Concurrency

Review: Concurrency

- Concurrent access to shared data may lead to synchronization errors
- For example:

```
void increment(int* ip)
{
    *ip = *ip + 1;
}
```

Concurrency

- The C code from the previous slide compiles into the following assembly:

```
increment:
  pushl %ebx
  movl 8(%esp), %eax # %eax = ip
  movl (%eax), %ebx  # %ebx = *ip
  add 1, %ebx        # %ebx = *ip + 1
  movl %ebx, (%eax)  # *ip = *ip + 1
  popl %ebx
  ret
```

Concurrency

- So far TOS only implements cooperative multi-tasking. I.e., a process must relinquish control voluntarily via a call to `resign()`
- In pre-emptive multitasking, a context switch can occur between any two machine instructions!
- In this case, concurrency issues may lead to what is called *race conditions*.
- TOS will soon support pre-emptive multi-tasking, so we have to learn how to deal with such problems.

Concurrency

- For the following two scenarios, we assume global variables as follows:

```
int x = 5;
int* p = &x;
int* q = &x;
```

- Let address of `x` be `0x5000`
- Process 1 executes: `increment(p)`
- Process 2 executes: `increment(q)`

Concurrency – Scenario 1

	Process 1	Process 2
Time		`movl 8(%esp), %eax`
	`movl 8(%esp), %eax`	
	`movl (%eax), %ebx`	

`%eax: 0x5000`	`%eax: 0x5000`
`%ebx: 5`	`%ebx: XXXXXXXX`

Memory address `0x5000`: `5`

Concurrency – Scenario 1

Process 1	Process 2
	`movl 8(%esp), %eax`
`movl 8(%esp), %eax`	
`movl (%eax), %ebx`	
`add 1, %ebx`	

`%eax: 0x5000`	`%eax: 0x5000`
`%ebx: 6`	`%ebx: XXXXXXXX`

Memory address `0x5000`: `5`

Concurrency – Scenario 1

Process 1	Process 2
	`movl 8(%esp), %eax`
`movl 8(%esp), %eax`	
`movl (%eax), %ebx`	
`add 1, %ebx`	
`movl %ebx, (%eax)`	

`%eax: 0x5000`	`%eax: 0x5000`
`%ebx: 6`	`%ebx: XXXXXXXX`

Memory address `0x5000`: `6`

Concurrency – Scenario 1

Process 1	Process 2
	`movl 8(%esp), %eax`
`movl 8(%esp), %eax`	
`movl (%eax), %ebx`	
`add 1, %ebx`	
`movl %ebx, (%eax)`	
	`movl (%eax), %ebx`

`%eax: 0x5000`	`%eax: 0x5000`
`%ebx: 6`	`%ebx: 6`

Memory address `0x5000`: `6`

Concurrency – Scenario 1

Process 1	Process 2
	`movl 8(%esp), %eax`
`movl 8(%esp), %eax`	
`movl (%eax), %ebx`	
`add 1, %ebx`	
`movl %ebx, (%eax)`	
	`movl (%eax), %ebx`
	`add 1, %ebx`
	`movl %ebx, (%eax)`
`%eax: 0x5000`	`%eax: 0x5000`
`%ebx: 6`	`%ebx: 7`

Memory address `0x5000`: `7`

Concurrency – Scenario 2

Process 1	Process 2
	`movl 8(%esp), %eax`
`movl 8(%esp), %eax`	
`movl (%eax), %ebx`	

`%eax: 0x5000`	`%eax: 0x5000`
`%ebx: 5`	`%ebx: XXXXXXXX`

Memory address `0x5000`: `5`

Concurrency – Scenario 2

Process 1	Process 2
	`movl 8(%esp), %eax`
`movl 8(%esp), %eax`	
`movl (%eax), %ebx`	
	`movl (%eax), %ebx`

`%eax: 0x5000`	`%eax: 0x5000`
`%ebx: 5`	`%ebx: 5`

Memory address `0x5000`: `5`

Concurrency – Scenario 2

Process 1	Process 2
	`movl 8(%esp), %eax`
`movl 8(%esp), %eax`	
`movl (%eax), %ebx`	
	`movl (%eax), %ebx`
	`add 1, %ebx`
	`movl %ebx, (%eax)`

`%eax: 0x5000`	`%eax: 0x5000`
`%ebx: 5`	`%ebx: 6`

Memory address `0x5000`: `6`

Concurrency – Scenario 2

Process 1	Process 2
	`movl 8(%esp), %eax`
`movl 8(%esp), %eax`	
`movl (%eax), %ebx`	
	`movl (%eax), %ebx`
	`add 1, %ebx`
	`movl %ebx, (%eax)`
`add 1, %ebx`	

`%eax: 0x5000`	`%eax: 0x5000`
`%ebx: 6`	`%ebx: 6`

Memory address `0x5000`: 6

Concurrency – Scenario 2

Process 1	Process 2
	`movl 8(%esp), %eax`
`movl 8(%esp), %eax`	
`movl (%eax), %ebx`	
	`movl (%eax), %ebx`
	`add 1, %ebx`
	`movl %ebx, (%eax)`
`add 1, %ebx`	
`movl %ebx, (%eax)`	
`%eax: 0x5000`	`%eax: 0x5000`
`%ebx: 6`	`%ebx: 6`

Memory address `0x5000`: `6`

Race Conditions

- Scenario 1 executes as expected.
- Scenario 2 leads to a so-called *race condition* because context switches happen at unfortunate moments.
- It is called race condition, because of a “race” between two processes.
- Race conditions only occur rarely, but are very difficult to debug.
- A pre-condition for a race condition is that two processes must access a shared resource (e.g., the same global variable).

Fixing Concurrency Bugs

- To fix this problem, we can use a *lock*.
- Operations on a lock: acquire and release
- When one task acquires a lock, no other task may acquire it until the first task calls release.
  - In other words, only one task at a time may hold the lock

Train Semaphore

- Semaphore signals train if it is safe to enter a “critical section”.

Fixing Concurrency Bugs

```
void increment(int* ip, lock* l)
{
    acquire(l);
    *ip = *ip + 1;
    release(l);
}
```

- Now, a task that begins to increment `*ip` must finish before another task may begin

Implementing Locks

- Need help from the hardware
- Instructions are *atomic*: once an instruction begins executing, nothing else happens until it is finished.

Implementing Locks

- Every modern architecture provides some useful primitives for implementing locks.
- Atomic test-and-set:
  - Test a value (e.g., is value == 0) and set it in a single atomic operation
- Intel x86 also provides atomic swap and atomic load-compute-store (`xchg`).
- Conceptually, `xchg %eax,(memaddr)` does the following:

```
pushl %ebx                  # save %ebx
movl  (memaddr), %ebx       # %ebx = *memaddr
pushl %ebx                  # swap %eax and %ebx
pushl %eax
popl  %ebx
popl  %eax
movl  %ebx,(memaddr)        # *memaddr = %ebx
popl  %ebx                  # restore %ebx
```

Spin Locks

```
lock:   dd 0            # The lock variable. 1 = locked, 0 = unlocked.

spin_acquire:
        mov $1, %eax       # Set the EAX register to 1.
loop:   xchg %eax,(lock)   # Atomically swap the EAX register with
                           # the lock variable. This will always
                           # store 1 to the lock, leaving previous
                           # value in the EAX register.
        test %eax, %eax    # Test EAX with itself. Among other
                           # things, this will set the processor's
                           # Zero Flag if EAX is 0. If EAX is 0,
                           # then the lock was unlocked and we just
                           # locked it. Otherwise, EAX is 1 and we
                           # didn't acquire the lock.
        jnz loop           # Jump back to the XCHG instruction if
                           # the Zero Flag is not set, the lock was
                           # locked, and we need to spin.
        ret                # The lock has been acquired, return to
                           # the calling function.
 spin_release:
        mov $0, %eax       # Set the EAX register to 0.
        xchg %eax,(lock)   # Atomically swap the EAX register with
                           # the lock variable.
        ret                # The lock has been released.
```

Spin Locks

- On the previous slide, the code tests a memory location (`lock`). If this memory location contains a 1, it means another process has already obtained the lock. If the memory location is 0, it means the lock is available. The atomic `xchg` instruction is used to attempt to do an exchange of 1 with the memory location. If `%eax` contains 0 after the `xchg` instruction, it means that the lock was achieved by the current process. If the `%eax` contains a 1 after the atomic `xchg` instruction this signifies that another process already has the lock.

Building a Better Lock

- The problem with spin locks: during a lengthy critical section, other tasks waste CPU cycles (which is called *busy wait*).
- Spin locks are great for short critical sections.
- For longer critical sections, we want a lock that will cause the task to go to "sleep" if the lock is not available.
- "Sleep": process is off the ready queue.

Building a Better Lock

```
struct lock {
    enum { HELD, AVAILABLE } status;
    PROCESS waiting;
}

void acquire(struct lock* l)
{
  if (l->status != AVAILABLE) {
      append(l->waiting, active_proc);
      remove_ready_queue(active_proc);
  }
  l->status = HELD;
}
```

Locks

- Problem: race condition inside `acquire()`
  - If there is a context switch after we check `status` but before changing it, two processes hold the lock simultaneously!
- Solution: the critical section inside `acquire()` is short, so use a spin lock

```
struct lock {
  spinlock slock;
  enum { HELD, AVAILABLE } status;
  PROCESS waiting;
}
```

Locks - Acquire

```
void acquire(struct lock* l)
{
   spin_acquire(l->slock);
   while (l->status != AVAILABLE) {
        append(l->waiting, active_proc);
        spin_release(l->slock);
        remove_ready_queue(active_proc);
        spin_acquire(l->slock);
   }
   l->status = HELD;
   spin_release(l->slock);
}
```

Locks - Release

```
void release(struct lock* l)
{
   spin_acquire(l->slock);
   l->status = AVAILABLE;
   spin_release(l->slock);
   add_ready_queue(l->waiting);
}
```