Notes on Operating Systems - Three Easy Pieces

Alex Pan

October 5, 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()`, which puts a calling thread to sleep, and `signal()`, 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.

Semaphores

Concurrency Bugs

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