CSE 451: Operating Systems
Winter 2009

Module 7
Synchronization

Mark Zbikowski
Gary Kimura
Synchronization

• Threads cooperate in multithreaded programs
  – to share resources, access shared data structures
    • e.g., threads accessing a memory cache in a web server
  – also, to coordinate their execution
    • e.g., a disk reader thread hands off blocks to a network writer thread through a circular buffer
Synchronization

• For correctness, we have to control this cooperation
  – must assume threads interleave executions arbitrarily and at different rates
    • Modern OS’s are preemptive
    • Most new machines are multicore
    • scheduling is not under application writers’ control (except for real-time, but that’s not of interest here).

• We control cooperation using synchronization
  – enables us to restrict the interleaving of executions

• Note: this also applies to processes, not just threads
  – (I’ll almost never say “process” again!)

• It also applies across machines in a distributed system (Big Research Topic)
Shared resources

• We’ll focus on coordinating access to shared resources
  – basic problem:
    • two concurrent threads are accessing a shared variable
    • if the variable is read/modified/written by both threads, then access to the variable must be controlled
    • otherwise, unexpected results may occur

• Over the next several lectures, we’ll look at:
  – mechanisms to control access to shared resources
    • low level mechanisms like locks
    • higher level mechanisms like mutexes, semaphores, monitors, and condition variables
  – patterns for coordinating access to shared resources
    • bounded buffer, producer-consumer, …
The classic example

• Suppose we have to implement a function to withdraw money from a bank account:

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```

• Now suppose that you and your S.O. share a bank account with a balance of $100.00
  – what happens if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?
• Represent the situation by creating a separate thread for each person to do the withdrawals
  – have both threads run on the same bank mainframe:

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```
Interleaved schedules

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:

- What’s the account balance after this sequence?  
  - who’s happy, the bank or you?

- How often is this unfortunate sequence likely to occur?

```
balance = get_balance(account);
balance -= amount;

context switch

balance = get_balance(account);
balance -= amount;
put_balance(account, balance);

context switch

put_balance(account, balance);
```
Other Execution Orders

• Which interleavings are ok? Which are not?

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```
int xfer(from, to, amt) {
    int bal = withdraw(from, amt);
    deposit(to, amt);
    return bal;
}
And This?

```c
i++;  
```

```c
i++;  
```
The crux of the matter

• The problem is that two concurrent threads (or processes) access a shared resource (account) without any synchronization
  - creates a race condition
    • output is non-deterministic, depends on timing

• We need mechanisms for controlling access to shared resources in the face of concurrency
  - so we can reason about the operation of programs
    • essentially, re-introducing determinism

• Synchronization is necessary for any shared data structure
  - buffers, queues, lists, hash tables, scalars, …
What resources are shared?

• 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!

• Global variables are shared
  – stored in the static data segment, accessible by any thread

• Dynamic objects are shared
  – stored in the heap, shared if you can name it
    • in C, can conjure up the pointer
      – e.g., `void *x = (void *) 0xDEADBEEF`
    • in Java/C#, strong typing prevents this
      – must pass references explicitly
Mutual exclusion

• We want to use mutual exclusion to synchronize access to shared resources

• Mutual exclusion makes reasoning about program behavior easier
  – making reasoning easier leads to fewer bugs

• 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
Critical section requirements

• Critical sections have the following requirements
  – mutual exclusion
    • at most one thread is in the critical section
  – progress
    • if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
  – bounded waiting (no starvation)
    • if thread T is waiting on the critical section, then T will eventually enter the critical section
       – Assumes threads eventually leave critical sections
    • vs. fairness?
  – performance
    • the overhead of entering and exiting the critical section is small with respect to the work being done within it
Mechanisms for building critical sections

- **Locks**
  - very primitive, minimal semantics; used to build others

- **Semaphores**
  - basic, easy to get the hang of, hard to program with

- **Monitors**
  - high level, requires language support, implicit operations
  - easy to program with; Java “synchronized()” as an example

- **Messages**
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems (SOAP, RPC)
Locks

- A lock is an object (in memory) that provides the following two operations:
  - `acquire()`: a thread calls this before entering a critical section
  - `release()`: a thread calls this after leaving a critical section
- Threads pair up calls to `acquire()` and `release()`
  - between `acquire()` and `release()`, the thread holds the lock
  - `acquire()` does not return until the caller holds the lock
    - at most one thread can hold a lock at a time (usually)
  - so: what can happen if the calls aren’t paired?
- Two basic flavors of locks
  - spinlock
  - blocking (a.k.a. “mutex”)
Using locks

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

• What happens when green tries to acquire the lock?
• Why is the “return” outside the critical section?
  – is this ok?
Spinlocks

• How do we implement locks? Here’s one attempt:

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

• Why doesn’t this work?
  – where is the race condition?

the caller “busy-waits”, or spins, for lock to be released ⇒ hence spinlock
Implementing locks (cont.)

• Problem is that implementation of locks has critical sections, too!
  – the acquire/release must be atomic
    • atomic == executes as though it could not be interrupted
    • code that executes “all or nothing”

• Need help from the hardware
  – disable/enable interrupts
    • to prevent context switches
  – atomic instructions
    • test-and-set, compare-and-swap, …
  – multiple processors?
Spinlocks redux: Test-and-Set

• CPU provides the following as one atomic instruction:

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

• Remember, this is a single instruction…
Spinlocks redux: Test-and-Set

• So, to fix our broken spinlocks, do:

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

– mutual exclusion?
– progress?
– bounded waiting?
– performance?
Real World Example

• Windows XP AcquireSpinlock

    AcquireSpinlock:
    ;
    ; Attempt to assert the lock
    ;
    lock bts dword ptr [LockAddress], 0
    jc      SpinLabel ; spinlock owned
    ret

SpinLabel:
    ;
    ; Was spinlock cleared?
    ;
    test   dword ptr [LockAddress], 1
    jz      AcquireSpinlock
    YIELD

    jmp    Spinlabel

    ; ...
Reminder of use …

• How does a thread blocked on an “acquire” (that is, stuck in a test-and-set loop) yield the CPU?
  – calls yield() (spin-then-block)
  – there’s an involuntary context switch
Problems with spinlocks

• Spinlocks work, but are horribly wasteful!
  – if a thread is spinning on a lock, the thread holding the lock cannot make progress
  – And neither can anyone else! Why?

• Only want spinlocks as primitives to build higher-level synchronization constructs
  – Why is this okay?

• When might the above points be misleading?
Another approach: Disabling interrupts

```c
struct lock {
}

void acquire(lock) {
    cli();     // disable interrupts
}

void release(lock) {
    sti();     // reenable interrupts
}
```
Problems with disabling interrupts

• Only available to the kernel
  – Can’t allow user-level to disable interrupts!

• Insufficient on a multiprocessor
  – Each processor has its own interrupt mechanism

• “Long” periods with interrupts disabled can wreak havoc with devices

• Just as with spinlocks, you only want to use disabling of interrupts to build higher-level synchronization constructs
Summary

• Synchronization can be provided by locks, semaphores, monitors, messages …

• Locks are the lowest-level mechanism
  – very primitive in terms of semantics – error-prone
  – implemented by spin-waiting (crude) or by disabling interrupts (also crude, and can only be done in the kernel)

• In our next exciting episode …
  – semaphores are a slightly higher level abstraction
    • less crude implementation too
  – monitors are significantly higher level
    • utilize programming language support to reduce errors