Notes for 11/9/09

Multiprocess

On x86, when we context switch between processes, we update the CR3 register to point to the new page table and flush the TLB.

fork()—system call to create/fork another process. For the parent process, fork() returns the child process id. For the child process, fork() returns 0. Because of copy-on-write, the parent and children do not have separate pages until one writes to a page, which copies the page and creates a new entry in that process's page table.

exec()—system call to replace the current process image with a new process image

Multithreading

Unlike processes, threads of a process share the same address space. But they typically have their own stacks and temporary registers. The way that they have different stacks is that, in the virtual address space, each thread has a different place for its stack.

Multiple threads can increase performance for two reasons:
Multiple threads can be scheduled on multiple CPU's concurrently.
If one thread blocks, then another thread can still do work, even with one CPU.

Concurrency

The problem—what if two threads are running the following code:

Thread 1	Thread 2

`1: la   s0, counter`
`2: lw   t0, 0(s0)`
`3: addi t0, t0, 1`
`4: sw   t0, 0(s0)`
	`1: la   s0, counter`
	`2: lw   t0, 0(s0)`
	`3: addi t0, t0, 1`
	`4: sw   t0, 0(s0)`

In this case, they are running the same code.

What if Thread 1 gets interrupted in between lines 2 and 3 and then Thread 2 resumes before line 1. Both threads will try to add 1 to the counter, but this will not actually happen because of a race condition. Suppose that counter is initially 5. Thread 1 reads the value of counter from memory into $t0, so $t0 == 5. When Thread 1 is interrupted, then Thread 2 reads the value of counter from memory, so its $t0 == 5, increments it to 6, and then writes back that value to memory. But then when Thread 1 resumes, it increments the stale value of counter in $t0, 5, to 6 and then writes back 6 to memory too. But we wanted it to write back 7, since we wanted each thread to add 1 to counter!

Solution: use mutual exclusion so that only one thread can execute this block of code, called the critical section, at once. With mutual exclusion, if you need to enter the critical section, you must obtain a lock on a mutex. If another thread is already in the critical section, then you must wait (block) until the thread leaves the critical section and then unlocks the mutex to obtain the lock.

With a mutex, you have only one lock to give out. With a counting semaphore, you have a number that you post (increment) and wait (decrement). When the number is 0, then you wait (block) until another thread posts to the semaphore.

Hardware support:
For performance reasons, most hardware provides support for atomic locks.
x86 has an atomic test-and-set instruction
MIPS requires you to keep trying to do an atomic update until it is atomic:

```
atomic_inc:
ll      v0, 0(a0) // a0 is pointer to integer to atomically increment
addu    v0, v0, 1 // increment v0
sc      v0, 0(a0) // store conditionally, only if nothing has
                  // happened at a0 since the ll instruction;
                  // this implements an atomic increment
                  //
beq     v0, zero, atomic_inc
jr      ra
```

Deadlock

Example: Thread 1 holds lock 2. Thread 2 holds lock 1. Thread 1 wants lock 1. Thread 2 wants lock 2. Problem: both threads are going to block indefinitely, since both need the other's lock to continue, but neither will release their locks, since they need another lock to continue.