Concurrency: Synchronization

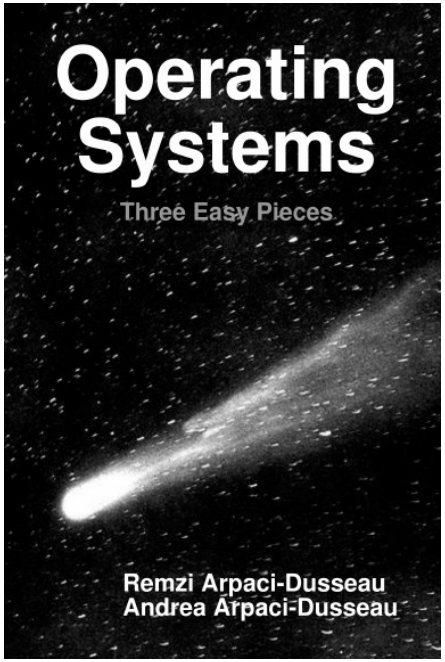
Methods & Tools for Software Engineering (MTSE)
Fall 2020

Arie Gurfinkel

UNIVERSITY OF WATERLOO

References

Operating Systems: Three Easy Pieces by R.H. Arpaci-Dusseau, A.C. Arpaci-Dusseau



- http://pages.cs.wisc.edu/~remzi/OSTEP/
- Chapter 26: Concurrency and Thread API
- Chapter 27: Thread API
- Chapter 28: Locks
- Chapter 31: Semaphores
- Chapter 32: Concurrency Bugs

- Code examples:
  - https://github.com/remzi-arpacidusseau/ostep-code

Slides courtesy of Prof. Seyed M. Zahedi

Multi-programming vs Multi-threading

Multi-programming

- Doing multiple things at once using multiple processes
- Inter-Process Communication (IPC)
  - e.g., fork and pipe
- Pro: memory safety protection from OS
- Cons: IPC is hard

Multi-threading

- Doing multiple things at once using multiple threads
- Shared memory communication
  - all threads share the same memory space
- Pro: light-weight, easy to communicate
- Cons: hard to get right, errors are hard to debug
  - race conditions, atomicity violations, deadlocks, starvation, …

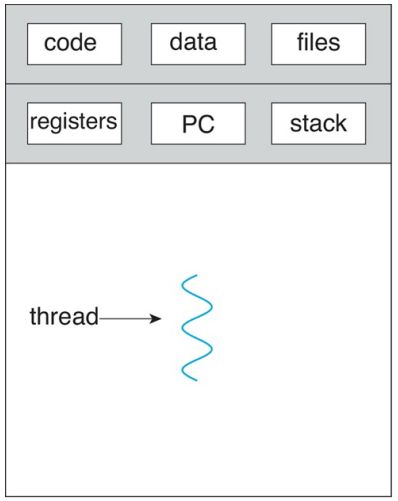
Traditional UNIX Process

Process is OS abstraction of what is needed to run single program

- Often called "heavyweight process"

Processes have two parts

- Sequential program execution stream (active part)
  - Code executed as sequential stream of execution (i.e., thread)
  - Includes state of CPU registers
- Protected resources (passive part)
  - Main memory state (contents of Address Space)
  - I/O state (i.e. file descriptors)



single-threaded process

Modern Process with Threads

Thread: sequential execution stream within process
(sometimes called "lightweight process")

- Process still contains single address space
- No protection between threads

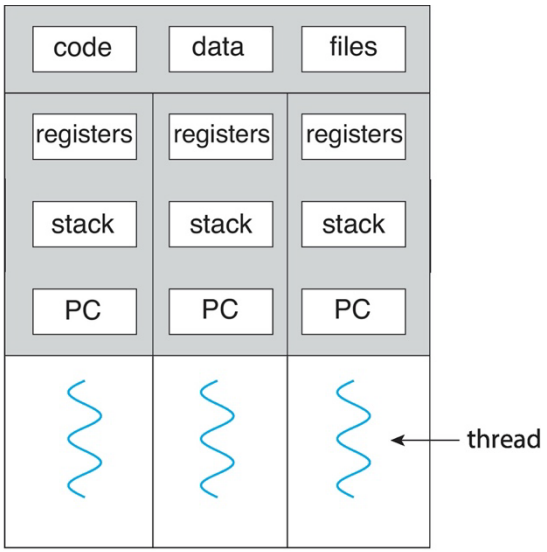
Multithreading: single program made up of different concurrent activities (sometimes called multitasking)

Some states are shared by all threads

- Content of memory (global variables, heap)
- I/O state (file descriptors, network connections, etc.)

Some states "private" to each thread

- CPU registers (including PC) and stack



multithreaded process

Threads Motivation

OS's need to handle multiple things at once (MTAO)

- Processes, interrupts, background system maintenance

Servers need to handle MTAO

- Multiple connections handled simultaneously

Parallel programs need to handle MTAO

- To achieve better performance

Programs with user interfaces often need to handle MTAO

- To achieve user responsiveness while doing computation

Network and disk programs need to handle MTAO

- To hide network/disk latency

Multithreading: Process versus Thread

Process provides an execution context for the program

- **unit of ownership**
- Memory, I/O resources, console, etc.
- Process pretends like it is a single entity controlling the execution environment
- Inter Process Communication (IPC) is “like” communicating between individual machines (but connected with super-fast network)

Thread represent a single execution unit (i.e., CPU)

- unit of scheduling
- ancient time: a process has one thread running on one physical CPU
- old time: a process has many threads sharing one physical CPU
- today: a process has many threads sharing many physical CPUs (multicore)
- all threads of a process share the same memory space!

Threads: Programmer's Perspective

A thread is a function that is ran concurrently with other functions

- It is like `fork()` followed by a call to a child process function
- **Except**: no new process is created. The new thread can access **all** the data of the **current** process

```
void * foo(void*) {...}
void * bar(void*) {...}

int main(void) {
  pthread_t t1, t2;
  void *data;
  ...
  pthread_create(&t1, NULL, foo, data);
  pthread_create(&t2, NULL, bar, data);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```

Threads: Programmer's Perspective

A thread is a function that is ran concurrently with other functions

- It is like fork() followed by a call to a child process function
- Except: no new process is created. The new thread can access all the data of the **current** process

```
void * foo(void*) {...}
void * bar(void*) {...}

int main(void) {
  pthread_t t1, t2;
  void *data;
  ...
  pthread_create(&t1, NULL, foo, data);
  pthread_create(&t2, NULL, bar, data);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```

Code that will execute concurrently

Start of concurrent execution

Main thread waits for others to finish

Race Conditions

CONCURRENCY ISSUES

Problem Is At The Lowest Level

- When threads work on separate data, order of scheduling does not change results

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

- Scheduling order matters when threads work on shared data

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

- What are possible values of x? (initially, y = 12)

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

x = 13

Problem Is At The Lowest Level

- When threads work on separate data, order of scheduling does not change results

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

- Scheduling order matters when threads work on shared data

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

- What are possible values of x? (initially, y = 12)

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

x = 5

Problem Is At The Lowest Level

- When threads work on separate data, order of scheduling does not change results

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

- Scheduling order matters when threads work on shared data

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

- What are possible values of x? (initially, y = 12)

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

x = 3

LOCKS

Too Much Milk Example

	Roommate A	Roommate B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
01:00		Arrive home, put milk away. Oh no!

Atomic Operations

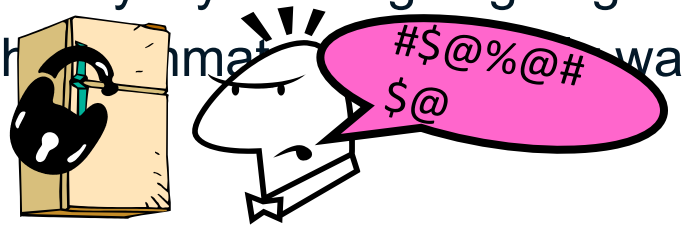
- Operation that always runs to completion or not at all
  - Indivisible: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block: if no atomic operations, then have no way for threads to work together
- On most machines, memory references 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 whole array

Definitions (1/2)

- Race condition: output of concurrent program depends on order of operations between threads
- Synchronization: using atomic operations to ensure cooperation between multiple concurrent threads
  - For now, only loads and stores are atomic
  - We will see that its hard to build anything useful with only load/store
- Mutual exclusion: ensuring that only one thread does a particular operation at a time
  - One thread excludes others while doing its task
- Critical section: piece of code that only one thread can execute at once
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing same thing

Definitions (2/2)

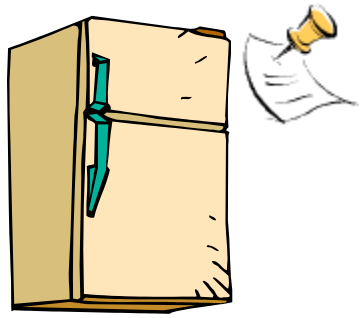
- Lock: prevent someone from doing something
  - Lock before entering critical section, before accessing shared data
  - Unlock when leaving, after done accessing shared data
  - Wait if locked
    - Important idea: synchronization involves waiting!
- Example: fix milk problem by putting a key on refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate wants OJ



Too Much Milk: Correctness Properties

- Be careful about correctness of your concurrent programs
  - Behavior could be non-deterministic
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
  - Always write down behavior first
- What are correctness properties of "too much milk" problem?
  - Never more than one person buys
  - Someone buys if needed
- In this lecture, we restrict ourselves to only atomic load/store

Too Much Milk (Solution #1)



- Use a note
  - Leave note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Would this work if computer program tries it?
  (remember, only memory load/store are atomic)

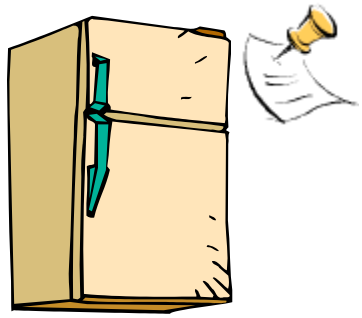
```
if (!milk) {
  if (!note) {
    leave note;
    buy milk;
    remove note;
  }
}
```

Solution #1 (cont.)

```
if (!milk) {
   if (!note) {
      leave note;
      buy milk;
      remove note;
   }
}
```

```
if (!milk) {
   if (!note) {
      leave note;
      buy milk;
      remove note;
   }
}
```

Try #1 (cont.)



- Conclusion
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution #1 makes problem worse since it fails intermittently
  - Makes it very hard to debug …
  - Programs must work despite what thread scheduler does!

Too Much Milk (Solution #1½)

- Clearly note is not blocking enough
- Let's try to fix this by placing note first

```
leave note;
if (!milk) {
   if (!note) {
        buy milk;
    }
}
remove note;
```

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

Too Much Milk (Solution #2)

- How about labeled notes?

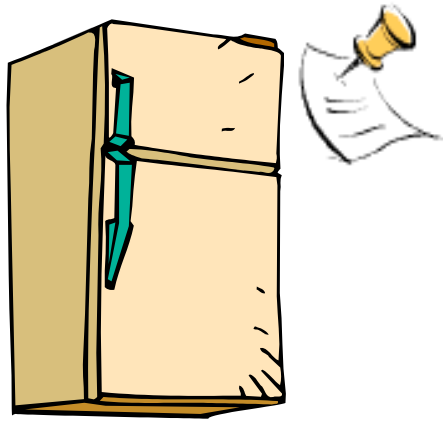
```
// Thread A
leave note A;
if (!note B) {
  if (!milk)
    buy milk;
}
remove note A;
```

```
// Thread B
leave note B;
if (!note A) {
  if (!milk)
    buy milk;
}
remove note B;
```

- Does this work?
  - It is still possible that neither of threads buys milk
  - This is extremely unlikely, but it's still possible

Problem with Solution #2

- I thought *you* had the milk! But I thought *you* had the milk!
- This kind of lockup is called "**starvation**!"



Too Much Milk (Solution #3)

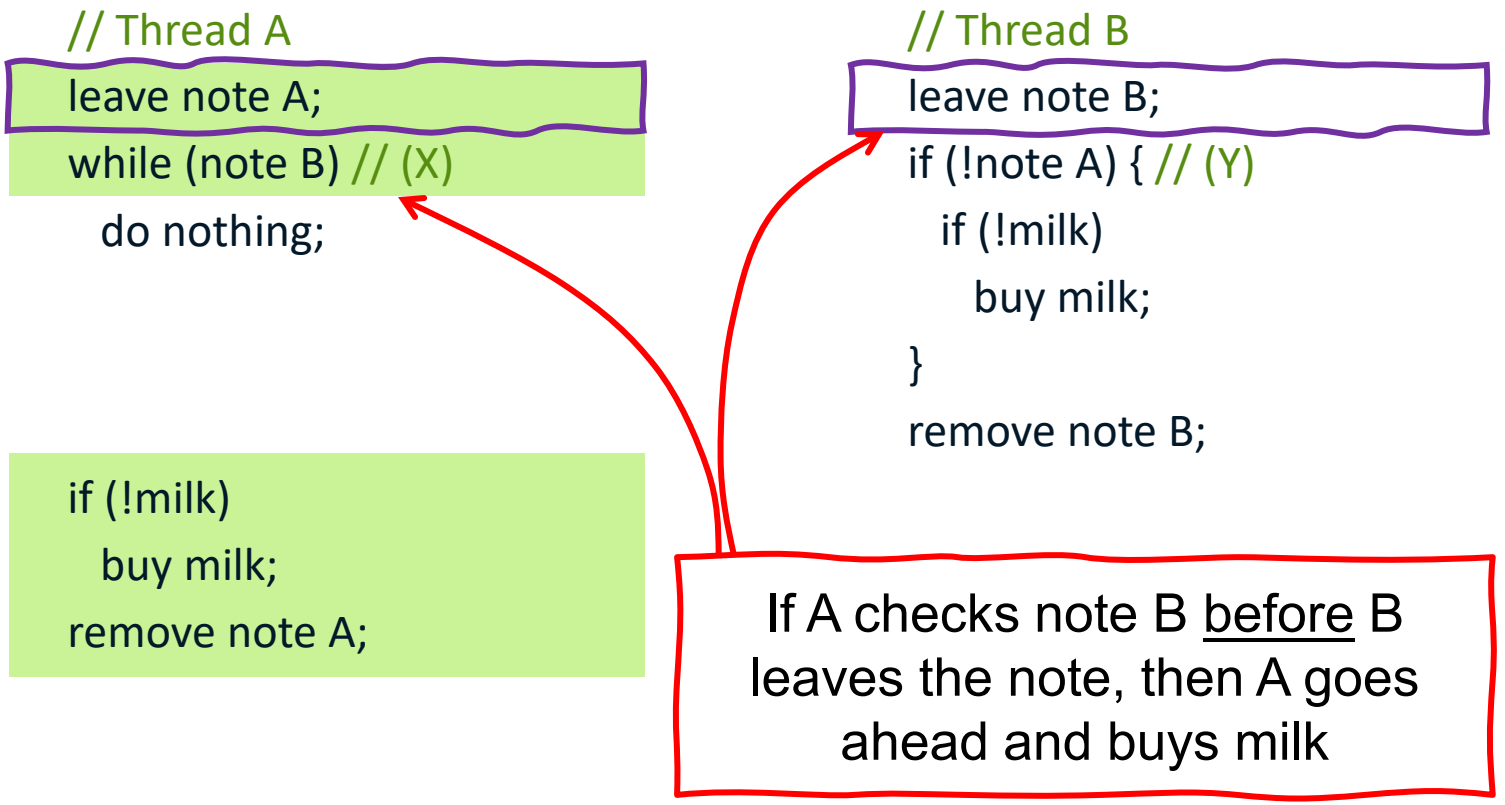
```
// Thread A
leave note A;
while (note B) // (X)
   do nothing;
if (!milk)
   buy milk;
remove note A;
```

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

- Does this work?
  - Yes! It can be guaranteed that it is safe to buy, or others will buy:
    it is ok to quit
- At (X)
  - If no note from B, safe for A to buy
  - Otherwise, wait to find out what will happen
- At (Y)
  - If no note from A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

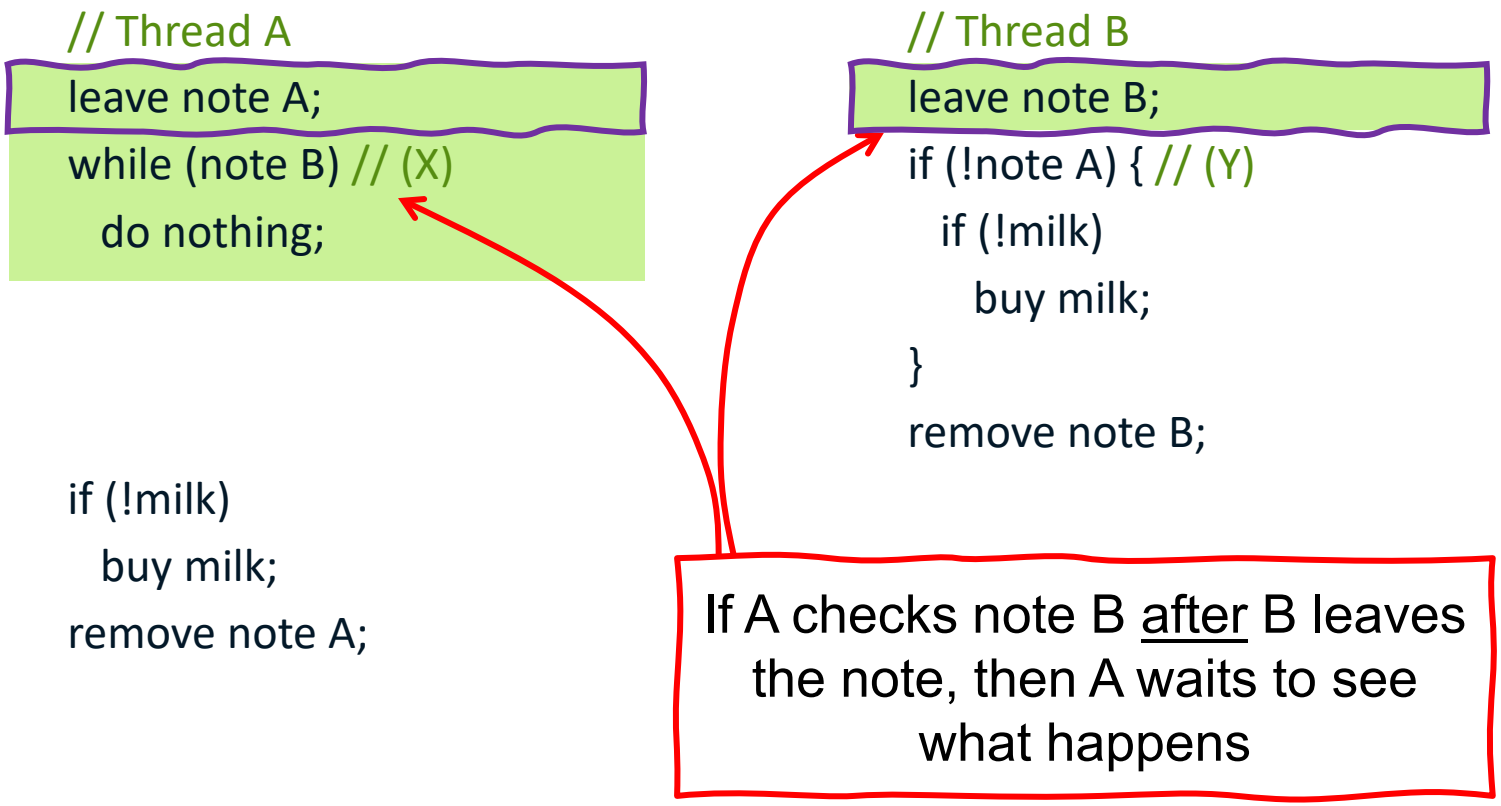
Case I.a

- A leaves note A before B checks



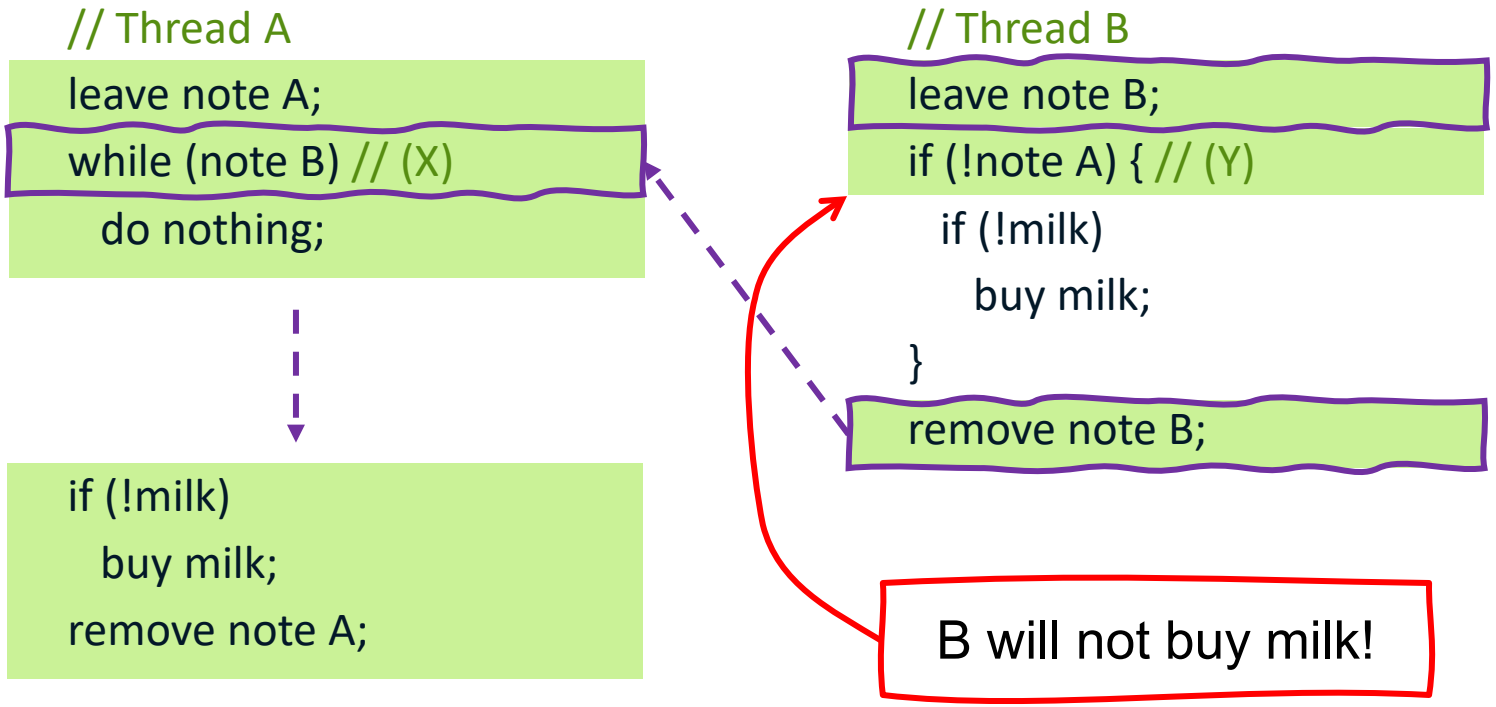
Case I.b

- A leaves note before B checks



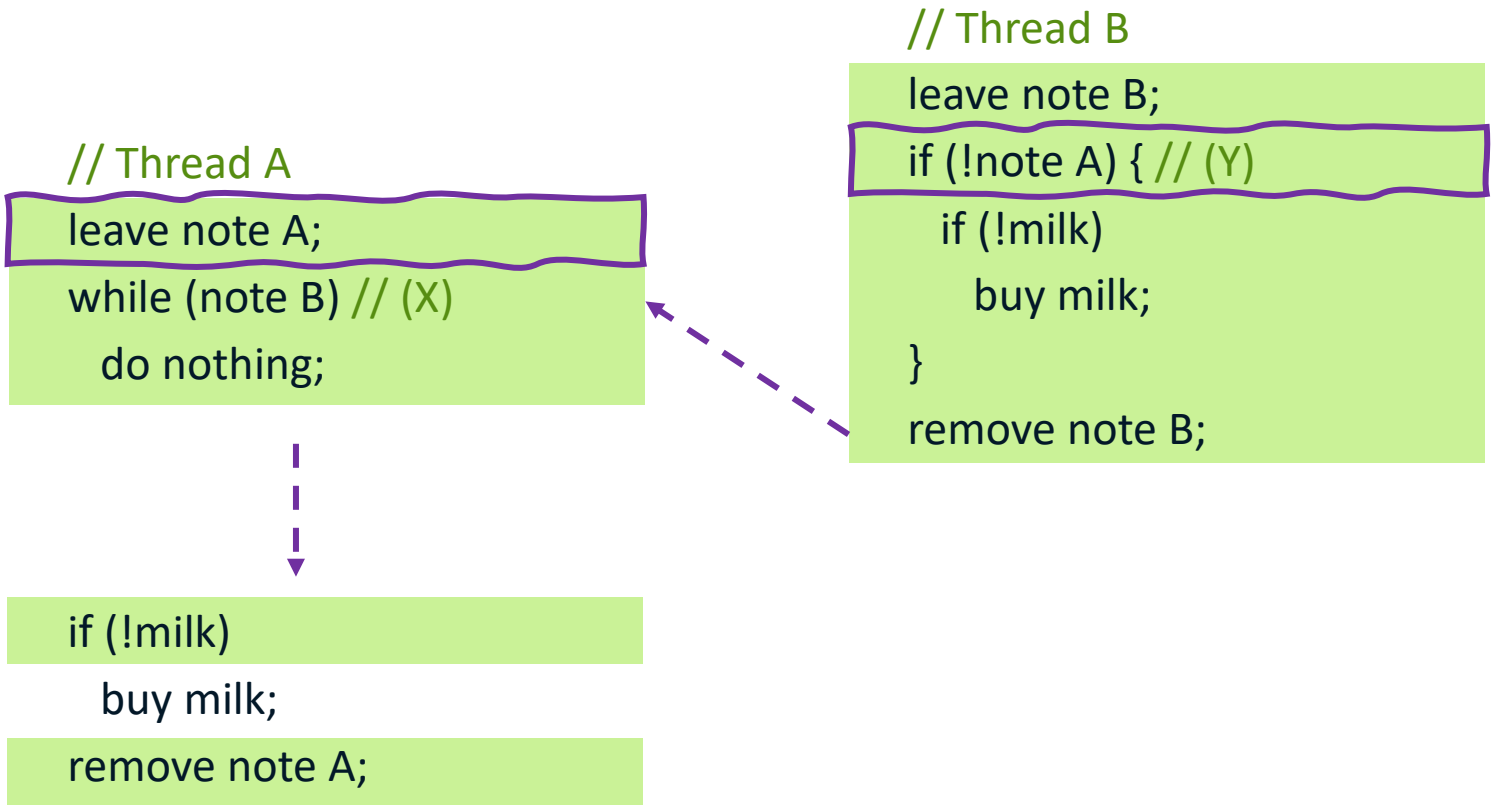
Case I.b (cont.)

- A leaves note before B checks



Case 2

- B checks note A before A leaves it



Solution #3: Discussion

- Our solution protects single critical section for each thread

```
if (!milk) {
   buy milk;
}
```

- Solution #3 works, but it's very unsatisfactory
  - Way too complex – even for this simple example
    - It's hard to convince yourself that this really works
    - Reasoning is even harder when modern compilers/hardware reorder instructions
  - A's code is different from B's – what if there are lots of threads?
    - Code would have to be slightly different for each thread (see **Peterson's** algorithm)
  - A is busy-waiting – while A is waiting, it is consuming CPU time
- There's a better way
  - Have hardware provide higher-level primitives other than atomic load/store
  - Build even higher-level programming abstractions on this hardware

Too Much Milk (Solution #4)

- Suppose we have some sort of implementation of a lock
  - 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 "too much milk" problem is easy to solve

```
milklock.Acquire();
if (nomilk)
  buy milk;
milklock.Release();
```

- Code between Acquire() and Release() is called **critical section**

Where Are We Going with Synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Interrupts Test&Set Compare&Swap

- We will see how we can implement various higher-level synchronization primitives using atomic operations
  - Everything is quite painful if load/store are the only atomic primitives
  - Hardware needs to provide more primitives useful at user-

How to Implement Locks?

- Locks are used to prevent 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: synchronization involves waiting
    - Busy-waiting is wasteful (should sleep if waiting for a long time)
- With only atomic load/store we get solutions like “Solution #3”
  - Too complex and error prone
- Is hardware lock instruction good idea?
  - What about putting threads to sleep?
    - How does hardware interact with OS scheduler?
  - What about complexity?
    - Adding each extra feature makes HW more complex and slower

Naïve Implementation of Locks

- Goal: building multi-instruction atomic operations
- Dispatcher gets control in two ways
  - Internal: thread does something to relinquish CPU
  - External: interrupts cause dispatcher to take CPU
- On uniprocessors, we can avoid context-switching by
  - Avoiding internal events
  - Preventing external events by disabling interrupts
- Consequently, naïve implementation of locks in uniprocessors

```
Acquire { disable interrupts; }
Release { enable interrupts; }
```

Problems with Naïve Implementation of Locks

- OS cannot let users use this!

```
Acquire();
while(TRUE) {;}
```

- In real-time systems, there is no guarantees on timing!
  - Critical sections might be arbitrarily long
  - What happens with I/O or other important eve
    - "Reactor about to meltdown. Help?"

Problem with Implementing Locks Using Interrupts

- Cannot 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 solution: atomic read-modify-write instructions
  - Read value from an address and then write new value to it *atomically*
  - Make HW responsible for implementing this correctly
    - Uniprocessors (not too hard)
    - Multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, this can be used in both uniprocessors and multiprocessors

Examples of Read-Modify-Write Instructions

- test&set (&address) {
  result = M[address];
  M[address] = 1;
  return result;
  }

  /* most architectures */
  /* return result from "address" and set value at "address" to 1 */

- swap (&address, register) {
  temp = M[address];
  M[address] = register;
  register = temp;
  }

  /* x86 */
  /* swap register's value to value at "address" */

- compare&swap (&address, reg1, reg2) {
  if (reg1 == M[address]) {
  M[address] = reg2;
  return success;
  } else {
  return failure;
  }
  }

  /* 68000 */

Implementing Locks Using test&set

- Simple implementation

```
int value = 0;                          // Free
Acquire() {
    while (test&set(&value));                   // while
busy
}

Release() {
    value = 0;
}
```

- Free lock: test&set reads 0 and sets value = 1
- Busy lock: test&set reads 1 and sets value = 1 (no change)
- What is wrong with this implementation?
  - Waiting threads consume cycles while busy-waiting

Locks with Busy-Waiting: Discussion

Upside?

- Machine can receive interrupts
- User code can use this lock
- Works on multiprocessors

Downside?

- This is very wasteful as threads consume cycles waiting
- Waiting threads 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!

In semaphores and monitors, threads may wait for arbitrary long time!

- Even if busy-waiting was OK for locks, it's not ok for other primitives
- Exam solutions should avoid busy-waiting!

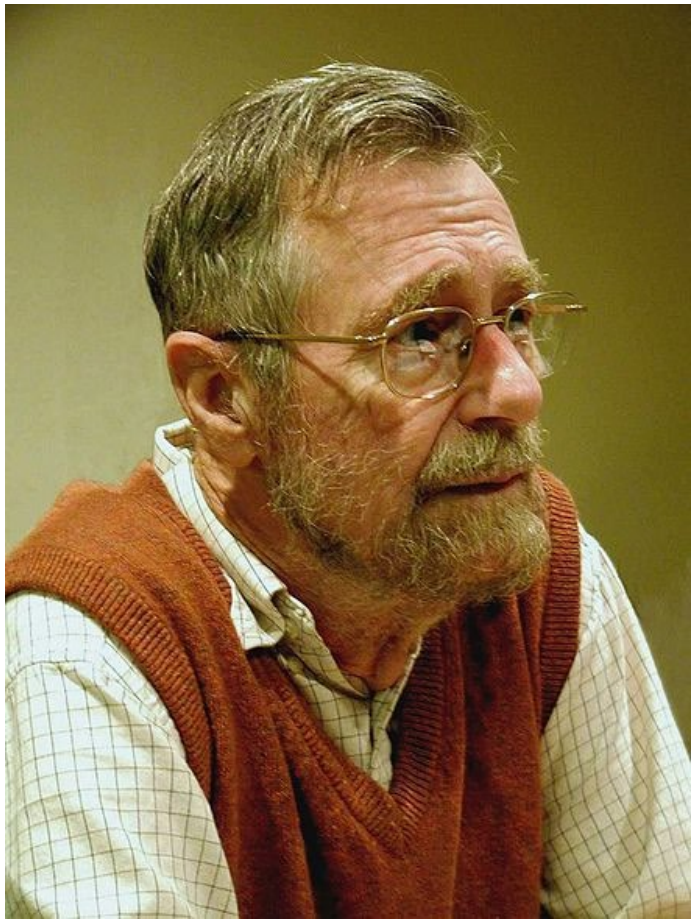
Rules for Using Locks

- Lock should be initially free
- Always acquire before accessing shared data
  - Best place for acquiring lock: beginning of procedure!
- Always release lock after finishing with shared data
  - Best place for releasing lock: end of procedure!
  - Only the lock holder can release
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger! Don't do it even if it's tempting!

Another synchronization primitive

SEMAPHORES

Edsger W. Dijkstra



(image courtesy of wikipedia.org)

Producer-Consumer with Bounded Buffer

- Problem definition
  - Producer puts things into shared buff
  - Consumer takes them out
  - 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: newspaper vending machine
  - Producer can put limited number of newspapers in mach
  - Consumer can't take newspaper out if machine is empty

Bounded Buffer Correctness Constraints

- Consumer must wait for producer if buffer is empty (scheduling constraint)
- Producer must wait for consumer if buffer is full (scheduling constraint)
- Only one thread can manipulate buffer at any time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine delivery person filling the vending machine, and somebody comes and tries to stick their money into the machine

Bounded Buffer Implementation with Locks

```
try_to_produce(item) {
  lock.acquire();
  success = FALSE;
  if (queue.size() < MAX) {
    queue.enqueue(item);
    success = TRUE;
  }
  lock.release();
  return success;
}
```

```
try_to_consume() {
  lock.acquire();
  item = NULL;
  if (!queue.isEmpty())
    item = queue.dequeue();
  lock.release();
  return item;
}
```

- If try_to_consume() returns NULL, do we know buffer is empty?
  - No! Producer might have filled buffer
  - We only know buffer was empty when we tried
  - After releasing lock our knowledge of buffer might not be accurate – we only know state of buffer while holding the lock!
- Is it a good idea to do while(try_to_consume() != NULL)?
  - This will delay producer's thread from putting items on buffer – bad for everyone!

Where Are We Going with Synchronization?

Programs	Shared Programs
Higher-level API	Locks **Semaphores** Monitors Send/Receive
Hardware	Load/Store Disable Interrupts Test&Set Compare&Swap

- We will see how we can implement various higher-level synchronization primitives using atomic operations
  - Everything is quite painful if load/store are the only atomic primitives
  - Hardware needs to provide more primitives useful at user-

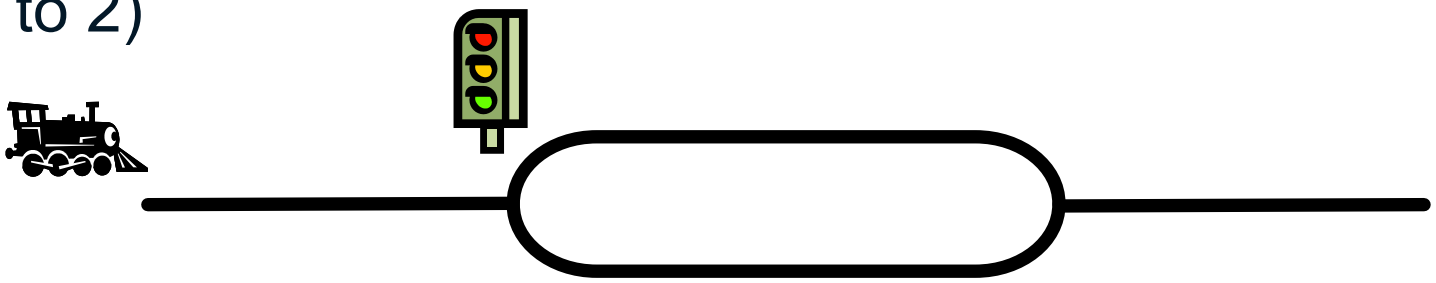
Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX

- Semaphore has non-negative integer value and 2 operations
  - P(): atomic operation that waits for semaphore to become positive, then decrements it by one
  - V(): atomic operation that increments semaphore by one, waking up a waiting P(), if any
  - In Dutch, P stands for *proberen* (to test) and V stands for *verhogen* (to increment)

Semaphores (cont.)

- Semaphores are like integers, except
  - No negative values
  - Only available operations are `P` & `V`
    (cannot read/write value, except to set it initially)
  - Operations must be atomic
    - Two `P`'s together can't decrement value below zero
    - Thread going to sleep in `P` won't miss wakeup from `V`
      (even if they both happen at same time)
- Example: semaphore from railway analogy (initialized to 2)



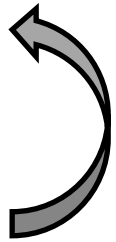
Example Uses of Semaphores

- Mutual exclusion with binary semaphore (initialized to 1)

```
semaphore.P();
// Critical section goes here
semaphore.V();
```

- Scheduling constraints (initialized to 0)

```
thread_join(…) {
  semaphore.P();
}

thread_exit() {
  semaphore.V();
}
```



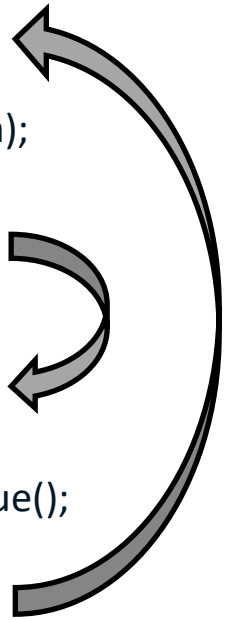
Recall: Bounded Buffer Correctness Constraints

- Consumer must wait for producer if buffer is empty (scheduling constraint)
- Producer must wait for consumer if buffer is full (scheduling constraint)
- Only one thread can manipulate buffer at any time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine delivery person filling the vending machine, and somebody comes and tries to stick their money into the machine

Bounded Buffer Implementation with Semaphore

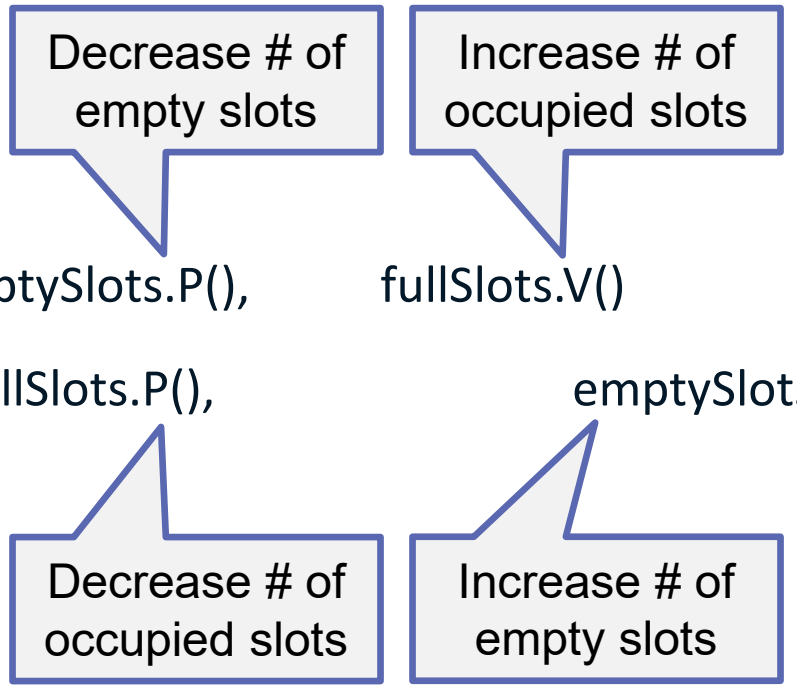
General rule of thumb is to use separate semaphore for each constraint

```
Semaphore emptySlots = MAX;          // producer's constraint
Semaphore fullSlots = 0;             // consumer's constraint
Semaphore mutex = 1;                 // mutual exclusion
produce(item) {
  emptySlots.P();                    // wait until there is space
  mutex.P();                         // wait until machine is free
  queue.enqueue(item);
  mutex.V();
  fullSlots.V();                     // tell consumer there is
}                                    // an additional item
consume() {
  fullSlots.P();                     // wait until there is an item
  mutex.P();                         // wait until machine is free
  item = queue.dequeue();
  mutex.V();
  emptySlots.V();                    // tell producer there is an
  return item;                       // additional empty space
}
```



Discussion about Solution

- Why asymmetry?
- Producer does: emptySlots.P(), fullSlots.V()
- Consumer does: fullSlots.P(), emptySlots.V()



Discussion about Solution (cont.)

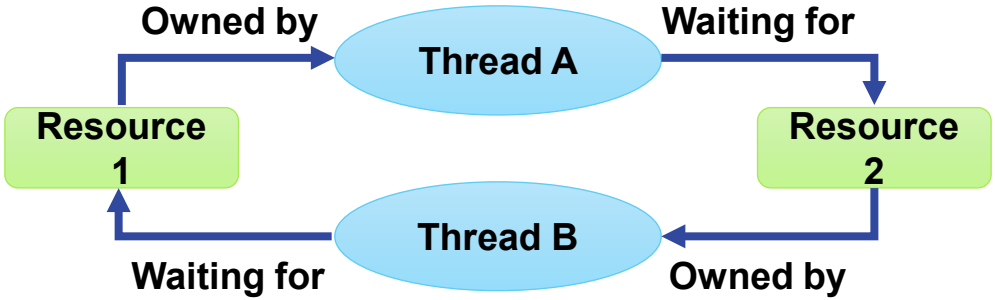
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, it only might affect scheduling efficiency

```
produce(item) {
        mutex.P();
        emptySlots.P();
        queue.enqueue(item);
        mutex.V();
        fullSlots.V();
}
consume() {
        fullSlots.P();
        mutex.P();
        item = queue.dequeue();
        mutex.V();
        emptySlots.V();
        return item;
}
```

DEADLOCK

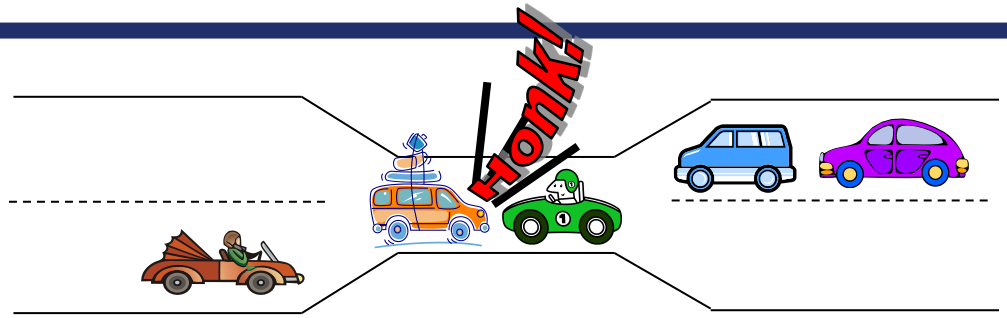
Starvation vs. Deadlock

- Starvation: thread waits indefinitely
  - e.g., low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
  - Thread B owns Res 2 and is waiting for Res 1



- Deadlock leads to starvation but not the other way around
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

Bridge Crossing Example



- Each segment of road can be viewed as resource
  - Cars must own segment under them and acquire segment they are moving to
- To cross bridge cars must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast $\Rightarrow$ no one goes west

Conditions for Deadlock

- Deadlock is not always deterministic

Thread A	Thread B
x.P();	y.P();
y.P();	x.P();
y.V();	x.V();
x.V();	y.V();

  - This code doesn't always lead to deadlock
    - Must have exactly right timing ("wrong" timing?)
    - So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant…

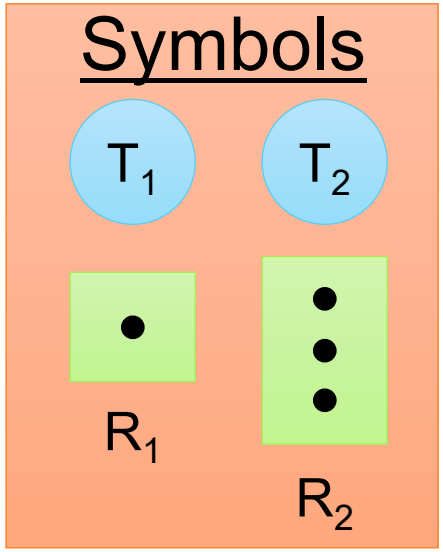
- Deadlocks occur with multiple resources
  - Can't solve deadlock for each resource independently

Four Requirements for Deadlock

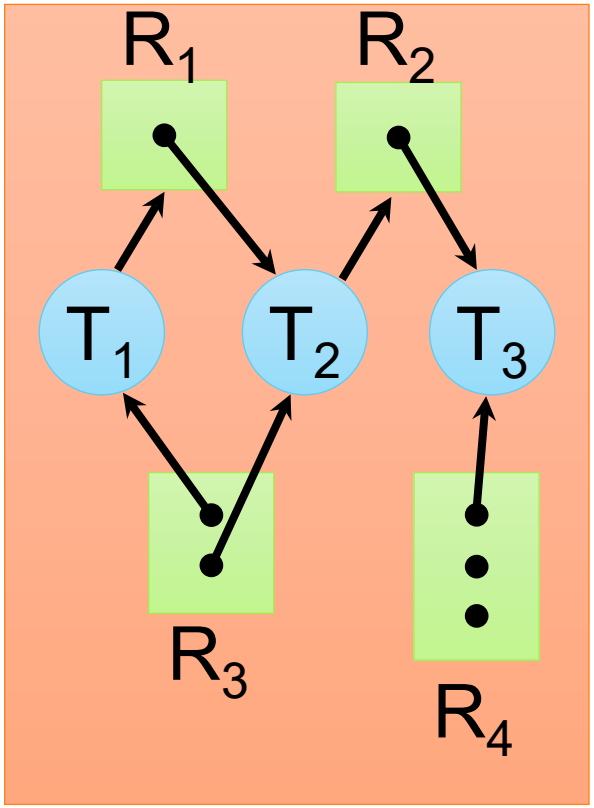
- Mutual exclusion
  - Only limited number of threads at a time can use resource
- Hold and wait
  - Thread hold resources while waiting to acquire additional ones
- No preemption
  - Resources are released only voluntarily by thread holding them
- Circular wait
  - There exists a set T_1, …, T_n of waiting threads
    - T_1 is waiting for resource that is held by T_2
    - T_2 is waiting for resource that is held by T_3
    - …
    - T_n is waiting for resource that is held by T_1

Resource Allocation Graph

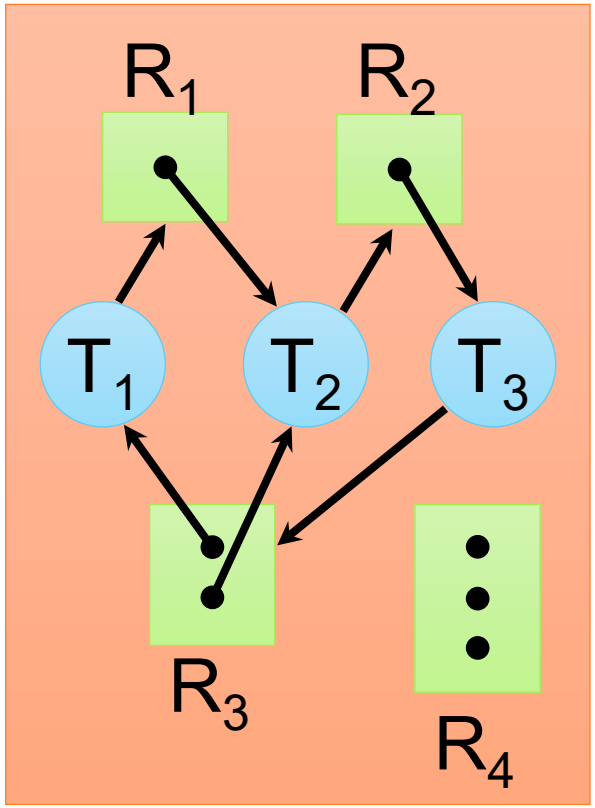
- System model
  - Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$
    - CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_{ij} instances
  - Each thread utilizes resources as follows
    - Request() / Use() / Release()
- Resource allocation graph
  - V is partitioned into two types
    - $T = \{T_1, \ldots, T_n\}$, set threads in system
    - $R = \{R_1, \ldots, R_m\}$, set of resource types in system
  - Request edge is directed edge $T_i \rightarrow R_j$
  - Assignment edge is directed edge $R_q \rightarrow T_p$



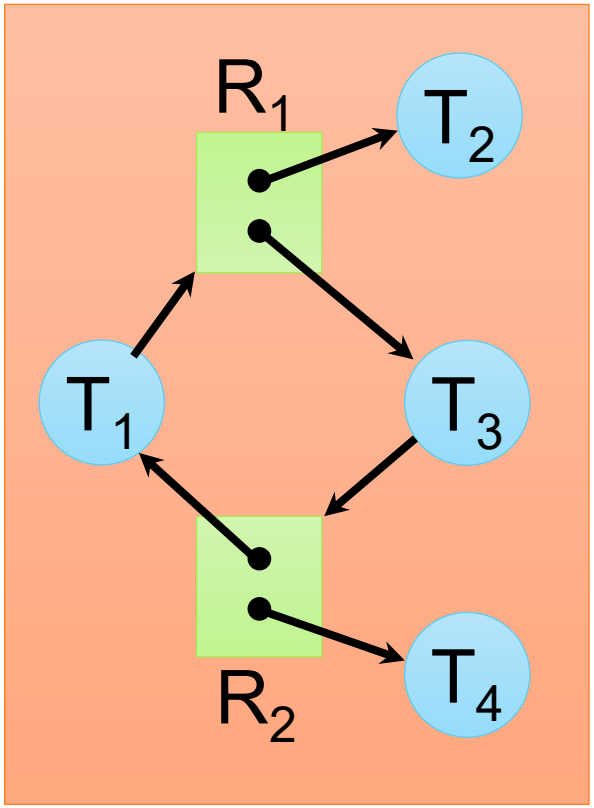
Resource Allocation Graph Examples



Simple resource allocation graph

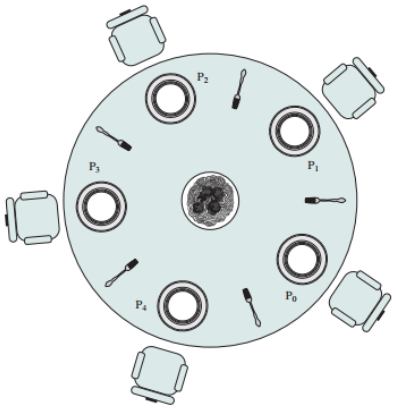


Allocation graph with deadlock



Allocation graph with cycle, but no deadlock

Dining ~~Philosophers~~ Politicians!



- Each politician needs two chopsticks to eat
- Each grabs chopstick on the right first (all right-handed)
- Deadlock if all grab chopstick at same time
- Deadlock depends on the order of execution
  - No deadlock if one was left-handed

Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Technique for forcibly preempting resources and/or terminating tasks
- Ensure that system will never enter deadlock
  - Need to monitor all resources acquisitions
  - Selectively deny those that might lead to deadlock
- Ignore problem and pretend deadlocks never occur
  - Used by most operating systems, including UNIX

Techniques for Preventing Deadlock

- Infinite resources
  - Include enough resources so that no one ever runs out of resources
    - Doesn't have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - Bay bridge with 12,000 lanes. Never wait!
    - Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Often true (most things don't depend on each other) but not very realistic in general
- Don't allow waiting
  - How phone company avoids deadlock
    - Call someone, either goes through or goes to voicemail
  - Technique used in Ethernet/some multiprocessor nets
    - Everyone speaks at once. On collision, back off and retry
  - Inefficient, since must keep retrying
    - Consider: driving to Toronto, when hit traffic jam, suddenly transported back and

Techniques for Preventing Deadlock (cont.)

- Make all threads request everything they'll need at the beginning
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - If need 2 chopsticks, request both at same time
    - Don't leave home until we know no one is using any intersection between here and where you want to go; only one car on the bridge at a time

- Force all threads to request resources in fixed order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (x.P, y.P, z.P,…)
    - Make tasks request disk, then memory, then…
    - Keep from deadlock on freeways by requiring everyone to go clockwise

CONDITIONAL VARIABLES

Semaphores Considered Harmful

"During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores."

Dijkstra "The structure of the 'THE'-Multiprogramming System"
Communications of the ACM v. 11 n. 5 May 1968.)"

Motivation for Monitors and Condition Variables

- Problem: 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

- Solution: use locks for mutual exclusion and Condition Variables (CV) for scheduling constraints

- Definition: monitor is one lock with 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 provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- CV is queue of threads waiting for event inside 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

Condition Variables Operations

- wait(Lock *lock)
  - Atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- signal()
  - Wake up a waiter, if any
- broadcast()
  - Wake up all waiters, if any

Properties of Condition Variables

- Condition variables are memoryless
  - No internal memory except a queue of waiting threads
  - No effect in calling signal/broadcast on empty queue
- ALWAYS hold lock when calling wait(), signal(), broadcast()
  - In Birrell paper, he says you can call signal() outside of lock – IGNORE HIM (this is only an optimization)
- Calling wait() atomically adds thread to wait queue and releases lock
- Re-enabled waiting threads may not run immediately
  - No atomicity between signal/broadcast and the return from wait

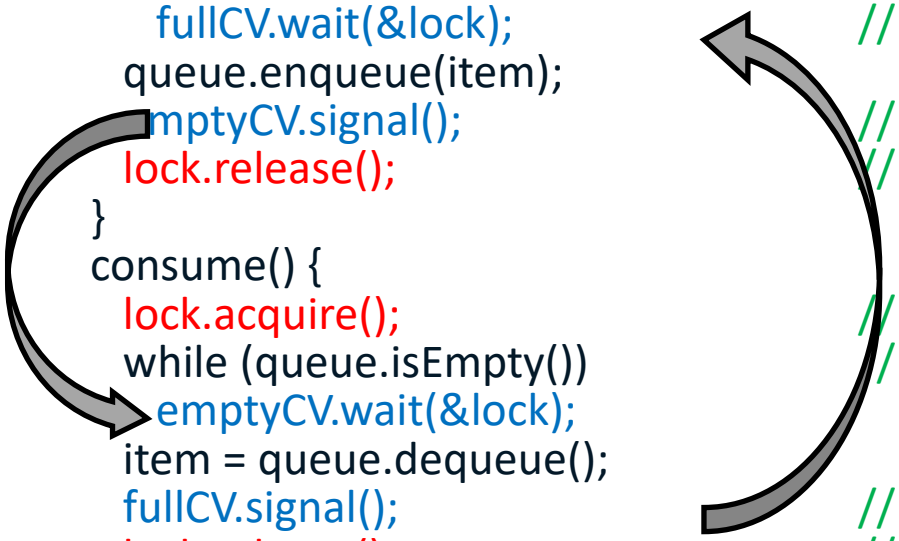
Condition Variable Design Pattern

```
method_that_waits() {
  lock.acquire();

  // Read/write shared state

  while (!testSharedState())
    cv.wait(&lock);

  // Read/write shared state

  lock.release();
}
```

```
method_that_signals() {
  lock.acquire();

  // Read/write shared state

  // If testSharedState is now true
  cv.signal();

  // Read/write shared state

  lock.release();
}
```

Bounded Buffer Implementation with Monitors

```
Lock lock;
CV emptyCV, fullCV;

produce(item) {
  lock.acquire();                          // get lock
  while (queue.size() == MAX)                        // wait until there is
    fullCV.wait(&lock);                    // space
  queue.enqueue(item);
  emptyCV.signal();                        // signal waiting costumer
  lock.release();                          // release lock
}
consume() {
  lock.acquire();                          // get lock
  while (queue.isEmpty())                  // wait until there is item
    emptyCV.wait(&lock);
  item = queue.dequeue();
  fullCV.signal();                         // signal waiting producer
  lock.release();                          // release lock
  return item;
}
```



Question

- Does k^{th} call to consume() return item put by k^{th} call to produce()?
  - No! thread calling wait() must re-acquire lock after waking up; before woken-up thread re-acquire lock, a newly arrived thread could acquire lock and consume item

Mesa vs. Hoare Monitors

- Consider piece of consume() code

```
while (queue.isEmpty())
    emptyCV.wait(&lock);
```

- Why didn't we do this?

```
if (queue.isEmpty())
    emptyCV.wait(&lock);
```

- Answer: it depends on the type of scheduling
  - Hoare-style
  - Mesa-style

Condition Variable vs. Semaphore

- CV's signal() has no memory
  - If signal() is called before wait(), then signal is waisted
- Semaphore's V() has memory
  - If V() is called before P(), P() will not wait
- Generally, it's better to use monitors but not always
- Example: interrupt handlers
  - Shared memory is read/written concurrently by HW and kernel
  - HW cannot use SW locks
  - Kernel thread checks for data and calls wait() if there is no data
  - HW write to shared memory, starts interrupt handler to then call signal()
    - This is called *naked notify* because interrupt handler doesn't hold lock (why?)
  - This may not work if signal comes before kernel thread calls wait
  - Common solution is to use semaphores instead

Communicating Sequential Processes (CSP/Google Go)

- Instead of allowing threads to access shared objects, each object is assigned to single corresponding thread
  - Only one thread is allowed to access object's data
- Threads communicate with each other solely through message-passing
  - Instead of calling method on shared object, threads send messages to object's corresponding thread with method name and arguments
- Thread waits in a loop, gets messages, and performs operations
- No race condition!

Bounded Buffer Implementation with CSP

```
while (msg = getNext()) {
  if (msg == GET) {
    if (!queue.isEmpty()) {
      // get item
      // send reply
      // if pending put, do it
      // and send reply
    } else {
      // queue get operation
    }
  } else if (msg == PUT) {
    if (queue.size() < MAX) {
      // put item
      // send reply
      // if pending get, do it
      // and send reply
    } else {
      // queue put operation
    }
  }
}
```

Locks/CVs vs. CSP

- Create a lock on shared data
  = create a single thread to operate on data
- Call a method on a shared object
  = send a message/wait for reply
- Wait for a condition
  = queue an operation that can't be completed just yet
- Signal a condition
  = perform a queued operation, now enabled

Execution of procedure with monitor lock is equivalent to processing message in CSP (monitor is, in effect, single-threaded while it is holding lock)

Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny
- Slides courtesy of Seyed Majid Zahedi