CSC369: Operating Systems

Fall 2014

Andrew Petersen

Anonymous Feedback

"I really want to know if im the only one found this course super hard"

- I am confident you are not the only one.
- If you're in this situation, please come talk to me. Let's figure out what you're having trouble with, and let's build a study plan for getting comfortable in the course.
- ... I doubt it will get better if you just hope that it gets better. The key question is, "What will you do about it?"

Any AI Questions?

What is a Thread?

- A thread is a single control flow through a program
  - What is a "control flow"?
  - How is control flow represented?
- A program with multiple control flows is multithreaded

Control Flow

- Control includes all of the values that select which instructions in a program are executed.
- Control flow, then, is the sequence of instructions being executed.
- The hardware uses the program counter (PC) and stack to make control flow decisions.

What does a multi-threaded address space look like?

Which segments are shared?

Synchronization

- Processes (and threads) interact in a multiprogrammed system
  - To share resources (such as shared data)
  - To coordinate their execution
- Arbitrary interleaving of thread executions can have unexpected consequences

- Synchronization is the mechanism that gives us control over interleavings
  - Restores determinism (or, at least, a semblance of it)

What Could Go Wrong?

- Two concurrent threads manipulate a shared resource without synchronization
  - The outcome depends on the order in which accesses take place
    - This is called a race condition
- We need to ensure that only one thread at a time can manipulate the shared resource
  - We need mutual exclusion
  - Synchronization can provide this

Deadlock and Starvation

- Deadlock occurs when each thread in a set is waiting for an event that can only be caused by another thread in the set
  - In other words, "circular dependency"
  - In the case of semaphores, the event is the execution of the signal operation
- A thread is suffering starvation (or indefinite postponement) if it is waiting for a signal that never comes (or is not guaranteed to be sent)

Caution!

- Synchronization problems can occur even with a simple shared variable, even on a uniprocessor:
  - T_1 and T_2 share variable X
  - T_1 increments X (X := X+1)
  - T_2 decrements X (X := X-1)
  - But at the machine level, we have:

```
T1: LOAD X
    INCR
    STORE X
```

```
T2: LOAD X
    DECR
    STORE X
```

- Same problem of interleaving can occur!

The Critical Section Problem

Mutual Exclusion

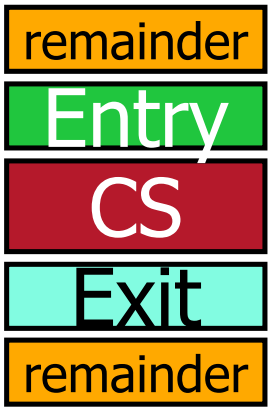
- Given:
  - A set of n threads, $T_0, T_1, \ldots, T_n$
  - A set of resources shared between threads
  - A segment of code which accesses the shared resources, called the critical section, CS
- We want to ensure that:
  - Only one thread at a time can execute in the critical section
    - Or access the critical data
  - All other threads are forced to wait on entry
  - When a thread leaves the CS, another enters

Critical Section Requirements

- Mutual Exclusion
  - If one thread is in the CS, then no other is
- Progress
  - The choice of next thread to enter cannot be postponed indefinitely
- Bounded waiting (no starvation)
  - All waiting threads are guaranteed to eventually get access to the CS
- Performance
  - The overhead of entering and exiting the CS is small with respect to the work being done within it

The Critical Section Problem

- Design a protocol that threads can use to cooperate
  - Each thread must request permission to enter its CS, in its entry section
  - The CS may be followed by an exit section



- Each thread is executing at non-zero speed
  - Make no assumptions about relative speed

Assumptions & Notation

- Assume no special hardware instructions, no restrictions on the number of processors (for now)
- Assume that basic machine language instructions (LOAD, STORE, etc.) are atomic:
  - If two such instructions are executed concurrently, the result is equivalent to their sequential execution in some order
  - On modern architectures, this assumption may be false
- If only two threads, we number them T_0 and T_1
  - Use T_i to refer to one thread, T_j for the other (j=1-i) when the exact numbering doesn't matter

2-Threads: 1st Try

- Let the threads share an integer variable *turn* initialized to 0 (or 1)
- If *turn*=i , thread T_i is allowed into its critical section

```
My_work(id_t id) {   /* id_t can be 0 or 1 */
    ...
    while (turn != id) ; /* entry section */
    /* critical section, access protected resource */
    turn = 1 - id; /* exit section */
    ...            /* remainder section */
}
```

✓ Only one thread at a time can be in its CS

✘ Progress is not satisfied
- Requires strict alternation of threads in their CS: if turn=0, T_1 may not enter, even if T_0 is in the remainder section

2-Threads: 2nd Try

```
My_work(id_t id) {   /* id can be 0 or 1 */
   ...
   while (flag[1-id]) ; /* entry section */
   flag[id] = true;  /* indicate entering CS */
   /* critical section, access protected resource */
   flag[id] = false; /* exit section */
   ...               /* remainder section */
}
```

- Replace *turn* with a shared flag for each thread
  boolean flag[2] = {false, false}
  - Each thread may update its own flag, and read the other thread's flag
  - If *flag[1-i]* is true, T_i may not enter its CS

2-Threads: 2nd Try

```
My_work(id_t id) {   /* id can be 0 or 1 */
   ...
   while (flag[1-id]) ; /* entry section */
   flag[id] = true;  /* indicate entering CS */
   /* critical section, access protected resource */
   flag[id] = false; /* exit section */
   ...               /* remainder section */
}
```

- Mutual exclusion is not guaranteed
  - Each thread executes *while* statement, finds *flag* set to *false*
  - Each thread sets own flag to true and enters CS
- Can't fix this by changing order of testing and setting flag variables (leads to deadlock)

2-Threads: 3rd Try

```
My_work(id_t id=0) {
    flag[id] = true;
    turn = id
    while (turn == id
      && flag[1-id]);
    /* critical section */
    flag[id] = false;
}
```

- Combine the two ideas -- use both a flag and turn variable.
- Does this work?

2-Threads: 3rd Try

```
My_work(id_t id=0) {
    flag[id] = true;
    turn = id
    while (turn == id
      && flag[1-id]);
    /* critical section */
    flag[id] = false;
}
```

- Basic idea: if both threads try to enter their CS at the same time, turn will be set to both 0 and 1 at roughly the same time. *Only one of these assignments will last.* The final value of turn decides which of the two threads is allowed to enter its CS first.
- This is the basis of Dekker's Algorithm (1965) and Peterson's Algorithm (1981)

Multiple-Thread Solutions

- Peterson's Algorithm can be extended to N threads
- Another approach is Lamport's Bakery Algorithm
  - Upon entering each customer (thread) gets a #
  - The customer with the lowest number is served next
  - No guarantee that 2 threads do not get same #
    - In case of a tie, thread with the lowest PID (or TID) is served first
    - Thread id's are unique and totally ordered

Synchronization Primitives

A Lock Implementation

- There are two operations on locks: acquire() and release()

```
boolean lock;


void acquire(boolean *lock) {
   while(test_and_set(lock));
}

void release(boolean *lock) {
   *lock = false;
}
```

- This is a spinlock
- Uses busy waiting - thread doesn't release CPU

Synchronization Hardware

- To build higher-level abstractions, it is useful to have some help from the hardware
  - On a uniprocessor, in the OS, we can disable interrupts before entering critical section (prevents context switches)
    - Why does this stop us from implementing user level locks?
- Disabling interrupts is insufficient on a multiprocessor
  - Need some special "atomic instructions"

Atomic Instructions: Test-and-Set

- The semantics of test-and-set are:
  - Record the old value of the variable
  - Set the variable to some non-zero value
  - Return the old value
- Hardware executes this atomically!
- Can be used to implement simple lock variables

```
boolean test_and_set(boolean *lock) {
   boolean old = *lock;
   *lock = true;
   return old;
}
```

Using Locks

Function Definitions

```
Withdraw(acct, amt) {
    acquire(lock);
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct,balance);
    release(lock);
    return balance;
}
```

```
Deposit(account, amount) {
    acquire(lock);
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct,balance);
    release(lock);
    return balance;
}
```

Possible schedule

```
acquire(lock);
balance = get_balance(acct);
balance = balance - amt;
```

```
acquire(lock);
```

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

```
balance = get_balance(acct);
balance = balance + amt;
put_balance(acct, balance);
release(lock);
```

Drawbacks to Machine Instructions

- Other hardware instructions are possible
  - For example, Swap (or Exchange) instruction
    - Operates on two words atomically
- Machine instructions have three problems:
  - Busy waiting
  - Starvation is possible
    - The scheduler affects who gets to enter their critical section next; with bad luck, a thread will never get a chance to enter
  - Deadlock is possible through priority inversion

A Motivating Example

The Read/Write Problem

- Readers/Writers Problem:
  - An object is shared among several threads
  - We can allow multiple concurrent readers
  - But only one writer
- How can semaphores control access to the object and implement this protocol?
- Use three variables
  - int readcount - number of threads reading object
  - Semaphore mutex - control access to readcount
  - Semaphore w_or_r - exclusive writing or reading

Readers/Writers

```
//number of readers
int readcount = 0;
//mutual exclusion to readcount
Semaphore mutex = 1;
//exclusive writer or reading
Semaphore w_or_r = 1;


Writer {
    wait(w_or_r);  //lock out others
    Write;
    signal(w_or_r); //up for grabs
}
```

```
Reader {
    wait(mutex); //lock readcount
    // one more reader
    readcount += 1;
    // is this the first reader?
    if(readcount == 1)
        //synch w/ writers
        wait(w_or_r);
    //unlock readcount
    signal(mutex);
    Read;
    wait(mutex); //lock readcount
    readcount -= 1;
    if(readcount == 0)
        signal(w_or_r);
    signal(mutex);
}
```