CSC 369
Spring 2007

Week 3: Synchronization

Reading: Text Ch. 2.2-2.3

Announcements

- Assignment 0 is due Friday!
- Assignment 1 out Monday
- group signup page is now working
  - Please email with any errors or unexpected behavior.

Brief preview of scheduling

We have:

- Multiple threads/processes ready to run
- Some mechanism for switching between them
  - Context switches
- Some policy for choosing the next process to run
  - This policy may be pre-emptive
  - Meaning thread/process can't anticipate when it may be forced to yield the CPU
  - By design, it is not easy to detect it has happened (only the timing changes)

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
  - We need a way to restrict the possible interleavings of executions
  - Scheduling is invisible to the application
- Synchronization is the mechanism that gives us this control

Motivating Example

- Suppose we write functions to handle withdrawals and deposits to bank account:

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

```
Deposit(acct, amt) {
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct,balance);
    return balance;
}
```

- Now suppose you share this account with someone and the balance is $1000
- You each go to separate ATM machines - you withdraw $100 and your S.O. deposits $100

Example Continued

- We can represent this situation by creating separate threads for each action, which may run at the bank's central server (or at the ATM):

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

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

- What's wrong with this implementation?
  - Think about potential schedules for these two threads

Interleaved Schedules

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

Schedule A

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

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

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

Context switch

Schedule B

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

```
balance = get_balance(acct);
balance = balance + amt;
```

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

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

- What is the account balance now?
- Is the bank happy with our implementation?
  - Are you?

What Went Wrong

- Two concurrent threads manipulated a shared resource (the account) without any synchronization
  - 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
  - So that we can reason about program behavior

➔ We need synchronization

Caution!

- Bank account problem 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!

Aside: What program data is shared?

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

Mutual Exclusion

- Given:
  - A set of n threads, T_0, T_1, …, 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
  - All other threads are forced to wait on entry
  - When a thread leaves the CS, another can enter

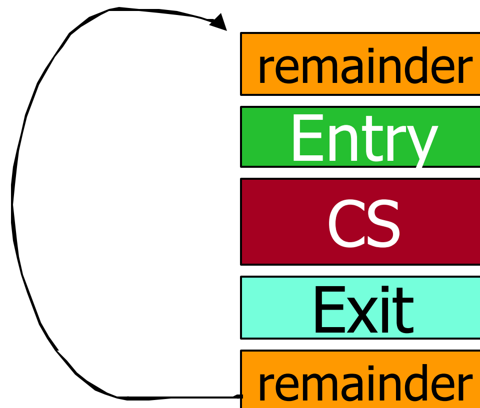
Critical Section Requirements

1) Mutual Exclusion
   - If one thread is in the CS, then no other is
2) Progress
   - If no thread is in the CS, and some threads want to enter CS, only threads not in the "remainder" section can influence the choice of which thread enters next, and choice cannot be postponed indefinitely
3) Bounded waiting (no starvation)
   - If some thread T is waiting on the CS, then there is a limit on the number of times other threads can enter CS before this thread is granted access

- 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
  - CS may be followed by an exit section
  - Remaining code is the remainder section



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

Some Assumptions & Notation

- Assume no special hardware instructions, no restrictions on the # 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 unknown 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
- Let's look at one solution…

2-Thread Solutions: 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 CS

```
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-Thread Solutions: 2nd Try

- First attempt does not have enough info about state of each process. It only remembers which process is allowed to enter its CS
- 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[i]` is true, T_i is ready to enter its CS

A Closer Look at 2nd Attempt

```
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-Thread Solutions: 3rd Try

- Combine key ideas of first two attempts for a correct solution
- The threads share the variables turn and flag (where flag is an array, as before)
  - 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 (text Fig 2.21) (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 id is served first
    - Thread id's are unique and totally ordered

Higher-level Abstractions for CS's

- Locks
  - Very primitive, minimal semantics
- Semaphores
  - Basic, easy to understand, hard to program with
- Monitors
  - High-level, ideally has language support (Java)
- Messages
  - Simple model for communication & synchronization
  - Direct application to distributed systems

Synchronization Hardware

- To build these 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)
- 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;
}
```

Alternate Defn of Test-and-Set

- We'll use "TAS" in the code example for "test-and-set"

```
boolean TAS(boolean *lock) {
        boolean old = *lock;
        *lock = True;
        return old;
}
```

```
boolean TAS(boolean *lock) {
    if(*lock == False) {
        *lock = True;
        return False;
    } else
        return True;
}
```

- lock is always True on exit from test-and-set
  - Either it was True (locked) already, and nothing changed, or it was False (available), but the caller now holds it
- Return value is either True if it was locked already, or False if it was previously available

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 continually executes while loop in acquire() , consumes CPU cycles

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);
```

More Special Instructions

- Swap (or Exchange) instruction
  - Operates on two words atomically
  - Can also be used to solve critical section problem
- Machine instructions have three problems:
  - Busy waiting
  - Starvation is possible (when a thread leaves its CS, the next one to enter depends on scheduling; a waiting thread could be denied entry indefinitely)
  - Deadlock is possible through priority inversion

Semaphores

- Semaphores are data structures that provide synchronization. They include:
  - An integer variable, accessed only through 2 atomic operations
  - The atomic operation wait (also called P or decrement) - decrement the variable and block until semaphore is free
  - The atomic operation signal (also called V or increment) - increment the variable, unblock a waiting a thread if there are any
  - A queue of waiting threads

Types of Semaphores

- Mutex (or Binary) Semaphore
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore
  - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
  - Multiple threads can pass the semaphore
  - Max number of threads is determined by semaphore's initial value, count
    - Mutex has count = 1, counting has count = N

Using Binary Semaphores

- Use is similar to locks, but semantics are different

Have semaphore, S, associated with acct

```
typedef struct account {
    double balance;
    semaphore S;
} account_t;

Withdraw(account_t *acct, amt){
    double bal;
    wait(acct->S);
    bal = acct->balance;
    bal = bal - amt;
    acct->balance = bal;
    signal(acct->S);
    return bal;
}
```

Three threads execute Withdraw()

```
wait(S);
bal = acct->balance;
bal = bal - amt;
```

```
wait(acct->S);
```

```
wait(acct->S);
```

```
acct->balance = bal;
signal(acct->S);
```

```
...
signal(acct->S);
```

```
...
signal(acct->S);
```

It is **undefined** which thread runs after a **signal**

Atomicity of wait() and signal()

- We must ensure that two threads cannot execute wait and signal at the same time
- This is another critical section problem!
  - Use lower-level primitives
  - Uniprocessor: disable interrupts
    - See OS/161 synch.c file
  - Multiprocessor: use hardware instructions

OS/161 Semaphores

kern/thread/synch.c

```
P(struct semaphore *sem) {
    spl = splhigh();
    while (sem->count==0) {
        thread_sleep(sem);
    }
    sem->count--;
    splx(spl);
}
```

```
V(struct semaphore *sem) {
    spl = splhigh();
    sem->count++;
    thread_wakeup(sem);
    splx(spl);
}
```

More Using Semaphores

- Readers/Writers Problem:
  - An object is shared among several threads
  - Some only read the object, others only write it
  - 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);
}
```

Notes on Readers/Writers

- If there is a writer
  - First reader blocks on w_or_r
  - All other readers block on mutex
- Once a writer exits, all readers can proceed
  - Which reader gets to go first?
- The last reader to exit signals a waiting writer
  - If no writer, then readers can continue
- If readers and writers are waiting on w_or_r, and a writer exits, who goes first?
  - Depends on the scheduler

Next Time:

- Monitors
- Atomic Transactions
- Deadlocks