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.