SE 350
Operating
Systems

Lecture 3: Synchronization and Deadlock

Prof. Seyed Majid Zahedi

https://ece.uwaterloo.ca/~smzahedi

Outline

- Race condition
- Mutual exclusion and critical section
- Mutex
- Semaphore
- Condition variable
- Deadlock

Programmer's View vs. Reality

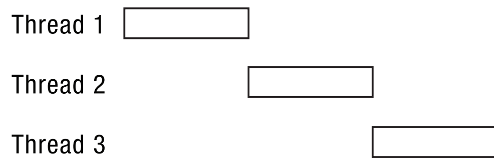
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	....	Thread is resumed.
		y = y + x;	....
		z = x + 5y;	z = x + 5y;

Atomic Memory Operations

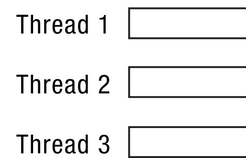
- In most architectures, load and store operations on single byte are atomic
  - Threads cannot get context switched in middle of load/store to/from a *word*
- In x86, load and store operations on *naturally aligned* variables are atomic
  - I.e., aligned to at least multiple of its own size
  - E.g., 8-byte int that is aligned to an address that's multiple of 8
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy whole array
- Unless otherwise stated, we assume loads and stores are atomic

Possible Interleavings

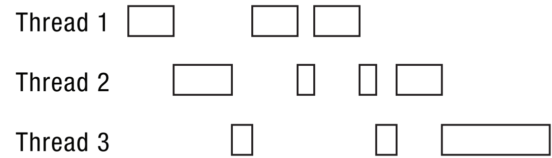
One Execution



Another Execution



Another Execution



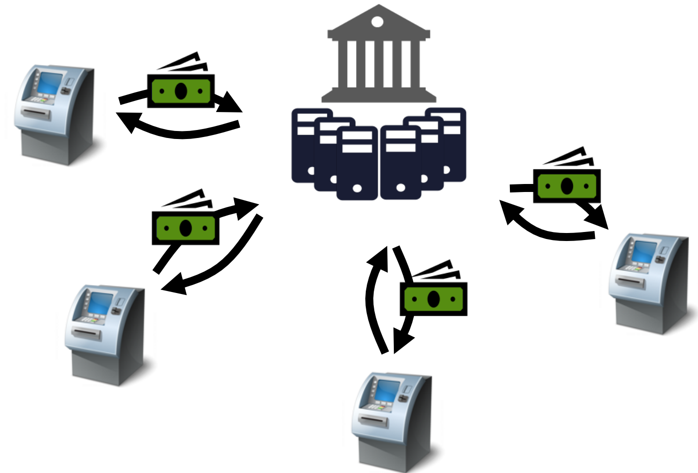
Can this cause any problem?

☐ Yes ☐ No

It depends! In some cases, if program is not written carefully, then **absolutely!**

ATM Example

- Suppose we wanted to implement server process to handle requests from 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 */
}
```

- Is this fast enough?
  - More than one request being processed at once
- How could we speed this up?
  - Multiple threads

ATM photo from iconexperience.com

ATM Example (cont.)

- Using threads could make this faster
  - E.g., using one thread per request
- Do threads make it easier?
  - Unfortunately, not!

Thread 1

```
load r1, acct->balance
```

Thread 2

```
load r1, acct->balance
add r1, amount2
store r1, acct->balance
```

```
add r1, amount1
store r1, acct->balance
```

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 (cont.)

- 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 (cont.)

- 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`

Race Condition

- Condition where output of concurrent program depends on order or timing of threads
- This often happens when threads share some data

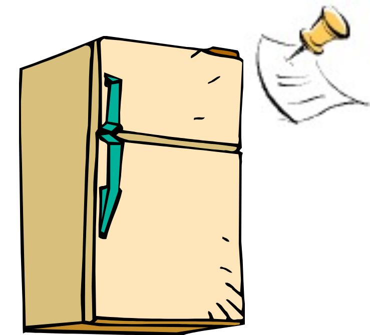
Too Much Milk Example

	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!

- What are correctness properties of "too much milk" problem?
  - At most one roommate should buy milk
  - Someone should eventually buy milk

Too Much Milk (Solution #1)

- Leave note before buying
- Remove note after buying
- Don't buy if there is already a note



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

- Would this work if it is executed by multiple threads?

Solution #1 (cont.)

Thread 1

```
if (!milk) {
```

Thread 2

```
if (!milk) {
   if (!note) {
```

Thread 1

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

Thread 2

```
      leave note;
      buy milk;
      remove note;
   }
}
```

Solution #1 (cont.)

- Conclusion
  - Still too much milk but only occasionally
  - A thread can get context switched after checking milk and leaving 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 leaving 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 human, probably nothing bad
  - With threads, 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
    - I thought *you* had the milk! But I thought *you* had the milk!
  - This kind of lockup is called "starvation!"
  - This is extremely unlikely, but it's still possible

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 so it is ok to quit
- At (X) if there is no note from B, it is safe for A to buy
  - Otherwise, A waits to find out what will happen
- At (Y) if there is no note from A, it is 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

```
// 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;
```

If A checks note B <u>before</u> B leaves the note, then A goes ahead and buys milk

Case I.b

- A leaves note before B checks

```
// 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;
```

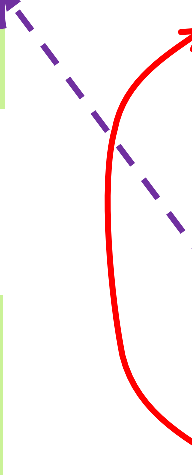
If A checks note B after B leaves the note, then A waits to see what happens

Case I.b (cont.)

- A leaves note before B checks

```
// Thread A
leave note A;
while (note B) // (X)
    do nothing;
```

```
if (!milk)
    buy milk;
remove note A;
```



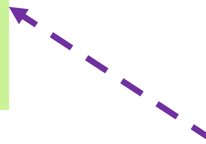
```
// Thread B
leave note B;
if (!note A) { // (Y)
    if (!milk)
        buy milk;
}
remove note B;
```

B will not buy milk!

Case 2

- B checks note A before A leaves it

```
// 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;
```

Mutual Exclusion and Critical Section

- Mutual exclusion: only one thread runs a particular code at any given time
  - One thread excludes others while doing its task
- Critical section: a particular code that only one thread can execute at once
  - Critical section and mutual exclusion are two ways of describing same thing

```
// 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;
```

Solution #3: Discussion

- It's very unsatisfactory
  - Only protects a single critical section
    - Extending this solution to protect multiple critical sections is nontrivial
  - Way too complex even for this simple example
    - It's hard to convince yourself that this really works
  - A's code is different from B's
    - What if there are more than two threads?
    - See Peterson's algorithm
  - A is busy-waiting while B is in critical section
    - While A is waiting, it is consuming CPU time
- It doesn't work on most of today's computers
  - It only works if instructions are executed in program order
  - Compilers and hardware could reorder instructions
    - E.g., store buffer allows next instruction to execute while store is being completed

Question: Can This Panic?

```
// Thread 1

p = someComputation();
pInitialized = true;
```

```
// Thread 2

While (!pInitialized);
q = someFunc(p);
If (q != someFunc(p))
   panic();
```

Aside: Memory Consistency in Multiprocessors

```
// initially flag = data = r1 = r2 = 0
```

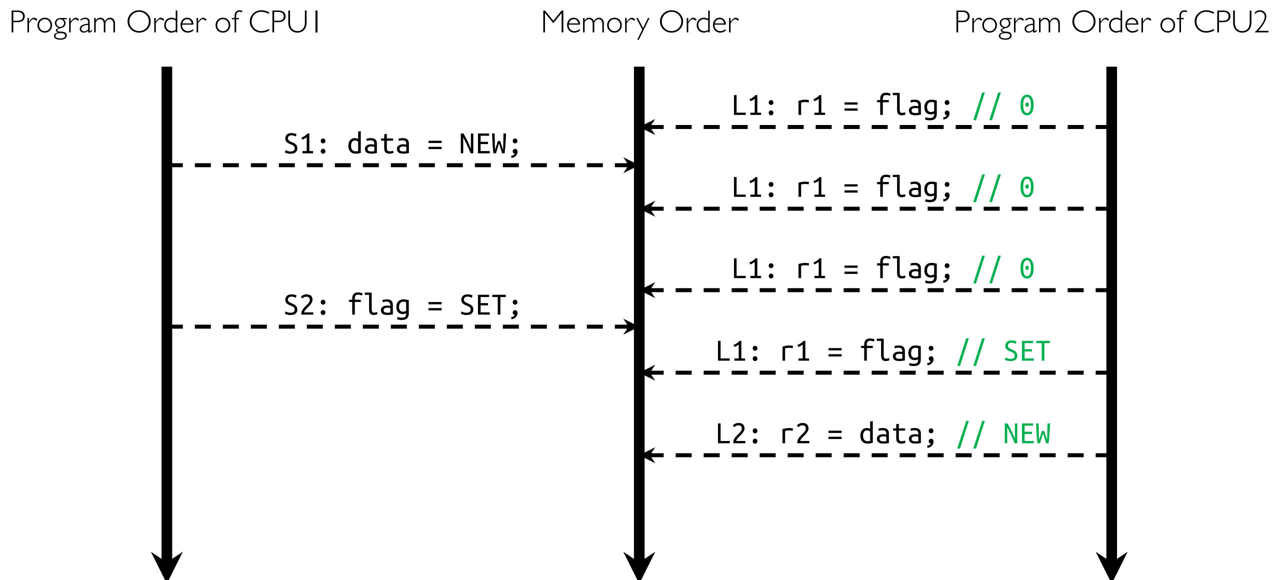
CPU1	CPU2
`S1: data = NEW;`	`L1: r1 = flag;`
`S2: flag = SET;`	`B1: if (r1 != SET) goto L1;`
	`L2: r2 = data;`

- Intuition says we should print `r2 = NEW`
- Yet, in some consistency models, this isn't required!

Sequential Consistency

"The result of any execution is the same as if the operations of all processors (CPUs) were executed in some sequential order, and the operations of each individual processor (CPU) appear in this sequence in the order specified by its program."

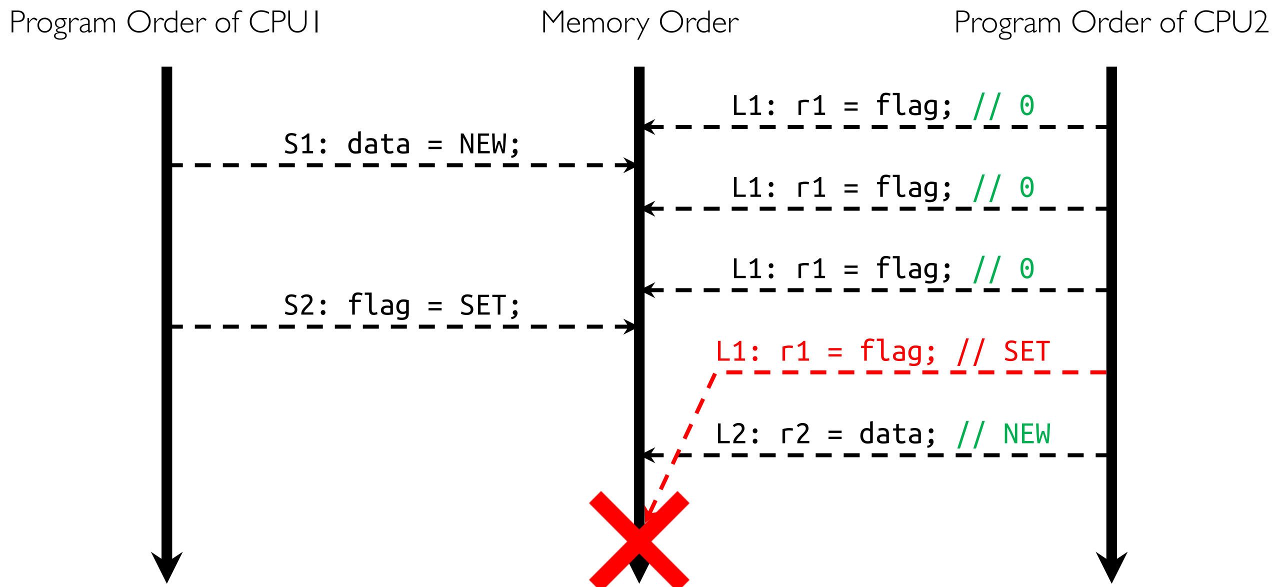
Lamport, 1979



Sequential Consistency (cont.)

"The result of any execution is the same as if the operations of all processors (CPUs) were executed in some sequential order, and the operations of each individual processor (CPU) appear in this sequence in the order specified by its program."

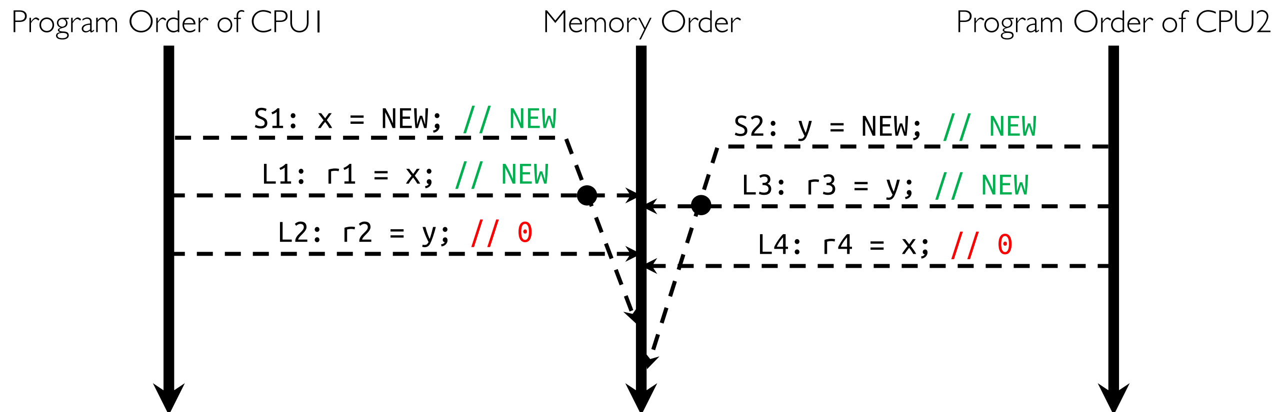
Lamport, 1979



x86 Memory-consistency Model

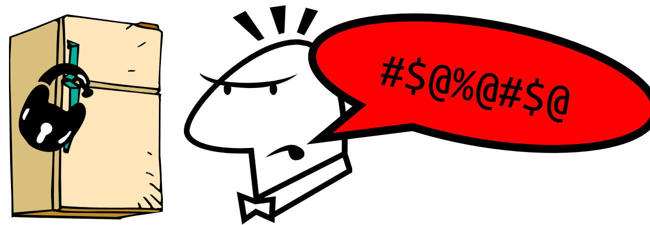
```
// initially x = y = r1 = r2 = r3 = r4 = 0
```

```
CPU1                    CPU2
S1: x = NEW;            S2: y = NEW;
L1: r1 = x;             L3: r3 = y;
L2: r2 = y;             L4: r4 = x;
```



Too Much Milk (Solution #4)

- Fix milk problem by putting a lock on refrigerator
  - Lock refrigerator 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

Solution #4 (cont.)

- Suppose we have some sort of implementation of a lock
  - `mutex.lock()` – wait until lock is free, then grab
  - `mutex.unlock()` – Unlock, waking up anyone waiting

```
milkMutex.lock();
if (no milk)
   buy milk;
milkMutex.unlock();
```

- Code between **`lock()`** and **`unlock()`** is the <u>critical section</u>

Rules for Using Mutex

- Always lock before accessing shared data
  - Best place for locking is beginning of procedure!
- Always unlock after finishing with shared data
  - Best place for unlocking is end of procedure!
  - Only the thread that locked mutex should unlock it
  - DO NOT throw locked mutex to someone else to unlock
- Never access shared data without lock
  - Danger! Don't do it even if it's tempting!

Lock Before Accessing Shared Data, ALWAYS!

```
getP() {
   if (p == NULL) {
      mutex.lock();
      if (p == NULL) {
         temp = malloc(sizeof(…));
         temp->field1 = …;
         temp->field2 = …;
         p = temp;
      }
      mutex.unlock();
   }
  return p;
}
```

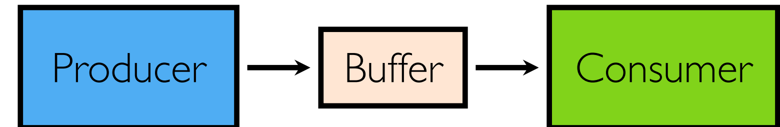
- Safe but expensive solution is

```
getP() {
   mutex.lock();
   if (p == NULL) {
      temp = malloc(sizeof(…));
      temp->field1 = …;
      temp->field2 = …;
      p = temp;
   }
   mutex.unlock();
   return p;
}
```

Does this work?

- No! Compiler/HW could make `p` point to `temp` before its fields are set
- This is called double-checked locking

Example: Bounded Buffer for Producer-Consumer Problem



- Problem definition
  - Producer puts things into shared buffer
  - 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
  - ls | grep foo

- Example 2: newspaper vending machine
  - Producer can put limited number of newspapers in machine
  - 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)

Example: Bounded Buffer With Mutex

```
try_to_produce(item) {
   BBMutex.lock();
   success = FALSE;
   if (buffer.size() < MAX) {
      buffer.enqueue(item);
      success = TRUE;
   }
   BBMutex.unlock();
   return success;
}
```

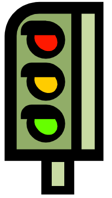
```
try_to_consume() {
   BBMutex.lock();
   item = NULL;
   if (!buffer.isEmpty())
      item = buffer.dequeue();
   BBMutex.unlock();
   return item;
}
```

- 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 unlocking mutex 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
  - This is bad for everyone!

Outline

- Race condition
- Mutual exclusion and critical section
- Mutex
- Semaphore
- Condition variable
- Deadlock

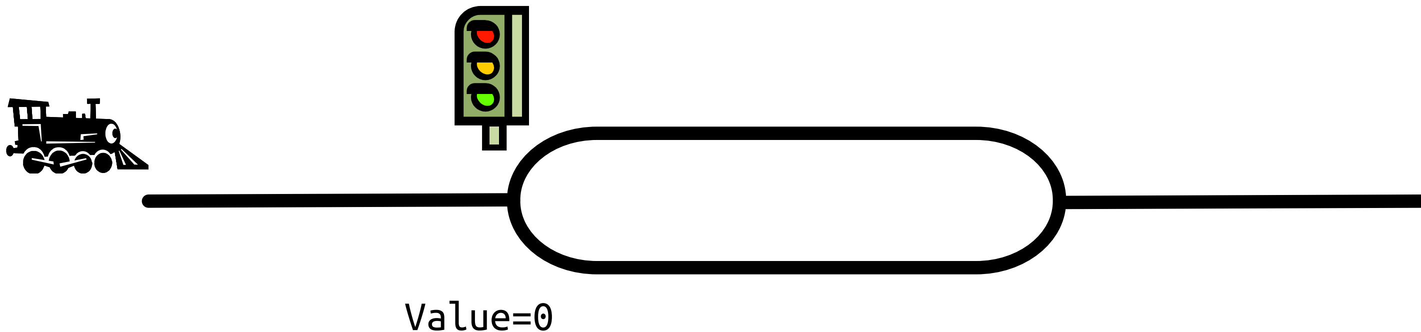
Semaphores



- 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, then wakes up a waiting `P()` (if any)
  - In Dutch, **P** stands for *proberen* (to test) and **V** for *verhogen* (to increment)

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)



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)

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

Initializing semaphores to right value is very important!

Initial Value for Semaphores

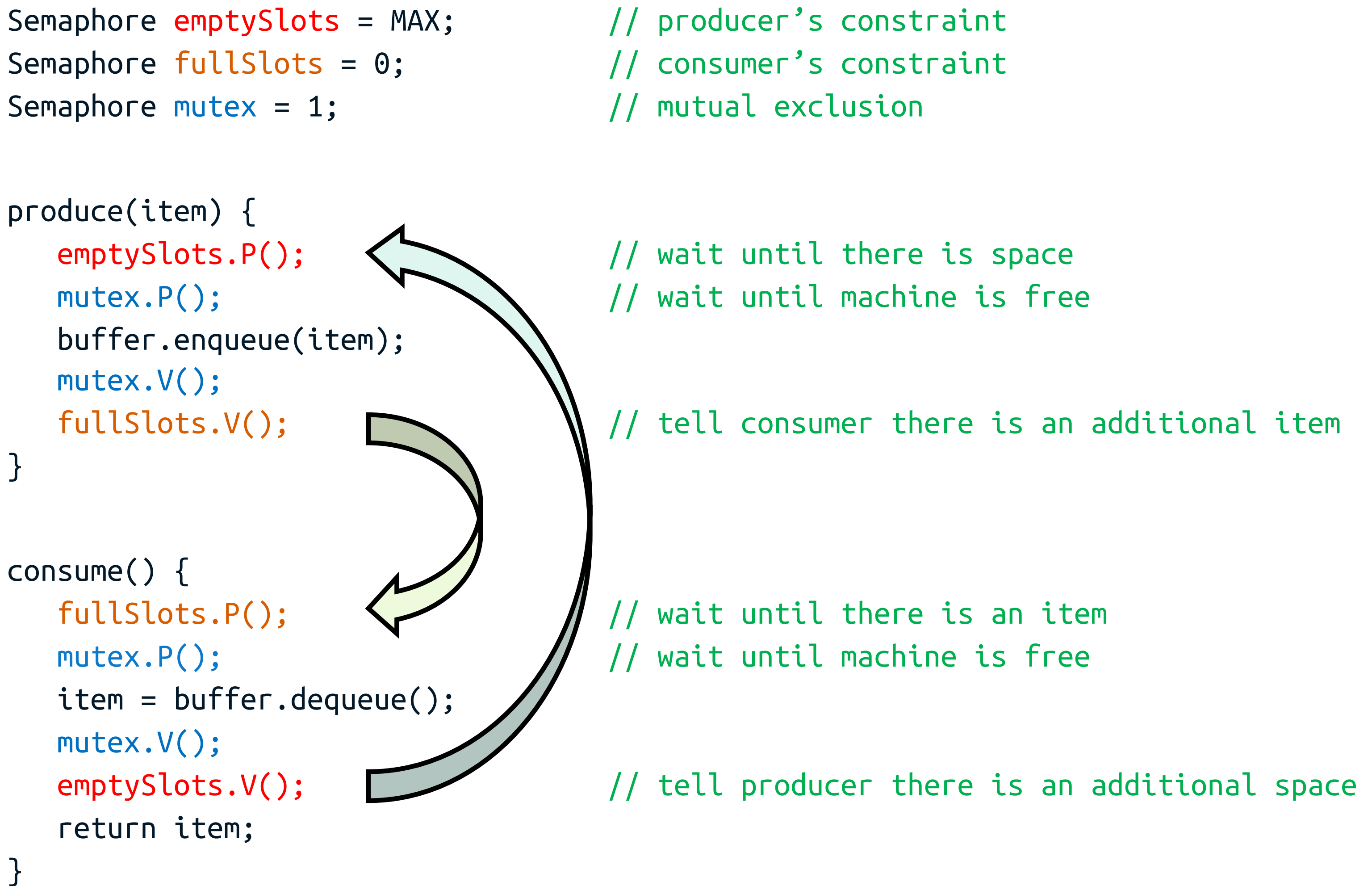
- Mutual exclusion with binary semaphore (initialized to 1)

```
semaphore.P();

// Critical section goes here

semaphore.V();
```

- Scheduling constraints (initialized to 0)

```
wait(…) {
   semaphore.P();
}

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

Example: Bounded Buffer with Semaphore



```
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
   buffer.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 = buffer.dequeue();
   mutex.V();
   emptySlots.V();                   // tell producer there is an additional space
   return item;
}
```

Order of Semaphore Operations

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

https://www.youtube.com/watch?v=kAM-YW-6vdU

```
produce(item) {
     mutex.P();
     emptySlots.P();
     buffer.enqueue(item);
     mutex.V();
     fullSlots.V();
}

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

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 <u>mutual exclusion</u>, on the other hand, the <u>private semaphores</u>."

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

Outline

- Race condition
- Mutual exclusion and critical section
- Mutex
- Semaphore
- Condition variable
- Deadlock

Monitors and Condition Variables

- Problem: analyzing code that uses semaphores is complex
  - They are dual purpose (both mutual exclusion and scheduling constraints)
  - E.g., fact that flipping of **P**'s in bounded buffer gives deadlock is not obvious
- Solution: use monitors
  - It consists of one mutex with zero or more condition variables (CV)
  - Mutex is used for mutual exclusion, CV's are used for scheduling constraints

Condition Variables

- CV is queue of threads waiting for an event inside critical section
  - Makes it possible to go to sleep inside critical section
  - Atomically unlocks mutex at time thread goes to sleep
  - With semaphores, threads cannot wait inside critical section (deadlock)

- CV operations
  - `wait(Mutex *CVMutex)`
    - Atomically unlocks mutex, puts thread to sleep, and relinquishes processor
    - Attempts to locks mutex when thread wakes up
  - `signal()`
    - Wakes up a waiter, if any
  - `broadcast()`
    - Wakes 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()`
- 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() {
   mutex.lock();

   // Read/write shared state

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

   // Read/write shared state

   mutex.unlock();
}
```

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

   // Read/write shared state

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

   // Read/write shared state

   mutex.unlock();
}
```

Example: Bounded Buffer With Monitors

```
Mutex BBMutex;
CV emptyCV, fullCV;


produce(item) {
   BBMutex.lock();                          // lock the mutex
   while (buffer.size() == MAX)             // wait until there is space
      fullCV.wait(&BBMutex);
   buffer.enqueue(item);
   emptyCV.signal();                        // signal waiting costumer
   BBMutex.unlock();                        // unlock the mutex
}

consume() {
   lock.acquire();                          // lock the mutex
   while (buffer.isEmpty())                 // wait until there is item
      emptyCV.wait(&lock);
   item = buffer.dequeue();
   fullCV.signal();                         // signal waiting producer
   lock.release();                          // unlock the mutex
   return item;
}
```

Mesa vs. Hoare Monitors

- Consider piece of `consume()` code

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

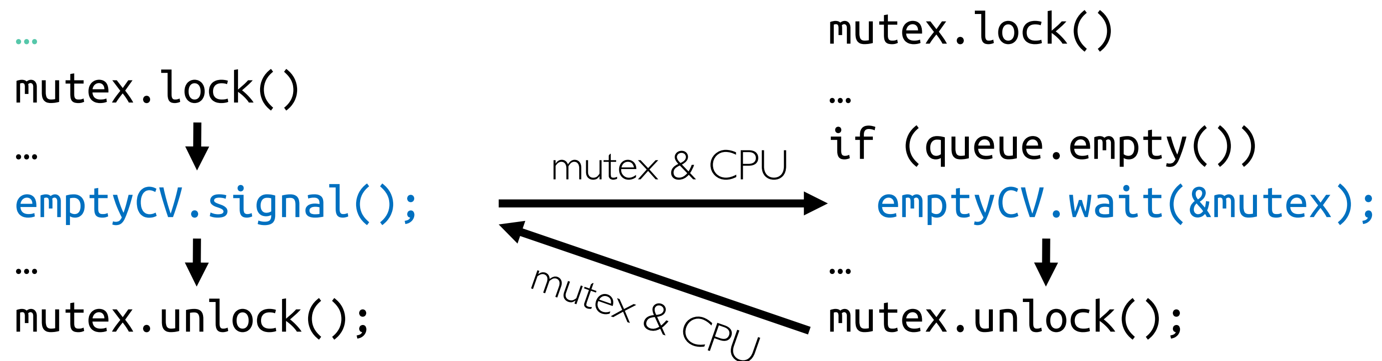
- Why didn't we do this?

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

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

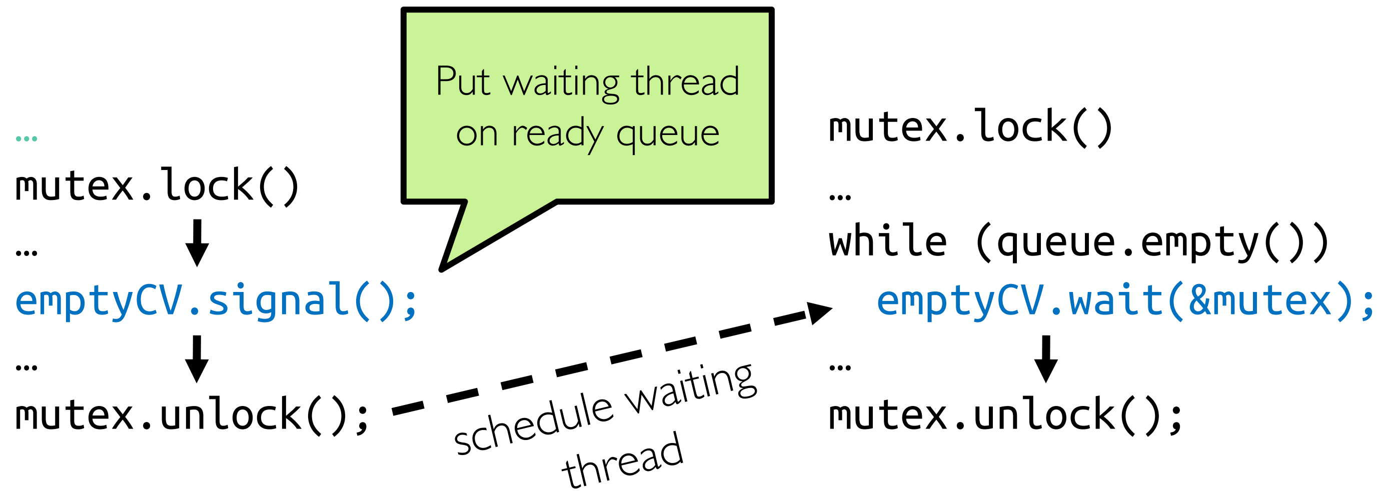
Hoare Monitors

- Signaler gives up mutex and processor to waiter – waiter runs immediately
- Waiter gives up mutex and processor back to signaler when it exits critical section or if it waits again



Mesa Monitors

- Signaler keeps mutex and processor
- Waiter placed on ready queue with no special priority
- Practically, need to check condition again after wait
- Most real operating systems



Mesa Monitor: Why "while()"?

- What if we use "**if**" instead of "**while**" in bounded buffer example?

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

```
produce(item) {
  mutex.lock();
  if (queue.size() == MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

Monitor

mutex: unlocked
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL
...

T1 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

Monitor

mutex: locked (T1)
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL
…

T1 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked

emptyCV queue → T1

CPU State

Running:

ready queue → NULL

...

T1 (Waiting)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

wait(&lock) puts thread on emptyCV queue and releases lock

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked
emptyCV queue → T1

CPU State

Running: T2
ready queue → NULL
...

T1 (Waiting)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T2 (Running)

```
produce(item) {
  mutex.lock();
  if (queue.size()==MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T2)
emptyCV queue → T1

CPU State

Running: T2
ready queue → NULL
...

T1 (Waiting)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T2 (Running)

```
produce(item) {
  mutex.lock();
  if (queue.size()==MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T2)
emptyCV queue → NULL

CPU State

Running: T2
ready queue → T1
...

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T2 (Running)

```
produce(item) {
  mutex.lock();
  if (queue.size()==MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

signal() wakes up and moves it to ready queue

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T2)
emptyCV queue → NULL

CPU State

Running: T2
ready queue → T1,T3
...

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T2 (Running)

```
produce(item) {
  mutex.lock();
  if (queue.size()==MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

T3 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked
emptyCV queue → NULL

CPU State

Running:
ready queue → T1,T3
...

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T2 (Terminated)

```
produce(item) {
  mutex.lock();
  if (queue.size()==MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

T3 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked
emptyCV queue → NULL

CPU State

Running: T3
ready queue → T1

T3 is scheduled first

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T3 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T3)
emptyCV queue → NULL

CPU State

Running: T3
ready queue → T1
...

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T3 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T3)
emptyCV queue → NULL

CPU State

Running: T3
ready queue → T1
...

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T3 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked
emptyCV queue → NULL

CPU State

Running:
ready queue → T1
...

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T3 (Terminated)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T1)
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL
…

T1 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T1)
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL
...

T1 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Error!

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T1)
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL
...

T1 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Check again if empty!

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked

emptyCV queue → T1

CPU State

Running:

ready queue → NULL

...

T1 (Waiting)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

When waiting upon a *Condition*, a spurious wakeup is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a *Condition* should always be waited upon in a loop, testing the state predicate that is being waited for

From Java User Manual

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 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
      }
   }
}
```

Monitors vs. CSP

- Use mutex on shared data
  = assign 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

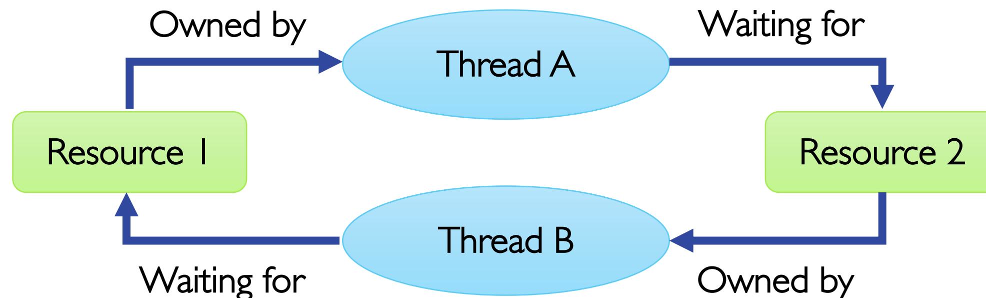
Execution of procedure with monitor lock is equivalent to processing message in CSP
(monitor is, in effect, single-threaded while it is holding lock)

Outline

- Race condition
- Mutual exclusion and critical section
- Mutex
- Semaphore
- Condition variable
- Deadlock

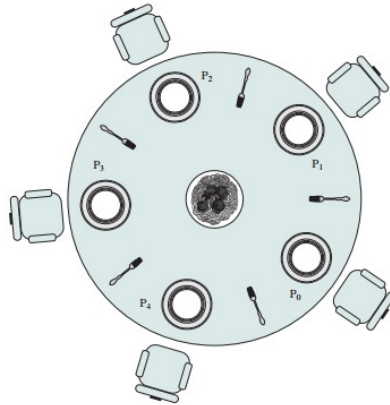
Starvation vs. Deadlock

- 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 resource 1 and is waiting for resource 2
  - Thread B owns resource 2 and is waiting for resource 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

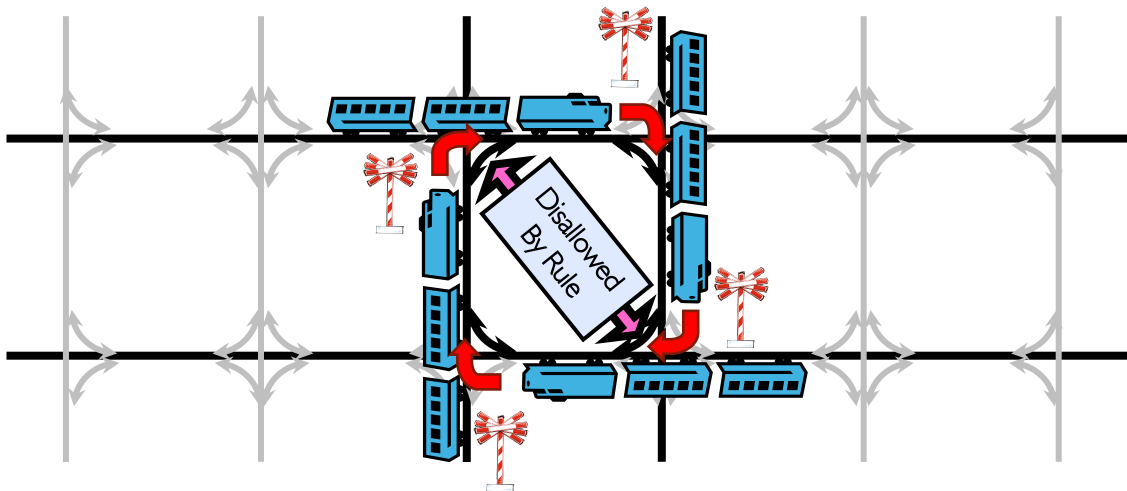
Dining ~~Philosophers~~ Politicians!



- 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

Wormhole-routed Network

- Each train wants to turn right but is blocked by other trains
- Similar problem to multiprocessor networks
- How to fix this? (Imagine grid extends in all four directions)
  - Force ordering of channels (tracks)
    - Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



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();
```

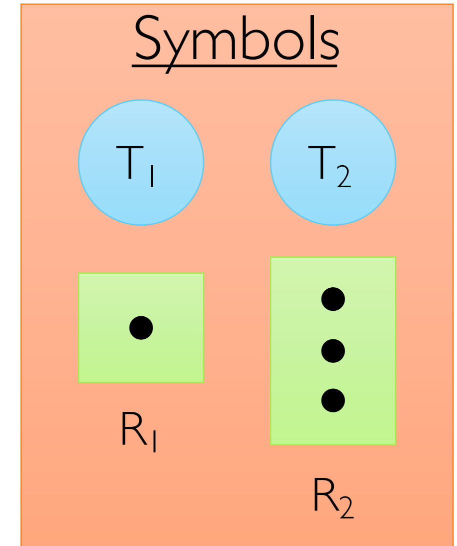
- This code doesn't always lead to deadlock
  - Must have exactly right timing ("wrong" timing?)
  - So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant…
- Deadlocks occur with multiple resources
  - Can't solve deadlock for each resource independently

Four Requirements for Deadlock

- Mutual exclusion
  - Only limited number of threads at a time can use resource
- Hold and wait
  - Thread holds resources while waiting to acquire additional ones
- No preemption
  - Resources are released only voluntarily by. the thread holding them
- Circular wait
  - There exists a set $T_1, \ldots, T_n$ 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

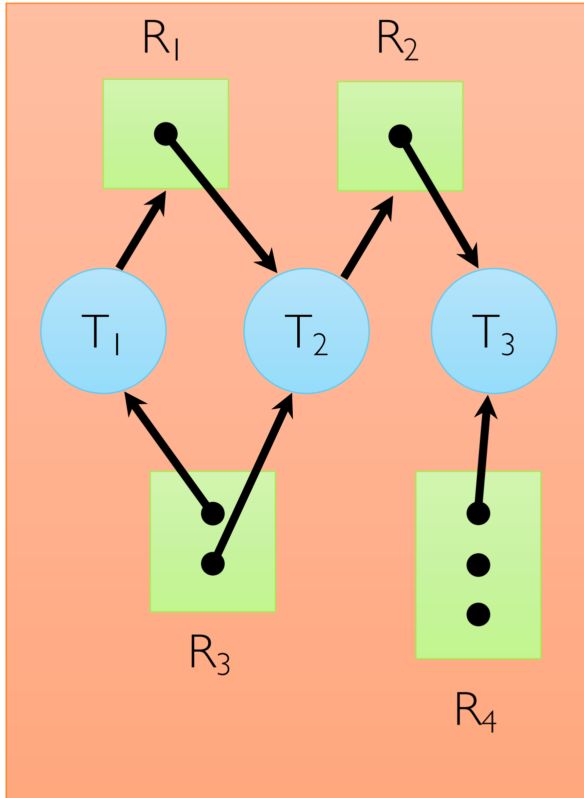
Resource Allocation Graph

- 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()`

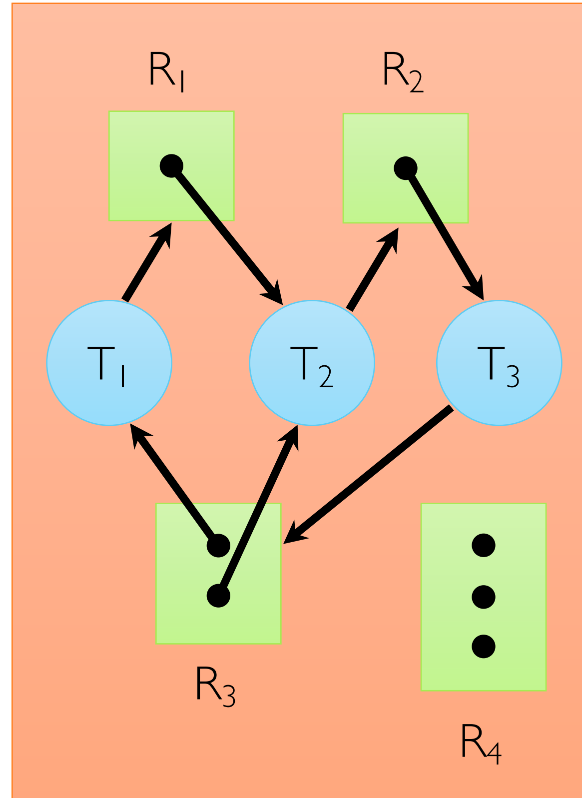


- Resource allocation graph
  - V is partitioned into two types
    - $T = \{T_1, \ldots, T_n\}$, set of threads in system
    - $R = \{R_1, \ldots, R_m\}$, set of resource types in system
  - Request edge is directed edge $T_i \rightarrow R_j$
  - Assignment edge is directed edge $R_q \rightarrow T_p$

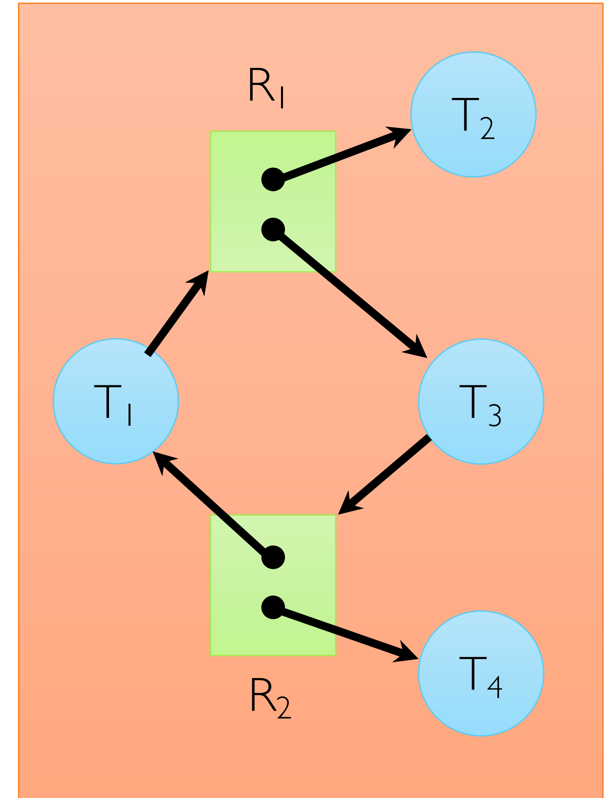
Resource Allocation Graph Examples



Simple resource allocation graph



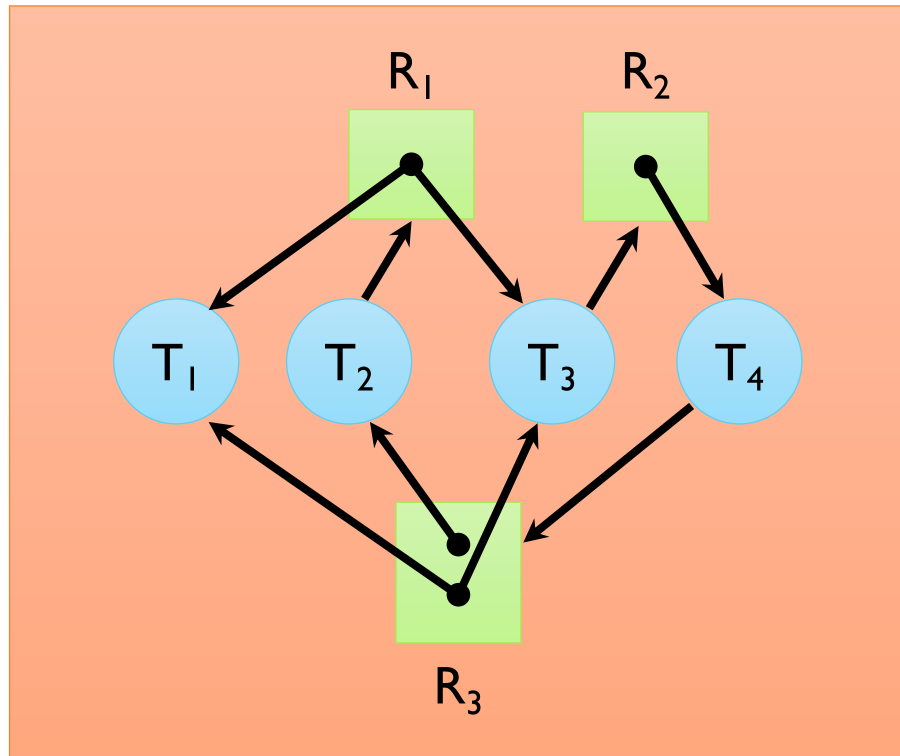
Allocation graph with deadlock



Allocation graph with cycle, but no deadlock

Resource Requests Over Time

- Applications usually don't know exactly when/what they'll request
- Resources are taken/released over time



Methods for Handling Deadlocks

- Ignore problem and pretend deadlocks never occur
  - Used by most operating systems, including UNIX

- Allow system to enter deadlock and then recover
  - Requires deadlock-detection and deadlock-recovery algorithms
- Ensure that system will never enter deadlock
  - Need to monitor all resource acquisitions
  - Selectively deny those that might lead to deadlock

Deadlock Detection

m is the number of resources and n is the number of threads

Resource capacities

$$[C_1 \quad C_2 \quad \dots \quad C_m]$$

Free resources

$$[F_1 \quad F_2 \quad \dots \quad F_m]$$

Resource allocations

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,m} \\ A_{2,1} & A_{2,2} & \dots & A_{2,m} \\ \vdots & \vdots & \dots & \vdots \\ A_{n,1} & A_{n,2} & \dots & A_{n,m} \end{bmatrix}$$

Resource requests

$$\begin{bmatrix} R_{1,1} & R_{1,2} & \dots & R_{1,m} \\ R_{2,1} & R_{2,2} & \dots & R_{2,m} \\ \vdots & \vdots & \dots & \vdots \\ R_{n,1} & R_{n,2} & \dots & R_{n,m} \end{bmatrix}$$

$$\sum_{i=1}^{n} A_{i,j} + F_j = C_j$$

Deadlock Detection (cont.)

```
Add all threads to UNFINISHED
do {
   done = true
   foreach i in UNFINISHED {
      if ([R_i,1 R_i,2 … R_i,m] <= [F_1 F_2 … F_m]) {
         remove i from UNFINISHED
         [F_1 F_2 … F_m] = [F_1 F_2 … F_m] + [A_i,1 A_i,2 A_i,m]
         done = false
      }
   }
} until(done)
```

look for threads that can run to completion

Once finished, thread's currently-held resources are released

Nonempty UNFINISHED ⇒ deadlocked

When to Detect Deadlock?

- Option 1: run every time a resource is requested
- Option 2: run every time a request cannot be granted
- Option 3: run periodically

Deadlock Recovery

- Terminate victim thread, force it to give up resources
  - Run deadlock detection algorithm again to determine if deadlock still exists
  - Repeat until the logjam is broken
  - But, not always possible: killing thread holding mutex leaves world inconsistent

- Proceed without the resource
  - Requires robust exception handling code
  - E.g., Amazon will say you can buy book, if inventory subsystem doesn't reply quickly enough (wrong answer quickly is better than right answer slowly)

- Roll back actions of deadlocked threads
  - There must be a saved state that was created in advance of a problem
  - The saved state is called a checkpoint and the act of creating and saving a checkpoint is called checkpointing
  - Of course, if you restart in the same way, may reenter deadlock once again

Deadlock Prevention

- 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)
- 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 told to retry!

Deadlock Prevention (cont.)

- Make all threads request everything they'll need at the beginning
  - E.g., If need 2 chopsticks, request both at same time
  - Problem: predicting future is hard, tend to over-estimate resources

- Force all threads to request resources in fixed order preventing any cyclic use of resources
  - E.g., number forks from 1 to 5, each politician must pick up the lower-numbered fork first, and then the higher-numbered fork
  - E.g.,

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

Safe/Unsafe State

- Each thread reports its max required resource
- A state is safe if there is some scheduling order in which every thread can run to completion even if all threads request their maximum resources
- Any state that is not safe is considered unsafe
- If system is in a safe state, then there is no deadlock
- Being in unsafe state doesn't mean there is deadlock, it means deadlock is possible
- We do a worst-case analysis: every thread immediately requests maximum resources it could ever use

Example: Being Safe

- There are three processes A, B, and C
- Assume there is only one resource, and a maximum of 10 instances exist
- Suppose A has 3 resources but may request up to 9
- B has 2 and may request up to 4
- C has 2 and may request up to 7
- There are 3 resources currently free

Thread	Has	Max
A	3	9
B	2	4
C	2	7

Thread	Has	Max
A	3	9
B	4	4
C	2	7

Thread	Has	Max
A	3	9
B	0	0
C	2	7

Thread	Has	Max
A	3	9
B	0	0
C	7	7

Thread	Has	Max
A	9	9
B	0	0
C	0	0

Example: Crossing the Line

- Suppose, A requests and gets another resource
- Now, there are 2 free resources

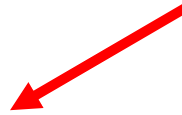
Thread	Has	Max
A	4	9
B	2	4
C	2	7

Thread	Has	Max
A	4	9
B	4	4
C	2	7

Thread	Has	Max
A	4	9
B	0	0
C	2	7

Banker's Algorithm

```
Add all thread to UNFINISHED
do {
       done = true
       foreach thread in UNFINISHED {
           if ([Max_{i,1} Max_{i,2} … Max_{i,m}]- [A_{i,1} A_{i,2} … A_{i,m}] <= [F_1 F_2 … F_m]) {
               remove node from UNFINISHED
               [F_1 F_2 … F_m] = [F_1 F_2 … F_m] + [A_{i,1} A_{i,2} … A_{i,m}]
               done = false
           }
       }
} until(done)
```



Each process might need "max" resources in order to finish

- Keeps system in "SAFE" state
- There exists a sequence $\{T_1, \ldots, T_n\}$ with T_1 requesting all its remaining resources and finishing, then T_2 requesting all remaining resources, etc.
- Algorithm allows sum of maximum resource needs of all current threads to be greater than total resources

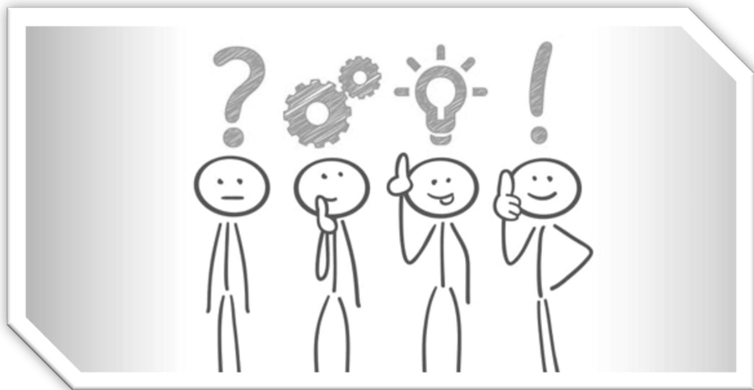
Deadlock Prevention: The Reality

- Deadlock prevention is HARD
  - How many resources will each thread need?
  - How many total resources are there?
- It is also slow and impractical
  - Matrix of resources/requirements could be big and dynamic
  - Re-evaluate on every request (even for small/non-contended)
  - Banker's algorithm assumes everyone asks for max
- REALITY
  - Most OSs don't bother
  - Programmers job to write deadlock-free programs (e.g., by ordering all resource requests).

Summary

- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent
- Synchronization objects can be used for mutual exclusion
  - Mutex is like lock with two operations – **`lock`** and **`unlock`**
  - Semaphore is like integers with only two operations – **`P`** and **`V`**
  - Monitor consists of mutex plus one or more condition variables
    - CV supports three operations: **`wait`**, **`signal`**, and **`broadcast`**
- Four conditions for deadlocks
  - Mutual exclusion, hold and wait, no preemption, circular wait
- Techniques for addressing deadlock
  - Allow system to enter deadlock and then recover
  - Ensure that system will never enter a deadlock
  - Ignore problem and pretend that deadlocks never occur in system

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Zarnett, Joseph, and Canny