Recall: How does Thread get started?

- Eventually, `run_new_thread()` will select this TCB and return into the beginning of `ThreadRoot()`
  - This really starts the new thread

Goals for Today

- Synchronization Operations
- Higher-level Synchronization Abstractions
  - Semaphores, monitors, and condition variables
- Programming paradigms for concurrent programs

Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic \( \Rightarrow \) Input state determines results
  - Reproducible \( \Rightarrow \) Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
  - Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.
Interactions Complicate Debugging

• Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc
  - Extreme example: buggy device driver causes thread A to crash "independent thread" B
• You probably don’t realize how much you depend on reproducibility:
  - Example: Evil C compiler
    » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack
• Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    » depends on scheduling, which depends on timer/other things
  - Original UNIX had a bunch of non-deterministic errors
  - Example: Something which does interesting I/O
    » User typing of letters used to help generate secure keys

Why allow cooperating threads?

• People cooperate: computers help/enhance people's lives, so computers must cooperate
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for "carefully laid plans"
• Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
• Advantage 2: Speedup
  - Overlap I/O and computation
    » Many different file systems do read-ahead
  - Multiprocessors - chop up program into parallel pieces
• Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    » Makes system easier to extend

High-level Example: Web Server

• Server must handle many requests
• Non-cooperating version:
  serverLoop() {
    con = AcceptCon();
    ProcessFork(ServiceWebPage(),con);
  }
• What are some disadvantages of this technique?

Threaded Web Server

• Now, use a single process
• Multithreaded (cooperating) version:
  serverLoop() {
    connection = AcceptCon();
    ThreadFork(ServiceWebPage(),connection);
  }
• Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are much cheaper to create than processes, so this has a lower per-request overhead
• Question: would a user-level (say one-to-many) thread package make sense here?
  - When one request blocks on disk, all block...
• What about Denial of Service attacks or digg / Slash-dot effects?
Thread Pools

- Problem with previous version: Unbounded Threads
  - When website becomes too popular - throughput sinks
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming

master() {
  allocThreads(worker, queue);
  while(TRUE) {
    con = AcceptCon();
    Enqueue(queue, con);
    wakeUp(queue);
  }
}

worker(queue) {
  while(TRUE) {
    con = Dequeue(queue);
    if (con == null)
      sleepOn(queue);
    else
      ServiceWebPage(con);
  }
}

ATM Bank Server

- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don't hand out too much money

ATM bank server example

- Suppose we wanted to implement a server process to handle requests from an ATM network:

BankServer() {
  while (TRUE) {
    ReceiveRequest(&op, &acctId, &amount);
    ProcessRequest(op, acctId, amount);
  }
}

ProcessRequest(op, acctId, amount) {
  if (op == deposit) Deposit(acctId, amount);
  else if ...
}

Deposit(acctId, amount) {
  acct = GetAccount(acctId); /* may use disk I/O */
  acct->balance += amount;
  StoreAccount(acct); /* Involves disk I/O */
}

- How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-processor, or overlap comp and I/O)

Event Driven Version of ATM server

- Suppose we only had one CPU
  - Still like to overlap I/O with computation
  - Without threads, we would have to rewrite in event-driven style
- Example

  BankServer() {
    event = WaitForNextEvent();
    if (event == ATMRequest)
      StartOnRequest();
    else if (event == AcctAvail)
      ContinueRequest();
    else if (event == AcctStored)
      FinishRequest();
  }

- What if we missed a blocking I/O step?
- What if we have to split code into hundreds of pieces which could be blocking?
- This technique is used for graphical programming
Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without “deconstructing” code into non-blocking fragments
  - One thread per request
- Requests proceed to completion, blocking as required:
  ```
  Deposit(actId, amount) {
    acct = GetAccount(actId); /* May use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
  }
  ```
- Unfortunately, shared state can get corrupted:

```
Thread 1                  Thread 2
load r1, acct->balance   load r1, acct->balance
add r1, amount1          add r1, amount2
store r1, acct->balance  store r1, acct->balance
```

Review: Multiprocessing vs Multiprogramming

- What does it mean to run two threads “concurrently”?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, ...
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks

```
Multiprocessing
A --> B --> C

Multiprogramming
A --> B --> C --> A --> B --> C
```

- Also recall: Hyperthreading
  - Possible to interleave threads on a per-instruction basis
  - Keep this in mind for our examples (like multiprocessing)

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn’t matter:

```
Thread A                  Thread B
x = 1;                    y = 2;
```

- However, What about (Initially, y = 12):

```
Thread A                  Thread B
x = 1;                    y = 2;
x = y+1;                   y = y*2;
```

- What are the possible values of x?

- Or, what are the possible values of x below?

```
Thread A                  Thread B
x = 1;                    x = 2;
```

- X could be 1 or 2 (non-deterministic!)  
- Could even be 3 for serial processors:
  - Thread A writes 0001, B writes 0010.
  - Scheduling order ABABABABA yields 3!

Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- Atomic Operation: an operation that always runs to completion or not at all
  - It is indivisible: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
  - Consequently – weird example that produces “3” on previous slide can’t happen
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array
Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- Example: Therac-25
  - Machine for radiation therapy
    - Software control of electron accelerator and electron beam/Xray production
  - Software control of dosage
  - Software errors caused the death of several patients
    - A series of race conditions on shared variables and poor software design
    - "They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred."

Space Shuttle Example

- Original Space Shuttle launch aborted 20 minutes before scheduled launch
- Shuttle has five computers:
  - Four run the "Primary Avionics Software System" (PASS)
    - Asynchronous and real-time
    - Runs all of the control systems
    - Results synchronized and compared every 3 to 4 ms
  - The Fifth computer is the "Backup Flight System" (BFS)
    - stays synchronized in case it is needed
    - Written by completely different team than PASS
- Countdown aborted because BFS disagreed with PASS
  - A 1/67 chance that PASS was out of sync one cycle
  - Bug due to modifications in initialization code of PASS
    - A delayed init request placed into timer queue
    - As a result, timer queue not empty at expected time to force use of hardware clock
  - Bug not found during extensive simulation

Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0;</td>
<td>i = 0;</td>
</tr>
<tr>
<td>while (i &lt; 10)</td>
<td>while (i &gt; -10)</td>
</tr>
<tr>
<td>i = i + 1;</td>
<td>i = i - 1;</td>
</tr>
<tr>
<td>printf(&quot;A wins!&quot;);</td>
<td>printf(&quot;B wins!&quot;);</td>
</tr>
</tbody>
</table>

- Assume that memory loads and stores are atomic, but incrementing and decrementing are not atomic
- Who wins? Could be either
- Is it guaranteed that someone wins? Why or why not?
- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

Hand Simulation Multiprocessor Example

- Inner loop looks like this:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>rl=0</td>
<td>rl=0</td>
</tr>
<tr>
<td>load rl, M[i]</td>
<td>load rl, M[i]</td>
</tr>
<tr>
<td>rl=1</td>
<td>rl=-1</td>
</tr>
<tr>
<td>add rl, rl, 1</td>
<td>sub rl, rl, 1</td>
</tr>
<tr>
<td>M[i]=1</td>
<td>M[i]=-1</td>
</tr>
<tr>
<td>store rl, M[i]</td>
<td>store rl, M[i]</td>
</tr>
</tbody>
</table>

- Hand Simulation:
  - And we're off. A gets off to an early start
  - B says "hmph, better go fast" and tries really hard
  - A goes ahead and writes "1"
  - B goes and writes "-1"
  - A says "HUH?? I could have sworn I put a 1 there"
- Could this happen on a uniprocessor?
  - Yes! Unlikely, but if you are depending on it not happening, it will and your system will break...
**Administrivia**

- Don’t Forget New Section!
  - Thursday 12-1, 320 Soda Hall
  - Need to know your TA!
- Sorry about HW 1
  - Got a little longer than we expected
  - Due next Monday! (HW 2 not handed out until Monday)
- No class on Monday! (Holiday)

---

**Motivation: “Too much milk”**

- Great thing about OS’s – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in Fridge. Out of milk</td>
<td>Look in Fridge. Out of milk</td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home, put milk away</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:25</td>
<td></td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td>Arrive home, put milk away</td>
</tr>
</tbody>
</table>

---

**Definitions**

- **Synchronization**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that its hard to build anything useful with only reads and writes
- **Mutual Exclusion**: ensuring that only one thread does a particular thing at a time
  - One thread excludes the other while doing its task
- **Critical Section**: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

---

**More Definitions**

- **Lock**: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
  - Important idea: all synchronization involves waiting
- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants OJ
  - Of Course – We don’t know how to make a lock yet
Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Always write down behavior first
  - Impulse is to start coding first, then when it doesn’t work, pull hair out
  - Instead, think first, then code
- What are the correctness properties for the “Too much milk” problem???
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):
  
  ```
  if (noMilk) {
      if (noNote) {
          leave Note; buy milk; remove note;
      }
  }
  ```

- Result?
  - Still too much milk but only occasionally!
  - Solution makes problem worse since fails intermittently
  - Makes it really hard to debug...
  - Must work despite what the dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let’s try to fix this by placing note first
- Another try at previous solution:
  
  ```
  leave Note;
  if (noMilk) {
      if (noNote) {
          leave Note; buy milk;
      }
  }
  remove note;
  ```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:
  
  ```
  Thread A
  leave note A;
  if (noNote B) {
      if (noMilk) {
          leave Note; buy Milk;
      }
  }
  remove note A;

  Thread B
  leave note B;
  if (noNote A) {
      if (noMilk) {
          buy Milk;
      }
  }
  remove note B;
  ```

- Does this work?
  - Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
  - Really insidious:
    - Extremely unlikely that this would happen, but will at worse possible time
    - Probably something like this in UNIX
Too Much Milk Solution #2: problem!

- *I'm* not getting milk, *You're* getting milk
- This kind of lockup is called “starvation!”

Too Much Milk Solution #3

- Here is a possible two-note solution:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>leave note A;</td>
<td>leave note B;</td>
</tr>
<tr>
<td>while (note B) { //X if (noNote A) { //Y</td>
<td></td>
</tr>
<tr>
<td>do nothing;</td>
<td>if (noMilk) {</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>if (noMilk) {</td>
<td>buy milk;</td>
</tr>
<tr>
<td>buy milk;</td>
<td>}</td>
</tr>
<tr>
<td>remove note A;</td>
<td>remove note B;</td>
</tr>
</tbody>
</table>

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At Y:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Solution #3 discussion

- Our solution protects a single “Critical-Section” piece of code for each thread:
  ```java
  if (noMilk) {
    buy milk;
  }
  ```
- Solution #3 works, but it’s really unsatisfactory
  - Really complex - even for this simple an example
    » Hard to convince yourself that this really works
  - A’s code is different from B’s – what if lots of threads?
    » Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    » This is called “busy-waiting”
- There’s a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock (more in a moment).
  - Lock.Acquire() - wait until lock is free, then grab
  - Lock.Release() - Unlock, waking up anyone waiting
  - These must be atomic operations - if two threads are waiting for the lock and both see it’s free, only one succeeds to grab the lock
- Then, our milk problem is easy:
  ```java
  milklock.Acquire();
  if (nomilk)
    buy milk;
  milklock.Release();
  ```
- Once again, section of code between Acquire() and Release() called a “Critical Section”
- Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  - Skip the test since you always need more ice cream.
Where are we going with synchronization?

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

<table>
<thead>
<tr>
<th>Programs</th>
<th>Shared Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher-level API</td>
<td>Locks  Semaphores  Monitors  Send/Receive</td>
</tr>
<tr>
<td>Hardware</td>
<td>Load/Store  Disable Ints  Test&amp;Set  Comp&amp;Swap</td>
</tr>
</tbody>
</table>

How to implement Locks?

- **Lock**: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - **Wait if locked**
    - Important idea: all synchronization involves waiting
    - Should sleep if waiting for a long time
- **Atomic Load/Store**: get solution like Milk #3
  - Looked at this last lecture
  - Pretty complex and error prone
- **Hardware Lock instruction**
  - Is this a good idea?
  - What about putting a task to sleep?
    - How do you handle the interface between the hardware and scheduler?
    - Complexity?
      - Done in the Intel 432
      - Each feature makes hardware more complex and slow

How can we build multi-instruction atomic operations?

- Recall: dispatcher gets control in two ways.
  - Internal: Thread does something to relinquish the CPU
  - External: Interrupts cause dispatcher to take CPU
- On a uniprocessor, can avoid context-switching by:
  - Avoiding internal events (although virtual memory tricky)
  - Preventing external events by disabling interrupts

Consequently, naïve Implementation of locks:

```
int value = FREE;

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (anyone on wait queue) {
        take thread off wait queue
        Place on ready queue;
    } else {
        value = FREE;
    }
    enable interrupts;
}
```

Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```c
int value = FREE;

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (anyone on wait queue) {
        take thread off wait queue
        Place on ready queue;
    } else {
        value = FREE;
    }
    enable interrupts;
}
```
New Lock Implementation: Discussion

- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```c
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue; Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

- Note: unlike previous solution, the critical section (inside Acquire()) is very short
  - User of lock can take as long as they like in their own critical section: doesn't impact global machine behavior
  - Critical interrupts taken in time!

Interrupt re-enable in going to sleep

- What about re-enabling ints when going to sleep?

```c
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue; Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

How to Re-enable After Sleep()?

- In scheduler, since interrupts are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts

```
Thread A
  disable ints
  sleep
  context

Thread B
  sleep
  return
  switch
  enable ints
```

Atomic Read-Modify-Write instructions

- Problems with previous solution:
  - Can't give lock implementation to users
  - Doesn't work well on multiprocessor
    - Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative: atomic instruction sequences
  - These instructions read a value from memory and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - on both uniprocessors (not too hard)
    - and multiprocessors (requires help from cache coherence protocol)
- Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors
Examples of Read-Modify-Write

- **test&set (address)** { /* most architectures */
  result = M[address];
  M[address] = 1;
  return result;
  }
- **swap (address, register)** { /* x86 */
  temp = M[address];
  M[address] = register;
  register = temp;
  }
- **compare&swap (address, reg1, reg2)** { /* 68000 */
  if (reg1 == M[address]) {
    M[address] = reg2;
    return success;
  } else {
    return failure;
  }
  }
- **load-linked&store conditional (address)** { /* R4000, alpha */
  loop:
  ll r1, M[address];
  movi r2, 1; /* Can do arbitrary comp */
  sc r2, M[address];
  beqz r2, loop;
  }

Implementing Locks with test&set

- Another flawed, but simple solution:
  int value = 0; // Free
  Acquire() {
    while (test&set(value)); // while busy
  }
  Release() {
    value = 0;
  }

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock

- **Busy-Waiting**: thread consumes cycles while waiting

Problem: Busy-Waiting for Lock

- **Positives for this solution**
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor

- **Negatives**
  - This is very inefficient because the busy-waiting thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - **Priority Inversion**: If busy-waiting thread has higher priority than thread holding lock \( \Rightarrow \) no progress!
  - Priority Inversion problem with original Martian rover
  - For semaphores and monitors, waiting thread may wait for an arbitrary length of time!
  - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
  - Homework/exam solutions should not have busy-waiting!

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

  ```c
  int guard = 0;
  int value = FREE;
  Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
    } else {
      value = BUSY;
      guard = 0;
    }
  }
  Release() {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
    } else {
      value = FREE;
      guard = 0;
    }
  }
  ```

  - Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?
Higher-level Primitives than Locks

- Goal of last couple of lectures:
  - What is the right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so - concurrency bugs
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - This lecture and the next presents a couple of ways of structuring the sharing

Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    » This of this as the signal() operation
- Note that P() stands for “proberen” (to test) and V() stands for “verhogen” (to increment) in Dutch

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V - can’t read or write value, except to set it initially
  - Operations must be atomic
    » Two P’s together can’t decrement value below zero
    » Similarly, thread going to sleep in P won’t miss wakeup from V - even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:

Two Uses of Semaphores

- Mutual Exclusion (initial value = 1)
  - Also called “Binary Semaphore”.
  - Can be used for mutual exclusion:
    ```plaintext
    semaphore.P();
    // Critical section goes here
    semaphore.V();
    ```
- Scheduling Constraints (initial value = 0)
  - Locks are fine for mutual exclusion, but what if you want a thread to wait for something?
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:
    ```plaintext
    Initial value of semaphore = 0
    ThreadJoin {
      semaphore.P();
    }
    ThreadFinish {
      semaphore.V();
    }
    ```
Producers-consumer with a bounded buffer

- **Problem Definition**
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
- **Example 1:** GCC compiler
  - `cpp | cc1 | cc2 | as | ld`
- **Example 2:** Coke machine
  - Producer can put limited number of cokes in machine
  - Consumer can’t take cokes out if machine is empty

Correctness constraints for solution

- **Correctness Constraints:**
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- **General rule of thumb:**
  - Use a separate semaphore for each constraint

Full Solution to Bounded Buffer

```c
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers; // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
    emptyBuffers.P(); // Wait until space
    mutex.P(); // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V(); // Tell consumers there is more coke
}

Consumer() {
    fullBuffers.P(); // Check if there’s a coke
    mutex.P(); // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // tell producer need more
    return item;
}
```

Discussion about Solution

- **Why asymmetry?**
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`
- **Is order of P’s important?**
- **Is order of V’s important?**
- **What if we have 2 producers or 2 consumers?**
  - Do we need to change anything?
Motivation for Monitors and Condition Variables

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores
  - Problem is that semaphores are dual purpose:
    » They are used for both mutex and scheduling constraints
    » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?
- Cleaner idea: Use locks for mutual exclusion and condition variables for scheduling constraints
- Definition: Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables

Monitor with Condition Variables

- Lock: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- Condition Variable: a queue of threads waiting for something inside a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

Simple Monitor Example

- Here is an (infinite) synchronized queue

  ```
  Lock lock;
  Condition dataready;
  Queue queue;

  AddToQueue(item) {
    lock.Acquire(); // Get Lock
    queue.enqueue(item); // Add item
    dataready.signal(); // Signal any waiters
    lock.Release(); // Release Lock
  }

  RemoveFromQueue() {
    lock.Acquire(); // Get Lock
    while (queue.isEmpty()) { // If nothing, sleep
      dataready.wait(&lock); // If nothing, sleep
    }
    item = queue.dequeue(); // Get next item
    lock.Release(); // Release Lock
    return(item);
  }
  ```

Summary

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, comp&swap, load-linked/store conditional
- Showed several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    » Shouldn't disable interrupts for long
    » Shouldn't spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable
- Talked about Semaphores, Monitors, and Condition Variables
  - Higher level constructs that are harder to “screw up”