Threads, Concurrency, and Mutual Exclusion (Too Much Milk!)

CS439: Principles of Computer Systems

February 4, 2015

Last Time

CPU Scheduling

- discussed the possible policies the scheduler may use to choose the next process (or thread!) to run
- criteria we use to evaluate policies
  - throughput, turnaround time, response time, CPU utilization, waiting time
- FIFO, Round Robin, SJF, Multilevel Feedback Queues

Today's Agenda

- Threads
  - Differences from processes
  - User vs. kernel
  - Creating, dispatching
  - Independent vs. Cooperating
- Too Much Milk
  - Race conditions, critical sections, and mutual exclusion

Threads

Processes: What We Think We Know

- A process is the abstraction used by the OS to manage resources and provide protection
- A process defines an address space
  - Identifies all addresses that may be touched by the program
- A process has a single *thread of control* that executes instructions sequentially
  - Creates a single sequential execution stream

What if we decouple the thread of control information from the process?

Threads

- A *thread* represents an abstract entity that executes a sequence of instructions
  - Short for "Thread of Control"
  - Defines a single sequential execution stream within a process
- A thread is bound to a single process
- Each process may have multiple threads of control
  - *Must* have one
  - Virtualizes the processor

Why Threads?

- Programmers create *multi-threaded* programs to:
  - Better represent the structure of the tasks
    - We think linearly
    - But the world is concurrent!
  - Improve performance
    - One thread can perform computation while another waits for I/O
    - Threads may be scheduled across different processors in a multi-processor architecture
- First, concrete examples of usefulness, then implementation, then interaction (or not) with the OS (which is based on thread type)

The Case for Threads: Web Servers

Consider a web server that performs these actions:

```
while()
    get network message (URL) from client
    get URL data from disk
    compose response
    send response
```

How well does this web server perform?

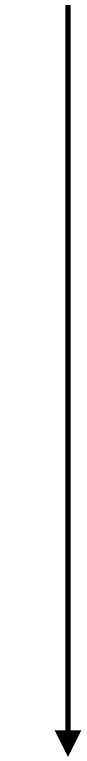
The Case for Threads: Web Servers

Consider a web server that performs these actions:

- Create a number of threads, and for each thread do:
  - get network message (URL) from client
  - get URL data from disk
  - compose response
  - send response

How well does this web server perform?

Overlapping Requests (Concurrency)



Time

Thread One, Request One

- get network message (URL) from client
- get URL data from disk

---disk access latency---

- send data over network

Thread Two, Request Two

- get network message (URL) from client
- get URL data from disk

---disk access latency---

- send data over network

=> Total time is less than request 1 + request 2

The Case for Threads: Arrays

Consider the following code fragment:

```
for(k = 0; k < n; k++)
    a[k] = b[k] * c[k] + d[k] * e[k];
```

Is there a missed opportunity?

The Case for Threads: Arrays

- How can this code take advantage of 2 threads?

```
for(k=0; k<n; k++)
  a[k]=b[k] * c[k] + d[k] * e[k];
```

- Rewrite this code fragment so that one thread computes the first half of `a` and the other computes the second half.
  - We'll see *how* to do this in a minute

- What did we gain?

Programmer's View

```
void thread_function(int arg0, int arg1, …) {…}


main() {
   …
   tid = thread_create(thread_function, arg0, arg1, …);
   …
}
```

At the point `thread_create()` is called:

- execution continues with the original thread in `main` function, and
- execution starts at `thread_function()` in new thread

in parallel (concurrently).

Programmer's View: Array Example

How can this code take advantage of 2 threads?

```
for(k = 0; k < n; k++)
  a[k] = b[k] * c[k] + d[k] * e[k];
```

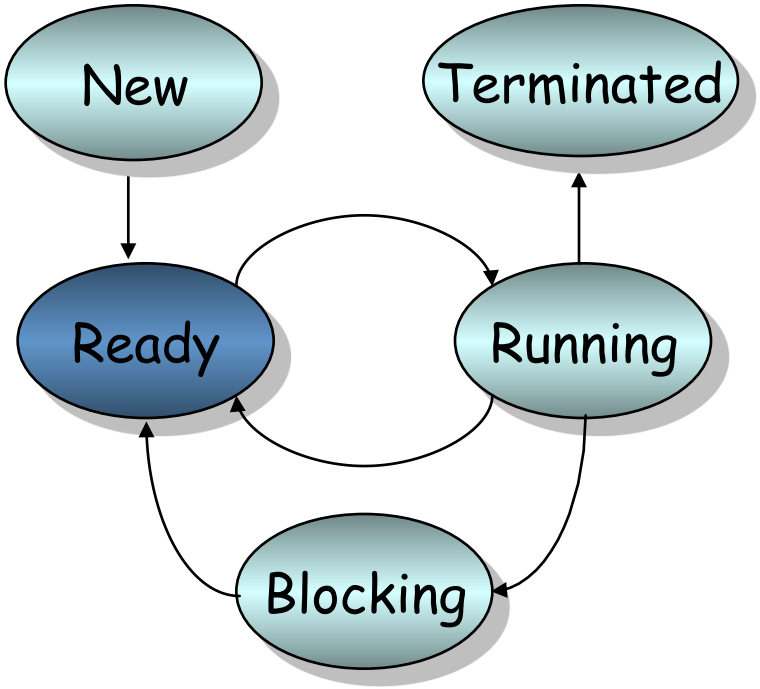
Rewrite this code fragment as:

```
do_mult(p, m) {              /*thread function*/
  for(k = p; k < m; k++)
   a[k] = b[k] * c[k] + d[k] * e[k];
}
main() {
   /*args are thread function name and then its args*/
   thread_create(do_mult, 0, n/2);
   thread_create(do_mult, n/2, n);
```

Threads: A Closer Look

Threads: A Closer Look

Threads (just like processes) go through a sequence of *new*, *ready*, *running*, *blocking*, and *terminated* states



Threads: A Closer Look

- Processes *define* an address space; threads *share* the address space
- Each thread has:
  - Its own stack
  - Exclusive use of the CPU registers while it is executing
- Each thread does NOT have:
  - Its own address space
    - Shared amongst all threads in the process
  - *What does that imply for process data?*
- So, threads are lightweight:
  - Creating a thread is cheaper than creating a process
  - Communication between threads is easier than between processes
    - Processes must set up a shared resource or pass messages or signals
  - Context switching between threads is cheaper (no address space switching!)

Threads and the Address Space

Threads within a single process *share* the address space

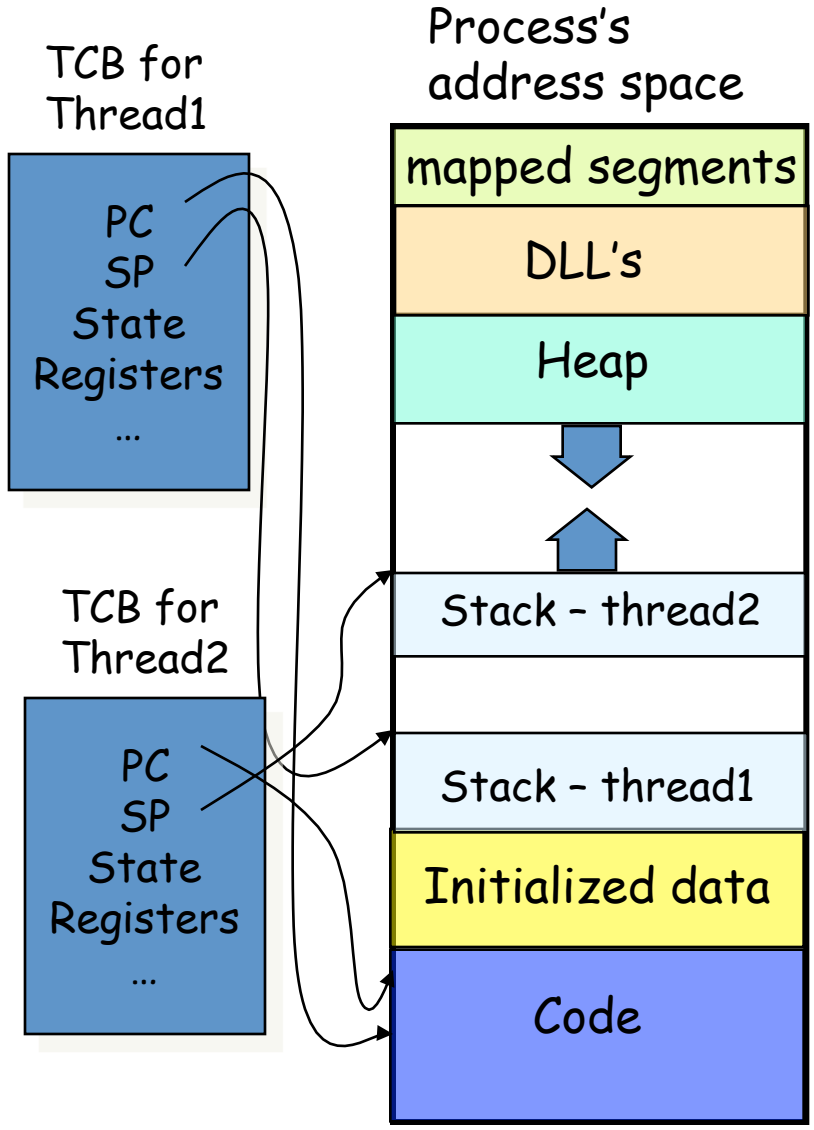
- All process data can be accessed by any thread
  - Particularly global data
  - Heap is also shared (*What about pointers into the heap?*)
- Threads have their own stacks, yes, BUT there is no protection
  - So any thread can modify another thread's stack
  - This is usually a bug

Threads and Registers

- A thread has exclusive use of the registers while it is executing
- When a thread is pre-empted, its register values are saved as part of its state
  - the new thread gets to use the registers!

Metadata Structures

- Process Control Block (PCB) contains process-specific information
  - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- Thread Control Block (TCB) contains thread-specific information
  - Stack pointer, PC, thread state (running, …), register values, a pointer to PCB, …



iClicker Question

Threads have their own …?

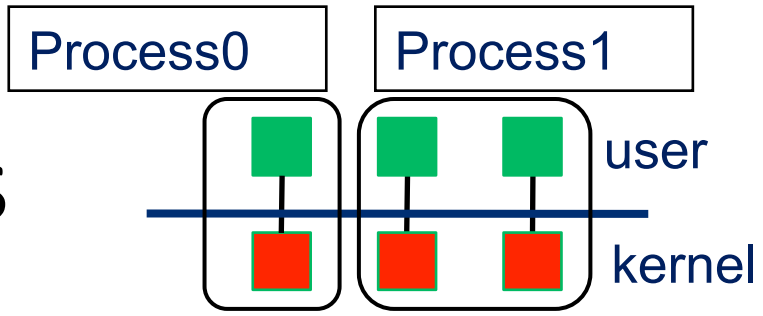
A. Address Space
B. PCB
C. Stack

Threads vs. Processes

Threads	Processes
A thread has no code or data segment or heap of its own. Each has its own stack & registers	A process has code/data/heap & other segments of its own. A process also has its own registers.
A thread cannot live on its own, it must live within a process. There can be more than one thread in a process---the original thread calls main and has the process's stack.	There must be at least one thread in a process.
If a thread dies, its stack is reclaimed	If a process dies, its resources are reclaimed & all threads die
Each (kernel) thread can run on a different physical processor	Each process can run on a different physical processor
Inexpensive creation and context switch	Expensive creation and context switch

Thread Types

Kernel-Level Threads



- A *kernel-level thread* is a thread that the OS knows about
  - Every process has at least one kernel-level thread
- Kernel manages and schedules threads (as well as processes)
  - System calls used to create, destroy, and synchronize threads
- Switching between kernel-level threads of the same process requires a small context switch
  - Values of registers, program counter, and stack counter must be switched
  - Memory management information remains since threads share an address space
- Also known as kernel threads

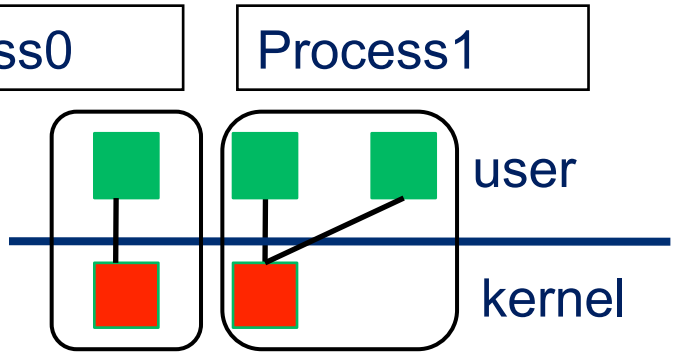
Kernel-Level Threads: Context Switches between threads of the same process

Similar to processes:

- Thread is running
- Thread blocks, is interrupted, *or voluntarily yields*
- Mode switch to kernel mode
- OS saves thread state (to TCB)
- OS chooses new thread to run
- OS loads its state (from TCB)
- Mode switch to user mode
- Thread is running

Except TCB is smaller

User-Level Threads



- A *user-level thread* is a thread the OS does **not** know about
- OS **only** schedules the process **not** the threads within a process
- Programmer uses a *thread library* to manage threads (create, delete, synchronize, and schedule)
  - User-level code can define scheduling policy
  - Threads *yield* to other threads or voluntarily give up the processor
- Switching threads does not involve a context switch

User-Level Threads: Context Switches (sort of)

Similar to processes and kernel-level threads:

- Thread is running
- Thread ~~blocks,~~ is interrupted *by a signal* or voluntarily yields
- ~~Switch to kernel~~
- Library saves thread state (to TCB)
- Library chooses new thread to run
- Library loads its state (from TCB)
- Thread is running

What happens if the thread blocks?

Comparison of Thread Types

	Kernel-Level	User-Level
Model	1-to-1	M-to-1
Managed by (creation, deletion, synchronization)	Operating system (through system calls)	User (through library calls)
Scheduled by	Operating system	User
Requires context switch	Yes	No
Blocks on system calls	Single thread; others may be scheduled	All threads in the process

Kernel-Level Threads

Advantages	Disadvantages
System calls do not block the process	Can be difficult to make efficient
Switching between threads within the same process is inexpensive (registers, PC, and SP are changed, memory management info does not)	
Only one scheduler	

User-Level Threads

Advantages	Disadvantages
Even faster to create and switch (no system calls or context switches necessary); may be an order of magnitude faster	All user-level threads in a process block on system calls (can use non-blocking versions, if they exist)
Customizable scheduler	User-level scheduler can fight with kernel-level scheduler (OS may run a process with only idle threads!)

iClicker Question

A context switch is most expensive for which entity?

A. Process
B. User-level thread
C. Kernel-level thread

So Why Use Kernel Threads?

- I/O: the OS can choose another thread in the same process when a thread does I/O
  - Non-blocking calls are good in theory, but difficult to program in practice
- Kernel-level threads can exploit parallelism
  - Different processors of a symmetric multiprocessor
  - Different cores on a multicore CPU
- Used by systems: Linux, Solaris 10, Windows, pthreads (usually)

So Who Uses User-Level Threads?

- Languages such as Java
  - Phasing out since kernel threads are efficient enough
- May become extinct

One Abstraction, Many Flavors

- Single-threaded processes
  - What we have been assuming
  - Each has exactly one kernel-level thread
  - Add protection
- Multi-threaded processes with user-level threads
  - Threads are created in user-space
  - Have exactly one kernel-level thread
  - Thread management through procedure calls
  - Scheduled by user-space scheduler
  - TCBs in user-space ready list
- Multi-threaded processes with kernel-level threads
  - Threads are created by the OS and thus the OS knows they exist
    - process requests the threads
  - Thread management through system calls
  - TCBs & PCBs on in-kernel ready list
  - Scheduled by OS
- In-kernel threads (New!)
  - Threads that are part of the OS (init, idle)

Note that kernel-level and user-level threads are UNRELATED to kernel-mode and user-mode execution.

One More Flavor: Independent vs. Cooperating Threads

- Independent threads have no shared state with other threads
  - Simple to implement
  - Deterministic
  - Reproducible
  - Scheduling order doesn't matter
- Cooperating threads share state
  - Non-deterministic
  - Non-reproducible
  - Give us concurrency!

Concurrency is great…

```
int a=1, b=2;
main() {
   createThread(fn1, 4);
   createThread(fn2, 5);
}

fn1(int arg1){
   if(a) b++;
}

fn2(int arg1){
   a=arg1;
}
```

What are the values of a and b after execution?

A. a=1, b=2

B. a=1, b=3

C. a=5, b=2

D. a=5, b=3

… but can be problematic

```
int a=1, b=2;
main() {
   createThread(fn1, 4);
   createThread(fn2, 5);
}

fn1(int arg1){
   if(a) b++;
}

fn2(int arg1){
   a=0;
}
```

What are the values of a and b after execution?

A. a=0, b=2

B. a=0, b=3

C. a=1, b=2

D. a=1, b=3

But we DO want cooperating threads…

- Share resources and/or information
  - One ticket database, many users
- Speedup
  - Overlap I/O and computation
  - Multiprocess
- Modularity
  - Factor large programs into simpler pieces

Too Much Milk

Too Much Milk!

You

- Arrive home
- Look in the fridge; out of milk
- Go to store
- Buy milk
- Arrive home; put milk away

Your Roommate

- Arrive home
- Look in fridge; out of milk
- Go to store
- Buy milk
- Arrive home; put milk away
- Oh, no!

Too Much Milk!

- What do we want to happen?
  - Only one person buys milk at a time AND
  - Someone buys milk if you need it

  These are the correctness properties for this problem.
- What happened?
  - Lack of communication!

Race Conditions

- What would the result have been if:
  - your roommate had arrived home for the first time after you had come back from the store?
  - you arrived home after your roommate came back from the store?
  - you were at the store when your roommate came back, but your roommate waited to look in the fridge until after you were back from the store?
- Instances where the result changes based on scheduling are *race conditions*
- What guarantees do we have about how our people/ threads will be scheduled?
- How can we solve this problem?

Too Much Milk: Solution #1

You (Thread A)

```
if(noMilk && noNote)
{
  leave note;
  buy milk;
  remove note;
}
```

Your Roommate (Thread B)

```
if(noMilk && noNote)
{
  leave note;
  buy milk;
  remove note;
}
```

Does this work? A. Yes B. No

Too Much Milk: Solution #2

You (Thread A)

```
leave note A
if(noNote B)
   if(noMilk)
      buy milk;

remove note A
```

Your Roommate (Thread B)

```
leave note B
if(noNote A)
   if(noMilk)
      buy milk;

remove note B
```

Does this work? A. Yes B. No

Too Much Milk: Solution #3

You (Thread A)

```
leave note A
while(note B)
   do nothing;

if(noMilk)
   buy milk;

remove note A
```

Your Roommate (Thread B)

```
leave note B
if(noNote A)
   if(noMilk)
      buy milk;

remove note B
```

Does this work? A. Yes B. No

Why is it correct?

Your Roommate (Thread B)

leave note B

if(noNote A)

  if(noMilk)

    buy milk;

remove note B

At this if, either there is a note A or not.

If not, it is safe for B to check and buy milk , if needed. (Thread A has not started yet.)

If yes, then thread A is checking and buying milk as needed or is waiting for B to quit, so B quits by removing note B.

Why is it correct?

You (Thread A)

leave note A

while(note B)

  do nothing;

if(noMilk)

  buy milk;

remove note A

At this while, either there is a note B or not.

If not, it is safe for A to buy since B has either not started yet or quit.

If yes, A waits until there is no longer a note B, and either finds milk that B bought or buys it if needed.

Why is it correct?

So Thread B buys milk (which Thread A finds) or not, but either way it removes note B. Since Thread A loops, it waits for B to buy milk or not, and then if B did not buy it, it buys the milk.

So it's correct, but… is it good?

1. It is too complicated. It was hard to convince ourselves this solution worked.
2. It is asymmetrical---thread A and thread B are different. *What would we need to do to add new threads?*
3. A is *busy waiting*, or consuming CPU resources despite the fact it is not doing any useful work.

Terminology

- *Atomic Operation*: an operation that is uninterruptible
  - More in a few minutes
- *Synchronization*: Using atomic operations to ensure cooperation between threads
  - More next week
- *Critical Section*: A piece of code that only one thread can execute at a time
  - More now
- *Mutual Exclusion*: Exactly one thread (or process) is doing a particular activity at a time. Usually related to critical sections.
  - More next

More Terminology (How to think about synchronization code)

…

```
entry section       //code to attempt entry into
                    //the critical section

critical section    //code that requires isolation
                    //(e.g., with mutual exclusion)

exit section        //cleanup code after
                    //execution of the critical section

non-critical section     //everything else
```

…

Critical Sections and Correctness

Four properties are required for correctness:

1. *Safety*: only one thread in the critical section
2. *Liveness*: if no threads are executing a critical section, and a thread wishes to enter a critical section, that thread must be guaranteed to eventually enter the critical section
3. *Bounded waiting*: if a thread wishes to enter a critical section, then there exists a bound on the number of other threads that may enter the critical section before that thread does
4. *Failure atomicity*: it's okay for a thread to die in the critical section

Safety and Liveness for Critical Sections

- Only one thread is concurrently in the critical section
  A. Safety
  B. Liveness
  C. Both

- A thread that wants to enter the critical section will eventually succeed
  A. Safety
  B. Liveness
  C. Both

- Bounded waiting: If a thread *i* is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section (only 1 thread is allowed in at a time) before thread *i*'s request is granted.

  A. Safety B. Liveness C. Both

Aside:
Safety and Liveness, More Generally

Properties defined over the execution of a program

- Safety: "nothing bad happens"
  - Holds in every finite execution prefix
    - Windows never crashes
    - No patient is ever given the wrong medication
    - A program never terminates with the wrong answer
- Liveness: "something good eventually happens"
  - No partial execution is irremediable
    - Windows always reboots
    - Medications are eventually distributed to patients
    - A program eventually terminates

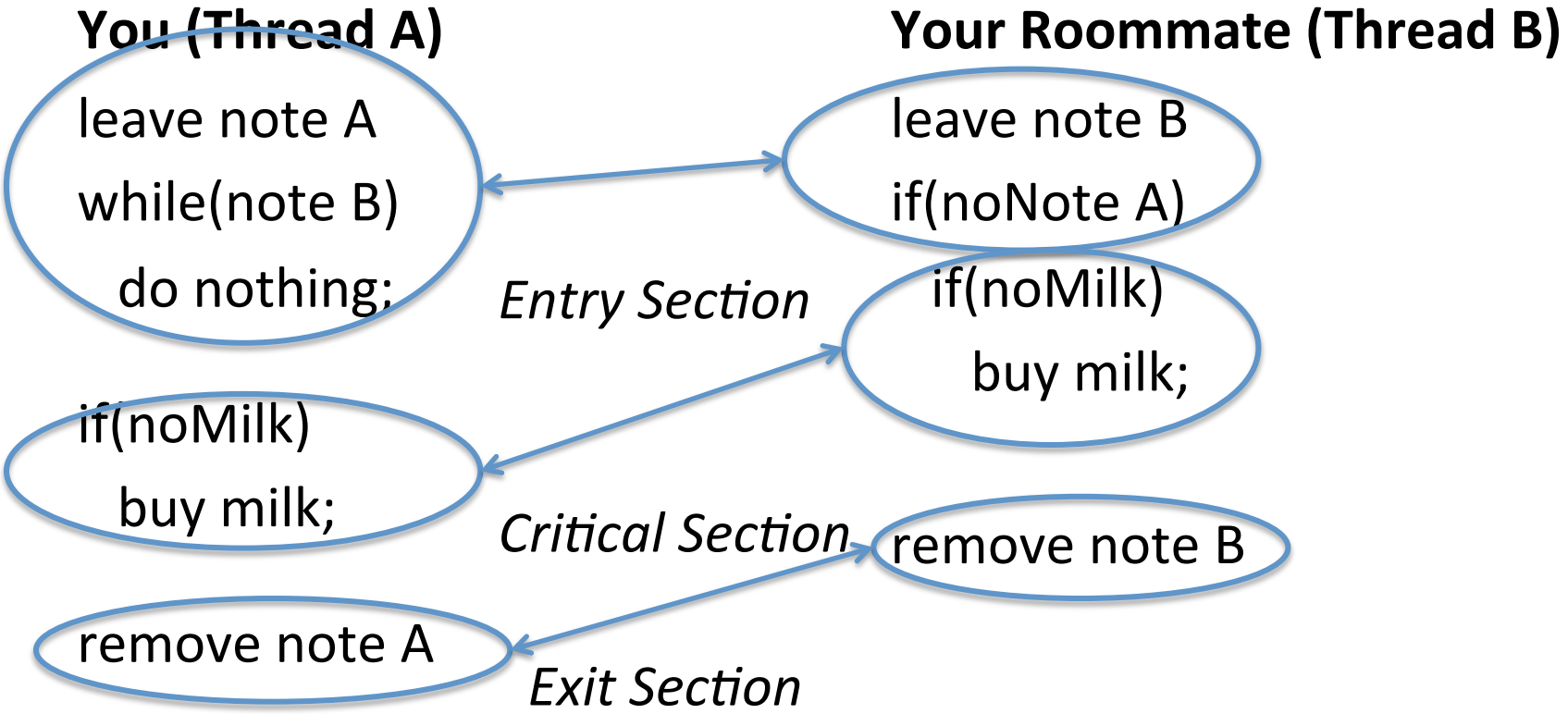
Mutual Exclusion

- Exactly one thread (or process) is doing a particular activity at a time. Usually related to critical sections.
  - Active thread excludes its peers
- Some computer resources cannot be accessed by multiple threads at the same time
  - E.g., a printer can't print two documents at once
- For shared memory architectures, data structures are often mutually exclusive
  - Two threads adding to a linked list can corrupt the list

Formalizing “Too Much Milk”

- Shared variables
  - “Look in the fridge for milk” – check a variable
  - “Put milk away” – update a variable
- Safety property
  - At most one person buys milk
- Liveness
  - Someone buys milk when needed

Formalizing "Too Much Milk"



Atomic Operations

- Operations that are uninterruptible---run to completion or not at all
  - What about x = x + 1?
    - load x
    - add 1
    - store x
  - if(x == 1) x=2?
    - load x
    - compare
    - store x (maybe)
- What operations are uninterruptible?

Language Support for Synchronization

Some programming languages provide support for *atomic routines* for synchronization

- *Locks*: One process holds a lock at a time, executes the critical section, releases the lock
- *Semaphores*: More general version of locks
- *Monitors*: Connects shared data to synchronization primitive

=> All require some hardware support (and waiting!).

Locks, Generally

A *lock* prevents another process from doing something

- Lock before entering a critical section or before accessing shared data
- Unlock when leaving a critical section or when access to shared data is complete
- Wait if locked

Locks, More Formally

- *Locks* provide mutual exclusion to shared data with two atomic routines:
  - *Lock::Acquire*: wait until lock is free, then grab it
  - *Lock::Release*: unlock and wake up any thread waiting in Acquire
- Rules for using a lock:
  - Always acquire the lock before accessing shared data
  - Always release the lock after finishing with shared data
  - Lock is initially free
- Next week we'll look at potential implementations of acquire() and release()

Locks and Too Much Milk

Our solution used notes as locks:

1. Leave a note (acquire a lock)
2. Remove a note (release the lock)
3. Do not buy any milk if there is a note (wait)

What would it look like with actual locks?

Too Much Milk: Lock Solution

You (Thread A)

```
Lock->Acquire();
if(noMilk)
   buy milk;
Lock->Release();
```

Your Roommate (Thread B)

```
Lock->Acquire();
if(noMilk)
   buy milk;
Lock->Release();
```

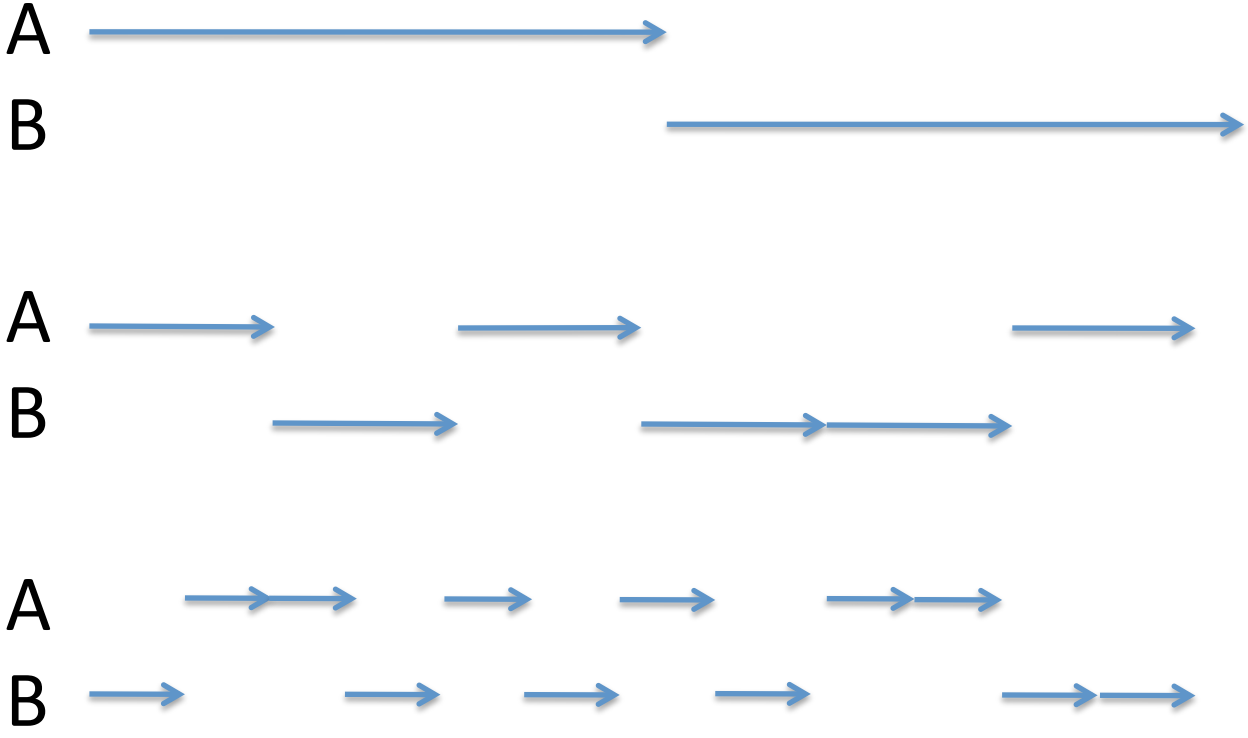
Summary

- Threads share the same address space
  - Processes have *separate* address spaces
  - Child processes start with a *copy* of their parent's address space
- Each thread has its own thread of control
  - Program counter, register values, execution stack
- It is easy for threads to inadvertently disrupt each other since they share the entire address space (!)
- Communication among threads is typically done through shared variables
- Operating system can switch from any thread at any time (assuming kernel threads)
- Critical sections identify pieces of code that cannot be executed in parallel by multiple threads
  - Typically code that accesses or modifies shared variables

Review: Threads and the Scheduler

(or, Why Multi-threaded Programming is Hard)

Given two threads, A and B, how might their executions be scheduled?



Announcements

- Homework 2 due Friday at 8:45a
  - Remember, two parts!
- Project 0 is up, due Friday, 2/13
  - Follow style guidelines
  - Keep pair programming log
  - Fill in README
- Project 1 will be up on Monday