Contents

1 — Introduction, Operating Systems, Security

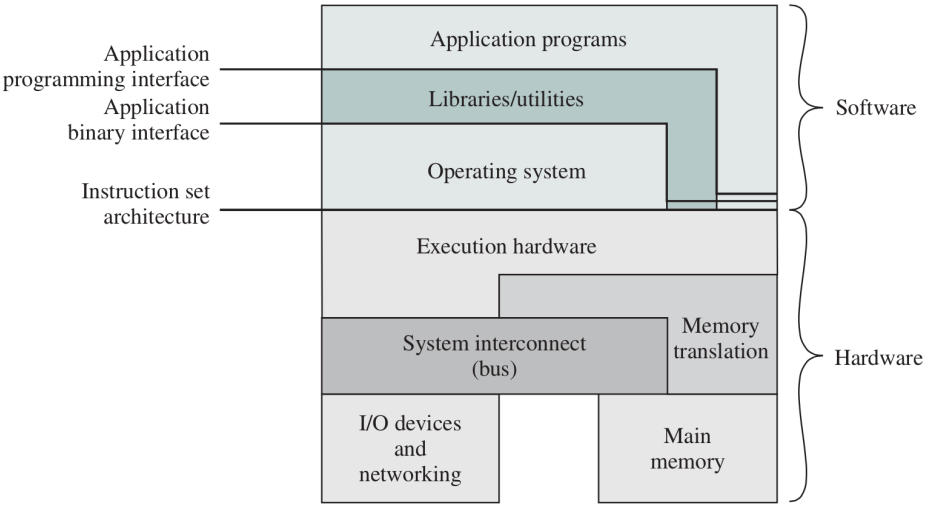
About the Course

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.

Remember, It's a Trap

Operating systems run, as previously discussed, on interrupts. In addition to the interrupts that will be generated by hardware and devices (e.g., a keyboard signalling that the F1 key has been pressed), there are also interrupts generated in software. These are often referred to as a *trap* (or, sometimes, an exception). The trap is usually generated either by an error like an invalid instruction or from a user program request.

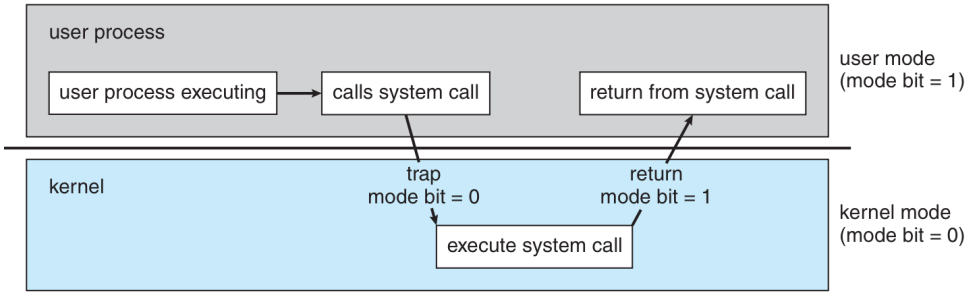
If it is simply an error the operating system will decide how to deal with it, and in desktop/laptop OSes, the usual strategy is sending the exception to the program that caused it, and this is usually fatal to the offending program. Your programming experience will tell you that you can sometimes deal with an exception (perhaps through the language equivalent of the Java/C# `try-catch-finally` syntax), but often an exception is unhandled and terminates the program.

The more interesting case 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.

Already 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.

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

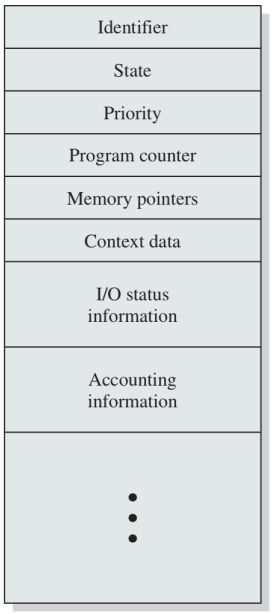
We will take a slight detour to the behind-the-scenes view of how the operating system manages a process, so that we can have a mental model of what may happen when a program is executing. The operating system's data structure for managing processes is the *Process Control Block* (PCB). This is a data structure containing what the OS needs to know about the program. It is created and updated by the OS for each running process and can be

thrown away when the program has finished executing and cleaned everything up. The blocks are held in memory and maintained in some container (e.g., a list) by the kernel.

The process control block will (usually) have [Sta18]:

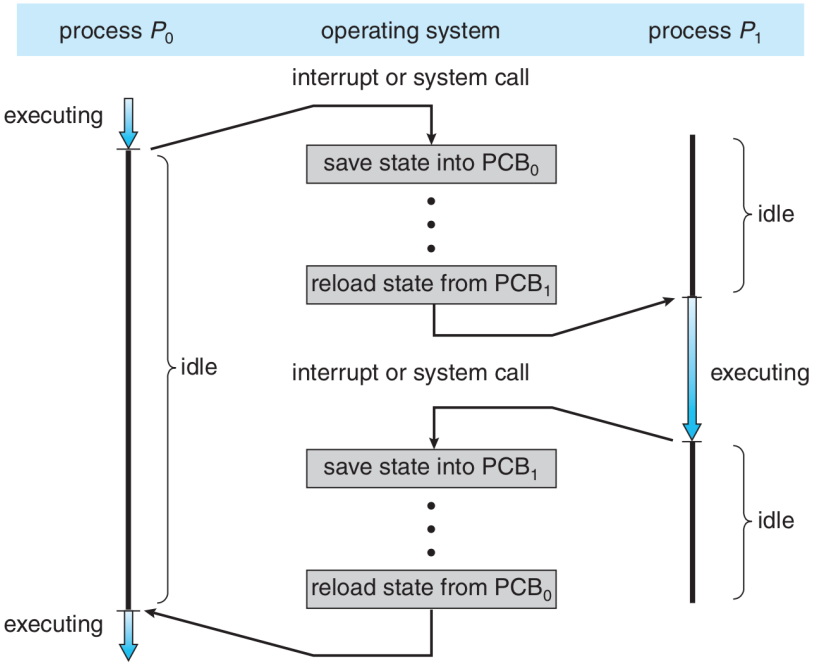
- **Identifier.** A unique ID associated with the process; usually a simple integer that increments when a new process is created and reset when the system is rebooted.
- **State.** The current state of the process.
- **Priority.** How important this process is (compared to the others).
- **Program Counter.** 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:



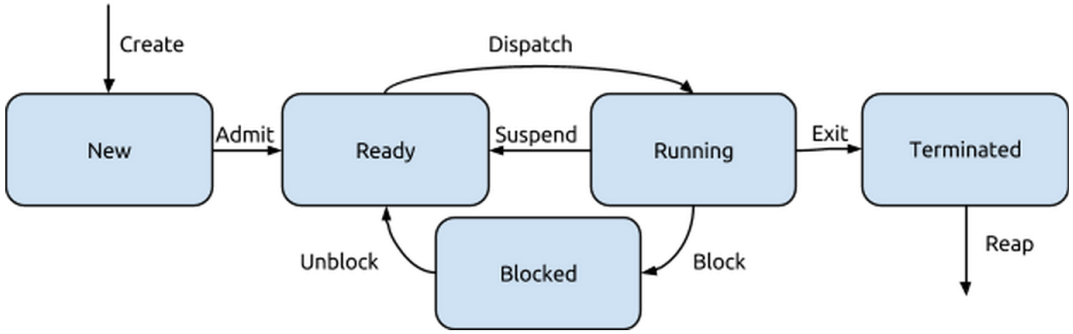
A simplified Process Control Block [Sta18].

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].

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



State diagram for the five-state model.

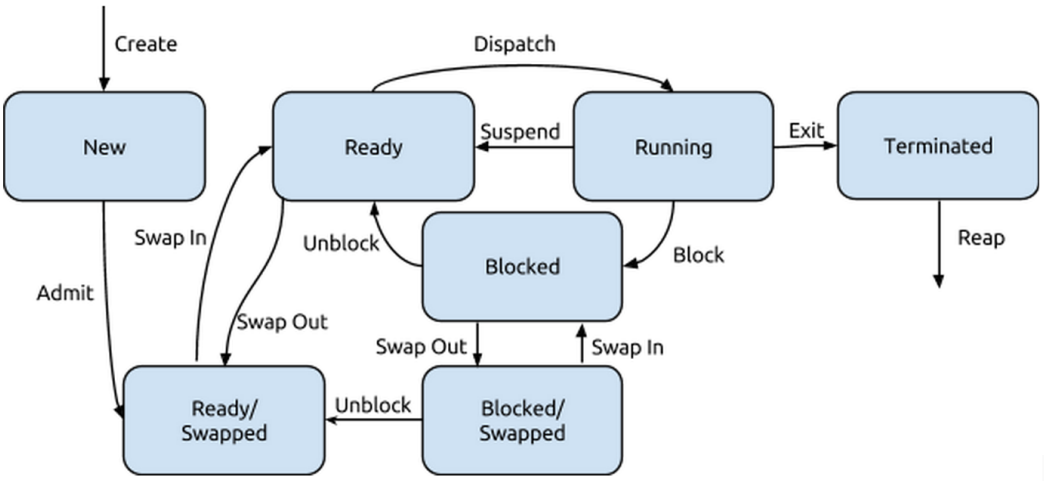
There are now eight transitions, most of which are similar to what we have seen before:

- **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.

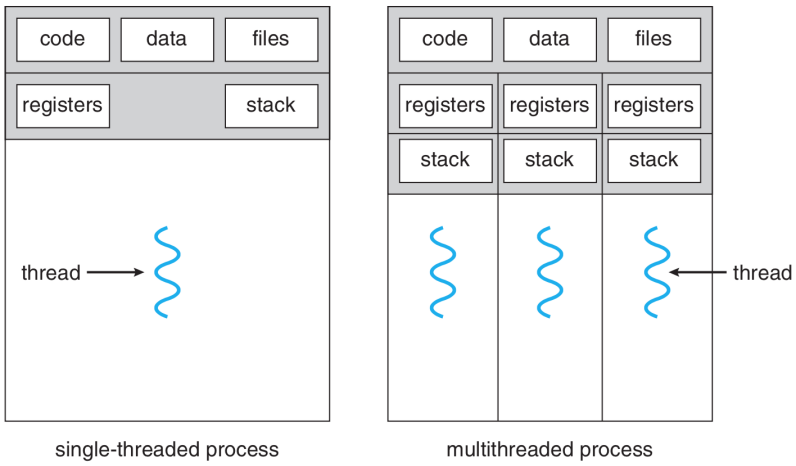
A multithreaded program is one that uses more than one thread, at least some of the time. When a program is started, it begins with an initial thread (where the `main` function is) and that main thread can create some additional threads if needed. Note that threads can be created and destroyed within 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...

12 — 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]:

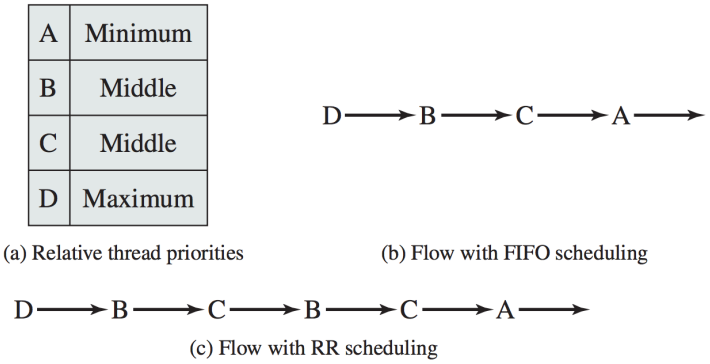
- `SCHED_FIFO`: First-In, First-Out Real-Time threads
- `SCHED_RR`: Round-Robin Real-Time threads
- `SCHED_OTHER`: Other (non-real-time) threads.

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.

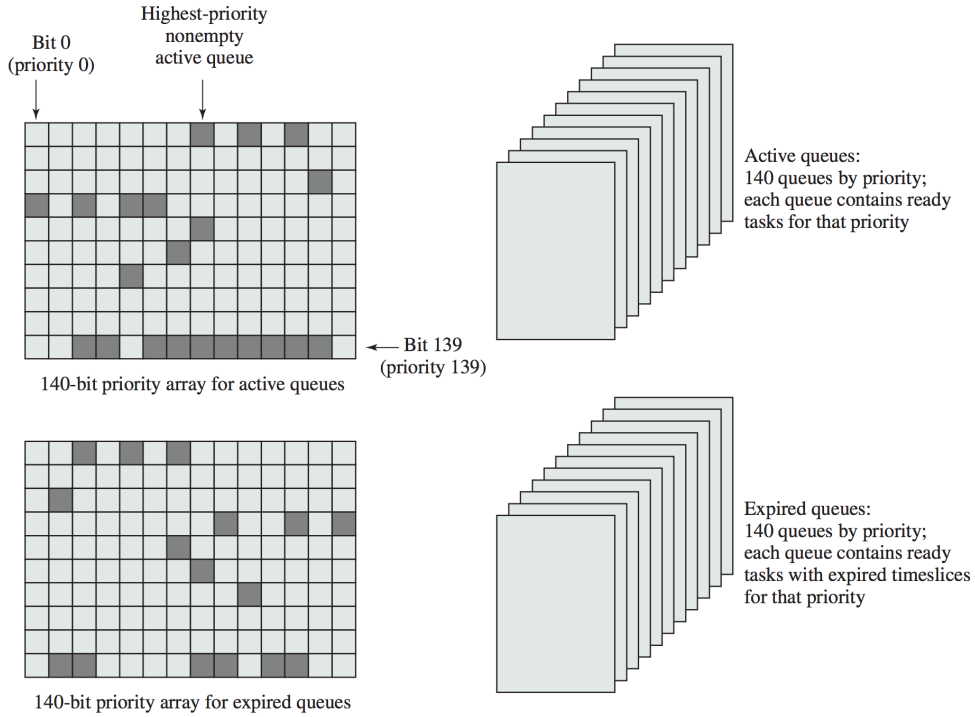
Finally, pre-emption was not possible; lower priority tasks would continue to execute while higher priority tasks were waiting. Only something getting blocked, a time slice expiration, or an interrupt might cause the scheduler to re-evaluate what process should run next.

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:

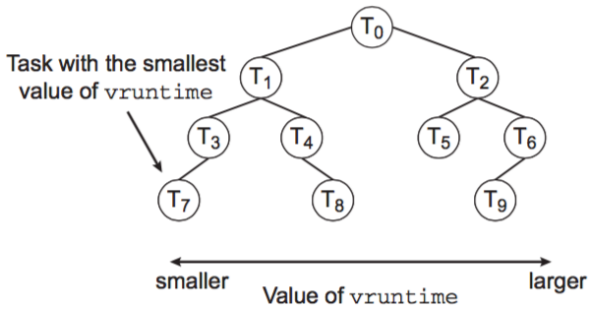


Linux $O(1)$ scheduler internal structures [Sta18].

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) [HZMG15].



The Completely Fair Scheduler's red-black tree structure of ready tasks [SGG13].

Rather than using a strict rule, the CFS scheduler assigns a proportion of CPU processing time to each task based on the nice value. A nice value may be in the range -20 to +19 (lower priority is still higher priority). The CFS does not use a particular length of time slice, but instead has a *target latency* which is an interval of time in which all ready tasks should get to run at least once. The CPU time is then doled out based on the targeted latency. There are usually default and minimum values, but targeted latency can increase if there is a big increase in the number of tasks to be executed [SGG13].

The linear ordering of execution time, called `vruntime` in the earlier diagram, is also called the *virtual run time*. This is a way of keeping track of how much time a task has been executing. As with a lot of history keeping, there is a decay factor so that more recent history is more highly weighed in the calculation. Higher priority processes' history decays faster; lower priority processes' history decays more slowly. For tasks at a normal priority (nice value of zero), the virtual run time equals the physical run time. If the physical run time is, say, 50 ms, a process with a nice value of 0 will have a virtual runtime of 50. If the process has a positive nice value, the virtual runtime will be larger than 50; if a negative nice value, the virtual runtime will be less than 50 [SGG13].

Tasks that spend a lot of time using the CPU will, under this system, normally get a lower priority than a task that spends a lot of time waiting for I/O (e.g., sleeping). So a process that is user-interactive and waiting for user input will get to execute fairly quickly, making the system seem responsive to the user. Which users, of course, like.

Another thing that is noteworthy in the CFS is the addition of group scheduling: we may designate a number of processes as belonging to a group. This is useful when a process spawns lots of threads or lots of new processes. Instead of treating every thread or process totally equally, a multithreaded program's threads can all be pooled together so that the group is equal to other processes. Within the group, the scheduler will try to treat the threads or processes fairly, too.

A Decade of Wasted Cores

In 2016, researchers published a paper, exposing serious problems in the Linux scheduler, with the dramatic title: "The Linux Scheduler: a Decade of Wasted Cores" [LLF+16]. The authors found four significant bugs in Linux multicore scheduling such that there were threads waiting to run even when cores were sitting idle. Performance degradation is in the range of 13-24%, but may be as much as 138 times when looking at some corner cases.

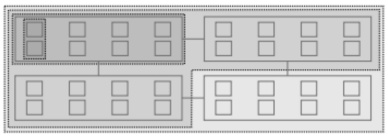
There are four different problems but they all cause the same behaviour: cores are left idle for a long time when runnable threads are waiting to execute. If it is brief, it is not a problem; but if it goes on for longer then it will be more of an issue. Suppose there are 4 CPUs, each of which is busy, and there is one thread waiting in the queue for CPU 0. If the thread in CPU 3 terminates, it may take a moment for the thread waiting on CPU 0 to move there; moving it takes some "effort" on the part of the scheduler (notice this situation, decide to do something about it, actually carry out the move) and potentially results in a few more cache misses. It may be better to leave it alone, But if that thread is waiting an unreasonably long time (in the few hundred milliseconds) then it is a problem.

Recall from earlier the completely fair scheduler we have discussed. There will be multiple run queues, one for each core. The simplest case for load balancing means two CPUs. If CPU 0 has one low priority thread and CPU 1 has three high priority threads, some sort of balancing will be needed, otherwise the high priority threads will run less than the low priority thread. Linux will periodically try to keep the queues balanced.

Unfortunately, load balancing is expensive and will run periodically but not often. But a completely idle core will result in emergency load balancing. There's a problem and we need to do something about it! And you might

imagine that load balancing is just look at how busy each of the cores is and move things from the most busy to the least busy core (... which is what most people do in the ECE 459 load balancing assignment!). That oversimplifies the solution because it does not consider cache locality or non-uniform memory access.

Thus, above the level of each core is a larger unit, a scheduling domain. Scheduling domains are configured by what hardware they have in common (e.g., level 2 cache). See the image below:



A machine with 32 cores, 4 groups, and SMT-sharing amongst pairs of cores [LLF+16].

In this image we have multiple levels: three groups are reachable from the first core (CPU 0) in one hop and the rest reachable in two hops. The scheduler will avoid duplicating work by making sure that one core is responsible for load balancing within that schedule domain. This is the lowest numbered idle core, or the lowest number overall if all are busy. The only way a core can get woken up is for another one to wake it up, and so a core that is busy and notices a lazy one sleeping nearby will wake it up and tell it to do load balancing [Coy16].

So what are the four bugs that caused this problem? The summary of these bugs from [Coy16]: (1) the group imbalance bug, (2) the scheduling group construction bug, (3) the overload on wakeup bug, and (4) the missing scheduling domains bug.

The Group Imbalance Bug Cores would attempt to steal work from other cores if the average load of the victim scheduling group is higher than the average load of the one doing the stealing. But averages can be misleading! The fix is to use the minimum load of the group, meaning the load of the least loaded core of the group. This means cores will steal more often, but this is better than leaving them idle. This can result in a 13% decrease in the runtime of `make`.

Scheduling Group Construction The Linux `taskset` command allows applications to be pinned to specific cores. If the groups are two hops apart, the load balancing thread might not steal them... This problem arises because all groups are constructed from the perspective of core 0. If, therefore, the load balancing is running on core 31 it might not steal from a neighbouring core because it thinks it is too far away because it is two hops from core 0.

Overload on Wakeup We have already discussed the idea of processor affinity, but sometimes, too much of a good thing is a problem. If a thread goes to sleep on group 1, when it and it gets unblocked later by some other thread, the scheduler will try to put it on one of the cores in group 1... even if other groups are not busy. This will reduce the number of cache misses, sure, but it means sometimes a thread gets in a queue that's busy rather than one that's free.

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

In conclusion: scheduling is by no means a solved problem. A simple scheduling algorithm that worked reasonably well in a single core environment was not adequate to the multiple core world. Averages can be misleading and optimizations sometimes do more harm than good.

13 — 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[1]. 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.

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

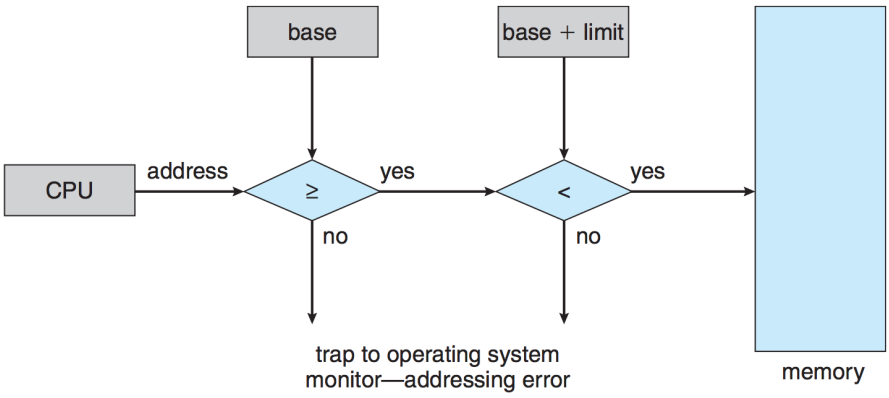
A program executed an instruction directly on a memory address, such as writing 385 into memory location 1024. Suppose you wanted to have two programs running at the same time. Immediately, a problem springs to mind: if the first program writes to address 1024, and the second program writes to address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

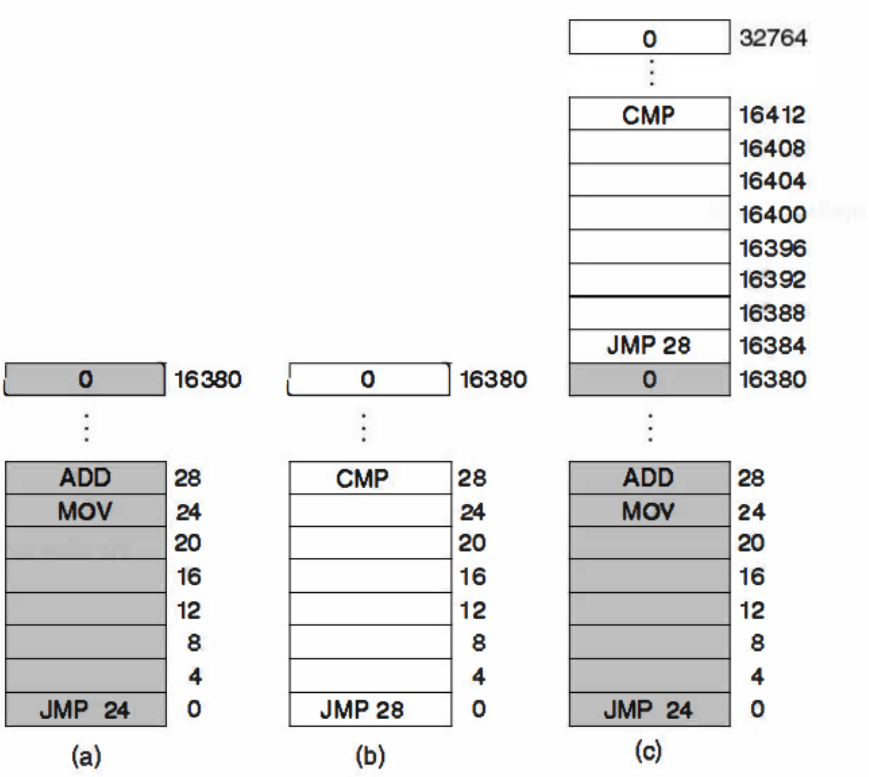
We can attempt to solve the problem of protection by keeping track of some additional information. The IBM 360 solved this problem by dividing memory into 2 KB blocks and each was assigned a 4-bit protection key, held in special registers in the CPU. The Program Status Word (PSW) also contained a 4 bit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

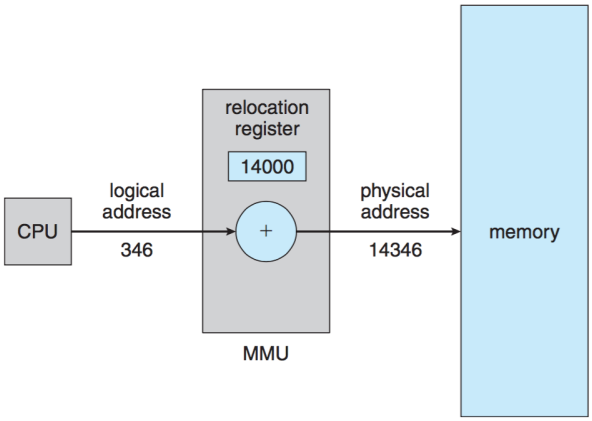
1. **Compile time**: the solution we saw first with commands like `JMP 28`. If we are certain where the process will be loaded into memory, at compile time we can convert those variables to address locations. This is what happens in assembly, and in the MS-DOS .COM format (like `command.com`).
2. **Load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

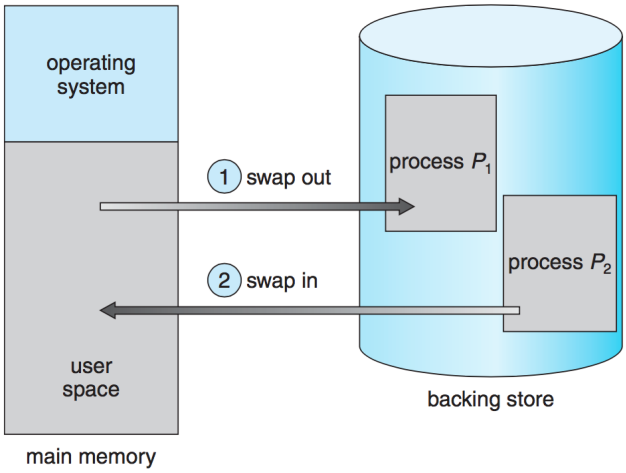
This scheme also gives us something new: we can relocate a process in memory if we change the relocation register's value accordingly. A process that is currently loaded into memory with a relocation register value of 14000 can easily be moved to another location. Copy all the memory from relocation 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[2]. So every memory access has a penalty associated with it to do the addition of the relocation register value to the issued CPU address.

[2]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].

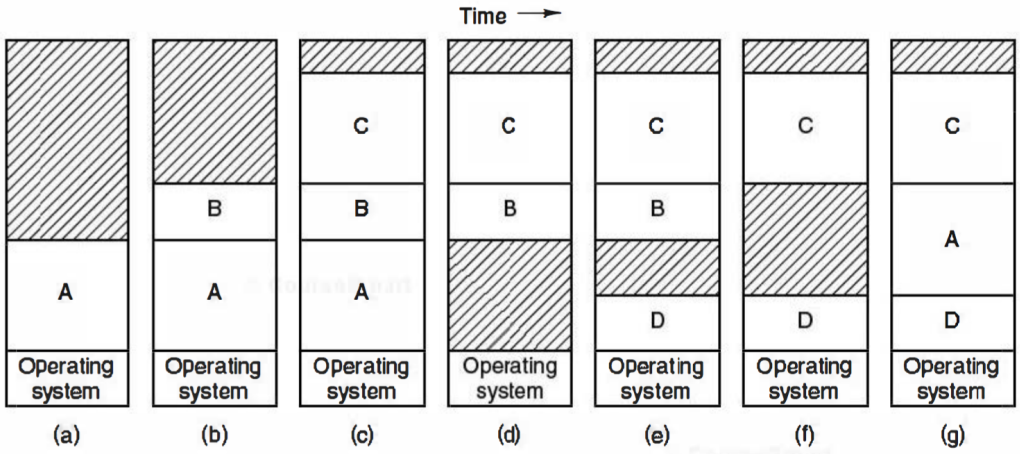
Unfortunately, swapping a process to disk is very painful. If the process is using 1 GB of memory, to swap a process out to disk, we need to write 1 GB of memory to disk. To load that process back later, it means reading another 1 GB from disk and putting it into main memory. If 1 GB strikes you as ridiculous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

14 — 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 [HZMG15].

Fixed Block Sizes, Multiple Size Options. Recognizing that some memory allocation requests are bigger than others, it might make sense to have several different block sizes; perhaps 1 KB, 2 KB, and 4 KB. These can generally be allocated and deallocated in $\Theta(1)$ time if we have one linked list for each different size of block [HZMG15].

Unfortunately, fixed block sizes suffer from a lot of internal fragmentation. This may be suitable for embedded systems where simplicity and speed of operations are more important than worrying about wasting memory. It is obvious from working with languages like C that this is not how `malloc()` works: 1 KB of memory is not allocated to store a 4-byte integer. What we need instead is a variable block size.

Variable Block Sizes

To a certain extent, variable block sizes are not that different from fixed block sizes; we just take the size of blocks down to the smallest they can be. In a typical system with byte-addressable memory, in a way, the smallest block is one byte. Now we have a different problem: keeping track of what is allocated and what is free.

Bitmaps. It is possible to divide memory into M units of n bits, and then to create a bit array of size M storing the status of each of those units. If a bit m in M is 0, it means that unit is unallocated; if it is 1 then that unit is allocated. How much memory is lost to this overhead? $100/(n+1)$% of the memory is used. If a unit is 4 bytes, the bitmap is about 3% of memory; if it is 16 bytes the bitmap takes about 0.8% of memory. Finding a block of k bytes requires searching the bitmap for a run of $\frac{8k}{n}$ zeros [HZMG15].

Linked Lists. The other approach, as in the case of fixed size blocks, is to use linked lists. The information of the linked list can be stored separately from all memory allocation or as part of the block of memory. Either approach is workable.

After startup, the linked list contains one entry, as all available memory is in one contiguous block. When a memory request is allocated, for example, to allocate 128 bytes, the block is divided up. Suppose we allocate the first 128 bytes. A new entry is placed in the list, at 128 bytes. The node that is added contains the start address, the length of the block, and a bit indicating it is allocated. The unallocated block's node will contain the updated entry: smaller size, new start address, and the bit indicating it is unallocated. When a block is deallocated, we simply find that block in the linked list and set the bit to zero to indicate it is now available again.

In a typical system there may be a lot of allocation and deallocation of memory. This will probably lead to breaking memory up into smaller pieces. We may end up with a situation where the free blocks are small and spread out, as in the figure below:

Allocated blocks in memory after some time; the "checkerboard" situation [HZMG15].

If this happens, it may be that there is a contiguous block of free memory available of size N, but this request cannot be fulfilled because the memory is logically split up into smaller pieces. To solve this, we need a way to recombine the split blocks, commonly called *coalescence*. See the updated figure below:

The "checkerboard" situation with the adjacent free blocks coalesced [HZMG15].

Coalescence. Coalescence is just the process of merging two (or more) adjacent free blocks into one larger block. It also makes sense that dividing memory should be a reversible operation. This solves the problem of a block of N contiguous bytes being unable to be allocated. Coalescence can be done periodically or whenever a block of memory is freed.

As pointed out in [HZMG15], coalescence makes it a good idea to maintain the memory blocks in a doubly-linked list. Recall a linked list has "next" pointers connecting the nodes and a doubly-linked list has "next" and "previous" pointers, to make it easier to traverse the list in both directions. When a block is freed, it may be in the middle of two free blocks, so it is convenient to have previous and next pointers so the adjacent sections can be merged efficiently.

Even with coalescence, we may have the problem that N free bytes exist in the system but spread out over many little pieces, so the request for N cannot be satisfied. When free memory is spread into little tiny fragments, this situation is called *external fragmentation*. It is analogous to internal fragmentation in that there are little bits of space that cannot be used for anything useful, except of course that they are not inside any block (hence external).

External Fragmentation. One way to reduce external fragmentation is to increase internal fragmentation. If a request for N bytes comes in and there is a block of $N + k$ available, where k is very small (and unlikely to be allocated on its own), it makes sense to allocate the whole $N + k$ block for the request and just accept that k bytes are lost to internal fragmentation. For example, if a free block contains 128 bytes and the request is for 120 bytes, it may not be worth the hassle and overhead to split this block into 120 and 8, as it is unlikely the 8 bytes will be filled anyway. Some systems round up memory allocations to the nearest power of 2 (e.g., a request for 28 bytes gets moved up to 32). Of course, this does not really help with satisfying the request for N bytes of memory; it just keeps external fragmentation down.

Another idea is *compaction*, which can also be thought of as *relocation*. The goal is simply to move the allocated sections of memory next to one another in main memory, allowing for a large contiguous block of free space. This is a very expensive operation; to do this successfully, the Java runtime, for example, must "stop the world" (halt all program execution) while it reorganizes memory. This tends to make Java unsuitable for use in writing a real-time operating system. But even if we are willing to pay the cost, it might not be possible to do.

In previous discussions of memory management from the perspective of the application developer, languages with garbage collection like Java or C# may do memory compaction as needed when the garbage collector runs. This can work in such languages, because variables are references and unless you are writing an `unsafe` block in C#, references can be moved around in memory at the garbage collector or runtime's convenience; all it needs to do is update every reference. This is not the case in languages like C where we operate directly on memory addresses, and thanks to things like pointer arithmetic and using integer variables as addresses, there is no reliable way to update all references.

The final way we can try to prevent or deal with external fragmentation is through different allocation strategies; that is, how to fit a memory request to a block of free memory. We will examine those strategies now.

Variable Allocation Strategies

Given a memory request of N, where do we allocate the memory? If there is no block of at least size N, the request cannot be satisfied. If there is only one, the decision is easy. As long as memory has two free blocks of sufficient size (N or more) that cannot be coalesced, a memory allocation request will require making a decision about which of those blocks to split to meet the allocation request. There are five strategies we will examine [HZMG15]:

1. First fit.
2. Next fit.
3. Best fit.
4. Worst fit.
5. Quick fit.

As a performance optimization, we could have two linked lists: one for allocated memory and one for unallocated memory. That way to find a free block we do not have to look through the allocated blocks.

First fit. The strategy of first fit is to start looking at the beginning of memory, and check each block. If the block is of sufficient size, split it to allocate the memory, and return the balance to the unallocated memory list. This algorithm has a runtime of $O(n)$ where n is the number of blocks. This algorithm is simple to implement.

Next fit. This strategy is a modification of the first-fit algorithm. Instead of starting at the beginning of memory and finding the first block that meets the request, keep track of where the last block was allocated, and then start the next search after that. This prevents the situation where there are a lot of small unallocated blocks (external fragmentation) all concentrated at the start of memory [HZMG15]. The runtime is still $O(n)$, as with first fit.

Best fit. Instead of just walking through the list and splitting up the first block equal to or larger than N, we could instead try to make a more intelligent decision. Considering all blocks, we choose the smallest block that is at least as big as N. This produces the smallest remaining unallocated space at the end.

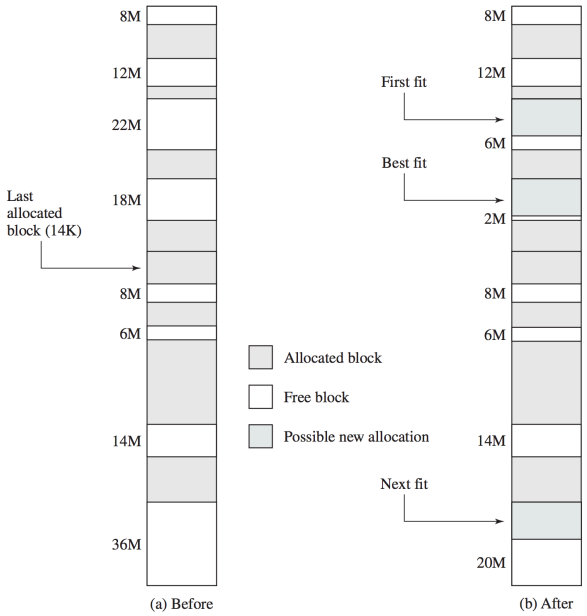
This would require either (1) checking every available block ($\Theta(n)$ runtime); or (2) keeping the blocks sorted by increasing size ($O(n)$ runtime). If we use an AVL tree or red-black tree, then we can get best fit to run in $\Theta(\ln(n))$, possibly better runtime than first fit [HZMG15].

Worst fit. The problem with best fit is that the leftover bits of memory are likely to be too small to be useful. Rather than trying to find the smallest block that is of size N or greater, choose the largest block of free memory. When the block is split, the remaining free block is, hopefully, large enough to be useful.

As with best fit, we must either (1) check each available block; or (2) keep the block sorted by size, though decreasing size this time. A max heap is appropriate, or a binomial or Fibonacci heap could also be appropriate [HZMG15].

Quick fit. Though not a solution on its own, quick fit is an optimization. If memory requests of a certain size are known to be common, e.g., requests for 1 MB, it might be ideal to keep a separate list of blocks that are of perhaps 1-1.1 MB in size, so that if the request for 1 MB does come in, it can be satisfied immediately and quickly.

Example: 16 MB Allocation. The diagram below illustrates where in memory a request of 16 MB may be placed:



An example of where the first, best, and next fit algorithms would place an allocation [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

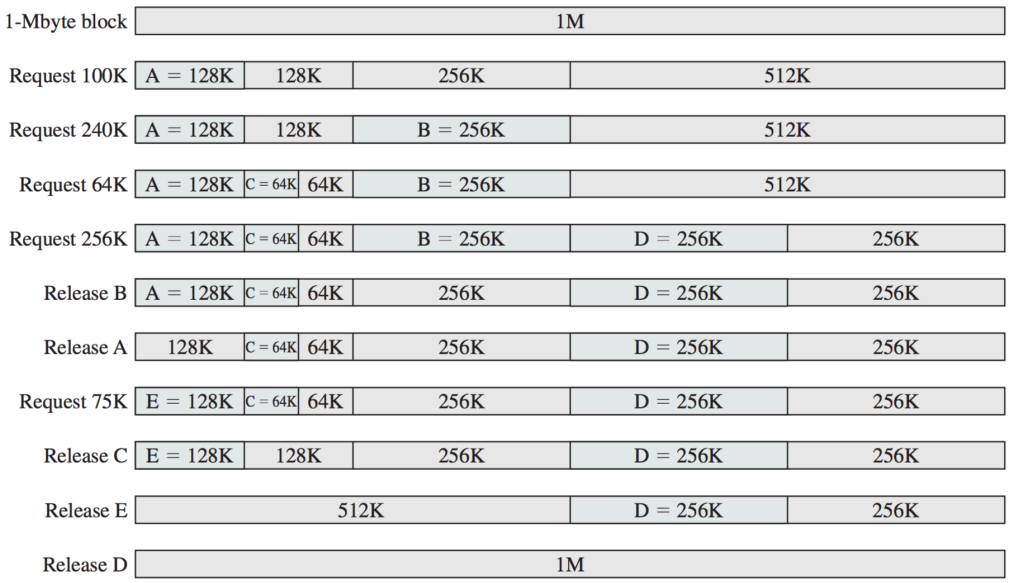
Now let us examine a compromise between fixed and variable allocation, as laid out in [Sta18]. There is some internal fragmentation, but it is a trade-off against how much external fragmentation we are willing to accept.

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].

15 — Memory: Segmentation and Paging

Memory Segmentation

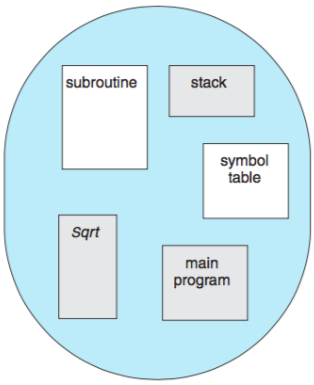
Though you've been repeatedly told that memory is a linear array of bytes, you have also likely been told that there's the stack and the heap, libraries and instructions. Both of these are true; they are simply views of memory at two different levels of abstraction. Each of the elements such as the stack,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

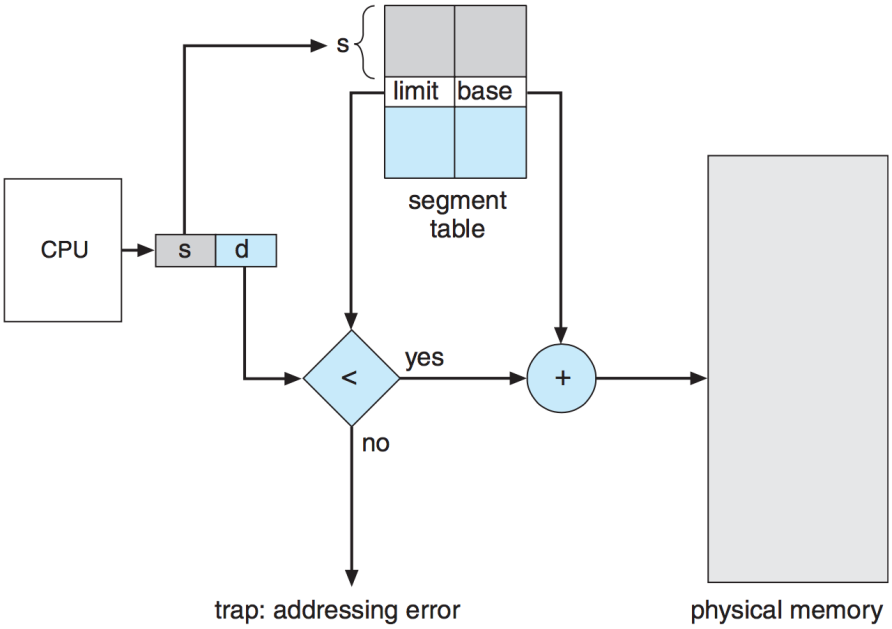
1. The code (instructions).
2. Global variables.
3. The heap.
4. The stack (one per thread).
5. The standard C library.

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

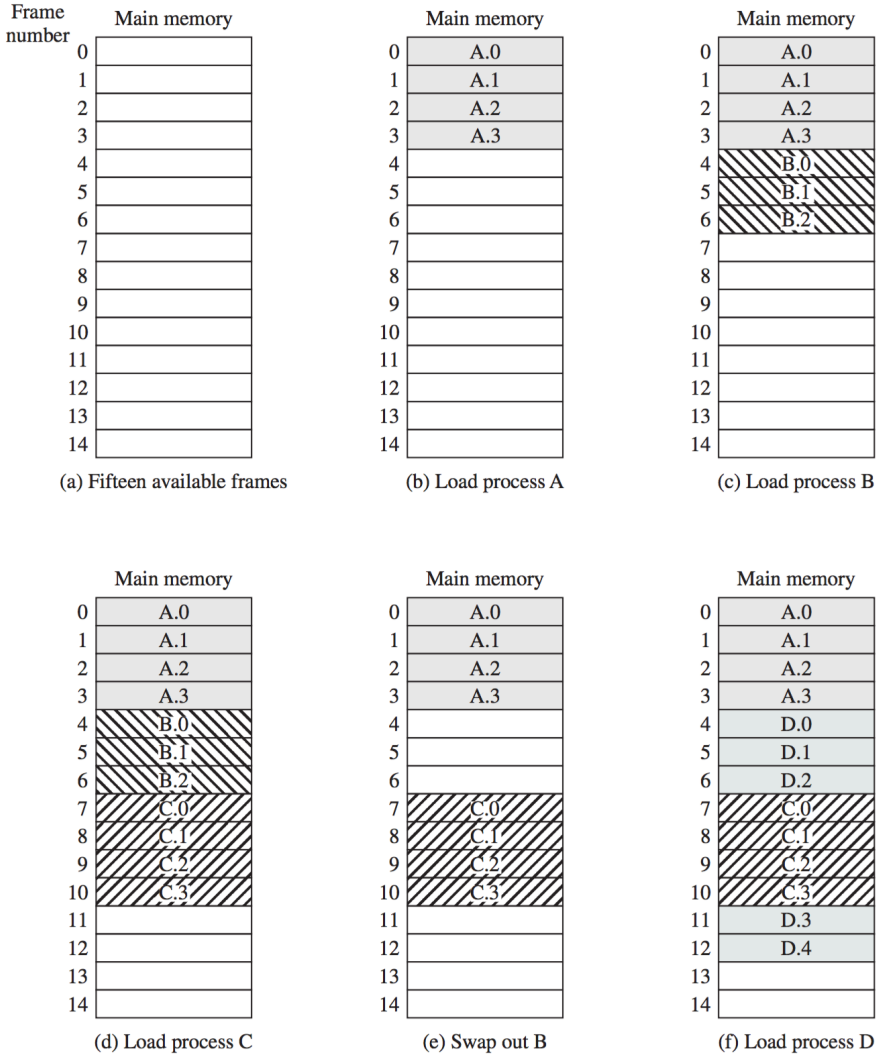
Fixed and variable sized partitions suffer from fragmentation, whether external or internal. Let us divide memory up into small, fixed-size chunks of equal size, called *frames*. Each process's memory is divided up into chunks the same size as a frame, called *pages*. Then a page can be assigned to a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

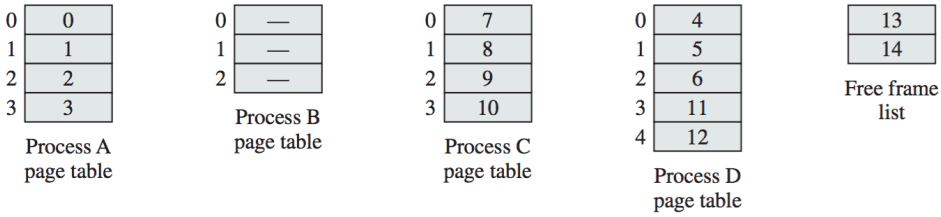
Pages provide the benefit of separating the logical address from the physical address: programmers may pretend the address space of the computer is 2^{64} bytes rather than however many GB of memory are in the physical machine.

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).



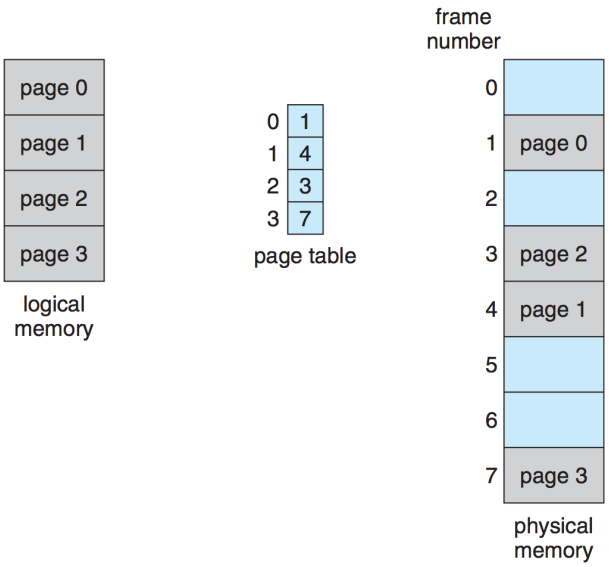
Assignment of process pages into free frames [Sta18].

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 [HZMG15]:

```
int tmp;
void swap( int *x, int *y ) {
  tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Page Table Structure

In the simplest form, the page table is just a standard table. This structure is simple, but page tables can be very large. If the system is 32-bit, and page sizes are 4 KB (2^{12}), then the page table has $2^{32}/2^{12} = 2^{20}$ pages, or about 1 million entries. Given that page tables themselves can be quite large, we will examine three strategies for how to structure the page table, other than the simple table structure:

1. Hierarchical paging.
2. Hashed page tables.
3. Inverted page tables.

Hierarchical Paging. Rather than have one big table, we have multiple levels in the page table; this means the page table can be broken up and need not be contiguous in memory. Suppose we have a two level system. If the page number is p, the first k bits indicate the *outer page*. The outer page then contains some information about where the *inner pages* are. The remaining $p - k$ bits identify the inner page. After the inner page is identified, the displacement d is then calculated from the inner page [SGG13].

Hashed Page Tables. Instead of the page table being an array of entries, turn it into a hash table. There is a hash function to assign pages to "buckets" and each bucket is implemented as a linked list. Then each element of the list is examined to find the matching page.

Inverted Page Tables. For 32-bit virtual addresses, a multilevel page table can work. But with 64-bit computers, with 4 KB pages, the page table requires 2^{52} entries, and if an entry is 8 bytes, then the table is over 30 million gigabytes (30 PB). Using this much memory for the page table is unrealistic. Instead, we can have an inverted page table: there is one entry per frame, rather than one entry per page. The entry keeps track of the process and page number. This saves a huge amount of space (1 GB of ram with a 4 KB page size means the page table requires only 2^{18} entries). The drawback is that it is no longer possible find a physical page by looking at the address; instead, we must search the entire inverted page table. Slow as this operation is, we can make it faster via hardware... [Tan08].

Paging: Hardware Support

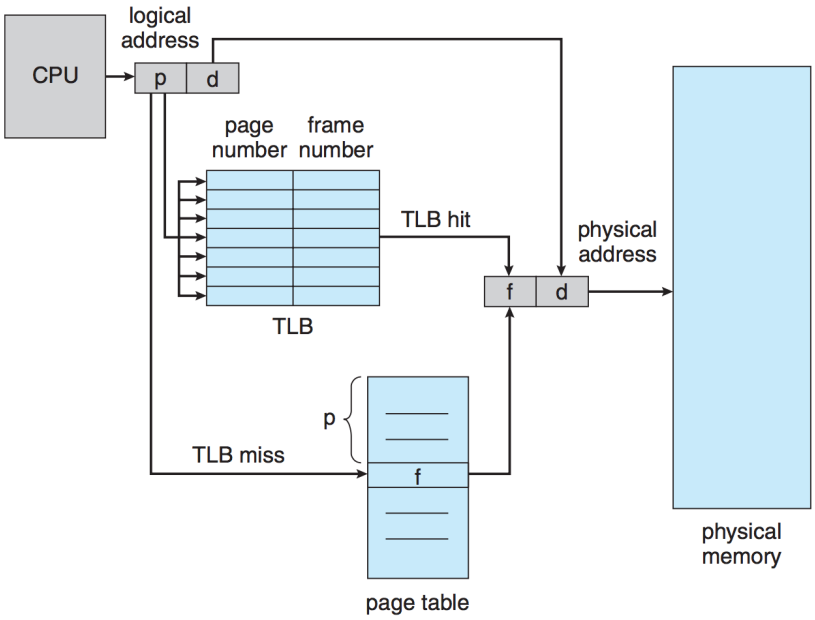
As we have repeatedly discussed, memory accesses are very frequent and often require additions and comparisons. After all, an operation as simple as adding two numbers requires fetching the add instruction, fetching the operands, and storing the result. To prevent abysmal performance, modern computers have hardware support, because hardware is much, much faster than doing these operations in software.

The simplest implementation of the page table is to use a set of dedicated registers. Registers are the fastest form of storage. When a process switch takes place, these registers, just as all other registers, are replaced with those of the process to run. The PDP-11 was an example of a system that had this architecture. Addresses were 16 bits and the page size was 8 KB. Yes, it was a very long time ago. The page table was therefore 8 entries and kept in fast registers. This might work if the number of entries in the page table is small (something like 256 entries). The page table can easily be something like 1 million entries, so it would be a little bit expensive to have that many registers. Instead, the page table is kept in main memory and a single register is used to point to the page table [SGG13].

Unfortunately, this solution comes with a big catch. To access a page from memory, we need to first figure out where it is, so that requires accessing the page table in main memory. Then after retrieving that, we can look in the page table to find the frame where the desired page is stored. Then we can access that page. So two memory accesses are required for every read or write operation. Remember that as far as the CPU is concerned, main memory already moves at a snail's pace. Doubling the amount of time it takes to do a read or write means it takes roughly forever. Thus, we will need to find a way to speed this up.

Surprise surprise, the solution is hardware: a fast cache called the *translation lookaside buffer* (TLB). You can think of the TLB as a key-value pair (think HashMap). The key is the logical address and the value is the physical address. To make the search fast, the comparison is done on all items simultaneously. To prevent this from being extremely expensive, the size of the TLB is limited; it's usually something around 32 to 1024 entries in size. Systems have evolved from having no TLBs to having multiple levels, over time [SGG13].

When a memory read or write is issued, the page number is checked against the TLB. If it is found in the TLB then the frame number is immediately known. If the page number is not found in the TLB, this is what we call a *TLB miss* and we must look in the full page table, which unfortunately is slower because it requires reading from memory. See the diagram below:



Mapping of logical memory to physical memory via the page table [SGG13].

The TLB idea is a specific instance of the strategy of caching. Much earlier, when talking about the basics of computer hardware, we mentioned that memory comes at different levels and and different speeds. Caching is a

critical idea in computers and operating systems. In fact, caching is such an important topic, that it will be the next topic examined.

16 — 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.

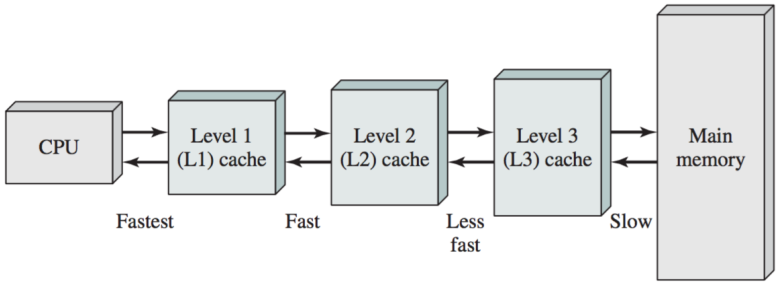
Caches do not have to operate on pages; they can operate on anything, but they are typically blocks of a given size. An entry in a cache is often called a *line*. We will assume for the balance of this discussion that a cache line maps nicely to a page.

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 [HZMG15]:

1. A rule of thumb in software engineering is that 10% of the source code will be executed 90% of the time. This is a variant on the Pareto Principle, also known as the 80/20 rule[3]. 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[4]. 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)

[3]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.

[4]Recall the joke "Why do you never see the newspaper headline that a psychic has won the lottery?"

and the M bit is set when the page is written to (modified). Once a bit is set, it remains so until the OS changes its value. The R and M bits can used to build a paging algorithm as follows [Tan08]:

Initially, R and M are 0. Periodically, the R bit is cleared to distinguish pages that have not been recently referenced. This may happen every clock interrupt, for example. When a replacement needs to take place, the operating system will examine all the pages and sort them into four buckets based on the R and M bits, in ascending order of precedence:

1. Not referenced, not modified.
2. Not referenced, modified.
3. Referenced, not modified.
4. Referenced, modified.

The OS will prefer to remove a page from the lowest-numbered class, when possible. This NRU algorithm is fairly easy to understand and may provide adequate performance.

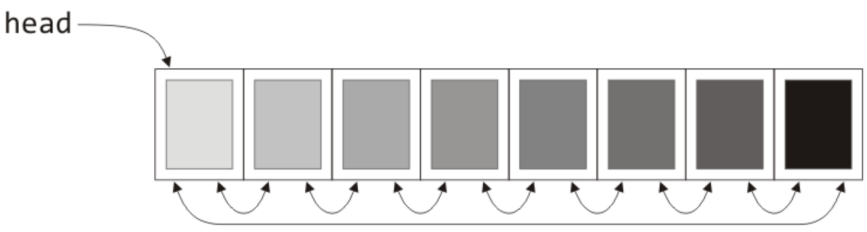
First-In-First-Out. First-In-First-Out is quite easy to understand and implement; the idea is some sort of temporal locality. If there are N frames, keep a counter that points to the frame that is to be replaced (the counter ranges from 0 to $N - 1$). Whenever a page needs to be replaced, replace the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

Some of the time this strategy works: if a page is not going to be referenced again (or at least, for a very long time), it is a good choice to get rid of. Often times, however, the same page is referenced repeatedly, and this algorithm does not take that into account. If a page is referenced often, we want to keep it in memory, not evict it just because it has been in the cache the longest. So FIFO is probably not the best choice.

A Second Chance (the Clock Algorithm). To improve on the FIFO algorithm, suppose we give pages a "second chance" based on whether or not the R bit is set. If the oldest page has not been used recently (the R bit is not set), then the oldest page is removed. If the oldest page has recently been referenced, the R bit is cleared. The search then goes backwards (to the next-oldest page) and repeats the procedure. This happens until a page is removed. Because the algorithm clears the R bit as it goes, even if all pages have recently been referenced, eventually a page will be selected and evicted. Thus the algorithm will eventually terminate. The index is updated whenever this happens so a page that got a second chance is not the "oldest" anymore. In some textbooks this is called the Clock replacement algorithm because we can think of the cache as a circular buffer and the current oldest page as being pointed to by the hand of the clock [Tan08].

This addresses the problem of a page that is frequently used eventually becoming the oldest and being evicted, only to be brought back into memory again immediately afterwards.

Least Recently Used (LRU). The least recently used (LRU) algorithm means the page that is to be replaced is the one that has been accessed most distantly in the past. You might consider time stamps and searching a list, but because there are only two operations, it need not be that complex. When a page in the cache is accessed, move that page to the back of the list. When a page is not found in cache, the page at the front of the list is removed and the new page is put at the back of the list. This requires nothing more than a cyclic doubly-linked list. Both operations can be performed in $\Theta(1)$ time [HZMG15].



A cyclic, doubly-linked list in which the `head` pointer indicates the least recently used page [HZMG15].

Not Frequently Used (NFU). The not frequently used (NFU) algorithm is similar to the LRU approach, but can be implemented in software rather than relying on hardware support that may not be present. Each page gets an associated software counter, which starts at 0. Whenever the R bit would have been updated to 1, 1 is added to the counter. When a page is to be replaced, the page with the lowest counter value is the one that is replaced [Tan08].

It may occur to you that this solution has a problem: like an elephant, it never forgets and counters can never decrease. A page that was accessed very frequently at the start of the program will accumulate a high counter value early on, and therefore might never be evicted, even though it is not needed again. What we need is a way for the count to decline over time.

The solution is called *aging*: counters are shifted to the right by 1 bit before the 1 is added; and instead of adding 1 (which increments the rightmost bit), set the leftmost bit to 1. Now we have a bit array of byte size (no need for a 4-byte integer, most likely). Thus, a page that has not been referenced in a while has its value decrease over time. The page replacement algorithm still evicts the lowest value page when it is time to replace a page. We lose a certain amount of precision compared to the LRU approach: if pages a and b both have patterns of `00000001` we know they were both last accessed 8 cycles ago, but all history before that point is lost. Which page between a and b was least recently used is unknown [Tan08].

Pre-Paging. Thus far all of our strategies for putting things in the cache have been "on-demand": it is only when a page is needed and not found in the cache that a page is loaded into the cache. Suppose instead that the operating system takes some steps to guess about what pages might be needed next, based on the principle of temporal or spatial locality. This might reduce the amount of time spent waiting for a page in the future.

Predicting which pages are likely to be used in the near future is, indeed, a matter of clairvoyance. There exist various techniques to determine the likely pages to be used frequently (called the "working set"), but they are complex and will not be examined further right now.

A situation in which pre-paging might be useful is when a program is started or swapped into memory. Technically, no pages of the process need to be loaded into the cache to start with; we can simply suffer through a whole bunch of page faults. This is rather slow; it might be better to load multiple pages into the cache to start with, though this will of course require some guesses about what pages will be needed [Tan08].

Choosing an Algorithm

Consider the following table that gives a quick overview of the algorithms we have discussed:

Algorithm	**Comment**
Optimal	Impossible to implement, but a benchmark to compare against
Not Recently Used	Not very good performance
First In First Out	Highly suboptimal
Second Chance	Much better than FIFO, but just adequate
Least Recently Used	Best performer, difficult to implement?
Not Frequently Used	Approximation of LRU
NFU + Aging	Better approximation of LRU

If hardware is available to support it, the LRU algorithm is the best. If not, the NFU + Aging scheme is the next best thing we can implement. Unless there is a specific reason to choose one of the other algorithms, LRU is very likely to be the algorithm selected.

Local and Global Algorithms

When a process switch occurs, we could dump the entire cache to make way for the next process to run, but this is most likely unnecessary work. If a process P_1 is suspended now, P_2 may run for a while and then P_1 runs again; it may be that when P_1 starts again, some of its pages are still in the cache and do not need to be loaded again. If P_2 did replace all the cache lines, then P_1 will have to load them all in again, but that is the worst case scenario. So we may have multiple processes with pages in the cache at a given time.

Another important consideration in the page replacement algorithm is whether that algorithm should care about which process the page belongs to or not. Suppose we are using the LRU algorithm. If process P_1 has a page fault, do we replace the least recently used page in all the cache (global replacement)? Or do we replace the least recently used page in the cache that belongs to P_1 (local replacement)?

Of course, different levels of cache might have different strategies: if the L1 cache is 16 KB and pages are 4 KB, there can be 4 pages in the L1 cache, so global replacement probably makes sense there. If L2 cache is 256 KB there are 64 pages and maybe local replacement makes sense there, but even 64 pages is fairly small. When caches are somewhat larger, however, things may be a bit more interesting.

Local algorithms give each process some (roughly) fixed number of pages in the cache. Global algorithms dynamically allocate cache space to different programs based on their needs, so the number of pages in the cache for each process can vary over time.

According to [Tan08], global algorithms work better, especially when processes' memory needs change over time. If the local algorithm is used, we may be wasting some of the space in the cache for a process that does not really need it.

Suppose we have a sufficiently large cache, and a dynamic allocation (global algorithm) with some intelligence to keep any process from having too many or too few pages in the cache. To manage the complexity of how much of the cache should be allocated to a particular process, we might wish to keep track of the number of page faults with a measure called the *page fault frequency* or PFF. In simplest terms, the PFF is the guide for telling whether a given process has too few, too many, approximately the right number of pages.

If the PFF is above some upper threshold, that indicates that more cache space is needed for that process. If the PFF is below some lower threshold, it suggests that the process has too much cache space allocated and could stand to part with some.

PFF relies on an assumption: that the page replacement algorithm has the property that fewer page faults will occur if a process has more pages assigned. This is the case for the LRU algorithm, but not necessarily true for the FIFO approach [HZMG15].

17 — Virtual Memory

Virtual Memory

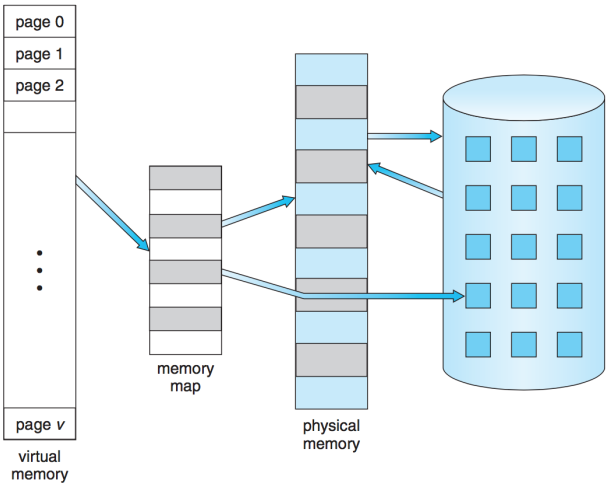
Even with paging and moving pages into and out of memory, there is a limit on what we can do with the system. Specifically, a program that requires more memory than the machine's physical memory cannot run. Maybe that seems ridiculous in the modern era of 4, 8, and 16 GB of memory, but server or supercomputer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

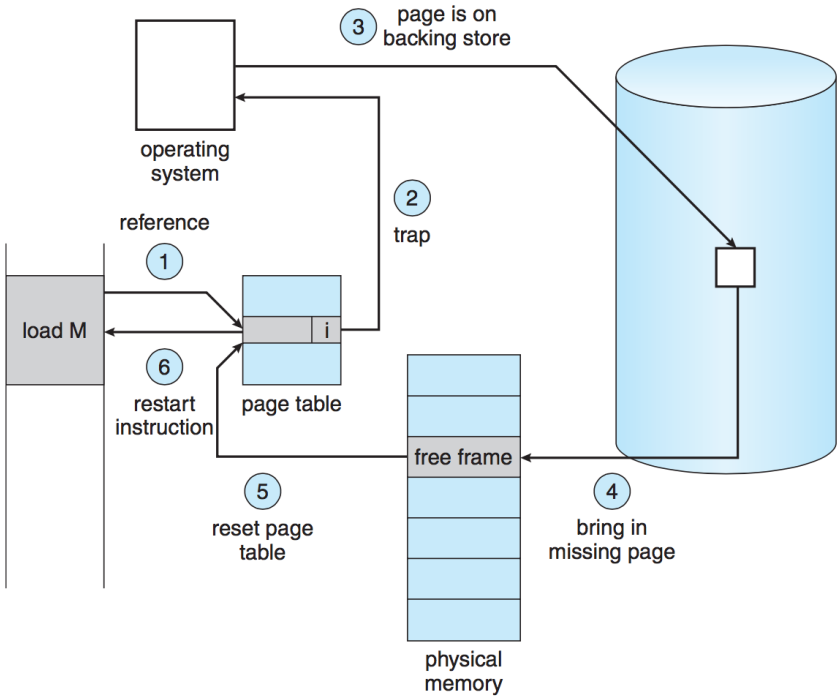
Apparently, if the NeXT operating system, NeXTStep[5] were booted on a machine with only 2 MB of RAM instead of the expected 4 MB, the steady state of the system would be a constant level of swapping [HZMG15]. This was particularly bad because the most common cause of failure in the NeXT boxes was hard disk drive. But they did have cool magnesium cases, so after they died they at least made impressive conversation pieces and doorstops.

With virtual memory, each memory reference is a six step process [SGG13]:

1. Check if the memory reference is valid or invalid (just as we have done before).
2. If the reference is invalid, terminate the program (segmentation fault). If it was valid, but the page referenced is not in memory, we will need to retrieve it.
3. Find a free frame (or make one by evicting some other page).
4. Request a disk read (and possibly write) to bring in the new page.
5. When the disk read is complete, update the records to show the new page is in memory.
6. Restart the instruction that referenced the page that needed to be brought into memory.

Or, to view this visually:

[5] 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[6], 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[7]. 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.

[6] 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.

[7] 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.

18 — Virtual Memory II

Virtual Memory, Continued

We will continue with our discussion of virtual memory by looking at a few advanced considerations in virtual memory.

Allocation of Frames

In a simple system where we do not do anything advanced, if there are n frames free in the system, we will demand page all of them. So the initial state is that all frames are empty, and as needed, pages are read into those frames. Once all n frames are filled with pages, page $n + 1$ must replace a page already in a frame (because there is no more space). When a process terminates, all its frames are marked as free. In theory, one process could fill all the frames in the system. This is as simple as it can be; from there we can build on it.

We might reserve a few pages to be free at all times for performance considerations. When we want to move a page into a frame, if all of the frames are full, we select a victim and write that victim out to disk if necessary. If we keep a few frames free, the newly-read page can be read into one of the free frames, and we can write the old page out to disk at a convenient time. The read does not need to wait for the write (flush of the old page) and therefore the user process gets to continue a bit sooner than it otherwise would.

Assuming, as we did with cache, that we might want to allocate different numbers of frames to different processes (and not necessarily let one run wild), we are constrained in the number of pages we can allocate. The maximum number of frames a process could have is the maximum number of frames in the system (obviously), but the minimum is more interesting.

The motivation behind allocating at least a minimum number of frames is ostensibly performance. As long as our page replacement algorithm is sane (i.e., not FIFO), adding more frames reduces the page fault rate. As we demonstrated earlier, a page fault is a huge performance decrease.

The absolute minimum number of frames is determined by the architecture of the system. Imagine a machine where a memory reference instruction may contain one memory address. In the worst case, the instruction and the address are in different pages, so we will need two frames to be able to complete this instruction. If the max frames for this process were 1, the instruction could never be completed. The IBM 370 `MVC` instruction is an extreme example: it moves data from one storage location to another. It takes six bytes, and the instruction itself can straddle two pages. That requires six frames. The worst case scenario is when the `MVC` instruction is the operand of an `EXECUTE` instruction that also straddles a page boundary. So eight frames are needed [SGG13].

In theory, the problem could be infinitely bad if the computer architecture allows referencing of an indirect address. The address being referenced could be on another page. That instruction could then reference another indirect address, and it's turtles all the way down. If that is the case, the entire virtual memory must be in main memory (so, not possible). The standard solution is to limit the levels of indirection to some manageable value, like 16. Rather like recursion leading to a stack overflow, it is possible that a stack overflow prevents a program that is correct from running, but more likely the program is in an infinite loop and it is better off terminated. If we limited the levels of indirection to 16, then the worst case scenario is a requirement of 17 pages [SGG13].

Assuming we do not allocate every process the minimum or maximum, there are a few allocation algorithms we

might follow. We already got a glimpse at this when we talked about local vs. global cache replacement.

If there are m frames in the system and the operating system reserves k of them for its own use, there are $m - k$ frames available for processes.

Equal Allocation. If there are n processes in the system, if we allocate them equally, each process gets $(m - k)/n$ frames. If this division produces a remainder, the leftover frames can be kept as a pool of free frames for performance purposes as above. There is an obvious flaw in this plan: why does a text editing program get the same amount of frames as a web browser and the same amount of frames as a game (which tends to be VERY demanding on memory).

Proportional Allocation. So how about proportional allocation: each process should get a share of the frames based on its needs. The strategy suggested in [SGG13] to do frame allocation runs something like this. Let the virtual memory size of a process p_i be defined as s_i. Thus $S = \Sigma s_i$. If the total number of frames in the system is, once again, m, and the operating system reserves k for itself, then we allocate a_i frames to a process p_i according to the following formula: $a_i = s_i/S \times (m - k)$.

This value of a_i is only an estimate. It may not divide evenly, and we can only allocate an integer number of frames to each process. So we will have to raise or lower a_i a bit to make it an integer. If a_i is below the minimum number of frames, then it needs to get bumped up to that minimum. We also must respect the limits of the system: the sum of all a_i values may not exceed the total frames of memory (minus the OS reserve), so a few of the larger processes may need to have their allocations nudged down.

Note that with proportional allocation, as with equal allocation, there is no regard given to the priorities of the processes. Normally, we would want to give more frames to higher priority processes so their page fault rates are lower and they can therefore execute more quickly. So we could modify the a_i values according to process priority, as well [SGG13]. The subject of priority is something we have not yet examined much yet, but will do so soon when we get to talking about scheduling; the next major topic. But, first, let's finish what we are doing with memory.

Local and Global Replacement. We already saw the idea of local replacement and global replacement in the cache. To refresh your memory, if we have chosen the LRU algorithm, in a global replacement policy, the least recently used page anywhere in the cache (memory) will be replaced. If we chose a local page replacement policy, it is the least recently used page that belongs to the current process.

Local is not affected much by our allocation concerns, so we can more or less leave that alone.

Suppose we choose global allocation. We will see the number of pages allocated to each change over time. If we assigned a colour to each process, it might be a bit like watching a little simulation of fictional countries as they fight over land, gaining and losing territory. Over time, a global replacement algorithm means processes that run more often will accumulate more pages in memory as they will acquire more pages than get taken from them.

It might make sense to watch to see how many frames each process has. It may be desirable to prevent a process from falling below the minimum number of pages, or to swap completely to disk a process that no longer meets that threshold.

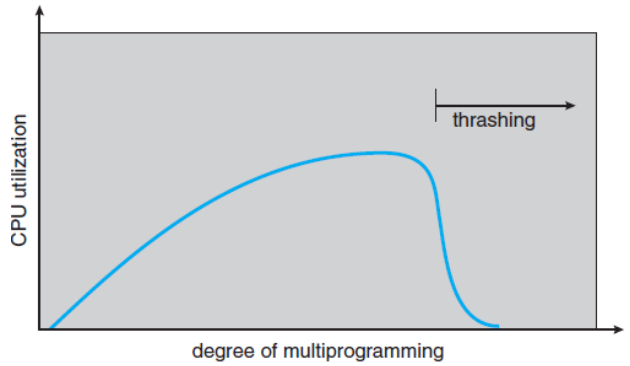
Thrashing

Thrashing received a little bit of mention in the introduction to virtual memory, but it deserves some further consideration. The quick definition of thrashing still applies: the operating system is spending so much time moving pages into and out of memory that no useful work gets done. Aside from intentionally depriving the system of RAM (as in the NeXTStep scenario), how can we get into this state, and how can we get out of it?

Let's consider a simple example from [SGG13]. In simple operating systems, the logic that controlled how many processes to run at a time would rely just on the CPU utilization. If CPU utilization is low, the CPU needs more work to do! Assign it more work by starting or bringing more processes into memory. The global page replacement policy is used here, so when a process gets a page fault, it takes a frame from another process. Under most circumstances, this works just fine.

This situation is not obviously wrong until one process starts to have a lot of page faults. This is not unreasonable; a compiler might be finished with reading and parsing the input files and moving to generation of binary code, for example, which requires a whole bunch of new instructions pages, plus the pages for the output. When this process does so, it starts taking pages from other processes. The victim processes need the pages they had, so when they get a turn to run, they too start generating page faults. So more and more requests are queued up for memory writes and reads, so the CPU is not very busy. Here's the fatal mistake: seeing that the CPU is not very busy, the OS schedules **more** programs to run.

A new process getting started will need at least the minimum number of pages to get started. These have to come from somewhere, so they will necessarily come from the pages currently belonging to other processes. This causes more page faults and more time spent paging and lower CPU utilization... prompting the OS to start more processes. No more work is getting done, because the system spends all its time moving pages into and out of memory, thrashing all around, acting like a maniac[8].



Thrashing [SGG13].

The solution is simple, actually; to increase CPU utilization we need to stop the thrashing which means we need fewer programs in memory at a time. What this really tells us is that CPU usage alone is not a sufficient indicator of whether more or fewer processes need to be running right now. It also matters *why* the CPU utilization is low. Another reason that might cause low CPU usage is, as you may recall, deadlock.

What if we stop using a global replacement policy and instead force local replacement? This limits some of the damage; if Process P_1 is thrashing,it cannot steal pages from P_2 and we do not get a cascade where all processes are constantly fighting over who has how many pages. But, if P_1 is spending all its time paging to and from disk, any other process that wants to use the disk (whether to swap or to interact with files) is going to have to spend more time waiting for the disk to be free.

We could decide whether we need to start or suspend programs not just based on CPU usage, but also on the number of page faults that occur in a given period of time. If there are too many page faults, it indicates we have too much going on. That is a reactive solution, however, and might only engage after a fair amount of time and effort has been wasted in thrashing. It would be better to be proactive and deal with this before thrashing has started.

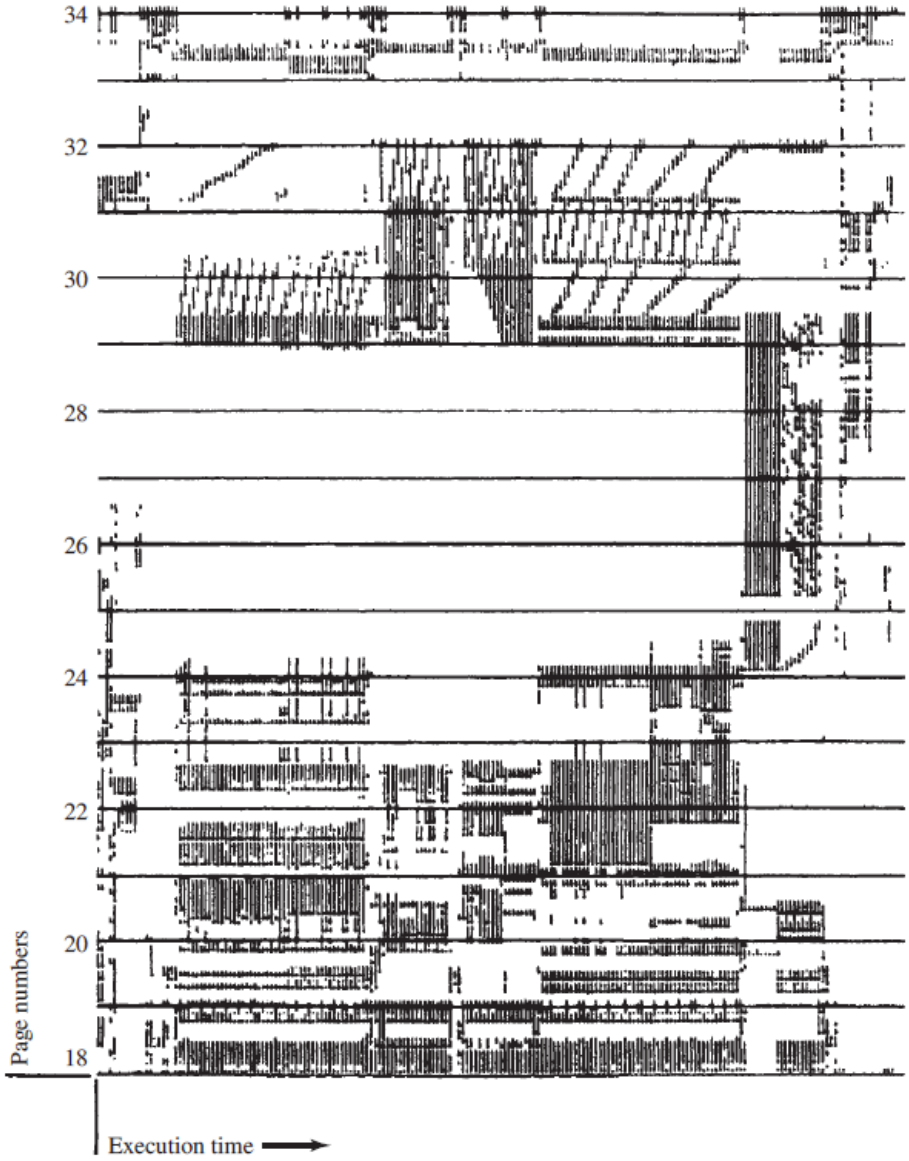
As is typical, a bit of clairvoyance would help here: how many pages is the process going to need and when will it need them? Unfortunately, there is no good way to know or to figure this out. So we will have to rely on (educated) guessing. You may hate the idea of guessing. As a general engineering practice, "let's do an experiment and find out" is good, provided you can analyze the result and if things go horribly wrong, no damage is done. Those last conditions make it kind of infeasible to do medical experiments just to see what would happen, or for NASA to play around with satellites in orbit. So in those cases, a simulation is probably the better approach. Even so, sometimes we don't have (or can't get) the data we need and we will therefore need to "make our best guess". But we are digressing.

What do we know about process memory accesses? They tend to obey the principle of locality, both temporal and spatial. Recall this from the discussion about caching. We could think of different parts of the program as different localities, as if they were areas on a map. And localities may overlap sometimes. So what we would like

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

to do, then, is give a process enough frames for its locality. If we do so, it can operate in this little area without encountering (too many) page faults. If we give insufficient pages, the process will be thrashing. Eventually it will leave the locality and that will result in some more page faults, but this is generally unavoidable [SGG13].

Before coming up with a solution that relies on locality, it is a good idea to check that the principle of locality holds. Fortunately we do not have to verify this for ourselves; someone else has already done so:



Graph showing the locality of memory accesses – 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

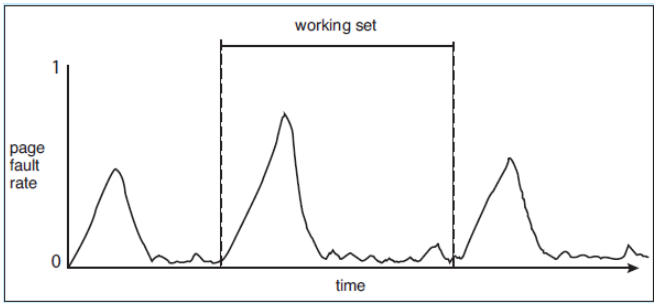
ten accesses, the working set will contain only k. So the size of the working set will change over time and can be anywhere from 1 to Δ pages.

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.

Bibliography

[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/.

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

[HZMG15] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2015. Online; version 0.15.08.17.

[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.

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

[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.