Notes on Operating Systems - Three Easy Pieces

Alex Pan

October 17, 2025

Contents

Concurrency and Threads

A **multi-threaded** program has multiple program counters. Each thread is like a separate process, except they share the same address space.

Each thread has its own private set of registers, so if there are two threads on a single processor, then a context switch with **thread control blocks** is needed.

If a process has n threads, then it has n distinct stacks, and anything on a thread's stack is called **thread-local** storage.

1.1 Why Use Threads?

Threads are useful for parallelism on machines with multiple processors. It is also useful for multiprogramming within a program - if one thread is blocked by an I/O request, another thread can still continue running code.

1.2 An Example: Thread Creation & Why It Gets Worse: Shared Data & The Heart Of The Problem: Uncontrolled Scheduling

A thread is created with `pthread_create()`, and synchronized with `pthread_join()`.

Thread execution order is nondeterministic, and up to the OS scheduler.

An example of how things could possibly go wrong with two threads incrementing a number in an address in memory:

1. Thread A loads the value at the address into a register
2. Thread increments the value in the register
3. The OS stops thread A from running, saving its registers
4. Thread B loads the value at the address into a register
5. Thread B increments the value in the register
6. Thread B writes the incremented value back into memory
7. Thread A writes its incremented value back into memory

In this process, the value in memory was effectively only incremented once, since B finished writing back to memory before A could finish. This is called a **data race**, a type of **race condition**.

This kind of code is called a **critical section**, where multiple threads access a shared variable at the same time. What we want is **mutual exclusion** - if one thread is running the critical section, no other thread should be.

1.3 The Wish For Atomicity

Atomic instructions, or blocks of code, are guaranteed to not be interrupted in the middle of their execution. Only the most important atomic instructions are supported, so we need to use **synchronization primitives** for our more specialized atomic blocks of code.

1.4 One More Problem: Waiting For Another

Another common theme in concurrent programming is communication between threads. For example, if a thread is waiting for some data in the address space to become available, then it goes to sleep instead of looping endlessly and wasting CPU, and another thread wakes it up when the data is available.

1.5 Summary

Thread API

2.1 Thread Creation

`pthread_create()` documentation or `man 3 pthread_create()`

```
#include <pthread.h>

int pthread\_create(pthread\_t *restrict thread,
                   const pthread\_attr\_t *restrict attr,
                   typeof(void *(void *)) *start\_routine,
                   void *restrict arg);
```

Arguments:

- `thread` is a pointer to a structure used to interact with the thread.
- `attr` is a pointer to a structure used to specify any attributes a thread might have, like the stack size or scheduling policy of a thread.
- `start_routine` is a function pointer to what the thread will execute, and
- `arg` is the only argument into `start_routine`. This is commonly packaged into a struct.

Returns 0 if successful, returns error number if error.

2.2 Thread Completion

`pthread_join()` documentation or `man 3 pthread_join()`

```
#include <pthread.h>

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

Arguments:

- `thread` specifies the thread to wait for.
- `retval` is a pointer to the return value expected from the routine ran by the thread.

Be careful about not returning a pointer to something on the thread's stack, though this also applies to any function.

2.3 Locks

`pthread_mutex_lock` documentation, or `man 3 pthread_mutex_lock`

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

```
int pthread\_mutex\_lock(pthread\_mutex\_t *mutex);
int pthread\_mutex\_unlock(pthread\_mutex\_t *mutex);
```

A lock can be initialized with `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER`, or `pthread_mutex_t lock; assert(pthread_mutex_init(&lock, NULL) == 0);`, where `NULL` can be replaced by an optional set of attributes.

`pthread_mutex_destroy()` needs to be called on the lock if allocated with `pthread_mutex_init()`, otherwise (with macro) not needed.

A thread can claim a lock with `pthread_mutex_lock()`, and free a lock with `pthread_mutex_unlock()`. No other thread can claim a lock when another thread has it, and will wait until the lock is free.

Good practice to use wrappers around lock calls, since they can fail silently (return a bad value, but not crash the program).

Two other lock functions:

```
int pthread\_mutex\_trylock(pthread\_mutex\_t *mutex);
int pthread\_mutex\_timedlock(pthread\_mutex\_t *mutex,
struct timespec *abs\_timeout);
```

`pthread_mutex_trylock()` returns failure if the lock is already held, and `pthread_mutex_timedlock()` returns after a timeout or the lock succeeds, whichever happens first.

`pthread_mutex_timedlock(&lock, 0)` is equivalent to `pthread_mutex_trylock(&lock)`, since if the lock is available, both return as expected, and if the lock isn't then both return an error code.

2.4 Condition Variables

Condition variables are useful when a thread wants to signal something to another.

`pthread_cond_timedwait` documentation or `man 3 pthread_cond_timedwait`

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);
```

To use a condition variable, a thread must also have a lock associated with the condition.

`pthread_cond_wait()` sleeps the calling thread, and waits for some other thread to signal it.

For example, if a thread runs:

```
// in a scope visible to other threads
// pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
// pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

And another thread runs:

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

The first thread waits for a signal from the second thread, being the `ready` variable.

`Pthread_cond_wait(&cond, &lock);` releases the lock for other threads to acquire it, but reclaims it when the signal is received.

The "sleeping" thread checks for the condition in a while loop.

2.5 Compiling and Running & Summary

Compile with `-pthread`, and everything here is in `pthread.h`.

`man -k pthread` to see entire pthread interface.

Locks

3.1 Locks: The Basic Idea & Pthread Locks

Locks are either **free** or **held**. When a thread tries to call `lock()` on a free lock, it instantly acquires the lock and continues running the code, and if the lock isn't free, it waits until the lock is free before acquiring the lock and continuing with code. A lock can be `unlock()`ed by a thread.

Threads, like processes, are also scheduled by the OS.

A lock in POSIX is called a **mutex** because it provides mutual exclusion between threads. Locks are declared as variables, allowing multiple locks to exist.

3.2 Building A Lock & Evaluating Locks

The OS gets some hardware primitives to help implement locks.

An implementation of a lock is graded on three factors: whether it works, fairness, and performance. Fairness measures how equal each contending thread's chances of acquiring a free lock are. A low fairness factor can imply thread starvation. Performance measures the time overhead of using locks. Important cases are when there is only one thread, when there are multiple threads on a single processor, and when there are multiple threads on multiple cores.

Following are multiple implementations of locks.

3.3 Controlling Interrupts

If on a single-core system running multiple threads, lock can be implemented by just disabling interrupts inside of a block of code. Then, the scheduler can't stop the thread to run other threads.

There are a few downsides:

- Disabling interrupts is a privileged operation, meaning malicious code can just permanently stop anything from interrupting it.
- This doesn't work on multiprocessors, since other threads will run even if interrupts are disabled.
- Turning off interrupts for too long can lead to interrupts being lost.

This implementation is only really viable for within kernel use, since it can be trusted to avoid all of these.

3.4 A Failed Attempt: Just Using Loads/Stores

Another idea is to just have a global flag - when a thread calls `lock()` and the flag is 0 (free), then the thread sets the flag to 1 and continues - otherwise the thread **spin-waits** in a while loop until the flag is free again. A flag is freed by calling `unlock()` and setting the flag to 0. This approach is called a **spin-lock**.

This approach also doesn't work - if thread A calls `lock()`, sees that the flag is free, and while issuing a write to memory, another thread B calls `lock()` as well, sees that the flag is free before A can finish claiming the lock.

It also isn't performant - a lot of CPU cycles are wasted on spin-waiting.

3.5 Building Working Spin Locks with Test-And-Set & Evaluating Spin Locks

* Genuinely don't understand how Peterson's algorithm works

The **test-and-set** operation takes a pointer and a value, and atomically fetches the old value at the pointer, sets it to the new value, and returns the old value.

This operation is guaranteed to be atomic by hardware, using some mechanism like cache coherence protocols or bus locking.

This instruction allows the spin-lock approach to work, since previously, it was possible for one thread to read the flag while another thread was writing to it, but now it isn't because the operation is atomic.

This implementation of the spin lock is correct, but doesn't guarantee no starvation.

3.6 Compare-And-Swap

The **compare-and-swap** operation takes a pointer, an expected value, and a new value. If the value of the pointer is the expected value, then test-and-set is run, setting the value of the pointer to the new value and returning the old one.

This is a more powerful version of test-and-set, meaning it can implement the spin-lock, as well as **lock-free synchronization** later.

3.7 Load-Linked and Store-Conditional

Load-linked and **store-conditional** are instructions used together to build locks.

Load-linked acts exactly like a normal load instruction, but also annotates the address it loads from. Store-conditional takes an address and a value, and if the value of the address hasn't been modified since the last load-linked, it updates the value. Otherwise, return false.

To acquire a lock, a thread spin-waits with a condition of `LoadLinked()` on a flag until it is

free. Then, if `StoreConditional()` is successful, then the thread has the lock. Otherwise, it keeps spinning because another thread's store-conditional has already modified the value.

3.8 Fetch-And-Add

Fetch-And-Add is an instruction that takes in a pointer and atomically increments its value. This can implement locks by having a global turn number, and when a thread wants a turn, it atomically increments the global turn number for its turn. When the turn number is equal to the thread's turn, then it runs.

This approach is guaranteed to be fair.

3.9 Too Much Spinning: What Now?

On single-core CPUs, threads without the lock will just spin for their entire time slices, wasting CPU time. On multi-core CPUs, as long as the critical section is short, there is not much overhead.

3.10 A Simple Approach: Just Yield, Baby

In single-threaded systems, the main thing we have to worry about is if a thread gets interrupted while in the critical section, forcing all of the other threads to wait until it is un-interrupted.

Instead of just spin-waiting, threads waiting for locks can try to call a `yield()` primitive provided by the OS, giving up the CPU. This solves the wasted cpu cycles problem, but context switches are still expensive. Additionally, a thread can still be starved.

3.11 Using Queues: Sleeping Instead Of Spinning

If a thread requests a lock that isn't available, it can be added to a queue, then slept. When a thread unlocks the lock, it removes the next thread in the queue and wakes it up. The only time spin-waiting will happen now is if a thread gets interrupted while it is acquiring a lock.

3.12 Different OS, Different Support

Linux's **futex** is a system call used for kernel-independent synchronization. Here is the documentation. The higher-level mutex functions are implemented using futexes. It provides two functions:

- `futex_wait(address, expected)`, which sleeps the calling thread until address is not equal to expected.
- `futex_wake(address)` wakes one (some specified number) thread in the `FUTEX_WAIT` queue corresponding to the address.

3.13 Two-Phase Locks

Linux's futex is an example of a two-phase lock, where after calling for a lock, a thread spins for a bit in case it gets it quickly (phase 1), then goes to sleep (phase 2).

3.14 Summary

Locks today are built with some specialized hardware primitives, and low-level system calls provided by the OS, like Linux's futex.

Locked Data Structures

Adding locks is one way to make a data structure **thread safe**, allowing it can experience the gains of concurrency while still being correct.

Some of these notes will be short because of my background in parallel programming.

4.1 Concurrent Counters

When doing concurrency, **perfect scaling** is achieved something runs n times faster with n threads.

A counter is a counter, just a variable to increment and decrement. A trivial concurrent approach with two threads one lock is two orders of magnitude slower than just one thread.

An **approximate counter** is implemented with every thread having a local counter, which gets periodically gets atomically added to the global counter. If the period is short, then there is more overhead but the global counter is more accurate, and if the period is long, then there is less overhead but the global counter is less accurate.

4.2 Concurrent Linked Lists

Concurrently inserting into a linked list is simple. As a challenge, we think about how to make it as performant as possible.

If a thread acquires the lock at the start of an insert instruction and releases it at the end, then a lot of the critical code is actually already thread-safe. For example, `malloc()` is thread-safe and time consuming, so it has no reason to be in the critical part of the code. Notably, when using locks, try to make the critical part of the code as short as possible.

Hand-over-hand locking is an idea to add a lock to every node of the list to make it scale better. In practice, there is too much overhead from acquiring and releasing locks for every single node in a traversal.

4.3 Concurrent Queues

One approach to implementing a concurrent queue is to have two separate locks for the head and tail. However, this often does not completely meet the needs of programs, and better implementations are studied in the next chapters.

4.4 Concurrent Hash Tables

The hash table scales very well, because a lock can be created for each bucket, and since a good hashing function is random, this usually allows many concurrent operations to take place at once. Note that the inner list implementation also has to be thread safe.

4.5 Summary

Profile code before optimizing anything - no point optimizing away a performance issue that doesn't exist.

Condition Variables

A **join** is when a parent thread wants to check if a child thread has completed before continuing. It can be implemented with a shared variable, but having the parent spin and waste CPU time isn't a good enough implementation.

5.1 Definition and Routines

A **condition variable** is a queue of threads that are waiting on some condition. When some other thread changes the state of the condition, then it can wake one or more of the waiting threads.

A `pthread_cond_t` can be statically initialized with `pthread_cond_t c = PTHREAD_COND_INITIALIZER`, and dynamically initialized with `pthread_cond_init(&c, NULL)`.

Condition variables have two associated operations: `wait(c)`, which puts a calling thread to sleep, and `signal(c)`, which signals that the condition has changed. In C, they are:

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

`pthread_cond_wait` takes a mutex as a parameter, which it assumes is locked. Since the thread is being put to sleep, `pthread_cond_wait` frees the lock, and when the thread is woken up again, the lock has to be acquired again before `pthread_cond_wait` returns.

As a general rule of thumb, a thread should always hold the lock when calling signal. The textbook presents a thread join operation, with multiple variations that fail for different reasons.

5.2 The Producer/Consumer (Bounded Buffer) Problem

The premise of the producer/consumer problem is that there are producer and consumer threads - producer threads put data in a buffer, and consumer threads take the data. Web servers are an example of this - producer threads put HTTP requests in a buffer, and consumer threads handle them. Because the buffer is a shared data structure, its correctness and performance becomes a concurrency problem.

If a lock is used, CPU time is wasted on spin-waiting, so we use condition variables instead.

Mesa semantics state that when a thread is woken by a signal, it is put on the ready queue. Importantly, it doesn't immediately run, meaning another thread can modify the data it woke up to see. **Hoare semantics** are harder to build, but provide a stronger guarantee that threads will run immediately after being signaled.

An example of Mesa semantics in action is: say a consumer thread c_1 runs, sleeping itself on

a condition C. Then, a producer p_1 runs, signaling the condition C and waking c_1. Then, another consumer c_2 runs, sees that the condition C is satisfied, and handles the data in the buffer. Then c_1 runs, and sees nothing is in the buffer.

The fix to this is to use a while loop instead of an if statement - if a thread is woken and sees that the data isn't actually there, it just goes back to sleep.

Be careful about using only one condition variable for multiple threads - a consumer thread might wake up another consumer thread, which would see that there is no data, making it go back to sleep, and since it signaled the consumer thread and not the producer thread, all threads end up going to sleep.

The solution to the single buffer producer/consumer problem is to use two signals, to signal a producer to wake up or a consumer to wake up.

5.3 Covering Conditions

Assume a group of threads need to share some amount of memory. T_1 needs 100 bytes, T_2 needs 10 bytes, and there are no free bytes. T_3 signals that it has just freed 50 bytes, but since threads wait on a condition in a queue, it tries to wake T_1, which goes back to sleep because 50 bytes isn't enough. This is solved with a **covering condition**, which sends a signal to every thread waiting on a condition.

5.4 Summary

Semaphores

6.1 Semaphores: A Definition

Semaphores are objects with an integer value that are manipulated with `wait()` and `post()`. `wait()` decrements the semaphore then waits if it is negative, and `post()` increments the semaphore then wakes a thread if the semaphore is negative.

If a semaphore is negative, its absolute value will always represent the number of waiting threads.

In C, they can be initialized with `sem_t s; sem_init(&s, 0, 1)`. `1` is the value the semaphore is initialized to, and `0` indicates that it is shared between all threads of a process.

6.2 Binary Semaphores (Locks)

A semaphore used the same way as a lock is called a **binary semaphore**.

Say there are three threads, T_1, T_2, T_3, and a semaphore S is initialized to 1. T_1 runs first, decrementing S to 0 and getting interrupted in its critical section. Then, T_2 runs, decrementing S to -1 and sleeping. Then, T_3 runs, decrementing S to -2 and sleeping. Then, T_1 finishes its critical section, incrementing the semaphore to -1, and waking thread T_2. T_2 finishes its critical section, incrementing the semaphore to 0, and waking T_3. T_3 finishes its critical section, incrementing the semaphore to 1, and everything is back to normal.

6.3 Semaphores For Ordering

Semaphores can also be used as a **ordering** primitive.

Say there is a parent thread and a child thread P and T, and a semaphore S is initialized to 0. Assume P creates T, and then gets interrupted for T. T runs, calls `post()` to signal its finish, incrementing the semaphore to 1 and returning. Then P runs later, which is desired behavior. Now assume P creates T, then runs `wait()`, decrementing S to -1 and sleeping. Then, T runs, incrementing S to 0 and wakes P, which is also desired behavior.

The general rule for setting a semaphore is to set it to the number of resources you can give away after immediately after initialization. So, with a lock, you can give away 1 thing immediately (the lock). With the condition variable, you can't give anything away, since the child has to be created first.

6.4 Bounded Buffer Problem

If there is only one producer and one consumer, then a semaphore `empty` initialized to the size of the buffer and a semaphore `full` initialized to 0 will solve the bounded buffer problem. When a producer thread wants to put something in the buffer, it calls `wait(empty)` to wait

for memory to be available, then `post(full)` to wake a consumer thread. When a consumer thread wants to take something from the buffer, it calls `wait(full)` to wait for something to show up in memory, then `post(empty)` to wake a producer thread.

This suffers from a race condition in between the wait and posts, since a thread may get interrupted while it is updating the current index of the buffer. To solve this, add a mutex semaphore inside each of the `wait ... post` blocks - if the mutex wraps the boxes, then deadlock occurs.

6.5 Reader-Writer Locks

Reader-writer locks show up when multiple threads want to read and write to a data structure. A special lock has to made because as long as no write is happening, a read can happen, but in order to write, no reads or writes can be happening.

The write lock is simple to implement for writers, since it's just a simple `wait` and `post` call. When the first reader wants to acquire a read lock, it also needs to acquire the write lock to make sure no writes are happening, and when the last reader wants to acquire a read lock, it needs to release the write lock to let writes happen again.

6.6 The Dining Philosophers

The premise of the **dining philosopher's problem** is that n philosophers (usually 5) are sitting around a table, with forks between them. In order to eat, a philosopher needs the fork to the left and the fork to the right of them. The goal is for there to be no deadlock, for no philosopher to starve, and for concurrency to be high.

If we just use 5 mutexes, then deadlock could occur if each philosopher grabs the fork on their left.

Dijkstra's solution was to select one philosopher and force them to grab forks in a different order from all of the other philosophers.

6.7 Thread Throttling

Thread throttling can happen if there are too many threads demanding too many resources at the same time, such as memory. A semaphore can be wrapped around a memory-intensive area of code to make sure not too many threads use a lot of memory at once.

6.8 How To Implement Semaphores

Semaphores can be implemented using a condition variable and a lock. `wait()` is implemented with locking the semaphore itself, calling `cond_wait()` on the semaphore's condition and the lock if the semaphore is less than or equal to 0, decrementing the semaphore, and unlocking it. (decrement is put after the check because if put before, can lead to race condition) `post()` is implemented by just locking the semaphore, incrementing its value, signaling the semaphore's condition, and unlocking the semaphore.

6.9 Summary

Concurrency Bugs

7.1 What Types Of Bugs Exist?

Roughly 30% of concurrency bugs found in a study on modern applications were deadlock.

7.2 Non-Deadlock Bugs

The two major types of non-deadlock bugs are **atomicity violation** and **order violation** bugs.

An example of atomicity violation is if thread 1 checks some condition, and runs some code based on that condition, then thread 2 updates that condition right after, making thread 1's code likely wrong. This can be fixed with proper locking.

An example of order violation is if thread 2 expects some data to be available by the time it runs, but thread 1 has been delayed and hasn't been able to provide the data. This can be fixed with condition variables.

97% of non-deadlock bugs in the study ended up being either atomicity or order violation bugs.

7.3 Deadlock Bugs

Deadlock can be seen from dependency graphs, where locks and threads are vertices and edges are dependencies. If there are cycles, then deadlock is possible.

In large code bases, complex dependencies can lead to unforeseen deadlocks occuring. For example, a virtual memory system might need to access the file system for swapping, and the file system might need to access the virtual memory system for a page to perform this swap.

Encapsulation is the idea of abstracting implementation details from parts of software to make stuff more modular. However, this doesn't work well with locking. For example, in Java, if `Vector v1, v2` are vectors and `v1.AddAll(v2)` and `v2.AddAll(v1)` are called by two different threads at the same time, deadlock can occur.

Four conditions must hold in order for deadlock to occur:

- Mutual exclusion, which means threads have exclusive control of resources.
- Hold-and-wait, which means threads hold these resources while waiting for additional resources.
- No preemption, which means locks cannot be forcibly removed from threads.
- Circular wait, which means there is a cycle in the thread-lock dependency graph.

Circular waits are the most common condition to target. They can be prevented using **total ordering** on lock acquisition, meaning always acquiring locks in a specific order. If threads require multiple locks at once, the ordering can be guaranteed using the address of the locks. **Partial ordering** is used in more complex codebases, like Linux's memory mapping code, where multiple sets of threads follow a total ordering with each other.

Hold-and-wait can be avoided by using locking to allow threads to acquire their locks "atomically". For example, forcing threads to acquire an overarching lock in order to acquire other locks means a thread can't be interrupted for another thread to claim a lock both of them need. As a downside, this will likely lead to decreased concurrency and performance.

No preemption can be addressed with the help of more flexible locking interfaces that allow threads to check if a lock is free before acquiring it, which is useful in scenarios with multiple locks. This could lead to **livelock**, which occurs when two threads alternately obtain and give up a lock forever.

Mutual exclusion can sometimes be eliminated with lock-free data structures and code, with the help of atomic instructions provided by hardware.

If full knowledge of tasks is known and one has control over hardware, they can schedule threads to run such that there will never be deadlock.

Database systems use deadlock detection and recovery techniques to provide continuous service. They build a resource dependency graph every once in a while and check for cycles, and either automatically restart the system, or are fixed manually.

7.4 Summary

Event-based Concurrency

I/O Devices

9.1 System Architecture

There is a hierarchy of **I/O Buses**, since making performant I/O buses is costly, and they slow down as the distance between devices becomes larger. So, the most important (low latency needed) I/O devices are placed closer to the CPU, and the less important I/O devices can be placed further. Similar to the memory hierarchy, the I/O hierarchy also has more space for devices as I/O gets slower.

In a real architecture, the CPU might have direct, high-performance connections to the graphics card and memory, as well as an I/O chip via a **direct media interface (DMI)**. This I/O chip handles the rest of the I/O, like SSDs through PCIe, peripherals through USB, or hard disks through eSATAs, etc.

9.2 A Canonical Device & The Canonical Protocol

When looking at a device, we need to consider its hardware **interface** to the rest of the system, and its **internal structure**, how it implements that interface.

A device interface usually implements some kind of **status** check, a **command** receiver, and a **data** stream.

When the OS waits for the device to be ready, it repeatedly runs the status check - this is called **polling**. When the CPU is involved in moving data, it is called **programmed I/O (PIO)**.

9.3 Lowering CPU Overhead With Interrupts

Instead of polling, the device can raise an interrupt when it is done, saving CPU cycles and removing the need for polling.

If a very fast device throws an interrupt each time it finishes, then this can slow down the CPU significantly - polling may be better than an interrupt in this situation. If the speed is unknown, then a hybrid approach can be taken, where the CPU polls for a bit, then the device switches to throw interruption mode. This is called a **two-phased** approach.

If a device generates too many interrupts, it can cause the OS to enter **livelock**, where it spends all of its time processing interrupts.

A device can **coalesce** its interrupts - combining multiple requests into one to reduce the frequency of interrupts, though this will increase the latency of a request.

9.4 More Efficient Data Movement With DMA

If a CPU was forced to perform all I/O to disks, then that would be a lot of wasted cycles on some trivial operations. A **direct memory access (DMA)** engine is a device that can help facilitate these transfers without CPU intervention.

9.5 Methods Of Device Interaction

One approach is to have explicit **I/O instructions**, like `in` and `out` on x86.

Another approach is called **memory-mapped I/O**, which virtualizes device registers, which lets the OS load and write to the device.

Both are in use today.

9.6 Fitting Into The OS: The Device Driver & Case Study: A Simple IDE Disk Driver

Device drivers abstract device interactions for the OS. This allows something like a file system to ignore exactly what type of disk it's working with, since it can just use the device drivers' API. Device drivers usually also provide lower-level API's for applications that need them.

A downside of all of this abstraction is it makes it harder for specific features of different devices to be useful - for example a device that has very rich error reporting needs its own device driver written for it.

Over 70% of OS code is dedicated to device drivers, and are a primary contributor to kernel bugs.

9.7 Case Study: A Simple IDE Disk Driver

This section gives an example of a protocol for a device and how to implement it, hard to explain without just repeating the exact example.

9.8 Historical Notes & Summary