CS 537
Lecture 6
Synchronization and IPC

Michael Swift

Questions for this Lecture

- How can multiple processes cooperate?
- How can multiple threads cooperate?

Interprocess Communication (IPC)

- To cooperate usefully, threads must communicate with each other
- How do processes and threads communicate?
  - Shared Memory
  - Message Passing
  - Signals

IPC: Shared Memory

- Processes
  - Each process has private address space
  - Explicitly set up shared memory segment within each address space
- Threads
  - Always share address space (use heap for shared data)
- Advantages
  - Fast and easy to share data
- Disadvantages
  - Must **synchronize** data accesses; error prone
- Synchronization: Topic for next few lectures

IPC: Message Passing

- Message passing most commonly used between processes
  - Explicitly pass data btween **sender** (src) + **receiver** (destination)
  - Example: Unix pipes, Windows LPC
- Advantages:
  - Makes sharing explicit
  - Improves modularity (narrow interface)
  - Does not require trust between sender and receiver
- Disadvantages:
  - Performance overhead to copy messages
- Issues:
  - How to name source and destination?
    - One process, set of processes, or mailbox (port)
  - Does sending process wait (I.e., block) for receiver?
    - Blocking: Slows down sender
    - Non-blocking: Requires buffering between sender and receiver

9/20/07

IPC: Signals

- Signal
  - Software interrupt that notifies a process of an event
  - Examples: SIGFPE, SIGKILL, SIGUSR1, SIGSTOP, SIGCONT
- What happens when a signal is received?
  - Catch: Specify signal handler to be called
  - Ignore: Rely on OS default action
    - Example: Abort, memory dump, suspend or resume process
  - Mask: Block signal so it is not delivered
    - May be temporary (while handling signal of same type)
- Disadvantage
  - Does not specify any data to be exchanged
  - Complex semantics with threads
  - Not implemented in Windows

9/20/07

Threads and Signals

- Problem: To which thread should OS deliver signal?
- Option 1: Require sender to specify thread id (instead of process id)
  - Sender may not know about individual threads
- Option 2: OS picks destination thread
  - POSIX: Each thread has signal mask (disable specified signals)
  - OS delivers signal to all threads without signal masked
  - Application determines which thread is most appropriate for handing signal

9/20/07

Shared Memory Thread 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 a block to a network writer
- For correctness, we have to control this cooperation
  - must assume threads interleave executions arbitrarily and at different rates
    - scheduling is not under application writers' control
  - we control cooperation using synchronization
    - enables us to restrict the interleaving of executions
- Note: this also applies to processes, not just threads
  - and it also applies across machines in a distributed system

9/20/07

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
- 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:

```
int withdraw(account, amount) {
  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?

Example continued

- Represent the situation by creating a separate thread for each person to do the withdrawals
  - have both threads run on the same bank mainframe:

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

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

- What's the problem with this?
  - what are the possible balance values after this runs?

Interleaved Schedules

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

Execution sequence as seen by CPU

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

context switch

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

context switch

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

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

What just happened?

- Threads share global memory
- When a process contains multiple threads, they have
  - Private registers and stack memory (the *context switching* mechanism needs to save and restore registers when switching from thread to thread)
  - Shared access to the remainder of the process “state”
- This can result in *race conditions*
  - **Race condition:** Result depends upon ordering of execution
    - Non-deterministic bug, very difficult to find

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, …

When are Resources Shared?

- Local variables are not shared
  - refer to data on the stack, each thread has its own stack
  - But… you must 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, strong typing prevents this
      - must pass references explicitly

Mutual Exclusion

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

Scheduler assumptions

```
Process a:
    while(i < 10)
      i = i +1;
    print "A won!";
```

```
Process b:
    while(i > -10)
      i = i - 1;
    print "B won!";
```

If i is shared, and initialized to 0

- Who wins?
- Is it guaranteed that someone wins?
- What if both threads run on identical speed CPU
  - executing in parallel

Scheduler Assumptions

- Normally we assume that
  - A scheduler always gives every executable thread opportunities to run
    - In effect, each thread makes *finite progress*
  - But schedulers aren't always fair
    - Some threads may get more chances than others
  - To reason about worst case behavior we sometimes think of the scheduler as an adversary trying to "mess up" the algorithm

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
  - performance
    - the overhead of entering and exiting the critical section is small with respect to the work being done within it
    - Do not busy wait (I.e., spin wait)
  - Fair
    - Don't make some processes wait longer than others

Mechanisms for Building Crit. 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 example
- Messages
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems

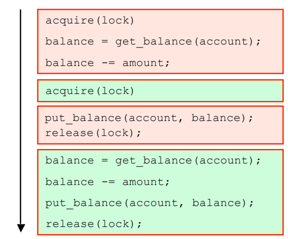
Locks

- A lock is a 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;
}
```

critical section



- What happens when green tries to acquire the lock?
- Why is the "return" outside the critical section?
  – is this ok?

Critical Section: Attempt #1

- Code uses a single shared lock variable

```
Boolean lock = false; // shared variable
Void withdraw(int amount) {
  while (lock) /* wait */ ;
  lock = true;

  balance -= amount; // critical section

  lock = false;
}
```

- Why doesn't this work? Which principle is violated?

Attempt #2

- Each thread has its own lock; lock indexed by tid (0, 1)

```
Boolean lock[2] = {false, false}; // shared
Void withdraw(int amount) {
  lock[tid] = true;
  while (lock[1-tid]) /* wait */ ;

  balance -= amount; // critical section

  lock[tid] = false;
}
```

- Why doesn't this work? Which principle is violated?

Attempt #3

- Turn variable determines which thread can enter

```
Int turn = 0; // shared
Void withdraw(int amount) {
  while (turn == 1-tid) /* wait */ ;

  balance -= amount; // critical section

  turn = 1-tid;
}
```

- Why doesn't this work? Which principle is violated?

Peterson's Algorithm: Solution for Two Threads

- Combine approaches 2 and 3: Separate locks and turn variable

```
Int turn = 0; // shared
Boolean lock[2] = {false, false};
Void withdraw(int amount) {
  lock[tid] = true;
  turn = 1-tid;
  while (lock[1-tid] && turn == 1-tid) /* wait */ ;

  balance -= amount; // critical section

  lock[tid] = false;
}
```


Peterson's Algorithm: Intuition

- Mutual exclusion: Enter critical section if and only if
  - Other thread does not want to enter
  - Other thread wants to enter, but your turn
- Progress: Both threads cannot wait forever at while() loop
  - Completes if other process does not want to enter
  - Other process (matching turn) will eventually finish
- Bouded waiting
  - Each process waits at most one critical section

Postscript

- These algorithm will not work with many modern CPUs
  - CPUs execute their instructions in an out-of-order (OOO) fashion
  - This algorithm won't work on Symmetric MultiProcessors (SMP) CPUs equipped with OOO without the use of memory barriers
- Compiler optimizations can break these algorithms
  - What if the compiler puts a variable in a register?
  - What if the compiler sees that a variable does not change inside a loop and removes the test?

Hardware Support: Test-and-Set

- CPU provides the following as one atomic instruction:

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

- So, to fix our broken spinlocks, do:

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


Problems with spinlocks

- Horribly wasteful!
  - if a thread is spinning on a lock, the thread holding the lock cannot make process
- How did lock holder yield the CPU in the first place?
  - calls yield( ) or sleep( )
  - involuntary context switch
- Only want spinlocks as primitives to build higher-level synchronization constructs

Disabling Interrupts

- An alternative:

```
struct lock {
}
void acquire(lock) {
   cli();   // disable interrupts
}
void release(lock) {
  sti();    // reenable interrupts
}
```

- Can two threads disable interrupts simultaneously?
- What's wrong with interrupts?
  - only available to kernel (why? how can user-level use?)
  - insufficient on a multiprocessor
    - back to atomic instructions
- Like spinlocks, only use to implement higher-level synchronization primitives