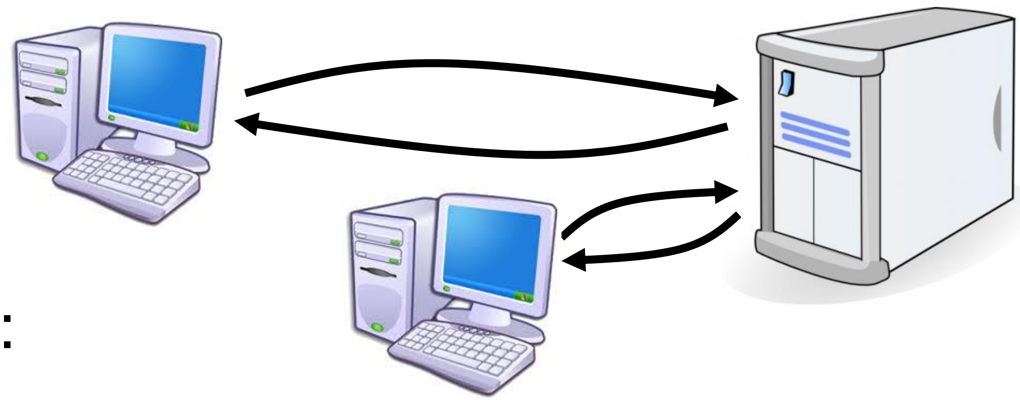
CS330: Synchronization

Instructor: Youngjin Kwon

Threaded Web Server



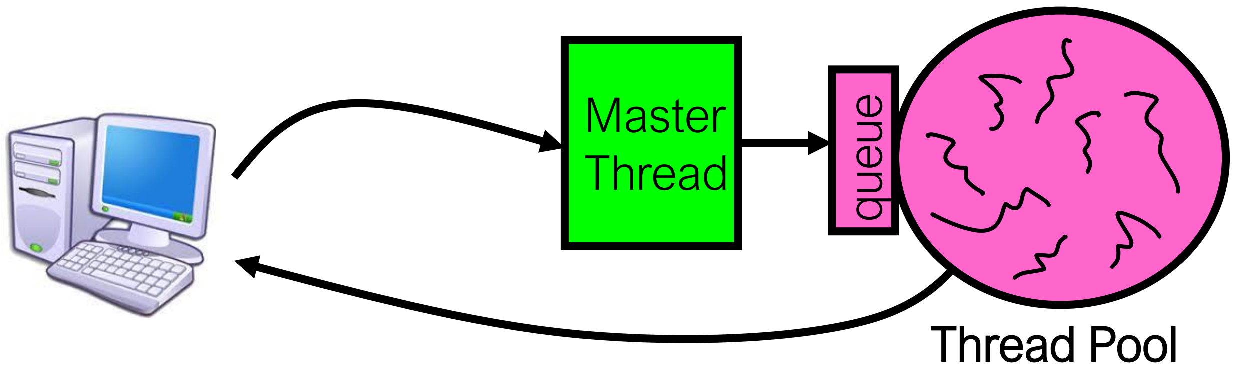
- Multi-threaded version:

```
serverLoop() {
    connection = AcceptConnection();
    ThreadCreate(ServiceWebPage(),connection);
  }
```

- Advantages of threaded version:
  - Can **share file caches** kept in memory, results of PHP scripts, other things
  - Threads are ***much* cheaper to create** than processes, so this has a lower per-request overhead
- What if too many requests come in at once?

Thread Pools

- Problem with previous version: Unbounded Threads
- Instead, allocate a bounded "pool" of threads, representing the maximum level of multiprogramming




```
master() {
    allocThreads(slave,queue);
    while(TRUE) {
        con=AcceptConnection();
        Enqueue(queue,con);
        wakeUp(queue);
    }
}
```

```
slave(queue) {
    while(TRUE) {
        con=Dequeue(queue);
        if (con==null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
```

Shared states are necessary evil

- Shared states are useful!
  - Shared variables of threads are much cheaper than those of processes

- Shared states are horrible!
  몰이해할.
  - **Programs must be insensitive to arbitrary interleavings**
  - Without careful design, shared variables can output completely inconsistent results

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, what if on shared data (initially, y = 0)?

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

- What are the possible values of x?

Thread A	Thread B
x = 1;	
x = y+1;	
	y = 2;
	y = y*2

x=1

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, what if on shared data (initially, y = 0)?

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

- What are the possible values of x?

Thread A	Thread B
	y = 2;
	y = y*2;
x = 1;	
x = y+1;	

x=5

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, what if on shared data (initially, y = 0)?

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

- What are the possible values of x?

Thread A	Thread B
	y = 2;
x = 1;	
x = y+1;	
	y= y*2;

x=3

Definitions

- **Race condition**
  - A situation where multiple processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order (interleaving) of such accesses

More Definitions

- **Critical Section**: a piece of code that accesses a shared resource (producing a race condition)
- **Mutual Exclusion**: ensuring that only one thread executes critical section
  - One thread *excludes* the other(s) while doing its task
- **Lock:** prevent someone from doing something
  - Lock before entering critical section, before accessing shared data
  - Unlock when leaving, after done accessing shared data
  - Wait if locked (all synchronization involves waiting!)

"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

Two guarantees

- Safety: A program never enters a bad state
  - Too much milk: Never more than one person buys milk
- Liveness: A program eventually a good state
  - Too much milk: If milk is needed, someone eventually buys it

Too Much Milk, Try #1

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)
- Try #1: leave a note

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

Too Much Milk: Solution #1

- Still too much milk but only occasionally!

```
    Thread A                                Thread B
  if (!Milk)
    if (!Note) {
                                          if (!Milk)
                                            if (!Note) {
      leave Note;
      buy milk;
      remove note;
    }
  }
                                                leave Note;
                                                buy milk;
                                                …
```

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

간헐적

Solution makes problem worse since fails intermittently

Makes it really hard to debug…

Must work despite how threads are interleaved

Too Much Milk, Try #2

Thread A

```
leave note A ①
if (!note B) {   ④
    if (!milk)
        buy milk
}
remove note A ⑤
```

Thread B

```
② leave note B
③ if (!noteA) {
       if (!milk)
           buy milk
   }
⑥ remove note B
```

→ Liveness problem

Too Much Milk Solution #2

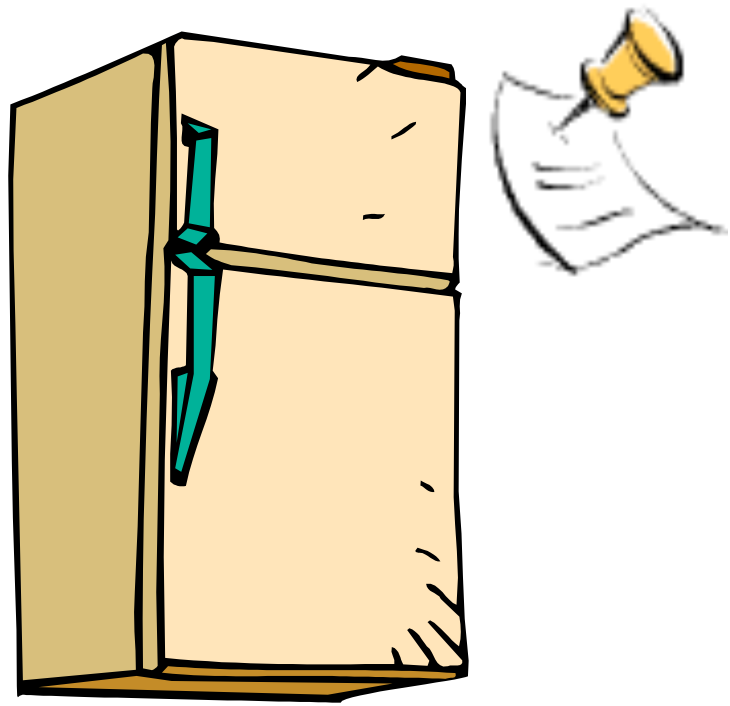
- Possible for neither thread to buy milk!

<u>Thread A</u> <u>Thread B</u>

```
leave note A;
                              leave note B;
                              if (!Note A) {
                          stop  if (!Milk) {
                                  buy Milk;
                                }
                              }
if (!Note B) {
  if (!Milk) {      stop
    buy Milk;
…
                                remove note B;
```

- Really insidious:
  - Unlikely that this would happen, but possible at worst case

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, Try #3

Thread A

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

Thread B

```
leave note B
if (!noteA) {   // Y
    if (!milk)
        buy milk
}
remove note B
```

Can guarantee at X and Y that either:

(i) Safe for me to buy

(ii) Other will buy, ok to quit

Lessons

- Solution is ***complicated***
  - "obvious" code often has bugs
- Generalizing to many threads/processors
  - Even more complex: see Peterson's algorithm

OS needs to provide a simple way to solve problems like too much milk!

Locks

- Lock::acquire
  - wait until lock is free, then take it
- Lock::release
  - release lock, waking up anyone waiting for it

Formal guarantees

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (liveness)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (liveness)

Too Much Milk, #4

Locks allow concurrent code to be much simpler:

```
lock.acquire();
if (!milk)
    buy milk
lock.release();
```

Why does the lock require operating system support?

Roadmap

Concurrent Applications

Shared Objects

Bounded Buffer   Barrier

Synchronization Variables

Semaphores   Locks   Condition Variables

Atomic Instructions

Interrupt Disable   Test-and-Set

Hardware

Multiple Processors   Hardware Interrupts

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
- Never access shared data without lock
  - Danger!

Race condition example

Two threads run on the same bank server

```
withdraw (account, amount) {
1000 balance = get_balance(account);
     balance = balance – amount;
     put_balance(account, balance);
     return balance;
}
```

```
withdraw (account, amount) {
1000 balance = get_balance(account);
     balance = balance – amount;
     put_balance(account, balance);
     return balance;
}
```

Suppose you have a balance of $1000
You visit online banking site with two web browsers
and try to withdraw $100 at the same time

What happens? What is/are possible outcome(s)?

$900 result balance with 2 x $100 withdrawals

Lock Example

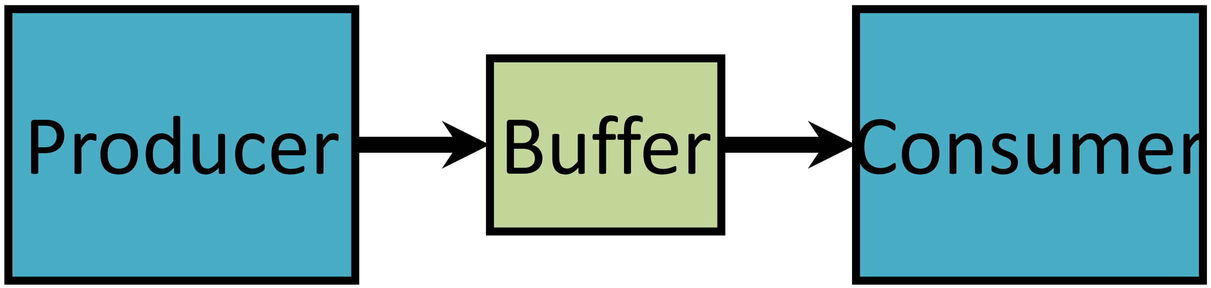
```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

Critical Section

```
acquire(lock);
balance = get_balance(account);
balance = balance – amount;

acquire(lock);

put_balance(account, balance);
release(lock);

balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
release(lock);
```

Bounded-Buffer (producer-consumer) Problem

- 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: Bounded Buffer

Initially: front = tail = 0; lock = FREE; MAX is buffer capacity

```
tryput(item) {
    lock.acquire();
    if ((tail – front) < size) {
        buf[tail % MAX] = item;   ②
        tail++;
    }
    lock.release();
}
```

```
tryget() {
    item = NULL;
    lock.acquire();
    if (front < tail) {
        item = buf[front % MAX];
  ①     front++;
    }
    lock.release();
③  return item;
}
```

Question

- If tryget returns NULL, do we know the buffer is empty?

No

- How can a thread know when a buffer is empty?
  - **Need another primitive for the purpose**

Condition Variables

- Waiting for a change to shared states
  - Called only when holding a lock

- Wait(): atomically release lock and relinquish processor
  - Re-acquire the lock when wakened
- Signal(): wake up a waiter, if any
- Broadcast(): wake up all waiters, if any

Condition Variable Design Pattern

cond_var is a condition variable

```
FunctionThatWaits() {
    lock.acquire();
    // Read/write shared state

    while (!testSharedState()) {
        cond_var.wait(&lock);
    }

    // Read/write shared state
    lock.release();
}
```

```
FunctionThatSignals() {
    lock.acquire();
    // Read/write shared state

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


    lock.release();
}
```

Conditional variable is memoryless

- CV does not have internal states other than a queue of waiting thread
- If no threads are in the waiting queue, a signal or broadcast has no effect
- CV does not have memory of earlier calls of signal or broadcast

Does it work?

```
methodThatWaits() {
    lock.acquire();
    // Read/write shared state
    lock.release();

    lock.acquire();
    while (!testSharedState()) {
        cv.wait(&lock);
    }
    lock.release();

    lock.acquire();
    // Read/write shared state
    lock.release();
}
```

여기에 오면 씹혀버림

```
methodThatSignals() {
    lock.acquire();
    // Read/write shared state

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

    lock.release();
}
```

전체를 묶어야됨.

Bounded buffer

- Producer

```
int i, loop = MAX_LOOP;
For (i = 0; i < loop; i++) {
    put(i)
}
```

- Consumer

```
int tmp;
while(1) {
    tmp = get();
    printf("%d\n", tmp);
}
```

Safety problem

Try 1: Bounded Buffer

Initially: front = tail = 0; MAX is buffer capacity
cond_var are condition variables

C1 C2 P
① block ② wake all ③ fill
④ Empty buf again!

```
put(item) {
   lock.acquire();
   if ((tail – front) == MAX) {
      cond_var.wait(lock);
   }
   buf[tail % MAX] = item;
   tail++;
   cond_var.signal(lock);
   lock.release();
}
```

C1 ①

②

③ C2

```
get() {
   lock.acquire();
   if (front == tail) {
      cond_var.wait(lock);
   }
   item = buf[front % MAX];
   front++;
   cond_var.signal(lock);
   lock.release();
   return item;
}
```

C1 empty

Liveness problem

Try 2: Bounded Buffer

Initially: front = tail = 0; MAX is buffer capacity
cond_var are condition variables

```
put(item) {
  lock.acquire();
  while ((tail – front) == MAX) {
    cond_var.wait(lock);
  }
  buf[tail % MAX] = item;
  tail++;
  cond_var.signal(lock);
  lock.release();
}
```

```
get() {
  lock.acquire();
  while (front == tail) {
    cond_var.wait(lock);
  }
  item = buf[front % MAX];
  front++;
  cond_var.signal(lock);
  lock.release();
  return item;
}
```

C1 C2 ①×2 ② C2 ③ ④ C2 ⑤ → empty C1

Solution: Bounded Buffer

Initially: front = tail = 0; MAX is buffer capacity
empty/full are condition variables

```
put(item) {
    lock.acquire();
    while ((tail – front) == MAX) {
        full.wait(lock);
    }
    buf[tail % MAX] = item;
    tail++;
    empty.signal(lock);
    lock.release();
}
```

```
get() {
    lock.acquire();
    while (front == tail) {
        empty.wait(lock);
    }
    item = buf[front % MAX];
    front++;
    full.signal(lock);
    lock.release();
    return item;
}
```

Summary: Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up

Summary: Condition Variables, cont'd

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

```
while (needToWait()) {
    condition.Wait(lock);
}
```

Remember the rules

Memoryless!

- Always use locks and condition variables for shared states
- Always acquire lock at "beginning of procedure, release at end"
- Always hold lock when using a condition variable
- Always wait in while loop
- Never use sleep() to wait for a thread to finish a task

Implementing Synchronization

Concurrent Applications

Shared Objects

Bounded Buffer   Barrier

Synchronization Variables

Semaphores   Locks   Condition Variables

Atomic Instructions

Interrupt Disable   Test-and-Set

Hardware

Multiple Processors   Hardware Interrupts

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**

Naïve use of Interrupt Enable/Disable

- How can we build "atomic operations"?
  - Recall: A thread loses its control in two ways.
    - Internal: Relinquishing the CPU (e.g., sleep, yield)
    - External: Interrupts (ex. timer)
- 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; }
```

Why implemented by kernel? : these are privileged instructions

Disabling interrupt

- Privileged instruction or not?

Yes

- If code between enabling/disabling interrupt is long, then what is a problem?

Concurrency: For example, if timer interrupt got off, then every other system with timer gets lost (ex, scheduler, ...)

Lock Implementation, Uniprocessor

```
Int lockValue = FREE;

Lock::acquire() {
    disableInterrupts();
    if (lockValue == BUSY) {
        waiting.add(TCB);
        TCB->state = WAITING;
        next = readyList.remove();
        switch(TCB, next);
        TCB->state = RUNNING;
    } else {
        lockValue = BUSY;
    }
    enableInterrupts();
}
```

Code goes here

```
Lock::release() {
    disableInterrupts();
    if (!waiting.Empty()) {
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        lockValue = FREE;
    }
    enableInterrupts();
}
```

How to Re-enable After switch()?

- 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		Thread B
.		
.		
disable ints		
switch	context switch →	
		return from switch
		enable ints
		.
		.
		.
		disable int
	← context switch	switch
return from switch		
enable ints		
.		

Enable/disable interrupt in **multiprocessors**

- Does it guarantee atomic instructions like uniprocessor case?

No,

E/D interrupt is only per processor

Disable

S

CPU1

S

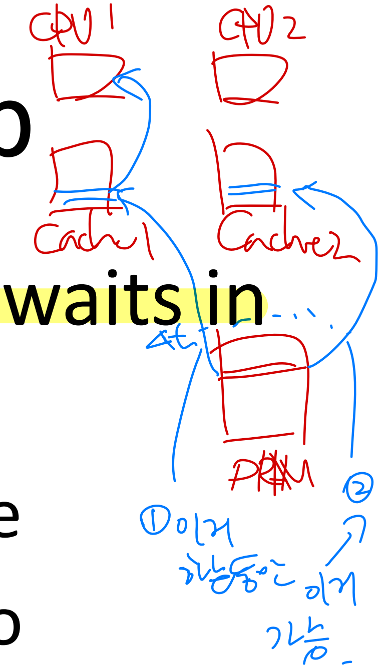
CPU2

Already running

What can OS do to provide atomic instructions?

Using global variable,

Spinlocks with busy loop



A spinlock is a lock where the processor waits in a loop for the lock to become free

- Assumes lock will be held for a short time
- Used to protect the CPU scheduler and to implement locks

```
Spinlock::acquire() {
  while (lockValue == BUSY)
            ;
  lockValue = Busy
}
```

```
Spinlock::release() {
  lockValue = FREE;
}
```

Problems in multiprocessors?

→ HW help

Multiprocessor

- Atomic read-modify-write instructions
  - Atomically read a value from memory, operate on it, and then write it back to memory
  - Intervening instructions prevented in **hardware**

끼어드는

- Examples
  - Test and set
  - Intel: xchgb, lock prefix
  - Compare and swap
- Any of these can be used for implementing locks and condition variables!

Spinlock with test-and-set instruction

BUSY = 1; FREE = 0;
Int lockValue = FREE;

What CPU does atomically:

```
int TestAndSet(int *old_ptr) {
    int old = *old_ptr; // fetch old value at old_ptr
   *old_ptr = BUSY;     // store BUSY into old_ptr
    return old;         // return the old value
}
```

Atomic

```
Spinlock::acquire() {
 while(testAndSet(&lockValue)
           == BUSY)
                ;
}
```

```
Spinlock::release() {
  lockValue = FREE;
}
```

Two main problems in the previous spinlock implementation

- It is spinning!
  - a thread waiting for the spinlock waste CPU cycles
  - Especially severe problem in uniprocessor

Multiple lock waiters

S S S S

try to acquire

lock

- Does not guarantee liveness
  - A thread may keep trying to hold the spinlock

Avoid spinning

```
int TestAndSet(int *old_ptr) {
    int old = *old_ptr; // fetch old value at old_ptr
   *old_ptr = BUSY;      // store BUSY into old_ptr
    return old;          // return the old value
}
```

```
Spinlock::acquire() {
 while(testAndSet(&lockValue)
    == BUSY)
    yield();
}
```

yield CPU

```
Spinlock::release()
{
  lockValue = FREE;
}
```

New HW primitive: fetch-and-add

also atomic

- int FetchAndAdd(int *ptr) {
  int old = *ptr;
  *ptr = old + 1;
  return old;
}

- Any ideas to solve the starvation of the previous spinlock implementation?

Ticket locks

Global variable

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;
```

```
void lock_init(lock_t *lock){
    lock->ticket = 0;
    lock->turn = 0;
}
```

```
void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ;    // spin
}
```

```
void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

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 decreases it by 1
    - Think of this as the **wait**() operation
  - **V():** an atomic operation that increases 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

Two Uses of Semaphores (1)

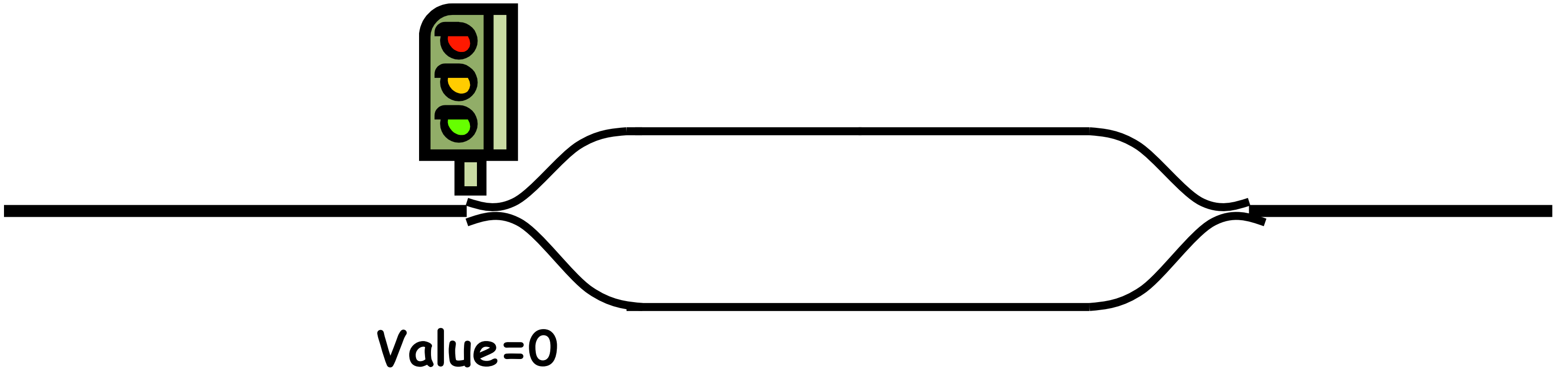
- Mutual Exclusion (initial value = 1)
  - Also called “Binary Semaphore”.
  - Can be used for mutual exclusion:

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

- **Counting** semaphore – integer value can range over an unrestricted domain

Semaphores

- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



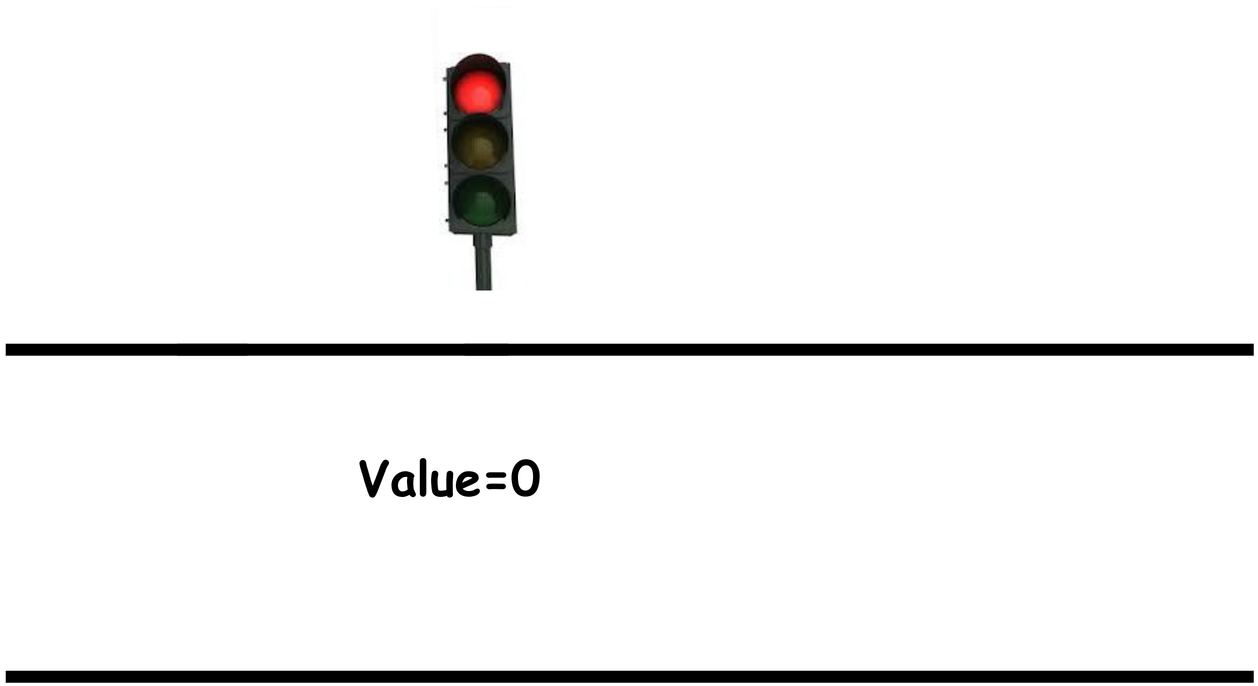
Two Uses of Semaphores (2)

- 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 **constraint** is satisfied
  - Example: suppose you had to implement ThreadJoin which must wait for the thread to terminate:

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

Semaphores

- Semaphore from railway analogy
  - Here is a semaphore initialized to 0 for scheduling constraint:
    - One train leaves only after another train comes



Semaphore Bounded Buffer

Initially: front = last = 0; MAX is buffer capacity
mutex = 1; emptySlots = MAX; fullSlots = 0;

```
get() {
    fullSlots.P();
    mutex.P();
    item = buf[front % MAX];
    front++;
    mutex.V();
    emptySlots.V();
    return item;
}
```

```
put(item) {
    emptySlots.P();
    mutex.P();
    buf[last % MAX] = item;
    last++;
    mutex.V();
    fullSlots.V();
}
```

Recall: conditional variable example

Initially: front = tail = 0; MAX is buffer capacity
empty/full are condition variables

```
put(item) {
    lock.acquire();
    while ((tail – front) == MAX) {
        full.wait(lock);
    }
    buf[tail % MAX] = item;
    tail++;
    empty.signal();
    lock.release();
}
```

```
get() {
    lock.acquire();
    while (front == tail) {
        empty.wait(lock);
    }
    item = buf[front % MAX];
    front++;
    full.signal();
    lock.release();
    return item;
}
```

Implementing Condition Variables using Semaphores (Try 1)

```
wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}
signal() {
    semaphore.V();
}
```

Conditional Variable

CV = memoryless

≠ Semaphore has memory

Solution: Only signal when wait is called

Implementing Condition Variables using Semaphores (Try 2)

```
wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}
signal() {
    if (semaphore is not empty)
        semaphore.V();
}
```

3 in one is not atomic

Not called when empty

Signal lost!

skip

okay

Implementing Condition Variables using Semaphores

```
wait(lock) {
    semaphore = new Semaphore;
    queue.Append(semaphore);   // queue of waiting threads
    lock.release();
    semaphore.P();
    lock.acquire();
}
signal() {
    if (!queue.Empty()) {
        semaphore = queue.Remove();
        semaphore.V();         // wake up waiter
    }
}
```

① ② Can't skip

Remember the rules

- Use consistent structure
- Always use locks and condition variables
- If you are not sure, always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop