CSE 124
January 28, 2016
Winter 2016, UCSD
Prof. George Porter

Includes material from Geoff Voelker’s CSE 120 course, used with permission.
Thought question

• When you are speaking with someone, and you both start speaking at the same time, how do you decide who continues to speak, and who stops?
Announcements

• HW 3 due Tuesday (2/2)
• HW 4 due Thursday (2/4), however it is simply an anonymous mid-course survey

• Today’s plan:
  – More on multitasking, multithreading, and sharing state between threads
Synchronization Overview
Synchronization

• Threads cooperate in multithreaded programs
  – To share resources, access shared data structures
    • Threads accessing a memory cache in a Web server
  – To coordinate their execution
    • One thread executes relative to another

• For correctness, we need to control this cooperation
  – Threads *interleave executions arbitrarily* and at *different rates*
  – Scheduling is not under program control

• We control cooperation using *synchronization*
  – Synchronization enables us to restrict the possible interleavings of thread executions

• Discuss in terms of threads, also applies to processes
Shared Resources

We initially focus on coordinating access to shared resources
• Basic problem
  – If two concurrent threads (processes) are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior
• In this lecture, we’ll look at ‘mutexes’, though there are many other techniques that are beyond the scope of 124
Classic Example

• Suppose we have to implement a function to handle withdrawals from a bank account:

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

• Now suppose that you and your significant other share a bank account with a balance of $1000.

• Then you each go to separate ATM machines and simultaneously withdraw $100 from the account.
Example Continued

• We’ll represent the situation by creating a separate thread for each person to do the withdrawals
• These threads run on the same bank machine:

```java
withdraw (account, amount) {
  balance = get_balance(account);
  balance = balance – amount;
  put_balance(account, balance);
  return balance;
}
```

• What’s the problem with this implementation?
  – Think about potential schedules of these two threads

```java
withdraw (account, amount) {
  balance = get_balance(account);
  balance = balance – amount;
  put_balance(account, balance);
  return balance;
}
```
Interleaved Schedules

• The problem is that the execution of the two threads can be interleaved:

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

• What is the balance of the account now?
• Is the bank happy with our implementation?
Shared Resources

• The problem is that two concurrent threads (or processes) accessed a shared resource (account) without any synchronization
  – Known as a race condition (memorize this buzzword)
• We need mechanisms to control access to these shared resources in the face of concurrency
  – So we can reason about how the program will operate
• Our example was updating a shared bank account
• Also necessary for synchronizing access to any shared data structure
  – Buffers, queues, lists, hash tables, etc.
When Are Resources Shared?

- Local variables are **not shared** (private)
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2

- Global variables and static objects are **shared**
  - Stored in the static data segment, accessible by any thread

- Dynamic objects and other heap objects are **shared**
  - Allocated from heap with malloc/free or new/delete
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
  – Example: sharing your bathroom with housemates

• What requirements would you place on a critical section?
Critical Section Requirements

1) **Mutual exclusion (mutex)**
   - If one thread is in the critical section, then no other is

2) **Progress**
   - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
   - A thread in the critical section will eventually leave it

3) **Bounded waiting (no starvation)**
   - If some thread T is waiting on the critical section, then T will eventually enter the critical section

4) **Performance**
   - The overhead of entering and exiting the critical section is small with respect to the work being done within it
About Requirements

There are three kinds of requirements that we'll use

• **Safety property**: nothing bad happens
  – Mutex

• **Liveness property**: something good happens
  – Progress, Bounded Waiting

• **Performance requirement**
  – Performance

• Properties hold for **each run**, while performance depends on **all the runs**
  – Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don't forget liveness!).
Locks

• A lock is an object in memory providing two operations
  – acquire(): to enter a critical section
  – release(): to leave a critical section

• Threads pair calls to acquire and release
  – Between acquire/release, the thread holds the lock
  – acquire does not return until any previous holder releases
  – What can happen if the calls are not paired?

• Locks can spin (a spinlock) or block (a mutex)
Using Locks

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

– What happens when blue tries to acquire the lock?
– Why is the “return” outside the critical section? Is this ok?
– What happens when a third thread calls acquire?
Synchronization in action
POSIX threads mutexes

• There are many synchronization mechanisms
  – Monitors and condition variables
  – Semaphores
  – Atomic instructions + lock free data structures
  – Message passing
  – ...

• In 124 we’ll look at mutexes provided by pthraeds
pthreads API

NAME

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

SYNOPSIS

#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

• pthread_mutex_lock acquires the mutex, unlock releases it
  – Blocking until the lock is acquired
• pthread_mutex_trylock tells you if you would have acquired the lock if you had tried
  – But not blocking
Demo: unprotected sharing
Demo: protected sharing