Lecturer: Dr. Michael J. May
Kinneret College

Cooperating Threads, Synchronization Mutual Exclusion, Semaphores

19 December 2024
Lecture 7

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

Thread lifecycle	Thread join	Kernel supported threads
User supported threads	Scheduler activation	yield()
switch()	Cooperating threads	

Topics for Today

- Concurrency challenge
- Motivation for Synchronization and Locks
- Atomic Read-Modify-Write Operations
- Higher Level Synchronization Atoms
  - Semaphores
  - Monitors

Concepts for today

ATOMic Operations

- To understand a concurrent program, we need to know what the underlying **indivisible 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

(1966-1973)

"As always, should you or any of your IM force be caught or killed, the Secretary will disavow any knowledge of your actions.

"Good luck, Jim. This tape will self-destruct in five seconds."

ATOMic Operations

- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
  - Consequently – weird example that produces “3” on previous slide can’t happen

- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

Correctness Requirements

Threaded programs must work for all interleavings of thread instruction sequences

Cooperating threads inherently non-deterministic and non-reproducible

Image source: http://knowyourmeme.com/photos/909991-futurama

Really hard to debug unless carefully designed!

Example: Therac-25

- Machine for radiation therapy
  - Software control of electron accelerator and electron beam/
    X-Ray production
  - Software control of dosage

- Software errors caused 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.”

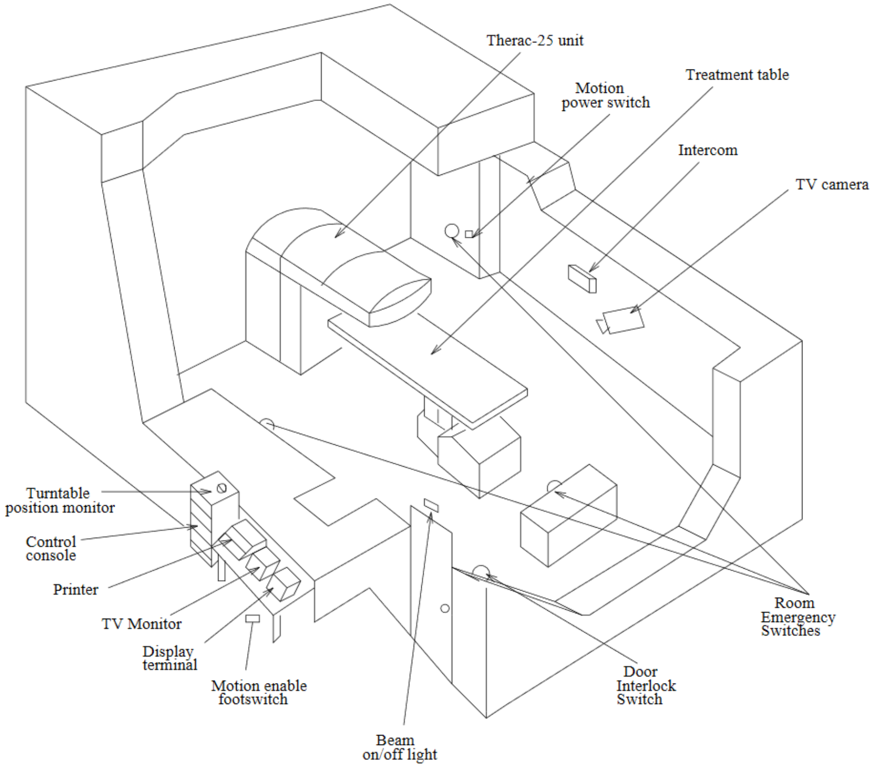


Figure 5: A typical Therac-25 facility after the final CAP.

Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

Thread A

```
i = 0;
while (i < 10)
   i = i + 1;
printf("A wins!");
```

Thread B

```
i = 0;
while (i > -10)
   i = i - 1;
printf("B wins!");
```

- Assume that memory loads and stores are atomic, but incrementing and decrementing are *not* atomic
- Who wins? Could be either
- Is it **guaranteed** that someone wins? Why or why not?
- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

Hand Simulation Multiprocessor Example

- Inner loop looks like this:

Thread A		Thread B	
r1=0	`load r1, M[i]`		
		r1=0	`load r1, M[i]`
r1=1	`add  r1, r1, 1`		
		r1=-1	`sub r1, r1, 1`
M[i]=1	`store r1, M[i]`		
		M[i]=-1	`store r1, M[i]`

- Hand Simulation:
  - And we’re off. A gets off to an early start
  - B says “hmph, better go fast” and tries really hard
  - A goes ahead and writes “1”
  - B goes and writes “-1”
  - A says “HUH??? I could have sworn I put a 1 there”

- Could this happen on a uniprocessor?
  - Yes! Unlikely, but if you are depending on it not happening, it will and your system will break…

So Far

- Concurrency challenge
- Motivation for Synchronization and Locks
- Atomic Read-Modify-Write Operations
- Higher Level Synchronization Atoms
  - Semaphores
  - Monitors

Motivation: “Too much humus”

- 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	Alice	Bob
3:00	Look in Fridge. Out of humus.	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of humus.
3:15	Buy humus	Leave for store
3:20	**Arrive at home, put humus away**	Arrive at store
3:25		Buy humus
3:30		**Arrive at home, put humus away**

Definitions

Synchronization: using atomic operations to ensure cooperation between threads

- For now, only loads and stores are atomic
- We are going to show that its hard to build anything useful with only reads and writes

Mutual Exclusion: ensuring that only one thread does a particular thing at a time

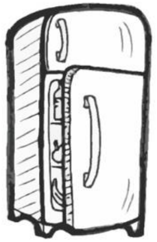
- One thread *excludes* the other while doing its task

Critical Section: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.

- Critical section is the result of mutual exclusion
- 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
- For example: fix the humus problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy humus
  - Fixes too much: roommate angry if only wants OJ



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

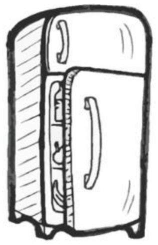
Too Much Humus: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Always write down 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 humus" 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 Humus: Solution #1

- Use a note to avoid buying too much humus:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if there’s a note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noHumus) {
   if (noNote) {
      leave Note;
      buy humus;
      remove note;
   }
}
```



- Result?
  - Still too much humus but only occasionally!
  - Thread can get context switched after checking humus and note but before buying humus!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

Too Much Humus: 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 (noHumus) {
   if (noNote) {
      leave Note;
      buy humus;
   }
}
remove note;
```

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

Too Much Humus Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

<u>Thread A</u>

```
leave note A;
if (noNote B) {
   if (noHumus) {
      buy Humus;
   }
}
remove note A;
```

<u>Thread B</u>

```
leave note B;
if (noNote A) {
   if (noHumus) {
      buy Humus;
   }
}
remove note B;
```

Too Much Humus Solution #2

Does this work?

Possible for neither thread to buy humus

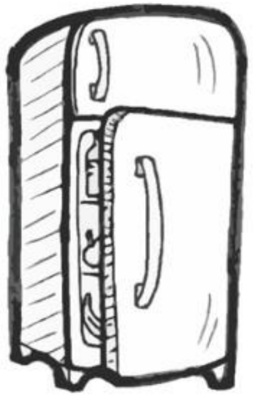
- Context switches at exactly the wrong time can lead each to think that the other is going to buy

Really insidious:

- Extremely unlikely that this would happen, but will at worst possible time
- Probably something like this in UNIX

Too Much Humus Solution #2 Problem

- *I'm* not getting humus, *You're* getting humus
- This kind of lockup is called "starvation!"



Too Much Humus Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) { //X
   do nothing;
}
if (noHumus) {
   buy Humus;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) { //Y
     if (noHumus) {
          buy Humus;
     }
}
remove note B;
```

Too Much Humus Solution #3

- 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 (noHumus) {
   buy humus;
}
```

- Solution #3 works, but it’s 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 you have many threads?
  - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is “busy-waiting”

Solution #3 discussion

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

Too Much Humus: Solution #4

- Let’s make an implementation of a lock (more later).
  - `Lock.Acquire()` – wait until lock is free, then grab
  - `Lock.Release()` – Unlock, waking up anyone waiting
  - Must be atomic operations – if two threads are waiting for the lock and both see it’s free, only one succeeds in grabbing the lock
- Then, our humus problem is easy:

```
humuslock.Acquire();
if (noHumus)
   buy humus;
humuslock.Release();
```

- Section of code between `Acquire()` and `Release()` is a “Critical Section”
- You can make this even simpler: suppose you are out of ice cream instead of humus
  - Skip the test since you always need more ice cream.

Where are we going with synchronization?

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

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

So Far

- Concurrency challenge
- Motivation for Synchronization and Locks
- Atomic Read-Modify-Write Operations
- Higher Level Synchronization Atoms
  - Semaphores
  - Monitors

Atomic Read-Modify-Write instructions

- Problems with interrupts only based solution:
  1. Can’t give lock implementation to users
  2. Doesn’t work well on multiprocessor
     - Disabling interrupts on all processors requires messages and would be very time consuming

- Alternative: Atomic Instruction Sequences

Instructions that read a value from memory and write a new value atomically

Hardware is responsible for implementing this correctly

- On uniprocessors (not too hard)
- On 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;
}
```

Examples of Read-Modify-Write

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

Examples of Read-Modify-Write

```
load-linked&store conditional(&address) {
      /* R4000, alpha */
    loop:
         ll r1, M[address];
         movi r2, 1;          /* Can do arbitrary comp */
         sc r2, M[address];
         beqz r2, loop;
}
```

Implementing Locks with test&set

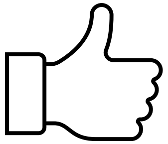
- Another flawed, but simple solution:

```
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
- Busy-Waiting: thread consumes cycles while waiting

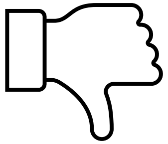
Problem: Busy-Waiting for Lock

Positives



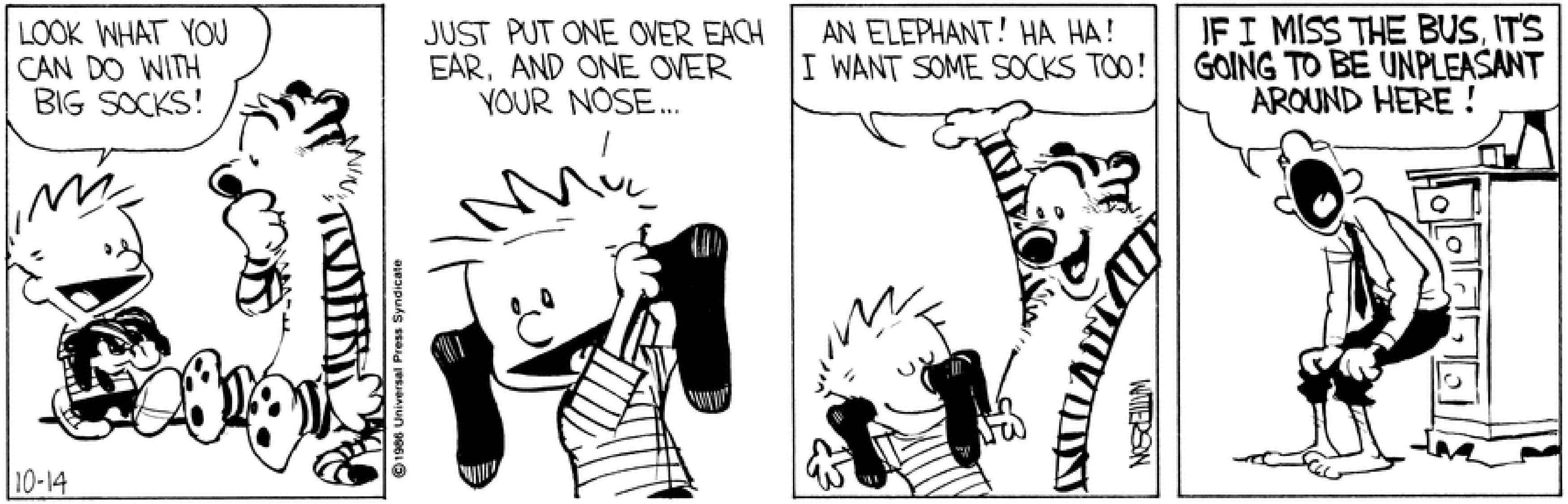
- Machine can receive interrupts
- User code can use the lock
- Works on a multiprocessor

Negatives



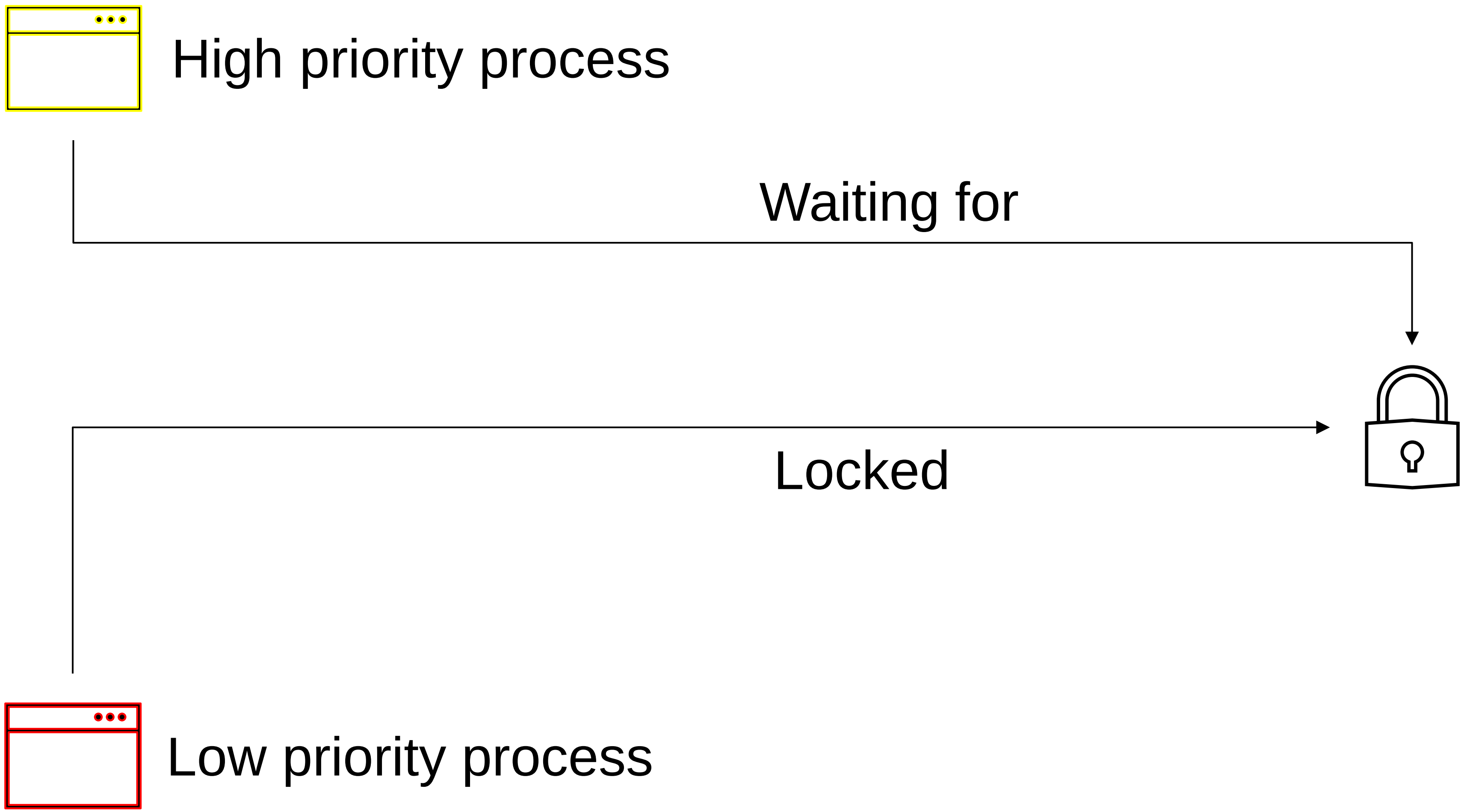
- Very inefficient because the busy-waiting thread consumes cycles waiting
- Waiting thread may take cycles away from thread holding lock (no one wins!)
- Priority Inversion: If busy-waiting thread has higher priority than thread holding lock $\Rightarrow$ no progress!
  - Priority Inversion problem with original Martian rover

Priority inversion

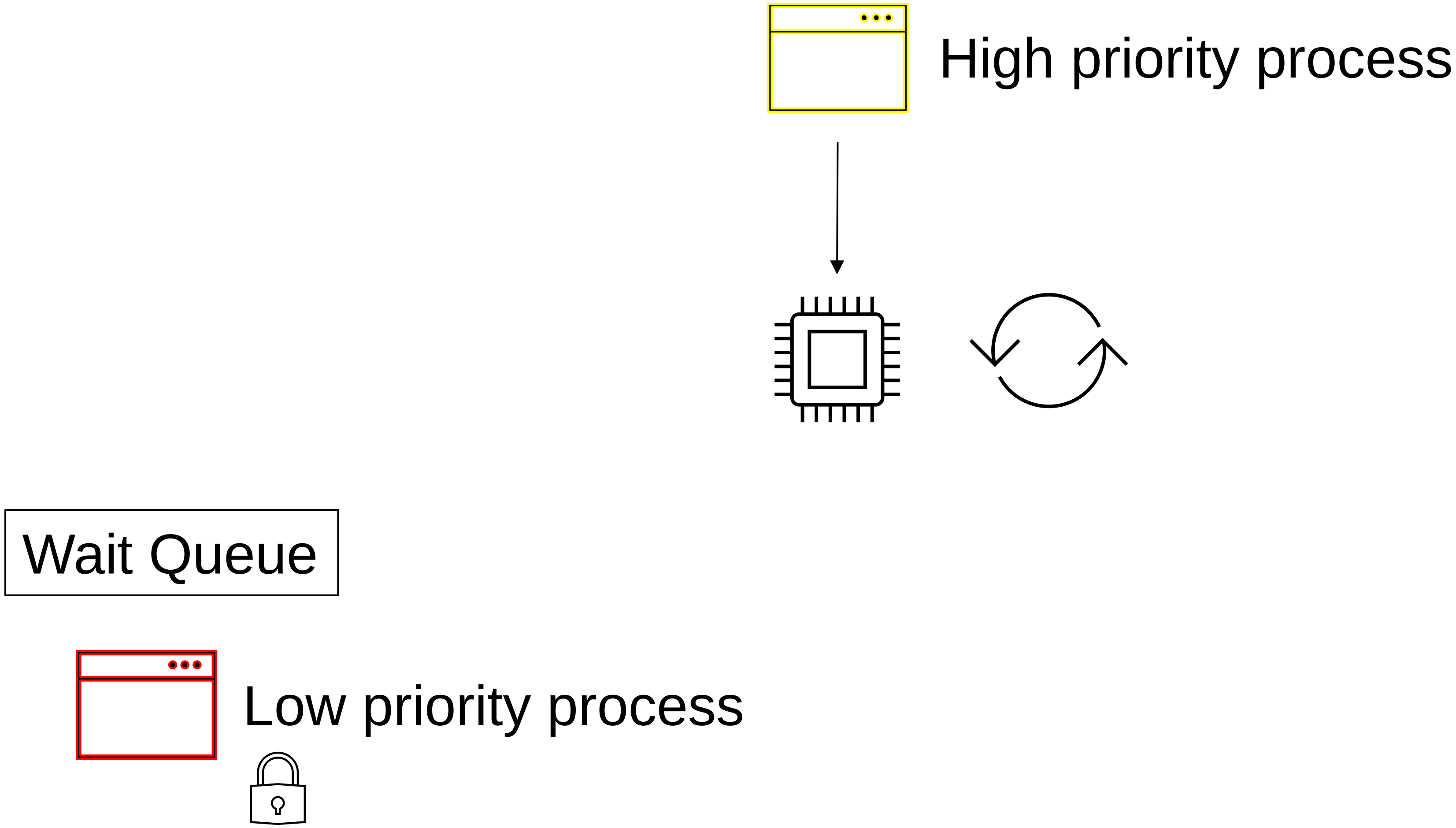


Calvin and Hobbes by Bill Watterson for October 14, 1986
https://www.gocomics.com/calvinandhobbes/1986/10/14

Priority Inversion



Priority Inversion



Problem: Busy-Waiting for Lock

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

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

Multiprocessor Spin Locks: test&test&set

- A better solution for multiprocessors:

```
int mylock = 0; // Free
Acquire() {
   do {
      while(mylock);   // Wait until might be free
   } while(test&set(&mylock)); // exit if get lock
}

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

- Simple explanation:
  - Wait until lock might be free (only reading – stays in cache)
  - Then, try to grab lock with test&set
  - Repeat if fail to actually get lock
- Issues with this solution:
  - Busy-Waiting: thread still consumes cycles while waiting
  - However, it does not impact other processors!

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
  - Why can't we do it just before or just after the `sleep`?

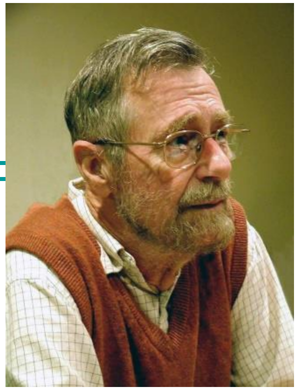
So Far

- Concurrency challenge
- Motivation for Synchronization and Locks
- Atomic Read-Modify-Write Operations
- Higher Level Synchronization Atoms
  - Semaphores
  - Monitors

Higher-level Primitives than Locks

- Goal so far:
  - What is the right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is stable now, but up until mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - We need paradigms to structure the sharing

Semaphores



- Semaphores are a kind of generalized lock
  - 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
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch

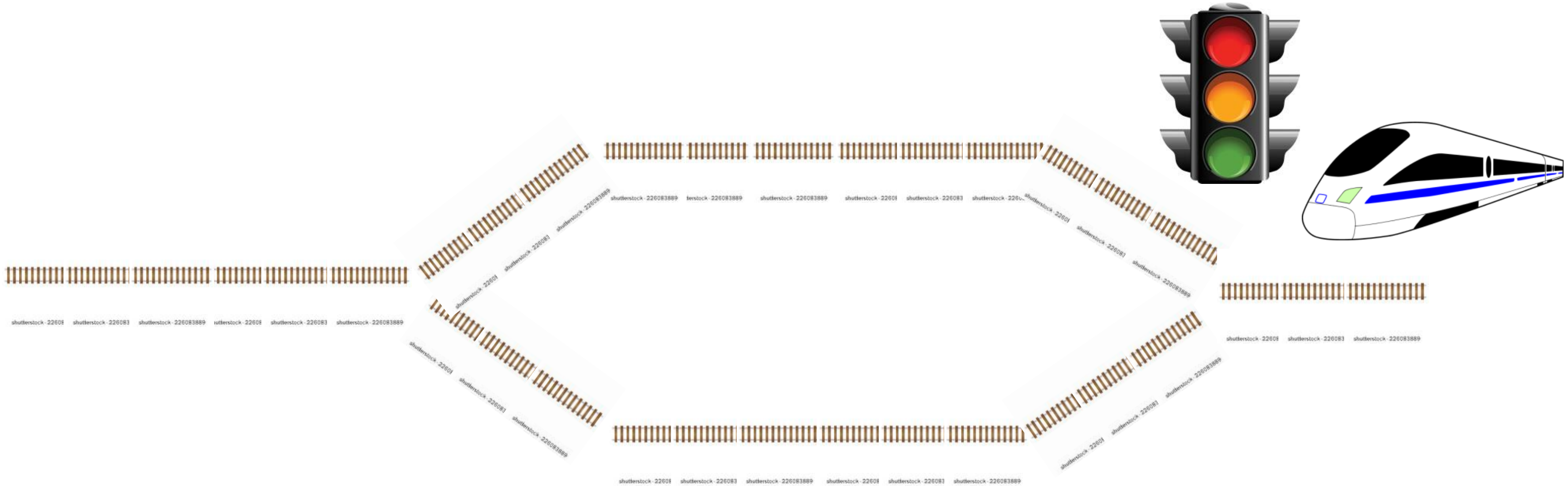
https://www.youtube.com/watch?v=LKQpy107yUY

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:

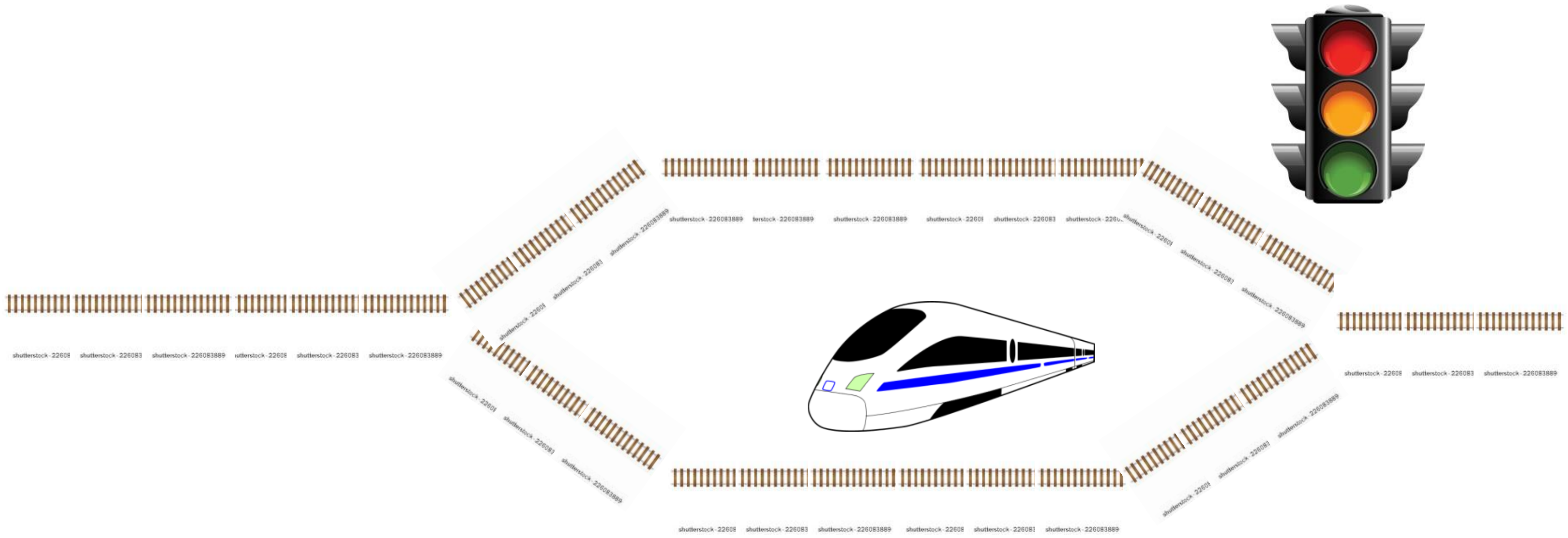
Railroad Analogy Step 1

Value=2



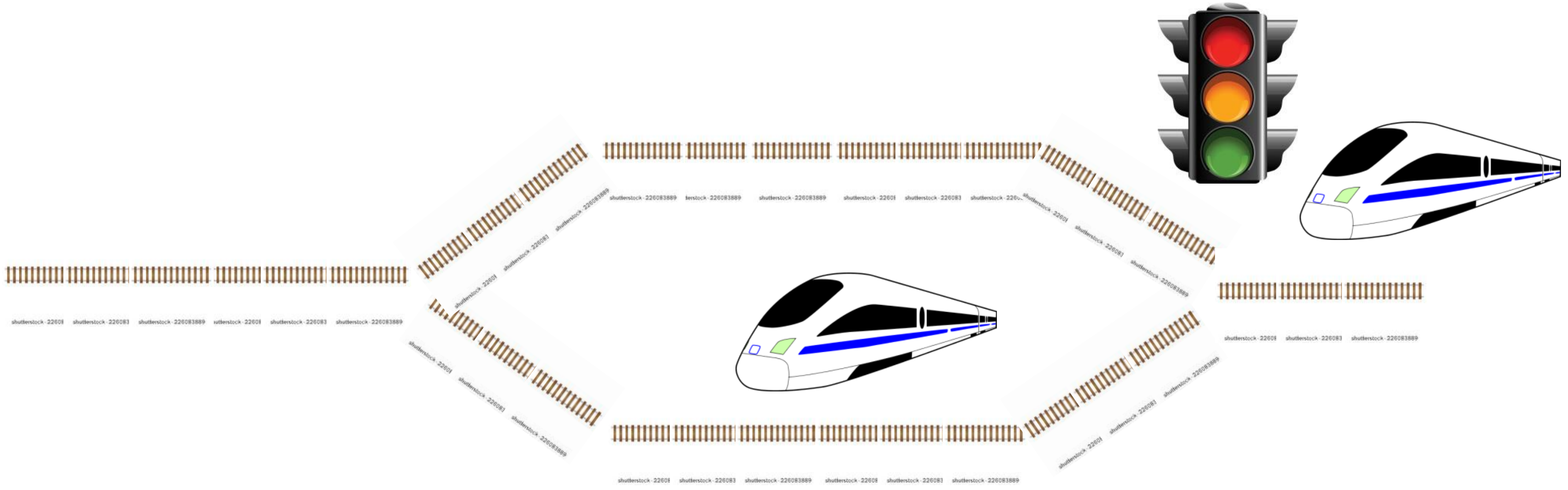
Railroad Analogy Step 2

Value=1

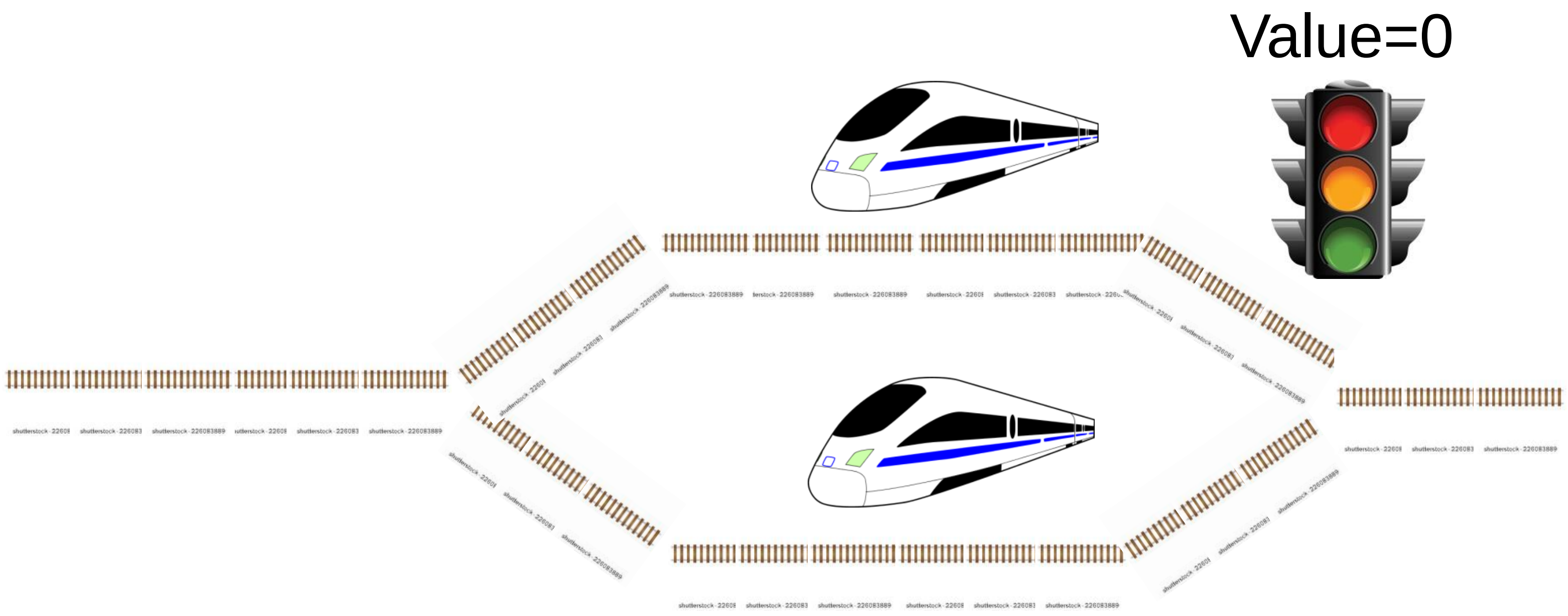


Railroad Analogy Step 3

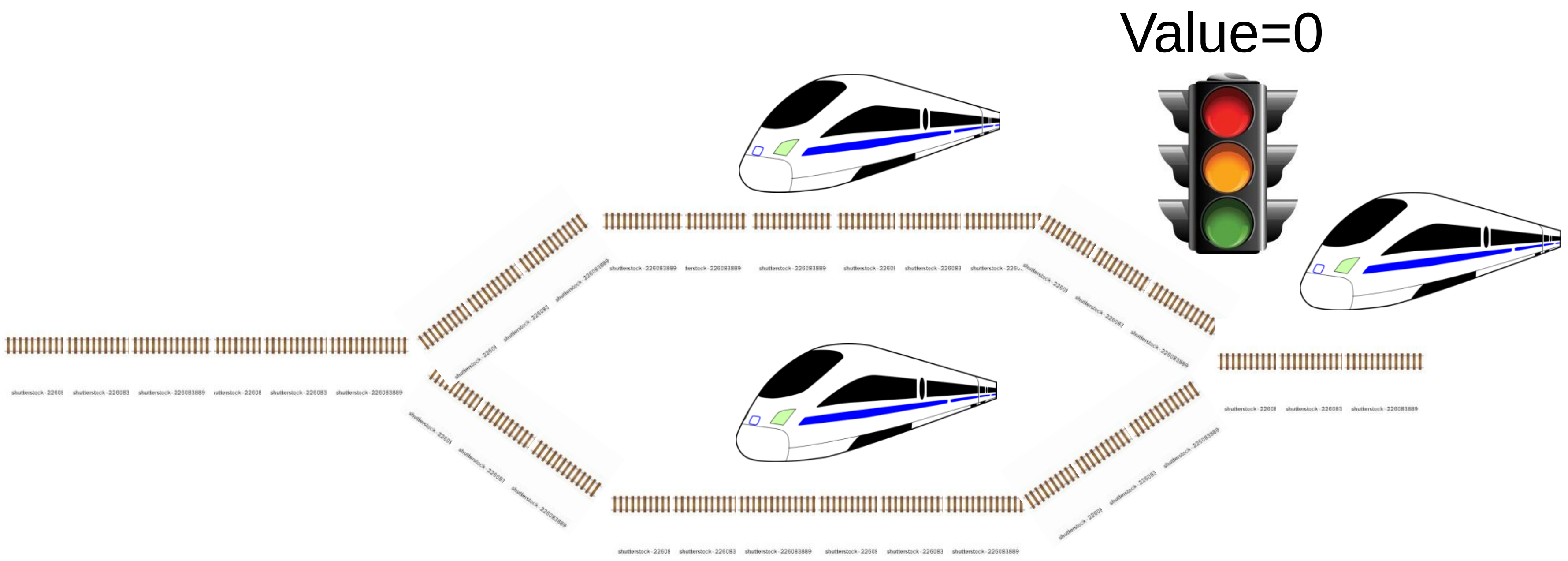
Value=1



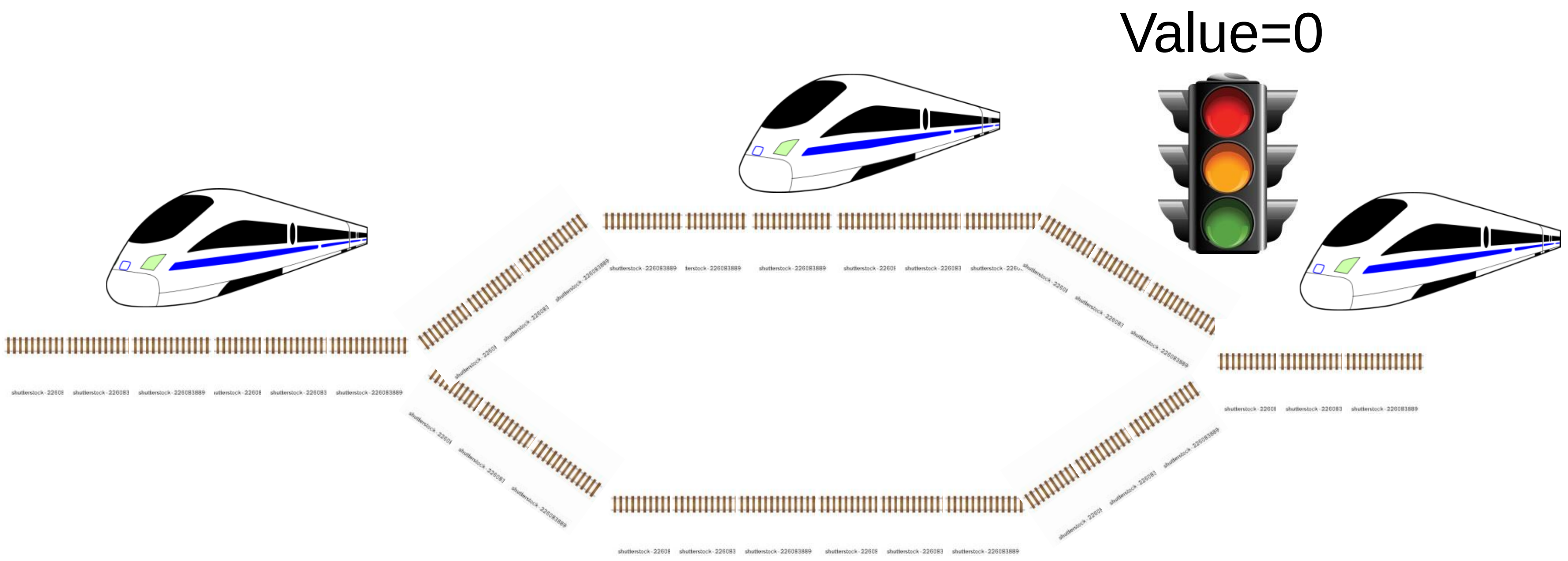
Railroad Analogy Step 4



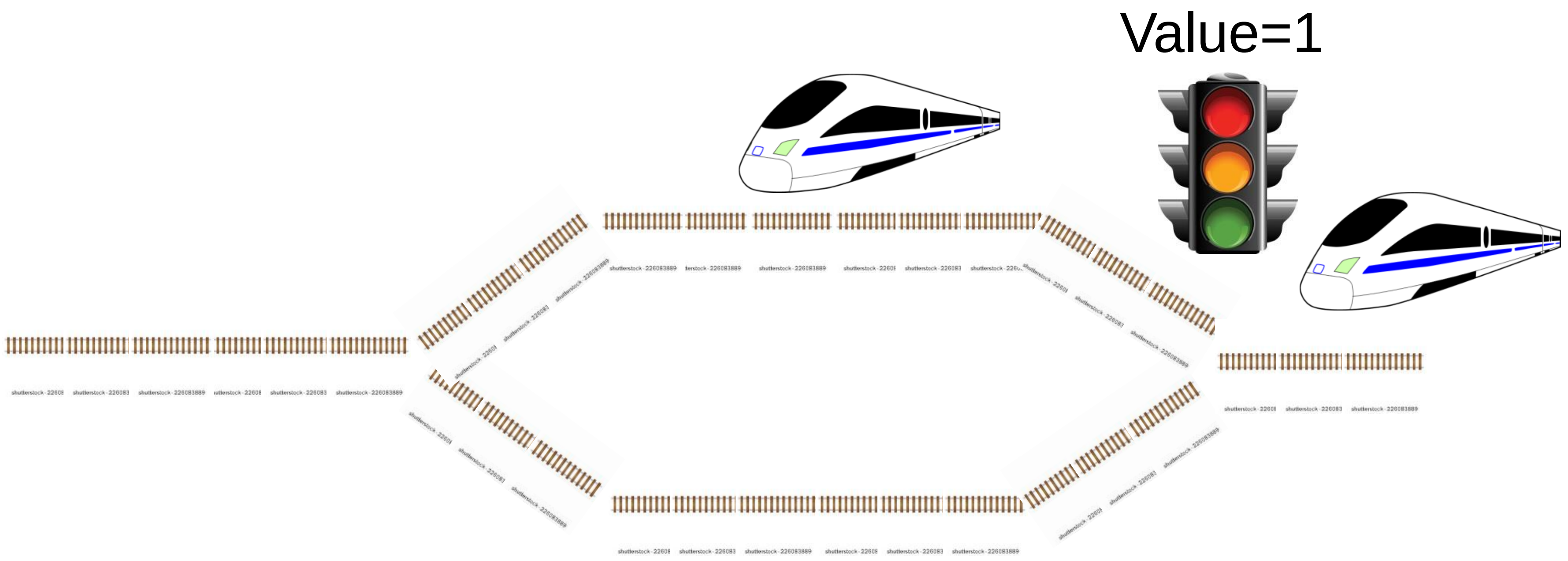
Railroad Analogy Step 5



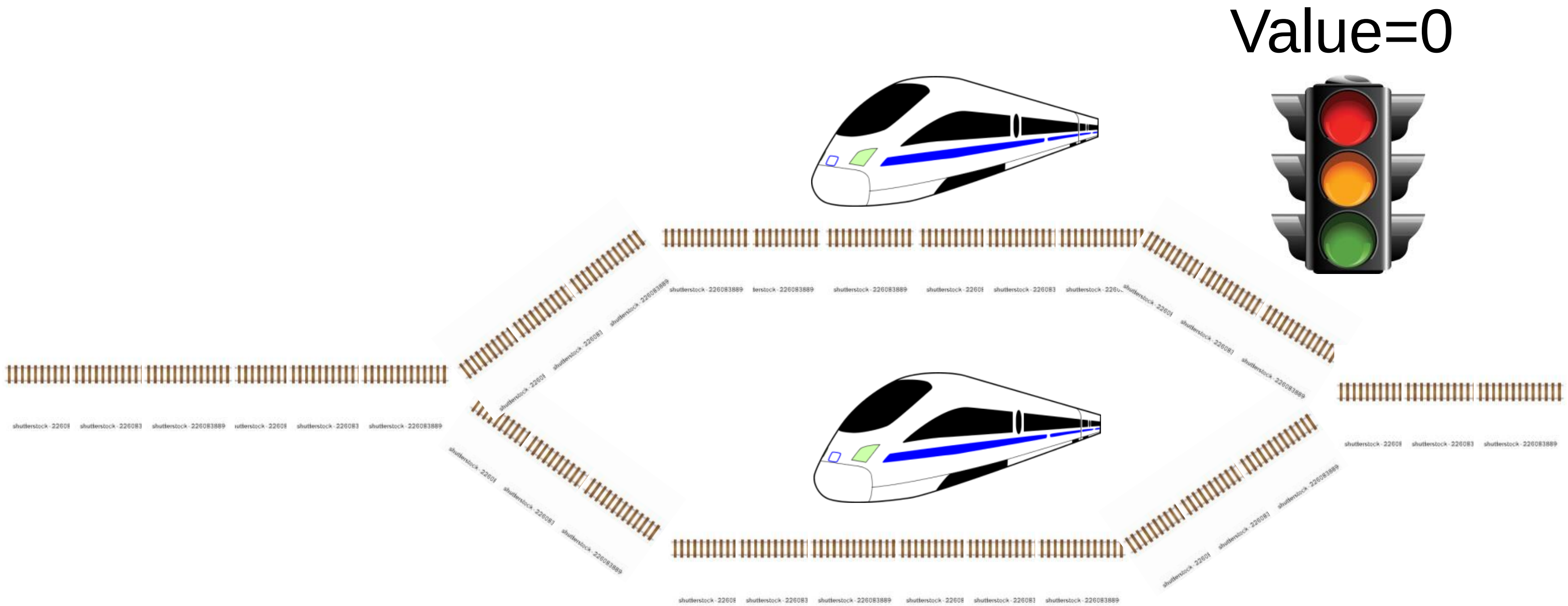
Railroad Analogy Step 6



Railroad Analogy Step 7



Railroad Analogy Step 7



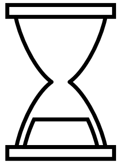
Two Uses of Semaphores

Mutual Exclusion

- Initial value = 1
- Also called “Binary Semaphore”.
- Can be used for mutual exclusion:

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

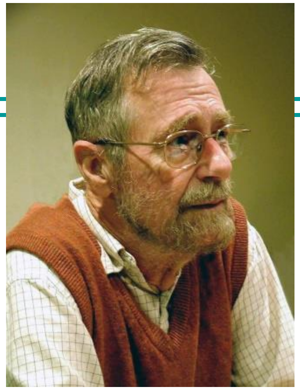
Scheduling Constraints



- Initial value = 0
- What if you want a thread to wait for something?
- Example: Implement ThreadJoin (wait for a thread to terminate):

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

Semaphores



- Semaphores are a kind of generalized lock
  - 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
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch

https://www.youtube.com/watch?v=LKQpy107yUY

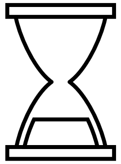
Two Uses of Semaphores

Mutual Exclusion

- Initial value = 1
- Also called “Binary Semaphore”.
- Can be used for mutual exclusion:

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

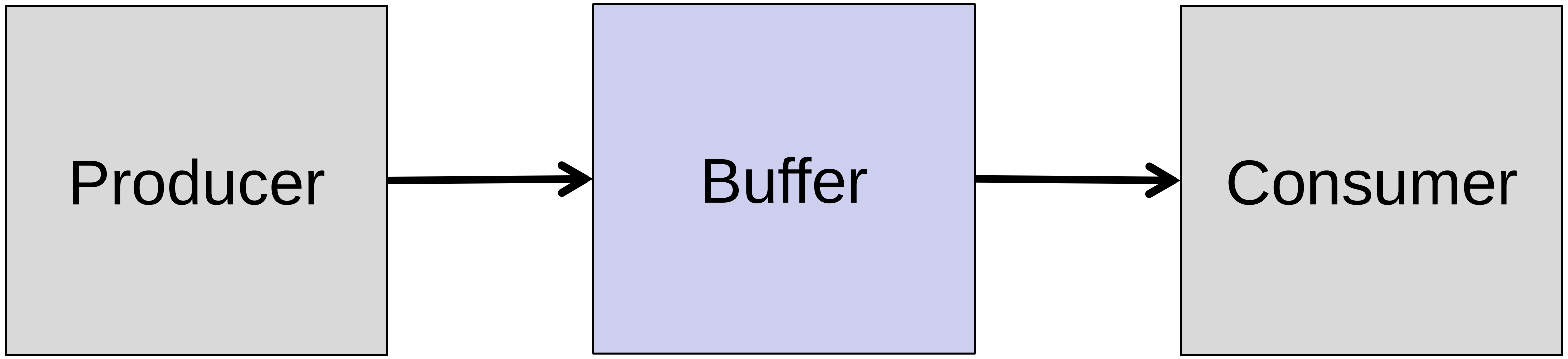
Scheduling Constraints



- Initial value = 0
- What if you want a thread to wait for something?
- Example: Implement ThreadJoin (wait for a thread to terminate):

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

A Bounded Buffer



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

- Synchronize access to the buffer
- Producer needs to wait if buffer is full
- Consumer needs to wait if buffer is empty

Example: Drink machine

- Producer can put limited number of bottles in machine
- Consumer can't take bottles out if machine is empty

Example: GCC

- `cpp | cc1 | cc2 | as | ld`

Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if all are empty (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all are full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine

Correctness constraints for solution

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

Full Solution to Bounded Buffer

```
Semaphore fullBuffer = 0;     // Initially, no pop
Semaphore emptyBuffers = numBuffers;
                              // Initially, num empty slots
Semaphore mutex = 1;          // No one using machine

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

Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`

- 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?

So Far

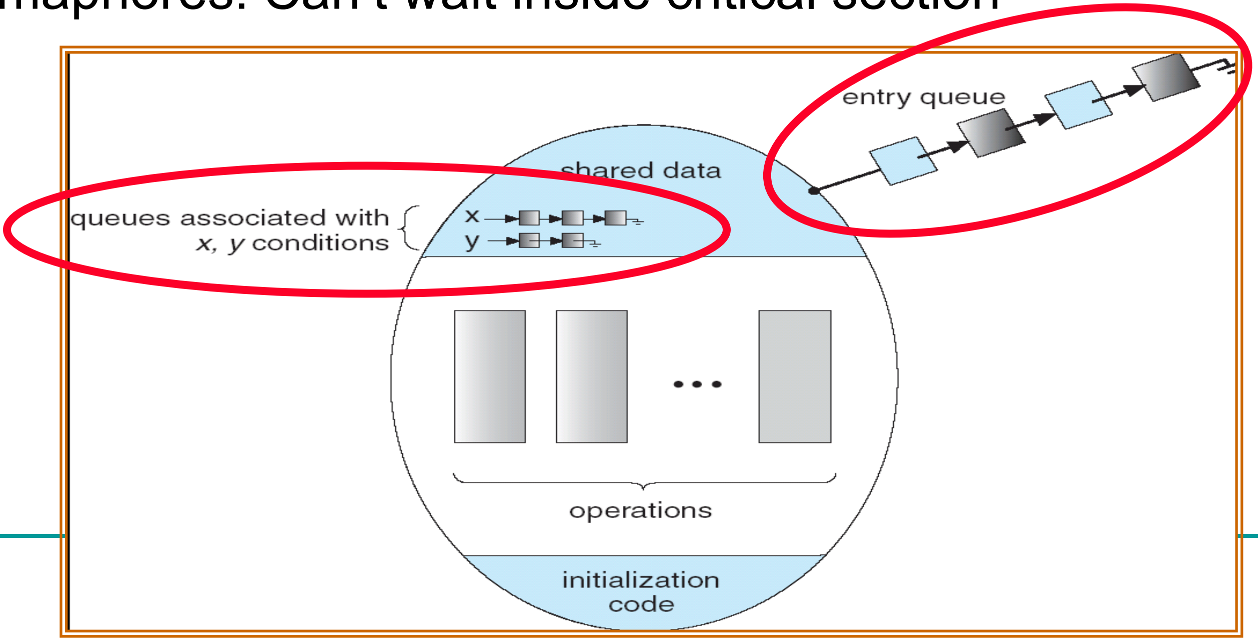
- Concurrency challenge
- Motivation for Synchronization and Locks
- Atomic Read-Modify-Write Operations
- Higher Level Synchronization Atoms
  - Semaphores
  - Monitors

Motivation for 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 purpose:
    - They are used for both mutex and scheduling constraints
    - Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?
- Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Definition: 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
  - Contrast to semaphores: Can't wait inside critical section



Simple Monitor Example (version 1)

- 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

- How do we change the `RemoveFromQueue()` routine to wait until something is on the queue?
  - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone

- 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

Condition Variables

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

- Rule: Must hold lock when doing condition variable ops!

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
```

Mesa vs. Hoare scheduling

Hoare-style (textbooks):

- Signaler gives lock and CPU to waiter
- Waiter runs immediately
- Waiter gives lock and CPU back to signaler when it exits critical section or waits again

Mesa-style (most real OS):

- Signaler keeps lock and processor
- Waiter placed on ready queue with no special priority
- Practically, need to check condition again after wait

Conclusion

- Concurrency challenge
- Motivation for Synchronization and Locks
- Atomic Read-Modify-Write Operations
- Higher Level Synchronization Atoms
  - Semaphores
  - Monitors