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
- Over the next two lectures, 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:

```c
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:

```c
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:
- Interleaved Schedules

- 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
  - 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

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

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

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

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

Spinlocks

• How do we implement locks? Here’s one attempt:

```
struct lock {
    int held = 0;
}
void acquire(lock) {
    while (lock->held);
    lock->held = 1;
}
void release(lock) {
    lock->held = 0;
}
```

• Why doesn’t this work?
  – where is the race condition?

Implementing locks (continued)

• Problem is that implementation of locks has critical sections, too!
  – the acquire/release must be atomic
    • atomic == executes as though it could not be interrupted
    • code that executes "all or nothing"
• Need help from the hardware
  – atomic instructions
    • test-and-set, compare-and-swap, ...
  – disable/reenable interrupts
    • to prevent context switches

```
Spinlocks redux: Test-and-Set

• CPU provides the following as one atomic instruction:

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

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 {
    int held = 0;
}
void acquire(lock) {
    cli();   // disable interrupts
    lock->held = 1;
}
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