CS162
Operating Systems and Systems Programming
Lecture 7

Concurrency

Professor Natacha Crooks & Matei Zaharia

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, , Alison Norman and Lorenzo Alvisi

Recall: Correctness with Concurrency is Hard

Threaded programs must work for all interleavings of thread instruction sequences

Cooperating threads are inherently non-deterministic and non-reproducible

Really hard to debug unless carefully designed!

The Milk Buying Problem

Motivating Example: “Too Much Milk”

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

Desired Correctness Properties

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

– Never more than one person buys milk

– Someone buys if needed

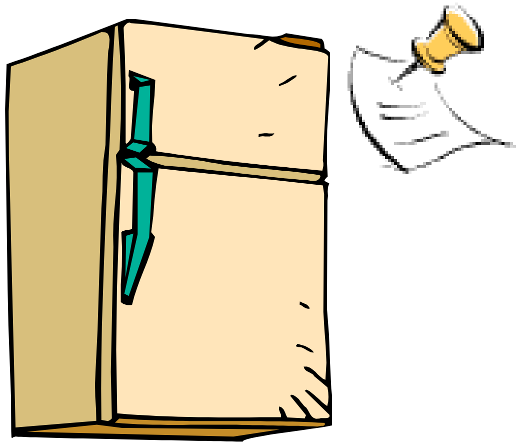
First attempt: Restrict ourselves to use only **atomic load and store operations** as building blocks

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;
   }
}
```



Too Much Milk: Solution #1

Thread A

```
if (noMilk) {


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

Thread B

```

if (noMilk) {
   if (noNote) {







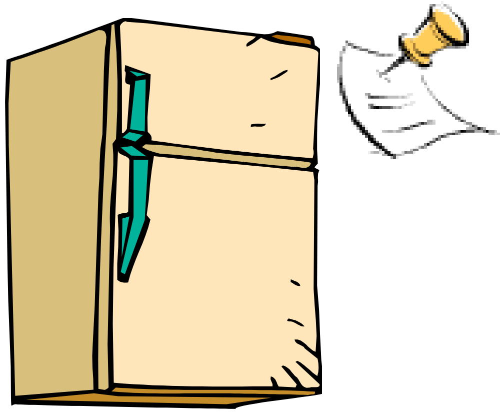
      leave Note;
      buy Milk;
      remove Note;
   }
}
```

Too Much Milk: Solution #1

Still too much milk but only occasionally!

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

Solution makes problem worse since fails intermittently
– Makes it really hard to debug…
– Must work despite what the dispatcher does!



Too Much Milk: Solution #1½

Let's try to fix this by placing note first

```
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:

Thread A

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

Thread B

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

Too Much Milk Solution #2

Possible for neither thread to buy milk

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

Really insidious:

– Extremely unlikely this would happen, but it will at the worst possible time

Too Much Milk Solution #3

Thread A

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

Thread B

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

Claim: this solution is correct, given certain assumptions about our hardware

To Reason about Whether This is Correct

Need some details about how our computer’s memory works (a “memory model”)

We already assumed that individual loads and stores are atomic

What can we assume about a series of loads and stores by our threads? In this example, we’ll assume **sequential consistency**, which means:

- There is a single, global order of loads and stores, and the operations issued by each thread are processed in program order

For example, when Thread A does “buy milk” followed by “remove note A”, Thread B will be able to “see” the milk before it “sees” that note A is gone

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

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




if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

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

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

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

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

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

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

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

happened before

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

Wait for note B to be removed

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"

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

happened before

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

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"

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

happened before

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

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

- “`if (noNote A)`” happens before “`leave note A`”

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

happened before

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

Wait for note B to be removed

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

To formally prove things like this: look up temporal logics, e.g. TLA+

This Solution Generalizes to n Threads

Lamport's Bakery Algorithm (1974)

Computer Systems

G. Bell, D. Siewiorek, and S.H. Fuller, Editors

A New Solution of Dijkstra's Concurrent Programming Problem

Leslie Lamport
Massachusetts Computer Associates, Inc.

A simple solution to the mutual exclusion problem is presented which allows the system to continue to operate

Solution #3 discussion

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"

Too Much Milk: Solution #4?

Recall our lock API:

- `acquire(&milklock)` – Wait until lock is free, then grab it
- `release(&milklock)` – Unlock, waking up anyone waiting
- These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock

Then, our milk problem is easy:

```
acquire(&milklock);
if (nomilk)
   buy milk;
release(&milklock);
```

Where Are We Going with Synchronization?

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

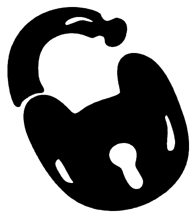
Implement various higher-level synchronization primitives using atomic operations

How to Implement Locks?

Prevents someone from doing something

Lock before entering critical section and
before accessing shared data

Unlock when leaving, after accessing shared data



Hardware Lock Instruction?

Is this a good idea?

What about putting a task to sleep?

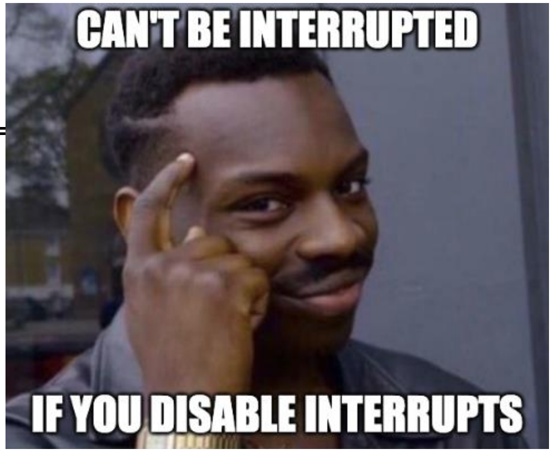
What is the interface between the hardware and scheduler?

Complexity?

» Done in the Intel 432

» Each feature makes HW more complex and slow

How about disabling interrupts?



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 (although virtual memory tricky)

» Preventing external events by disabling interrupts

How about disabling interrupts?

Naïve implementation of locks:

LockAcquire { disable Ints; }

LockRelease { enable Ints; }

Problems with this approach?

How about disabling interrupts?

Consider the following:

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

Real-Time system—no guarantees on timing!

Critical Sections might be arbitrarily long

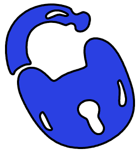
What happens with I/O or other important events?

“Reactor about to meltdown. Help?”

Disabling Interrupts – But more smartly

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
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

New Lock Implementation: Discussion

Why do we need to disable interrupts at all?

- Avoid interruption 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, the critical section (inside `Acquire()`) is very short

Interrupt Re-enable in Going to Sleep

What about re-enabling interrupts when going to sleep?

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

Interrupt Re-enable in Going to Sleep

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?

Interrupt Re-enable in Going to Sleep

What about re-enabling ints when going to sleep?

Enable Position

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

After putting the thread on the wait queue?

Interrupt Re-enable in Going to Sleep

What about re-enabling ints when going to sleep?

Enable Position

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

After putting the thread on the wait queue?

How to Re-enable After Sleep()?

In scheduler, since interrupts are disabled when you call sleep:
- Responsibility of the next thread to re-enable interrupts
- When the sleeping thread wakes up, returns to acquire and re-enables interrupts

Thread A		Thread B
.		
.		
disable ints		
sleep	context switch →	
		sleep return
		enable ints
		.
		.
		.
		disable ints
	← context switch	sleep
sleep return		
enable ints		
.		

Atomic Read-Modify-Write Instructions

Problems with previous solution:

- Can’t give lock implementation to users
- Doesn’t work well on multiprocessor

Alternative: atomic instruction sequences

- These instructions read a value 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

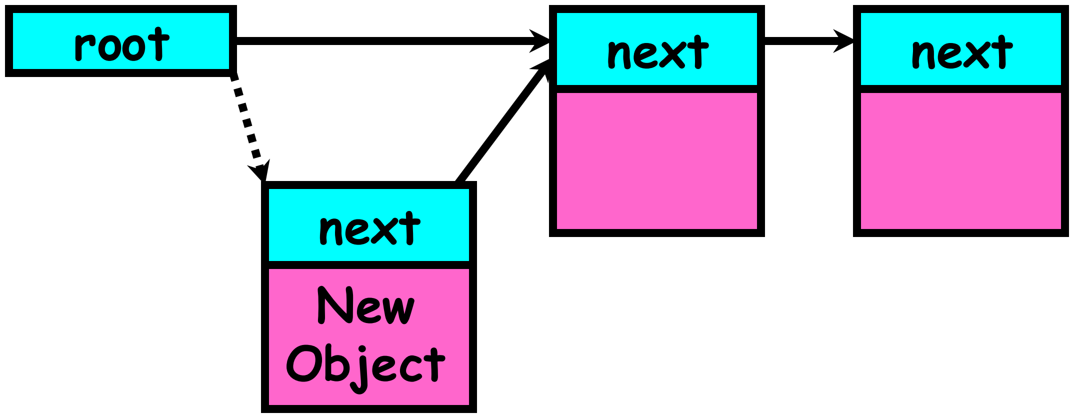
- ```
  test&set (&address) {           /* most architectures */
      result = M[address];        // return result from "address" and
      M[address] = 1;             // set value at "address" to 1
      return result;
  }
  ```

- ```
  swap (&address, register) {     /* x86 */
      temp = M[address];          // swap register's value to
      M[address] = register;      // value at "address"
      register = temp;
  }
  ```

- ```
  compare&swap (&address, reg1, reg2) { /* x86 (returns old value), 68000 */
      if (reg1 == M[address]) {   // If memory still == reg1,
          M[address] = reg2;      // then  put reg2 => memory
          return success;
      } else {                    // Otherwise do not change memory
          return failure;
      }
  }
  ```
  ```

Using of Compare&Swap for queues

```
addToQueue(&object) {
    do {                    // repeat until no conflict
        ld r1, M[root]      // get ptr to current head
        st r1, M[object]    // save link in new object
    } until (compare&swap(&root, r1, object));
}
```



Implementing Locks with test&set

Simple lock that doesn't require entry into the kernel:

```
// Storage: just a memory location in userspace; 0 means "unlocked"
int mylock = 0; // Interface: acquire(&mylock);
                //            release(&mylock);

acquire(int *thelock) {
   while (test&set(thelock)); // Atomic operation!
}


release(int *thelock) {
   *thelock = 0;               // Atomic operation!
}
```

Implementing Locks with test&set

Simple explanation:

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

Busy-Waiting: thread consumes cycles while waiting

- For multiprocessors: every test&set() is a write, which makes value ping-pong around in cache (using lots of network bandwidth)

Problem: Busy-Waiting for Lock

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

Negatives
- This is very inefficient as thread will consume cycles waiting
- Waiting thread may take cycles away from thread holding lock (no one wins!)
- Homework/exam solutions should avoid busy-waiting!

Better Locks using test&set

Idea: only busy-wait to atomically check lock value

```
int guard = 0; // Global Variable!
int mylock = FREE; // Interface: acquire(&mylock);
                   //            release(&mylock);
```

```
acquire(int *thelock) {
   // Short busy-wait time
   while (test&set(guard));
   if (*thelock == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
      // guard == 0 on wakeup!
   } else {
      *thelock = BUSY;
      guard = 0;
   }
}
```

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

Linux futex: Fast Userspace Mutex

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val,
         const struct timespec *timeout );
```

uaddr points to a 32-bit value in user space

futex_op

- FUTEX_WAIT – if val == *uaddr sleep till FUTEX_WAKE
  - » ***Atomic*** check that condition still holds after we disable interrupts (in kernel!)
- FUTEX_WAKE – wake up at most val waiting threads
- FUTEX_LOCK_PI, FUTEX_WAKE_OP, FUTEX_CMP_REQUEUE: More interesting operations!

timeout

- ptr to a *timespec* structure that specifies a timeout for the op

Linux futex: Fast Userspace Mutex

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val,
          const struct timespec *timeout );
```

Interface to the kernel sleep() functionality!

– Let thread put themselves to sleep – conditionally!

futex is not exposed in libc; it is used within the implementation of pthreads

– Can be used to implement locks, semaphores, monitors, etc…

Example: First try: T&S and futex

```
int mylock = 0; // Interface: acquire(&mylock);
                //            release(&mylock);
```

```
acquire(int *thelock) {
   while (test&set(thelock)) {
      futex(thelock, FUTEX_WAIT, 1);
   }
}
```

```
release(int *thelock) {
   thelock = 0; // unlock
   futex(thelock, FUTEX_WAKE, 1);

}
```

Sleep interface by using futex – no busywaiting

No overhead to acquire lock

Every unlock has to call kernel to potentially wake someone up – even if none

Example: Try #2: T&S and futex

```
bool maybe = false;
int mylock = 0; // Interface: acquire(&mylock, &maybe_waiters);
                //            release(&mylock, &maybe_waiters);

acquire(int *thelock, bool *maybe) {
   while (test&set(thelock)) {
      // Sleep, since lock busy!
      *maybe = true;
      futex(thelock, FUTEX_WAIT, 1);

      // Make sure other sleepers not stuck
      *maybe = true;
   }
}
```

```
release(int *thelock, bool *maybe) {
   thelock = 0;
   if (*maybe) {
      *maybe = false;
      // Try to wake up someone
      futex(thelock, FUTEX_WAKE, 1);
   }
}
```

This is syscall-free in the uncontended case

– Temporarily falls back to syscalls if multiple waiters, or concurrent acquire/release

- But it can be considerably optimized!

– See "Futexes are Tricky" by Ulrich Drepper

Where are we going with synchronization?

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

Higher-level Primitives than Locks

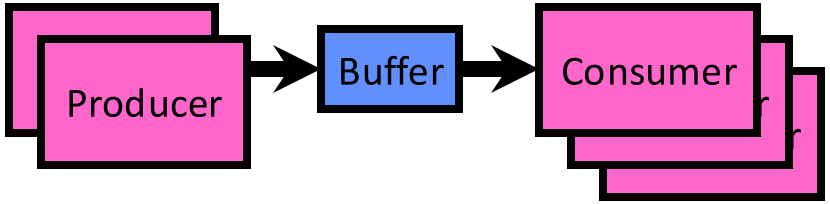
Goal of last couple of lectures:

- What is right abstraction for synchronizing threads that share memory?
- Want as high-level a primitive as possible

Synchronization is a way of coordinating multiple concurrent activities that are using shared state

- This lecture and the next presents some ways of structuring sharing

Producer-Consumer with a Bounded Buffer



Problem Definition

– Producer(s) put things into a shared buffer

– Consumer(s) take them out

– Need synchronization to coordinate producer/consumer

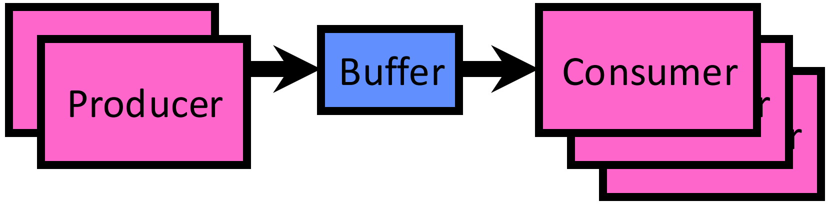
Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them

– Need to synchronize access to this buffer

– Producer needs to wait if buffer is full

– Consumer needs to wait if buffer is empty

Producer-Consumer with a Bounded Buffer



Example 1: GCC compiler

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

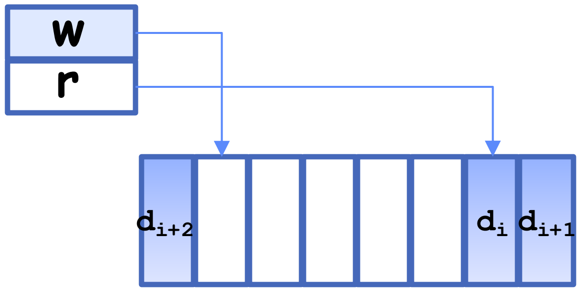
Example 2: Coke machine

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

Others: Web servers, Routers, ….

Circular Buffer Data Structure (sequential case)

```
typedef struct buf {
  int write_index;
  int read_index;
  <type> *entries[BUFSIZE];
} buf_t;
```



Insert: write & bump write ptr (enqueue)

Remove: read & bump read ptr (dequeue)

How to tell if Full (on insert) Empty (on remove)?

And what do you do if it is?

What needs to be atomic?

Circular Buffer – first cut

```
mutex buf_lock = <initially unlocked>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) {}; // Wait for a free slot
  enqueue(item);
  release(&buf_lock);
}

Consumer() {
  acquire(&buf_lock);
  while (buffer empty) {}; // Wait for arrival
  item = dequeue();
  release(&buf_lock);
  return item
}
```

Will we ever come out of the wait loop?

Circular Buffer – 2nd cut

```
mutex buf_lock = <initially unlocked>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) {release(&buf_lock); acquire(&buf_lock);}
  enqueue(item);
  release(&buf_lock);
}

Consumer() {
  acquire(&buf_lock);
  while (buffer empty) {release(&buf_lock); acquire(&buf_lock);}
  item = dequeue();
  release(&buf_lock);
  return item
}
```

What happens when one is waiting for the other?
- Multiple cores ?
- Single core ?

Semaphores

Semaphores are a type of generalized lock

First defined by Dijkstra in late 60s

Main synchronization primitive in original UNIX

Semaphores

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

– Set value when you initialize

– Down() or P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1

passeren?
prolaag?
procure

» Think of this as the wait() operation

– Up() or V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any

vrijgave?
verhogen?
vacate

» This 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 initially

– Operations must be atomic

» Two P's together can't decrement value below zero

» Thread going to sleep in P won't miss wakeup from V – even if both happen at same time

Two Uses of Semaphores

Mutual Exclusion (initial value = 1)

Also called “Binary Semaphore” or “mutex”.

Can be used for mutual exclusion, just like a lock:

```
      semaP(&mysem);
// Critical section goes here
      semaV(&mysem);
```

Two Uses of Semaphores

Scheduling Constraints (initial value = 0)

Allow thread 1 to wait for a signal from thread 2

– thread 2 schedules thread 1 when a given event occurs

Suppose you had to implement ThreadJoin which must wait for thread to terminate:

```
Initial value of semaphore = 0
ThreadJoin {
   semaP(&mysem);
}
ThreadFinish {
   semaV(&mysem);
}
```