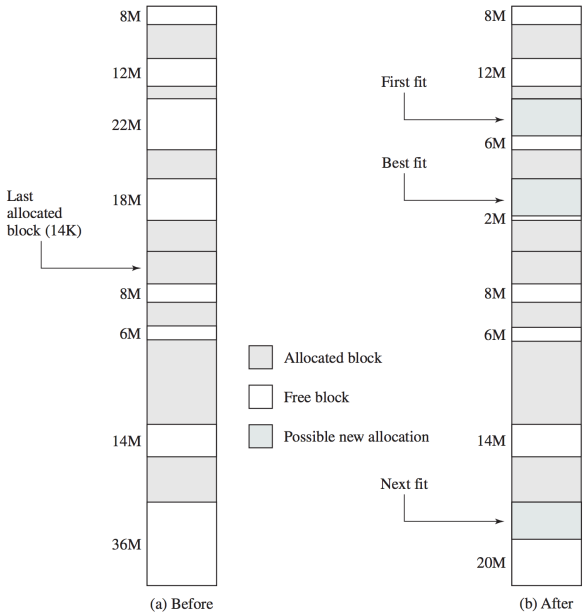
This would require either (1) checking every available block ($\Theta(n)$ runtime); or (2) keeping the blocks sorted by increasing size ($O(n)$ runtime). If we use an AVL tree or red-black tree, then we can get best fit to run in $\Theta(\ln(n))$, possibly better runtime than first fit [HZMG20].

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 [HZMG20].

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 [Sta18].

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 [Sta18], 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

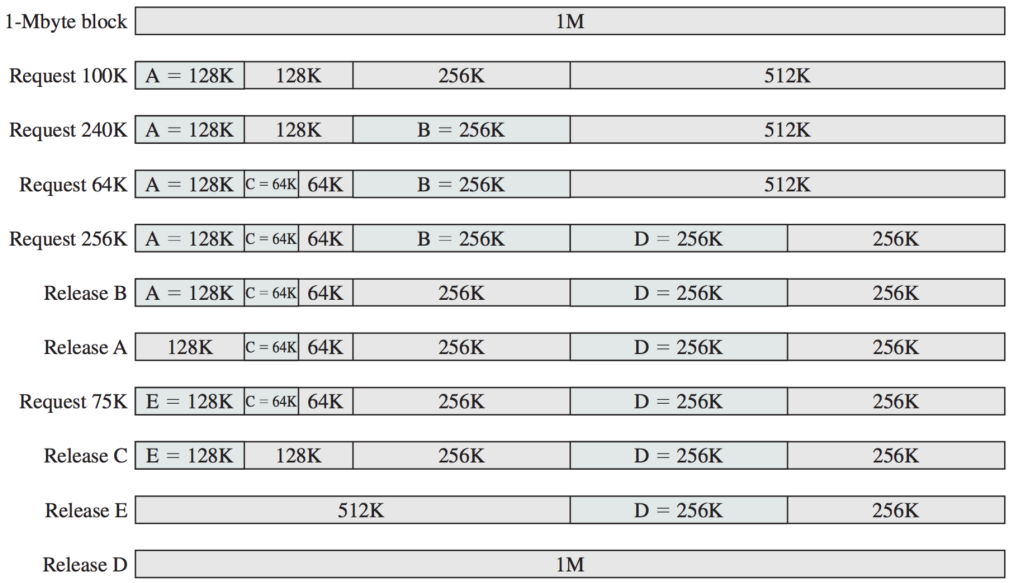
Now let us examine a compromise between fixed and variable allocation, as laid out in [Sta18]. There is some internal fragmentation, but it is a trade-off against how much external fragmentation we are willing to accept.

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 [Sta18].

8 — Memory: Segmentation and Paging

Memory Segmentation

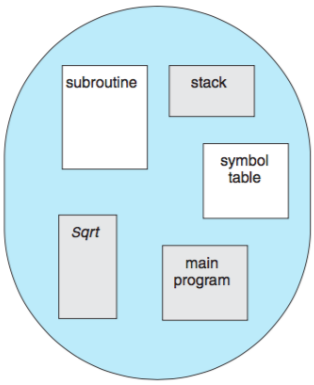
Though you've been repeatedly told that memory is a linear array of bytes, you have also likely been told that there's the stack and the heap, libraries and instructions. Both of these are true; they are simply views of memory at two different levels of abstraction. Each of the elements such as the stack, 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]:

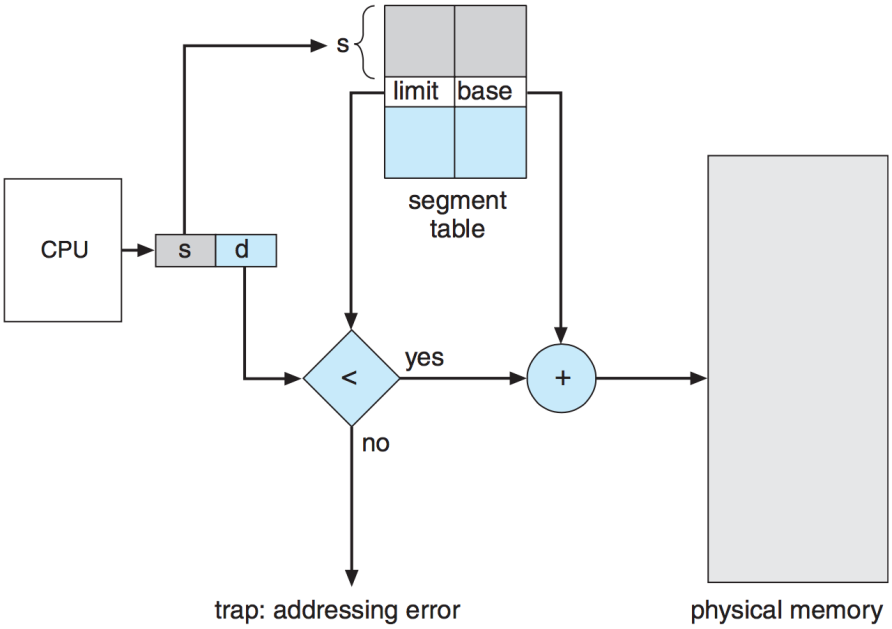
1. The code (instructions).
2. Global variables.
3. The heap.
4. The stack (one per thread).
5. The standard C library.

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

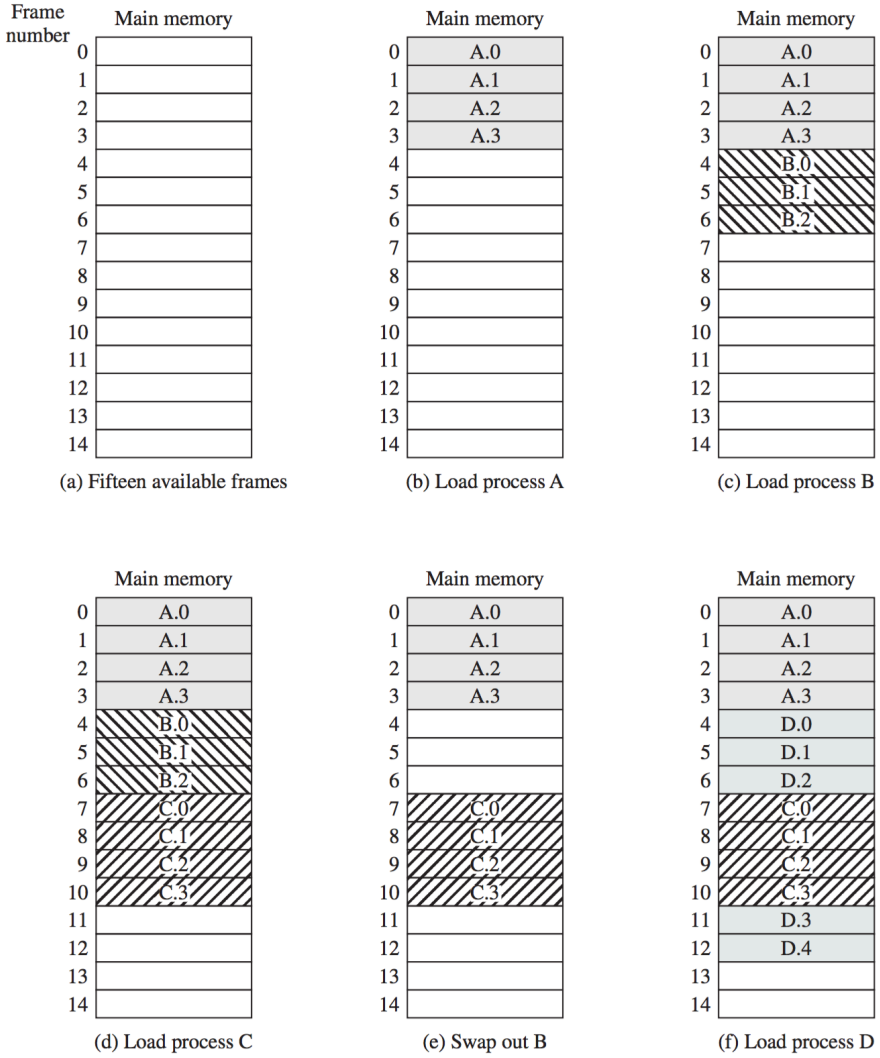
Fixed and variable sized partitions suffer from fragmentation, whether external or internal. Let us divide memory up into small, fixed-size chunks of equal size, called *frames*. Each process's memory is divided up into chunks the same size as a frame, called *pages*. Then a page can be assigned to a 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.

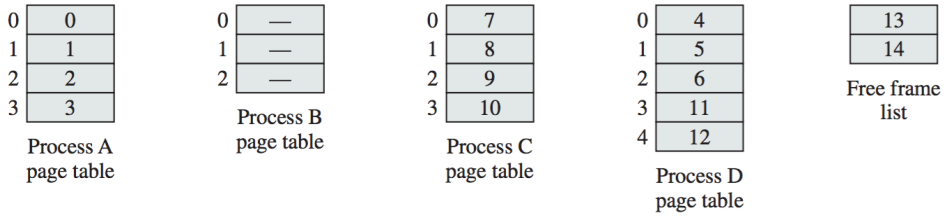
Pages provide the benefit of separating the logical address from the physical address: programmers may pretend the address space of the computer is 2^{64} bytes rather than however many GB of memory are in the physical machine.

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).



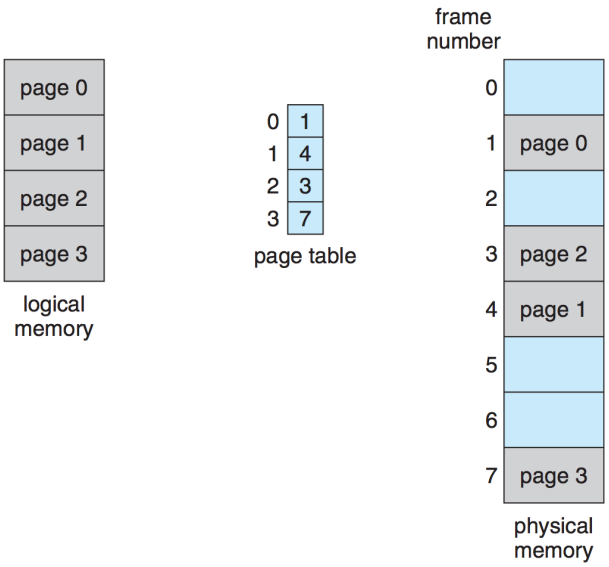
Assignment of process pages into free frames [Sta18].

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 [Sta18].

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 [HZMG20]:

```
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

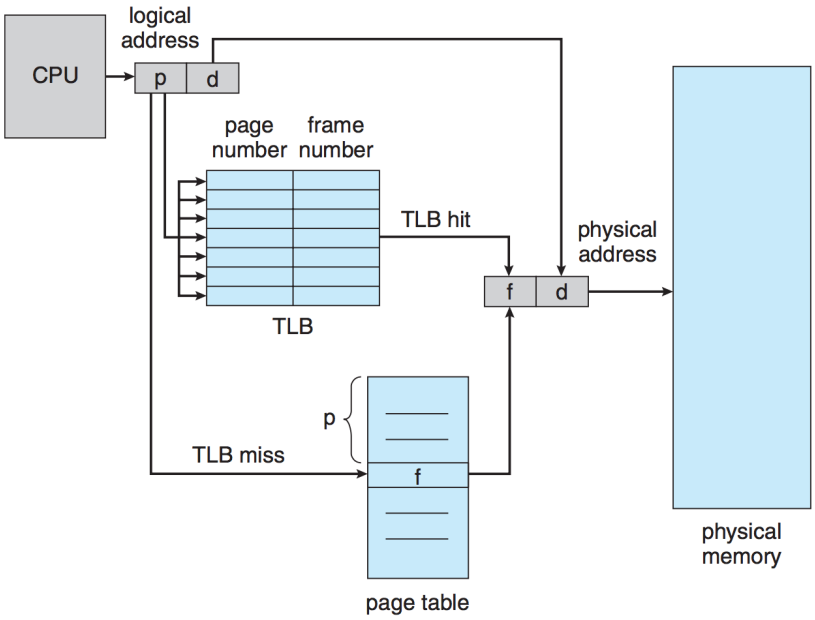
As we have repeatedly discussed, memory accesses are very frequent and often require additions and comparisons. After all, an operation as simple as adding two numbers requires fetching the add instruction, fetching the operands, and storing the result. To prevent abysmal performance, modern computers 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.

9 — 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.

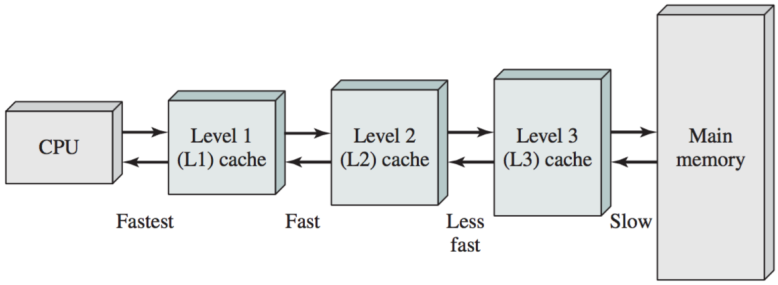
Caches do not have to operate on pages; they can operate on anything, but they are typically blocks of a given size. An entry in a cache is often called a *line*. We will assume for the balance of this discussion that a cache line maps nicely to a page.

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 [Sta18].

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 [HZMG20]:

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[4]. 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[5]. 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)

[4]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.

[5]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.

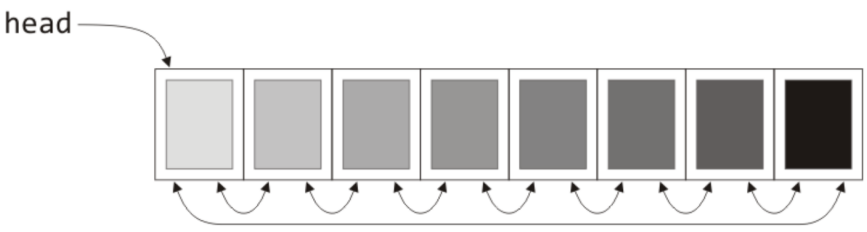
First-In-First-Out. First-In-First-Out is quite easy to understand and implement; the idea is some sort of temporal locality. If there are N frames, keep a counter that points to the frame that is to be replaced (the counter ranges from 0 to $N - 1$). Whenever a page needs to be replaced, replace the 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 [HZMG20].



A cyclic, doubly-linked list in which the `head` pointer indicates the least recently used page [HZMG20].

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 [HZMG20].

10 — Virtual Memory

Virtual Memory

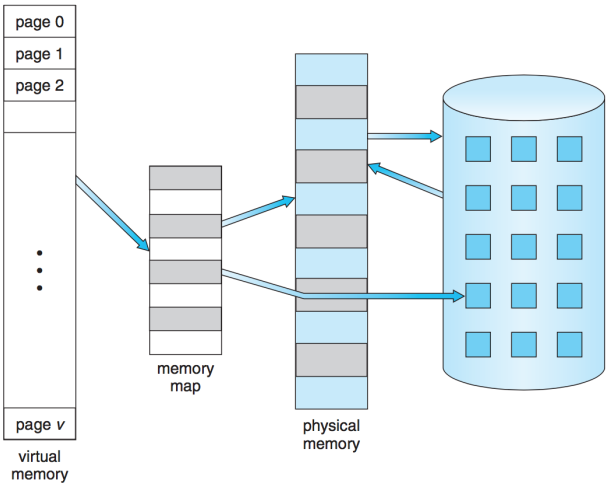
Even with paging and moving pages into and out of memory, there is a limit on what we can do with the system. Specifically, a program that requires more memory than the machine's physical memory cannot run. Maybe that seems ridiculous in the modern era of 4, 8, and 16 GB of memory, but server or supercomputer 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.

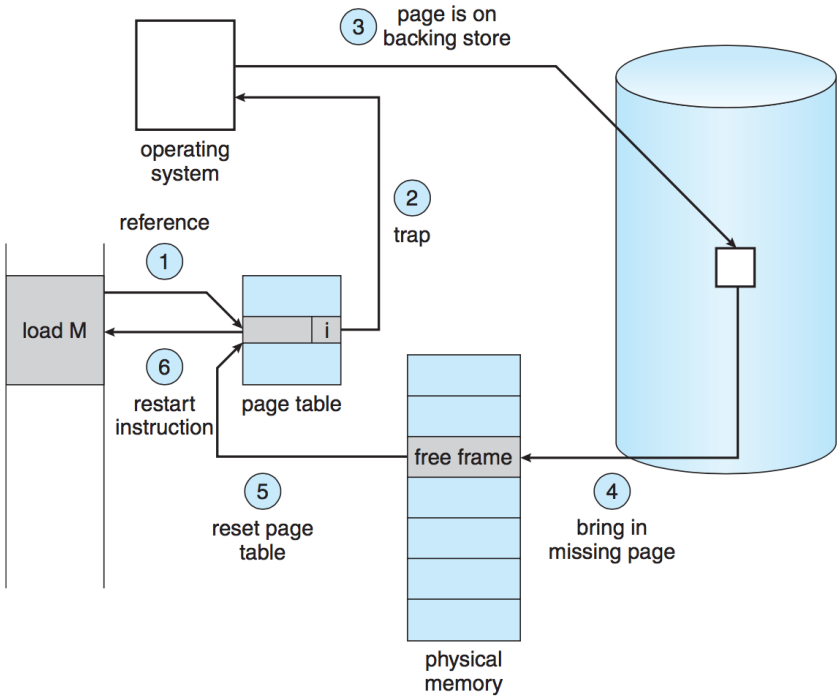
Apparently, if the NeXT operating system, NeXTStep[6] were booted on a machine with only 2 MB of RAM instead of the expected 4 MB, the steady state of the system would be a constant level of swapping [HZMG20]. This was particularly bad because the most common cause of failure in the NeXT boxes was hard 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:

[6] 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[7], 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[8]. 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.

[7] 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.

[8] 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.

11 — 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.

Memory-Mapped Files

When we talked about inter-process communication in a previous course, one option that we covered was based around memory-mapped files. Now we can actually talk about how it works.

It should not be surprising, given what we know so far, that the basic premise is that disk blocks for the file are mapped to memory and if a chunk of the file is referenced that is not in memory, it is treated as if it is a normal page fault and a page-sized piece of the file is brought into memory [SGG13]. In this regard it is then expected that other accesses are just memory reads and writes on the data in memory without going to disk. When the file is closed, then the data is written back to disk in its modified format.

In the one-process-using-it scenario, we don't have to worry so much about synchronizing changes, but in the inter-process communication version it was important to make sure that we chose to synchronize the file because changes might not be visible immediately. If two or more processes have the same file in memory at a time, the memory-mapped file can be in the virtual address space for the two different processes. But remember, we only go and fetch a block of the file for this process if we need it, so some explicit synchronization is needed to be certain that the data is the same for everyone.

Arbitrary Addresses: Memory-Mapped I/O

Another advanced consideration for I/O when we think of memory is the need for memory-mapped I/O – to communicate with a certain hardware device, an area of memory is mapped to the I/O device's memory. This means that although it looks like an ordinary memory read or write to main memory, in reality it might be going over the bus to a totally different device. This simplifies some operations and implementations, because communicating with the device works exactly the same way as communicating with memory – nothing special is needed. Part of the reason behind treating everything like a file in UNIX is to simplify the operations; something similar applies here.

This contrasts with the older-style *port-mapped* I/O where there are different CPU instructions and separate registers used to communicate with the device. It's not that this does not work, but it's just more difficult to work with.

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.

Another possible thing that limits the number of frames that we can move around is the need to lock pages in memory. One possible cause of this locking is an I/O operation: if the I/O has been handed off to a DMA controller, the DMA has a source and destination address and one of those may be a specific memory location and we cannot move those pages to different frames or swap them out [SGG13]. When the I/O operation is started, lock the relevant frame or frames, and they are ineligible for replacement while locked; when the operation is complete, then unlock it/them. Frames is plural because it is possible that the data, even if small, crosses at least one page boundary. We could avoid the need for locking if, however, data reads and writes do not take place in user-level process memory and instead the data is copied to/from some kernel buffers.

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.

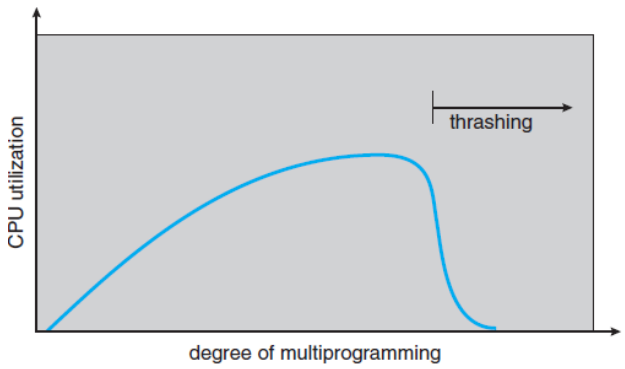
Thrashing

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[9].



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

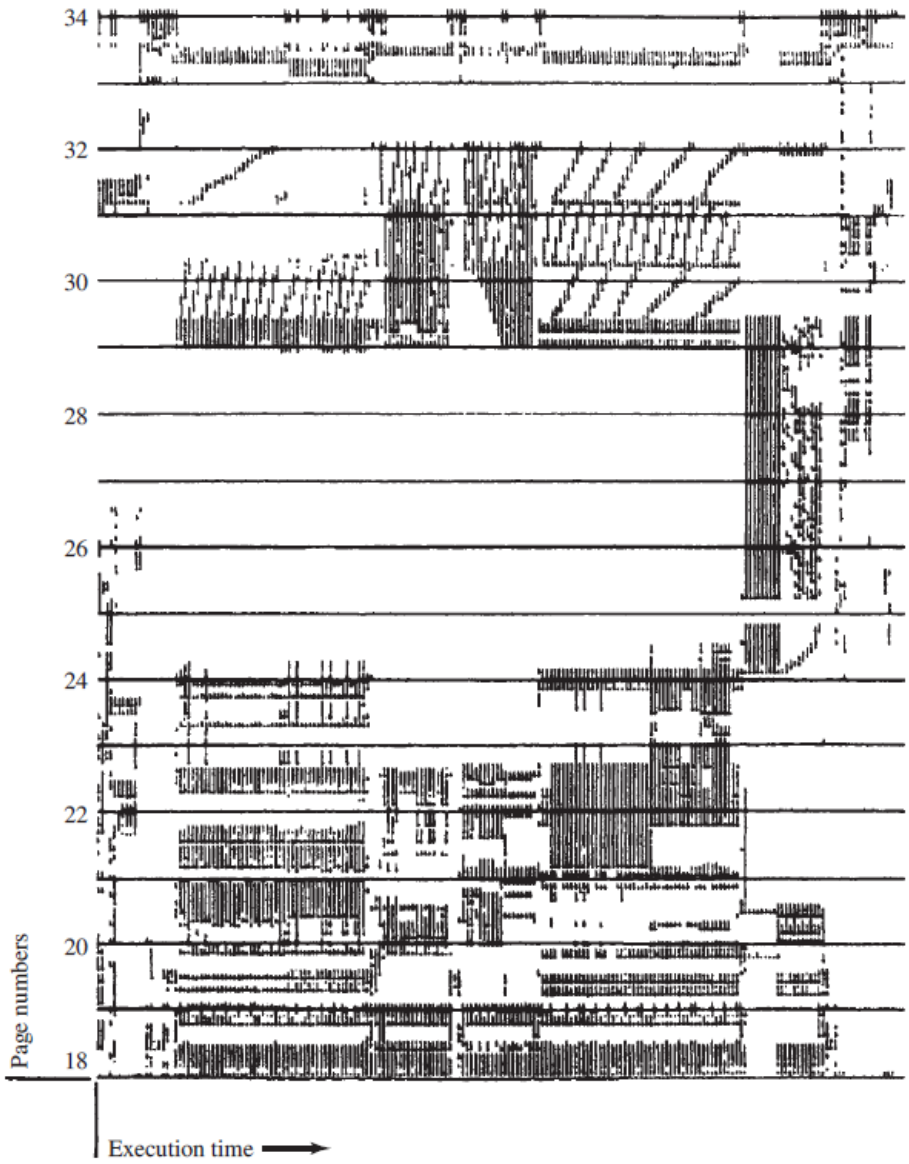
[9] Whiplash! ... If this joke means nothing to you, it's also because I'm old.

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 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 – it is real! [Hat72].

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.

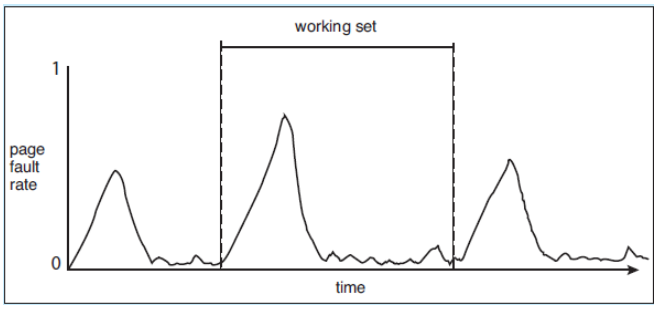
If Δ is too small, the working set will not encompass the entire locale. If it is too large, it will cover multiple locales. Underestimating the size of the locale is bad, because it will mean more page faults being caused. Overestimating is also bad, because it means fewer processes will be allowed 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.

12 — Uniprocessor Scheduling

Uniprocessor Scheduling

As you may imagine, processor scheduling can be very complex when we have multiple threads and multiple processors. So, 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 threads in the system, we are going to have to make some decisions about when different threads run. There are four types of scheduling, but one of the four is I/O scheduling which we will come back to after the scheduling topic in general. The other three bear more examination [Sta18]:

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 3B or 4A?). Short term scheduling is deciding what course you are going to study right now (I'll study ECE 350 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 or laptop 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, nor will it complain.

We do sometimes encounter long-term scheduling, but it's usually a remote service that is deciding to manager its resources. If you are trying to log into certain games on launch day, 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. I used to reference here my experience with the game Diablo III, but I think that's far enough in the past that it is no longer relatable.

Mobile operating systems, notably Android, can be more aggressive about telling you they reached their limits. 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 a long period of time has passed. 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.

The medium-term scheduler doesn't serve as much of a purpose as it used to, because the era of SSDs means that disk transfer times are no longer as painful as they were when we had magnetic drives. So onward.

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 something the operating system invokes frequently and it can make the difference between an excellent user experience and a miserable one. The short term scheduler runs when there's a need, and we'll go over a few different situations that might cause it.

If we have co-operative multitasking, short term scheduling will only take place in one of two circumstances: the currently executing threads yields the CPU or the currently executing thread/process terminates (voluntarily or with an error). We saw previously that yielding is possible but rare. Some ancient operating systems (e.g., Mac OS 9) required processes to yield to make the best use of the CPU. 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 thread, is responsible for deciding when it's time to switch threads, with the exception of when a process or thread decides to voluntarily terminate.

The dispatcher will certainly run when a thread becomes blocked, such as on an I/O operation. If a thread requested a write to the network and is blocked, now the short term scheduler needs to decide what thread runs next. If it gets blocked on a semaphore or mutex, that is also an occasion to switch to another thread. 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 thread is suspended already so why not leave it in that state?).

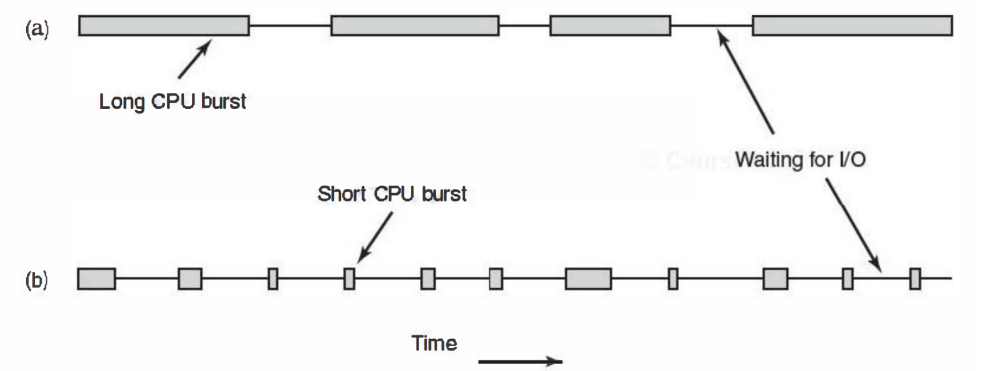
System calls like `fork` and even posting on a semaphore may also provide good opportunities to switch from one thread to another. Again, by invoking the operating system through the system call, the calling thread is suspended (the trap handler runs and the OS takes over). So it is necessary to decide what thread 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 threads waiting on the semaphore will be the one to receive the signal, and even then, which of the signalling thread and waiting thread will resume.

Finally, there is also time slicing. If time slices are defined as t units, if a thread executes for 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 thread to run, so that different threads run (seemingly-)concurrently.

Thread Behaviour: Rate Limiting Step

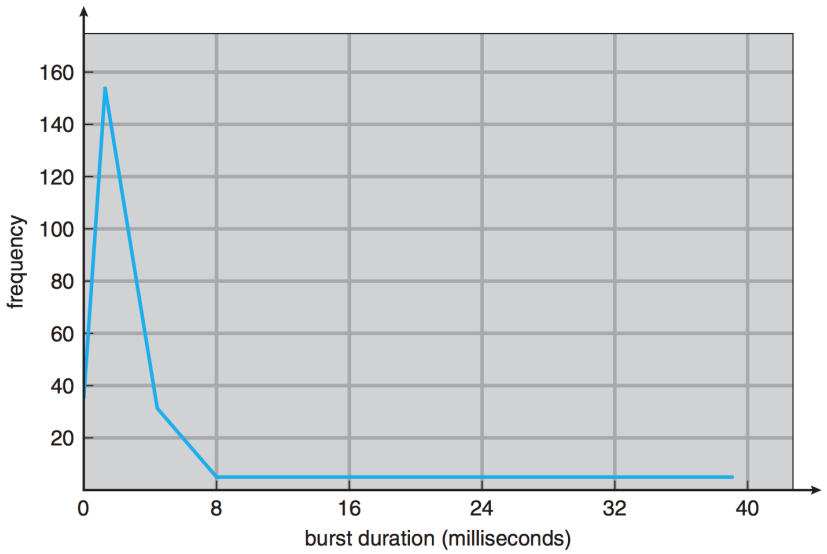
Threads tend to alternate periods of computing with input/output requests. A typical program exhibiting this pattern would be one that loads some data and processes it, before going on to the next part of the data. That means the program will do a lot of I/O to start with, and we call the waiting period an *I/O Burst*; after that we have a period of time where the CPU does a lot of work, called a *CPU Burst*. Then these steps repeat. Not every thread follows this alternation pattern, and the time spent on each is rarely exactly 50-50, but instead a spectrum. How much of CPU or I/O burst a thread does allows for classification of the thread into one of two categories.

Threads 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, or access to more CPUs. The alternative is a thread 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 thread and (b) an I/O-Bound thread [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 the one you can get today is measurably better (in benchmarking tests) than the one from six months ago. I/O standards like Serial-ATA and USB and such change very slowly, over the course of years. When there *is* a jump forward in I/O speed, like USB-2 to USB-3, it's often a large jump. 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 ways we could make use of this. 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 threads are which.

Another is that if the disk is the slowest device and the one for which we spend the most time waiting, when the disk has nothing to do, the short term scheduler should immediately schedule a thread that is likely to issue a disk request. This ensures that we do not waste any of the disk's time.

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), that's something you'd like to know when choosing what scheduling algorithm to implement. 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, but we know that's silly here. So let's come up with some criteria to evaluate scheduling algorithms. Our goal is not to choose the one and only best algorithm, but to understand the strengths and weaknesses of each so we can choose what makes sense for our situation. We will examine the following scheduling criteria [Sta18]:

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 thread 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 my computer to convert a file from FLAC format to mp3 format so I can import it to my phone, 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. 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.

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.

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 threads 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 threads, 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, because that's money that we could be earning. 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 threads 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.

Priorities. Processes and threads can be assigned priorities: a way of saying that this process or thread is more or less important compared to others. Scheduling should respect this, within reasonable limits. One example of priority with reasonable limits is something like priority boarding when flying: those who get priority boarding get to get on the plane first. This has some advantages – if overhead bin space (a resource!) is limited, getting on first helps in getting that – but everyone who has a seat on the plane will eventually get to board and arrive at the destination. The highest priority process/thread should probably run most often or first, but probably 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 threads so both the CPU and I/O are kept busy. If we have more information about what sort of I/O the threads will need, so much the better: we can choose a thread that will use the printer if run, when we know the printer is idle, and avoid choosing a thread 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 (and threads) 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 or thread'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 in England they drive on the left and in Canada we drive on the right.

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. This might come in to play, for example, when two threads are waiting on a semaphore. If process P_3 posts (signals) on the semaphore, the resource of that "signal" may be sent to the higher priority process P_1, allowing it to proceed first.

The operating system or the program author may each be partially responsible for assigning a priority to each of the threads. These priorities may change over time and when certain conditions are fulfilled (e.g., a process that uses a lot of CPU in a short period of time might be reassigned a lower priority by the operating system temporarily).

System administrators can usually change the priority of a thread or process. Regular users may have a say in the priority of their own threads/processes, in 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 hurts the performance of the system and makes users unhappy (even though that's what they explicitly asked for).

In some systems, the highest priority non-blocked thread will always run. This is a great way of making sure that higher priority threads have right-of-way, but a terrible way of ensuring fairness and preventing starvation. We could easily have a situation where low priority threads never get a chance to run, because there are too many threads of middle or high priority. So, scheduling is not necessarily as easy as just finding the highest priority thing to do...

13 — Scheduling Algorithms

Scheduling Algorithms

In the last topic, we saw one example of a simple scheduling algorithm: always choose the highest priority (non-blocked) task, and execute it. It's good to be King, I guess, but that solution has some drawbacks. 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 will maintain some data about each process and this may be used in making decisions about scheduling, depending on the algorithm [Sta18]:

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 the 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 processing systems 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 also have a priority heap, or just one big linked list or array that we keep sorted by priority. However it's stored isn't actually the critical part here.

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)[10], but I never get to them because something else always comes up... work to do, another book to read, a social event...

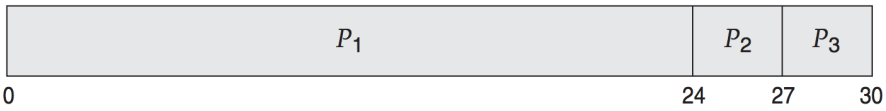
In some systems this is a desirable property. It does not fulfill all short term scheduling criteria, notably response time and fairness. It may be suitable for life-and-safety-critical systems, such as the process to control a robot arm, to prevent a situation where the robot arm goes through the wall and the building falls down and you're dead. Fairness is not as important as the lives of people, in this situation.

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 simpler than 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.

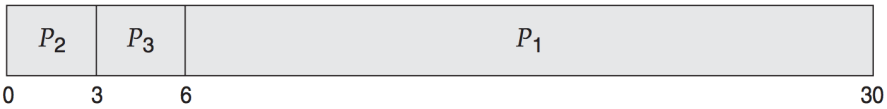
Life is full of FCFS systems and many of them result in a lot of waiting in line: if you call the phone company because they billed you incorrectly – again – then you get the displeasure of waiting in line for a very long time to speak to the next available agent who is currently busy assisting other callers. If you don't want to or can't wait so long, the best you can do is try calling again at a less busy time.

FCFS can result in some suboptimal 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

[10] I first wrote the paragraph about this book in 2015 and guess what, by the time I review and revise this content in 2022 I have managed to read exactly zero pages of this book. I'd say this year is the year, but we both know that's not true.

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 [Sta18].

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

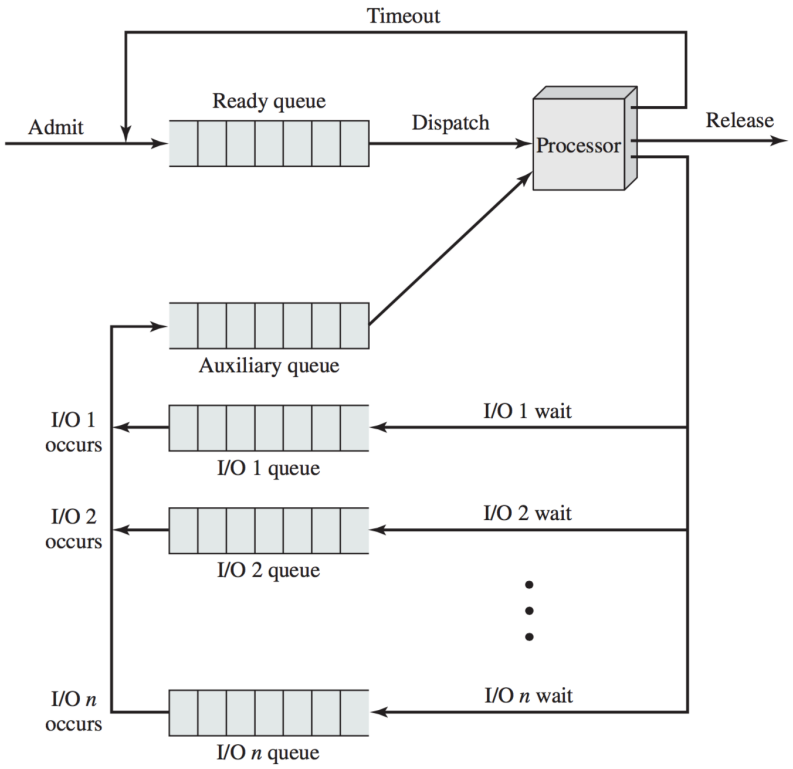
The idea of time slicing has already been introduced: every t units of time, a timer generates an interrupt that is the prompt to run the short-term scheduler. Time slicing itself can be combined with many of the strategies for choosing the next process, but when it is combined with FCFS we get Round 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 [Sta18].

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 [Sta18].

Shortest Process Next

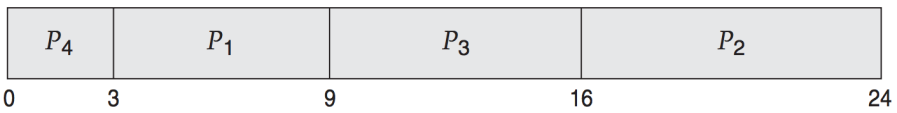
If some information is available about the total length of execution then we may wish to give priority to short processes. We let short processes go first to get them over with quickly. The reason is made clear by the second example of FCFS from earlier; though it still took 30 units of time, total, 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 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, 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 [Sta18]:

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^{n} T_i$$

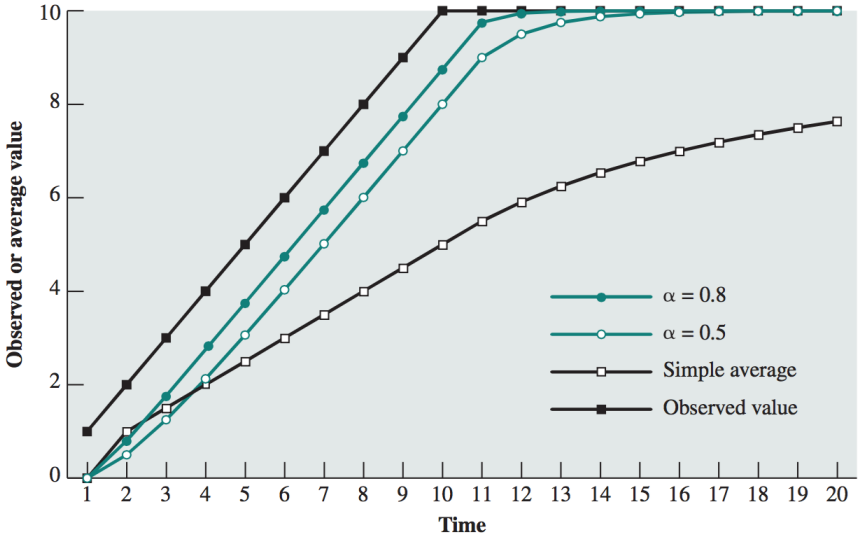
where: T_i is the burst time for the ith instance of this process; S_i is the predicted value for the ith instance; and S_1 is a guess at the first value (not predicted, because we have no data to go on at that point). Instead of picturing it as a sum each time, we can modify the equation to just update the value:

$$S_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

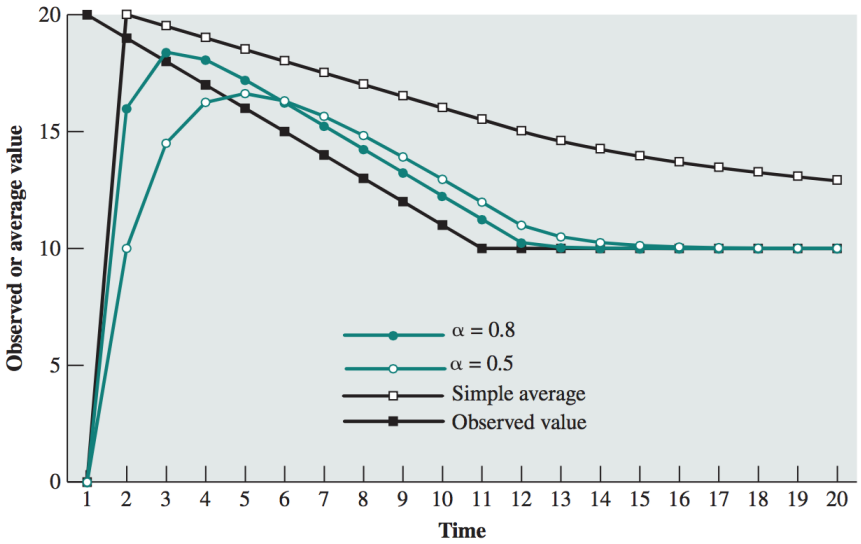
This routine, as you may have guessed, gives each term in the summation equal weight of $\frac{1}{n}$. What we would like to do is give greater weight to the more recent values. The strategy for doing so, as in [Sta18] is exponential averaging. We define a weighting factor α, somewhere between 0 and 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.



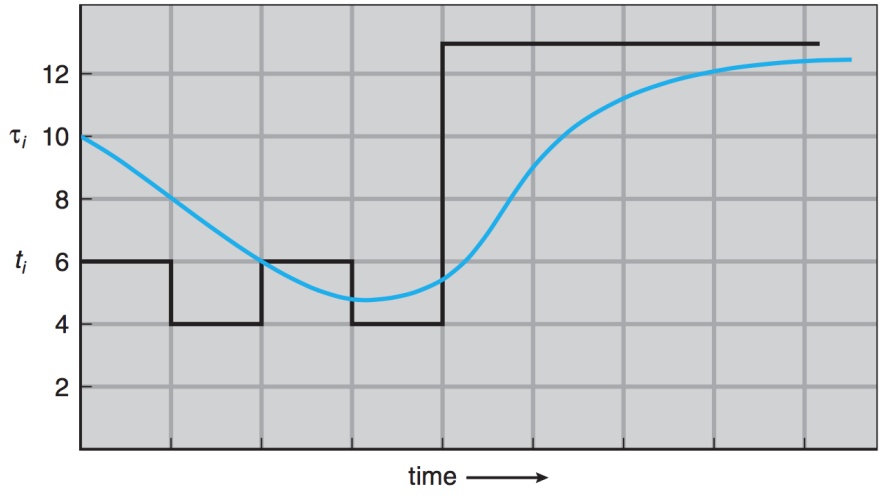
(a) Increasing function



(b) Decreasing function

Use of Exponential Averaging [Sta18].

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... [Sta18]

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...

14 — 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 [Sta18].

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 [Sta18]:

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 [Sta18] 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.

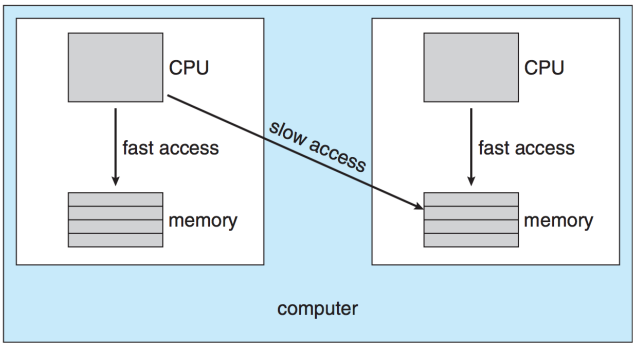
If we choose asymmetric multiprocessing, we have a boss processor and this one alone is responsible for assigning work and managing the kernel data structures (e.g., process control blocks). If instead, the system uses symmetric multiprocessing, each processor is responsible for scheduling itself. We 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].

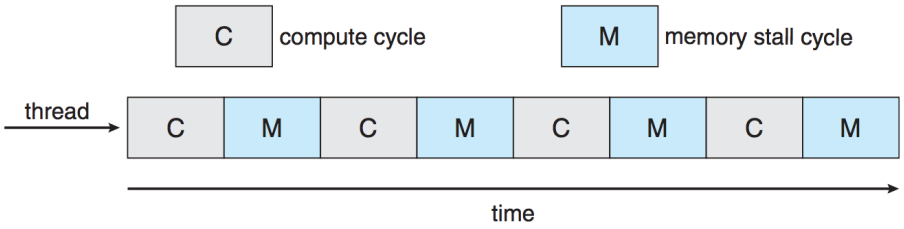
There are two, non-exclusive approaches to redistributing the load: *push* and *pull* migration. It is called migration because a process migrates from one processor to another (it moves homes). If there is push migration, a task periodically checks how busy each processor is and then moves processes 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.

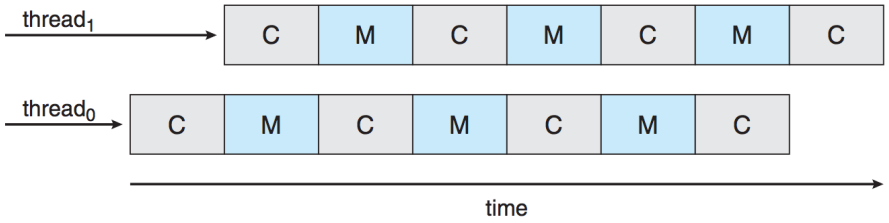
When a processor issues a memory read or write that is a cache miss (so the read has to go to memory), the CPU core can spend a lot of time (perhaps as much as 50%) of its time waiting for that read to take place. We might refer to periods of time where there is computation as a compute cycle, and time 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).

15 — Real Time Scheduling

Real-Time Scheduling

It's been a while since we introduced it, but we started earlier by saying a real-time system 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[11]. There's also a 4th year ECE technical elective on real-time systems.

The term "task" is used to refer to something that needs doing. Yes, the scheduler operates on threads rather than specific things to do, but we will say for the sake of the discussion that each task corresponds to one thread that is trying to carry out some work. When we say that the system schedules a task, please understand it means it schedules the thread associated with that task.

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 [Sta18].

Sometimes there's a distinction between *firm* and *hard* real-time in the literature. In that situation, it's really just about the severity of the consequences. A firm deadline is one in which the response is useless if it arrives a little too late; a hard deadline is one in which the system itself fails if it doesn't meet the deadline. If the Mars rover misses a deadline and drives off a cliff to its theatrical-style doom, that qualifies as a truly hard deadline. If we are calculating the position of a satellite to transmit the data and the answer arrives too late, the location we calculate is too far out of date to be useful, but we can try again, so we might call this firm instead.

Real-Time vs Non-Real-Time Operating System

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. 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.

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.

That's a user-space thing, though. As an example of why the OS isn't suitable for real-time operations, the system provides you no real guarantee as to the length of time it will take to execute a system call. If you want to read from disk, the system call takes an arbitrary amount of time to execute on its own and will also have to wait an arbitrary amount of time to get the data from the disk depending on the speed of the device and how busy it is. We can estimate how long it will be, of course, and we can measure things, but even though it might be okay on average, what about the worst case scenario? Are we sure that will be within some bounded time limit?

[11] Abandon sanity, all ye who enter here.

The answer is that you probably cannot be sure, and if this is a motor vehicle or industrial machinery or similar, certainty about time limits can be a question of life or death.

If we think of other possible reasons why a task may take an arbitrary amount of time, you would likely think of interrupts and concurrency control functionality. Interrupts are supposed to be quick, and we talked about things you can and cannot do in an interrupt handler (e.g., signal) but there's no enforcement of the rule and no time limit on the length of the interrupt handling routine; there's also no limit on the number of interrupts that can be generated in a given period of time. Similarly, when a thread is blocked on a mutex, there could be arbitrarily many threads ahead of it in the queue – even if you used a queueing system that did not have any risk of starvation – so the wait to enter the critical section could accordingly be a very long time indeed.

So it's fair to say that a real-time operating system is just different from a general purpose one. A lot of the difference comes down to how things are scheduled, which is why the topic of the real-time operating system comes up now. To illustrate, let's consider two scenarios in which a hard real-time task could fail to meet 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? We could certainly argue that a more intelligent scheduling decision – start this task earlier – may have made it possible to complete it 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. Again, was scheduling here the culprit? Maybe; if the interrupts were treated differently perhaps it would have been possible to complete the original task.

Properties of Real-Time Systems

Real-time systems are considered to be unique in five key areas [Sta18]:

1. Determinism
2. Responsiveness
3. User (administrator) control
4. Reliability
5. Fail-Soft operation

Determinism. Determinism means that operations are predictable, either performing them at fixed times or within certain time limits. When there are multiple things happening concurrently, perfect determinism is not going to happen. And even for a system that does only one thing, if it is triggered by some external factors, determinism is not possible because of the randomness of life. For most RTOS scenarios, it's sufficient to know that things will happen within some fixed time period, that is, that no matter how unlucky the timing or sequence of events, we can still start the task on time to successfully complete it.

Nondeterminism is not necessarily bad. We learned already about caching and this can mean that some requests take less time to service than others, just because the data happens by chance to be in the cache of the CPU doing the work. If you wanted to minimize nondeterminism you could disable the CPU cache (or use one without cache). That wouldn't make the system faster; on the contrary, it would make it slower, but it would make it more likely that every request takes about the same amount of time. I can't say I really endorse this strategy, because the presence of caching makes the worst case no worse and the best case much better – this sounds like free performance to me! – but there may be a system I haven't thought of where less determinism is actually preferable.

Responsiveness. Responsiveness sounds like it might be the same as determinism, but the key distinction is that determinism is how long it takes before the operating system takes before acknowledging the request/interrupt and responsiveness is how long it takes after acknowledgement to handle it [Sta18]. Responsiveness includes not only the time to execute the interrupt handler, but also the time it takes to start the interrupt handler and what happens if the interrupt is itself interrupted by another higher-priority interrupt.

Administrator Control. Administrator control is often quite different in a real-time operating system and there are two ways that it can go: more control or no control. A non-real-time system usually lands somewhere in between, where users and administrators get control of some things but not everything.

In the no-control scenario, the system takes no instructions from users or administrators and simply runs as it has been programmed and configured. Users have no way to start new processes or tell the system to change priorities. That's pretty reasonable in certain contexts – you should not be able to play solitaire on the fire suppression system, should you?

In the more control scenario, administrators get to choose a lot. The OS itself has no way of knowing which tasks are real-time and which (if any) are not, nor can it know which of the real-time tasks are soft/firm/hard. Accordingly, an administrator must be able to specify what is what, and maybe even be able to make other choices like what scheduling algorithm should be used. The obvious downside is that if this is put in the hands of the user or administrator, they could always get it wrong.

The typical desktop or mobile OS goes somewhere in between; a regular user can typically start processes and even, to some extent, choose their priorities. Administrators can set certain elements of scheduling policy, but maybe not everything, so perhaps they cannot change the scheduling algorithm.

Reliability and Fail-Soft Operation. Actually, these topics aren't as relevant to scheduling for the moment, so let's defer that discussion until we get a bit further in the course. At this point, all we should consider is that fail-soft means that if it's somehow not possible to succeed in completing all tasks, the system will try its best to complete as many tasks as it can, with priority given to the hard real-time tasks.

Scheduling Is Central

Having examined a little of why real-time systems are different than a typical system, we recognize that scheduling is critical to making it work. Choosing a bad scheduling routine where important things are ignored guarantees failure, obviously. However, the kind of scheduling algorithms we have examined so far are not suitable to the task of real-time systems.

In fact, for a real-time system, fairness and minimizing average response times are not important; the real objective is to ensure that all of the hard real-time tasks complete before their deadline and as many as possible of the soft real-time tasks [Sta18]. If the right conditions are present, then success is possible but it is not guaranteed: even an optimal scheduling algorithm cannot make a task that takes 120 seconds complete in 90 seconds, nor can it find a way to complete 400 tasks per day when the system has a maximum throughput of 350.

Any of the non-preemptive scheduling algorithms will not be suitable for a real-time system, because the whole idea is based around prioritization of hard real-time tasks. If one of those arrives while an unimportant task is in progress, it's not sensible for the high priority task to wait until the currently-executing one gets blocked, voluntarily yields the CPU, or exits. Similarly, some routine based solely around time slices probably does not work either, since it would not be optimal to force the higher priority task to wait for up to a full timeslice before it gets a turn. A more realistic approach is immediate preemption, which is just a fancy way of saying that as soon as something more important comes along, the system drops what it is doing to deal with it.

You might wonder why that's so important to drop everything, if we can make timeslices nearly arbitrarily small? Yes, but sometimes the speed of the response we need is in the order of microseconds or a few milliseconds, and there are big consequences for being late, so we need a strategy that recognizes that.

What Kind of Task is It?

There are four kinds of task that are relevant to our discussion: fixed instance, periodic, aperiodic, and sporadic. Each task can be categorized into those four kinds based on both how often it runs and how much time it takes to execute when it does run.

Fixed-Instance Tasks. A fixed-instance task is something that executes a fixed number of times (hence the name) and that is frequently just once, for initialization or cleanup purposes [HZMG20]. One-off tasks like initializing the database are important, but the analysis that we want to consider is what happens at runtime steady-state (after initialization but before a shutdown is called).

Periodic Tasks. If a 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 [HZMG20]:

$$U = \sum_{k=1}^{n} \frac{c_k}{\tau_k}$$

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 8:30-9:50, course ECE 350 (a "task", if you will) takes place in classroom E7 5353 (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. But we'll come back to the subject of this sort of scheduling soon enough.

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.

Aperiodic Tasks. 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) [HZMG20].

Sporadic Tasks. 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?) [HZMG20].

It Takes How Long?

Up until this point we have talked about execution time of a task as if it's a known, fixed quantity, but where do those come from? When the goal is to ensure that hard deadlines are met, we need to know how long tasks take, assuming the worst-case scenario. And no matter how unlikely the worst-case scenario is, that's the value

we choose. We'll consider two ways to estimate the worst-case execution time: source code analysis and empirical testing.

Whatever strategy we choose, we want to over-estimate the expected execution time or tasks. Are there tradeoffs to assuming the worst will happen? Certainly! It's possible that we would calculate that we'd need much more capacity than is actually needed during normal operation. That might mean spending more money or buying a more powerful device and reducing the battery life [HZMG20]. Is it bad to have more capacity than you need?

If you've ever been unfortunate enough to have a bad experience with an airline where one cancelled flight results in all sorts of chaos for everyone trying to travel, it's partly caused by the fact that airlines plan their operations such that there's no (or minimal) additional capacity. Thus, even a minor disruption can easily result in a cascade of problems that affect passengers whose trip is unrelated. That's how you end up with your Calgary to Vancouver flight cancelled or delayed because it's snowing in Nova Scotia; the flight crew that was supposed to travel from Halifax to Calgary didn't leave on time and there's no extra flight crew sitting in Calgary to take you to Vancouver. Thus, it doesn't matter that the weather in your departure destination is sunny and clear. Air travel is important, to be sure, but imagine that we're talking about a life or safety-critical system. Building in some extra capacity is vastly preferable to the alternative.

Code Analysis. One approach for estimating the runtime is through code analysis. This looks at the algorithm and the possible inputs and execution paths and trying to figure out how long the longest path could take since we could work out execution times of each step. That will be an overestimate since it won't account for things like CPU pipelining or a compiler optimization that improves the actual runtime [HZMG20].

Some things that we might do fairly frequently in a non-real time system make it difficult or impossible to accurately estimate the runtime. If you take a look at, say, the NASA/JPL coding guidelines[12], you'll find that they have important rules like: (1) no recursion and no `goto` statements; (2) loops must have a fixed-bound; (3) no dynamic memory allocation after initialization. In addition to each of these things being things that are easy to get wrong, they also qualify as things that make it hard to estimate how long a function is going to take.

The restriction around recursion as well as the fixed-iterations loops make sense, since it's hard to estimate how long a function will take if we don't know how many iterations of the loop are required. The memory one looks a little bit strange though, what's wrong with allocation of memory? As with numerous other system calls, it's hard to know exactly how long it will take for `malloc` or `free` to carry out your request. Is it usually pretty fast? Yes, but this is worst-case analysis, and because `malloc` promises nothing and may have to scour all of memory to find something, it could take arbitrarily long. Oh dear. Remember how we talked about the binary buddy routine?

Empirical Observation. The other approach is to measure instead of estimate. If you already have a version n of the system and want to build version $n + 1$ it might be easy to look at existing data and extrapolate, but if you want to build a new system then you'll need to do a simulation. How to effectively do a simulation could be a whole course on its own, but suffice it to say that if your simulation makes incorrect assumptions or is otherwise constructed poorly, the data you get will be useless or misleading.

If you do prepare a simulation and run it you'll presumably get some distribution of task runtimes. I don't want to get into the statistical math of it, but based on this, it's possible to estimate using known mathematical techniques the worst case runtime with the *confidence interval*. When we talk about confidence interval, we might say that the maximum time is t with 99% confidence, based on the standard deviation. You might want to push that to 99.99% confidence if required. That will give you the worst-case estimate that you're willing to use.

Hurry Up and Do It

Now that we have some understanding of real-time systems, how to classify tasks in them, and even how to estimate how long a task takes, we can finally get to actually scheduling some!

[12] `https://nasa.github.io/fprime/UsersGuide/dev/code-style.html`

16 — Real Time Scheduling Algorithms

Real-Time Scheduling Algorithms

Given our understanding of scheduling algorithms in general and an undemanding of what makes real-time system scheduling a little bit different from just regular scheduling, we can consider some other scheduling algorithms that work for real-time systems. We already covered the idea of timeline scheduling. If everything is predictable and orderly then making a schedule that looks like a schedule of classes is effective. But we also know that's not the case, so we'll talk about some other scheduling algorithms that are able to handle aperiodic and sporadic tasks. For now we'll go back to the uniprocessor view of the world, because it's simpler – but eventually we must indeed talk about the multiprocessor scenario again.

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. There's a proof of this in [HZMG20] if you'd like to see it.

Part of what makes this work is preemption, because a task could arise that is more urgent (i.e., has an earlier deadline) than the currently-executing one. From the point of view of the operating system, on completion of the system call to handle the request to schedule the new task, suspend the previously-executing task and start running the new one. This might mean a periodic task being preempted by an aperiodic or sporadic task [HZMG20].

To implement this, a priority queue is reasonable. A simple view says that the priority is determined by the deadline: keep it sorted in ascending order of the time of deadline. However, this doesn't account for the possibility that soft-real time tasks may have a sooner deadline than a firm- or hard-real time task. If the system is not overloaded then there is no issue, and ideally good system design means that overload isn't an issue. But if it is, then things get a little more interesting.

Do we skip the soft-real time task? Do we start it but cancel it if the situation gets dire for a more important task? Let's come back to that after discussing a couple of other algorithms.

Deadline Interchange. This deadline-based approach is subject to a problem that very much resembles priority inversion. Suppose a task A has locked a mutex and then a new task B arrives that also needs that mutex and has a sooner deadline than A. Task A would therefore be preempted in favour of B, at least until B gets blocked. If there are other tasks C, D, E... that also want this resource, A could be waiting a long time to proceed. In fact, it could be waiting so long that task B could miss its deadline!

To solve this, A needs to finish the critical section and release the mutex. The best way to make that happen would be to assign to A a new deadline, specifically the soonest deadline from all the tasks waiting for it. That looks a

lot like priority inheritance, doesn't it? It's a very similar problem, so it is not surprising that the solution looks similar.

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.

Much like the earliest deadline first approach, a queue where priority is determined by the slack makes for a reasonable implementation. Some work is needed periodically to recalculate the slack for each task, though.

Rate-Monotonic Scheduling

Unlike the previous two scheduling algorithms, the name doesn't explain as much about how this one works. We consider the rate monotonic algorithm because something like the earliest deadline first or least slack first approach is that they focus solely on deadlines but do not consider priorities otherwise.

Rate-Monotonic scheduling is based around the idea of basing priority on the period – tasks that execute more often are given higher priority. Tasks with higher frequency require more frequent attention, and therefore should be given higher priority. So each task needs to be assigned a priority number based on that period and this priority does not change dynamically at runtime. Higher-priority tasks will preempt lower-priority ones as needed.

This algorithm is, however, not perfect in the sense that it's possible to fail to schedule things in such a way that all tasks meet their deadlines even if utilization is less than 1. An example from [HZMG20]: Suppose we want to dynamically schedule n periodic tasks with the form (C_k, τ_k): (1, 4), (3, 7), and (3, 10). If we try to schedule these, we'll find that the third task fails to meet its deadline: task 1 runs, then 2, then 1 again, then 3. So far so good. Then task 2 runs, but it gets interrupted by task 1, so task 1 runs to completion, then 2 resumes, and 3 misses its deadline. But utilization is less than 1 so it should be possible to solve this one: $(1/4 + 3/7 + 3/10 = 137/140)$. Earliest Deadline First would have been able to schedule this such that it met all deadlines. Oops.

To figure out if it's possible to schedule things, there's a test. It turns out that it's very difficult to do this dynamically in real-time, so we have to use a less-precise formula for calculating whether a group of tasks may be scheduled or not. Given that real-time systems are pessimistic and would prefer to say no when it is schedulable rather than say yes when it is not. There are some formulae and proofs linked in the source [HZMG20] but it is easy enough for the computer to calculate: $\Pi_{k=1}^{n}(1 + \frac{c_k}{\tau_k}) \leq 2$.

Remember, though, if it's not possible to guarantee that all deadlines can be fulfilled, it doesn't mean that the tasks are certain to fail to meet their deadlines. Task times are always worst-case, and so things might actually be fine in reality.

Deadline-Monotonic. A variant of this is deadline-monotonic scheduling, in which case priority is assigned based on deadlines. The task with the shortest deadline is assigned the highest priority. It's not that different or interesting, but it is possible to come up with some scenarios that Rate-Monotonic would not find a good solution and Deadline-Monotonic would.

Aperiodic Servers

Let's return to the idea of aperiodic tasks in the earliest-deadline-first approach. A task with a soft deadline is challenging to schedule here, because it's hard for the scheduler to know whether a task is soft- or hard-real time. One possibility is to say that aperiodic or soft-deadline tasks are always lower in priority than any firm- or hard-real time task, but that may not be optimal.

We'll examine an approach called a polling server as explained in the original paper introducing the idea. A polling server is, in its own way, a little container in which aperiodic tasks occur. The server is itself a hard-deadline periodic task with a fixed execution time budget and a deadline equal to its period [GB95]. Every time this task runs, it's really a trojan horse for the aperiodic tasks that want to run. During the execution time of the server task, the aperiodic tasks waiting will run sequentially. If there are too many tasks or they otherwise take too long and they do not finish, the aperiodic tasks just carry over to the next time the server runs. If there are not enough aperiodic tasks and there's time left over, just end the server task execution (throw away the extra time) and let something else run.

An analogy that makes some sense here is a lunch break at work. You don't have to use this time period solely for eating, but you can use it to do various other tasks if you want or need: go to the bank, go shopping, etc. The lunch break task is "important" in the sense that you cannot skip it or reschedule it indefinitely. But when it is lunch break time, it's your time to do with as you wish to do tasks that otherwise might be difficult to schedule.

It's also possible to have multiple servers to handle different types of aperiodic tasks. Higher priority tasks could go into one server and lower priority tasks in another one. In the lunch break analogy, you might try to go to the supermarket at lunch time (larger, higher priority task) and do some smaller task (Duolingo?) in a coffee break.

This approach does not affect the ability of the system to complete the hard-deadline tasks, because the background server task runs following a schedule that's sufficiently known and predictable that the other hard-deadline tasks can also be scheduled with certainty that they will complete. Excellent!

The disadvantage of the polling server is around the response time for the aperiodic tasks; it's half the server period plus the average execution time and the only way to improve that is to make the server period shorter [GB95]. That might not be desirable, though, because it increases the overhead significantly.

Variant: Deadline-Deferrable Server. An improved version of this that gives better performance for aperiodic tasks and it's called the *deferrable server*: it's like the polling server, but instead of throwing away the time budget if it's unused at the end of the period and there's nothing to do, save that leftover time in case something arrives [GB95]. Eventually the period expires and at that point the remaining execution time is lost, or the time is exhausted by actually doing tasks.

To be clear, in the polling server situation, if the polling server task runs out of things to do with 40% of its time budget available, the polling server task ends and the remaining time is thrown away. In the deferrable approach, that budget is retained until the end of the period for the deferrable server, so if something comes in during that remaining window, it can start immediately. The advantage here is effectively that an aperiodic task will likely be served faster than they would have been if they arrive a bit later.

Continuing the lunch analogy let's say you have an hour and you use it for lunch and finish eating in 45 minutes. In the polling server approach you go back to work immediately even if your lunch break time isn't used up and your remaining 15 minutes of break time is discarded. In the deferrable approach, your extra 15 minutes of lunch time isn't used up and you could use those 15 minutes later on in the day to take a personal call. Still, when the day is over, if you didn't use it for something else, the remaining 15 minutes is lost. The next day you get another full hour for lunch, but the extra time from the day before didn't carry over. Note that none of this is a substitute for legal advice from an employment lawyer about the rules and regulations around legally required breaks during the workday.

Variant 2: Deadline Sporadic Server. The next idea for a high-priority task that handles the aperiodic is called the sporadic server, and it's like deadline-deferrable, but for two things: (1) instead of losing the extra unused period it can be saved for the future, indefinitely; and (2) the replenishment of the budget forces execution to be spread out more evenly [GB95].

Why does spreading it out matter? The deadline server could theoretically schedule two runs of the server back-to-back, which is convenient for some aperiodic tasks (the ones that are there and ready), but inconvenient for others. When I was doing my first co-op term in Toronto, I would take the Finch West bus to and from work. The schedule said "frequent service" which was TTC-speak for "every ten minutes, or more often". It was subject to a phenomenon that's called "bunching" in the literature, by which I mean you could stand at the bus stop for 30 minutes and then three busses would arrive and no more for the next 30. This is convenient if you happen to be standing at the bus stop at the time when the three busses arrive because you can get aboard the least-busy one

and ideally have a more pleasant ride. But if you just missed it, it's going to be a long wait.

As for spreading it out, the basic plan is that the execution time for the server is broken down into chunks. When there's something to do and the server runs that aperiodic task, the amount of time that it runs is deducted from the available budget of chunks. The decrease can be fractions of a chunk, so, for example, if there are 10 chunks total, the budget is full, and an aperiodic task runs for 1.5, the budget is now at 8.5 chunks. When a chunk is depleted, it's scheduled for replenishment at a fixed time in the future. That spreads out the replenishment. If chunks are exhausted, the server is suspended until the next period.

This sort of replenishment model is also used in other contexts, like, say, database credits for Amazon AWS database instances. You start out with some number of credits, they are depleted by using the database CPU time above a certain threshold, and replenished at a steady rate over time. There is a cap so you can't stockpile too many credits.

I would explain this as being a little bit like the hearts system in Duolingo. If you're not familiar with it, it's a language learning app with a green owl mascot. You start with five hearts, which is also the maximum number of hearts. If you make a mistake when doing an exercise, you lose a heart. If you run out of hearts, you cannot continue and have to wait for hearts to refill and hearts refill at a rate of about one every four hours. You can pay money to avoid this but that's a different subject. The key difference here is that in Duolingo the hearts only decrease if you get an exercise wrong; in the deadline sporadic server approach the budget always decreases from running aperiodic tasks.

Variant 3: Deadline Exchange Server. This is an approach builds on the Deadline-Deferrable server, and simplifies the idea of tracking the chunks. Chunks require individual tracking so there's a lot of overhead of keeping a handle on them. The strategy is that any remaining execution time is discarded when the request queue is empty and the wait for replenishment is based on the last chunk of execution time used [GB95]. This is simpler to implement since there's only ever one replenishment to track and the replenishment is always the full quota.

But does it work? The simulation studies show that different forms of the period server approach are effective for different workloads. The paper introducing this suggests that you get good average response time by making the period of the server be the same as the shortest period of the regular period tasks [GB95]. But it's possible to have different servers configured for different aperiodic tasks to give the best balance of prioritization and performance.

Multiprocessor Math

So far when we've talked about real-time scheduling, we have not considered the possible complexities introduced by making it multiprocessor. In this section we talk about tasks as the things that need doing, an a job as a specific instance of the task to run.

If there are multiple processors in the system, we need to make a decision about whether tasks have dedicated CPUs or whether they can move between CPUs as needed. There are three interesting decisions that are relevant [MA05]: whether preemption is permitted, whether job migration is permitted, and whether job parallelism is forbidden. The last one is just that a job may execute on at most one processor at any given point in time.

If tasks have dedicated CPUs that means the second assumption is answered with a definite no. In a real-time situation it's possible to just do manual assignment that says Tasks A and B are on CPU 0, Task B and C are on CPU 1... all the way to saying that aperiodic tasks are assigned CPU 4 and that's it. Manual assignment makes the multiprocessor situation almost indistinguishable from a uniprocessor situation. It's possible to just look at each CPU individually and the tasks that are scheduled to run on it and verify that given the requirements of the system, it's possible to meet all hard deadlines.

If you actually want the computer to schedule the tasks by assigning tasks to processors, what we actually have is a variant of the bin-packing problem which is known to be NP-complete [MA05]: that is, to find the globally optimal solution, it's necessary to try *all* possibilities, but it's possible to get an approximation in polynomial time. Doing so in advance for an all-periodic system is doable, but for systems with sporadic and aperiodic tasks, the cost of doing this math might be prohibitive at runtime.

No migration does mean that there is a possibility that tasks are failing to meet their deadlines on CPU X for lack of CPU time even when CPUs Y and Z have plenty of idle capacity. That feels frustrating. Things are more

interesting if there is migration allowed. Migration can alleviate the problem of computing resources being unused but increases the complexity and makes it much harder to predict whether we can guarantee all deadlines will be met.

A simple analysis might say that there's no cost to migrating a task from one CPU to another. We know that this is not true, based on our understanding of the hardware: when a task moves CPU, that new CPU may have none of the relevant pages in its cache, resulting in more cache misses and slower execution – not only of that task, but maybe other tasks as well if this increases contention for the bus. So, do we allow migration or not?

Let's step back a bit and consider our options as enumerated in [GRL+12]. The first is global scheduling where all tasks go in one queue and the scheduler assigns tasks to available CPUs. There is potentially some overhead of managing this single queue, and migrations are allowed. The second option is referred to as *partitioned* scheduling and that's what happens when tasks are statically assigned to a CPU and each CPU manages its own queue. Some research shows here that this approach may only allow 50% utilization of the system. The reasoning is that The paper introduces third option is called semi-partitioned scheduling, where most tasks are fixed to specific processors but others are allowed to move. It's probably somewhat intuitive that this mixed approach works well because it allows for some flexibility, but not too much.

None of these approaches are actually solutions that ensure we have an optimal scheduling for multicore system. Remember that optimal in this sense means that if it's possible to schedule it such that all tasks meet their deadlines, the system finds a way. What we need is a different approach...

P-Fairness. The P in P-Fair scheduling stands for "proportional" and the goal is to allocate CPU time in such a way that tasks make progress at steady rates, or in a less-nice way of phrasing it, tasks are forced to proceed at proportionate rates. More formally, an application can request time x_i time units every y_i time quanta (time slices) and guarantees that over any T quanta (greater than zero) then a continuously-running application receives at least $\lfloor x_i/y_i \times T \rfloor$ time and $\lceil x_i/y_i \times T \rceil$ quanta of service [CAS01]. In other words, time is divided up into little pieces and each process gets a proportional share of the CPU time and is never more than one timeslice away from the amount of time that it "should" receive.

The scheduling algorithm as defined in [BCPV96] is simple enough to explain once appropriate terms are defined. The term *lag* is used to reflect the difference between the actual allocated time and the time a task should have. If lag is greater than zero, the task is behind on the CPU time it should have had; if lag is below zero, the task has had more CPU time than it should normally have. A task is considered *urgent* if lag is greater than 0 and if the lag would exceed 1 full quantum if that task doesn't run in the next quantum. A task is *tnegru*[13] its lag is negative and would remain negative even if the task does not run during the next quantum. If a task is neither urgent nor tnegru, we'll call it "other". Then the scheduling algorithm consists of three parts:

1. Schedule all urgent tasks.
2. Do not schedule tnegru tasks.
3. Schedule other tasks in order of highest lag to lowest until capacity is filled.

In short, to make sure that the lag for any task is always between -1 and $+1$, the scheduler prioritizes the tasks with the highest lag to make sure they do not exceed 1. It also deliberately ignores tasks with negative lag so that their lag increases and brings them back closer to the zero point where it's trying to keep every task. If it's possible to find an appropriate schedule where all tasks meet their deadlines, the algorithm not only will find it, but execution is fair in the sense that all tasks make progress at around the same rate.

Scheduling in a P-Fair approach is much more tractable because each task is broken down into small slices and each subtask executes on the CPU when it can (and must be before its deadline). This avoids the bin-packing problem because subtasks just go where they can go and there's limited wasted space. In fact, the original paper [BCPV96] shows that a P-fair schedule is always periodic, which means it's quite suitable for real-time applications.

There exist a number of papers building on the P-Fair approach to address some of its shortcomings and adapt it to some more scenarios. One possible disadvantage of the system is that it requires a lot of computation whenever

[13] I hate this word, but this is "urgent" backwards. I wish it were something else but that's what's in the literature and we're stuck with it now.

one of the timeslices ends – updating the lag of every process and reclassifying which processes fall into which category.

But is it worth it? To wrap it up, is P-Fairness a worthwhile algorithm for real-time scheduling? I'll argue yes for a multicore system. Other approaches have worst-case behaviour meaning that utilization is limited (in some cases to as little as 50%) to ensure that even in the very worst-case scenario that all deadlines will be met. Whatever the overhead of the calculations in P-Fairness, surely it's not 50% of the CPU capacity...

17 — Scheduling Algorithm Evaluation; UNIX & Windows

Evaluating Scheduling Algorithms

Although in some cases, like real-time scheduling, it is pretty obvious what algorithms we might want to use, in many cases, it isn't immediately obvious what scheduling routine makes the most sense or gives the best overall performance. This necessarily implies that there is a clear definition of best overall performance. We talked about some ten earlier and acknowledged they're not all equally important in a given system. Some are ignored altogether. That's okay.

Let's imagine, though, that you've made a decision of what's important in your system. However those are decided, we need to evaluate the algorithms we've discussed in terms of how well they achieve that goal. While it is possible to just have a theoretical discussion about it where we say we think A is better than B but worse than C, that's not a very scientific approach. What we'd like to do instead is gather some data and make the decision using it. Given that we don't have specific criteria for a system, this discussion will be restricted more to the idea of how to decide rather than walking through a specific decision.

Deterministic Modelling. Deterministic modelling looks a lot like writing test scenarios for a lab project. The test scenario has some inputs and some outputs and then we can evaluate the outputs to check for both correctness and to assess the performance. Why correctness? It's usually assumed that whatever algorithm that we are talking about is correct in its implementation, but evaluation of correctness in this context may mean things like processes do not starve.

Simple test cases will create all processes to run at the beginning and then put them into the ready queue. An example of a simple test case looks like this, where all five processes arrive at time 0 in this order [SGG13]:

Process ID	Burst Time
1	10
2	29
3	3
4	7
5	12

We can then run the target scheduling algorithm(s) and get some answers; the source shows that First-Come-First-Served results in an average wait time of 28 time units; Shortest-Job-First 13; Round-Robin with a timeslice of 10 time units has a waiting time of 23. In all cases, the total time of execution is the same – there's no avoiding the work to be done – but waiting times are different. In this example, the waiting time is the measure that we care about.

A more precise version of this evaluation would assign a penalty to the process switch. FCFS and SJF each have four process switches since processes do not get interrupted in the run. Round robin will have seven context switches, so it would be appropriate to say that the total execution time for the round robin evaluation is higher, although if context switches are very small as compared to the execution time, the tiny difference may be irrelevant.

Test cases that look more realistic will not have everything be statically initialized at the beginning and all tasks waiting. They will include tasks that arrive at the start, but also tasks that arrive after some time has passed, and may include periods where the idle task runs. They may also include other things, such as changing the priority of one or more threads as this will have an impact on the execution order of the various processes.

The nice thing about the deterministic modelling approach is that the tests are reproducible and produce consistent results on every run. Therefore, swapping or tweaking the algorithm under test and re-running the tests will make it pretty easy to identify the impact of the change (if any) that was made.

Queueing Theory. Deterministic approaches are somewhat limited because the real behaviour of most systems looks very different from how the deterministic approach models it. Rather than a nice model of the world where things always happen in a fixed way, real systems have a lot of randomness in them: the time when a user requests something is, from the point of view of the computer, not really predictable.

Yes, it may be the case that most people log in to a server at the start of their workday, but everyone's start of the workday is a little bit different, even if everyone is supposed to start at 09:00. Some people will be there a bit earlier, someone might be late because there's a crash on the highway, etc, or because a colleague stopped by their desk before they entered their username and password... A model with some randomness built in might actually better reflect what happens... But not total randomness, because perhaps there are indeed some patterns like log-ins take place at the start of the day but not in the middle of the night.

That leads us to queueing theory! It is, as the saying goes, literally the theory of queues. What makes queues appear, how will they behave, and how do we make them go away? It does actually have the kind of modelling that we're looking for where we can treat arrivals of new things to do as random, and also have some sort of mathematical distribution for how long tasks execute. Given these models of what are called arrivals and departures we could then do some calculations around things like how busy the system is and how long processes have to wait.

The basic problem with queueing theory analysis is that it is mathematically complex and not suitable for every kind of scheduling algorithm; it also tends to force modelling things like request arrival times in ways that are easier to implement but not necessarily accurate [SGG13].

Okay, queuing theory is too advanced for this course and we'll have to save it for a future one. It typically shows up as a graduate-level topic, but somehow, hints of it find their way into fourth year courses here and there.

Simulations and Emulation. Whatever modelling approach we choose, deterministic or statistical, we would get more accurate evaluations of the system by running a simulation. The simulation, if designed appropriately, will account for more things like clock interrupts and context switch times.

It's possible to do an all-software simulation, in which you just write some code and it runs as a regular process inside an existing operating system. The simulation has time steps and at each time step, the state of the system is updated as if certain things happened: the clock advances, timer interrupt runs, context switch takes place, etc. The simulation is only as accurate as you make it and there are a lot of things that might be difficult to simulate accurately (e.g., how often is there a cache miss?). It's difficult to make a simulation, but of course, it does not have to be perfect, just better than the alternatives.

Deciding what test scenarios you'd like to simulate is similar to the question about what test cases you want to execute in deterministic modelling. One possibility is the idea of traces [SGG13]: the basic idea is to record some actual executions on the system and use that as the basis for the test scenarios you'd like to run.

If you're writing code for another platform (e.g., an embedded system), that platform may come with an emulator which would be an even better place to evaluate the scheduling algorithm. The emulator is much slower than just running the simulation as a process inside another operating system, but it does give a much better understanding of how the system will actually perform since the emulator will have proper implementations of things like hardware and timer interrupts. And perhaps most importantly: you don't have to write the emulator; just use the existing one!

And then the real world... Simulations can only tell us so much about the actual performance of a scheduling algorithm, just because the simulation is always based on certain assumptions around user behaviour, external factors (network, etc), and the hardware that the user has. A general purpose operating system needs to work at

an acceptable level on many different hardware configurations.

And on that subject, well, we should take some time to consider the scheduling algorithms actually used in commercial desktop and server operating systems. Specifically, the traditional UNIX, Windows, and modern Linux scheduling approaches. I'm by no means saying that any of these operating systems have everything right and their form of scheduling is the optimal or ideal for the target use case, but if their choices were awful then user/-customer pressure would surely force a change...

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. Truth be told, the traditional UNIX approach is dated, but it's worth talking about because a lot of the evolution of scheduling in Linux is based on overcoming the shortcomings of the traditional scheduler approach.

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 [Sta18].

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.

Upgrading to SRV4. In UNIX SVR4 the system for a complete overhaul, trying to give the highest priority to "real-time" processes, then kernel processes, and then user-mode preferences; its two big differences are (1) more priority levels – 160 broken down into three types – and (2) preemption points [Sta18].

Such preemption points are needed, because the original versions of the UNIX kernel were themselves not well-suited to preemption because it was not expected in the design that the kernel's execution could itself be preempted at any time. So preemption points are placed in the code where it would be okay for the kernel to stop execution and do another operation. In practice, this means that all kernel data structures are updated, or there are appropriate locks that would prevent concurrent modification and prevent race conditions [Sta18].

FreeBSD. BSD, the Berkeley Software Distribution, is another form of UNIX, and FreeBSD is one of its descendants. FreedBSD has a scheduling process that's quite similar to that of SRV4. There are more priority levels than SRV4 – 256 rather than 160 – and they are subdivided into five categories rather than just three. This implementation is intended to support multiprocessor systems better than the traditional UNIX approaches.

What's new and interesting here is the interactivity scoring mechanism. Interactivity score is intended to identify which threads are user-interactive and which ones are CPU-intensive. Threads that are user-interactive should get a higher priority to run; in this system, lower numbers equal higher priority. Interactivity is judged based on how much the thread in question gets blocked: if it's waiting for the user or network or similar, then it would be considered interactive.

The calculation looks like this [Sta18]: We'll define the maximum interactivity score as m, the runtime of the thread as r and the sleep time of the thread as s. For threads that have more sleep time than run time, the interactivity score is calculated as $(\frac{m}{2})(\frac{r}{s})$ and for those that have more run time than sleep time: $(\frac{m}{2})(1+\frac{r}{s})$.

This system ensures that threads that sleep more than they run are always in the lower half, and threads that run more than they sleep are in the upper half of their priority bands. The mechanism is a little bit crude – interactivity is a bit more complex than just whether the threads get blocked. One might expect that the solution is something like whether the program interacts with the console or UI fields, but users are good at gaming the system like this and doing unnecessary writes to the console to make the program appear more interactive than it is. So a crude method provides less reward for gaming the system, so to speak.

We've already discussed in some detail how thread affinity works and load-balancing across the different CPUs in the system. The FreeBSD scheduler uses both pull and push mechanisms to move threads to cores that are otherwise idle. The pull mechanism is simple enough: when a CPU has no work to do, it sets a bit in a bit mask to indicate that it's idle, and if a new thread is about to be added to a queue, the assignment mechanism checks for processors that are idle and sends it to them. The push migration is accomplished by the short-term scheduler twice per second and looks at the highest- and lowest-loaded processors and equalizes their run queues [Sta18].

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].

Windows also includes a little mechanism to make sure that a process does not starve; processes that have a low priority can temporarily get a boost to a priority of 15 to make sure that they get a chance to run and also to mitigate the impact of a potential priority inversion scenario [Sta18].

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.

18 — 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 [Sta18]:

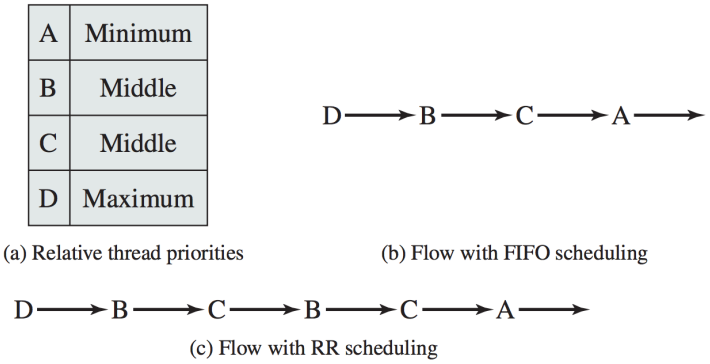
- `SCHED_FIFO`: First-In, First-Out Real-Time threads
- `SCHED_RR`: Round-Robin Real-Time threads
- `SCHED_OTHER`: Other (non-real-time) threads.

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 [Sta18]:

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) [Sta18].

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 [Sta18]:

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.

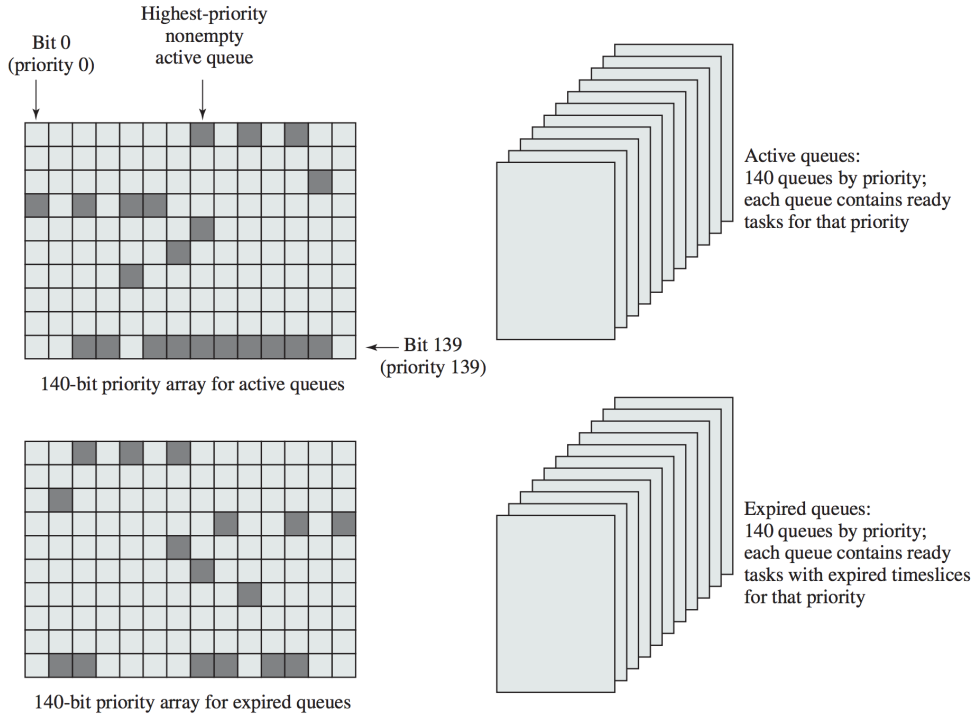
Finally, pre-emption was not possible; lower priority tasks would continue to execute while higher priority tasks were waiting. Only something getting blocked, a time slice expiration, or an interrupt might cause the scheduler to re-evaluate what process should run next.

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 [Sta18]:

```
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:

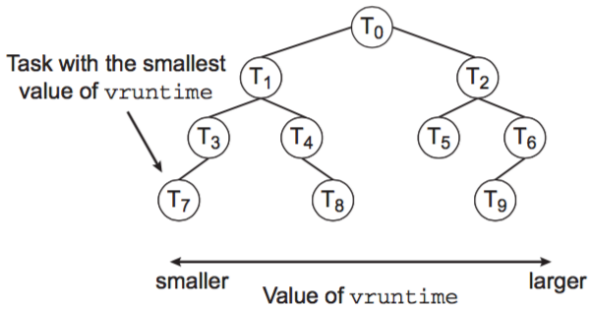


Linux $O(1)$ scheduler internal structures [Sta18].

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 [Sta18].

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) [HZMG20].



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.

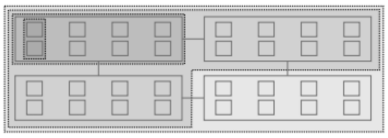
There are four different problems but they all cause the same behaviour: cores are left idle for a long time when runnable threads are waiting to execute. If it is brief, it is not a problem; but if it goes on for longer then it will be more of an issue. Suppose there are 4 CPUs, each of which is busy, 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.

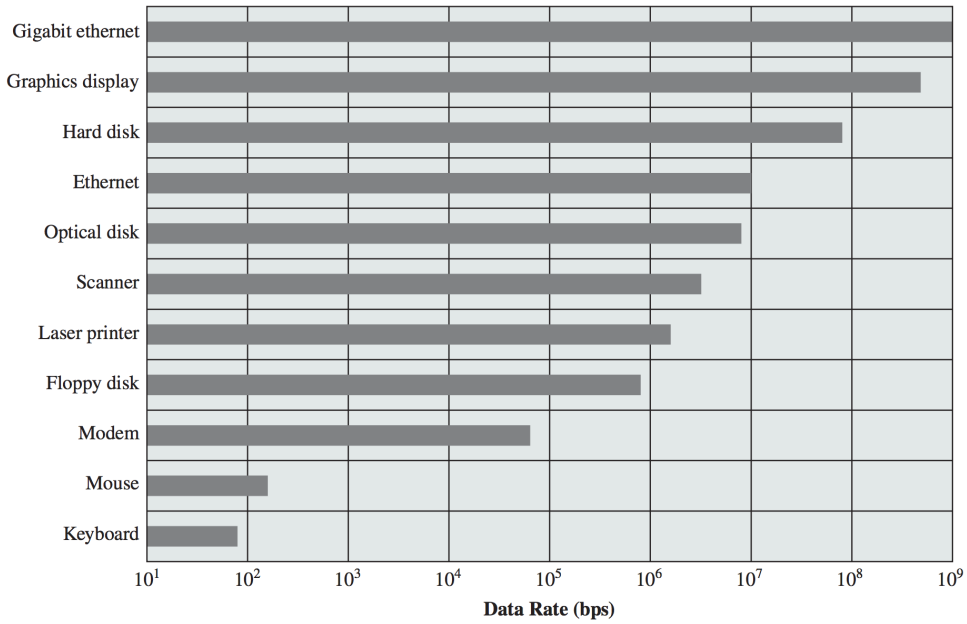
Missing Scheduling Domains The last bug appears to have been caused by an error during refactoring. When a core was removed and re-added a step was skipped after the refactoring changes which could cause all threads of an application to run on a single core instead of all of them.

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.

19 — Input/Output Devices & Drivers

Input/Output Devices

Though the computer's name and typical use kind of suggests the purpose of the machine is computation, input/output is just as important to the usefulness of a computer. A computer that takes no inputs and produces no outputs is, presumably, not very useful[14]. I/O is, unfortunately, a messy business. Whereas there are only a small number of CPU types your typical desktop OS might run (and the AMD and Intel processors, for example, can execute the same binary code), there are uncountably many I/O devices in the world. And they're all different. Keyboards, hard drives, printers, and headsets are all I/O devices, but they serve very different purposes and work in different ways. This diagram shows the dramatic differences in speed between various devices of a typical PC:



Typical I/O Device Data Rates [Sta18].

We might like to think that USB has taken away some of the complexity, but that's just the way a device connects to the computer; managing the device itself is still as complicated as ever. All of the examples just listed in the previous paragraph can be connected via USB (Universal Serial Bus). And indeed, just because the computer and device have compatible physical connectors does not mean that the computer knows how to interact with the device. This can be kind of frustrating.

Disk I/O, when discussing magnetic hard disks, has a huge impact on system performance that it will receive its own discussion. But first, a little discussion about I/O in general.

[14] Schrödinger's computer: did the program actually run if there was no output to observe?

Communication with Devices

There are three ways, realistically, that we can communicate with a hardware device: polling, interrupts, and DMA (direct memory access). This isn't a hardware course, but I think we should review this topic at least a little bit before we go on. It's fair to say that these options are not all equally good; polling would be something we do because we must, not because it's efficient.

An I/O port represents the connection, and may typically have four registers: the data-in register (input), data-out register (output), status register, and control register [SGG13]. The control register is how the user or operating system gives commands to the device, and the status register tells us the current state of the device. Those aren't too exciting.

The literature tends to refer to the main system as the "host", because the code that's interacting with the device could be the OS or it could be user-level code depending on the situation. So we'll use the word host.

Polling. The most basic way to interact is polling, and as we've just discussed, it is the last choice but there might be the only choice. When a command is issued, the controller uses a bit to mark the device as "busy" during the time that the operation is taking place. When the operation is done, the bit is cleared. After that the OS can take the next action, whether that's reading the data requested or scheduling the next write or moving on from the operation to the next thing.

This isn't super exciting, but it does work. The host has to check the state of the device repeatedly until the busy bit is cleared. It's not really efficient, because the CPU executes lots of instructions that are not doing useful work while it repeatedly reads the value of the busy bit.

A first thought would be to change the logic a bit such that instead of tight polling, the host checks periodically rather than immediately. That does decrease the wasted CPU cycles, but has a risk that the data is lost because the host does not respond in time. If the data storage for the device itself is very limited, if the data currently ready is not retrieved fast enough, then the data might be overwritten by new data, causing some input to be lost [SGG13].

Interrupts. Interrupts are something we've spent a ton of time talking about, but remember that this will operate on the basis of there being some sort of interrupt request line (hardware) and when the CPU sees the signal that indicates the presence of an interrupt, the interrupt handler is executed to deal with it [SGG13]. The interrupt handler will do whatever it's supposed to do and then clear the interrupt signal.

This description is fully correct, but overlooks some of the complexity around interrupts. For one thing, not all interrupts are equally important – incoming phone call is not as urgent as a fire alarm. The other part is that interrupts should be something the CPU can (mostly) turn off it that makes sense.

The interrupt approach helps to deal with the problem of not responding in time to the availability of data: when the interrupt is triggered, take the data from the device and put it in memory somewhere. This does not mean that the data has to be processed immediately, and it might actually be preferable to delay handling it until enough data has arrived. But as long as the data is in memory it won't get overwritten unexpectedly.

Direct Memory Access. Direct Memory Access or DMA is based on the idea that it's preferable to hand off some work from the CPU to a delegate, the DMA controller. The idea is that the DMA controller is given instructions about what to do: the operation to perform (either a read or write), the source, the destination, and how much data to be transferred. This can be efficient – if copying data from, say, a USB drive to the hard drive, the data doesn't have to go to the CPU in between.

When the I/O operation is completed, the DMA controller will notify the CPU with an interrupt; but one thing that may not be obvious from the description is that the DMA controller might slow down the CPU a little bit because it's using the bus and that can make the CPU wait [SGG13]. That's still probably less than if it was an interrupt-based system.

Application I/O Interface

The use of DMA and interrupts are intended to increase the efficiency of the system, because we know that most devices are very slow when compared to the speed of the CPU and even memory [Sta18]. A lot of the efficiency gains, though, are based on what hardware is available so not necessarily something we can control when developing an operating system. Yes, the hardware is useless if the operating system does not use it, but the OS cannot compensate for the lack of a DMA module.

Ideally, when a general-purpose operating system is written, it will accept new devices being added to the system without editing/reinstalling/recompiling the code. Your experience with object oriented programming gives you some familiarity with the solution of how we should accomplish this goal. We want to abstract away the details of the hardware, to the extent we can, and provide a uniform *interface* to interact with.

In the very early days of operating systems, the hardware the computer shipped with was all the hardware it ever supported. If the vendor came out with a new module, they would have a new operating system update to introduce support for that device. This got to be unmanageable in the era of the IBM PC because anybody could create hardware and attach it via a standard interface. Relying on IBM or Microsoft or whoever your OS vendor was to implement support for a random piece of hardware was not realistic.

And from the operating system developer point of view, it's hard to argue. How are they to know what hardware devices exist, and even if they did, how they work? Some sort of standard interface only gets you so far. You can have one printer masquerade as another – many years ago my parents had a Panasonic printer that you could tell the operating system was an Epson printer. That sort of works, but what about a new type of device or a device that is similar to an existing one but has some additional features or functionality?

Operating system developers thought they were very clever. They realized that they could shift the work to the hardware developers through a concept called *device drivers*. The device driver plugs in to the operating system through a standard interface and tells the operating system a bit about the hardware and translates commands from the operating system to hardware instructions. Where this fell apart was the fact that hardware developers often made extremely poor drivers (whether this was due to inability or lack of caring is not clear).

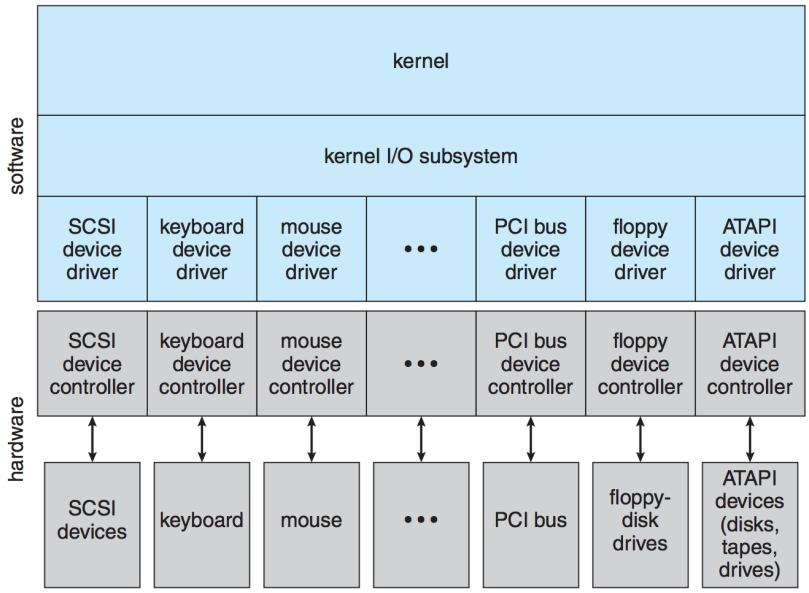
The problem was exacerbated by a Windows design decision that device drivers would run in the system at the same protection level as the kernel. Some other operating systems have user-mode drivers, where possible (in Mac OS X and Linux systems, for example, there is the ability to have file system drivers in user space through a program called FUSE) or at an intermediate level between that of user space and the kernel. In any event, in Windows, if a device driver is unhappy with the current state of the system, contains a programming error, or encounters a situation it cannot handle, the driver can invoke a system call that brings up everyone's favourite feature: the Blue Screen of Death (BSOD).

Microsoft, rightly or wrongly, was blamed for a lot of those blue screens of death. If it appears after upgrading or patching the operating system, users blame the most recent thing changed, even if that's not what caused it. It also didn't help that the message said something about Windows has encountered a problem. That sounds like Windows is the one with the problem.

Microsoft took two approaches to remedy this problem. One was to write and include in Windows a lot more device drivers. Including the drivers isn't too bad, but writing them yourself is painful. But user experience is key, and some effort invested into this probably does pay off. "It's not my job" is nice to say but sometimes getting the customer to sign the deal is more important than feeling right on principle.

The other approach Microsoft took was to introduce the static driver verifier, a complex piece of static analysis software used to test, at compile/build time, whether the driver will behave badly. Passing this test is required to get a sticker of approval from Microsoft to put on the box. So, the battle rages on about who is responsible for writing the drivers, unfortunately.

Device drivers, when they exist, connect into the kernel's I/O subsystem to mediate between the kernel's I/O subsystem and the hardware device controller. See the diagram:



Kernel I/O structure, showing the placement of device drivers [SGG13].

Abstracting away details of the hardware makes the job of the operating system developer easier: the interface provides a standard way of interacting with the hardware device. In Windows and Linux, for example, there exists another layer to add some more indirection between the OS and platform-specific hardware. It is called, obviously, the hardware abstraction layer. The HAL is supposed to make it easier to port the operating system to new hardware.

Unfortunately, every piece of hardware could be completely different. Even if the goal is just classifying devices based on similarity, devices can vary on numerous dimensions. Here's a list of a few examples from [SGG13]:

- **Data transfer mode**. A device may operate on a character level (transfer one character at a time) like a keyboard, or may operate on a block (group of bytes) all at once, such as a disk operating on a 4 KB block.
- **Access method**. A device may allow only sequential access (that is, reading only the next element or piece of data or writing to the next location) such as a modem, or it could allow random access (reading from or writing to any point) as in a hard drive.
- **Transfer schedule**. The schedule may be synchronous (transferring data with predictable response times) as in the case of reading from tape, or asynchronous (with unpredictable response times), such as awaiting a response that will come over the network.
- **Dedication**. A device may be shareable (allows multiple concurrent threads) such as the speakers, or dedicated (allowing only one thread to use it at a time), such as a printer.
- **Device Speed**. Recall the diagram from earlier showing the dramatic differences in transfer speeds.
- **Transfer Direction**. Some devices are input only (such as a temperature sensor), other devices are output only (speakers), and some devices are capable of both (hard disk).

We would like to, as much as possible, keep the details above from the operating system, though devices will typically be grouped into a few categories so appropriate system calls can be issued. If a device is block-oriented, the OS should be issuing block read and write commands, not trying to do it one character at a time. Just for the sake of efficiency.

Operating systems also usually have an *escape* system call that allows passing of a command directly from an application or the kernel to a device driver. This allows us to issue commands to a device that the OS designers have not thought of and created system calls for. The UNIX system call for this is `ioctl` ("I/O Control") and it takes three parameters: a file descriptor indicating the hardware device (everything in UNIX is a file, remember), the command number (an integer), and a pointer to an arbitrary data structure (presumably `void*`) that has any necessary control information and data [SGG13].

Block and Character I/O

The block device interface is used for block devices such as hard disk drives. Any device will support `read` and `write` commands, and if it is a random-access device, it will have a `seek` command to jump to a specific block. An application usually accesses the hard disk through the file system, but going even one level lower, the OS can work on the hard drive using these two/three commands without being concerned with how that command is actually transmitted to the physical hardware.

We can abstract things a little bit further, from the perspective of the application developer, by having a memory-mapped file. Then, rather than using the block oriented operations directly, the application just writes to and reads from "memory" and the OS handles the behind-the-scene coordination to make writes go out to the correct block and read from the correct block.

A character-oriented device is something like the keyboard; the system calls are `get` and `put`. Libraries and other structures may exist to work on a whole line at a time (compare to, for example, the Java constructs for reading and writing files: the default streams are one character at a time, but if you want anything other than terrible performance, use a buffered reader which can read a whole line). This is a good match for input devices that produce data in small amounts and at unpredictable times, and perhaps for printers and sound output which operate naturally on a linear stream of bytes [SGG13].

Network Devices

Network devices are fundamentally different from those that are directly attached to the system. Thus, the `read`, `write`, and `seek` routines are not really appropriate. The model we are most familiar with from previous courses is the socket model. A socket is treated like a file, and at a basic level we can just ignore the details.

But we also learned about the `select` and `poll` functionality to support servers with multiple clients. That manages a set of sockets. Invoking this function returns information about what sockets have a packet waiting to be received and which are available for sending a packet. Proper use of `select/poll` eliminates polling and busy-waiting in a situation where delays are unpredictable (which is always the case with the network). Most likely, however, you remember that asynchronous I/O is difficult to manage.

Spooling and Reservations

A *spool* is a buffer for a device, like a printer, that can serve only one job at a time. Unlike disk, where it might read for P_1 now and then write for P_2 immediately afterwards, a printer needs to finish a whole print job before it starts the next. Printing is the most obvious example, but it is by no means the only one. The operating system centralizes all communication to the printer and runs it through the spooling system.

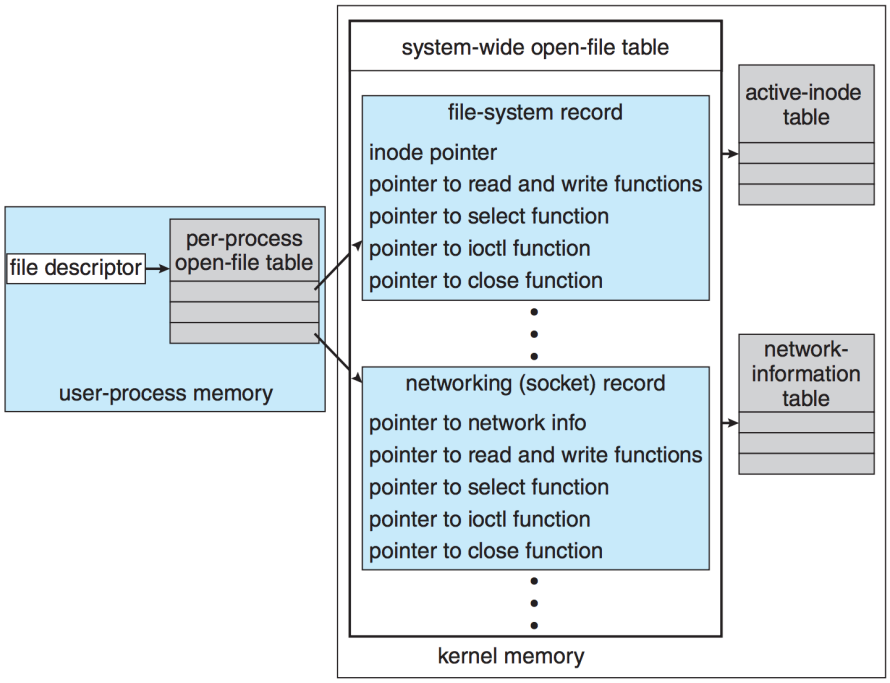
I/O Protection

Recall from much earlier the idea of kernel mode and user mode instructions; some processor instructions are restricted such that only the OS can issue them (in kernel mode), otherwise it is an error (illegal instruction). We want all user accesses to I/O to be mediated through the operating system, so the OS can check to see if the request is valid. If it is, allow it to proceed; otherwise terminate the process with an error. This helps minimize errors and problems where people do bad things like cancel another process's request so theirs can go first. Our typical tradeoff: increased safety in exchange for reduced performance.

In certain circumstances, for performance reasons, we might want to allow direct access. In the case of a game, for example, we want to allow the game to work directly on the graphics card's memory (despite the fact that it is an I/O device). Mediating every access through the kernel would result in unacceptably poor performance (which is to say, the other team would totally call us noobs). So a section of graphics memory, corresponding to a particular window on the screen, will be locked by the kernel to be accessible by the game's process [SGG13].

Kernel I/O Data Structures

The kernel must, as is rather obvious, keep track of what I/O devices are in use by which processes and the general state of the I/O device ("PC Load Letter"). The diagram below gives some indication of what the UNIX kernel structures are like for open files:



The UNIX structures to manage I/O [SGG13].

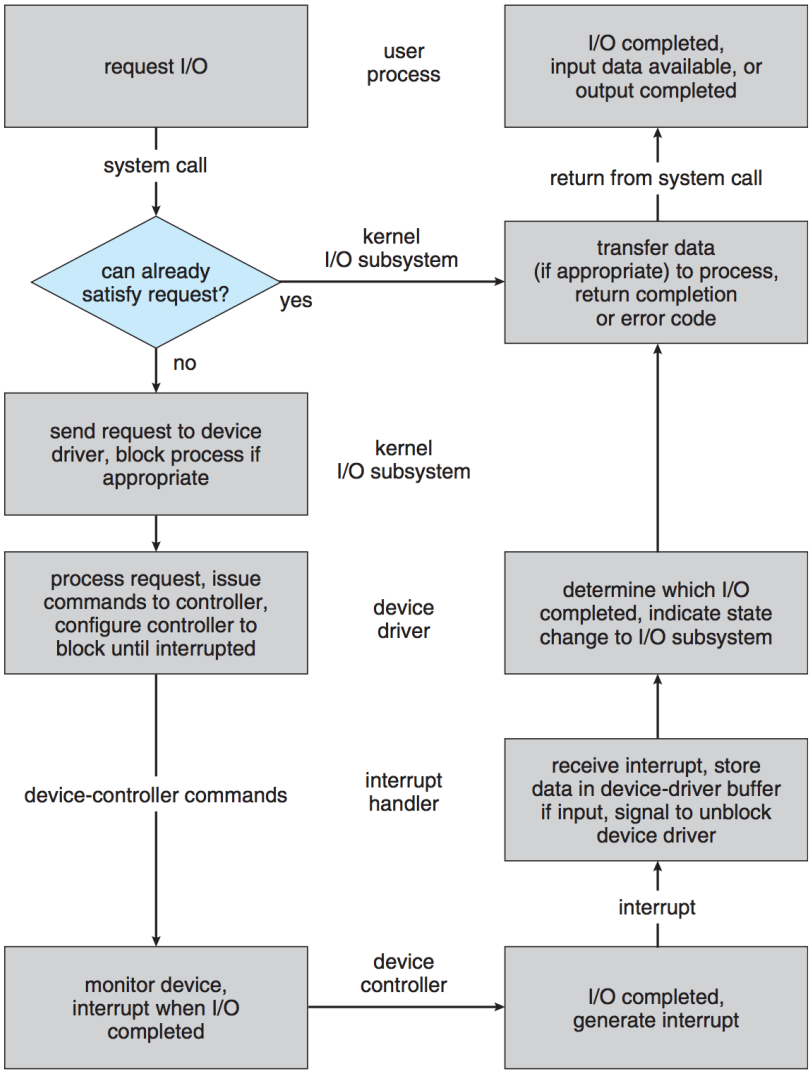
This now explains why, if you print a file descriptor number and you see something like 3 or 4 – that's the index into the per-process file table, which itself just references the system open-file table.

20 — More About Input/Output Devices

Transforming I/O Requests to Hardware Operations

In the previous section we discussed the idea of taking a command like `read` and said it is the job of the device driver to translate this command into a hardware operation. Reading from the file system on disk, for example, requires a few steps. If I want to open a file like `example.txt`, the file system (not yet discussed) will associate this file with some information about where on the disk this is (a set of disk blocks representing the file). Then, to read the file, `read` commands can be issued to get those blocks into memory so I can edit it with `vi`.

The diagram below shows the life cycle of an I/O request:



The life cycle of an I/O request [SGG13].

In general, the life cycle follows some ten steps from start to finish [SGG13]:

1. A process issues a `read` command (assume the file is already open).
2. The system call routine checks the parameters for correctness. If the data is in a cache or buffer, return it straight away.
3. Otherwise, the process is blocked waiting for the device and the I/O request is scheduled. When the operation is to take place, the I/O subsystem tells the device driver.
4. The device driver allocates a buffer to receive the data. The device is signalled to perform the I/O (usually by writing into device-control registers or sending a signal on a bus).
5. The device controller operates the hardware to do the work.
6. The driver may poll for status, await the interrupt when finished, or for the DMA controller to signal it is finished.
7. The interrupt handler receives the interrupt and stores the data; it then signals the device driver to indicate the operation is done.
8. The device driver identifies what operation has just finished, determines the status, and tells the I/O subsystem it is done.
9. The kernel transfers the data (or error code or whatever) to the address space of the requesting process, and unblocks that process.
10. When the scheduler chooses that process, it resumes execution.

Buffering

Regardless of whether a device is block- or character-oriented, the operating system can improve its performance through the use of buffering. A buffer is nothing more than an area of memory that stores data being transferred, whether it is from memory to a device, device to memory, or device to device. A buffer is a good way to deal with a speed mismatch between devices.

You may be familiar with the idea of buffering from something like watching an online video. If each piece of the video is delivered from the server to your computer exactly when it's needed, even a small slowdown or hiccup in your network connection results in the video pausing or stuttering until the next chunk arrives. To have a smoother experience for the viewer, the video player most likely gets and stores the next few chunks a bit in advance, so if there is some temporary delay in getting a chunk then the playback is not interrupted.

The video player example is simple and familiar, but it covers all the important parts. We need a place to store the data temporarily to decouple the producing (sending from the server) and consuming (playing the video). It requires some decisions about how big the buffer should be. Bigger is not always better, mind you: if the video player wants to have n chunks before starting the video, then as n grows, it delays the start of playing the video. If n is too large users are frustrated because the video does not start playing for a long time. The other reason is that some extra, unnecessary work will be done: if the viewer starts the video but then decides to skip forward 30%, the player will have downloaded some chunks that were not needed. If you have a very fast internet connection, maybe this doesn't matter very much because the costs are low.

The following calculation is crude, but it does illustrate at least a little bit the performance impact of a buffer, from [Sta18]: We'll say T is the time required to input one block and C is the computation time between input requests. With no buffer, the execution time to complete a block is $T+C$; with a buffer it is $\max(C,T)+M$, where M is the time to move the data from a system buffer to process memory.

A buffer is usually created with a bounded capacity: the buffer can hold x bytes of data and once that buffer is full, nothing else can fit in it right now so there's choices to make: either the source of data can be blocked until space is available, or older data can be overwritten with the newer data. Sometimes books talk about unbounded capacity, where the queue length can be infinite, but that's just an illusion. The memory space of the system where the data is to be stored is not infinite, even if it can seem so big that it appears to be unlimited. At some point, space *will* run out, even if it's after 30 Terabytes, and then we have the same problem as the bounded buffer size: what do we do when it's full? Tempting as it is to not handle that situation, probably we should.

Implementing a Buffer. Actually implementing a buffer is relatively straightforward operation from the point of view of the operating system because it really only requires two operations: add something to and remove something from the buffer. The implementation looks exactly like the producer-consumer problem covered in a certain prerequisite course. Here's the pseudocode version of it that you might be tired of seeing:

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]
```

The important thing to note is that the use of the mutex and semaphore will block and unblock threads as necessary when the buffer is full, busy, or empty. We already covered earlier in the course the implementation of those constructs so there's no need to revisit that. You may also choose to use the simpler version without the mutex if there's only one producer and one consumer.

Double Buffering and Circular Buffering. Users type very slowly, from the perspective of the computer, and if we're writing to a file, it would be awfully inefficient to ask the disk, a block oriented device, to update itself on every single character. It is much better if we wait until we have some certain amount of data (e.g., a whole line or whole block or full buffer, whichever it is) and then write this out to disk all at once.

The write is, however, not instantaneous and in the meantime, the user can still keep typing. Thus, to solve this, the typical solution is *double buffering*, that is, two buffers. While buffer one is being emptied, buffer two receives any incoming keystrokes. Double buffering decouples the producer and consumer of data, helping to overcome speed differences between the two [SGG13].

The obvious extension beyond having two buffers is having more than two, so n buffers. In that case, it's referred to as circular buffering, with each individual buffer being one of many in the circular buffer [Sta18].

Not Magic. Buffering, is, however, not capable of solving all problems. If the rate at which the buffer or buffers is/are being filled consistently exceeds the rate at which the buffer or buffers is/are being emptied, at some point the buffers will be full. When that happens, the producer side is going to get blocked until such time as space is available. The converse also applies: if the buffers are being emptied much faster than they are being filled then the consumer side will get blocked instead. Buffering does help to smooth out the peaks and valleys in the speed of these, but only up to a certain point.

I/O Scheduling and Performance

There are potentially multiple I/O requests in progress at a time, so the operating system will need to keep track of these I/O requests in some sort of device-status table that contains an entry for each I/O device [SGG13]. Like we talked about for, say, a semaphore, the OS needs to keep track of the device status and also have an associated queue where threads that are blocked for that device are waiting.

Managing that queue looks significantly similar to concepts we've already examined. When a thread wants to use the device, check the device status: if it's available then we can mark the device as busy and submit the request to the device. Until the operation is complete, mark the requesting thread as blocked. If a thread shows up and wants to use a device that's already in use, just block it and add that thread to the queue. When the operation is complete, unblock the thread waiting for it and let it continue. If there are no further requests waiting, mark the device as idle (not in use). Otherwise, keep the status as busy and begin the next request.

This scheme is going to be first-come-first-served and in the discussion of how to implement a semaphore or mutex, that was typically adequate as an approach. It's very fair, but maybe not optimal. We learned in discussing scheduling that sometimes things like priorities matter more than absolute fairness.

We sometimes want to schedule I/O requests differently because I/O devices may have non-uniform access times. Many devices do have uniform access times: if we're reading a file off a USB flash drive, it doesn't matter where

on the drive the drive the data is physically located. That's not true for all devices. And when it's not, it matters for the sake of efficiency.

The efficiency part is easy to motivate if we refer back to the diagram discussing the speed of various devices. I/O devices are very slow compared to the CPU so anything that uses the slow device more efficiently will speed up the execution of the system significantly.

A simple analogy: Imagine you need to go to the grocery store, the dry cleaners, and the bank. The bank is located 1 km to the west of your current location, and the grocery store is 3 km west. The dry cleaners is in the same plaza as the grocery store. It is obvious that it would be fine to go to the bank, then the dry cleaners, then the grocery store, but not to go to the dry cleaners, then the bank, then the grocery store. The unnecessary back-and-forth wastes time and energy (whether walking or fuel depends on your mode of transportation).

Clearly, the operating system will want to avoid wasting effort with I/O requests. It will maintain a structure of requests and can then re-arrange them to be accomplished most efficiently. The literature sometimes refers to this as a queue but... is it really a queue when it does not exhibit the first-in, first-out behaviour?

This will, naturally, have some limits: requests should presumably get scheduled before too much time has elapsed even if it would be "inconvenient". It might also take priority into account, dealing with the I/O requests of a high priority process even if they are not particularly nearby to other requests. Like what we've discussed around scheduling, there are tradeoffs necessary to balance utilization and fairness. In particular, this is important when examining hard disk drive operation. And that will therefore be the next subject we will consider (but in the next course topic).

Can a System Be Too Responsive? We've discussed the idea already that interrupts are more efficient than polling, you might question if there's such a thing as too responsive. Network traffic, for example, may be the cause of a lot of interrupts and a very high rate of context switches [SGG13].

For downloading a file, say, the number of interrupts is probably manageable: as each chunk arrives, the interrupt is triggered, and the chunk is handled. Eventually, all the file is downloaded and there we are. That's not the scenario of concern; the scenario of concern is what happens if I use `ssh` to log in to one of the ECE servers and then use `vim` to open the C file I plan to work on.

Every time I type a character on my keyboard, a keyboard interrupt takes place, the kernel handles it, sends it to the user process (`ssh` client), which then wants to send it to the remote system. To do so, it uses a network I/O system call to send it and it goes out to the remote system. The remote system gets an interrupt on the arrival of the packet and it handles it by delivering it to the appropriate process (`vim`). And then `vim` will process the keypress and add the letter to what I'm typing. But it then updates the screen so another network request is generated from the remote machine and sent back to mine, where another interrupt is generated and has to be handled to update the view on my screen [SGG13]. That's a lot of system calls every time I press a key. I personally don't type super quickly from the point of view of the computer system, but imagine it's the day an assignment is due and a hundred people are using the system.

How to address that, then? In the book there are two approaches that I would describe as somewhat questionable. The first one is the idea of having the `sshd` program (well, the book says `telnet` but let's not revisit ancient history... this time) run in kernel mode so there are fewer switches. That's kind of risky, if you ask me... The other thing suggested is having some sort of specialized hardware to handle terminal connections, some sort of specific-purpose CPU to handle this sort of workload. That's weird too, because we typically do not want to buy specialized hardware just because we log into the box remotely, right? But on the other hand, there are situations where we have worked with special-purpose hardware.

Things like the translation lookaside buffer were a specific example of taking logic that was in software and moving it to hardware to make it faster. For complex devices, including hard drives, some logic will be pushed to the actual device itself, which can speed up the operations and simplify the operating system's interaction with the device. There is a cost in terms of money, of course, because more hardware on the device costs more money. It's also harder to fix any issues that are discovered because it may be expensive or impossible to fix the hardware bug. Sometimes such a hardware bug can be worked around in software, but that may undo the performance benefit of shifting it to hardware in the first place.

Other solutions look like reducing the amount of time it takes to handle an interrupt, waiting until a bigger chunk

of data has arrived (or a time limit reached), or make use of DMA hardware to reduce the involvement of the CPU in moving the data around. Those might actually be plausible for the majority of scenarios.

Security

As device drivers run at a high level of trust in the operating system, they offer an easy route for attackers to be able to get access they are not supposed to have. If a driver allows for privilege escalation or snooping the data of another process, attackers will make use of it. Let's consider an examination of a malware technique called BYOVD – Bring Your Own Vulnerable Driver, which is nothing new having been around since at least 2012. The source for this is [Goo22].

Of course, the vulnerable driver has to be on the target system in the first place. If an administrator wishes, they can intentionally install a driver with a known vulnerability to be able to do something bad. But operating system design can't really defend against this sort of thing because the attacher is already a system administrator so they can already do bad things in other ways.

In principle, bad drivers should not get installed automatically because of driver signing that we talked about earlier on. So a deliberately-malicious driver is unlikely to get past the validation process and the seal of approval. Trying to install a driver that's not signed tends to result in the operating system throwing a ton of warnings to say please be sure you want to do this...

Ah, but this doesn't solve the problem of what happens if there's a vulnerability in an otherwise-legitimate driver. Such a driver would have been tested, approved, and signed and the problem is only discovered after the driver is released. If a fix is prepared and published – and that is a big if sometimes – then anyone who installs the new version is fine (even if not everyone installs patches). But that doesn't prevent installing the old version with the vulnerability.

What is needed is some idea of revocation: a previously-approved driver should be possible to un-approve in some way so that the OS won't allow the vulnerable version to be used. The article reveals that Microsoft has not been properly applying updates to the driver block-list which resulted in exploitable drivers being installed. Oops!

Microsoft did address this once it was reported, but a little late: there were some known cases of actual exploits taking place using drivers in the block-list. And as you may imagine, the installation rate for patches to any operating system is never 100%, so vulnerable systems will continue to exist...

21 — Disk Scheduling

Disk Scheduling

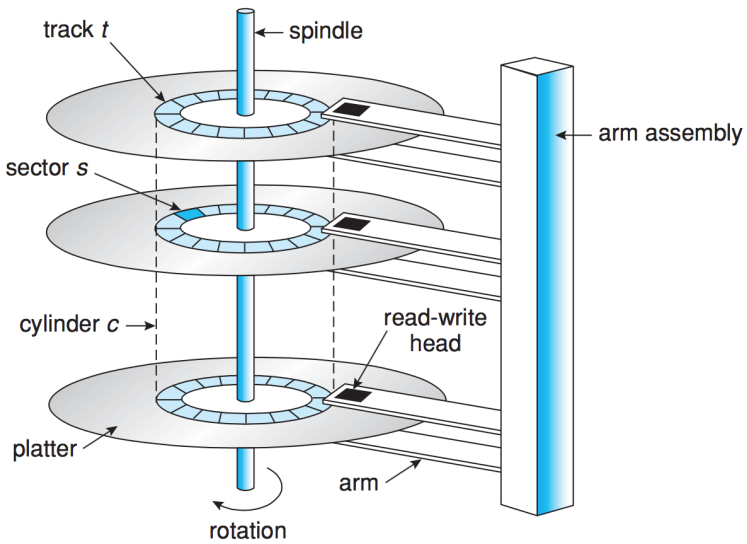
The disk is a very slow device, as far as the CPU is concerned, so we want to avoid going to disk, if it can be helped, whenever possible. We will examine the hardware behind magnetic hard disks and see why they are so slow and consider several algorithms that could be used to speed up operations.

I'm going to imagine that you got a solid-state drive (SSD) in your computer when you purchased it. But back before those were common, the mass and permanent storage of data in your system was on a magnetic hard disk. Due to the physical nature of how disk drives work, there is an associated delay with moving to a new location and reading the data there. Thus, we would like to devise efficient schedules for reading and writing, just as in the example from earlier, we wished to avoid going back-and-forth when completing a set of weekly errands.

We will exclude SSDs from consideration here. SSDs do not contain moving disk heads or spinning platters. Thus, an algorithm to schedule the optimal movement of the disk heads is not relevant. Read access times from the SSD are consistent and uniform even if data is being requested from random locations. So, the SSD disk scheduling algorithm can be as simple as first-come, first-served.

That prompts the question: why bother learning about this if magnetic drives are going away? The first is that while magnetic drives might be on the way out, they're not dead yet and many systems still use them, including large databases. The other is that disks are a convenient example, but not the only I/O device that needs to seek around to read and write data, so what we examine here may be applicable to other sorts of devices too.

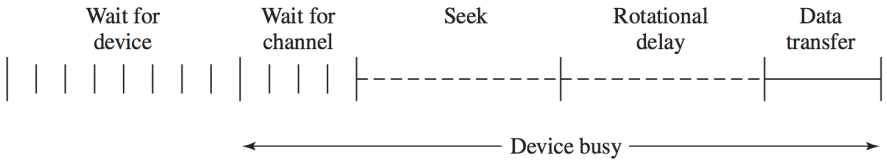
The diagram below is a quick overview of what hard drives look like internally. Hard drives come in carefully sealed packages and if you open one (please don't; this is very bad for your ability to read data from it) it looks something like this:



Moving-head disk mechanism [SGG13].

A read-write head is suspended a very small distance above the surface of each platter and reads or writes data directly beneath it. This is why they are carefully sealed: if a bit of dust lands on the platter, the read-write head can run into it (which is bad). It gets even worse if a disk head makes contact with the platter: this is called a head crash and it tends to permanently destroy data. The platter surface is divided logically into different circular tracks, which are respectively divided up into sectors (blocks). A set of tracks stacked vertically is called a column. When the disk is in use, a motor spins the platters at high speed and another one manipulates the positions of the arm.

The performance of the disk can be broken down into two values. The first is the *transfer rate*; the speed at which data can be moved from the disk to the computer or vice-versa. The other is the *random-access time*; how long it takes to get to a particular piece of data. The random access time itself is broken down into the *seek time*, how long it takes to move the disk arm to the right position, and the *rotational latency*, how long it takes to rotate the platters to the right position. Seek times and rational latencies tend to run in the millisecond range [SGG13].



Disk I/O transfer timing [Sta18].

The total average access time, T_a, for a disk operation can be defined as [Sta18]:

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

Where: T_s is the average seek time, r is the rotation speed (revolutions per second), b is the number of bytes to be transferred, and N is the number of bytes on a track.

Example. Let's take an example from [Sta18] that illustrates just how much variance we can have in disk performance. In this scenario, the disk has an average seek time of 4 ms, a rotation speed of 7 500 RPM, and 512-byte sectors with 500 sectors per track. We want to read a file that is 2 500 sectors (a total of 1.28 MB).

If the file is stored as compactly on disk as it can be, it is *sequentially organized*: the file occupies all the sectors on 5 adjacent tracks. To read the first track, it will take 16 ms: 4 ms to seek, 4 ms rotational delay, and then 8 ms to read 500 sectors. Because the data is sequential, no additional seek time is necessary, so we just keep reading subsequent sectors which means only the rotational delay. Each additional track takes 12 ms (4 ms rotational delay + 8 ms to read it). Thus the total time is $16 + (4 \times 12) = 64$ ms.

What if instead the data was randomly distributed on the disk and not sequential? The average seek and rotational delay times don't change. Reading one sector takes 0.016 ms. So each read of a sector will take a total of 8.016 ms $(4 + 4 + 0.016)$. There are 2 500 sectors so the total is $2\,500 \times 8.016 = 20\,040$ ms.

20 000 ms - twenty full seconds to read 1.28 MB of data? Yes. Seriously. This speed would be considered utterly unacceptable by the users. So it is clear that the order in which sectors are read from the disk makes a huge difference. This suggests that decisions about how to store data on disk is extremely important. Not only is the placement important, but how we schedule the reads and writes.

Disk Scheduling

We should introduce a final metric, the *bandwidth*: the total number of bytes transferred, divided by the total time between the request for service and completion of the transfer. This is a measure of how much data is effectively transferred in a period of time. This is one of the measures we would like to improve, as well as the access time.

When a process needs to read from or write to the disk, the system call contains the following information [SGG13]:

1. If the operation is a read or write.
2. The disk address for the transfer.
3. The memory address for the transfer.
4. How much data to transfer (how many sectors).

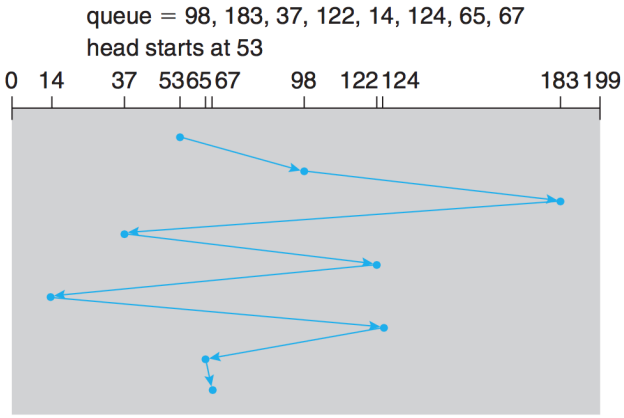
Nobody actually seriously advocates random scheduling, but it is a baseline against which to compare various scheduling algorithms. We can assess the various options based on their improvement relative to random scheduling. We could do disk accesses based on process priority, paying no attention to how (in)convenient it is from the perspective of the disk; the highest priority process's request gets serviced first.

Similarly, if there is only one request for a read or write, there is not much of a decision to make. If the disk is otherwise idle, service that request, regardless of where it is. Assuming we are not going to choose the priority option and we have more than one operation to perform, we have some decisions to make.

First-Come, First-Served

This algorithm, also sometimes called First-In, First-Out, is fair and simple, but does not necessarily provide the fastest service because no attempt is made to group, organize, or rearrange the requests.

Example: requests for I/O to/from cylinders: {98, 183, 37, 122, 14, 124, 65, 67}; the disk head begins at 53.



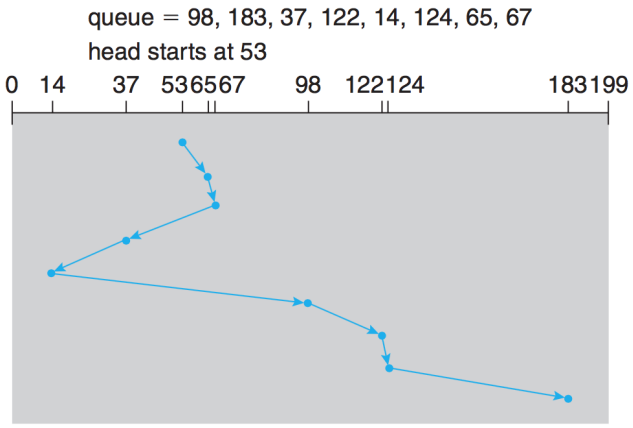
FCFS disk scheduling [SGG13].

So the disk head moves from 53 to 98, then 98 to 183, and so on, for a total movement of 640 cylinders. This example intentionally includes the sequence {122, 14, 124} to illustrate the problem: a wild swing back and forth over a long distance. It would be obviously better to do visit 122 and 124 after one another.

Shortest Seek Time First

Based on the idea that we want adjacent operations to take place consecutively, instead of just choosing the next item in the queue, choose the request with the least seek time from the current head position.

Using the same example as before, the closest request to the start position of 53 is cylinder 65. From there, 67 is next, then 37, 14, 98, 122, 124, and 183. Following this routine, the total head movement is 236 cylinders, a remarkable improvement over the 640 from before.



SSTF disk scheduling [SGG13].

This routine is unfortunately subject to starvation. Suppose the disk head is at position 14. While is is there, a new request at 24 arrives, making the request at 98 wait. If enough low-numbered requests arrive to arrive during execution, a request at a high number may be put off, potentially indefinitely. The more requests occur, the more likely starvation is.

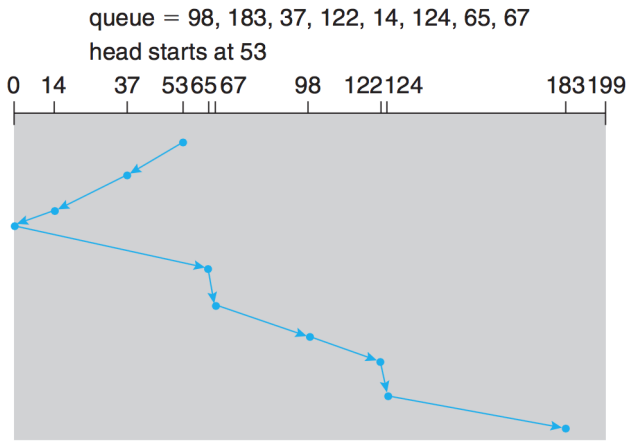
Though the SSTF algorithm is an improvement, it is not optimal. In the example, we would reduce the total amount of movement to 208 cylinders if we did the move from 53 to 37, even though it is not the closest, and then to 14, before moving to 65, 67, 98, 122, 124, and 183 [SGG13].

This algorithm provides for some spatial locality. If there are a number of requests that are located near one another on the disk, the SSTF algorithm takes advantage of this and will carry out all those requests relatively quickly.

SCAN Scheduling

In the SCAN algorithm, instead of a "zig-zag" pattern where we might move in one direction and then immediately double back to another, such as 25 then 32, then 17, we move in one direction at a time until it reaches the "end" of the disk. After that, the direction is reversed. The SCAN algorithm is sometimes called the elevator algorithm, because it works like an elevator in a building; first all requests going up are serviced, then the elevator reverses direction and services all requests going down [SGG13].

Let us assume the current direction of the head (starting at 53) is moving towards 0. So the algorithm will go to: 37, 14, then reverse direction and go to 65, 67, 98, 122, 124, and 183. If a request arrives just in front of the head (e.g, 129 arrives before 124 is serviced), it will be serviced virtually right away; if it is just after where the head has been (e.g., 123 after 124 has been serviced) it will wait until the direction reverses again.

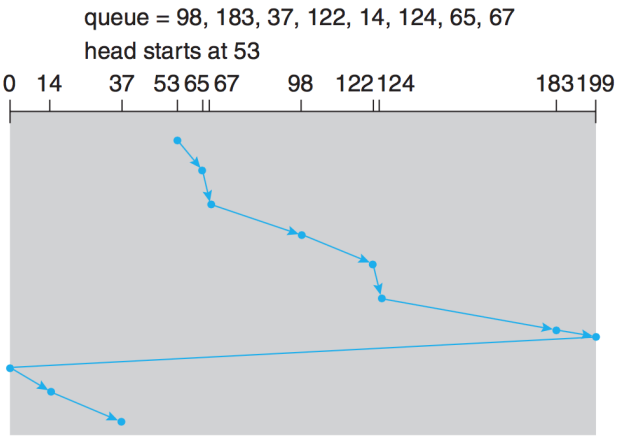


SCAN disk scheduling [SGG13].

Note that the SCAN policy does not take advantage of spatial locality as SSTF does. If the head has just moved from an area, it can be a long time before it returns to that area. Thus, if there are multiple accesses in the same area, they are likely to be at least partly spread out in time.

C-SCAN Scheduling

An improvement on SCAN, known as C-SCAN is designed to exploit the fact that when the disk has just reached one end, most requests are likely at the other end of the disk (if they were nearby they would have been serviced already). So instead of reversing the direction and servicing requests on the way, jump back to the start of disk immediately and start at the beginning. It is called C-SCAN because C is for "Circular"[15], as if the last cylinder just wraps around to the first.

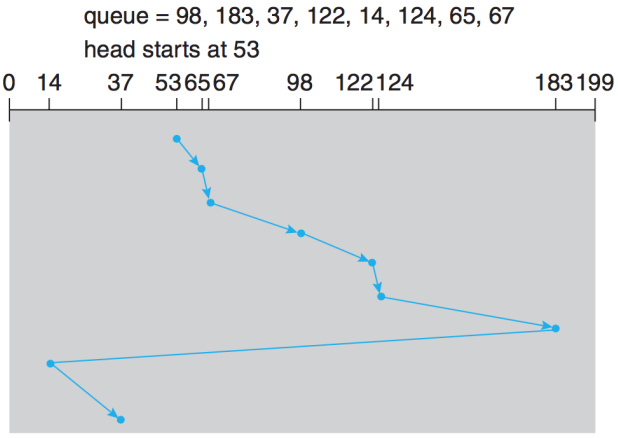


C-SCAN disk scheduling [SGG13].

If the amount of time it takes to scan from the start to end of disk is t, the expected service interval for sectors at the periphery is $2t$ when using the SCAN algorithm. C-SCAN reduces this interval to approximately $t + s_{max}$ where s_{max} is the maximum seek time [Sta18].

LOOK Scheduling

The SCAN and C-SCAN algorithms as described can be optimized in a little way: instead of going all the way to the end of disk every time, instead just go to the final request and only then reverse direction (or go back to the start). The names for this variant are LOOK and C-LOOK, respectively.



C-LOOK disk scheduling [SGG13].

[15] C is also for cookie.

Some More Details of Scheduling Algorithms

Although we can probably agree that FCFS is not the best choice, how do we decide between the other options? The LOOK approach seems like it would prevent the starvation problem, which might otherwise would be an issue with SSTF. Although we expect that with the SCAN or C-SCAN algorithm, all requests get serviced, it could happen that a process is effectively starved or has to wait an inordinately long time because the request at the end of the disk is constantly postponed as more and more and more requests come in.

A strategy to prevent this is a modification of the SCAN algorithm such that it has two queues for requests (think double buffering). While one queue is being emptied, the other is being filled. Thus, a request will not wait indefinitely; if the queue is of capacity C then any particular read or write will wait, at most, C accesses. The larger the value of C, the more the performance resembles SCAN; a smaller value of C means behaviour is more like FCFS. The choice of C is a trade off; we will sacrifice a bit of performance to increase the fairness [Sta18].

The scheduling algorithms above consider only the seek times, and not the rotational latency, even though they can be about the same size. This is because it is very difficult for the OS to schedule for improved rotational latency, because the disk itself is responsible for the physical placement of the logical blocks. So, to make this a little easier, the hard disk controller takes on some of the scheduling options. The OS can provide to the controller a grouping of requests, and then the controller will figure out how to schedule them such that it takes into account the rotational latency as well as the seek time [SGG13].

If the speed of disk reads and writes were the only thing to be concerned about, the operating system would probably not worry about disk scheduling and it would just be the job of the hard disk controller. However, the OS may have certain goals that should take precedence over the highest performance option. For example, loading a page into main memory might need to take priority over an application writing a file to disk. Higher priority processes, especially in real-time systems, should not be waiting (for too long at least) for lower priority processes' disk writes. So, under some circumstances, the operating system needs to manage the reads and writes and not just leave it up to the disk controller [SGG13].

22 — File System Implementation

File System Implementation

Now it is time to go behind the scenes of how the file system lives up to the interface we have just finished discussing. The implementation is somewhat complicated, and to keep the size of the problem manageable we will worry about storing files on hard disks. Hard disks themselves make a good choice for this: they are sufficiently large and sufficiently cheap, to start with, to store the data that we want to store. We can read an arbitrary part of the disk. Finally, we can write to the same part of disk as many times as we want (disks don't, at least on the time frames that we are concerned about, wear out). Recall also that disks operate on blocks which, in their physical representation, comprise one or more sectors.

Let us take a look at the layers of the file system's design, from [SGG13]. As we descend down the list we will get closer to the hardware and less abstraction.

The File System. The file system user interface is intended for the convenience of the user and for application programmers. This is the level we have just examined in the previous topic; the more interesting part is what happens when we need to fulfill the promises that the file system makes.

I/O Control. At the I/O control level, we are dealing with device drivers and interrupt handlers to transfer data. The inputs are fairly high-level commands, along the lines of "read block 1234". Its outputs are hardware specific-instructions to the hardware controller. Usually this is by writing bit patterns in the I/O controller memory.

Basic File System. Despite the awful name, this is the level at which we start to deal with physical blocks on the disk. A physical block is identified by its numerical physical address: drive 0, cylinder 12, track 7, sector 1. This layer is also responsible for buffers and caches that are used to hold various commonly-accessed regions (e.g., the temporary directory). Yes, caching and buffers can appear in the hard disk as well, with performance improvements traded off against the risk of data loss if power is suddenly cut.

File Organization Module. This module is aware of files and their logical and physical blocks. This translates a logical block address to a physical block address and keeps track of free space (unallocated blocks).

The Logical File System. This last level is for managing metadata: file system structure, directory structure, and maintaining the file structure. File data is maintained in a *file control block* (FCB). The UNIX term for this is *inode* and it is the place where file info is stored: ownership, permissions, locations of the file contents.

Disk Organization

Although there are a million different file systems (UFS, HFS+, ZFS, NTFS, ext3, FAT32...) that are all significantly different, there are some general principles that we can examine. A file system will need to keep track of the total number of blocks, the number and locations of free blocks, the directory structure, and the files themselves.

On at least one disk somewhere in the system, there will need to be some information about how to boot up the operating system. Not every disk has the operating system on it, but if there is an OS the boot loader is usually put in the first block. When the power button is pressed on the case, the BIOS starts up and transfers control to

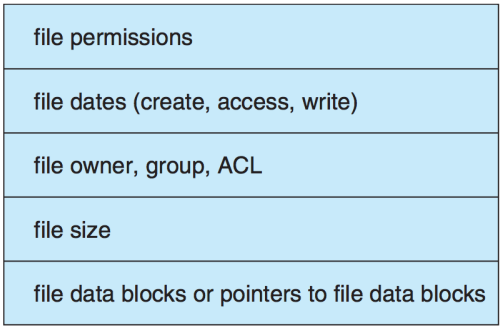
whatever is found at that first block (which is hopefully your boot loader that launches the OS, or gives you the option of which OS to start).

Disks may be split, logically, into several different areas, or *partitions*. Accordingly, there will be a partition table (sometimes called the superblock or master file table) that indicates what part of the disk belongs to which partition. In the Windows world we often see the whole disk is in one logical partition (the C: drive, for example, taking the whole primary disk). In Linux we often see the disk divided up to have partitions for different things: temporary/swap directory, home directories, boot partition, et cetera...

There are several structures that are likely to be in memory for performance reasons [SGG13]:

1. **Mount Table**: Information about each mounted volume (disk/partition).
2. **Cache**: Directory info for recently accessed directories.
3. **Global Open File Table**: Copy of the FCB for each open file.
4. **Process Open File Table**: References to the global open file table, sorted by process.
5. **Buffers**: places where data read from or to be written to disk resides after or before the actual disk operation.

Creating a new file is the job of the logical file system: allocation of a new FCB (or re-use of an existing free FCB). A typical FCB might look like this:



File Control Block (FCB) example [SGG13].

If a user (application) actually wants to make use of a file, the open system call is needed. Open operates on names, the user comprehensible version. The file system must then check the global open file table to see if the file is already open somewhere in the system. If so, if that file is opened for exclusive access, then the open routine returns with an error. If it is open but for non-exclusive access, then there is no need to search for the file or retrieve it by directory; we just make another reference in the process open file table. If the file is not already open, then it needs to be retrieved; once the file is found, the FCB is copied into the global open file table and the appropriate reference is added to the process table [SGG13].

The process open file table can contain some additional information, like the next section to read or write, the access mode when the file is open, and so on. The open system call returns a pointer to the file table and this is the route through which the application performs all file operations. In UNIX this reference is called a *file descriptor*; in Windows it is called a *file handle* [SGG13].

The opposite operation to opening a file is obviously to close it. When a process closes a file, the entry from the process open file table can be removed. If this is the last reference to the file a process has, then the file can also be removed from the global open file table. Metadata may be updated on close.

Searching with Spotlight. A small aside on the subject of metadata.

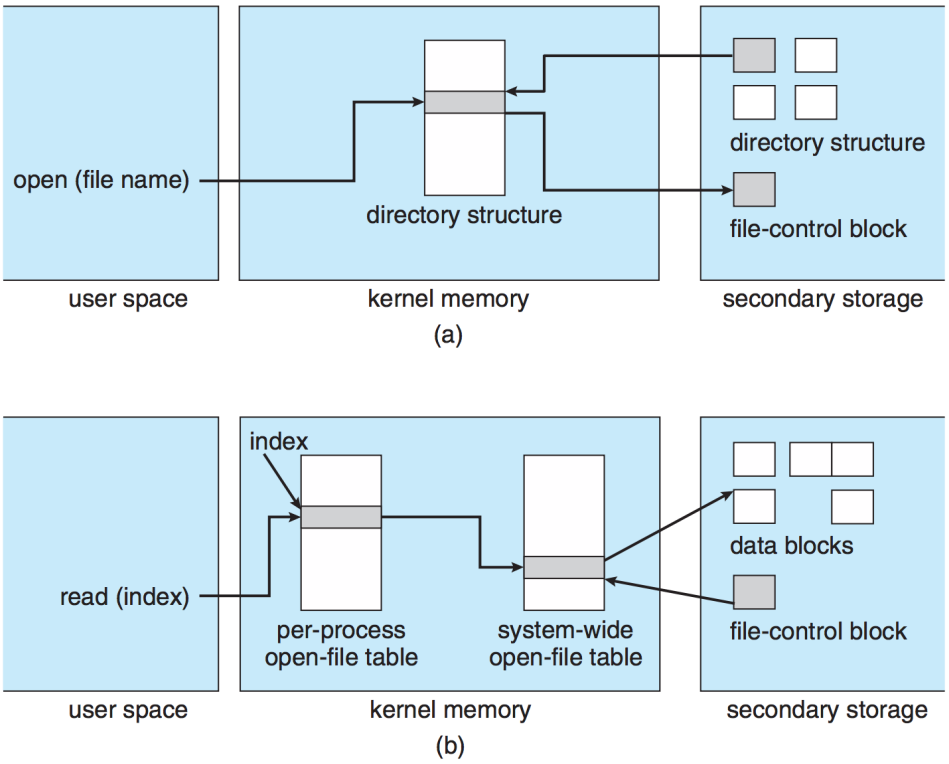
An example from Mac OS X: the Spotlight system-wide desktop search. Spotlight was introduced quite some time ago but it was a revelation compared to the previous search options that were slow and did not necessarily include

the file content, just the file names. Spotlight runs queries quickly over the metadata of the system. Spotlight examines each file on creation and modification and uses it to update the metadata related to that file. Those items are then added to the Spotlight index which can be queried very rapidly [Sir05].

This is a specific case of preparing the data in advance to make searching or exporting operations fast. Another example is in exporting data to Excel. To export this data, there are two possible ways to do it. One is to examine and process the data for the Excel file at the time of the user request. The other is to maintain that data separately, and when the Excel report is asked for, take that data and put it in the Excel file.

Part of the difficulty with maintaining metadata is when to update it. The longer the time between metadata updates, the higher the chance that there will be a user request that gets back out-of-date data. The faster the metadata updates, the more time and processing power is spent creating and updating this metadata.

The diagram below shows how a file is opened and how a read takes place:

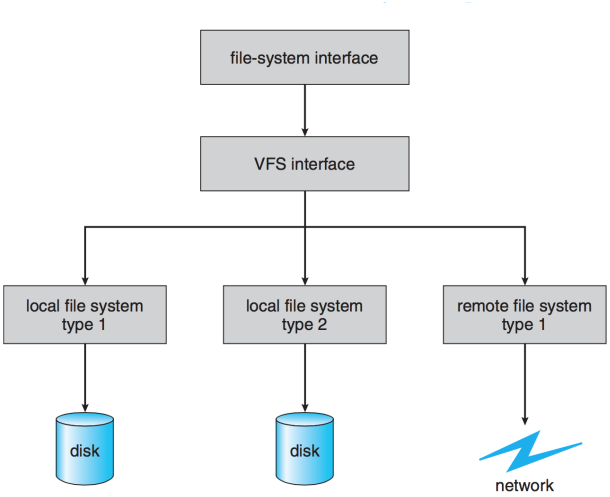


File System Structures for (a) file open and (b) file read [SGG13].

Virtual File System

We mentioned earlier that there are a lot of different file systems, like ext3, ZFS, etc. These may co-exist in the same system, though a partition will be formatted in only one of them. From the user's perspective, however, these operate identically. This is because of an additional layer of abstraction, called the virtual file system (VFS).

There are two main purposes to the VFS. The first is to separate the file system operations (like reading, writing, opening, and closing files) from the actual implementation; thus operations can be done with whatever file system is actually implemented. The second is that it provides a mechanism for representing a file, uniquely, throughout a network. The file representation structure is called a *vnode*, which is rather like an inode, but inodes are unique only within one file system. The VFS distinguishes between local and remote files [SGG13].



Schematic view of the VFS Interface [SGG13].

The VFS architecture in Linux has four main objects [SGG13]:

1. inode (an individual file)
2. file (an open file)
3. superblock (the file system)
4. dentry (a directory entry)

For each of those, there are a set of operations defined. In the Linux kernel, these operations are defined in the `fs.h` headers and may include functions such as:

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *, fl_owner_t id);
int (*release) (struct inode *, struct file *);
```

The VFS therefore abstracts away a lot of the details of the underlying file system implementation.

Directory Implementation

See the table below for some information about what data may appear in a typical file directory.

Basic Information	
File Name	Name as chosen by creator (user or program). Must be unique within a specific directory
File Type	For example: text, binary, load module, etc.
File Organization	For systems that support different organizations
Address Information	
Volume	Indicates device on which file is stored
Starting Address	Starting physical address on secondary storage (e.g., cylinder, track, and block number on disk)
Size Used	Current size of the file in bytes, words, or blocks
Size Allocated	The maximum size of the file
Access Control Information	
Owner	User who is assigned control of this file. The owner may be able to grant/deny access to other users and to change these privileges.
Access Information	A simple version of this element would include the user's name and password for each authorized user.
Permitted Actions	Controls reading, writing, executing, and transmitting over a network
Usage Information	
Date Created	When file was first placed in directory
Identity of Creator	Usually but not necessarily the current owner
Date Last Read Access	Date of the last time a record was read
Identity of Last Reader	User who did the reading
Date Last Modified	Date of the last update, insertion, or deletion
Identity of Last Modifier	User who did the modifying
Date of Last Backup	Date of the last time the file was backed up on another storage medium
Current Usage	Information about current activity on the file, such as process or processes that have the file open, whether it is locked by a process, and whether the file has been updated in main memory but not yet on disk

Information elements of a file directory [Sta18].

A directory is, after all, not much beyond a set of files. And yet, there is a choice to be made in implementing them: a linear list versus a hash table. Both concepts should be familiar to you from data structures and algorithms.

The linear list option is certainly the simplest. Creating a new file requires searching the directory to see if there is a matching file name. If not, insert that new entry in the list. Deletion is also simple: search the list for the matching file and remove it; if it is the last reference to that file, we can free up the space. The real disadvantage to linear lists is that searching takes a long time. Searching through a directory can take quite a while, and the problem is compounded if the search is supposed to include the content of the file as well.

Alternatively, we could use a hash table. The hash is computed based on the file name, and of course there will need to be some strategy for dealing with hash collisions. Still, neither of these is satisfactory; what we really want is a more complex structure: a tree.

An AVL tree might be good as a way of storing ordered data, but it does not match well with the block system of a hard drive. Instead, we will examine a B-Tree, a variation on a binary search tree.

Each node should occupy one block, so each node can contain a lot of information. File and directory information is linearly ordered, but a block does not need to be full. If a leaf node gets full, we can split it into two half-full blocks. Keeping the tree balanced means it takes about the same amount of time to find an item wherever it is.

A B-Tree structure, formally, has the following characteristics [Sta18]:

1. The tree is made up of nodes (have children) and leaves (have no children).
2. Each node contains at least one key identifying a file record, and more than one pointer to child nodes or child leaves.
3. Each node has some maximum number of keys.
4. The keys in a node are stored in non-decreasing order.

And a B-tree of degree d has the following properties [Sta18]:

1. Every node has at most $2d - 1$ keys and $2d$ children ($2d$ pointers).
2. Every node other than the root has at least $d - 1$ keys and d pointers. Each internal node except the root is at least half full and has at least d children.
3. All leaves appear on the same level.
4. A non-leaf node with k pointers contains $k - 1$ keys.

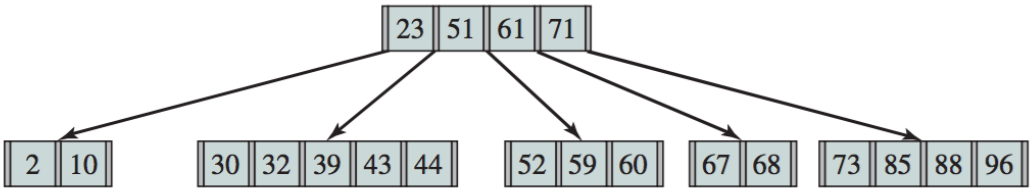
To find something in a B-Tree, the general algorithm is fairly simple [Sta18]:

1. Start at the root node.
2. If the key is in the current node, it is found, and the algorithm terminates.
3. If the key is less than the smallest key in this node, follow the leftmost pointer; go to step 2.
4. If the key is greater than the largest key in this node, follow the rightmost pointer; go to step 2.
5. If the key is between the values of two adjacent keys, follow the pointer between them; go to step 2.

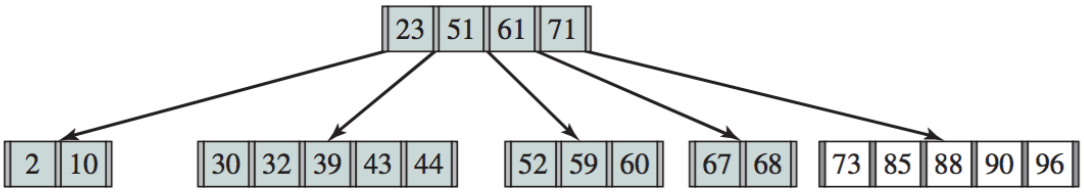
Insertion into the B-Tree is more complicated, because of the rules that keep the tree balanced [Sta18]:

1. Search the tree for the key. If it is not found, then we are at least looking at the block where it would be if it were there.
2. If this node has fewer than $2d - 1$ keys (that is, it is not full), insert the key into this node in the proper sequence. The algorithm terminates.
3. If the node is full, split this node around the median key into two new nodes with $d - 1$ keys each. Promote the median key to the higher level. If the new node is less than the median key, insert it in the left-hand new node; otherwise into the right.
4. The promoted node is inserted into the parent node in order, splitting the parent if the parent is already full.
5. If the process of promotion reaches the root node and the root is full, then splitting and promotion occurs and the height of the tree increases by 1.

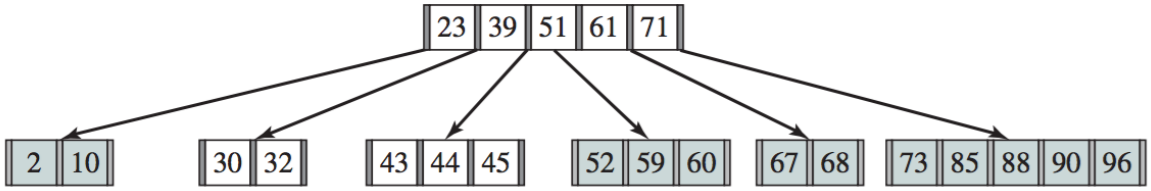
Some examples of how the insertion algorithm runs are shown below:



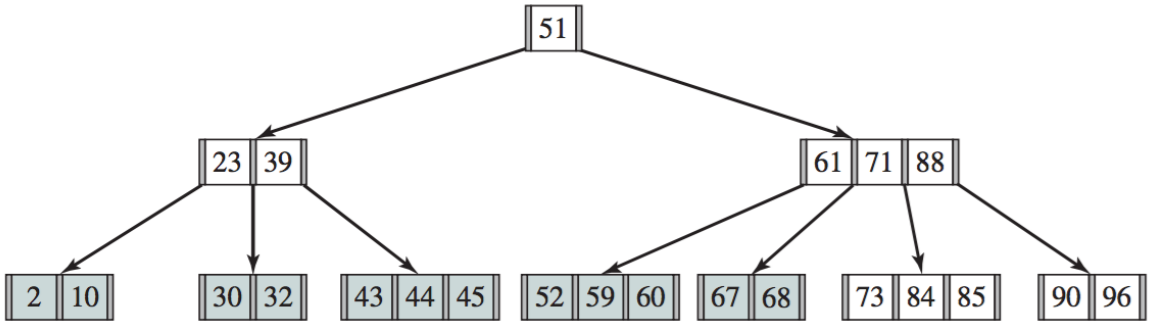
(a) B-tree of minimum degree $d = 3$.



(b) Key = 90 inserted. This is a simple insertion into a node.



(c) Key = 45 inserted. This requires splitting a node into two parts and promoting one key to the root node.



(d) Key = 84 inserted. This requires splitting a node into two parts and promoting one key to the root node. This then requires the root node to be split and a new root created.

Insertion of nodes 90, 45, and 84 into a B-Tree [Sta18].

23 — File Allocation Methods

File Allocation Methods

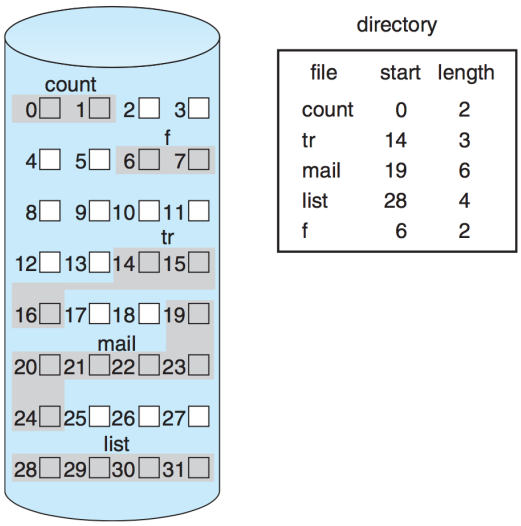
Much like memory allocation, there is a need to choose a strategy for how to allocate disk space. There are three major ways that we could allocate the disk space to files: contiguous, linked, and indexed; each has its advantages and disadvantages.

Contiguous Allocation

The contiguous allocation strategy means that a file occupies a set of contiguous blocks on disk. So a file is allocated, starting at block b and is n blocks in size, the file takes up blocks $b, b+1, b+2, ..., b+(n-1)$. This is advantageous, because if we want to access block b on disk, accessing $b+1$ requires no head movement, so seek time is nonexistent to minimal (if we need to move to another cylinder).

All that we need to maintain are the values of b and n: the start location and length of the file. Both sequential and direct access are very easy: the first block of a file is at b. To access a block i at some offset into the file, it is simply at the base address b plus i. Checking if the access is valid is also an easy operation: if $i < n$ then it is valid.

See the diagram below to get an example of contiguous allocation:



Contiguous allocation of disk space [SGG13].

This takes us back to a problem we have seen once before: the memory allocation problem. If we need a memory block of size N: (1) can we find a contiguous block of N or greater to meet that allocation? and (2) if there is more than one block, which one do we choose? As before, we suffer the problem of external fragmentation, plus a bit of internal fragmentation in the last block of the file.

There was also talk on the subject of memory allocation of compaction. This was moving memory allocations

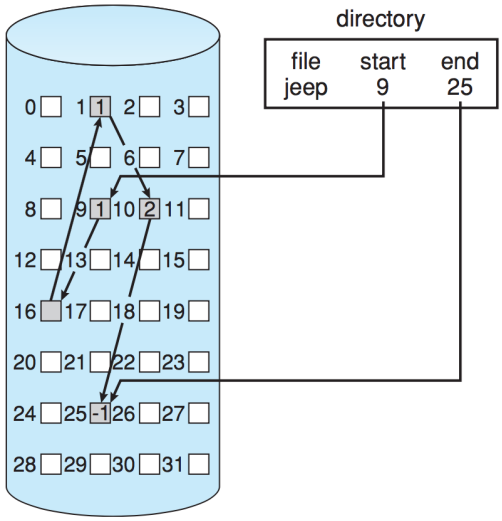
around in memory to create some larger free spaces that can be allocated. But for languages like C, it was not realistic, because of the impossibility of updating all pointers (unlike Java where we can update all references). We can do compaction on disk, but it takes a very, very long time and is computationally expensive. Doing compaction can be done when the system has nothing else to do (idle priority operation) or on some schedule when it would be minimally disruptive (e.g., middle of the night... mind you... a lot of programmers do most of their work in the middle of the night...).

Another problem, of course, with contiguous allocation is that old clairvoyance issue: how much space is a file going to take? If it is just a copy-paste operation, the copy is the same size as the original, so there is no problem. When a user opens a new document, how does the OS know how big the file is going to be? Is this a 3 page essay or a 300 page thesis? If we allocate too little space, we may be able to tack on space at the end, or that block may be allocated, forcing us to move the file and reallocate it. If the value we choose is too large, then significant space will be wasted for small files (and many files tend to be relatively small) [SGG13].

Linked Allocation

Linked allocation is a solution to the problems of contiguous allocation: instead of a file being all in consecutive blocks, we maintain a linked list of the blocks, and the blocks themselves may be located anywhere on the disk. The directory listing just has a pointer to the first and last blocks (head and tail of the linked list).

If a new file is created, it will be created with size zero and the head and tail pointers are null. When a new block is needed, it can come from anywhere and will just be added to the linked list. Thus, compaction and relocation are not really an issue. Unfortunately, however, accessing block i of a file is no longer as simple as computing an offset from the first block; it requires following i pointers (a pain).

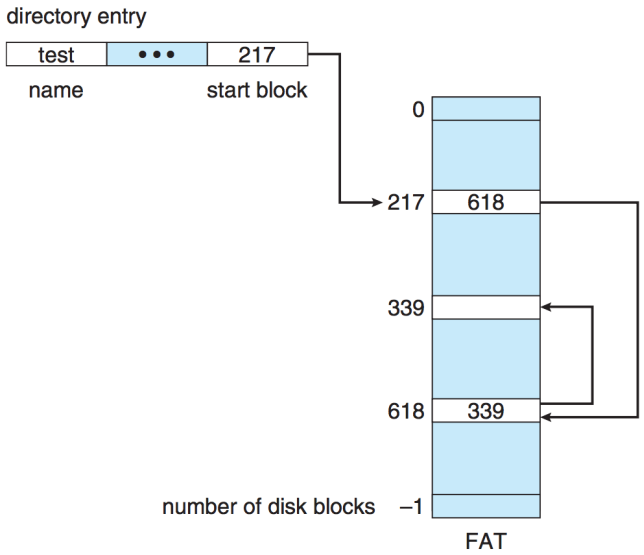


Linked allocation of disk space [SGG13].

A possible solution to the problem of following so many pointers (and the overhead of maintaining so many) is to group up the blocks into *clusters*, where a cluster is comprised of, say, four blocks. Then we waste less memory maintaining pointers and it improves disk accesses because there is less seeking back and forth to various disk locations [SGG13].

One variation of linked allocation is the File Allocation Table (FAT), which was used by the MS-DOS operating system (FAT16 and FAT32 were used in Windows before NTFS came in with NT/2000/XP and their descendants). The FAT32 file system hangs on these days as the file system of choice for formatting USB flash drives because it means the data will be readable in Windows, Mac OS X, and Linux. This is primarily because Microsoft has not made the NTFS file system standard public and Windows systems do not support too many other file systems, and Windows has such a big marketshare. So the FAT32 option is used, despite its known problems (and limitations, like 4 GB max file size), because it's the one system everyone can understand. Isn't the state of personal computing so... uplifting?

The FAT allocation scheme works like this: at the beginning of the disk, there is a table to maintain file allocation data (hence the name). The table has one entry for each block and is indexed by block number. It works like a linked list; the directory entry has the first block of the file and the table entry under that block number has the index of the next block. The chain continues until the last block where there is a special end-of-file value. An unused block has a table value of 0. Thus, to allocate a new block, find the first 0-valued entry, and replace the previous end-of-file value with the address of the new block [SGG13]. See the example below:

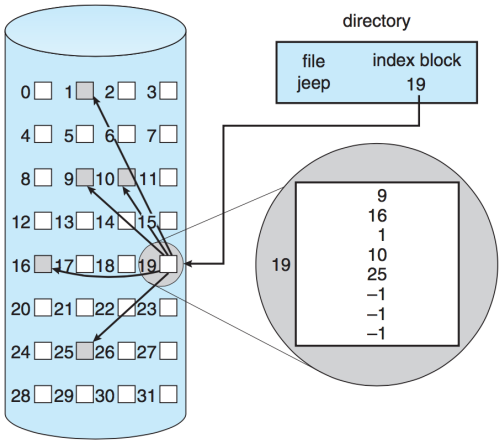


File Allocation Table (FAT) [SGG13].

The FAT itself should be cached in memory, otherwise the disk is going to have to seek back to it unbearably often.

Indexed Allocation

If we stuck to pure linked allocation, we still have the problem that accessing some part in the middle of the file is a pain because we have to follow and retrieve a lot of pointers to the different blocks. The idea of indexed allocation is to take all the pointers and put them into one location: an index block. So, the first block of the file contains a whole bunch of pointers. To get to block i, just go to index i of the index block and we can get the location of block i much more efficiently than we could in linked allocation. All pointers to blocks start as null, and when we add a new block, add its corresponding entry into the index block [SGG13]. See the diagram below:



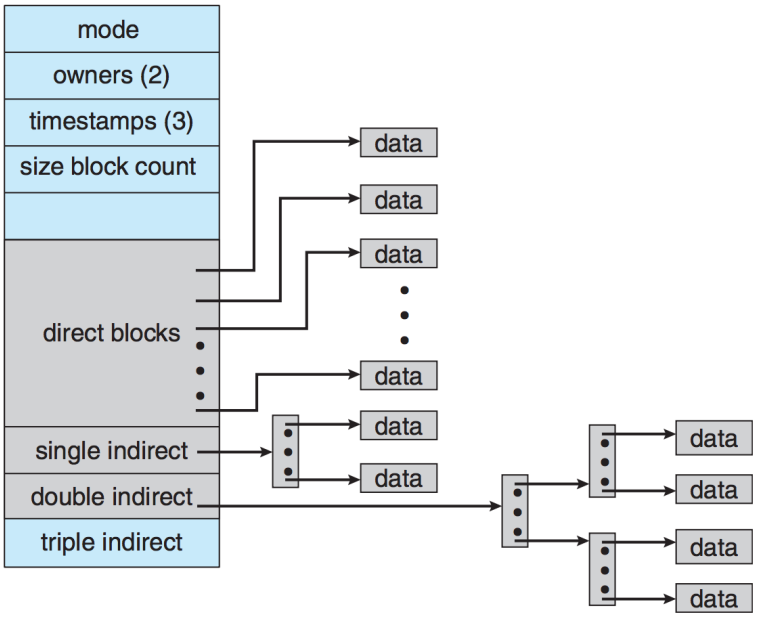
Indexed allocation of disk space [SGG13].

Like many of the other systems we have examined, there is a need to make a decision about the size of a block. If a file needs only 1-2 blocks, one whole block is allocated for the pointers which contains only 1-2 entries. That

suggests we want the index to be small, but what if we need more pointers than fit into one block? There are a few mechanisms for this [SGG13]:

1. **Linked Scheme**: An index block is a disk block, and we can link together several index blocks. The last entry in the index block is either null or a pointer to the next index block.
2. **Multilevel Index**: A variant of the linked scheme that has multiple levels. The first level block points to the second level block; the second level block points to the actual file data. This can go to as many levels are necessary based on the maximum file size. If a block is 4 KB, we can have 1024 4-byte pointers, so two levels would allow a maximum file size of up to 4 GB.
3. **Combined Scheme**: The all-of-the-above option. This is used in UNIX. Keep the first 15 pointers of the index block in the inode structure; 12 of them point directly to file data. The next three pointers refer to indirect blocks. The 13th is an index block containing the addresses of blocks with data. The 14th points to a double indirect block (addresses of blocks containing addresses of blocks). The 15th points to a triple indirect block[16].

With that out of the way, we can finally show a visual representation of an inode.



The UNIX inode. Triple indirection is left to the reader's imagination [SGG13].

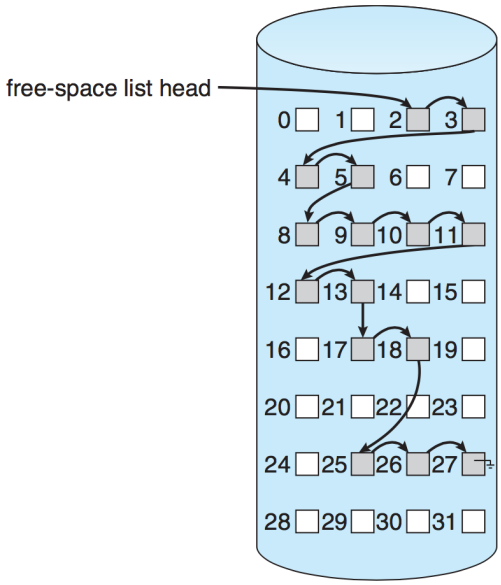
Other Disk Issues

As with memory, the system will keep track of the free space available.

Bit vectors work just the way we would expect: create a structure in memory where a bit represents a block. If it is free, the bit is 1; allocated is 0. So a bit vector looks like `0100110111011111000000...`. The problem with this strategy is, of course, that the bigger the disk, the more overhead is needed to store this bit vector.

The next rather obvious approach is a linked list: the head points to the first free space block, and we can traverse the list to get to the next free block. See the diagram below:

[16] Yo dawg, we heard you like index blocks...



Linked free-space list on disk [SGG13].

Grouping makes this slightly more interesting: the first free block will be used to store the addresses of n free blocks, the first $n-1$ of which are actually free. The last block contains a block with another set of n free blocks. So if we need a large number of free blocks, we can find them quickly, instead of walking through the list one block at a time, which takes a lot of disk accesses. Counting is a slight improvement on this where we also store a number k of free contiguous blocks after each address. So, if block 27 is followed by three consecutive free blocks, instead of having 27, 28, 29, 30 as entries, it will show (27, 4). The entries may be stored in a balanced tree for efficient operations [SGG13].

The efficient use of a disk depends on the disk-allocation and directory algorithms. UNIX inodes, for example, are preallocated, so even a disk containing no files has some of its space taken up by the inodes. Preallocation of inodes and distribution of them improves the file system performance, because the UNIX allocation and free-space algorithms keep the file data near the inode, where possible, to reduce seek time [SGG13].

Consistency Checking and Journalling

Unfortunately, an error, crash, or power failure or something similar may result in a loss of data or inconsistent data in the file system. The directory structures, pointers, inodes, FCBs, et cetera are all data structures and if they become corrupted it may lead to serious problems.

We could check for inconsistent data periodically (e.g., on system boot up) and many operating systems do so. This is, of course, an operation that will consume a very large amount of time while the whole disk is scanned. The UNIX tool for this is `fsck` (... not exactly something you want to say out loud) and the Windows tool is `chkdsk` (check disk). These tools will look for inconsistent states (e.g., a file that claims to be 12 blocks but the linked list contains only 5) and will attempt to repair the data structures. Its level of success depends on the nature of the problem and the implementation of the file system.

Obviously we would like to prevent the problem, if we can. Recall from much earlier the concept of atomic operations: an operation should either succeed completely, or not take place at all. Your first experience with such a structure may have been in version control, when using subversion: either a commit takes place in its entirety or it is as if it never happened. This approach is used in the Windows NTFS system as well as Mac OS HFS+ (if journalling is enabled).

Though we might be familiar, at this point, with the concept of the *transaction*, now let us take a moment to see how it actually works. The simple explanation is: before making any changes, make a list of all the things we plan to do. Then do the things written down. Then we consider the transaction complete.

All metadata changes are written sequentially to a log file; once the changes are written to the log, control may return to the program that requested the operation. Meanwhile, the log entries are actually carried out. As changes

are made, a pointer is updated to indicate which of the log entries have really happened and which have not. When an entire transaction is completed, it is removed from the log file. If the system crashes, the log file will contain zero or more transactions. If there are zero there is no problem: nothing was in progress at the time of the crash. If there are some, then the transactions were not completed and the operations should still be carried out. If a transaction was aborted (not committed), we walk backwards through the log entries to undo any completed operations and go back to the state before the start of the transaction [SGG13].

Even though a particular write may not have taken place because of a crash, resulting in some data loss, at least the system will always remain in a consistent state. As a side benefit, we can sometimes re-order the writes to get better performance (e.g., schedule them in such a way that we get better disk utilization).

The approach in the Solaris ZFS approach is similar, but not identical. Blocks are never overwritten with new data. Instead, a transaction writes all data and metadata to new blocks. Only when the transaction is complete, any references to the old blocks are replaced with the location of the new blocks. Then the old pointers and blocks can be cleaned up (reused or disposed of).

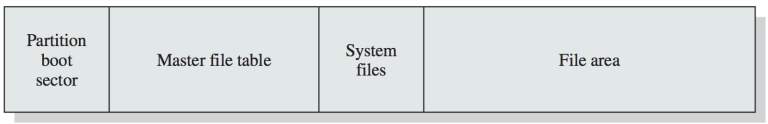
Example: NTFS (Windows File System)

Though UNIX and similar systems have often been a focus of the examples, in this case, we will instead examine NTFS, the default file system for Windows since Windows NT and used in 2000, XP, Vista, 7, 8... NTFS supports large disks and large files, and uses journalling.

NTFS uses several different storage levels [Sta18]:

1. **Sector**: Smallest physical storage unit on disk (usually 512 bytes).
2. **Cluster**: One or more contiguous sectors (grouped in a power of 2).
3. **Volume**: A logical partition on disk, consisting of one or more clusters.

The cluster is the fundamental unit of allocation of NTFS. This allows the file system to be independent of the size of physical sectors on the disk (which is convenient). A volume contains file system information, a collection of files, and free space. The logical volume may be some of a physical disk, all of one, or spread across multiple physical disks [Sta18]. A volume is laid out as follows:



Standard layout of an NTFS volume [Sta18].

The Master File Table (MFT) contains information about all the files and folders. Following the that, a block is allocated to system files that contain some important system information [Sta18]:

1. **MFT2**: Mirror of the first few rows of the MFT (in case the original is damaged).
2. **Log File**: The journalling transaction log.
3. **Cluster Bitmap**: Bitmap showing which of the clusters are in use.
4. **Attribute Definition Table**: Attribute types supported on this volume.

The table below shows the NTFS attributes for files and directories.

Attribute Type	Description
Standard information	Includes access attributes (read-only, read/write, etc.); time stamps, including when the file was created or last modified; and how many directories point to the file (link count)
Attribute list	A list of attributes that make up the file and the file reference of the MFT file record in which each attribute is located. Used when all attributes do not fit into a single MFT file record
File name	A file or directory must have one or more names.
Security descriptor	Specifies who owns the file and who can access it
Data	The contents of the file. A file has one default unnamed data attribute and may have one or more named data attributes.
Index root	Used to implement folders
Index allocation	Used to implement folders
Volume information	Includes volume-related information, such as the version and name of the volume
Bitmap	Provides a map representing records in use on the MFT or folder

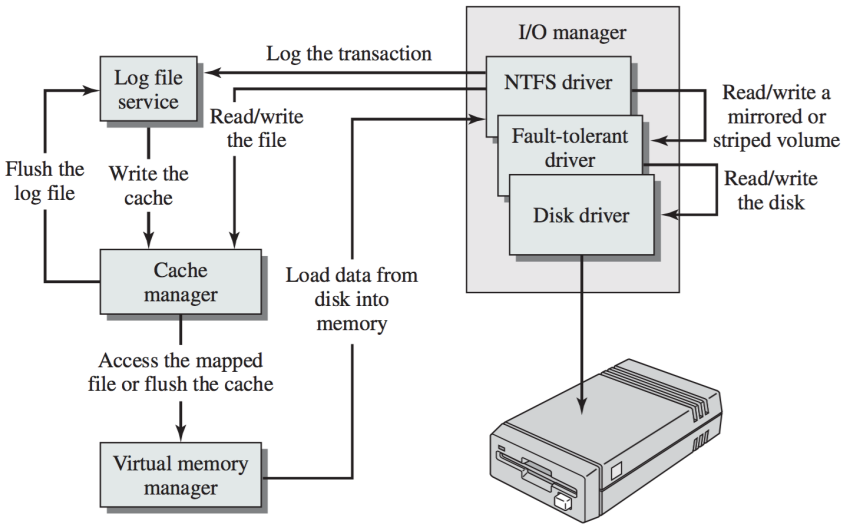
NTFS File and Directory Attribute Types (green blocks are mandatory) [Sta18].

NTFS uses journalling to ensure that the file system will be in a consistent state at all times, even after a crash or restart. There is a service responsible for maintaining a log file that will be used to recover in the event that things go wrong. Note that the goal of recovery is to make sure the system-maintained metadata is in a consistent state; user data can still get lost. This was a decision on the part of Microsoft to make the recovery operations significantly simpler and faster.

The actual implementation of journalling works as follows [RSI11]:

1. Record the change(s) in the log file in the cache.
2. Modify the volume in the cache.
3. The cache manager flushes the log file to disk.
4. Only after the log file is flushed to disk, the cache manager flushes the volume changes.

Or, to represent it visually (although, for some strange reason, with a floppy disk drive?!):



Overview of the NTFS recoverability elements [Sta18].

Backups

One final note about hard disk drives. We often think of them as places for permanent storage of data, but hard drives can and do fail. So please, take backups of important data.

24 — Reliability: RAID

Failure is Always an Option

The previous topic discussed the idea of using transactions in case there is a failure in the system to ensure that data is always in a consistent state, but we closed on somewhat of a darker note that said, well, hard drives die. And they do. So what do we do? Backups are a part of the solution but they're not the only way to ensure that we don't lose data.

Backups are not necessarily always up-to-date; if you have daily backups you could lose the full day's work; hourly backups might lose an hour, etc. And more than that, if the disk dies, you will need to replace it (maybe that's not instant) and then it takes nontrivial time to copy the data from the backup storage location to the new disks and get the system going again. Finally, there's a saying that you really only have backups if you've tested that you can restore from those backups successfully.

So, as it stands, the untimely death of a hard drive means potentially the loss of some data (decreased reliability), and until the situation is addressed, the system is down, or at least at reduced performance (decreased availability). What if I told you there exists a solution to this where you not only get higher reliability and availability, but also maybe better performance? That's not just wishful thinking, it's possible... but it does cost money. Maybe not as much as we might think, though.

To be clear, though, the solution I am describing is not magic: if all your hard drives die at once then there will still be data loss. Similarly, if your data centre catches on fire and burns to the ground, it will decrease availability. But for common scenarios, we can do a lot to reduce the likelihood and impact of a problem.

RAID

The solution is called RAID: Redundant Array of Independent Disks (previously, the I stood for "Inexpensive"). The idea, in short, is to have multiple different independent disks that work together but appear to the user as if they are one drive/volume. The RAID array may be managed in software or in hardware, but it does not matter for our discussion of the topic today.

The idea behind the former use of the word "inexpensive" is that if we combine more than one relatively-inexpensive disk, we can get higher reliability than we would if we spent more money on higher-quality drives. But drives have become a lot less expensive in general so the independent part has taken over the letter I.

How much redundancy we need is highly dependent on the data that we need to store. As you may know, I lost a hard drive in about 2014, but I wasn't upset about it because it mostly contained my Steam game library and it was easy to just download and reinstall all the games again (even if I still never played most of them!). Not having redundancy there is okay because, to some extent, Valve corporation is the source of my backups. Losing this data is inconvenient, slightly, but not a tragedy. But suppose I have a database backing up my web application where I sell services – downtime here is going to be a problem.

For any device (hard drive, SSD, or even CPU) we can talk about the *mean time to failure* of the device. The mean time to failure is always an estimate: if this is the first time that ABC corporation has produced this particular device, they don't really know how long it will last. They may be able to compare to the previous year's version and use some internal test results to create their estimate, but it's hard to know if the device will last ten years

before ten years have elapsed. Also, the estimates assume some usage patterns that may not be how you actually use it. Graphics card manufacturers may assume that you want to use the RTX 3080 to play Cyberpunk 2077 and base the lifetime estimates on you playing this game x hours per day on average... but if you actually overclock it and use it to mine cryptocurrency (please don't), then its lifetime will be shorter than expected because using it at 110% 24 hours a day will cause a failure sooner than predicted.

How bad is the problem? Let's say you bought a disk that says its mean time to failure is 100 000 hours (11.4 years) – you might think that this is perfectly fine, even if it lives only half its expected life it's still more than five years and you'll replace it by then – but if you are running a data centre and you have 100 of these, the mean time to failure of one of these disks is really only 100 000 / 100 = 1 000 hours or 41.66 days [SGG13]. The textbook example is a little bit silly – hard drive deaths are not evenly distributed but the example makes it seem like one of them dies every six weeks. It's more likely that everything is fine for a while and then drives start dying in quick succession.

No Free Lunches. So if we want to have improved reliability, it's obvious that this is going to have a cost. The cost is in the redundancy; having more than one copy of the data and those copies are stored on different physical storage media. The cost can be thought of in two ways: either we have to spend more money for more disks, or, if we have a fixed budget available, we are trading some of the total capacity for redundancy. Both scenarios are easy to visualize: if we want to have redundancy for a 2 TB hard drive then you could buy another 2 TB hard drive for redundancy, or use the same money to have two 1 TB hard drives. Either way, we have less space available than the nominal sum of the drives purchased, but that's inevitable.

Given that we have two drives of the same size, well then, the obvious approach is what is called *mirroring*, which is to say that every write that takes place is duplicated to both disks. If one of the disks dies, then the data is on the other disk and it's not lost. Excellent! Of course, having lost one of the disks, it's important to replace the dead one so that the reliability is restored. How quickly?

Quicker is obviously better, but if you want a quick calculation: if the mean time to failure is 100 000 hours and the mean time to repair is 10 hours, the mean time to data loss is $(100\ 000)^2/(2 \times 10) = 500 \times 10^6$ hours (about 57 000 years) [SGG13]. That's certainly a long time, but is it realistic? What we're saying effectively, is, how long will it take before disk 1 fails but cannot be replaced in time before the failure of disk 2. Assuming we attend to the problem quickly, you can imagine this is rare.

... Or is it? What we're assuming here is that disk failures are independent. We are not expecting that disks will actually survive 57 000 years – let's not kid ourselves – but only that hypothetically, if a disk dies and we replace it fast enough, and maintain this watch forever, then we expect the average time to lose data is 57 000 years. But that's not really how likely it is to lose data, because in reality these things may not be independent at all. Leaving aside things like the data centre catching on fire, if we buy two disks from the same manufacturer from the same manufacturing run, it's entirely possible that both of these disks will live about the same amount of time. So it makes it more likely their demises are closer together and not independently randomly distributed. And if we do throw in a new disk to the system and have to copy all the data from the old disk, that is in fact a heavy load on the surviving disk and if it's already fragile, that might be the last thing that pushes the old disk over the edge. Oops.

The simple version of mirroring we've considered does have additional costs and questions: every write needs to be done twice and what happens if one such write succeeds and one fails? It's a good question, but let's save that for the moment, because there are more possible arrangements than just mirroring.

Level, Please

RAID is describes as having different "levels"; each level has a different configuration of the drives. Some of them have more redundancy at the cost of space (or money), others have less. There's also potentially some performance benefits to choosing one configuration or another. The different levels should not be viewed as higher equals better; it's important to choose the right level based on your needs and budget.

There are seven basic levels that have broad acceptance in industry (RAID 0 through 6) but there are also some combinations of these that are relevant [Sta18]. Some researchers or companies may suggest different levels or numbers, but we'll restrict the discussion here to the commonly-accepted ones that we find in textbooks and other discussions. Even amongst the levels that exist, not all of them are actually commonly used.

No Thanks. It wouldn't be right to exclude the do-nothing option here, which is to say, no RAID configuration at all. This is sometimes called JBOD – Just a Bunch Of Disks – and it's simply choosing not to configure the disks you have in a RAID array and simply treat them as individual disks. No redundancy and no reliability increase, but it's entirely manual how you want to handle this.

This isn't entirely insane: you could say that using, say, an external hard drive for backups is like this. Or if you have two internal SSDs, copying important files from one to another to guard against data loss. But okay, let's get on with the actual RAID levels, shall we?

RAID 0: What Redundancy? The first option to consider is RAID 0: disk striping. This just splits the data and files across multiple disks, so Disk 1 will have block 0, Disk 2 block 2, Disk 3 block 3...

The major advantage of the RAID 0 configuration is that it is faster! Writes can be done in parallel: e.g., writing a file that has many blocks to a RAID 0 array of n disks can be done at a speed of n blocks at a time, which increases the parallelism by a factor of n. The same is true for reading, it can also be sped up by a factor of n. So throughput for reads and writes goes way up.

There is no redundancy at all in this system. It is faster to do reads and writes, but at the cost of reliability: if one drive dies, all the data is lost! That might seem weird, because topic we're considering here is reliability; this sounds like the opposite of what we want.

There are scenarios where you would be fine with the risks of RAID 0. Suppose you have a transit app that gives you the bus, subway, and train schedules. Instead of retrieving the schedules from the database on every request, you could keep them in a cache. If memory is big enough, great, keep them in memory. But if it's not and you need them to be on disk, you might want to make a RAID 0 array of two smaller disks so that you get better speed for reading and writing things in this cache. In this scenario, we can live with the fact that any one disk's untimely death takes the whole thing down, because the cache is just a more-easily accessible copy of the original data and it's trivial to regenerate that at any time.

Another reason we might want to use a RAID 0 configuration is because we want to combine it with another option. We haven't discussed those yet, so we can return to this idea of combining things later.

RAID 1: Mirror, Mirror. RAID 1 is mirroring, rather like we discussed in the introduction to the section. The data that appears on each disk is replicated exactly to the other so that they are identical. That is a lot easier to explain than the upcoming RAID levels that involve parity schemes. This works with an even number of disks, as half of the disks are mirrors of the other half.

Read speed is improved, since any piece of data could be found on one of two disks, so we can get the data from the less-busy drive. If it's a larger request, we can read from two disks in parallel and get the data faster. If we have many more reads than writes in our application, this is potentially a big performance improvement alongside more reliability.

In terms of write speed, it's realistically no worse than just writing directly to one disk; there's no write penalty. If one of the disks is busier than the other, the write is not truly complete until it has been written to both. This might require waiting, which would be slower than just writing to one disk. The alternative approach is to write it to one disk and asynchronously carry out the replication to the other.

Replication is fine, but it comes with some additional headache of what happens if that replication fails. Or, alternatively, if two different in-progress transactions failed We have tools to deal with that scenario: transactions, recovery, or even NVRAM (non-volatile RAM) to ensure that the queued operations proceed even in the event of a power failure [SGG13].

Should one the disks be unavailable, we can just get whatever data we need from the disk that survives and the read speed may be reduced but is no worse than when there was only one drive in the first place.

If we have two disks, it makes sense that our usable capacity is half of the purchased capacity, because each disk is an exact copy of the other. But if we have four disks, two disks are a copy of the other two. Eight disks means four and four. That's excellent for reliability, but if the scenario we want to guard against is the failure of one drive, why are we using four backups? Better safe than sorry, but at some point it is overkill. What if we didn't need

quite so many disks to guard against the failure of one of them?

RAID 2: Bit Parity RAID 2 looks a lot like how memory does error correction in ECC RAM; memory uses parity bits to detect if there is an error. Remember how parity bits work from previous courses? Does Hamming code sound familiar?

If you just want to detect single-bit errors, each byte has a parity bit that is 0 if there's an even number of 1s in the byte and the parity bit is 1 if there's an odd number of bits. In the event of data corruption of some sort, we can detect the error but it's hard to know which bit is flipped. If we have a two or more bit scheme, we can correct single-bit errors and detect two-bit errors. And the trend continues: more parity bits equals more ability to detect and possibly correct them. More overhead, but also more ability to correct data.

ECC can be used in disk arrays. One such example is that if you have the first bit of each byte stored on disk 1, the second bit on disk 2, and so on such that the 8th bit of each byte is stored on disk 8 and you have yet another disk that stores the parity bits; in the event of a loss of one of the disks, for each byte you have seven of the eight bits and you can work out from the parity bit whether the missing bit should be a zero or one [SGG13]. You don't actually need nine drives, though it does help make the organization a little clearer.

Based on this scheme, a read or a write will necessarily engage all of the disks in the array so it can read data quickly and write data quickly, but it's also painful to split every read or write up to the bit level. It might work better for larger files since we could read multiple blocks at once; for small files it requires reading eight blocks where one would normally do. So random access performance is not great.

This does have slightly lower overhead than RAID 1 as you don't have 50% of the disks given over to replication, but it's still pretty high overhead. If you have four disks of original data, you might need three parity disks for it. It's hard to come up with a use case for RAID 2: it makes sense where lots of disk errors occur and data elements are frequently corrupted, but exiting hard drives are reasonably reliable so there's not much benefit to using this scheme [Sta18].

RAID 3: Byte Parity RAID 3 looks a lot like RAID 2, but instead of having each bit of a byte spread across different disks, bytes are spread across the disks and parity is slightly different: a parity bit is computer for the set of bits at the same position on all the data disks.

An example from [Sta18] about how to calculate the parity: if we have drives called D0 through D4 and D4 is the parity disk, the parity for bit i is calculated as $D4(i) = D3(i) \oplus D2(i) \oplus D1(i) \oplus D0(i)$ (where $\oplus$ is the XOR operator). If drive D1 has died, then you can recover the data by calculating $D1(i) = D4(i) \oplus D3(i) \oplus D2(i) \oplus D0(i)$.

Like RAID 2, every read or write needs to be performed synchronously across all disks. That can help with speed since the data is being access across multiple devices simultaneously, but we might end up doing a lot of extra reads to read a small file.

One advantage is that only one extra disk is required no matter the number of drives in the system and it stores all the parity information. Less overhead, and a lot less – one disk for redundancy out of eight rather than four. Bit there is the computational overhead of doing the parity math, although this is relatively small in comparison to the times for disk reads and writes. And maybe can be offloaded to the hardware.

Another thing that's nice is that if a disk fails, all data is still available if somewhat slower. Any missing data can be compensated for by calculation of what it should be using the parity information. Writes need some additional bookkeeping so that when the system is restored to full functionality, everything is where it should be. However, this is nice to have since it means the system can continue executing in a reduced capacity until the problem is solved, rather than going down [Sta18].

RAID 4: Block Parity RAID 4 looks a lot like what we've seen already, but instead of having bit- or byte-level parity we have block level parity. There is block-level striping as in RAID 0 and the parity blocks are stored on a dedicated parity disk. As with other schemes using parity, data that is lost due to the failure of a disk can be restored by doing the calculations using the data on the other disks.

Unlike RAID 2 or 3, each disk can independently be accessing a different block at the same time.

There is some penalty to doing an update, if the update is small. If a whole new block is being written, writing the parity block for it is straightforward. If, however, the write is small, the existing information needs to be read and modified, so every write may result in reading the original block and parity information, making the modifications, writing it and the parity data out for a total of two reads and two writes [Sta18].

One potential downside to this scheme is that the disk containing the parity information is modified a lot. That means this drive may be worn out sooner than the others in the system (which is, admittedly, more of a problem for SSDs that can get their writes exhausted). But this could become a bottleneck, also [Sta18].

RAID 5: Distributed Block parity RAID 5 looks like RAID 4, except instead of one disk dedicated solely to the parity information, the parity information is distributed across all disks. That is, each disk has some data, and some parity information of other disks.

Parity information could be stored in a round-robin fashion. Obviously, a parity block cannot be stored on the same disk where the original information is, because a loss of this disk would cause an unrecoverable loss of data.

RAID 5 is, in my experience, and according to the textbook [SGG13], the most common parity RAID system. It provides a lot of what we've been looking for – reliability to avoid data loss if any one drive should die, without doubling the number of drives needed.

RAID 6 RAID 6 just looks like RAID 5 but has extra information that allows the system to survive more disk failures. Rather than parity, some other error correcting codes are used such that we have, say, two bits of redundancy information for every four bits of data; in this case the system could survive two disk failures. But there is a substantial write penalty here since every write requires modification of two parity blocks; tests show that RAID 6 has read performance about the same as RAID 5, but writes are about 30% slower [Sta18].

Combining Levels

There are a number of ways that RAID may be combined, such as RAID 01, 10, 50... These are best thought of as being something like 1 + 0 rather than ten, because what we're doing is combining both these things at two different levels.

Consider RAID 0 + 1: we have two sets of disks combined in RAID 0, and then on top of that we build in mirroring so we have a RAID 1 of the combined disks. So if we have four disks in total, we have a RAID 0 formed of disks 1 and 2, another RAID 0 formed of disks 3 and 4, and then build a RAID 1 array of the 1+2 and 3+4 disks.

In principle you can combine any strategies any number of times: every time a group of disks is combined using a particular RAID level, you can treat that as a single disk input to the next level.

Choosing the Right Level

How do we choose the right RAID level for our system? It comes down to some important factors that we've already discussed:

- How critical is it that data is not lost?
- What is the budget?
- How important is performance?
- Do we need to be able to carry on in the event a disk dies?
- Are rebuild times important?

There are really no one-size-fits-all answers, though certainly we can see that some levels of RAID (2, 3, 4, and to a lesser extent, 0) are not very popular for obvious reasons. Every use case is different, so choose what makes sense and what you can afford.

25 — Reliability: Fail-Soft Operation

Task Failed Successfully

Except in the recent discussion of how hard drives may die and in a previous course's discussion of the Byzantine Generals Problem, we usually go through life assuming that computer systems will work as they are designed. That does not rule out the possibility of a design problem or implementation bug, of course. But in many situations, we actually have surprisingly low expectations of computer systems. Things crash, we reboot them. Something going wrong? Retry it.

Downtime is sometimes okay to address a problem, whether it's by fixing the system or replacing it. If my laptop should go for an unplanned swim in a lake, it's out of action, probably permanently. I can just buy a new laptop. I'm not happy, to be sure, but my laptop makes no promises of uptime. In a less dramatic scenario, I might install an OS update and it will be down for some period of time. The OS gives me an estimate of how long it's going to be, but that's a guess and not a promise. But maybe I need a system where that doesn't happen, even if some hardware or software update is required.

Reliability is an important element for real-time systems. Downtime may be intolerable in the case of life- or safety-critical systems, or just very expensive in the sense of needing to meet a service level agreement and having to pay contract penalties to customers if the agreement is not maintained.

If I am travelling to work via a car and get a flat tire, the car is out of action while the tire is changed. After that, I can go on: that's a repair with downtime. But some cars have the concept of run-flat tires, which allow you to travel for a limited distance at reduced speed if a tire is flat. This is to say, the system remains functional at reduced capacity until the problem can be addressed.

What we'd like to have for a real-time system is generally two distinct things: resiliency and fail-soft operation. A system that is more resilient is capable of carrying on in the event of a failure, even if not at full capacity; and fail-soft operation says that if things do indeed fail then the system will preserve as much capability as it can or terminate gracefully if it must [Sta18].

In the previous topic, we discussed what we can do to mitigate the impact of hard drive failure. That established the fact that it's not realistic to try to get ironclad devices that will last a hundred years, but instead through redundancy. If you want the system to be capable of surviving the death of a CPU, it needs to have more than one CPU in the system. Right?

Resiliency

In the words of Rocky Balboa, "It's not about how hard you hit. It's about how hard you can get hit and keep moving forward. How much you can take and keep moving forward.". When designing a system for resiliency, the question to consider is how much resiliency do you really need?

If you are designing something to not have downtime because the company loses some money when the system is down, your considerations are different to a situation where you need to avoid downtime because lives are on the line. Remember also that there are some scenarios that you cannot and should not handle because they're either too outlandish or just plain apocalyptic. If Russia decides it wishes to launch its nuclear missiles[17] and your data

[17] August 29, 1997 2:14 AM Eastern Time... `https://www.youtube.com/watch?v=4DQsG3TKQ0I`

centre is in one of the target zones, does it matter if your app is going to have some downtime right now? I think there are bigger problems to worry about.

Nevertheless, this is an engineering design tradeoff between cost and the amount of resiliency you want to have. Up to a certain point, more money will get more redundancy and more resiliency. Remember the earlier example about how a lack of spare crew capacity can result in delaying of your flight from Calgary to Vancouver? If a certain airline were more open to increasing the cost, they could have more capacity. That would make the system more resilient: a snowstorm in Nova Scotia wouldn't ruin the travel plans of unrelated people several thousand kilometres away. On the other hand, at some point it becomes wasteful: the airline should not be paying excessive amounts of people to hang around doing nothing just in case.

Let's imagine that something has actually gone wrong, for whatever reason. There are a few different ways to respond or handle this situation. For the sake of the example, we'll consider primarily hardware failure, but software failures matter too.

Can We Fix It? Yes We Can! The best option for resiliency is if the system can correct the problem and carry on! That may not be sensible if the scenario is an unrecoverable hardware failure: if the magic smoke has come out of the hard drive, no amount of rebooting it will make it go back in again. However, if the failure would be solved by re-initializing the device, why not?

Fixing some things may be easier if it's a software problem. Remember that we talked in a previous course about the ideas around deadlock recovery. Deadlock is a failure, and perhaps the system can recover from failure by pursuing one of those approaches, such as killing processes or rebooting the system.

During the process of recovery, though, the system is probably running at a lower capacity. So let's consider that...

Stiff Upper Lip. If the system cannot be restored to full capacity automatically, then the next best thing is if the system can continue as best it can, most likely at a reduced capacity.

Suppose the system has 4 CPU cores and one of them dies but the CPU was never used at more than 50% capacity even when all CPU cores were healthy. The system is worse off, yes, and its maximum capacity has been reduced, but in practice the system is still running at its original capacity. That's one reason for redundancy in the system, to be sure. That's the ideal case, but did the system really get designed with that much extra capacity?

The next possibility is that the system is running at reduced capacity until external forces come to repair the system. Reduced capacity in this sense may be that it takes longer to get answers, or less work can be done in the same amount of time.

In a real-time operating system, this loss of capacity might mean it's no longer possible to meet all deadlines. The system is considered *stable* if it will always meet the deadlines of its most critical tasks even if lower-priority tasks do not get completed [Sta18]. In this sense, it means that in the event that painful choices must be made, the most important things are prioritized. We'll consider some other scenarios for handling a problem, but then we'll return to focus on how to carry on at reduced capacity.

Shut It Down! Perhaps instead of throwing the emergency stop when things go wrong, it's possible to do an orderly shutdown. The traditional UNIX system, if it detects kernel data corruption, will write the contents of memory to disk for analysis and stop execution [Sta18]. If the problem encountered is something we cannot live with but also cannot fix, then this is a valid solution to making sure no additional damage has been done.

Stop. Hammertime. One option for failing is to just cease all execution or operation immediately. If the industrial machinery controller is in a bad state, maybe the best thing to do is actually to stop everything so the equipment stops moving. That may reduce the potential for damage to the materials being manufactured or risk to the people working in the facility. Downtime for the system is bad, but not as bad as the system killing someone.

Faults and Fault Tolerance

In the previous section, we just said vague things about something going wrong like hardware dying or something like that. But let's take a look at what is a fault and what does it mean to be fault tolerant.

A failure is the term we use when the response (outcome) deviates from the specification as a result of an error; and an error is a manifestation of a fault [HZMG20]. But what's a fault? The IEEE Standards Dictionary and some textbooks define a fault as an erroneous hardware or software state of some variety [Sta18]:

- **Permanent**: A fault that is always present after it occurs and can only be fixed by actually replacing or repairing the faulty component. If a GPU ceases to work due to overheating causing a failure of its internal components, that is now permanently broken and can maybe be repaired but probably has to be replaced, so it's a permanent fault. Software bugs are also permanent faults: the bug was always in the code, even if it wasn't triggered for some reason (e.g., the bug occurs only on the 29th of February) and accordingly must be fixed by a code change.
- **Intermittent**: A fault that recurs at random, unpredictable times. A faulty chip may produce the right answer most of the time but the wrong answer rarely and unpredictably. That's super frustrating because it's hard to track down and identify what's wrong.
- **Transient**: A fault that occurs once but does not recur. The standard example of this is cosmic radiation flipping a random bit from 0 to 1, but I don't think this is actually that great of an example because things like satellites (especially the ones outside the earth's magnetic field) have to contend with this as just a fact of life, making it more intermittent than transient. For terrestrial systems, maybe this is an appropriate example.

Fault Prevention. Although it's tempting to jump straight to the idea of fault tolerance, which is about how to carry on in the face of the problem, it's even better if we can prevent the problem from occurring in the first place. That leads us to fault avoidance, which is some combination of some of the following steps [HZMG20]:

1. Make sure you understand the physical environment and protect the system from physical damage through appropriate shielding, waterproofing, etc.;
2. Use hardware with the intended reliability and lifetime (don't cheap out on the components!);
3. Within software development...
   (a) Have a sufficiently detailed and rigorous specification;
   (b) Use an appropriate design and test methodology;
   (c) Use the right programming language (Rust?);
   (d) Use analysis tools.
4. Verify that components and tools work as they advertise!

Following appropriate design, implementation, and testing processes will only make faults less likely. It is also important to resist the pressure to drop or degrade these processes when user time pressure. Perfection is generally unachievable, so it's more likely we'll have to think about how to deal with faults.

Fault Tolerance. Some of the things we covered much earlier in the course, or in an earlier course, play a role in supporting the resilience of the system in the face of software faults [Sta18]:

- **Process Isolation**: The most obvious example of how this is relevant is that the OS prevents one process from accessing the memory of another or the memory of the operating system itself. This limits the potential damage if something goes wrong in one process (e.g., dereference a bogus pointer), it does not affect other processes or the operation of the OS.
- **Dual-Mode Operation**: The OS monopolizing access to hardware devices and other common resources will, again, reduce the potential damage of a thread, maliciously or otherwise, interfering with the execution of others by cancelling other tasks.
- **Preemptive, Priority-Based Scheduling**: As with the previous example, preemptive and priority-based scheduling mean that misbehaving processes or threads have limited impact on other threads that want to run. Without this, a rogue thread could monopolize CPU time and negatively impact others.

- **Checkpoints, Transactions, Rollback**: These were covered in the concurrency course and can be used to recover from failures, transient or otherwise.

As for hardware failure, we already learned some of the important elements about fault tolerance through discussing the various RAID levels. In that topic we covered the idea of having backups or mirrored/striped volumes which gave is information redundancy: there's more than one copy of the data so the loss of one copy does not result in the permanent loss of data.

Most likely you covered some ideas of information redundancy in other courses around communication. Checksums, parity bits, error correcting codes, etc. all allow for detecting and possibly correcting errors through the use of data redundancy.

There is also the idea of physical redundancy, which is what happens if we have, say, two CPU chips in the machine such that if one of them dies then the system can carry on with reduced capacity. But it also describes the idea of having more than one physical machine or instance of your application, so if one instance or machine goes down (even for maintenance) the other can continue on. Ideally this sort of physical redundancy extends to having the systems in more than one physical location so that your application can still be available even if the Amazon US East 1 data centre is currently offline.

There's a third kind of redundancy called temporal redundancy: repeating an operation when an error is detected [Sta18]. This is what happens in, for example, TCP network communication, where a receiver who notices a packet is missing or damaged can request that this packet be re-sent to make sure the data received is correct. You may also consider the kind of redundancy where you have more than one execution of the task to verify that the results agree.

The space shuttle had a combination of physical and temporal redundancy for important calculations. It had five computers, but normally only four of them were used in a majority vote system. If one of the systems fails, it is outvoted by the others; if two systems fail in different ways then they get different answers and the two correct systems win the vote; if two systems fail in the same way and there's a tie, the fifth computer is activated and it will compute the problem and then vote to break the tie [HZMG20].

The important thing is that the system should have no single point of failure. A system that has such a single point of failure is vulnerable to total failure if that one component is out of action. And while it's sometimes possible to fix this in the system design phase, the deployment matters too: if you have two systems with multiple CPUs and redundant disks and extra RAM, but they share a network connection, they're not really independent because of that shared component.

Now I Have Two Problems

If we have two distinct systems that need to work together, we've just unlocked a whole new tier of problems called distributed systems problems. We actually covered one of the possible issues in distributed systems in the past when we talked about the Byzantine Generals Problem. That was framed around finding traitors when trying to complete the mission of the Emperor, but there's lots of other things that we will have to consider. We should examine one in detail – clock synchronization.

What Time Is It? The idea of temporal redundancy introduces the problem of time and that it is not trivial to get two independent systems to agree on what time it is. Inevitably, all clocks except the universal reference clock are off by some amount; it's just a question of how much! This is not really an indication of anything wrong with the clock synchronization, but just a question of the laws of physics as we'll see.

It's easy to imagine scenarios where independent systems who don't agree on what time it is will misbehave. Suppose the primary system is supposed to take some action at exactly 08:00 and this action takes 0.5 seconds to complete and it should signal completion. If the action is not completed within 1 second, the backup system will assume that the primary has failed and it will perform the action. If their clocks disagree by 1 second, then the backup system will do the action again unnecessarily (or perhaps cause an error by doing so). Is that example a little bit silly? Yes, but if you've ever been in a video call with delay and it's really frustrating because you each keep talking over one another ("you go ahead", "no, after you..."), you have experienced the pain of what happens when we don't agree on what time it is.

Time zones also matter, but I'm going to recommend using UTC because it doesn't have weird things like Daylight Savings Time where you can get a jump in the time forward or back. And back is worse because it means a certain minute appears to happen more than once (defeating our normal expectation of linear time). That can really be a problem if an action that is supposed to be taken once per day is taken twice or skipped. UTC also avoids issues where the California system thinks it's Monday and the Singapore system thinks it's Tuesday. UTC might be weird, though, if your application is running on a Mars Rover; at that point you might as well use Stardates (Captain's Log, Stardate 46119.1...).

Clocks frequently use quartz for synchronization, but there is always drift and measurement error; a quartz clock will typically vary by about half a second per day, so the idea of systems being off by a full second is quite reasonable if System A is fast by 0.5s and System B is slow by 0.5s [HZMG20].

In a non-real time operating system, it's often okay to just change the clock to the correct time and just jump there. But for a real-time system, breaking the expectation of linear time can cause events to run again, so typically we do not wish to do that. The solutions are effectively a graduated slowdown or speedup (i.e., counting a second as a little longer or shorter until things are back in sync). But that presupposes that we know what time we are trying to adjust to, and how do we know that?

That's a more complicated question than we want to discuss here, but a possible solution is something like the Network Time Protocol – NTP – which allows querying a reference source. The protocol itself is designed to compensate for some forms of network delay, although not all. Effectively, it's difficult or impossible to get more than one system to agree on what time it is.

Ultimately, this means the design of communication and synchronization between the systems must account for the communication time but also that the clock times will not be identical, to the point where messages could appear to arrive before they're sent.

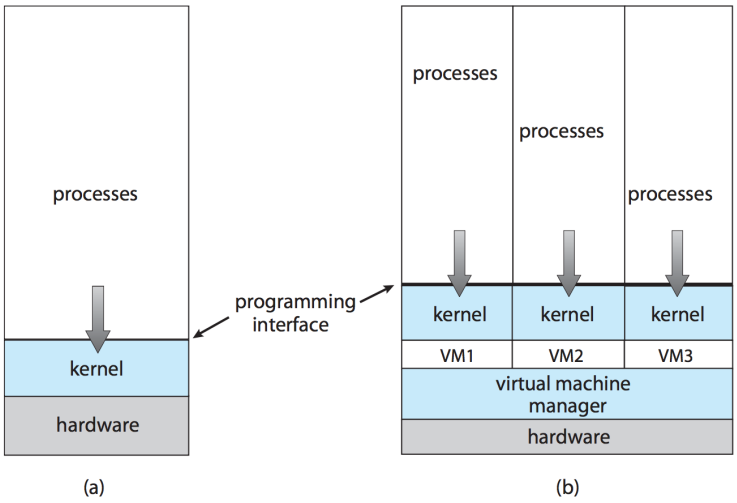
Distributed Systems Course. All of this is just a very simple overview of some of the issues that might arise when we have multiple systems for redundancy. This is a complicated subject and is a whole 4th year ECE technical elective that you could take if this is of interest to you.

26 — Virtualization and Containers

Virtualization

The word virtualization itself can refer to many different aspects of computing, but the part that we really want to talk about is "virtual machines". The goal is to abstract the hardware of a single computer into several different execution environments, where we might have different operating systems running, or multiple copies of the same operating system, depending on what is desired. From the perspective of the operating system, however, it does not usually know that it is executing on an abstraction of the hardware. Comparisons to the movie "The Matrix" are apt: "How would you know the difference between the dream world and the real world?"

At the lowest level, there is the *host*, the underlying hardware system. Above that is the *virtual machine manager* (VMM), sometimes called the *hypervisor* that creates an interface that looks like the host, but can have multiple copies. The *guests* interact with their own virtual copy of the host, and we can have multiple operating systems existing concurrently on the same physical machine [SGG13].



System models: (a) Nonvirtual machine and (b) Virtual machine [SGG13].

This is by no means the same thing as *emulation*. When we have virtualization, for example, both Windows and Linux can be running on the same x86_64 architecture as guests. In the case of the Android emulator running in an x86_64 machine, the code of the emulator is running on the x86_64 environment to simulate an Android hardware device that would have a completely different CPU. Thus, an Android app (which would not otherwise run on an Intel/AMD machine) runs in a simulation of a mobile environment. The emulation operation is incredibly slow, unfortunately, as anyone who has tried to use the Android emulator has found out. Sometimes, however, it does not much matter; if you are trying to play a classic game that ran under MS-DOS, emulating an 486 computer does not take much by way of resources and the modern computer can do so with ease.

Those who run operating systems that are not named "Microsoft Windows" are likely to have had the situation where one or more programs that are needed for some purpose (work, school, whatever) function only under

Windows. And as the versions of Windows have proliferated and evolved, older programs have sometimes stopped working, necessitating a past version (Windows XP will never, ever die...).

Another neat thing is the ability to suspend (pause) execution of a running virtual machine. This is like pausing a process: the current state is taken and saved, and can be restored at a later time. Also like a process, it can be moved around to another system and resume after that move has taken place, or cloned to get an identical copy somewhere else.

A third reason for virtualization is protection. The guests are isolated from the host and vice versa. If a virus infects one of the guests, the other guests are not affected. And with a virtual machine, it will be that much easier to delete and reinstall the guest or roll it back to an earlier state. This would be exactly the state that we have stored when suspending it... just take periodic copies of it.

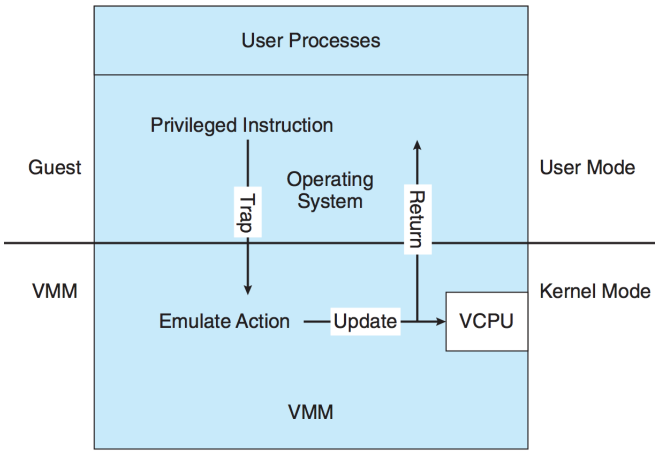
Finally, we often see consolidation in data centres: there are many servers running that could be sharing the same physical hardware. Instead of having a lot of lightly used physical systems, convert them to virtual and put them all in the same physical machine. This is often combined with a utility to convert a physical machine to a virtual one; all the system data and configuration and such is copied and turned into a snapshot and this snapshot is then started up inside the virtual machine.

Behind the Scenes

One of the key building blocks of virtualization is the *virtual CPU* (VCPU). It does not actually execute code; it is just the state of the CPU according to the guest machine. The VMM is responsible for maintaining the state of the VCPU. Much like the process control block, the VCPU is a data structure that is used to store the state when the guest is not running and the state is restored from the VCPU structure when the guest is scheduled to run [SGG13].

Recall from earlier the concept of user mode and kernel mode; application processes run in user mode and the kernel runs in kernel mode, having access to privileged instructions like I/O instructions and poking around with registers and other hardware. The guest operating system runs in user mode, but it will want to do some things that require kernel mode, so we will need to have virtual user mode and virtual kernel mode. Actions like an I/O request or interrupt that would normally lead to a switch from user to kernel mode needs to cause a transfer from virtual user mode to virtual kernel mode.

The first strategy for implementing this is called *trap-and-emulate*. If the guest attempts a privileged instruction, it will generate a trap (error) because it is in user mode. The VMM should then pick this up and executes, emulating (or simulating, if you prefer) the requested operation.



The trap-and-emulate virtualization implementation [SGG13].

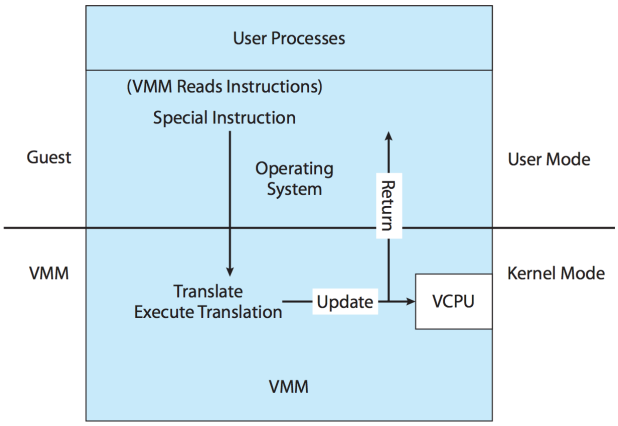
Non-privileged instructions just execute natively on the hardware, so they are about as fast as they would be if they were actually being executed outside a virtual machine. Unfortunately, with trap-and-emulate, privileged instructions have this extra overhead, causing the guest to run more slowly than it otherwise would. To get around this problem, hardware designers have come to the rescue again: some CPUs have more than just the two simple

modes (user/kernel) and keep track in hardware of virtual user and virtual kernel mode. That relieves the VMM from the responsibility of keeping track of it all in the VCPU [SGG13].

Sadly, some CPUs do not have clear definitions of privileged vs. non-privileged instructions, including the intel x86 architecture (and its descendant, the x86_64). Without editorializing too much, there were a lot of decisions that went into things like the x86 architecture, C programming language, design of Windows, et cetera, that make no sense if we look at them with what we know today, but they "seemed like a good idea at the time". The x86 architecture started back in the early 1970s and we can hardly fault the designers for not anticipating what was going to happen in computing technology 30-40 years later. As a Danish proverb of disputed origin says, "Making predictions is hard, especially about the future."

Returning from that small interruption, the x86 has an instruction `POPF` that illustrates the problem. It loads the flag register from the contents of the stack. If the CPU is in kernel mode, all flags are replaced from the stack; otherwise only some flags are replaced. No trap will be generated if `POPF` is executed in user mode, so the trap-and-emulate solution will not catch this and react. All instructions that fall into this category are usually referred to as *special instructions* because they require special handling [SGG13].

To get around this problem, we will use a technique called *binary translation*. It is more or less exactly what it sounds like. If the guest VCPU is in user mode, the guest can run its instructions natively. If the guest VCPU is in kernel mode, then the guest believes it is running in kernel mode. The VMM looks at every instruction before they get to the CPU to execute. If they are regular instructions, they can execute natively. If they are special instructions, they are translated into (replaced with) alternative instructions that produce the same result [SGG13].



The binary translation virtualization implementation [SGG13].

Although we have a performance decrease as a result of having to examine and replace some of the instructions, most instructions can run natively and only a small number of them need to be emulated. The empirical test of "what is being used in industry" seems to indicate that the performance of binary translation is adequate, as there is now widespread adoption of virtual machines running on x86 hardware.

Okay, the last paragraph was a little bit misleading, because additional hardware support has come along to make virtualization work a lot better than it would have. Starting in 2005, Intel added virtualization support to x86 processors and AMD added it in 2006. This breaks down the old rules of dual-mode (kernel/user) processors into multi-mode operation. The VMM can enable host mode, define a guest's characteristics, and then switch to guest mode, passing control to the guest. If the guest tries to access a virtualized resource, the VMM takes over to manage the interaction [SGG13].

The Impact

The short version of what happens under virtualization is: things get complicated. There are more demands on and difficulties in resources. We will consider the impact on scheduling, memory management, I/O, and disk [SGG13].

Scheduling

Even if there is only one CPU in the physical machine, virtualization presents one or more virtual CPUs to the guests. The challenge then, is to schedule the virtual CPUs' operations on the physical CPU(s). A thread may be a VMM thread or a guest thread.

A guest system is configured with some number of CPUs and as long as there are enough CPUs in the system to meet the allocation commitments (virtual CPU count is less than n), we have no problem. Map each virtual CPU to a physical CPU and we are all set.

If the resources are fully committed (e.g., there are n virtual CPUs allocated), it gets a bit more interesting. The VMM does not (usually) need too much time on its own, so it can basically "steal" cycles here and there. VMM operations run on CPUs that are not busy at the moment, or taking evenly from all the CPUs so as to be "fair".

If the situation is overcommitted (there are more virtual CPUs than physical ones), the problem is more interesting. The VMM will have to figure out a way to map the virtual CPUs to the physical ones according to some scheduling strategy. Like scheduling processes and threads, we can use one of the scheduling algorithms we discussed earlier to schedule all of the threads (or at least choose which guest operating system gets to run right now).

When overcommitment is the situation, the expectation of the guest operating system of certain time deadlines becomes inaccurate. If the scheduler in the guest operating system defines a timeslice as, for example, 50 ms, in reality the actual length of a time slice will vary based on how often the VMM chooses to run that guest's threads. It could, in fact, be significantly longer than the 50 ms intended. This is frustrating to the users and has a tendency to get the system clock out of whack, but may be fatal for any real-time operating system or any task with serious wall-clock deadlines.

Memory Management

Virtualization makes the memory problem a lot worse than it otherwise would be. The processes that run take up plenty of memory all on their own, and where we previously had one operating system and its structures in memory, now there are multiple operating systems and their structures taking up memory space, too. The problem is only exacerbated if memory is overcommitted (the amount of memory the guests are allocated exceeds the RAM of the physical machine). There are a few strategies to alleviate the problem, as outlined in [SGG13].

Nested Page Tables. The guest operating system, unaware it is in a virtual machine, thinks it controls memory and page table management. In reality, the virtual machine manager has a nested page table that re-translates the guest's page table to the real (physical) page table. The VMM can provide double-paging, where it has its own page replacement strategy and tries to help out the guest. The problem is that the VMM knows less about the guest's memory patterns than the guest itself, so this strategy is inefficient.

Device Driver. The next idea is then to install, where possible, a device driver into the guest that allows the VMM to exercise some measure of control over the guest. Where the guest OS allows, the device driver from the VMM is installed. When needed, this "balloon" memory manager is told to request a whole bunch of (empty) memory and asks the guest to pin its pages in physical memory. This makes the guest think that memory is in short supply and the guest will start to free up memory. The VMM knows that the balloon pages are not real and can allocate them to some other guest. If the memory pressure in the whole system goes down, the balloon pages can be deallocated or unpinned, allowing the guest to feel as if it has more free memory.

Duplicate Detection. A third idea is for the VMM to look to see if the same page is loaded more than once. This is obviously more likely if the guests are identical (i.e., the same OS). This will result in significant savings, as the operating system may take up a significant portion of memory on its own. To make the operation efficient, a hash value for memory may be taken; if two hashes match then a byte-by-byte comparison will reveal whether they are actually identical. If so, no need to keep both copies in memory. If there is a modification to a shared page, we need to copy the page before the modification is made.

Input/Output

Unlike CPUs and RAM, the guest OSes are likely to tolerate the fact that the I/O devices may change periodically and during operation, such as when a user plugs in a USB key or moves into range of a WiFi network. In fact, we have device drivers as the interface to the OS, regardless of what type of device it is. When running in a virtual machine, a guest may see several virtual devices as if they were real [SGG13].

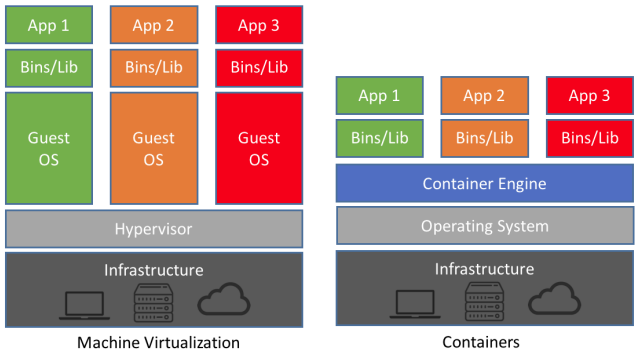
The VMM can decide how to allocate I/O devices to guests; one way is to just dedicate the I/O device to a guest such as a USB key being allocated to one guest in particular. In other cases, the VMM provides drivers that are just translating commands into the actual device commands.

Disk

The typical approach is to create a *disk image*, that contains all the contents of the root disk of the guest. This is one big file (though it can be split up into manageable-sized chunks) and as far as the guest is concerned it has the run of that whole disk, totally unaware that the disk is just a file inside another system. This makes it a lot easier to move a virtual machine around from one system to another, if desired.

Containerize This!

Containerization gives many of the advantages of this separation of VMs, but without nearly so much overhead of the guest operating systems (both its maintenance and runtime costs). Containers are run by a container engine so there is some abstraction of the underlying hardware, and the container is assembled from a specification that says what libraries, tools, etc. are needed. And thus when the container is built and deployed it is sufficiently isolated but shares (in read only mode) where it can. So a container is a very lightweight VM, in some sense. See this diagram from [Cha18]:



Containers are a very effecting way of allowing virtualization, but a major potential limitation is that container applications can only run on hosts with the same OS and same virtualization features [Sta18].

Bibliography

[Bar14] Blaise Barney. POSIX Threads Programming, 2014. Online; accessed 1-March-2015. URL: https://computing.llnl.gov/tutorials/pthreads/.

[BCPV96] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996. doi:10.1007/BF01940883.

[CAS01] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: bridging the theory and practice of proportionate pair scheduling in multiprocessor systems, 2001. doi:10.1109/RTTAS.2001.929861.

[Cha18] Doug Chamberlain. Containers vs. Virtual Machines (VMs): What's the Difference?, 2018. Online; accessed 2019-12-16. URL: https://blog.netapp.com/blogs/containers-vs-vms/.

[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/.

[Dow08] Allen B. Downey. *The Little Book of Semaphores (2nd Edition)*. Green Tea Press, 2008.

[GB95] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1):31–67, 1995. doi:10.1007/BF01094172.

[Goo22] Dan Goodin. How a microsoft blunder opened millions of pcs to potent malware attacks, 2022. Online; accessed 17-March-2023. URL: https://arstechnica.com/information-technology/2022/10/how-a-microsoft-blunder-opened-millions-of-pcs-to-potent-malware-attacks/.

[GRL+12] Joël Goossens, Pascal Richard, Markus Lindström, Irina Iulia Lupu, and Frédéric Ridouard. Job partitioning strategies for multiprocessor scheduling of real-time periodic tasks with restricted migrations. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, page 141–150, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2392987.2393005.

[Hat72] D. Hatfield. Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance. IBM Journal of Research and Development, 1972.

[HS91] S. Haldar and D. Subramanian. Fairness in Processor Scheduling in Time Sharing Systems. Operating Systems Review, 1991.

[HZMG20] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2020. Online; version 0.2020.07.19.

[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.

[MA05] Arezou Mohammadi and Selim G. Akl. Scheduling algorithms for real-time systems. Technical report, Queen's University, 2005.

[RSI11] M. Russinovich, D. Solomon, and A. Ionescu. *Windows Internals: Covering Windows 7 and Windows Server 2008 R2*. Microsoft Press, 2011.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Sir05] John Siracusa. Mac OS X 10.4 Tiger, 2005. Online; accessed 24-May-2015. URL: http://arstechnica.com/apple/2005/04/macosx-10-4/.

[Spo00] Joel Spolsky. Strategy letter ii: Chicken and egg problems, 2000. Online: accessed 18-November-2022. URL: https://www.joelonsoftware.com/2000/05/24/strategy-letter-ii-chicken-and-egg-problems/.

[Sta18] William Stallings. *Operating Systems Internals and Design Principles (9th Edition)*. Pearson, 2018.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.