Lecture 5: Synchronization w/Locks

CSE 120: Principles of Operating Systems
Alex C. Snoeren

Lab 1 Due 5/01
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
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
Suppose we have to implement a function to handle withdrawals from a bank account:

```c
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:

```c
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
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
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

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

- 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;
}
```

- 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.

```c
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 Kindergarten?
  - Let’s assume only two threads, and take turns

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

```c
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!

```c
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)
The semantics of test-and-set are:

- Record the old value \textit{and}
- Set the value to indicate available \textit{and}
- Return the old value

Hardware executes it atomically!

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

When executing test-and-set on “flag”

- What is \textit{value of flag} afterwards if it was initially False? True?
- What is the \textit{return result} if flag was initially False? True?
Using test-and-set

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

```c
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
Another implementation of acquire/release is to disable interrupts:

```c
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)
  - Effectively gives thread exclusive access to the CPU
- 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
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

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
- Project groups; otherwise we will assign them for you!
- Homework 2 assigned Thursday