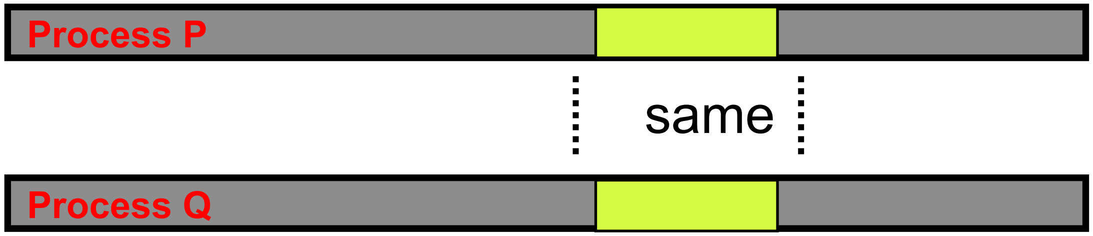
PROCESS SYNCHRONIZATION

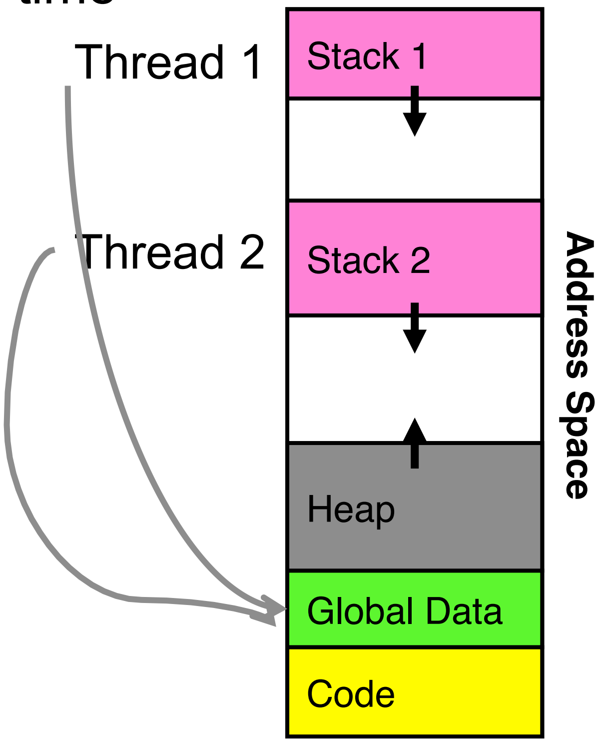
READINGS: CHAPTER 5

ISSUES IN COOPERING PROCESSES AND THREADS – DATA SHARING

Shared Memory

- Two or more processes share a part of their address space
- Incorrect results whenever two processes (or two threads of a process) modify the same data at the same time





EXAMPLE 1: PRODUCER - CONSUMER

count – # of items in the buffer, shared variable

```
Producer

while (count == BUFFER.SIZE)
  ; // do nothing

// add an item to the buffer
buffer[in] = item;
in = (in + 1) % BUFFER.SIZE;
++count;
```

```
Consumer

while (count == 0)
  ; // do nothing

// remove an item from the buffer
item = buffer[out];
out = (out + 1) % BUFFER.SIZE;
--count;
```

RACE CONDITION

count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

count-- could be implemented as

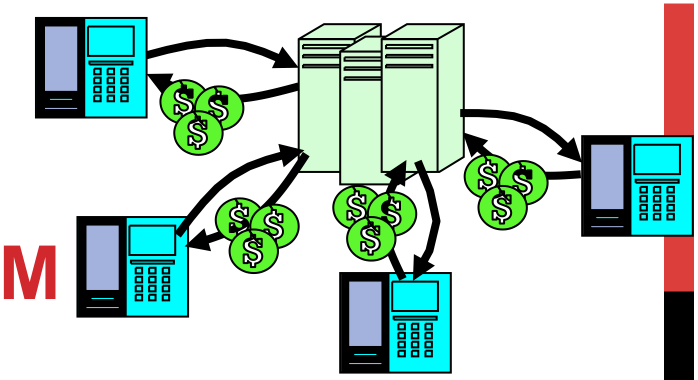
```
register2 = count
register2 = register2 - 1
count = register2
```

Consider this execution interleaving with "count = 5" initially:

T0: producer execute register1 = count {register1 = 5}
T1: producer execute register1 = register1 + 1 {register1 = 6}
T2: consumer execute register2 = count {register2 = 5}
T3: consumer execute register2 = register2 - 1 {register2 = 4}
T4: producer execute count = register1 {count = 6 }
T5: consumer execute count = register2 {count = 4}

count++ and count-- are not atomic operations!

EXAMPLE 2: BANKING PROBLEM



Speed up server by using multiple threads (one per request)

- Can use multi-processor, or overlap comp and I/O

Requests proceeds to completion, blocking as required:

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

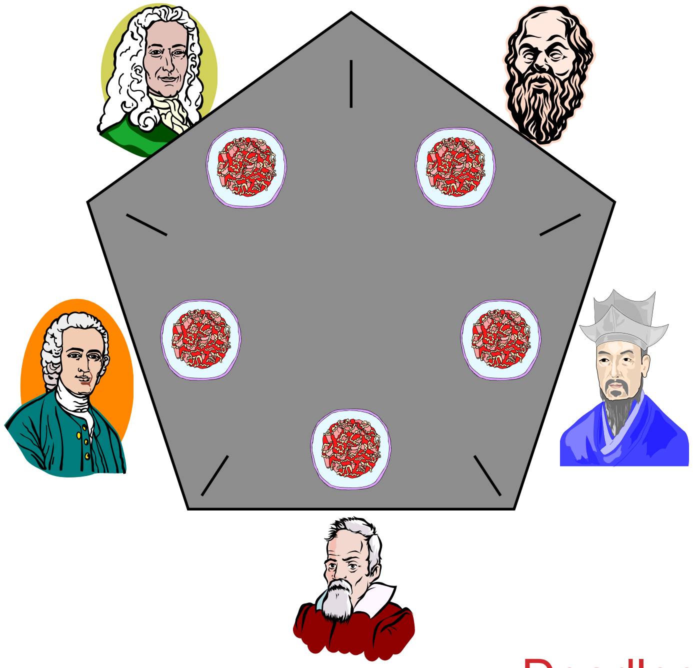
Unfortunately, shared state can get corrupted:

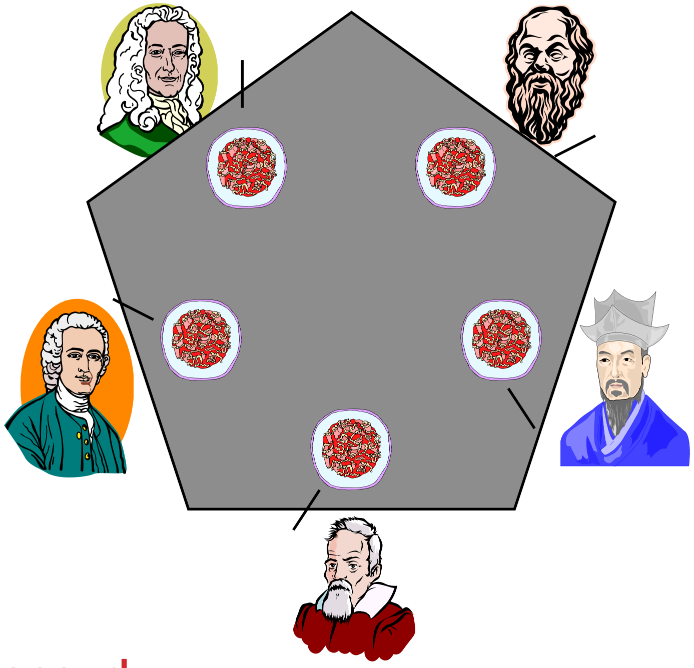
```
            Thread 1                            Thread 2
 load r1, acct->balance
                                  load r1, acct->balance
                                  add r1, amount2
                                  store r1, acct->balance
add r1, amount1
store r1, acct->balance
```

EXAMPLE 2: DINNING PHILOSOPHER'S PROBLEM

First suggested by Dijkstra in 1971

- Philosophers eat/think
- Eating needs 2 chopsticks
- Pick one chopstick at a time





Deadlock may occur!

EXAMPLE 3: SOJOURNER ROVER

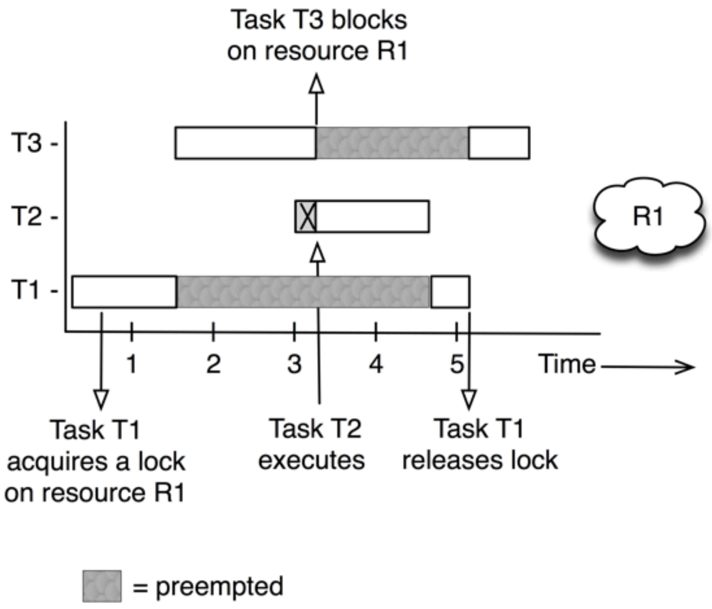
Mars Pathfinder, a NASA space probe landed a robot, the Sojourner rover, on Mars in 1997

Shortly after the Sojourner began operating, it started to experience frequent computer resets.

Priority: T3 > T2 > T1

Problem: T3 may be blocked for a long period of time

Solution: priority inheritance



DEFINITIONS

Synchronization: using atomic operations to ensure cooperation between threads

- For now, only loads and stores are atomic

Critical Section: piece of code that only one thread can execute at once

Mutual Exclusion: ensuring that only one thread executes critical section

- One thread excludes the other while doing its task
- Critical section and mutual exclusion are two ways of describing the same thing

REQUIREMENTS

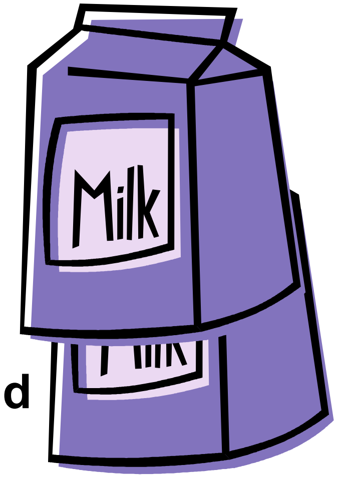
Mutual exclusion: No two processes may be simultaneously into their critical sections for the same shared data

Progress: No process should be prevented to enter its critical section when no other process is inside its own critical section for the same shared data

No starvation: No process should have to wait forever to enter a critical section

Starvation with progress?

MOTIVATION: “TOO MUCH MILK”



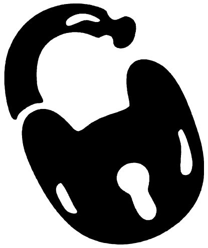
Great thing about OS’s – analogy between problems in OS and problems in real life

- Help you understand real life problems better
- But, computers are much stupider than people

Example: People need to coordinate:

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

LOCK

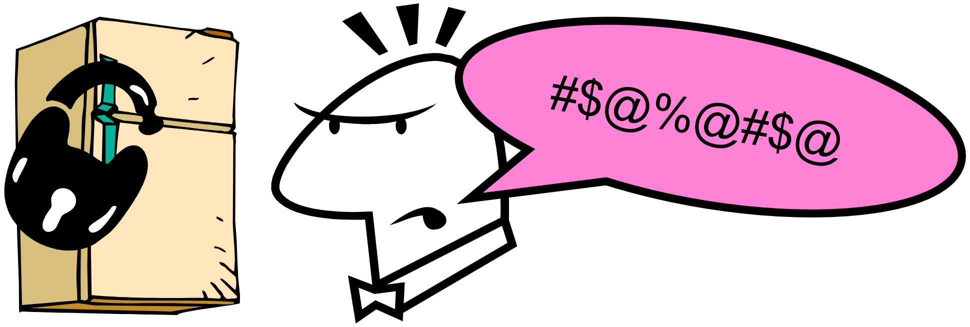


Prevents someone from doing something

- Lock before entering critical section and before accessing shared data
- Unlock when leaving, after accessing shared data
- Wait if locked
    - Important idea: all synchronization involves waiting

Example: fix the milk problem by putting a lock on refrigerator

- Lock it and take key if you are going to go buy milk
- Fixes too much (coarse granularity): roommate angry if only wants orange juice



- Of Course – We don't know how to make a lock yet

TOO MUCH MILK: CORRECTNESS PROPERTIES

Need to be careful about correctness of concurrent programs, since non-deterministic

- Always write down desired behavior first
- Impulse is to start coding first, then when it doesn’t work, pull hair out
- Instead, think first, then code

What are the correctness properties for the “Too much milk” problem?

- Never more than one person buys
- Someone buys if needed

Restrict ourselves to use only atomic load and store operations as building blocks

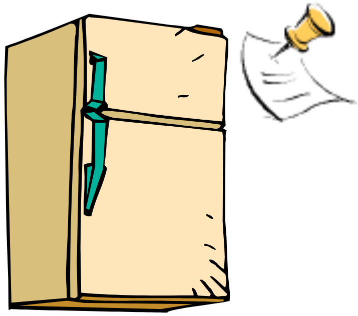
TOO MUCH MILK: SOLUTION #1

Use a note to avoid buying too much milk:

- Leave a note before buying (kind of “lock”)
- Remove note after buying (kind of “unlock”)
- Don’t buy if note (wait)

Suppose a computer tries this (remember, only memory read/write are atomic):

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



Result?

TOO MUCH MILK: SOLUTION #1

Still too much milk but only occasionally!

```
Thread A                      Thread B

 if (noMilk)

   if (noNote) {

                               if (noMilk)

                                 if (noNote) {

     leave Note;
         buy milk;

     remove note;

   }

 }

                                   leave Note;
                                       buy milk;
```

Thread can get context switched after checking milk and note but before leaving note!

Solution makes problem worse since fails intermittently

- Makes it really hard to debug…
- Must work despite what the thread dispatcher does!

Check and setting
are not atomic

TOO MUCH MILK: SOLUTION #1½

Clearly the Note is not quite blocking enough

- Let’s try to fix this by placing note first

Another try at previous solution:

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

What happens here?

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

TOO MUCH MILK SOLUTION #2

How about labeled notes?

- Now we can leave note before checking

Algorithm looks like this:

```
Thread A                    Thread B

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

Does this work?

TOO MUCH MILK SOLUTION #2

Possible for neither thread to buy milk!

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

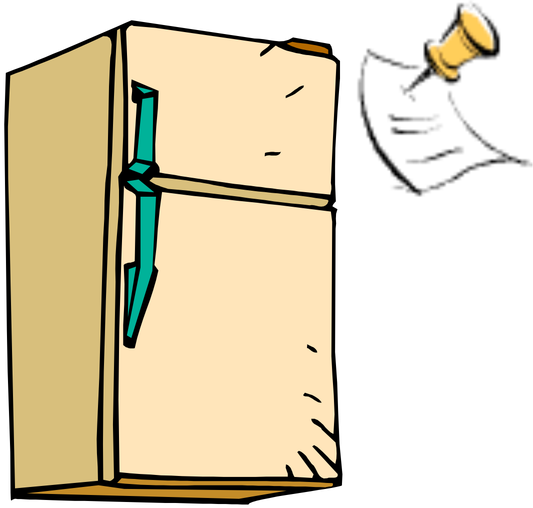
Really insidious:

- Unlikely that this would happen, but will at worse possible time

TOO MUCH MILK SOLUTION #2: PROBLEM!

I’m not getting milk, You’re not getting milk

This kind of lockup is called “starvation!”



TOO MUCH MILK SOLUTION #3

Here is a possible two-note solution:

```
Thread A                    Thread B

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

Does this work? Yes. Both can guarantee that:

- It is safe to buy, or
- Other will buy, ok to quit

At X:

- if no note B, safe for A to buy,
- otherwise wait to find out what will happen

At Y:

- if no note A, safe for B to buy
- Otherwise, A is either buying or waiting for B to quit

SOLUTION #3 DISCUSSION

Our solution protects a single "Critical-Section" piece of code for each thread:

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

Solution #3 works, but it's really unsatisfactory

- Really complex – even for this simple an example
    - Hard to convince yourself that this really works
- A's code is different from B's – what if lots of threads?
    - Code would have to be slightly different for each thread
- While A is waiting, it is consuming CPU time
    - This is called "busy-waiting"

There's a better way

- Have hardware provide better (higher-level) primitives than atomic load and store
- Build even higher-level programming abstractions on this new hardware support

HIGH-LEVEL PICTURE

The abstraction of threads is good:

- Maintains sequential execution model
- Allows simple parallelism to overlap I/O and computation

Unfortunately, still too complicated to access state shared between threads

- Consider “too much milk” example
- Implementing a concurrent program with only loads and stores would be tricky and error-prone

We’ll implement higher-level operations on top of atomic operations provided by hardware

- Develop a “synchronization toolbox”
- Explore some common programming paradigms

TOO MUCH MILK: SOLUTION #4

Suppose we have some sort of implementation of a lock (more in a moment)

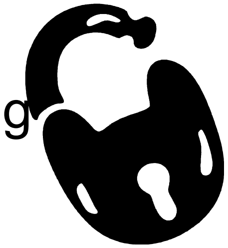
- Lock.Acquire() – wait until lock is free, then grab
- Lock.Release() – unlock, waking up anyone waiting
- These must be atomic operations – if two threads are waiting for the lock, only one succeeds to grab the lock

Then, our milk problem is easy:

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

Once again, section of code between Acquire() and Release() called a "Critical Section"

HOW TO IMPLEMENT LOCK?



Lock: prevents someone from accessing something

- Lock before entering critical section (e.g., before accessing shared data)
- Unlock when leaving, after accessing shared data
- Wait if locked
  - Important idea: all synchronization involves waiting
  - Should sleep if waiting for long time

ROADMAP

How to implement Acquire() and Release()

1. **By disabling/enabling interrupt**
   - A bad implementation
   - A better implementation
2. **Using atomic read/write**
   - A bad implementation that may busy wait a long time
   - A better implementation
3. **A more sophisticated lock – semaphore**
4. **A safer implementation – monitor and conditional variable**

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

NAÏVE USE OF INTERRUPT ENABLE/DISABLE

How can we build multi-instruction atomic operations?

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

Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }

LockRelease { enable Ints; }
```

NAÏVE USE OF INTERRUPT ENABLE/DISABLE: PROBLEMS

Can't let user do this! Consider following:

```
LockAcquire();
While(TRUE) {;}
```

Real-Time system—no guarantees on timing!

- Critical Sections might be arbitrarily long

BETTER IMPLEMENTATION OF LOCKS BY DISABLING INTERRUPTS

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

```
int value = FREE;
```

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

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

NEW LOCK IMPLEMENTATION: DISCUSSION

Disable interrupts: avoid interrupting between checking and setting lock value

- Otherwise two threads could think that they both have lock

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

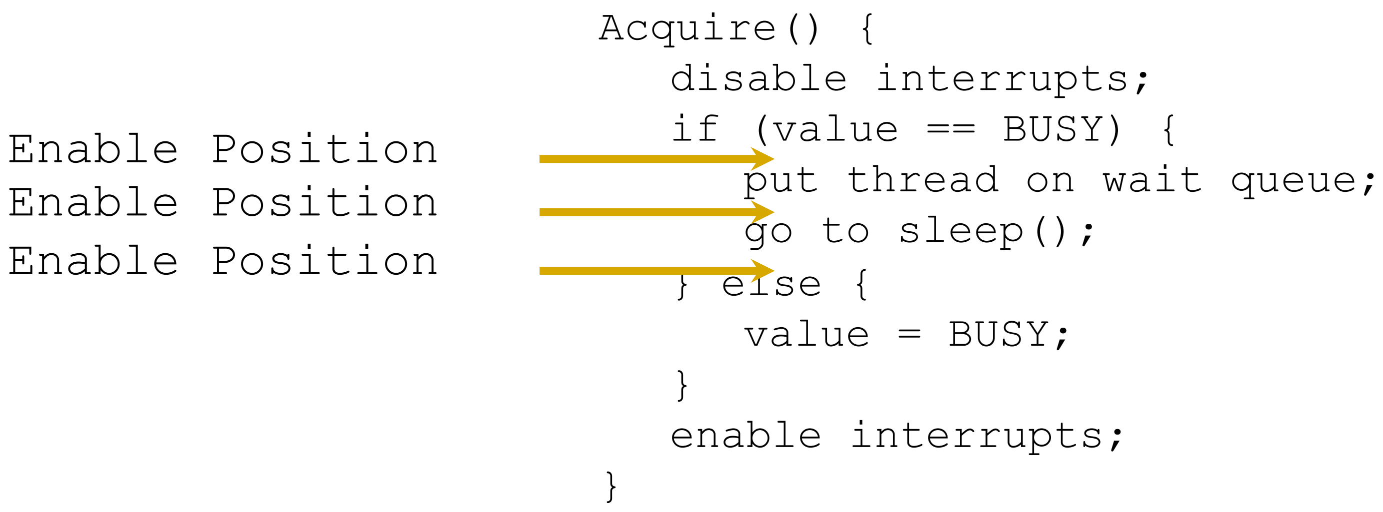
Critical Section

Note: unlike previous solution, critical section very short

- User of lock can take as long as they like in their own critical section
- Critical interrupts taken in time

INTERRUPT RE-ENABLE IN GOING TO SLEEP

What about re-enabling ints when going to sleep?



Enable Position
Enable Position
Enable Position

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

Before putting thread on the wait queue?

- Release can check the queue and not wake up thread

After putting the thread on the wait queue

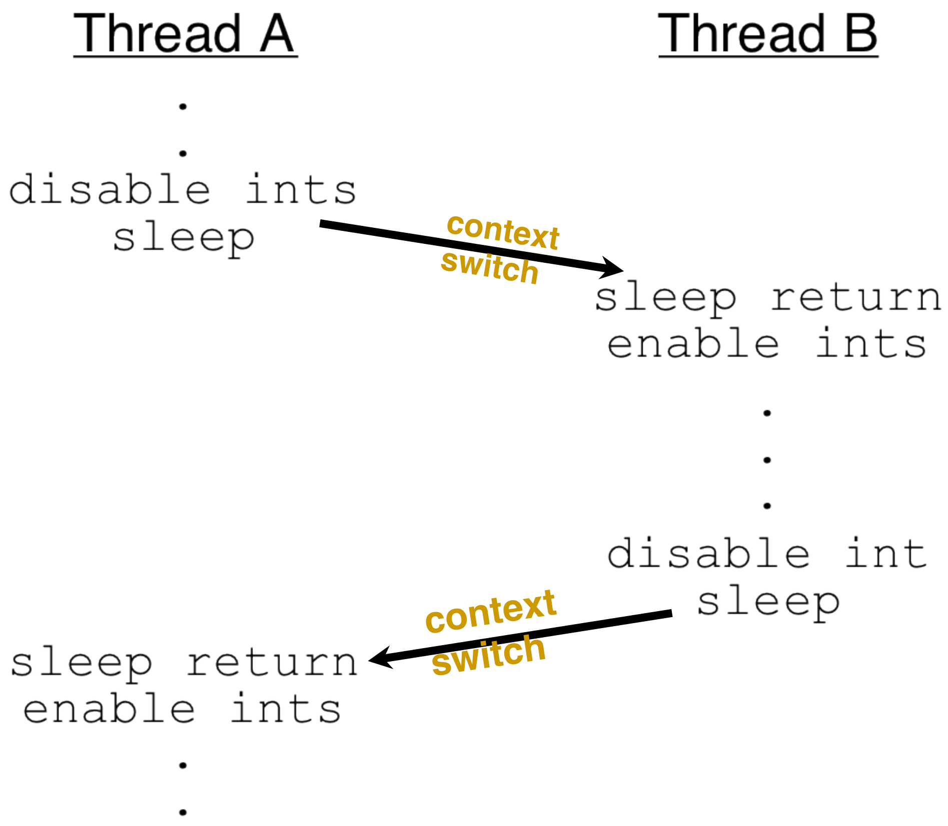
- Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep

Want to put it after sleep(). But, how?

HOW TO RE-ENABLE AFTER SLEEP()?

Since ints are disabled when you call sleep:

- Responsibility of the next thread to re-enable ints
- When the sleeping thread wakes up, returns to acquire and re-enables interrupts



Thread A

```
.
.
disable ints
sleep
```

context switch

Thread B

```
sleep return
enable ints
.
.
.
disable int
sleep
```

context switch

Thread A

```
sleep return
enable ints
.
.
```

NACHOS.THREAD.LOCK

```
public class Lock {
        /**
         * Allocate a new lock. The lock will initially be <i>free</i>.
         */
        public Lock() {}
        /**
         * Atomically acquire this lock. The current thread must not already hold this lock
         */
        public void acquire() {
                Lib.assertTrue(!isHeldByCurrentThread());
                boolean intStatus = Machine.interrupt().disable();
                KThread thread = KThread.currentThread();
                if (lockHolder != null) {
                        waitQueue.waitForAccess(thread);
                        KThread.sleep();
                } else {
                        waitQueue.acquire(thread);
                        lockHolder = thread;
                }
                Lib.assertTrue(lockHolder == thread);
                Machine.interrupt().restore(intStatus);
        }
```

NACHOS.THREAD.LOCK

```
/**
 * Atomically release this lock, allowing other threads to acquire it.
 */
public void release() {
        Lib.assertTrue(isHeldByCurrentThread());

        boolean intStatus = Machine.interrupt().disable();

        if ((lockHolder = waitQueue.nextThread()) != null)
                lockHolder.ready();

        Machine.interrupt().restore(intStatus);
}
```

ROADMAP

How to implement Acquire() and Release()

1. **By disabling/enabling interrupt**
   - A bad implementation
   - A better implementation
2. **Using atomic read/write**
   - A bad implementation that may busy wait a long time
   - A better implementation
3. **A more sophisticated lock – semaphore**
4. **A safer implementation – monitor and conditional variable**

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

ATOMIC READ-MODIFY-WRITE INSTRUCTIONS

Problems with interrupt-based lock solution:

- Can’t leave lock implementation to users
- Doesn’t work well on multiprocessor
    - Disabling interrupts on all processors requires messages and would be very time consuming

Alternative: atomic instruction sequences

- These instructions read a value from memory and write a new value atomically
- Hardware is responsible for implementing this correctly
- Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors

EXAMPLES OF READ-MODIFY-WRITE

```
test&set (&address) {      /* most architectures */
        result = M[address];
        M[address] = 1;
        return result;
}


swap (&address, register) { /* x86 */
        temp = M[address];
        M[address] = register;
        register = temp;
}


compare&swap (&address, reg1, reg2) { /* 68000 */
        if (reg1 == M[address]) {
                M[address] = reg2;
                return success;
        } else {
                return failure;
        }
}
```

IMPLEMENTING LOCKS WITH TEST&SET

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

Simple solution:

```
int value = 0; // Free

Acquire() {
      while (test&set(value)); // while busy
}

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

Simple explanation:

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

PROBLEM: BUSY-WAITING FOR LOCK

Positives for this solution

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

Negatives

- Inefficient: busy-waiting thread will consume cycles waiting
- Waiting thread may take cycles away from thread holding lock!
- Priority Inversion: If busy-waiting thread has higher priority than thread holding lock no progress!

Priority Inversion problem with original Martian rover

For semaphores and monitors, waiting thread may wait for an arbitrary length of time!

- Even if OK for locks, definitely not ok for other primitives
- Project/exam solutions should not have busy-waiting!

BETTER LOCKS USING TEST&SET

Can we build test&set locks without busy-waiting?

- Can’t entirely, but can minimize!
- Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
```

```
Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

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

Note: sleep has to be sure to reset the guard variable

LOCKS USING TEST&SET VS. INTERRUPTS

Compare to “disable inter  **solution**

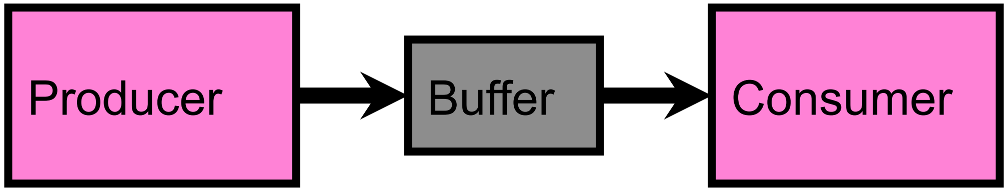
```
int value = FREE;

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

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

Basically replace

- disable interrupts → `while (test&set(guard));`
- enable interrupts → `guard = 0;`

PRODUCER-CONSUMER WITH MUTEX LOCK



Producer

Buffer

Consumer

```
void *Producer()
{
    int i, produced=0;
    for(i=0;i<100000;i++)
    {
        pthread_mutex_lock(&mVar);
        if(count < BUFFERSIZE) {
       buffer[in] = '@';
           in = (in + 1)% BUFFERSIZE;
       count++;
       produced++;
    }
        pthread_mutex_unlock(&mVar);
    }

    printf("total produced = %d\n", produced);

}
```

PRODUCER-CONSUMER WITH MUTEX LOCK

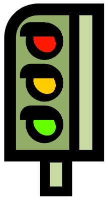
```
void *Consumer()
{
    int i, consumed = 0;
    for(i=0;i<100000;i++){
        pthread_mutex_lock(&mVar);
        if(count>0)
        {
            out = (out+1)%BUFFERSIZE;
            --count;
            printf("Consumer: count = %d\n", count);
        }
        pthread_mutex_unlock(&mVar);
    }
}
```

ROADMAP

How to implement Acquire() and Release()

1. **By disabling/enabling interrupt**
    - A bad implementation
    - A better implementation
2. **Using atomic read/write**
    - A bad implementation that may busy wait a long time
    - A better implementation
3. **A more sophisticated lock – semaphore**
4. **A safer implementation – monitor and conditional variable**

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



SEMAPHORES

Semaphores are a kind of generalized locks

- First defined by Dijkstra in late 60s
- Main synchronization primitive used in original UNIX

Definition: a Semaphore has a non-negative integer value and supports the following two operations:

- P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - Think of this as the wait() operation
- V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - This of this as the signal() operation
- Note that P() stands for “proberen” (to test) and V() stands for “verhogen” (to increment) in Dutch

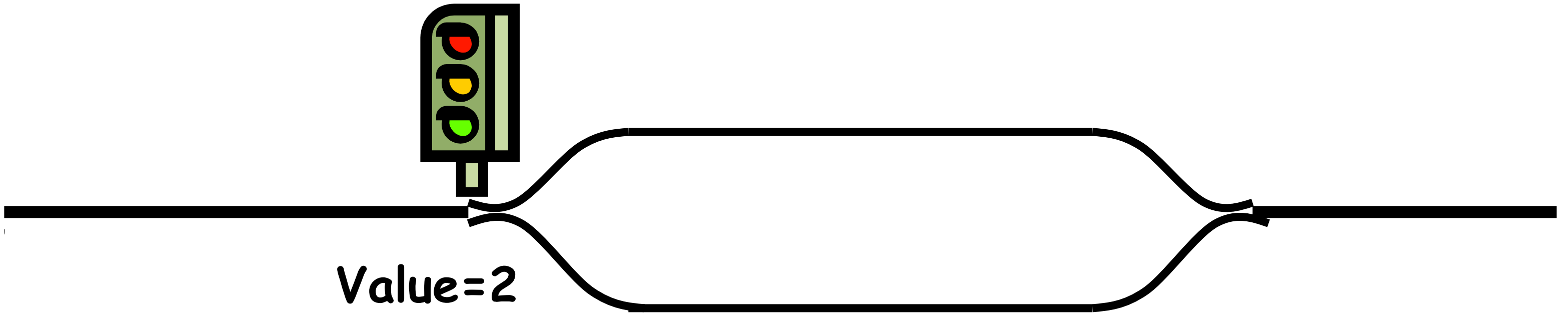
SEMAPHORES LIKE INTEGERS EXCEPT

Semaphores are like integers, except

- No negative values
- Only operations allowed are P and V – can't read or write value, except to set it initially
- Operations must be atomic
  - Two P's together can't decrement value below zero
  - Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time

Semaphore from railway analogy

- Here is a semaphore initialized to 2 for resource control:



TWO USES OF SEMAPHORES

Mutual Exclusion (initial value = 1)

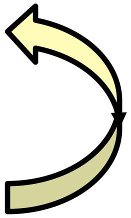
- Also called “Binary Semaphore”.
- Can be used for mutual exclusion:

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

Scheduling Constraints (initial value = 0)

- Allow thread 1 to wait for a signal from thread 2, i.e., thread 2 schedules thread 1 when a given constrained is satisfied
- Example: suppose you had to implement ThreadJoin which must wait for thread to terminiate:

```
Initial value of semaphore = 0
ThreadJoin {
   semaphore.P();
}
ThreadFinish {
   semaphore.V();
}
```



NACHOS.THREAD.SEMAPHORE

```
public class Semaphore {
          /**
           * Allocate a new semaphore.
           * @param initialValue the initial value of this semaphore.
           */
          public Semaphore(int initialValue) {
                    value = initialValue;
          }
          /**
           * Atomically wait for this semaphore to become non-zero and decrement it.
           */
          public void P() {
                    boolean intStatus = Machine.interrupt().disable();
                    if (value == 0) {
                              waitQueue.waitForAccess(KThread.currentThread());
                              KThread.sleep();
                    } else {
                              value--;
                    }
                    Machine.interrupt().restore(intStatus);
          }
```

NACHOS.THREAD.SEMAPHORE

```
public void V() {
        boolean intStatus = Machine.interrupt().disable();
        KThread thread = waitQueue.nextThread();
        if (thread != null) {
                thread.ready();
        } else {
                value++;
        }

        Machine.interrupt().restore(intStatus);
}
```

PRODUCER-CONSUMER USING SEMAPHORE

Problem Definition

- Producer puts things into a shared buffer
- Consumer takes them out
- Need synchronization to coordinate producer/consumer

Correctness Constraints:

- Consumer must wait for producer to fill slots, if empty (scheduling constraint)
- Producer must wait for consumer to make room in buffer, if all full (scheduling constraint)
- Only one thread can manipulate buffer queue at a time (mutual exclusion)

CORRECTNESS CONSTRAINTS FOR SOLUTION

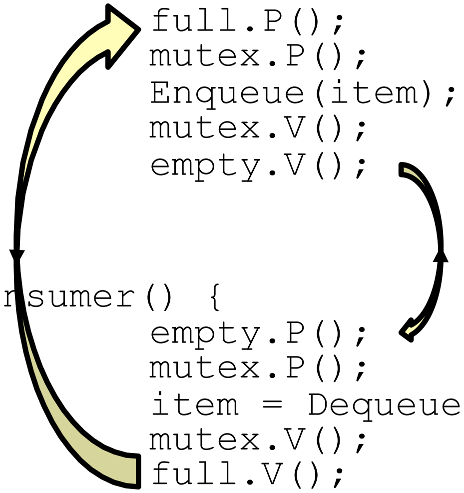
General rule of thumb: Use a separate semaphore for each constraint

- Semaphore full; // producer’s constraint
- Semaphore empty;// consumer’s constraint
- Semaphore mutex;	// mutual exclusion

Initial values?

FULL SOLUTION TO BOUNDED BUFFER

```
  Semaphore empty = 0;    // Initially, buffer empty
  Semaphore full = bufSize; // Initially, buffszeempty slots
  Semaphore mutex = 1;    // No one using machine
```



```
Producer(item) {
        full.P();           // Wait until space
        mutex.P();          // Wait until machine free
        Enqueue(item);
        mutex.V();
        empty.V();          // Tell consumers there is
                                    // more coke
}
Consumer() {
        empty.P();          // Check if there’s a coke
        mutex.P();          // Wait until machine free
        item = Dequeue();
        mutex.V();
        full.V();           // tell producer need more
        return item;
}
```

DISCUSSION ABOUT SOLUTION

Why asymmetry?

Decrease # of empty slots

Increase # of occupied slots

- Producer does: `full.P(), empty.V()`
- Consumer does: `empty.P(), full.V()`

Decrease # of occupied slots

Increase # of empty slots

DISCUSSION ABOUT SOLUTION

Is order of P's important?

- Yes! Can cause deadlock

Is order of V's important?

- No, except that it might affect scheduling efficiency

What if we have 2 producers or 2 consumers?

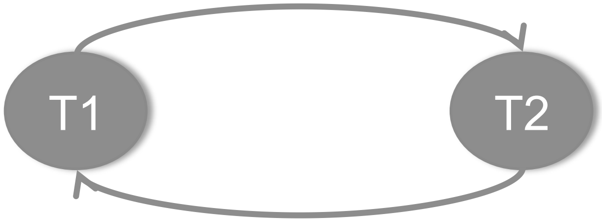
- Do we need to change anything?

```
 Producer(item) {
     mutex.P();
     full.P();
     Enqueue(item);
     mutex.V();
     empty.V();
}
 Consumer() {
     empty.P();
     mutex.P();
     item = Dequeue();
     mutex.V();
     full.V();
     return item;
 }
```

ANOTHER EXAMPLE OF DEADLOCK USING SEMAPHORE

```
Thread 1
cond1.P()
cond2.P()
…
cond2.V()
cond1.V()
```

```
Thread 2
cond2.P()
cond1.P()
…
cond1.V()
cond2.V()
```



MONITORS AND CONDITION VARIABLES

Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores

Problem is that semaphores are dual purposed:

- They are used for both mutex and scheduling constraints
- Example: the fact that flipping of P’s in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?

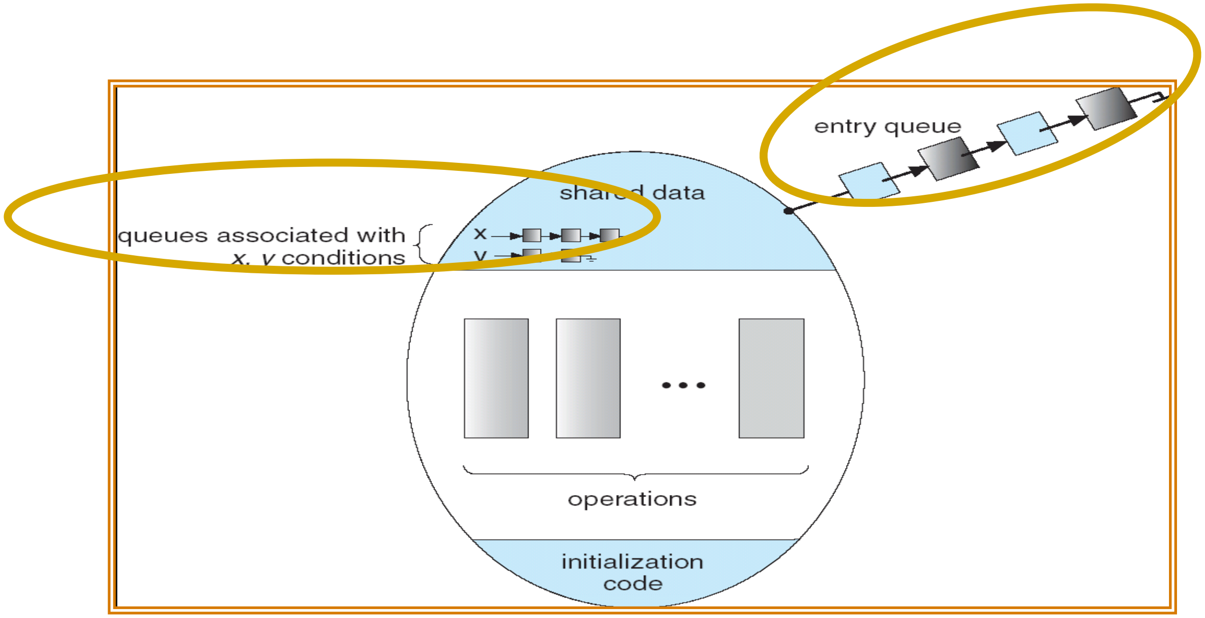
MOTIVATION FOR MONITORS AND CONDITION VARIABLES

Cleaner idea: Use locks for mutual exclusion and condition variables for scheduling constraints

Monitor: a lock and zero or more condition variables for managing concurrent access to shared data

- Some languages like Java provide this natively
- Most others use actual locks and condition variables

MONITOR WITH CONDITION VARIABLES



Lock: the lock provides mutual exclusion to shared data

- Always acquire before accessing shared data structure
- Always release after finishing with shared data
- Lock initially free

Condition Variable: a queue of threads waiting for something inside a critical section

- Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep

SIMPLE MONITOR EXAMPLE

Here is an (infinite) synchronized queue

```
Lock lock;
Queue queue;


AddToQueue(item) {
        lock.Acquire();   // Lock shared data
        queue.enqueue(item);      // Add item
        lock.Release();   // Release Lock
}


RemoveFromQueue() {
        lock.Acquire();   // Lock shared data
        item = queue.dequeue();// Get next item or null
        lock.Release();   // Release Lock
        return(item);     // Might return null
}
```

Not very interesting use of “Monitor”

- It only uses a lock with no condition variables
- Cannot put consumer to sleep if no work!

CONDITION VARIABLES

Condition Variable: a queue of threads waiting for something inside a critical section

- Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
- Contrast to semaphores: Can't wait inside critical section

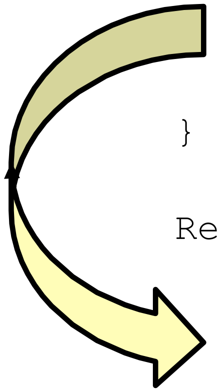
Operations:

- Wait(&lock): Atomically release lock and go to sleep. Re-acquire lock later, before returning.
- Signal(): Wake up one waiter, if any
- Broadcast(): Wake up all waiters

Rule: Must hold lock when doing condition variable operations!

COMPLETE MONITOR EXAMPLE (WITH CONDITION VARIABLE)

Here is an (infinite) synchronized queue



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


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


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

MESA VS. HOARE MONITORS

Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

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

- Why didn’t we do this?

```
if (queue.isEmpty()) {
        dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue();   // Get next item
```

Answer: depends on the type of scheduling

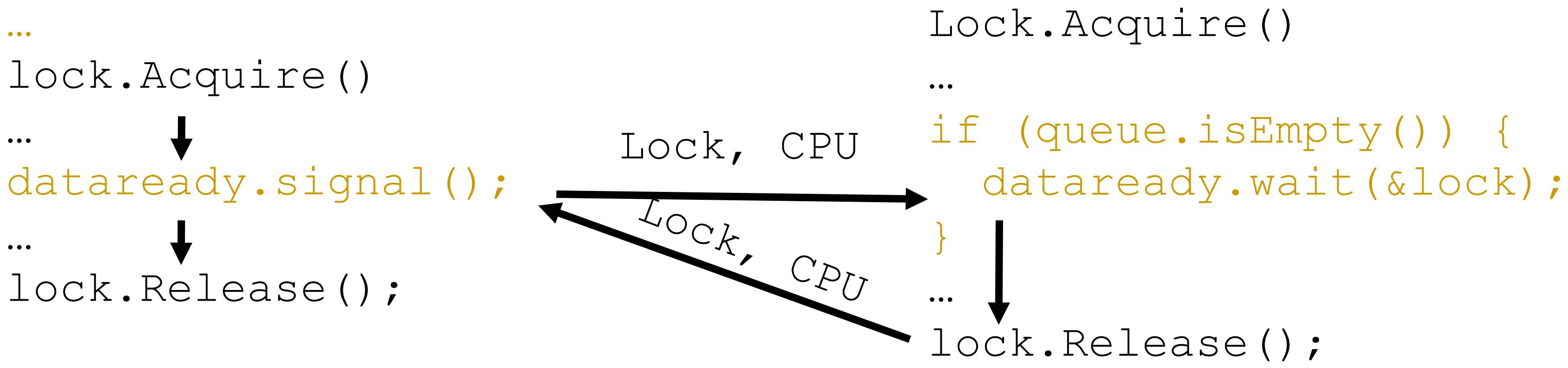
- Hoare-style
- Mesa-style

HOARE MONITORS

Signaler gives up lock, CPU to waiter; waiter runs immediately

Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again

Most textbooks



```
…
lock.Acquire()
…
dataready.signal();
…
lock.Release();
```

```
Lock.Acquire()
…
if (queue.isEmpty()) {
  dataready.wait(&lock);
}
…
lock.Release();
```

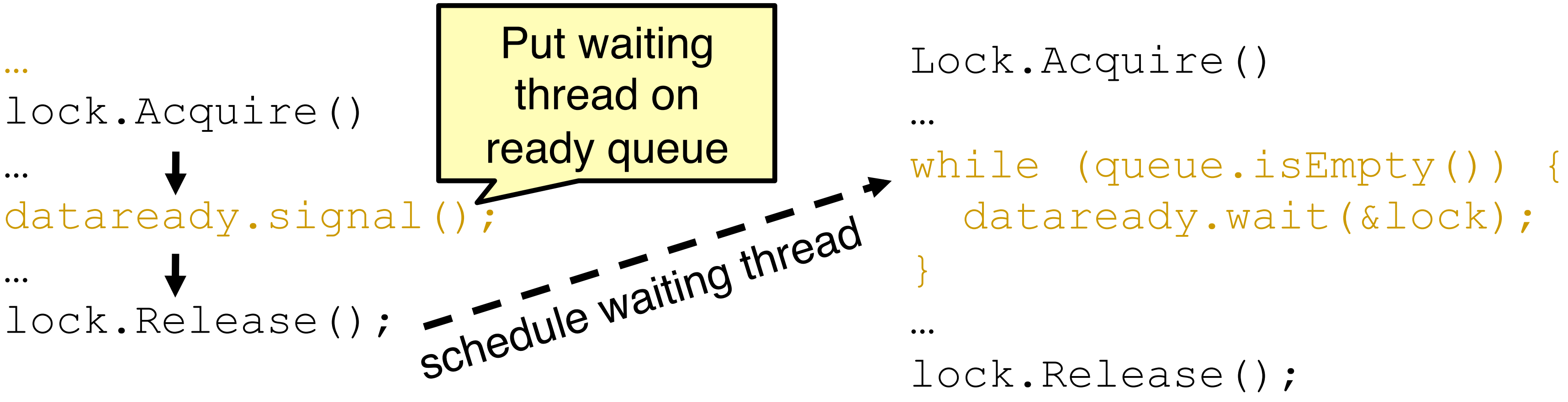
MESA MONITORS

Signaler keeps lock and processor

Waiter placed on a local “e” queue for the monitor

Practically, need to check condition again after wait

Most real operating systems (and Nachos!)



```
…
lock.Acquire()
…
dataready.signal();
…
lock.Release();
```

```
Lock.Acquire()
…
while (queue.isEmpty()) {
  dataready.wait(&lock);
}
…
lock.Release();
```

NACHOS.THREADS.CONDITION

```
public class Condition {
        /**
         * Allocate a new condition variable.
         *
         * @param conditionLock
         *            the lock associated with this condition variable. The current
         *            thread must hold this lock whenever it uses <tt>sleep()</tt>,
         *            <tt>wake()</tt>, or <tt>wakeAll()</tt>.
         */
        public Condition(Lock conditionLock) {
                this.conditionLock = conditionLock;

                waitQueue = new LinkedList<Semaphore>();
        }
```

NACHOS.THREADS.CONDITION

```
/*
sleep(): atomically release the lock and relinkquish the CPU
until woken; then reacquire the lock.*/
public void sleep() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());
        Semaphore waiter = new Semaphore(0);
        waitQueue.add(waiter);
        conditionLock.release();
        waiter.P();
        conditionLock.acquire();
}
```

NACHOS.THREADS.CONDITION

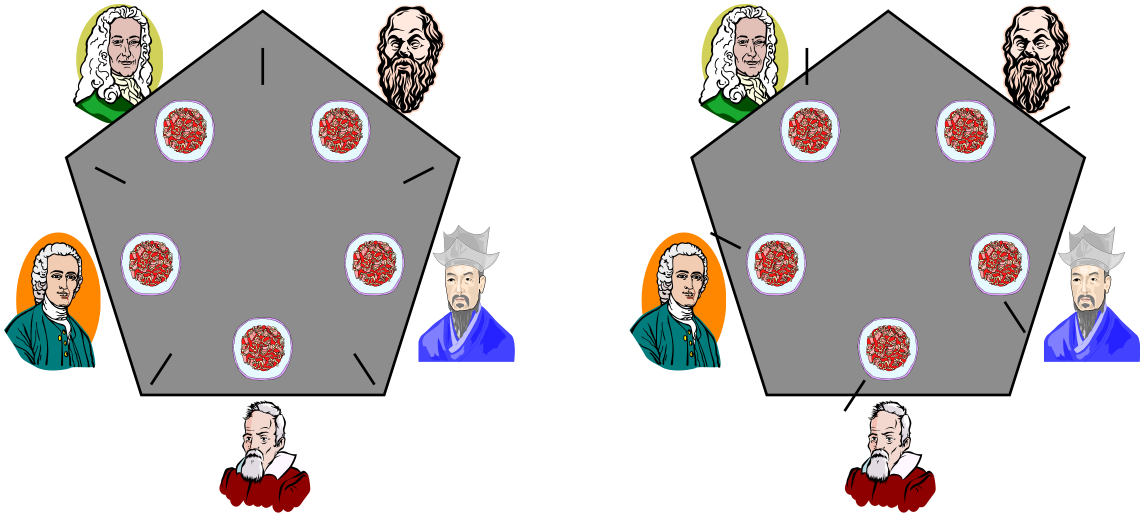
```
/**
 * Wake up at most one thread sleeping on this condition variable. The
 * current thread must hold the associated lock.
 */
public void wake() {
		Lib.assertTrue(conditionLock.isHeldByCurrentThread());

		if (!waitQueue.isEmpty())
				((Semaphore) waitQueue.removeFirst()).V();
}
public void wakeAll() {
		Lib.assertTrue(conditionLock.isHeldByCurrentThread());

		while (!waitQueue.isEmpty())
				wake();
}
```

PRODUCER-CONSUMER USING CONDITION VARIABLE

```
void *Producer()
{
    int i, produced=0;
    for(i=0;i<100000;i++) {
        pthread_mutex_lock(&mVar);
        while (count==BUFFERSIZE)
                 pthread_cond_wait(&Buffer_Not_Full,&mVar);
        buffer[count++]='@';
        pthread_cond_signal(&Buffer_Not_Empty);
        pthread_mutex_unlock(&mVar);
    }
}
```

```
void *Consumer()
{
    int i, consumed = 0;
    for(i=0;i<100000;i++){
        pthread_mutex_lock(&mVar);
        while(count==0)
               pthread_cond_wait(&Buffer_Not_Empty,&mVar);
        out = (out+1)%BUFFERSIZE;
        count--;
        pthread_cond_signal(&Buffer_Not_Full);
        pthread_mutex_unlock(&mVar);
    }
}
```

DINNING PHILOSOPHER



Correctness condition:

- **mutual exclusion: no more than one person can have access to one chopstick**
- **progress: no deadlock**
- **no starvation**

Note that philosophers alternate between eating & thinking

USING SEMAPHORE

```
semaphore chopstick[5];

do {
         wait(chopstick[i]);
         wait(chopstick[(i+1) % 5]);
...
/* eat for awhile */
...
         signal(chopstick[i]);
         signal(chopstick[(i+1) % 5]);
...
/* think for awhile */
...
} while (true);
```

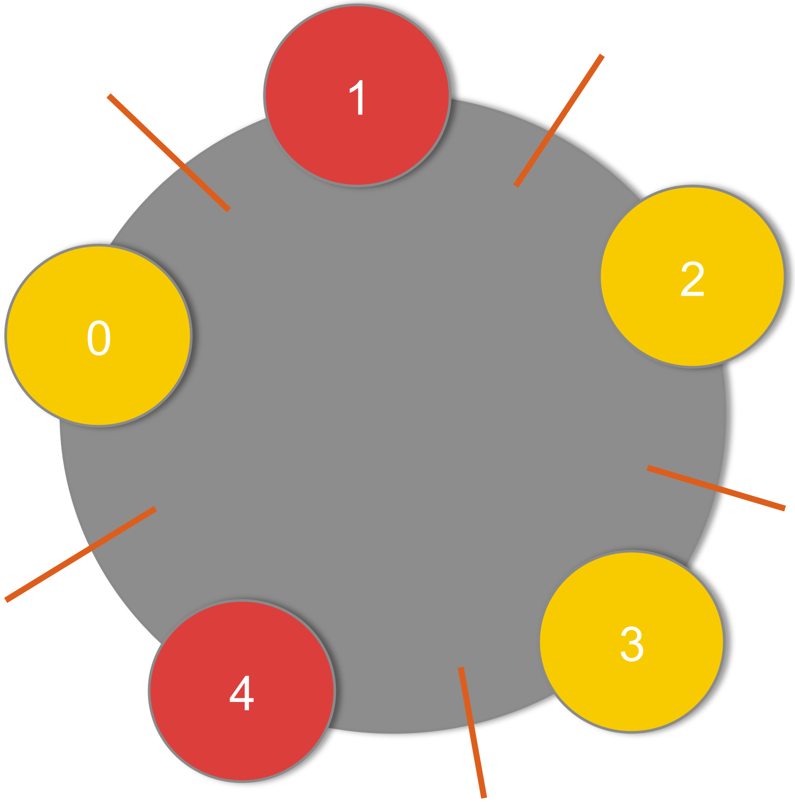
USING MONITOR

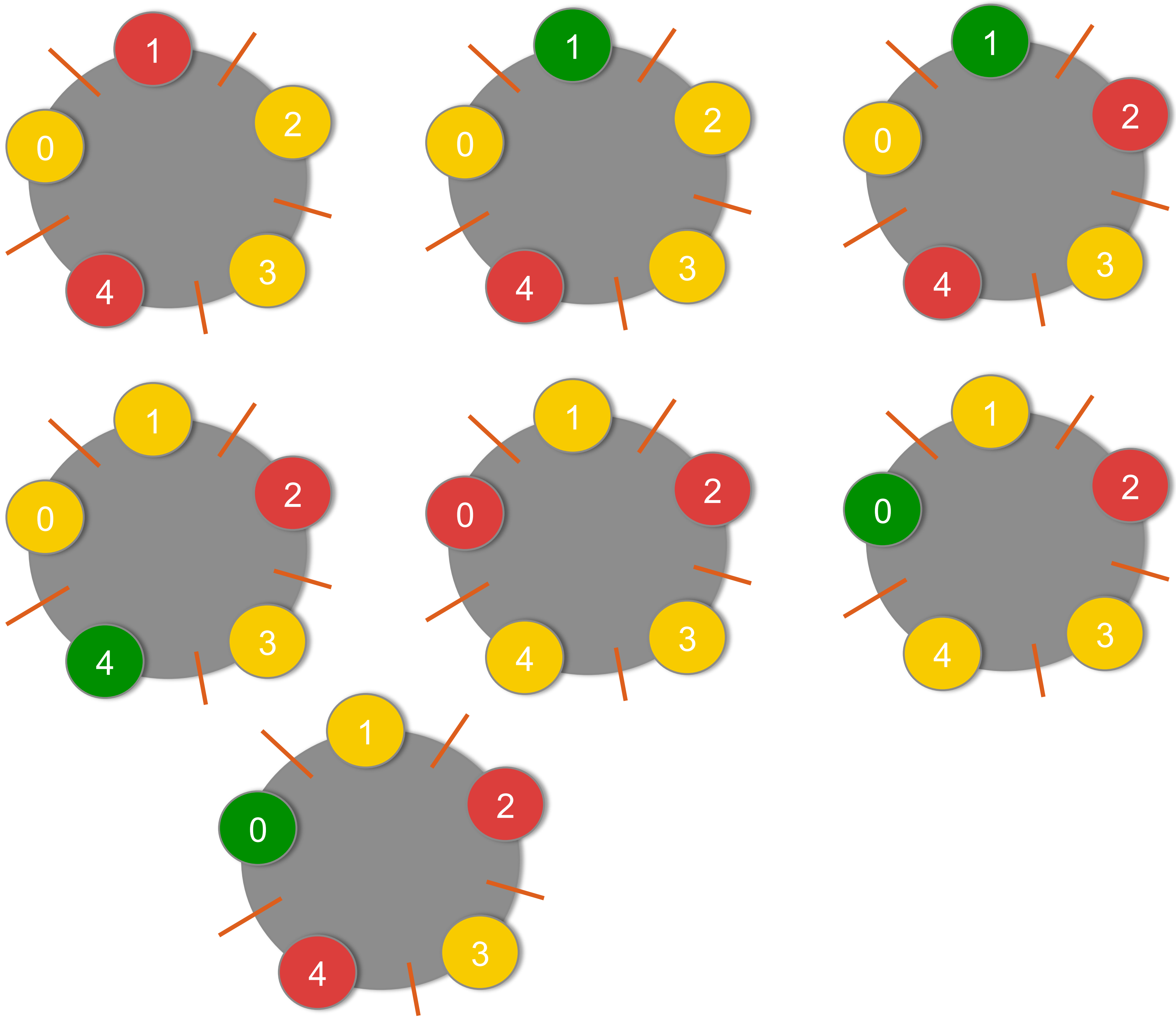
One philosopher picks two chopsticks only when both of them are available

```
monitor DiningPhilosophers {
 enum {THINKING, HUNGRY, EATING} state[5];
 condition self[5];
 void pickup(int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING)
                  self[i].wait();
}
void putdown(int i) {
         state[i] = THINKING;
         test((i + 4) % 5);
         test((i + 1) % 5);
}
```

```
void test(int i) {
  if ((state[(i + 4) % 5] != EATING) &&  (state[i]
== HUNGRY) && (state[(i + 1) % 5] != EATING)) {

      state[i] = EATING;

      self[i].wake();

  }

}

initialization code() {
  for (int i = 0; i < 5; i++)

      state[i] = THINKING;

  }

}
```

CORRECT?





1
2
0
3
4
1
2
0
3
4
1
2
0
3
4
1
2
0
3
4
1
2
0
3
4
1
2
0
3
4
1
2
0
3
4

COMPARISON

- **Lock, semaphore, monitor can all be used for achieving mutual exclusion of critical section**
- **Semaphore and condition variables useful for scheduling/ synchronization among multiple processes**
  - If implemented using Lock will have to use BUSY WAIT
  - Semaphore is good for multiple resources

SUMMARY

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