Concurrency

> We want a solution that's *more correct* than letting everyone grab out a peach at once, but *more efficient* than having them all line up.

An Introduction

Building on top of previous abstractions, which provided the illusion of multiple *virtual CPUs* and a large and private *virtual memory*, we can have an abstraction for points of execution. Classical programs have just one point of execution, but we could plausibly have multiple. The result *threads*, looks like multiple processes, except that they all share the same address space.

Each thread has a program counter and a private set of registers (so switching threads involves a *context switch*, though we don't have to switch page tables). Each thread has its own stack (so multiple function executions can happen in the address space at once), sometimes called *thead-local storage*.

Why Use Threads?

- *Parallelism*: it's possible to more effectively use multi-core CPUs
- *Avoiding blocking*: overlap of IO/blocking activities and other activities becomes possible within one program

The Basic API

On Unix, threads can be created with `pthread_create`, and `pthread_join` makes the calling thread wait for another thread to terminate before it proceeds. The order that threads are run is otherwise non-deterministic, governed by some scheduling algorithm. This can lead to *data races*, where uncontrolled scheduling can produce non-deterministic results when code is run. This basically happens because operations are often not atomic meaning the program can be pre-empted in the middle of their execution, causing the end result to come out different from what's expected.

Mutual Exclusion

To solve this, we need *mutual exclusion*, to ensure that only one point of execution passes through certain *critical sections* at a time. One approach would be to have more sophisticated instructions, like `memory-add` that are processed atomically, but this breaks down first because we would have to have tons of them, and second because we'd end up with the same problem composing these operations.

Instead, we build a set of *synchronization primatives* in hardware which allow us to section off critical sections in a way that's much more flexible.

Condition Variables

Another common situation is when we have one thread that needs to wait for another to finish executing before moving forward. For example, we could have a thread that initiates some IO, and we want it to sleep until the IO is finished and be awoken then. This will involve *condition variables*.

Interlude: Unix Threat API

- Threads are created with `pthread_create`
- A thread can wait for another to finish by calling `pthread_join`
- A lock can be initialized by calling `pthread_mutex_init`, locked with `pthread_mutex_lock`, and unlocked with `pthread_mutex_unlock`
- Condition variables can be implemented with `pthread_cond_wait` and `pthread_cond_signal`

Code using these primitives can be compiled with `pthread.h`.

Locks

Locks allow *critical sections* to behave as if they are just one atomic instruction.

Locks: The Basic Idea

The state of a lock is tracked by a *lock variable*, which is either *acquired* or *available* at a given time. Calls to `lock` try to acquire the lock (and succeed if it's available), and calls to `unlock` render it free again. This basic API gives the programmer some control over scheduling back. POSIX locks are called *mutexes*. These locks are implemented using a combination of hardware support and implementation in the OS.

Evaluating Locks

- Locks should provide *mutual exclusion* (basically, they need to work)
- Locks should be as *fair* as possible, with no process getting *starved*
- Locks shouldn't introduce a significant performance overhead

Some Lock Implementations

The simplest possible implementations is having calls to `lock` disable interrupts and calls to `unlock` re-enable them. The guarantees that operations within the lock run atomically, but require trusting the user program to call a privileged instruction and not abuse it. This also doesn't work on multiprocessors, since the code section could easily be run on other processors that don't have interrupts disabled. The approach can also be inefficient, and losing interrupts can cause negative effects. Thus, the approach is not really used, except in limited cases within the OS.

Without Hardware Support

An attempt at a solution that doesn't require this trust could use a flag variable which is set and unset when the lock is held and released. When the variable is locked, a waiting process can just spin-wait, doing nothing and burning cycles till it has a turn. The problem with this naive implementation is a failure of mutual exclusion: multiple threads could get access to the flag at a time.

Spin Locks

A simple hardware primitive, `test_and_set`, helps out. The instruction returns the old value and updates it atomically, which is enough to implement a *spin-lock* in software (set the variable to acquire the lock, check the return value to decide whether to proceed into the critical section). Spin locks are correct, but not fair (they don't provide guarantees against starvation). Performance will differ based on the number of available cores – lots of performance is sacrificed if lots of threads are contending for a lock on one CPU core, since entire cycles would be spent spinning. But across multiple cores, this cost is less because while one lock spins on one core, actual progress through the critical section could be made on another.

Compare and Swap

A slightly more powerful instruction than `test_and_set` is `compare_and_swap`, which checks if the value at an address equals some `expected`, and updates it to some new value if so. It's straightforward to implement a spin-lock using this primitive (set the flag if it's unset, otherwise don't set it), but it can also be used to support *lock-free synchronization* (described later).

Load-Linked and Store-Conditional

Another way we could acquire locks is by retrieving the value of the lock flag in one step, but only updating it if it remains unchanged from the time it was read from memory. This is the idea behind `load_linked` and `store_conditional`.

Fetch-And-Add

A final primitive is `fetch_and_add`, which atomically increments a value while returning the old value. This can be used to implement a lock that guarantees progress for all threads: each value has a turn for some value of a counter, and whenever a thread wants access to a lock it increments the counter.

Reducing Spinning

Each of the above approaches involves a lot of spinning, which we would ideally want to reduce. A naive approach is to just have processes yield whenever they try to acquire the lock and fail. This sort of works, but when we have a lot of threads going we can still end up wasting a lot of CPU time checking if the lock is available and finding it's not.

Using Queues: Sleeping Instead of Spinning

Rather than just relying on the scheduler to determine when processes check for the locks, we could assert more control. One way to do this is to have the OS maintain a queue of processes waiting on a lock, which are put to sleep when the request fails, and have the OS wake them up when the lock becomes available. This is just like yielding, except the process ends up asleep and doesn't burn CPU cycles. These ideas are implemented in terms of `park` and `unpark` on Solaris and as futexes on Linux.

Lock-based Concurrent Data Structures

When thinking about how to allow concurrent access to data in data structures, we want to add the locks such that operations are all *correct*, without sacrificing too much *performance*.

Concurrent Counter

A simple "data structure" is just a counter that gets incremented across several threads. By acquiring a lock before modifying it and releasing it, we can guarantee correctness. Even in this case, we can observe how poorly performance scales (the goal is *perfect scaling*).

Scalable Counting

A tradeoff to improve performance of counters are *approximate counters*. Each CPU keeps its own counter (with a lock, so threads on that CPU can contend for it), and periodically synchronizes with a global counter by acquiring the global lock.

More Complex Data Structures

In general, scaling for more complex data structures involves protecting specific parts of the structure with locks, rather than having one global lock.

Condition Variables

Besides locking data structures, we also want to be able to check *condition* and then conditionally continue execution (ex. `join`).

Definitions and Routines

When a thread wants to wait for something, it an put itself in a queue to be awoken when a signal comes in for them (*condition variable*). On Unix, this is done with `pthread_cond_wait` and `pthread_cond_signal`.

`wait` takes a lock, in order to avoid concurrency problems where the parent incorrectly detects the status of the child, which matters if it decides to call `join`. Since updating isn't an atomic operation, without a lock it could read an incorrect status and stay asleep forever.

Producers and Consumers

A common situation is having a buffer where some process (*producer*) is putting stuff in, and another (*consumer*) is taking stuff out (for example, Unix pipes). Since this happens concurrently, we need to use locks somehow. This problem can be solved generally with two *condition variables* indicating the number of available and filled slots in the buffer (so we know the exact status of the buffer and know which processes to signal when the buffer is completely full or empty).

Covering Conditions

One common problem (encountered in the producer-consumer example) is dealing with uncertainty while sending and receiving signals. For example, when multiple processes request memory that isn't yet available, they might all be put to sleep, but when memory doesn't become available we don't want to accidentally wake up one that requested more than was newly freed. A simple but inefficient solution is to *broadcast* signals to all processes, which then wake up and check if the condition they wanted is true, and go to sleep if it isn't.

Semaphores

In POSIX, a *semaphore* is a integer value plus an interface with two functions: `sem_wait` and `sem_post`.

- `sem_wait` decrements the value of the semaphore and waits if the result is negative (and returning immediately otherwise). Thus, when the value is negative, it equals the number of waiting threads.
- `sem_post` increments the value of the semaphore and wakes at most one waiting thread

Using a Semaphore as a Lock

To use a semaphore as a lock, we simply initialize the value to one. For this case, we can think of the value as the number of concurrent accesses to the section that we can allow. Semaphores used as locks are called *binary semaphores.*

Semaphores for Ordering

Semaphores can also be used like condition variables for *ordering* events. For example, we might often want to have a parent start a child process and then sleep until it finishes (and then be woken, rather than sleeping forever). To do this, we initialize the semaphore value as 0. When the child finishes, it'll increment the semaphore, so that the parent doesn't wait (as is desired). If the parent runs first, it'll be put to sleep by the call to `sem_wait` and awoken when the child finishes and increments the semaphore with `sem_post`.

Semaphores for the Producer-Consumer Problem

To solve the producer consumer problem, we need two semaphores and two mutexes. The semaphores keep track of how many slots are empty and full in the buffer, allowing signals to properly be sent and received. Mutexes are used to avoid losing data – a producer could fill the buffer, but be preempted before updating the semaphore. To avoid this problem, mutexes are used to ensure it's atomic.

Reader-Writer Locks

One way to make locks more fine-grained is to distinguish between types of accesses. Since reads don't modify data structures, we can safely allow concurrent accesses (so long as nobody is writing at the same time). To make this work, we can have separate read and write locks, and have the first reader try to acquire the write lock, blocking any writers for the duration of the reads, as well as a read lock (which keeps track of the number of concurrent readers). We can also come up with fancier policies – this one starves writers.

Dining Philosophers

A canonical concurrency problem considers 5 philosophers eating in a circle, with 5 forks (one between each). Each need to use two forks to eat, so a naive solution to the problem is guarding each with a semaphore. The problem is this can result in *deadlock*, where everyone waits and nobody can make forward progress because they all end up grabbing one fork and waiting for another to become available. The solution is to have at least one have a different policy for how they grab forks (for example, trying to grab the right one before the left).

Implementing Semaphores

Semaphores can be implemented with a lock, a condition variable, and a state variable. Basically, the entire `wait` and `post` operations are locked, ensuring they are updated atomically.

Common Concurrency Problems

In general, concurrent programming is difficult. Concurrency bugs fall into two categories: *deadlock* bugs and *non-deadlock* bugs.

Non-deadlock Bugs

There are two major non-deadlock bugs: *atomicity violation* bugs and *order violation* bugs.

- *Atomicity-violation bugs* arise when we embed incorrect assumptions about operation atomicity into our code. For example, if we check is a value is non-null and then do something else, but change it's value to null in another thread, we'd have an atomicity violation (because the thread that checks and then uses the value could be preempted after checking and before using). It's straightforward to fix these bugs with locks.
- *Order-violation bugs* arise when we embed incorrect order assumptions. For example, a thread might be preempted before it finishes initializing a value, causing a null pointer dereference in another thread. These bugs can generally be fixed using a combination of condition variables and locks (ex. sleep till initialization is done).

Deadlock Bugs

Deadlock related bugs arise when all of the following conditions hold:

- *Mutual exclusion*: threads claim locked resources
- *Hold-and-wait*: threads claim resources and hold them while they wait for other to become available
- *No preemption*: once something grabs a lock, it holds it indefinitely
- *Circular wait*: there's a circular chain of resources, with each holding resources that are being requested by their successor (like the dining philosophers problem)

Thus, to avoid deadlock bugs we can focus on resolving *any* of these problems:

- *Circular wait*: we can impose total or (carefully constructed) partial orderings. Imposing a linear ordering guarantees no cycles, but relies on the programmers to get right.
- *Hold-and-wait*: we can make lock-grabbing atomic, using a global lock. This decreases concurrency though.
- *No preemption*: we want to deal with this problem, indirectly, because often when we're trying to acquire a lock we're holding one or more others. Thus, we don't want to end up going to sleep while waiting for the lock. This problem is solved with `try_lock`, which tries to acquire the lock, and returns an error if it fails. This allows us to avoid situations where one lock tries to acquire two locks, gets the first, but fails to acquire the second (which is being held by another thread, which is waiting for the first lock).
- *Mutual exclusion*: removing mutual exclusion is generally fairly difficult, and the idea behind *lock-free data structures*. Generally, more powerful hardware instructions are used in such a way that data races aren't possible, at the cost of increased implementation difficulty.

Avoidance via Scheduling

A cool (but not very practical) idea is to avoid deadlock by scheduling in such a way that code that acquires the same locks isn't not scheduled to run at the same time. The performance penalty is can be high though, since code is made significantly less concurrent. More importantly, it requires global information about lock holding behavior that isn't generally known ahead of time.

Detect-and-Recover

A final approach is to run code that can deadlock, but setup mechanisms to detect when it occurs and restart the system (doing whatever cleanup is necessary beforehand).

Event-based Concurrency

Another approach to concurrency doesn't need threads, but instead is based around the idea of *events*. The appeal is much more control is granted to the programmer compared to the prior approach, where lots of behavior was determined by the scheduler (this can also be a downside, if the programmer accidentally blocks within an event loop).

The basic flow is events are processed in a loop, and when a new event occurs we just check that type it is and act accordingly. In pseudocode:

```
loop {
    for e in events {
        proceessEvent(e)
    }
}
```

Basically, by deciding what to do based on the event, the user gets to "do scheduling" within the scope of their program. The basic API on linux is `select`, which allows a function to block until events are ready to be processed on a set of file descriptors.

Blocking calls and Asynchronous IO

Making a blocking call in an event loop blocks the entire process, which is obviously undesirable. The solution is to make blocking calls asynchronous, tracking state in minimal data structures, and the OS delivers a signal when the request is completed. Upon receiving the signal, we can look at stored state to figure out what to do.

Other Problems

There are other problems that are more difficult to deal with. In general, running event loops on multiple cores re-introduces the complexity we set out to avoid. Though we were able to deal with explicit blocking pretty well, there remains *implicit blocking* due to things like page faults which we can't avoid. Finally, the interfaces the programmer needs to work with can be difficult.