W9: PROCESS AND THREAD SYNCHRONIZATION

CSCE-313 Spring 2017

Motivation

- Concurrent processes improve Computer System resource utilization
  - But concurrency introduces inherent cost of context switching
    - Threading a process reduces the cost of context switching because we allow threads to share global context (memory, IO State) of their parent process
      - But this sharing can be dangerous if not handled properly

Synchronization Motivation

Thread 1

```
p = someFn();
Initialized = true;
```

Thread 2

```
while (! Initialized ) ;
q = aFn(p);

if q != aFn(someFn())
    panic
```

Goals for This Lecture

- Concurrency examples and sharing
- Synchronization
- Hardware Support for Synchronization

Note: Some slides and/or pictures in the following are adapted and/or used verbatim from slide content in Silberschatz, Galvin, and Gagne (2014), Anthony D. Joseph (2014 Berkeley), Tom Anderson (2014 UW), Bettati (2014 TAMU), Gu (2014 TAMU)

Correctness Requirements

- **Threaded programs must work for all interleavings of thread instruction sequences**
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- **Example: Therac-25**
  - Machine for radiation therapy
    - Software control of electron accelerator and electron beam/ Xray production
    - Software control of dosage
    - Therac-20 used to accomplish this in Hardware
  - Software errors caused overdoses and the death of several patients
    - A series of race conditions on shared variables and poor software design
    - "They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred."

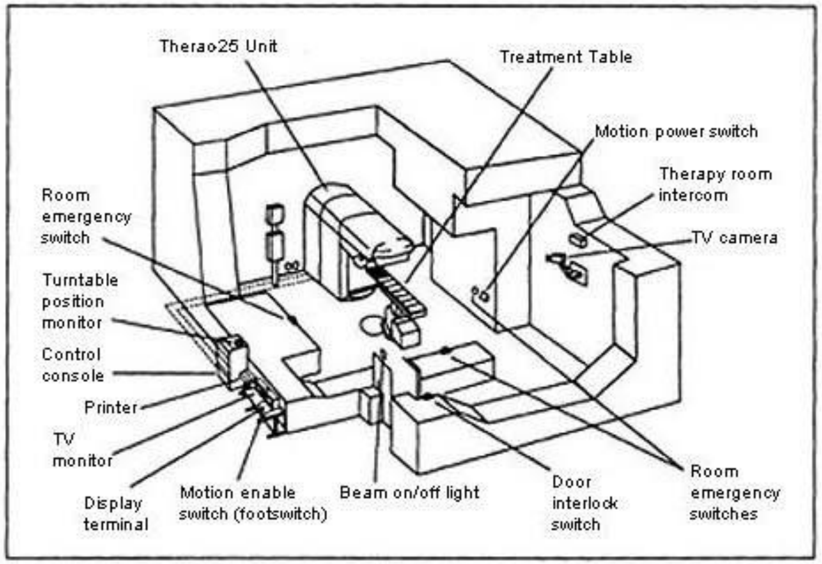
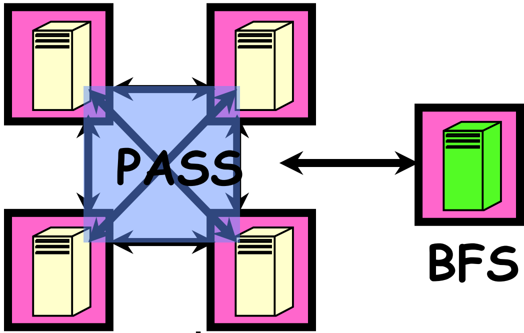


Figure 1. Typical Therac-25 facility

Space Shuttle Example

- Original Space Shuttle launch aborted 20 minutes before launch
- Shuttle has five computers:
  - Four run the "Primary Avionics Software System" (PASS)
    - Asynchronous and real-time
    - Runs all of the control systems
    - Results synchronized and compared 440 times per second
  - The Fifth computer is the "Backup Flight System" (BFS)
    - Stays synchronized in case it is needed
    - Written by completely different team than PASS



- Countdown aborted because BFS disagreed with PASS
  - Bug due to modifications in initialization code of PASS
    - A delayed init request placed into timer queue
    - As a result, timer queue not empty at expected time to force use of hardware clock
  - Bug not found during extensive simulation

Race Condition

- **Race condition:** Output of a concurrent program depends on the order of operations between threads
- Sequential Model of thinking does not work for concurrent threads
  - Cannot make any assumptions about relative speed at which the threads operate (i.e. interleaving is a given)
  - Program execution can be non-deterministic (scheduler, processor frequencies, etc.)
  - Compilers can reorder instructions
    - Out-of-order execution relies on compiler optimizations to circumvent operand dependencies

Race Condition – Compiler Effect

- Simple threaded code (assume x=0)

Thread1	Thread2
x=x+1;	x=x+2;

Compiler Generated:

```
load r1, x
add r2, r1, 1
store x, r2
```

Values of x can be 1, 2, or 3 depending on the order of execution

```
load r1, x
add r2, r1, 1
store x, r2
                load r1, x
                add r2, r1, 2
                store x, r2
```

X=3

```
                load r1, x
load r1, x
                add r2, r1, 2
                store x, r2
add r2, r1, 1
store x, r2
```

X=1

Concurrency Challenges

- Multiple computations (threads) executing concurrently to
  - share resources, and/or
  - share data
- Fine grain sharing:
  - ⇑ Increase concurrency → better perf.
  - ⇓ more complex
- Coarse grain sharing:
  - ⇑ Simpler to implement
  - ⇓ Lower performance

- **Cannot make any assumptions about relative speed at which the threads operate**
- **Program execution can be non-deterministic**
- **Compilers can reorder instructions**

Atomic Operations

- To understand a concurrent program, we need to know what the underlying atomic operations are!
- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic

Concurrency Coordination Landscape

Concurrent Applications

Shared Coordinated Objects

Flag
Bounded Queue
Ordered List
Dictionary
Barrier

Synchronization Variables

Locks
Condition Variables
Semaphore
Monitors
Send/Receive

Atomic Operations

Interrupt Disable/Enable
Test-and-Set

Hardware

Interrupts
Controllers
Multiple Processors
xchng
cmp&swap
fetch&inc
LL + SC

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
- Example: People need to coordinate:

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

Definitions

- **Synchronization**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic

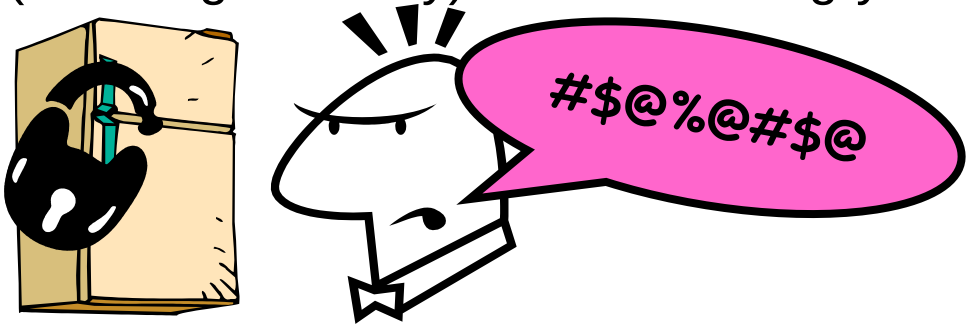
- **Critical Section**: piece of code that only one thread can execute at once

- **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

More Definitions

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

- 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 (safety)
    - **i.e. the program never enters a bad state**
  - Someone buys if needed (liveness)
    - **i.e. the program eventually achieves a good state**

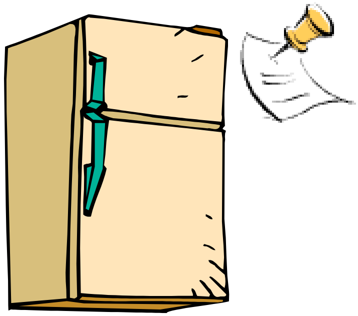
- Restrict ourselves to use only atomic load and store operations as building blocks

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;
                                   remove note; }}
```

- 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!

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

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

Thread B

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

- Does this work?

Too Much Milk Solution #2

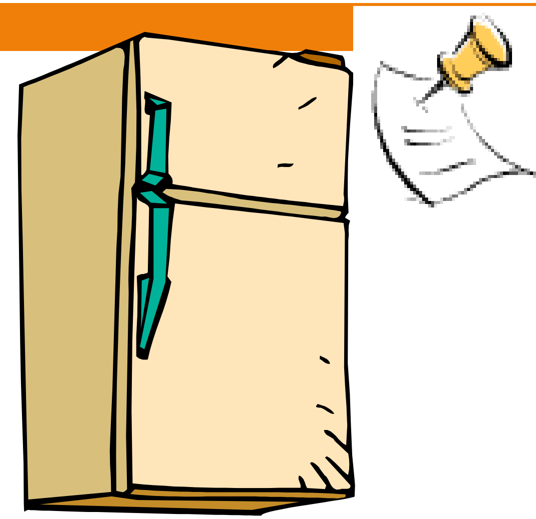
- Possible for neither thread to buy milk!

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* getting milk
- This kind of lockup is called "starvation!"

Too Much Milk Solution #3

- Here is a possible two-note solution:

<u>Thread A</u>

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

<u>Thread B</u>

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

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At Y:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Solution #3 discussion

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

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

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is called "busy-waiting"
- There's a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

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

Concurrency Coordination Landscape

Concurrent Applications

Shared Coordinated Objects

Flag | Bounded Queue | Ordered List | Dictionary | Barrier

Synchronization Variables

Locks | Condition Variables | Semaphore | Monitors | Send/Receive

Atomic Operations

Interrupt Disable/Enable | Test-and-Set

Hardware

Interrupts | Controllers | Multiple Processors | xchng | cmp&swap | fetch&inc | LL + SC

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, 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

- Hardware lock instructions
  - Is this a good idea?
  - What about putting a task to sleep?
    - How to handle interface between hardware and scheduler?
  - Complexity?
    - Each feature makes hardware more complex and slower

Where are we going with synchronization?

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

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

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:

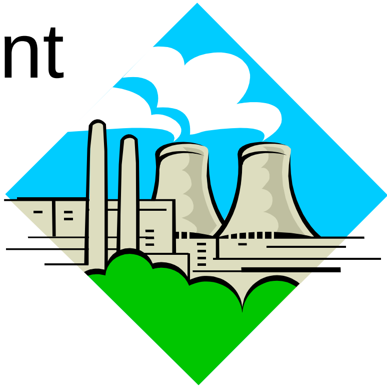
```
LockAcquire { disable Ints; }
LockRelease { enable Ints; }
```

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
- What happens with I/O or other important events?
  - "Reactor about to meltdown. Help?"



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 on the 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;
}
```

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

- Before putting thread on the wait queue?
  - Release can check the queue and not wake up thread until next lock acquire/release
- 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
  - Misses wakeup and still holds lock (deadlock!)
- 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

<u>Thread A</u>		<u>Thread B</u>
.		
.		
disable ints		
sleep	context switch →	
		yield return
		enable int
		.
		.
		.
		disable int
	← context switch	yield
sleep return		
enable ints		
.		
.		

Summary

- Introduced important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives

- Showed construction of Locks using interrupts
  - Using careful disabling of interrupts
  - Must be very careful not to waste/tie up machine resources
    - Shouldn’t disable interrupts for long
  - Key ideas: Use a separate lock variable, and use hardware mechanisms to protect modifications of that variable

More HW Assisted Solutions

Goals

- Atomic instruction sequence
  - Hardware assisted solutions

- Continue with Synchronization Abstractions
  - Semaphores (possibly, Monitors and condition variables)

Atomic Read-Modify-Write instructions

- Problems with interrupt-based lock solution:
  - Can't 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: atomic instruction sequences
  - These instructions read a value from memory and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - on both uniprocessors (not too hard)
    - and multiprocessors (requires help from cache coherence protocol)
  - 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;
  }
  ```

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;
}
```

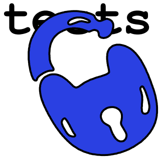
- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock

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 $\Rightarrow$ no progress!
- Priority Inversion problem with original Martian rover

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; //protects lock value
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 interrupt" 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;`

Locks using test&set vs. Interrupts

- Compare to "disable interrupt" solution

```
int value = FREE;
Acquire() {
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // guard = 0;
   } else {
      value = BUSY;
   }
   guard = 0;}
```

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

- Basically replace
  - **disable interrupts → while (test&set(guard));**
  - **enable interrupts → guard = 0;**

Recap: Locks


```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
}
```

```
Release() {
  enable interrupts;
}
```

If one thread in critical section, no other activity (including OS) can run!

```
int value = 0;
Acquire() {
  // Short busy-wait time
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}
```

```
Release() {
  // Short busy-wait time
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Recap: Locks


```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

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

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

Threads waiting to enter critical section busy-wait

```
int guard = 0;
int value = 0;
Acquire() {
  // Short busy-wait time
  while(test&set(guard));
  if (value == 1) {
    put thread on wait-queue;
    go to sleep()& guard = 0;
  } else {
    value = 1;
    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 = 0;
  }
  guard = 0;
}
```

Where are we going with synchronization?

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

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

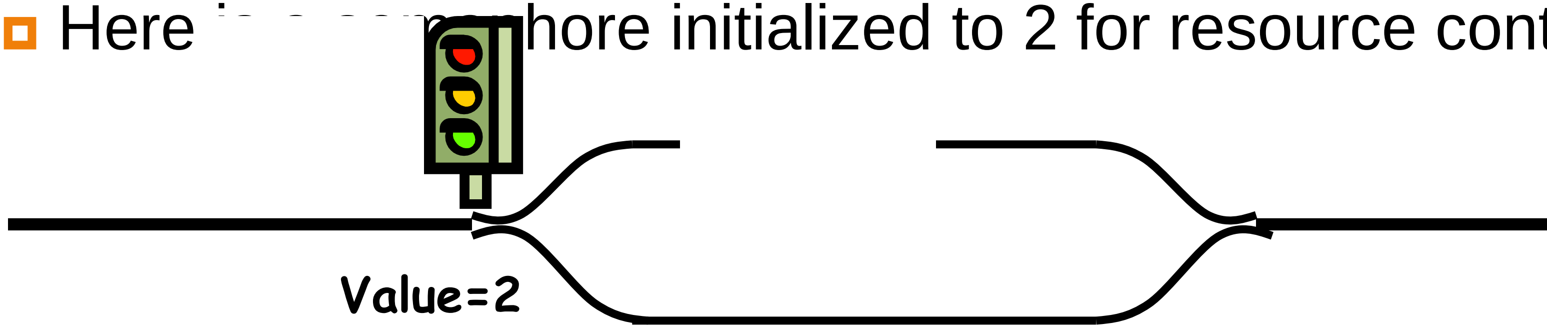
Semaphores

- 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
    - Think of this as the signal() operation

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

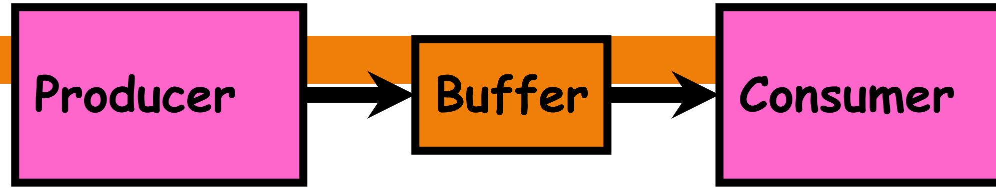
- Mutual Exclusion (initial value = 1)
  - Also called "Binary Semaphore".
  - Can be used for mutual exclusion:

```
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 terminate:

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

Producer-consumer with a bounded buffer



- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

- Example: Coke machine
  - Producer can put limited number of cokes in machine
  - Consumer can’t take cokes out if machine is empty

Correctness constraints for solution

- 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)

- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullSlots; // consumer's constraint`
  - `Semaphore emptySlots;// producer's constraint`
  - `Semaphore mutex;        // mutual exclusion`

Full Solution to Bounded Buffer


```
Semaphore fullSlots = 0;  // Initially, no coke
Semaphore emptySlots = bufSize;
                               //Initially,num empty slots
Semaphore mutex = 1;           // No one using machine

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

Discussion about Solution

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

Decrease # of empty slots

Increase # of occupied slots

Decrease # of occupied slots

Increase # of empty slots

One is creating space, the other is filling space

Discussion about Solution

- Is order of P's important?
- Is order of V's important?
- What if we have 2 producers or 2 consumers?

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

Discussion about Solution

- Is order of P's important?
  - Yes! Can cause deadlock

BEFORE

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

AFTER

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

Discussion about Solution

- Is order of V's important?
  - No, except that it might affect scheduling efficiency

BEFORE

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

AFTER

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

Discussion about Solution

- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?
    - NO

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

Summary

- Threading is great for performance
- Threading is tricky and carry hazards!