Operating Systems
(1DT020 & 1TT802)

Lecture 5
Process Synchronization : mutual exclusion, critical sections
April 21, 2008

Léon Mugwaneza

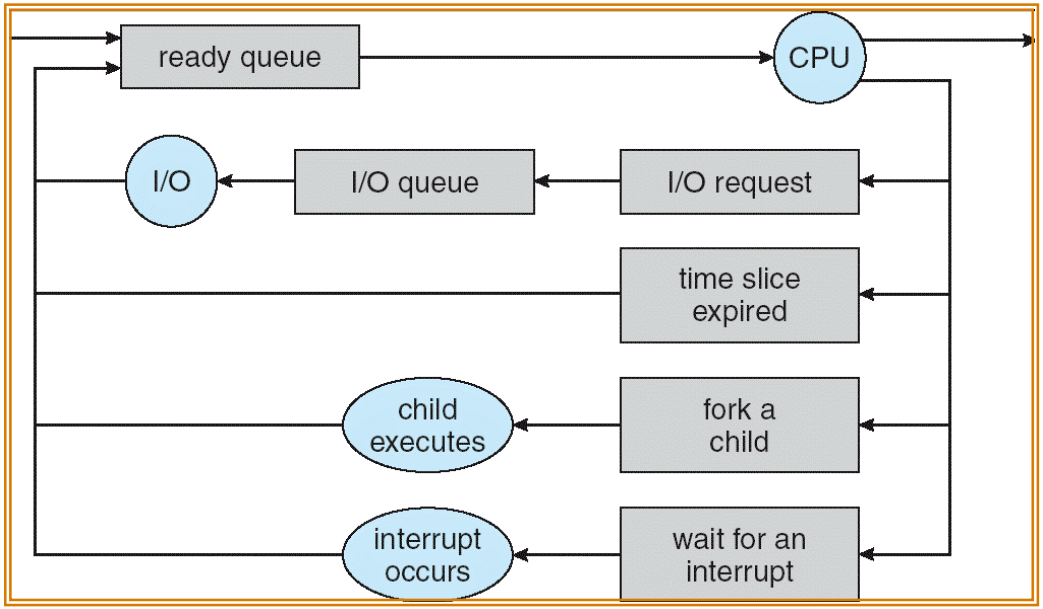
http://www.it.uu.se/edu/course/homepage/os/vt08

Goals for Today

- A final word on scheduling
- Concurrency examples
- Need for synchronization
- Examples of valid synchronization

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)

Review : CPU Scheduling



☛ **Scheduling: selecting a process/thread from the ready queue and allocating the CPU to it**

- **Scheduling Policy Goals/Criteria**
  - **Minimize Response Time [elapsed time to do an operation - or job]**
    - » **Response time is what the user sees (real-time tasks must meet deadlines imposed by World)**
  - **Maximize Throughput [#operations (or jobs) per second]**
    - » **Two parts to maximizing throughput**
      - **Minimize overhead (for example, context-switching)**
      - **Efficient use of resources (CPU, disk, memory, etc)**
  - **Fairness [share CPU among users in some equitable way]**
    - » **Better *average* response time by making system *less* fair**

Review : CPU Scheduling

- **FCFS Scheduling**:
  - Run threads to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones
- **Round-Robin Scheduling**:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: Poor when jobs are same length
- **Shortest Job First (SJF)/Shortest Remaining Time First (SRTF)**:
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- **Multi-Level Feedback Scheduling**:
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example

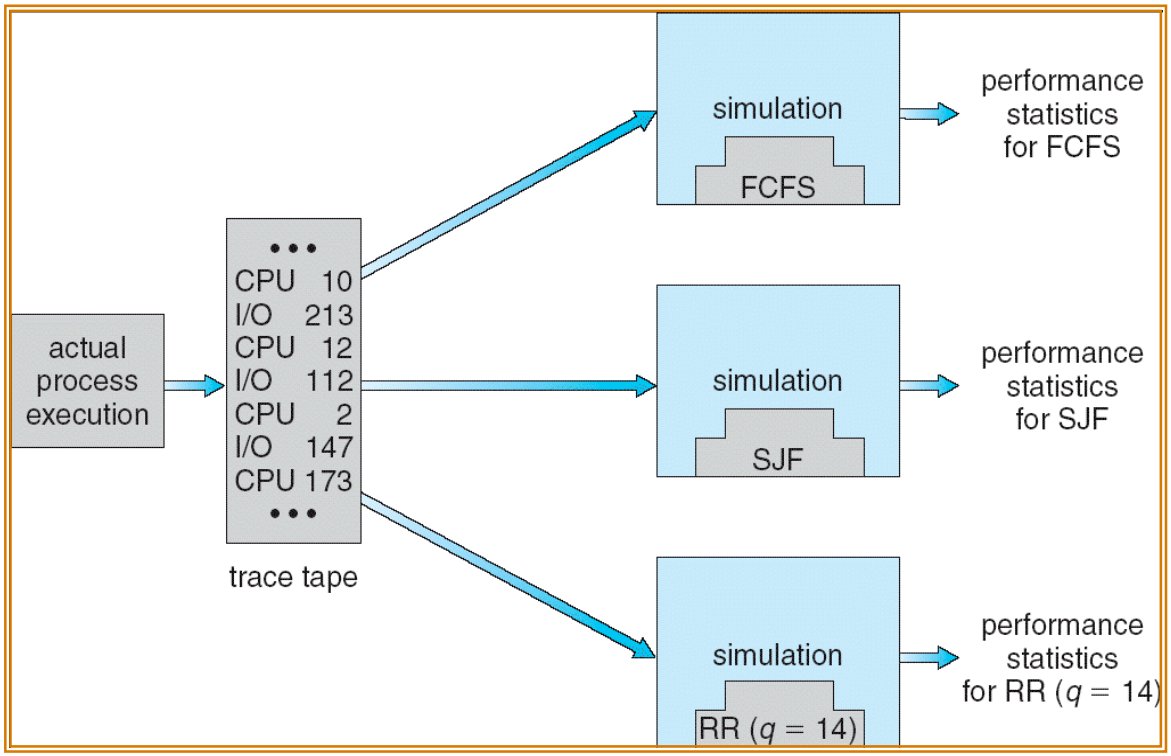
- **Lottery Scheduling Example**
  - **Assume short jobs get 10 tickets, long jobs get 1 ticket**

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

  - **What if too many short jobs to give reasonable response time?**
    - » **In UNIX, if load average is 100, hard to make progress**
    - » **One approach: log some user out**

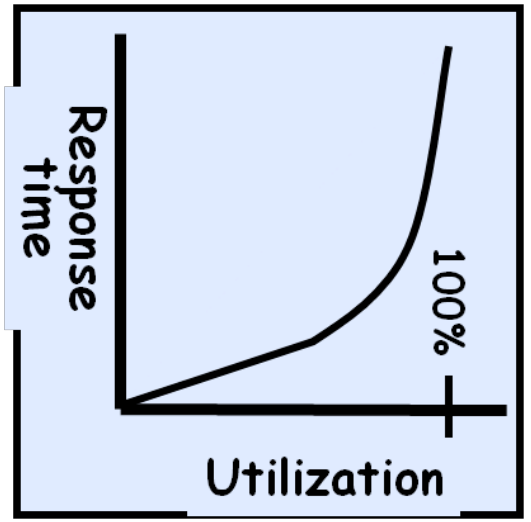
How to Evaluate a Scheduling algorithm?

- **Deterministic modeling**
  - **takes a predetermined workload and compute the performance of each algorithm for that workload**
- **Queuing models**
  - **Mathematical approach for handling stochastic workloads**
- **Implementation/Simulation:**
  - **Build system which allows actual algorithms to be run against actual data. Most flexible/general.**



A Final Word on Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - » Assuming you're paying for worse response time in reduced productivity, customer angst, etc…
    - » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization⇒100%



- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit "knee" of curve

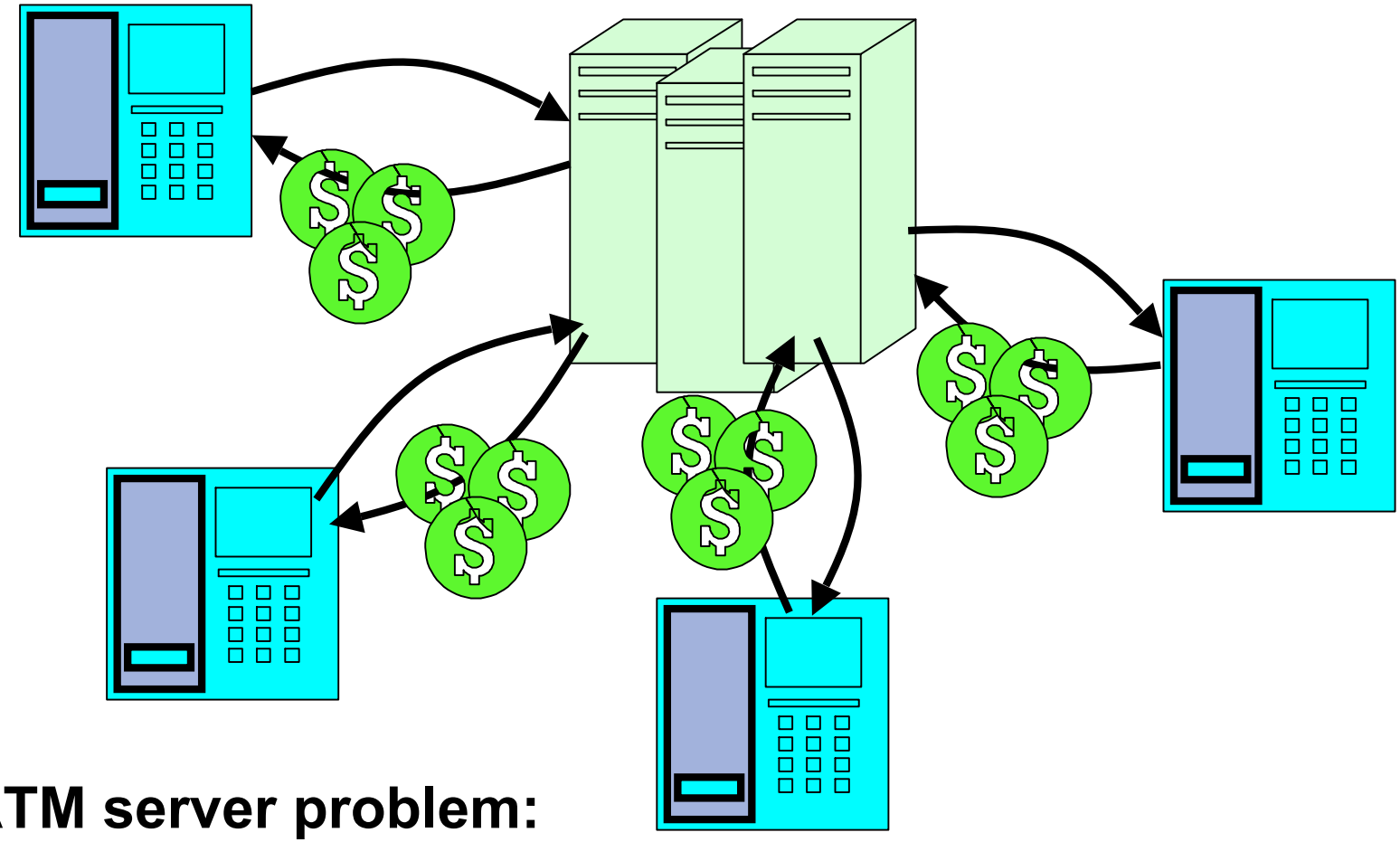
Cooperating processes/threads

- **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**

Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
- Independent Threads:
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ 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"

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-proc, 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() {
    while(TRUE) {
        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 proceeds to completion, blocking as required:**

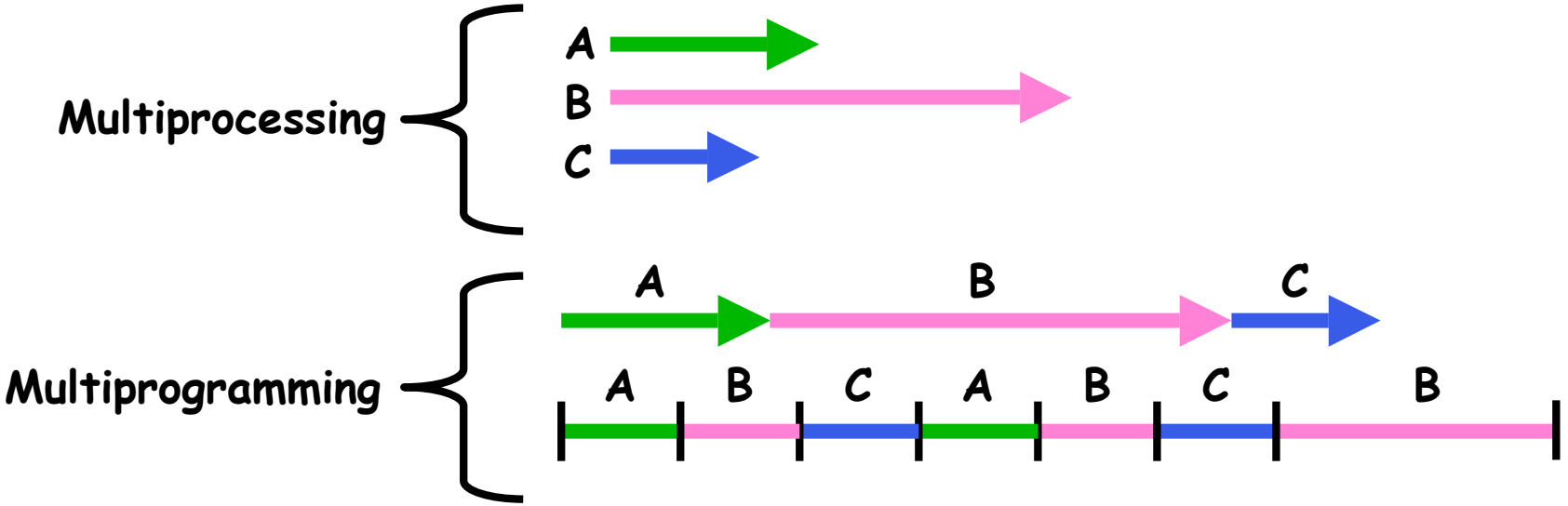
```
Deposit(acctId, amount) {
  acct = GetAccount(actId); /* May use disk I/O */
  acct->balance += amount;
  StoreAccount(acct);       /* Involves disk I/O */
}
```

- **Unfortunately, shared state can get corrupted:**

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



- 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 ABABABBA yields 3!

☛ Race condition

Race conditions

- **What is a race condition?**
  - **two or more threads have an inconsistent view of a shared memory region (I.e., a variable)**
- **Why do race conditions occur?**
  - **values of memory locations replicated in registers during execution**
  - **context switches at arbitrary times during execution**
  - **threads can see "stale" memory values in registers**

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
- 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

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

Thread A

```
i = 0;
while (i < 10)
    i = i + 1;
printf("A wins!");
```

Thread B

```
i = 0;
while (i > -10)
    i = i - 1;
printf("B wins!");
```

- 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

- "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:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

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 in which a thread can be accessing shared data. When the thread is executing in its critical section, no other thread is allowed to execute in its critical section (wrt the same shared data).**
  - **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**

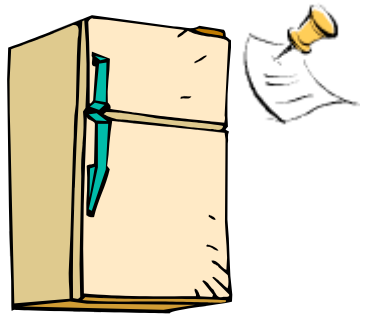
Too Much Milk: Correctness Properties

- 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 (attend to your other own affairs)
- 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!
  - Thread can get context switched after checking milk and note but before leaving note !
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

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) {
    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 #3

- **Here is a possible two-note solution:**

Thread A

```
leave note A;
while (note B) { //X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) { //Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- **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:

```
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 the next lecture)
  - `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:

```
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
  - Only Critical Sections need to be accessed in a synchronized way

Where are we going with synchronization?

Programs	Cooperating Processes			
Higher-level API	Locks	Semaphores	Monitors	Send/Receive
Hardware	Load/Store	Disable Ints	Test&Set	Comp&Swap

- 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

Summary

- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available
- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent
- 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
- Showed how to protect a critical section with only atomic load and store ⇒ pretty complex!