CS 537
Lecture 20
Synchronization

Michael Swift

Questions for this Lecture

- How can multiple threads cooperate?

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

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:

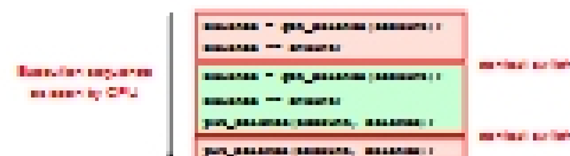
```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
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:



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

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