Notes on Operating Systems - Three Easy Pieces

Alex Pan

September 29, 2025

Contents

Introduction

Von Neumann model of computing: fetch instruction $\rightarrow$ decode instruction $\rightarrow$ execute instruction.

The Operating System is a piece of software that makes it easy to run programs. It does this through **virtualization** - taking a physical resource and transforming it into an easier-to-use virtual version of itself.

Programs interact with the OS through an API of **system calls**.

The OS has the responsibility to manage and distribute resources.

1.1 Virtualizing the CPU & Virtualizing Memory

The OS virtualizes the CPU, making programs believe they have their own CPU.

The OS uses policies to determine which process gets a resource at what time.

The OS virtualizes memory, making giving programs their own virtual address space.

Virtual address spaces are usually randomized to protect against stack smashing attacks.

1.2 Concurrency

Not much to say (yet!!!!).

1.3 Persistence

The **file system** is the part of the operating system that is responsible for managing persistent files.

The disk is not virtualized for every application, since applications usually want to share files between each other.

Journaling and **copy-on-write** are protocols that ensure that the state of a write to disk can be recovered even if the system crashes.

1.4 Design Goals

Abstraction as a concept means to do something without doing its underlying implementation, like writing a C file and not needing to write assembly, or writing assembly and not needing to write binary.

Operating systems need to perform tradeoffs for performance, protection, reliability, energy-efficiency, security, and mobility.

1.5 Some History

In the beginning, only one process could run at a time, and the user decided when to run each process. This is called batch processing.

User applications run in **user mode**, and can enter **kernel mode** through system calls. When a system call is made, a **trap** instruction is called, and the hardware transfers control to a **trap handler**, which starts kernel mode. When the operation is done, the OS passes control back to the user with a **return-from-trap** instruction, and kernel mode is stopped.

Multiprogramming is when the OS loads a number of processes into memory and switches rapidly between them, giving the illusion of running multiple processes at once.

1.6 Summary

Processes

A **process** is a **running program**. A program is just a bunch of lines of code that can be run.

An OS creates an illusion of each process having its own CPU by **virtualizing** the CPU through a technique called **time sharing**, where CPU time is independently shared between each running process.

Policies are algorithms that the OS uses to make decisions. The OS uses a **scheduling policy** to determine which process to run on the CPU.

2.1 The Abstraction: A Process

The **machine state** of a process is what a process can read or update. Components are:

- Memory, the data that a process can read and write is called its **address space**
- Registers, importantly the program counter, and stack and frame pointers.
- I/O information, a list of files the process currently has open.

2.2 Process API

Any operating system must implement some kind of process API:

- A way to **create** a process.
- A way to **destroy** a process. Most processes run normally to completion and exit themselves, but a user or the OS might want to kill a running process.
- A way to **wait** for a process to stop running.
- Some **miscellaneous controls**, such as pausing and resuming a process.
- A way to get information about the **status** of a process, such as how long it has run for, it state, and resources used.

2.3 Process Creation: A Little More Detail

To run a program, the OS has to **load** its code and static data from **disk** into its designated address space. Code and static data is stored in disk as a binary **executable**.

Eager loading is when the entire program's code is loaded at once, **lazy** loading is when code is only loaded as needed during program execution.

After loading a program's code, the OS needs to allocate and populate the process' **run-time stack** (stack). It may also allocate memory for a program's **heap**, which may grow as the process requests memory.

Additionally, the OS initializes the process' three **file descriptors** - stdin, stdout, stderr.

Finally, the OS jumps to the `main()` function of the program, and control of the CPU is given to the process.

To summarize, an OS starts a process by:

1. Loading code and static data from disk.
2. Allocating the process' stack and heap, and populating the stack.
3. Initializing the process' stdin, stdout, stderr.
4. Running the `main()` function of the process.

2.4 Process States

At any moment, a process can be in one of three states:

- When a process is **running**, its instructions are currently begin executed by the processor.
- When a process is **ready**, it is ready to run but the OS has chosen not to run it.
- When a process is **blocked**, it has performed an operation that makes it not ready to run until some other event takes place, like allocating memory or file I/O.

When a process is **scheduled**, it is moved from ready to running. Likewise, when it is **descheduled**, it is moved from running to ready. When some blocking operation is called, the process moves from running to blocked, and when the operation finishes, the process moves from blocked to ready.

To use these words, a process that is running can be descheduled to allow another ready process to be scheduled and start running. If process A blocks, then while the relevant device (RAM, disk, etc.) is processing its request, the OS can schedule process B, which is ready. When process A unblocks, the OS can deschedule process B and schedule process A again.

2.5 Data Structures

The OS maintains some kind of **process list**, with all processes and their information (**process control block (PCB)** or **process descriptor**). In a very simple operating system, this information includes **register context**, the registers of the CPU of the process at its last executed instruction, its process state, pointers to its memory or kernel stack, and more.

When a process is finished, it may end up in a **zombie** state, before it is cleaned up and deleted forever, which allows other processes to examine the return code of the process.

2.6 Summary

Process API

UNIX systems allows processes to create other processes with `fork()` and `exec()`, and to wait for a process' completion with `wait()`.

Every process has a **process identifier**, or **PID**.

3.1 The `fork()`, `wait()`, and `exec()` System Calls

`fork()` creates an almost exact copy of the calling process. The main difference is that the result of the `fork()` call in the parent process is PID of the child process, and in the child process it is 0.

Not stated in the chapter, but the child process has virtually the same stack, heap, and page tables as the parent. In reality, no information is duplicated until the child tries to write something to memory. When this happens, the OS uses copy-on-write to duplicate modified pages.

`wait()`, when called in a parent process after fork, blocks it from running until the child process finishes executing.

`exec()` takes in an executable and arguments, loads code and static data from the executable, and overwrites the process' current code segment, as well as its stack, heap and other parts of its memory. It effectively replaces the current process with the process passed into `exec()`.

3.2 Why? Motivating the API

The combination of `fork()`, `wait()`, and `exec()` is very powerful - after all, everything in UNIX is written with these.

The shell works by taking a user input, calling `fork()` to create a child process, and calling `wait()`. The child process then runs `exec()` on the user input, becoming the requested process. When the child exits, the shell's `wait()` blocker gets lifted, and is ready to take user input again.

Open file descriptors are kept across `exec()` calls. This allows for UNIX's redirection. In the following code:

```
close(STDOUT_FILENO);
open("out.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
execvp(proc);
```

The parent process' stdout is closed, and `out.txt` is opened. Since the stdout file descriptor is free, `out.txt` takes stdout's place, and the output of `proc` will be redirected to the `out.txt`.

3.3 Process Control And Users

The `kill()` system call sends **signals** to a process. <C-c> in the terminal sends SIGINT (interrupt), and <C-z> (stop) sigmal pauses the process.

Processes can use the `signal()` system call to catch signals.

3.4 Useful Tools

Go read the manuals of `ps, top, kill, killall`.

3.5 Summary

`fork()` has problems, and some researchers advocate for `spawn()` to start new processes.

Direct Execution

How do we virutalize the CPU without adding excessive overhead to the system?

4.1 Basic Technique: Limited Direct Execution

Direct execution is a protocol where a list of processes is maintained, and the OS iterates through the list and runs each process serially until completion.

4.2 Problem #1: Restricted Operations

To prevent processes from doing whatever they like, most processes run in **user mode**, where they are restricted in what they can do.

The OS runs in **kernel mode**, and it can do whatever it likes.

When you do something like `open()` in C, it is a procedure call, and because it has a trap instruction it becomes a system call.

When executing a trap in x86, the processor pushes the program counter and a few other registers onto a per-process **kernel stack**, and when return-from-trap is called, these are popped and normal execution is resumed.

To prevent processes from running arbitrary code in kernel mode, the OS kernel sets up a **trap table** of **trap handlers** at boot time, which says what code should run when a system call or a hardware interrupt occurs.

The **trap** instruction saves register state to the kernel stack, changes hardware status to kernel mode, and jumps to the trap table.

The **return-from-trap** instruction pops registers from the kernel stack back into the CPU, changes hardware status to user mode, and jumps back to the process' current program counter.

The instruction to tell hardware where trap tables are is a privileged operation.

System calls are identified with **system-call numbers**. Users must specify a system call number to use, and thus cannot execute privileged instructions by jumping to some address.

Limited direct execution is a variant of direct execution. At boot, the OS initializes a trap table. When a process is queued:

1. The OS inserts the process into the process list, allocates memory, sets up the stack and program counter, and inserts the program counter and other registers into the kernel stack. Runs return-from-trap

2. The hardware pops from the kernel stack, moving the process' registers to the CPU, switches to user mode, and executes executes `main()`.
3. The process performs a system call, and runs trap.
4. The hardware saves the process' registers to the kernel stack, switches to kernel mode, and jumps to the trap handler (from the trap table created at boot).
5. The OS handles the trap, and runs return-from-trap.
6. The hardware pops from the kernel stack, moving the process' registers to the CPU, switches to user mode, and jumps to the current PC.
7. Steps 3-6 are repeated while the process runs.
8. Process runs `exit()`, and traps back to the OS.
9. OS frees the process' memory, and removes it from the list.

Importantly, whenever the process runs a system call, important registers are saved to the kernel stack, the system call is performed, and the registers are popped back into the CPU.

4.3 Problem #2: Switching Between Processes

With limited direct execution, only one process can run at a time, meaning even the OS itself can't run. This is a big issue if the currently running process takes very run to long or infinitely loops.

The **cooperative** approach is where the OS trusts processes to give up control of the CPU periodically with a **yield** system call. The OS can also regain control when a process tries to do something illegal, which usually ends up in the process being killed.

Another approach is a **timer interrupt**, where a hardware device is programmed to periodically generate an interrupt, with which the OS uses to run its own instructions.

When the OS is running, it has to decide which process to continue running with a **scheduler**.

If the OS decides to switch from process A to process B, it runs a **context switch**, which saves A's registers onto A's kernel stack, pops B's registers from its kernel stack into the CPU, changes the kernel stack pointer to B's kernel stack, and starts running B.

4.4 Worried About Concurrency?

What happens if an interrupt occurs while another interrupt is being processed?

One approach the OS can take is to disable interrupts during interrupt processing, but this could lead to interrupts being lost.

Another approach is **locking** schemes, which we learn about later.

4.5 Summary

Limited direct execution is like "baby proofing" a room - getting rid of anything that might hurt the baby, and to do anything remotely dangerous the baby has to get it through its caretaker.

In my opinions, this analogy isn't that good - we don't want to kill the baby if they step into the neighboring baby's room.

CPU Scheduling

This chapter is about various scheduling policies that the OS may use to determine which processes get to run when.

5.1 Workload Assumptions & Scheduling Metrics

The **workload** is the set of processes, or **jobs**, currently running in the system.

To begin, the following assumptions are made.

1. Each job runs for the same amount of time
2. All jobs arrive at once
3. Once started, each job runs to completion
4. All jobs only use the CPU
5. It is know how long each job runs for

A **scheduling metric** gives a measurable quantity for a desirability of a given scheduling algorithm.

Turnaround time is one possible scheduling metric, which measures how long passed between a job arrived and was completed. In formal terms, $T_{turnaround} = T_{completion} - T_{arrival}$

Turnaround time is a **performance** metric.

Response time is another scheduling metric, defined as difference between when the job arrives to the first time it is scheduled. In formal terms, $T_{response} = T_{firstrun} - T_{arrival}$.

Another metric is **fairness**, which is usually inversely related to performance.

FIFO, SJF, STCF address turnaround time, and Round Robin addresses response time and fairness.

5.2 First In, First Out (FIFO)

The first process that arrives is the first process that gets ran to completion.

This strategy is optimal with all assumptions in place.

This policy falls apart when assumption 1 is dropped, and the first job queued is very long - this is called the **convoy effect**.

5.3 Shortest Job First (SJF)

The shortest job is run first, then the next shortest, and so on.

This strategy is optimal with assumptions 2, 3, 4, and 5 in place.

5.4 Shortest Time-to-Completion First (STCF)

The process with the shortest time to completion is always ran first. The OS can have many different times where it decides to switch jobs, whether it be through triggered interrupts from new jobs arriving, or a timer interrupt.

This strategy is optimal with assumptions 3, 4, and 5 in place.

5.5 Round Robin

Fix some duration t. Round robin goes through its list of processes, runs process 1 for t, then runs process 2 for t, and so on, until it cycles back to process 1.

t is also known as a **time slice**.

If the time slice is too small, then the total cost of context switches will become more and more expensive.

Round Robin performs extremely well under the response time metric, but extremely poorly under the turnaround time metric. This is an example of fairness being inversely related with performance.

5.6 Incorporating I/O

A scheduler should try to **overlap** process blockers like I/O as much as possible with other processes - when a process is blocked, the scheduler should try to run another process.

5.7 No More Oracle and Summary

The last assumption, that the scheduler knows how long each job runs for, is the most unrealistic one. SJF and STCF rely on this, so they aren't practical in the real word. The **multi-level feedback queue**, introduced in the next chapter, uses information from past jobs to try to predict the future.

Multi-level Feedback

The **Multi-level Feedback Queue (MLFQ)** is a scheduling policy that won its creators Corbato et al. the Turing Award. It optimizes turnaround time and response time for interactive processes with realistic assumptions of the kinds of jobs it will receive. Variants of the MLFQ are implemented in modern operating systems.

6.1 MLFQ: Basic Rules

The MLFQ has a number of distinct, disjoint **queues** (more like lists), each assigned a different **priority level**. At any given time, the queue with the higher priority level runs its jobs in round-robin.

This can be summarized into two rules:

- **Rule 1:** If Priority(A) > Priority(B), then A runs and B doesn't.
- **Rule 2:** If Priority(A) = Priority(B), then A and B run in RR.

The priority level of a process is determined based on how often it blocks, or stops using the CPU. This is because interactive processes are more likely to frequently block than non-interactive processes.

Importantly, the priority of a process constantly updates based on its activity in recent time slices. Otherwise, only processes of the highest priority level would run, and nothing else would.

6.2 Attempt #1: How To Change Priority

A job's **allotment** is the amount of time a job can spend at a given priority before the scheduler reduces its priority.

- **Rule 3:** When a job enters the system, it is placed at the highest priority.
- **Rule 4:** If a job uses up its allotment while running, its priority is reduced. If a job gives up the CPU before using up its allotment, its priority stays the same.

Examples are long to describe, but this current ruleset properly prioritizes short and interactive jobs, which is good for turnaround time (SJF aspect), as well as response time (RR aspect).

When a process is added, it is assumed to be short, and as it runs longer, its priority drops. By prioritizing high-blocking processes, a high amount of overlap is achieved as well.

This current ruleset has issues: if there are too many interactive jobs, then non-interactive jobs will **starve**, since they will never receive CPU time. Malicious programs (or a user's own) can be written to keep their priorities high to hog CPU time, and if a program starts as non-blocking but changes, then it has no way to increase its priority.

6.3 Attempt #2: The Priority Boost

To solve the last problem of process behavior changing over time, we have a new rule:

- **Rule 5:** Periodically move all jobs in the system to the highest priority.

Now, processes are guaranteed not to starve, and if a process changes behavior to being interactive, it has a chance to stay at a high priority again.

6.4 Attempt #3: Better Accounting

To solve the issue of gaming the scheduler into letting a process hog CPU time, we rewrite rule 4:

- **Rule 4:** If a job uses up its allotment at a given level, its priority is dropped.

This might seem like it completely ignores the importance of interactivity, but it doesn't. This is because interactive jobs take up less CPU time and more time blocked, so they lose priority slower than CPU-intensive jobs.

6.5 Tuning MLFQ and Other Issues

To actually implement the MLFQ, a lot more questions need to be answered, such as the number of priority levels, time slice sizes, allotment, period of priority reset, etc.

High priority jobs should have small time slice sizes, since they will overlap a lot. Low priority jobs should have longer time slice sizes, since context switching is expensive.

6.6 Summary

MLFQ can deliver reasonably both fast turnaround times and fast response times. A base implementation is defined by the following rules:

- **Rule 1:** If Priority(A) > Priority(B), then A runs and B doesn't.
- **Rule 2:** If Priority(A) = Priority(B), then A and B run in RR.
- **Rule 3:** When a job enters the system, it is placed at the highest priority.
- **Rule 4:** If a job uses up its allotment at a given level, its priority is dropped.
- **Rule 5:** Periodically move all jobs in the system to the highest priority.

Lottery Scheduling

Proportional-share, or **fair-share** tries to guarantee that each job obtains a certain percentage of CPU time.

Lottery scheduling periodically holds a lottery to determine which process gets to run. More important processe have a higher chance of winning.

7.1 Basic Concept: Tickets Represent Your Share & Ticket Mechanisms

The more tickets a process has, the more often it wins the lottery. Users can be given tickets to allot to processes.

Processes can initiate **ticket transfers** to other processes to pass off CPU time. This is useful in client/server settings, where a client can tranfer tickets to the server to do work.

In a system where processes trust each other, **ticket inflation** can be really effective if a process accurately declares when it needs more or less CPU time by increasing or decreasing its ticket count.

Randomness has no worst case, is easy to implement (!), and has very little overhead.

7.2 Implementation & An Example

Implementation can be done with a linked list of processes, iterating through them, and subtracting ticket counts from a randomly generated number in $[0, totalTickets)$, and choosing the process where the number hits 0.

To have the least number of iterations, the list can be sorted so the jobs with the most tickets are at the front.

If two jobs have the same run time and ticket count, then the longer the run time, the more fair lottery scheduling is (common theme from probability).

7.3 How To Assign Tickets

One approach is to let the user assign tickets. However, the "ticket-assignment problem" remains open.

7.4 Stride Scheduling

Stride scheduling is a deterministic variant of lottery scheduling. A process' **stride** is inversely proportional to the number of tickets it has. A process' **pass value** is initially set to 0, and the OS maintains a priority queue to always run the job with the lowest pass value. After running, a job's pass value is incremented by its stride. So, if a process has a low stride, then its pass value grows slower, and it is run more often.

Stride scheduling struggles if jobs arrive at different times. Then, newer jobs will hog the CPU until their pass value surpasses a longer-running job. Lottery scheduling doesn't struggle with this.

7.5 The Linux Completely Fair Scheduler (CFS)

Operating systems must try to reduce scheduling overhead as much as possible. Linux's **Completely Fair Scheduler** achieves fair-share scheduling in a highly efficient and scalable manner.

Following is a very very condensed summary of the implementation described in the textbook. Go read the textbook and its linked research articles for a real understanding.

As a process runs, its `vruntime` increases, and whenever a time slice finishes, CFS picks the process with the lowest `vruntime` to run next. Two parameters `sched_latency` and `min_granularity` control the sizes of time slices. Users can add weights to processes to increase or decrease their time slice size, and `vruntime` ratio. A red-black tree is used to store running jobs for efficient lookups. If a process blocks for a long time, then CFS sets its `vruntime` to the minimum `vruntime` found in the tree to prevent starvation of other jobs.

7.6 Summary

Different schedulers work well for different systems - the best one for a PC will not be the best one for a data center.

Multi-CPU Scheduling

Address Spaces

9.1 Early Systems & Multiprogramming and Time Sharing

In early machines, two processes resided in memory: the OS (which wasn't even a process at the time, more just a loaded library), and the currently running process.

Over time, more and more processes needed to be run, so **multiprogramming** and later **time sharing** as **interactivity** became important.

One approach to time sharing from the memory perspective is to run a process for a bit, save all of its memory to disk, load another process' memory from disk, and run that new process. This is obviously really slow. Operating systems had to figure out a way to let multiple programs reside in memory at once.

Memory protection becomes an important issue, so that processes can't read or write some other processes's memory.

9.2 The Address Space

The **address space** of a process is the abstraction and virtualization of physical memory. It contains all of a process' code, stack, heap, static variables, and everything else specific to the process.

Importantly, the address space is not a direct mapping onto physical memory - it may or may not be discontinuous. If a process tries to load from virtual address `p`, the OS translates `p` to a physical address probably not equal to `p`.

9.3 Goals

Virtual memory should be implemented in a way that is invisible to the program - the program should behave as if it has its own private physical memory.

Virtual memory should also be efficient and protected, for performance and security reasons.

9.4 Summary

The OS gets some dedicated hardware to help translate virtual addresses into physical addresses.

Note: when `malloc()` is ran, space is only reserved in the virtual address space, physical memory is only allocated when the corresponding page is accessed.

Memory API

Since this chapter is about memory management in C, which I have a lot of experience with already, notes will be very short and sparse.

Remember to allocate an extra byte for strings.

Buffer overflows happen when you write past an allocated buffer. Scarily, these might not cause the program to error if the OS doesn't detect an invalid write - it might've overwritten some of a process' own data.

`malloc()` and `free()` are library calls, not system calls. The malloc and free libraries manage virtual address space, and are built on system calls themselves. One such system call is `brk`, which is used to change the process' **break**, the location of the end of the heap.

`mmap()` creates a memory region associated with swap space.

`malloc()` allocates a region of memory, with some metadata containing information about it at the front. The returned pointer points to right after the metadata, where the actual user's data starts. This is why free() doesn't need to take in a byte count, and also why it must be called on the original pointer.

Address Translation

Hardware provides **address translation**, which quickly and efficiently translates virtual addresses into physical addresses.

This is only part of the full implementation the OS uses to abstract and manage memory.

11.1 Assumptions

For now, assume that:

1. Process' address space maps continuously onto physical memory.
2. Address space is strictly less than the size of physical memory.
3. Every process' address space is the same size.

11.2 An Example & Dynamic (Hardware-based) Relocation & Hardware Support: A Summary

Say a process has an address space that starts at 0 and ends at 16KB. However, in physical memory, it occupies the region 32KB - 48KB.

Dynamic relocation, or **base and bounds**, stores the start (base) and size (bound) of a process' physical address space somewhere (in registers). The process' virtual address space starts at 0, and translation is just $address_{physical} = address_{virtual} + base$. If the resulting address is between $base$ and $base + bound$, then it is a valid address.

The two registers storing base and bound are referred to as the **memory manage unit (MMU)**. We will be adding a lot more functionality to this MMU.

A single bit on the CPU indicates whether the CPU is in user mode or kernel mode. The base and bound registers can only be modified with privileged instructions.

The CPU generates **exceptions** when a process tries to access memory illegally. The user program is paused and the exception handled by the OS' **exception handler**, likely ending up in the process being killed.

11.3 Operating System Issues

The OS maintains a **free list** to find room for newly created processes, and to update free space whenever a process is ended.

The base and bounds registers, like other CPU registers, must be saved to the process' PCB when a context switch occurs.

And OS can move around a process' address space when the process is in the ready state if needed. This is done by just changing the base register in the process' PCB.

Exception handlers functions to be called when an exception occurs, and these are initialized at boot.

11.4 Summary

Dynamic relocation as we know it right now sucks - there is a lot of **fragmentation**, since a process might not actually use all of its address space, leading to wasted memory.

Segmentation

Dynamic relocation, with the base and bounds registers, cannot support address spaces larger than physical memory, and is inefficient for the free space of a process between its stack and heap.

12.1 Segmentation: Generalized Base/Bounds

Instead of just one pair of base and bounds for the entire process, maintain a list of base and bounds for each independent memory segment (code, stack, heap, etc.) Now, a process' virtual memory doesn't have to map continuously onto physical memory. Assuming code, stack, and heap are the only three things a program needs to store, our MMU can be expanded to have 3 pairs of base and bounds registers.

If a process tries to access a virtual address that isn't in any of its segments, then it causes a **segmentation violation**, or a **segmentation fault**.

12.2 Which Segment Are We Referring To?

Two ways of figuring out which segment an address belongs to are the **explicit** and **implicit** approaches.

With the explicit approach, the segment is encoded in the first few bits. Since we have three segments, the first two bits determine which segment a virtual address belongs to.

With the implicit approach, hardware figures out what part of the process the virtual address came from. If the address is an instruction fetch, then it is in the code segment, if the address is an offset from the stack pointer, then it is in the stack segment, and otherwise it's in the heap segment.

12.3 What About The Stack?

The stack grows backwards towards lower addresses, and the heap grows towards higher addresses. So, another bit needs to be added in registers to determine whether a segment grows positively or negatively.

12.4 Support for Sharing

System designers discovered that it was efficient to share certain segments between address spaces. For example, the same physical code segment can be shared across multiple identical processes while maintaining the illusion of a private address space.

To implement this securely, more bits need to be added to the hardware to verify whether a program can read or write to the segment it's accessing.

12.5 Fine-grained vs. coarse-grained Segmentation

Splitting an address space into 3 large continuous segments as we are now is called **coarse-grained** segmentation. **Fine-grained** segmentation involves a large number of small segments. This requires a **segment table** for efficient conversions of memory to segments.

12.6 OS Support

The virtual address space of a process is still the same, the space between the stack and the heap still exist, it just hasn't actually been allocated and mapped onto physical memory yet.

The heap doesn't always grow when `malloc()` is called. If the existing heap has enough space for the requested memory then a pointer to that is returned. If not, then the memory allocation library performs the `sbrk()` system call to grow the heap. If the process' heap is already too large or if there is no more physical memory, then the OS rejects the request. Otherwise, the OS updates segment size registers, and allows more memory to be alloted.

With segmentation approaches, physical memory becomes **externally fragmented**, or filled with many small free holes that are unusuable for a large allocation.

12.7 Summary

One issue with segmentation is that they are variable-sized, since memory becomes fragmented over time.

Another issue is that segmentation can't support our ideal sparse address space. For example, if a process' heap is big but sparsely used, then the entire heap still has to be stored in physical memory, which is very inefficient.

Free Space Management

Free space management is a problem that arises in process heaps and virtualizing memory with segmentation. When variable size buffers are allocated and freed, **external fragmentation** occurs.

An example of external fragmentation is if we have a continuous region of 10 bytes free, 10 byte occupied, and 10 bytes freed. 20 bytes are free, but a process cannot allocate 20 bytes, since the free regions are separated.

An **allocator** is a mechanism that hands out and reclaims chunks of memory on request. They exist at multiple levels - process-level allocators manage the process' own heap, kernel-level allocators manage pages. This chapter is about process-level allocators.

13.1 Assumptions

- A user has a simple interface to the memory, like `malloc()` and `free()`.
- We are concerned with external fragmentation. **Internal fragmentation** is when an allocator returns memory bigger than requested.
- Once a process allocates memory, nothing can move it until the process frees the memory (or the process gets killed). This means **compaction**, as defined the last chapter, can't be used.
- The allocator manages a fixed-size continuous region of bytes.

Unlike assumptions lists of previous chapters, like CPU scheduling, these are all actually reasonable besides the last one.

13.2 Low-level Mechanisms & Basic Strategies

When an allocator receives a request for memory, it can **split** a free block into two smaller blocks, and return a pointer to one of the blocks of memory.

When an allocator receives a request to free memory, it can **coalesce** neighboring free blocks into a larger free block.

Blocks of memory also contain a header, which is just metadata at the front that describes how large the block of memory is, and whether it is free or not. This is also how `free()` knows exactly how many bytes to free.

This makes the blocks of memory a linked list - to get from one block to the next, just add the size of the header and how big the current block is.

If no more space is free, then an allocator can either grow the heap by requesting more memory (`sbrk` system call), or returning `NULL`.

Here are some strategies an allocator can use. When a strategy finds a chunk, it splits it if the chunk is too big.

- The **best fit** strategy has the allocator find the smallest free chunk with size bigger than or equal to the requested memory. Performance sucks, since the entire heap needs to be searched.
- The **worst fit** strategy has the allocator find the largest free chunk. The entire heap needs to be searched again, and severe fragmentation occurs.
- The **first fit** strategy has the allocator find the first chunk large enough. Performance is good, but the front of the list becomes filled with small objects.
- The **next fit** strategy uses first fit as a subroutine, but starts each new search from where the last one left off. This makes fragmentation and object distribution more uniform.

13.3 Other Approaches & Ideas

The **segregated list** idea involves maintaining a separate list if a process frequently requests memory of a constant size. Fragmentation is less common, and a linked list isn't needed. Other sized memory requests go to a general allocator as described above. Downsides are that usefulness will vary widely between different processes, so it's hard to determine how much memory should go to the general allocator and the segregated list.

At boot, **slab allocator** puts commonly-used kernel objects into object caches, which are segregated lists with fast allocation and free times. When a cache needs more memory, it requests it from a more general memory allocator, and when it has excess free space, it submits it to the general memory allocator. In this way, often-used code can be allocated and freed quickly.

The **binary buddy allocator** recursively divides memory into two equally-sized chunks called buddies until a chunk's size is `nextpow2(requested_size)`. When a chunk is freed, coalescing is as simple as checking if its buddy is free, and this is propagated up the binary buddy tree. This suffers from internal fragmentation. Buddies are easy to find, since pointers of buddies at different levels will always differ by a specific bit.

Linked lists scale horribly, so advanced allocators use data structures like balanced binary trees, splay trees, or partially-ordered trees.

TODO: Read about how the glibc allocator works.

13.4 Summary

Introduction to Paging

Since splitting memory into variable-sized pieces results in fragmentation, we start to consider splitting memory into fixed-size pieces. This idea is called **paging**. A process' address space is split into **pages**, and physical memory is viewed as an array of **page frames**.

14.1 A Simple Example And Overview

There is an example in the book with figures and explanations.

With paging, a process' address space is split into equally-sized pages. These are called **virtual** pages, indexed by **virtual page numbers**. Physical memory is also split into equally-sized pages frames, indexed by **physical frame numbers**. Any virtual page corresponds to a physical frame, and the mapping is stored in a per-process **page table**.

Because page sizes are usually powers of 2, the page virtual page number of an address can be determined by looking at the first few bits (assuming virtual addresses start at 0).

14.2 Where Are The Page Tables Stored & What's Actually In The Page Table?

Assume for now that page tables are stored in the OS' address space (which itself can be virtualized with its own page table).

Also assume that the data structure used is just an array, with index i being virtual page i's corresponding **page table entry**. This method ends poorly if a lot of pages are needed to cover a virtual address space.

Page table entries should be small, but there are also a few very useful bits used to help achieve an address translator's goals:

- A **valid bit** that says whether the physical page has actually been allocated. This key detail allows for virtual address spaces bigger than physical memory, since an extremely large virtual address space doesn't have to actually be fully allocated.
- **Protection bits**, which say whether a page can be read or written to. If a protected page is accessed without permission, a trap is generated to the OS.
- A **present bit**, which indicates whether a page is in memory or swap space.
- A **dirty bit**, which indicates whether the page has been modified since it was brought into memory from swap space or a paging file. This is useful if the page needs to be moved back to disk - if the page is clean, then the page on disk doesn't actually have to be modified.
- A **reference bit**, which tracks whether a page has been accessed, which is useful in determining which pages actually get used.

The Intel manuals contain information about x86 paging support. The referenced document is 5000 pages of Intel's documentation about the x86 instruction set.

14.3 Paging: Also Too Slow & A Memory Trace

When a process loads from a virtual address, the virtual page number needs to be calculated, the physical location of the page table needs to be fetched, the corresponding page frame needs to be looked up, and the offset needs to be added before we finally get to a physical address. Not to mention the checks to ensure segmentation and protection faults are properly thrown. So, a lot of extra care needs to be put into hardware and software design to optimize this process.

Assuming no cache, each instruction fetch of a program generates a physical memory reference to the page table, then a physical memory reference to find the actual instruction.

When we trace the memory references of a process, we learn that some pages are a lot more frequently accessed and important than others.

14.4 Summary

Translation Lookaside Buffers

A **translation-lookaside buffer**, or **address translation cache** is part of a chip's MMU. It is a specialize cache for popular address translations.

15.1 TLB Basic Algorithm

If a virtual page is requested and it exists in the TLB, then the physical address can be returned quickly. This is called a TLB hit. Otherwise (TLB miss), the hardware gets the page table entry from memory, puts the requested virtual page number into the cache, and retries the instruction with the virtual page number already in the TLB.

Advanced Page Tables

Swapping: Mechanisms

Swapping: Policies

Complete VM Systems