Synchronization

Ch. 5 Synchronizing Access to Shared Objects

Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Compiler/hardware instruction reordering
- Multi-word operations are not atomic

Question: Can this panic?

Thread 1

```
p = someComputation();
pInitialized = true;
```

Thread 2

```
for !pInitialized { }
q = someFunction(p);
if q != someFunction(p)
    panic
```

Why Reordering?

- Why do compilers reorder instructions?
  - Efficient code generation requires analyzing control/data dependency
  - If variables can spontaneously change, most compiler optimizations become impossible
- Why do CPUs reorder instructions?
  - Write buffering: allow next instruction to execute while write is being completed

Fix: **memory barrier**

- Instruction to compiler/CPU
- All ops before barrier complete before barrier returns
- No op after barrier starts until barrier returns

Challenges

Definition: Race Condition

- Output of a concurrent program depends on the order of operations between threads

What are the possible final value of *x*?

Thread 1	Thread 2
x = 1;	x = 2;

x = 1 or *x* = 2

What are the possible final value of *x*?

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

x = 13 or *x* = 25

Thread 1	Thread 2
LOAD $y, D0	LOAD $y, D1
ADD #1, D0	MULT #2, D1
STORE D0, $x	STORE D1, $y

What are the possible final value of *x*?

Thread 1	Thread 2
x = x + 1;	x = x + 2;

Interleaving 1		Interleaving 2		Interleaving 3	
LOAD $x, D0		LOAD $x, D0		LOAD $x, D0	
ADD #1, D0			LOAD $x, D1		LOAD $x, D1
STORE D0, $x		ADD #1, D0		ADD #1, D0	
	LOAD $x, D1		ADD #2, D1		ADD #2, D1
	ADD #2, D1	STORE D0, $x			STORE D1, $x
	STORE D1, $x		STORE D1, $x	STORE D0, $x	
[redacted]					

What are the possible final value of *x*?

Thread 1	Thread 2
x = x + 1;	x = x + 2;

Interleaving 1		Interleaving 2		Interleaving 3	
LOAD $x, D0		LOAD $x, D0		LOAD $x, D0	
ADD #1, D0			LOAD $x, D1		LOAD $x, D1
STORE D0, $x		ADD #1, D0		ADD #1, D0	
	LOAD $x, D1		ADD #2, D1		ADD #2, D1
	ADD #2, D1	STORE D0, $x			STORE D1, $x
	STORE D1, $x		STORE D1, $x	STORE D0, $x	
x==3		x==2		x==1	

Definitions

- **Atomic operations:** Indivisible operations that cannot be interleaved with other operations
  - Modern processors load/store 32-bit word from/to memory is atomic operation

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Definitions: Correctness Properties

- **Safety:** The program never enters a bad state
- **Liveness:** The program eventually enters a good state

- Correctness Property: For the Too much milk problem
  1. **Safety:** At most one person buys milk (at any one time).
  2. **Liveness:** If milk is needed, eventually somebody buys milk.
     (but it can take a while; this statement does not offer a time guarantee.)

Too Much Milk, Try #1

- Try #1: leave a note

```
if !note {
    if milk == 0 {
        note = true     // leave note
        milk++          // buy milk
        note = false    // remove note
    }
}
```

Too Much Milk, Try #1

Thread A

```
if !note {
    if milk == 0 {
        note = true
        milk++
        note = false
    }
}
```

Thread B

```
if !note {
    if milk == 0 {
        note = true
        milk++
        note = false
    }
}
```

Too Much Milk, Try #2

Thread A

```
noteA = true
if !noteB {
    if milk == 0
        milk++
}
noteA = false
```

Thread B

```
noteB = true
if !noteA {
    if milk == 0
        milk++
}
noteB = false
```

Too Much Milk, Try #3

Thread A

```
noteA = true
while noteB        // X
    do nothing
if milk == 0
    milk++
noteA = false
```

Thread B

```
noteB = true
if !noteA {          // Y
    if milk == 0
        milk++
}
noteB = false
```

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
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
- Generalizing to many threads/processors
  - Even more complex: see Peterson's algorithm

So what do we need?

- Structured synchronization to enable sharing between threads
  - Define and limit how state can be accessed
  - Coordinate thread access to shared state
  - Best practices for writing code to implement shared objects

Roadmap

Concurrent Applications

Shared Objects

Bounded Buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

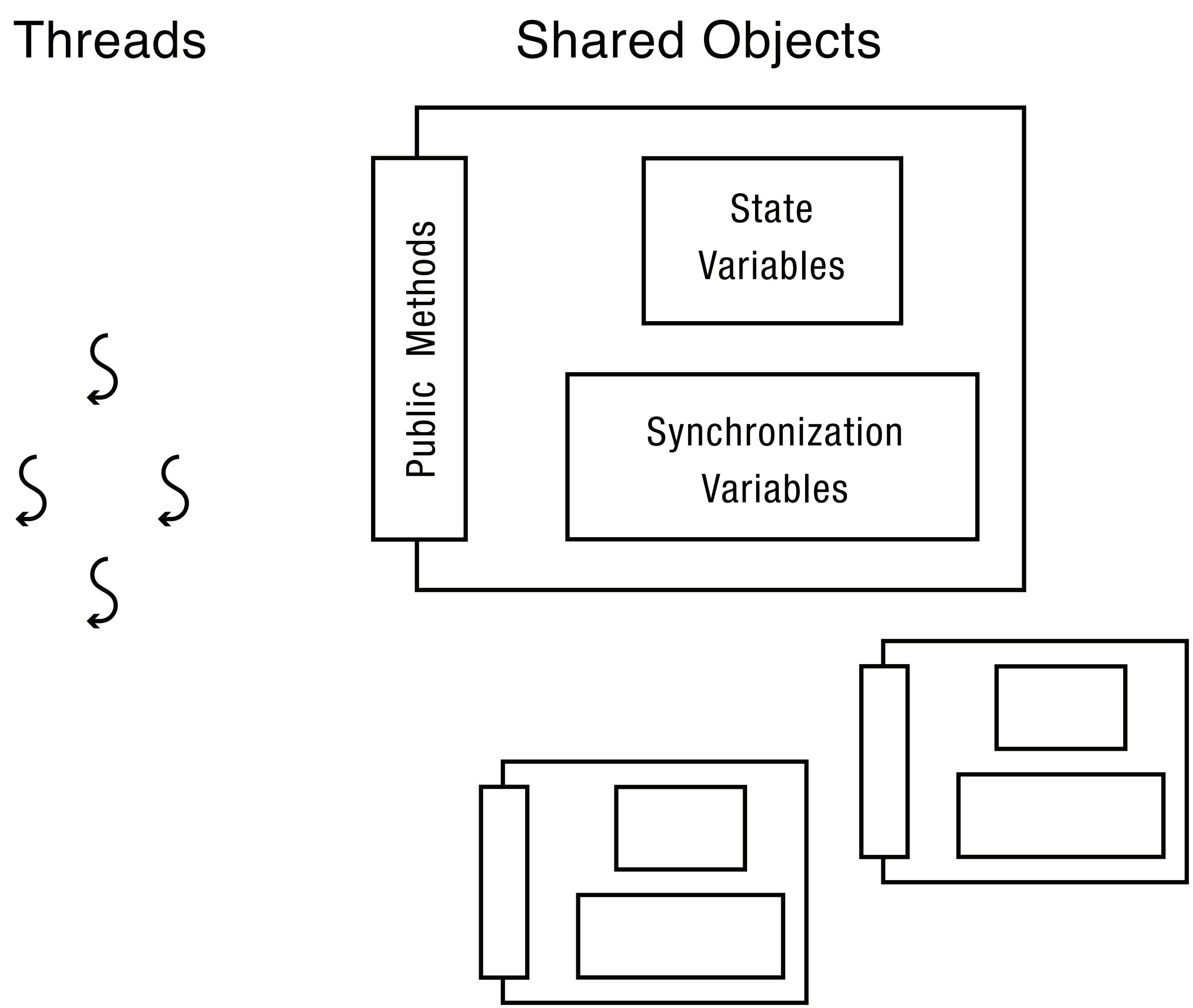
Atomic Instructions

Interrupt Disable Test-and-Set

Hardware

Multiple Processors Hardware Interrupts

Shared Objects and Threads



Definitions

- **Mutual exclusion:** only one thread does a particular thing at a time
- **Critical section:** piece of code that only one thread should execute at any one time

Definitions

- **Lock:** prevent someone from doing something
  - (synchronization primitive that provides mutual exclusion)
  - Lock before entering critical section,
    - before accessing shared data
  - Unlock when leaving,
    - after done accessing shared data
  - Wait if locked (all synchronization involves waiting!)

Locks

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

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

Question: Why only Acquire/Release

- Suppose we add a method to a lock, to ask if the lock is free. Suppose it returns true. Is the lock:
  - Free?
  - Busy?
  - Don’t know?

Too Much Milk, #4

Locks allow concurrent code to be much simpler:

```
lock.acquire();
if milk == 0 {
    milk++
}
lock.release();
```

Lock Example: Malloc/Free

```
char *malloc (n) {
    heaplock.acquire();
    p = allocate memory
    heaplock.release();
    return p;
}
```

```
void free(char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
}
```

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
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Formal Properties of Locks

1. Mutual Exclusion: At most one thread holds the lock. (Safety)
2. Progress: If no thread holds lock, and any thread attempts to acquire lock, then eventually some thread succeeds to acquire lock. (Liveness)
3. Bounded Waiting: If thread T attempts to acquire lock, then there exists a bound on the number of times other threads successfully acquire the lock before T does. (Liveness)

Non-goal for Lock Primitive

- Locks do enforce any order on acquire() called among threads.
- No FIFO ordering

Will this code work?

```
if (p == NULL) {
    lock.acquire();
    if (p == NULL) {
        p = newP();
    }
    lock.release();
}
use p->field1
```

```
newP() {
    p = malloc(sizeof(p));
    p->field1 = …
    p->field2 = …
    return p;
}
```

Example: Bounded Buffer

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

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

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

Case Study:
Thread-Safe Bounded Queue

- Bounded Queue:
  - Fixed size limit
- Used by OS for:
  - IPC, TCP/UDP sockets, I/O requests etc.
  - (since kernel has finite memory)
- Can create many such queues
  - Each with its own lock and state variables

```cpp
#ifndef _TSQUEUE_H_
#define _TSQUEUE_H_
#include "Lock.h"
#include "thread.h"

// TSQueue.h
// Thread-safe queue interface

const int MAX = 10;

class TSQueue {
  // Synchronization variables
    Lock lock;

  // State variables
    int items[MAX];
    int front;
    int nextEmpty;

  public:
    TSQueue();
    ~TSQueue(){};
    bool tryInsert(int item);
    bool tryRemove(int *item);
};
#endif
```

Thread-Safe Bounded Queue: C++

```cpp
#include <assert.h>
#include "TSQueue.h"

// TSQueue.cc
// Thread-safe queue implementation.

// Try to insert an item. If the queue is
// full, return false; otherwise return true.
bool
TSQueue::tryInsert(int item) {
    bool success = false;

    lock.acquire();
    if ((nextEmpty - front) < MAX) {
        items[nextEmpty % MAX] = item;
        nextEmpty++;
        success = true;
    }
    lock.release();
    return success;
}
```

```cpp
// Try to remove an item. If the queue is
// empty, return false; otherwise return true.
bool
TSQueue::tryRemove(int *item) {
    bool success = false;

    lock.acquire();
    if (front < nextEmpty) {
        *item = items[front % MAX];
        front++;
        success = true;
    }
    lock.release();
    return success;
}

// Initialize the queue to empty
// and the lock to free.
TSQueue::TSQueue() {
    front = nextEmpty = 0;
}
```

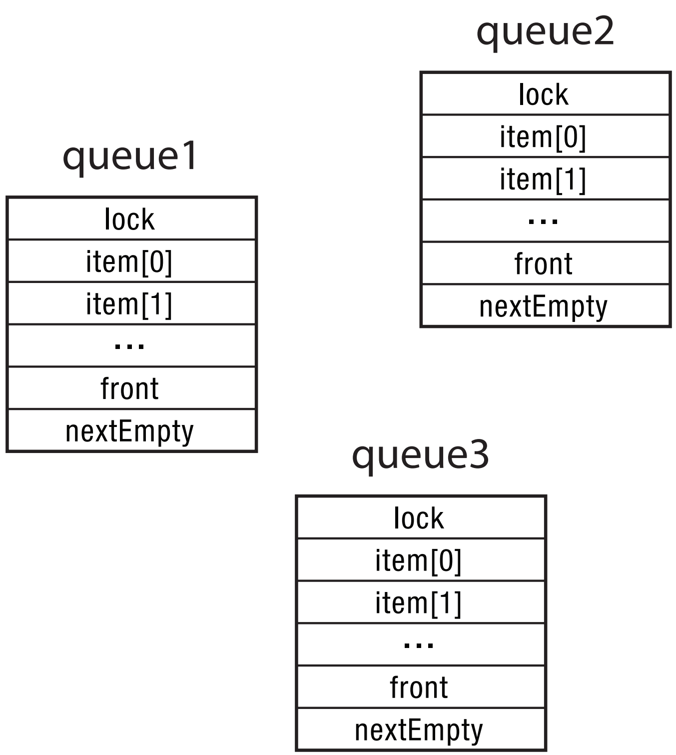
Thread-Safe Bounded Queue: Go

Shared Objects

- A shared object q1 may have many critical sections:
  - q1.tryInsert()
  - q1.tryRemove()
  - q1.length()
- Only one thread will be able to execute these at a given time

- A program may have many shared object instances:
  - q1.tryInsert()
  - q2.tryRemove()
  - q3.tryInsert()

Instances of Shared Object TSQueue



- Each object has their own lock
- These objects can be used concurrently
- But one such object can only be used by one thread at a time

Question

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

- If we poll tryget in a loop, what happens to a thread calling tryput?

CONDITION VARIABLES

Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock
- Wait:
  - Atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

Condition Variable Design Pattern

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

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

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

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

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



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

Example: Bounded Buffer

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

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

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

Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - front <= tail
  - front + MAX >= tail
- These are also true on return from wait
- And at lock release
- Allows for proof of correctness

Pre/Post Conditions

```
methodThatWaits() {
    lock.acquire();
    // Pre-condition: State is consistent

    // Read/write shared state

    while (!testSharedState()) {
        cv.wait(&lock);
    }
    // WARNING: shared state may
    // have changed!  But
    // testSharedState is TRUE
    // and pre-condition is true

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

```
methodThatSignals() {
    lock.acquire();
    // Pre-condition: State is consistent

    // Read/write shared state

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

    // NO WARNING: signal keeps lock

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

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is used to synchronize on 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
- Wait atomically releases lock
  - What if wait, then release?
  - What if release, then wait?

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);
}
```

Java Manual

When waiting upon a Condition, a “spurious wakeup” is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition *should always be waited upon in a loop, testing the state predicate that is being waited for*.

Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
- If need to wait
  - while(needToWait()) { condition.Wait(lock); }
  - Do not assume when you wake up, signaler just ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

Remember the rules

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()

Mesa vs. Hoare semantics

- Mesa
  - Signal puts waiter on ready list
  - Signaler keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaler
  - Nested signals possible!

FIFO Bounded Buffer (Hoare semantics)

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

```
put(item) {
    lock.acquire();
    if ((tail – front) == MAX) {
        full.wait(lock);
    }
    buf[last % MAX] = item;
    last++;
    empty.signal(lock);
  // CAREFUL: someone else ran
    lock.release();
}
```

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

empty/full are condition variables

FIFO Bounded Buffer (Mesa semantics)

- Create a condition variable for every waiter
- Queue condition variables (in FIFO order)
- Signal picks the front of the queue to wake up
- CAREFUL if spurious wakeups!

- Easily extends to case where queue is LIFO, priority, priority donation, …
  - With Hoare semantics, not as easy

FIFO Bounded Buffer (Mesa semantics, put() is similar)

```
get() {
    lock.acquire();
    myPosition = numGets++;
    self = new Condition;
    nextGet.append(self);
    while (front < myPosition
            || front == tail) {
        self.wait(lock);
    }
    delete self;
    item = buf[front % MAX];
    front++;
    if (next = nextPut.remove()) {
        next->signal(lock);
    }
    lock.release();
    return item;
}
```

Initially: front = tail = numGets = 0; MAX is buffer capacity
nextGet, nextPut are queues of Condition Variables

Implementing Synchronization

Concurrent Applications

Semaphores Locks Condition Variables

Interrupt Disable Atomic Read/Modify/Write Instructions

Multiple Processors Hardware Interrupts

Implementing Synchronization

Take 1: using memory load/store

- See too much milk solution/Peterson's algorithm

Take 2:

```
Lock::acquire()
   { disable interrupts }
Lock::release()
   { enable interrupts }
```

Lock Implementation, Uniprocessor

```
func Lock() {
  disableInterrupts()
  if lockState == BUSY {
    waiting.add(myTCB)
    myTCB.state = WAITING
    next = readyList.remove()
    myTCB = switch(myTCB, next)
    myTCB.state = RUNNING
  } else {
    lockState = BUSY
  }
  enableInterrupts()
}
```

```
func Unlock() {
  disableInterrupts()
  if !waiting.Empty() {
    next = waiting.remove()
    next.state = READY
    readyList.add(next)
  } else {
    lockState = FREE
  }
  enableInterrupts()
}
```

Multiprocessor

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

Spinlocks

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 (testAndSet(&lockValue) == BUSY)
      ;
}
Spinlock::release() {
   lockValue = FREE;
  memorybarrier();
}
```

How many spinlocks?

- Various data structures
  - Queue of waiting threads on lock X
  - Queue of waiting threads on lock Y
  - List of threads ready to run
- One spinlock per kernel?
  - Bottleneck!
- Instead:
  - One spinlock per lock
  - One spinlock for the scheduler ready list
    - Per-core ready list: one spinlock per core

What thread is currently running?

- Thread scheduler needs to find the TCB of the currently running thread
  - To suspend and switch to a new thread
  - To check if the current thread holds a lock before acquiring or releasing it
- On a uniprocessor, easy: just use a global
- On a multiprocessor, various methods:
  - Compiler dedicates a register (e.g., r31 points to TCB running on the this CPU; each CPU has its own r31)
  - If hardware has a special per-processor register, use it
  - Fixed-size stacks: put a pointer to the TCB at the bottom of its stack
    - Find it by masking the current stack pointer

Lock Implementation, Multiprocessor

```
Lock::acquire() {
    disableInterrupts();
    spinLock.acquire();
    if (value == BUSY) {
        waiting.add(myTCB);
        suspend(&spinlock);
    } else {
        value = BUSY;
    }
    spinLock.release();
   enableInterrupts();
}
```

```
Lock::release() {
    disableInterrupts();
    spinLock.acquire();
    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler->makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
    enableInterrupts();
}
```

Compare Implementations

```
Semaphore::P() {
    disableInterrupts();
    spinLock.acquire();
    if (value == 0) {
        waiting.add(myTCB);
        suspend(&spinlock);
    } else {
        value--;
    }
    spinLock.release();
  enableInterrupts();
}
```

```
Semaphore::V() {
    disableInterrupts();
    spinLock.acquire();
    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler->makeReady(next);
    } else {
        value++;
    }
    spinLock.release();
    enableInterrupts();
}
```

Lock Implementation, Multiprocessor

```
Sched::suspend(SpinLock *lock) {
   TCB *next;

   disableInterrupts();
   schedSpinLock.acquire();
   lock−>release();
   myTCB−>state = WAITING;
   next = readyList.remove();
   thread_switch(myTCB, next);
   myTCB−>state = RUNNING;
   schedSpinLock.release();
   enableInterrupts();
}
```

```
Sched::makeReady(TCB *thread) {

   disableInterrupts ();
   schedSpinLock.acquire();
   readyList.add(thread);
   thread−>state = READY;
   schedSpinLock.release();
   enableInterrupts();
}
```

Lock Implementation, Linux

- Most locks are free most of the time
  - Why?
  - Linux implementation takes advantage of this fact
- Fast path
  - If lock is FREE, and no one is waiting, two instructions to acquire the lock
  - If no one is waiting, two instructions to release the lock
- Slow path
  - If lock is BUSY or someone is waiting, use multiproc impl.

- User-level locks
  - Fast path: acquire lock using test&set
  - Slow path: system call to kernel, use kernel lock

Lock Implementation, Linux

```
struct mutex {
 /* 1: unlocked ; 0: locked;
    negative : locked,
    possible waiters */
 atomic_t count;
 spinlock_t wait_lock;
 struct list_head wait_list;
};
```

```
// atomic decrement
// %eax is pointer to count
lock decl (%eax)
jns 1f // jump if not signed
          // (if value is now 0)
call slowpath_acquire
1:
```

Semaphores

- Semaphore has a non-negative integer value
  - P() atomically waits for value to become > 0, then decrements
  - V() atomically increments value (waking up waiter if needed)
- Semaphores are like integers except:
  - Only operations are P and V
  - Operations are atomic
    - If value is 1, two P's will result in value 0 and one waiter
- Semaphores are useful for
  - Unlocked wait: interrupt handler, fork/join

Semaphore Bounded Buffer

```
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();
}
```

Initially: front = last = 0; MAX is buffer capacity

mutex = 1; emptySlots = MAX; fullSlots = 0;

Implementing Condition Variables using Semaphores (Take 1)

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

Implementing Condition Variables using Semaphores (Take 2)

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

Implementing Condition Variables using Semaphores (Take 3)

```
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
   }
}
```

Communicating Sequential Processes (CSP/Google Go)

- A thread per shared object
  - Only thread allowed to touch object's data
  - To call a method on the object, send thread a message with method name, arguments
  - Thread waits in a loop, get msg, do operation
- No memory races!

Example: Bounded Buffer

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

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

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

empty/full are condition variables

Bounded Buffer (CSP)

```
while (cmd = getNext()) {
  if (cmd == GET) {
    if (front < tail) {
      // do get
      // send reply
      // if pending put, do it
      // and send reply
    } else
      // queue get operation
  } else { // cmd == PUT
    if ((tail – front) < MAX) {
      // do put
      // send reply
      // if pending get, do it
      // and send reply
    } else
      // queue put operation
  }
```

Locks/CVs vs. CSP

- Create a lock on shared data
  = create a single thread to operate on data
- Call a method on a shared object
  = send a message/wait for reply
- Wait for a condition
  = queue an operation that can't be completed just yet
- Signal a condition
  = perform a queued operation, now enabled

Remember the rules

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()