Lecture 5: Synchronization w/Locks

CSE 120: Principles of Operating Systems

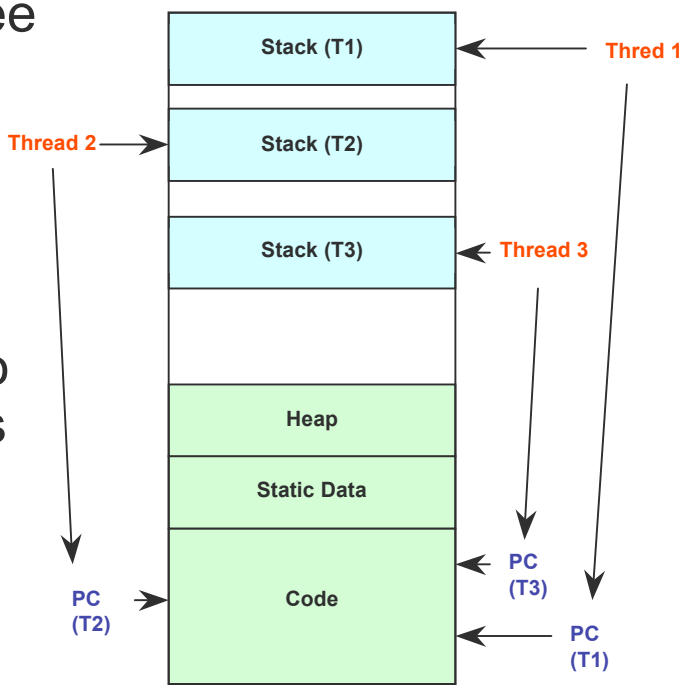
Alex C. Snoeren

Lab 1 Due 10/19

UCSDCSE

Threads Are Made to Share

- Global variables and static objects are shared
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objects are shared
  - Allocated from heap with malloc/free or new/delete
- Local variables are not shared
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on another thread's stack



Thread Scheduling

- The thread scheduler determines when a thread runs
- It uses queues to keep track of what threads are doing
  - Just like the OS and processes
  - But it is implemented at user-level in a library
- Run queue: Threads currently running (usually one)
- Ready queue: Threads ready to run
- Are there wait queues?
  - How would you implement thread_sleep(time)?
- Two types of scheduling
  - Preemptive and Non-preemptive

Non-Preemptive Scheduling

- Threads voluntarily give up the CPU with thread_yield

Ping Thread

```
while (1) {
    printf("ping\n");
    thread_yield();
}
```

Pong Thread

```
while (1) {
    printf("pong\n");
    thread_yield();
}
```

- What is the output of running these two threads?

thread_yield()

- Wait a second. How does thread_yield() work?
- The semantics of thread_yield are that it gives up the CPU to another thread
  - In other words, it context switches to another thread
- So what does it mean for thread_yield to return?
  - It means that *another thread* called thread_yield!
- Execution trace of ping/pong
  - printf("ping\n");
  - thread_yield();
  - printf("pong\n");
  - thread_yield();
  - …

Implementing thread_yield()

```
thread_yield() {
   thread_t old_thread = current_thread;
   current_thread = get_next_thread();
   append_to_queue(ready_queue, old_thread);
   context_switch(old_thread, current_thread);
   return;
}
```

As old thread

As new thread

- The magic step is invoking context_switch()
- Why do we need to call append_to_queue()?

Thread Context Switch

- The context switch routine does all of the magic
  - Saves context of the currently running thread (old_thread)
    - » Push all machine state onto its stack (*not* its TCB)
  - Restores context of the next thread
    - » Pop all machine state from the next thread's stack
  - The next thread becomes the current thread
  - Return to caller as new thread
- This is all done in assembly language
  - It works **at** the level of the procedure calling convention, so it cannot be implemented using procedure calls
  - See code/threads/switch.s in Nachos

Preemptive Scheduling

- Non-preemptive threads have to voluntarily give up CPU
  - A long-running thread will take over the machine
  - Only voluntary calls to thread_yield(), thread_stop(), or thread_exit() causes a context switch
- Preemptive scheduling causes an involuntary context switch
  - Need to regain control of processor asynchronously
  - Use timer interrupt
  - Timer interrupt handler forces current thread to "call" thread_yield
    - » How do you do this?
  - Nachos is preemptive
    - » See use of thread->yieldOnReturn in code/machine/interrupt.cc

The Trouble with Threads

- One basic problem
  - If two concurrent threads are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior
  - Especially tricky with preemptive scheduling
- Over the next couple of lectures, we will look at
  - Mechanisms to control access to shared resources
    - » Locks, mutexes, semaphores, monitors, condition variables, …
  - Patterns for coordinating accesses to shared resources
    - » Bounded buffer, producer-consumer, etc.

Synchronization

- Threads cooperate in multithreaded programs
  - To share resources, access shared data structures
    - » Threads accessing a memory cache in a Web server
  - To coordinate their execution
    - » One thread executes relative to another (recall ping-pong)
- For correctness, we need to control this cooperation
  - Threads interleave executions arbitrarily and at different rates
  - Scheduling is not under program control
- Cooperation is controlled using synchronization
  - Restrict the possible interleavings
- We'll discuss in terms of threads, also applies to processes

Classic Example

- Suppose we have to implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {
        balance = get_balance(account);
        balance = balance – amount;
        put_balance(account, balance);
        return balance;
}
```

- Now suppose that you and your significant other share a bank account with a balance of $1000.
- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account.

Example Continued

- We'll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run in the same bank process:

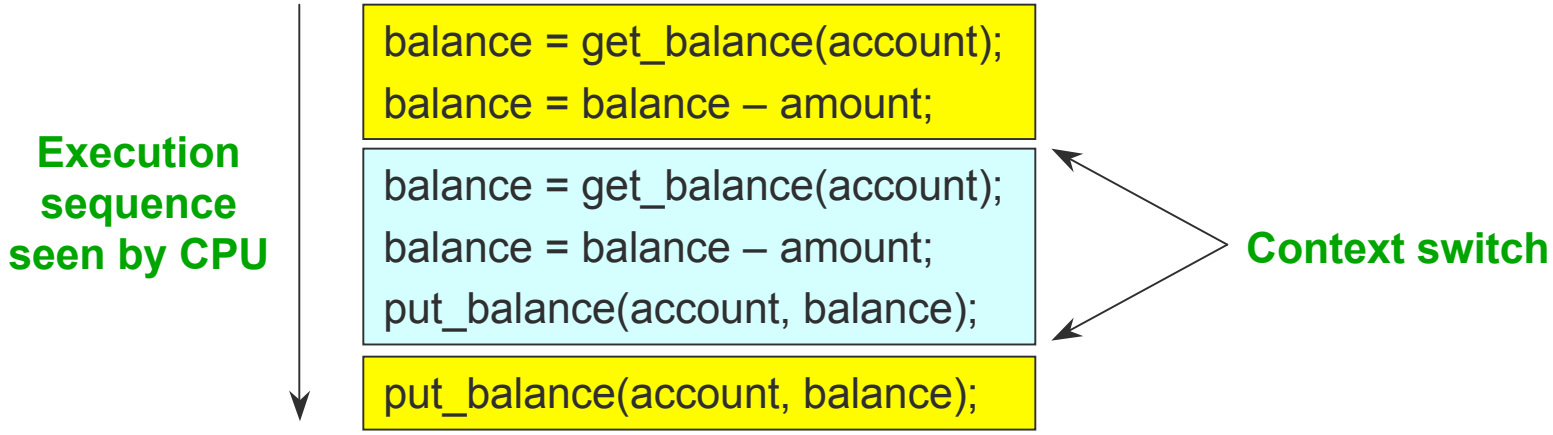
```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- What's the problem with this implementation?
  - Think about potential schedules of these two threads

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:



- What is the balance of the account now?
- This is known as a race condition
  - Each thread is “racing” to put_balance() before the other

Mutual Exclusion

- One way to ensure who wins the race is to only let one thread "compete"; this is called mutual exclusion
- Code that uses mutual exclusion to synchronize its execution is called a critical section
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

Critical Section

Critical Section Requirements

1) Mutual exclusion
   - If one thread is in the critical section, then no other is
2) Progress
   - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
3) Bounded waiting (no starvation)
   - If some thread T is waiting on the critical section, then T will eventually enter the critical section
4) No assumptions on performance
   - Requirements must be met with any number of CPUs with arbitrary relative speeds

Locks

- One way to implement critical sections is to “lock the door” on the way in, and unlock it again on the way out
- A lock is an object in memory providing two operations
  - acquire(): before entering the critical section
  - release(): after leaving a critical section
- Threads pair calls to acquire() and release()
  - Between acquire()/release(), the thread holds the lock
  - acquire() does not return until any previous holder releases
  - What can happen if the calls are not paired?

Using Locks

```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

Critical Section

```
acquire(lock);
balance = get_balance(account);
balance = balance – amount;
```

```
acquire(lock);
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
release(lock);
```

- What happens when blue tries to acquire the lock?
- Why is the “return” outside the critical section? Is this ok?
- What happens when a third thread calls acquire?

First Try: Spin Locks

- How do we implement locks? Here is one attempt:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

busy-wait (spin-wait) for lock to be released

- This is called a spinlock because a thread spins waiting for the lock to be released
- Does this work?

Spin Locks

- No. Two independent threads may both notice that a lock has been released and thereby acquire it.

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

A context switch can occur here, causing a race condition

Take Turns?

- How did we solve this problem in Kindergarden?
  - Let's assume only two threads, and take turns

```
struct lock {
    int turn = 0;
}
void acquire (lock) {
    while (lock->turn != this_thread);
}
void release (lock) {
    lock->turn = other_thread;
}
```

- Does this work? Why not?

Declaring Intent

- Problem was we didn't know if other thread was ready
  - Let's be polite and wait until the other thread isn't interested

```
struct lock {
    int interested[2] = [FALSE, FALSE];
}
void acquire (lock) {
    lock->interested[this_thread] = TRUE;
    while (lock->interested[other_thread]);
}
void release (lock) {
    lock->interested[this_thread] = FALSE;
}
```

- Now will it work?

Peterson's Algorithm

- Take turns only if somebody else is interested; otherwise just go!

```
struct lock {
    int turn = 0;
    int interested[2] = [FALSE, FALSE];
}
void acquire (lock) {
    lock->interested[this_thread] = TRUE;
    turn = other_thread;
    while (lock->interested[other_thread] && turn==other_thread);
}
void release (lock) {
    lock->interested[this_thread] = FALSE;
}
```

Other Approaches

- Problem is that we need to know who else is playing
- How do we do this is in general?
- The implementation of acquire/release must be atomic
  - An atomic operation is one which executes as though it could not be interrupted
  - Code that executes “all or nothing”
- How do we make them atomic?
  - Atomic HW instructions (e.g., test-and-set)
  - Disable/enable interrupts (prevents context switches)

test-and-set

- The semantics of test-and-set are:
  - Record the old value *and*
  - Set the value to indicate available *and*
  - Return the old value
- Hardware executes it atomically!

```
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

- When executing test-and-set on "flag"
  - What is value of flag afterwards if it was initially False? True?
  - What is the return result if flag was initially False? True?

Using test-and-set

- Here is simple lock implementation with test-and-set:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return?
- What about multiprocessors?

Problems with Spinlocks

- The problem with spinlocks is that they are wasteful
  - If a thread is spinning on a lock, then the thread holding the lock cannot make progress
- How did the lock holder give up the CPU in the first place?
  - Lock holder calls yield or sleep
  - Involuntary context switch
- Only want to use spinlocks as primitives to build higher-level synchronization constructs

Disabling Interrupts

- Another implementation of acquire/release is to disable interrupts:

```
struct lock {
}
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

- Note that there is no state associated with the lock
- Can two threads disable interrupts simultaneously?

On Disabling Interrupts

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
  - This is what Nachos uses as its primitive
- In a "real" system, this is only available to the kernel
  - Why?
- Disabling interrupts is insufficient on a multiprocessor
  - Back to atomic instructions
- Like spinlocks, only want to disable interrupts to implement higher-level synchronization primitives
  - Don't want interrupts disabled between acquire and release

Summarize Where We Are

- Goal: Use mutual exclusion to protect critical sections of code that access shared resources
- Method: Use locks (spinlocks or disable interrupts)
- Problem: Critical sections can be long

Spinlocks:

- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted

```
acquire(lock)
…
Critical section
…
release(lock)
```

Disabling Interrupts:

- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)

Higher-Level Synchronization

- Spinlocks and disabling interrupts are useful only for very short and simple critical sections
  - Wasteful otherwise
  - These primitives are "primitive" – don't do anything besides mutual exclusion
- Need higher-level synchronization primitives that:
  - Block waiters
  - Leave interrupts enabled within the critical section
- All synchronization requires atomicity
- So we'll use our "atomic" locks as primitives to implement them

Next time…

- Read Chapter 6.7 – 6.10
- Have a great weekend!