

Reading Reference:
Textbook: Chapter 4

THREAD SYNCHRONIZATION

Tanzir Ahmed
CSCE 313 Spring 2020

Goals for This Lecture

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

Note: Some slides and/or pictures in the following are adapted and/or used verbatim from slide content in Silberschatz, Galvin, and Gagne (2014), Anthony D. Joseph (2014 Berkeley), Tom Anderson (2014 UW), Bettati (2014 TAMU), Gu (2014 TAMU), Tyagi (2016 TAMU)

Outline

- This lecture is a bit up-side down
- First, we learn how to **USE** Locks, Semaphores, Conditions
- Then, we will see how to **IMPLEMENT** Locks
 - *Programming Assignment 4 will ask you to implement Semaphores*
- The reason for this is to prepare you for PAs before exploring the nitty-gritty details

Atomic Operations

- **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*
- Each instruction in the Instruction Set is atomic
 - *An instruction fully finishes before the current process/thread can be preempted/interrupted*

Synchronization Variable – Lock to Provide Mutual Exclusion

- First step towards making shared data thread-safe
- The idea is to make instructions atomic to stop context switch from happening
- General Idea:

```
Lock();  
load r, x  
add r, r, 1  
store x, r  
Unlock();
```

Critical
Section
(No Context
Switch)

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

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

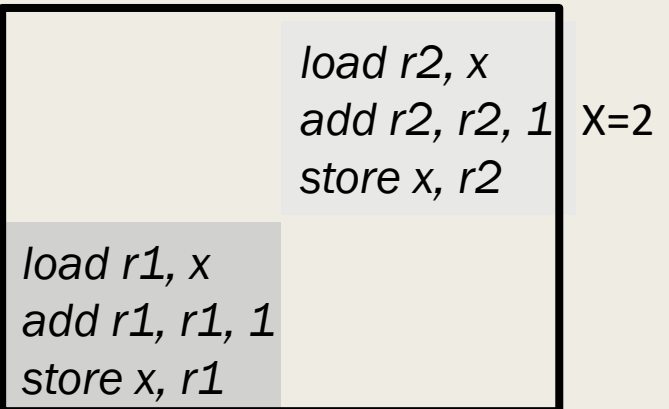
X=2

Synchronization Variable – Lock to Provide Mutual Exclusion

- First step towards making shared variable thread-safe
- The idea is to make instructions atomic to stop context switch from happening
- General Idea:

```
Lock();  
load r, x  
add r, r, 1  
store x, r  
Unlock();
```

Critical
Section
(No Context
Switch)



Mutex in C++ to Thread Safety

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

void func (int * p, int x, mutex* m) {
    // increment *p x times
    m->lock();
    for (int i=0; i<x; i++){
        *p = *p + 1;
    }
    m->unlock();
}

int main(int ac, char** av) {
    int data = 0;
    int times = atoi (av [1]);
    mutex m;

    thread t1 (func, &data, times, &m);
    thread t2 (func, &data, times, &m);

    t1.join(); // pauses until first finishes
    t2.join(); // pauses until second finishes
    cout << "data = " << data << endl;
}
```

Critical Section

2. Lock it before the "critical section"

3. Unlock after "critical section"

1. Create a mutex and share it across the threads

```
osboxes@osboxes:~/ $ ./a.out 10000000
data = 20000000
osboxes@osboxes:~/ $ ./a.out 10000000
data = 20000000
osboxes@osboxes:~/ $ ./a.out 10000000
data = 20000000
```

Mutex in C++ - Finer Locking

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
```

```
void func (int * p, int x, mutex* m) {
    // increment *p x times
    for (int i=0; i<x; i++){
```

```
        m->lock();
```

Critical
Section

```
        *p = *p + 1;
```

```
        m->unlock();
```

```
    }
```

```
}
```

```
int main(int ac, char** av) {
    int data = 0;
    int times = atoi (av [1]);
    mutex m;
```

```
    thread t1 (func, &data, times, &m);
    thread t2 (func, &data, times, &m);
```

```
    t1.join(); // pauses until first finishes
    t2.join(); // pauses until second
                finishes
    cout << "data = " << data << endl;
```

This is more
fine-grained,
produces the
same correct
result

- The previous approach puts entire thread under lock
- This effectively makes the threads completely sequential
 - *No threading/interleaving happens at all*
- This is “coarse-grained” locking
- You can make locking “finer” (see the example on left)
- The result is correct in both cases
- The choice would depend on other factors
 - *Locking and unlocking usually take time*

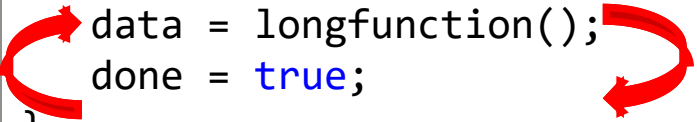
```
osboxes@osboxes:~/ $ ./a.out 10000000
data = 20000000
osboxes@osboxes:~/ $ ./a.out 10000000
data = 20000000
osboxes@osboxes:~/ $ ./a.out 10000000
data = 20000000
```


Producer-Consumer Synchronization

- Just thread-safety is not adequate for many problems
- For instance, there is no order between operations in Thread 1 & 2 in the previous example:
- What if you want to Thread 1 to finish completely before you run Thread 2?
 - *Probably, Thread 1 produces a result that Thread 2 uses*
- This is called Producer-Consumer problem
- You need another synchronization primitive called “conditional variable”
 - *Because the naïve approach does not work for “instruction reordering”*

```
bool done = false;
int data = -1;
Thread1 (){
    // takes a long time
    data = longfunction();
    done = true;
}

// thread 2's function
Thread2 (){
    while (!done); // wait
    int newdata = compute (data);
}
```

A diagram illustrating instruction reordering. Two red curved arrows are shown. The first arrow starts at the line 'data = longfunction();' in the Thread1 function and points to the line 'done = true;'. The second arrow starts at the line 'done = true;' and points back to the line 'data = longfunction();', indicating that the compiler might reorder these two statements.

Producer-Consumer

```
bool done = false;
int data = -1;
condition_variable cv;
mutex m;
void Thread1 (){
    data = longfunction();
    m.lock();
    done = true;
    m.unlock();
    cv.notify_all();
}
```

```
void Thread2 (){
    unique_lock<mutex> l (m);
    cv.wait (l, []{return done == true;});

    data = compute (data);
    cout << "Data is: " << data << endl;

    l.unlock();
}
```

- Step 1: Declare a condition variable and a mutex
- Step 2: The producer
 - *produces data*
 - *Set the predicate variable (done = true) under lock*
 - *Calls notify_one/all() on the condition to wake up the consumers*

Producer-Consumer

```
bool done = false;
int data = -1;
condition_variable cv;
mutex m;
void Thread1 (){
    data = longfunction();
    m.lock();
    done = true;
    m.unlock();
    cv.notify_all();
}
```

```
void Thread2 (){
    unique_lock<mutex> l (m);
    cv.wait (l, []{return done == true;});

    data = compute (data);
    cout << "Data is: " << data << endl;

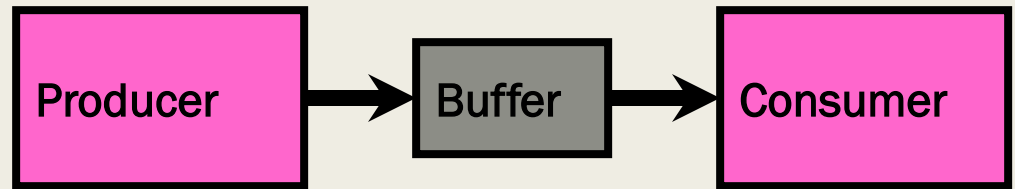
    l.unlock();
}
```

- Step 3: The consumer(s) do the following:
 - *Calls wait() on the condition*
 - *Wait() needs a “wrapped” lock (as unique_lock) and a predicate function*
 - The “wrapper” also locks the lock
 - In the above I just used a “lambda” function for the predicate instead of defining it elsewhere
 - *Consume data*
 - *Unlock the lock*

A Few More Points on Wait()

- First, the `cv.wait()` function wakes up the waiting threads **sequentially**, NOT all at once
 - *This provides a Critical Section until `unlock()` function is called on the lock later on*
 - *As a result, every awake thread gets to do something without running into race condition*
- The `wait()` function internally unlocks the lock before going to `sleep()`
 - *Otherwise, even the producer cannot produce, which requires the lock itself*
 - *However, when returning from `wait()`, the lock is grabbed again and that causes the “one-at-a-time” safety*
- **Spurious wake ups** are possible
 - *Waiting threads may wake up even when the predicate is not true*
 - *That’s why using the “predicate” is absolute necessity*

Producer-consumer with a bounded buffer



■ Problem Definition

- *Producer puts things into a shared buffer*
- *Consumer takes them out*
- *Need synchronization to coordinate producer/consumer*

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

- *Need to synchronize access to this buffer*
- *Producer needs to wait if buffer is full*
- *Consumer needs to wait if buffer is empty*

■ Example: Coke machine

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



Correctness constraints for solution

■ Correctness Constraints:

- *Consumer must wait for producer to fill slots, if empty (scheduling constraint)*
- *Producer must wait for consumer to make room in buffer, if all full (scheduling constraint)*
- *Only one thread can manipulate buffer queue at a time (mutual exclusion using lock)*

■ Nice Features

- *Inherent rate control:*
 - Consumer is limited by Production Rate
 - Producer is limited by buffer size and consequently Consumption Rate

■ Application is universal

- *Networks, Inter Process Communication etc.*

Implementing BoundedBuffer

The following is a skeleton for the BoundedBuffer, that is:

Unsafe: because multiple threads can push into and pop from it simultaneously, leading to race condition

Unbounded: because nothing stops from the buffer from growing to infinity when producer thread(s) is(are) much faster than the consumer(s)

We need to implement thread-safety and bounds on overflow and underflow

- We also want **efficient** **wait until** the overflow/underflow conditions are gone, so Producer/Consumer do not have to “retry”

```
/* a unsafe and unbounded buffer */
/* add necessary changes */
class BoundedBuffer{
    queue<vector<char>> q;
    int maxcap; // max capacity
    // add sync. variables here
public:
    BoundedBuffer (int _cap) :maxcap(_cap){
    vector<char> pop (){
        vector<char> data = q.front();
        q.pop();
        return data;
    }
    void push (vector<char> data){
        q.push (data);
    }
};
```

Let's Implement BoundedBuffer

- Note that we need to implement a sophisticated wait facility
- The user will call pop() or push() only once, your function will wait until the time is right for that operation
 - *If the buffer is full, make the push() function wait until there is space (because somebody popped something out)*
 - *If the buffer is empty, pop() function waits until somebody pushes something*
 - Of course, if many pop() functions are waiting and only 1 push happens, only 1 pop() function will get out of sleep
 - Similar reasoning for push functions as well
- Here is a naïve implementation that will NOT work:

```
vector<char> BoundedBuffer::pop() {  
    while (buffer.size () == 0)  
        sleep(n) ;  
  
    // now consume  
}
```

There is a race condition

BoundedBuffer – Take 2

```
vector<char> BoundedBuffer::pop() {  
    mtx.Lock(); // mtx is a mutex, defined  
                as class member variable  
    while (buffer.size () == 0)  
        sleep(n);  
    mtx.Unlock();  
    // now consume  
}
```

No unwanted switches now. But, sleeping with
Mutex Locked!!! Even the producers cannot
replenish buffer while this thread is waiting

Take 3

```
Vector<char> BoundedBuffer::pop() {  
    mtx.Lock();  
    while (buffer.size () == 0) {  
        mtx.Unlock();  
        // 1. thread switch???  
        sleep(n);  
        // 2. thread switch???  
        mtx.Lock();  
    }  
    mtx.Unlock();  
}
```

Producer
thread
runs, you
just missed
it

Another
consumer
takes it,
again missed

Condition Variables for BB

- Problems with the previous solution:
 - *Misses wakeups*
 - *Does busy-looping (takes CPU cycles away from Producers)*
 - *Sleep(x) is dependent on x. What if $x=1\text{sec}$, and data is produced every 5ms? The consumer is not very responsive!!*
 - *What if $x=1\text{ms}$ and data are produced every 10 sec. The consumer is taking up unnecessary CPU cycles, possibly away from the Producer*
 - *In summary, a synchronous solution is not elegant*
- This is exactly where condition variables are absolutely essential

BoundedBuffer using Condition Variables

```
class BoundedBuffer
{
private:
    int cap;
    queue<vector<char>> q; /*queue where each item is a vector
of char so that we can store variable length data */

    /*mutex to protect the queue from simultaneous producer
accesses or simultaneous consumer accesses */
    mutex mtx;

    /*cond. that tells the consumers that some data is there */
    condition_variable data_avail;
    /*cond. that tells the producers some slot is available */
    condition_variable slot_avail;
```

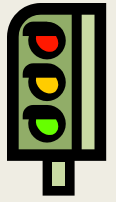
Condition Variables for BB – A working (almost) solution

```
vector<char> BoundedBuffer::pop(){
    unique_lock<mutex> l (mtx);
    // keep waiting as long as q.size() == 0
    data_avail.wait (l, [this]{return q.size() > 0;});
    // pop from the queue
    vector<char> data = q.front();
    q.pop();
    // notify any potential producer(s)
    slot_avail.notify_one ();
    // unlock the mutex so that others can go in
    l.unlock();
    return data;
}
```

1. Atomically unlocks the mutex and goes to sleep
2. Returns when condition set and with mutex locked
3. Return does not mean buffer has something
4. Condition is set by the Producer
5. Only 1 consumer will consume, others will wait for the next Production

Summary So Far

- We have learned about 2 very important synchronization primitives:
 - *Mutexes*
 - *Condition variables*
- We should be able to solve most synchronization problems using these two
 - *Let us try a few synchronization problems to convince ourselves*
- There are other more convenient primitives that can be built using Mutex and Condition variable. They can be very convenient and developer friendly
 - *Example: Semaphores, Synchronized methods in Java*



Semaphores

- Semaphores are a kind of generalized locks
 - *First defined by Dijkstra in late 60s*
 - *Main synchronization primitive used in original UNIX*
- The main purpose is to make our code simpler from what we saw before using condition variables and mutex
 - *We need to use a mutex and a condition variable just to indicate one event*
 - *This makes our code complicated*
- Definition: a Semaphore has a **non-negative integer value** and supports the following two **atomic** operations:
 - *P(): Waits until value > 1, then decrements it by 1*
 - Think of this as the wait() operation, or the lock() operation
 - *V(): Increments value by 1*
 - May wake up a waiting P, if any
 - Think of this as the signal() operation, or the unlock() operation

Semaphore Implementation

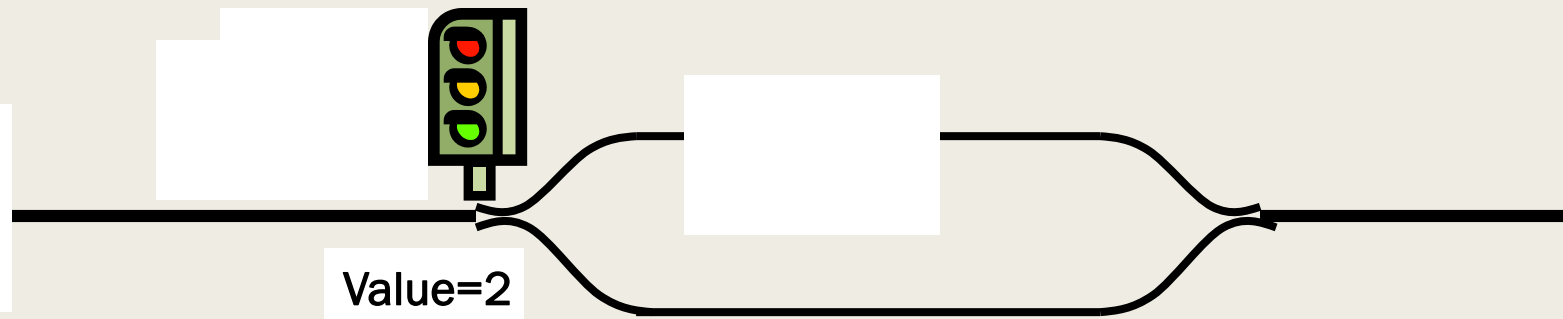
```
class Semaphore{
private:
int value;
mutex m;
condition_variable cv;
public:
Semaphore (int _v):value(_v){}
void P(){
    unique_lock<mutex> l (m);
    // wait until the value is positive
    cv.wait (l, [this]{return value > 0;});
    // now decrement
    value --;
    l.unlock (); // this is optional
}
void V(){
    unique_lock<mutex> l (m);
    value ++;
    cv.notify_one();
    l.unlock(); // this is optional
}
};
```

Wait until value > 0, so it can be decremented. Then decrement it.

Always notify on the way out, but do not notify_all() to avoid spurious wake ups

Semaphores Like Integers Except

- No negative values
- Only operations allowed are P and V – can't read or write value, except to set it initially
- Operations must be atomic
 - *Two P's together can't decrement value below zero*
 - *Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time*
- Semaphore from railway analogy
 - *Here is a semaphore initialized to 2 for resource control:*



Two Uses of Semaphores

■ Mutual Exclusion (initial value = 1)


- Also called “Binary Semaphore”.
- Can be used for mutual exclusion:

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

■ Scheduling Constraints (initial value = 0)

- Allow thread 1 to wait for a signal from thread 2, i.e., thread 2 **schedules** thread 1 when a given **constrained** is satisfied
- Example: suppose you had to implement ThreadJoin which must wait for a thread to terminate:

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

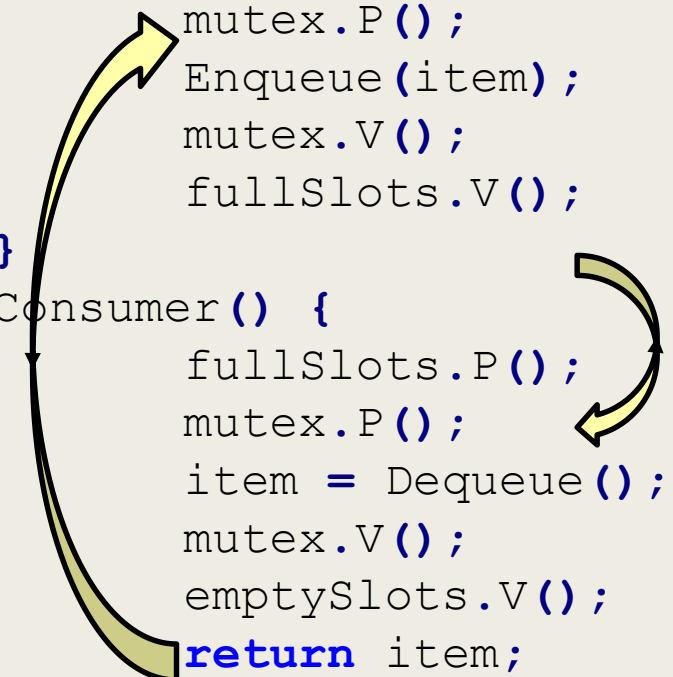


Full Solution to Bounded Buffer

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

Producer(item) {
    emptySlots.P();          // Wait until space
    mutex.P();              // Wait until machine free
    Enqueue(item);
    mutex.V();
    fullSlots.V();          // Notify there is more coke
}

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



Thoughts

■ Why asymmetry?

- *Producer does: `emptySlots.P()`, `fullSlots.V()`*
- *Consumer does: `fullSlots.P()`, `emptySlots.V()`*

Decrease # of
empty slots

Increase # of
occupied slots

Decrease # of
occupied slots

Increase # of
empty slots

One is creating space, the other is filling space

More Thoughts

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

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

More Thoughts

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

BEFORE

```
Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

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

AFTER

```
Producer(item) {
    mutex.P();
    emptySlots.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

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

More Thoughts

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

BEFORE

```
Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

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

AFTER

```
Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    fullSlots.V();
    mutex.V();
}

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

More Thoughts

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

```
Producer(item) {  
    emptySlots.P();  
    mutex.P();  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();  
}
```

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


Some Classic Synchronization Problems

- Reader-Writer problem
- Dining-Philosopher Problem
- Monkey crossing bridge

Readers-Writers Problem

- Many processes share a database
- Some processes write to the database
- Only one writer can be active at a time
- Any number of readers can be active simultaneously
- First Readers-Writers Problem:
 - *Readers get higher priority, and do not wait for a writer*
- Second Readers-Writers Problem:
 - *Writers get higher priority over Readers waiting to read*
 - Courtois et al.

Readers-Writers

```
Semaphore wr1 (1); // to
exclude readers and writers
int rcount = 0; // to count
how many active readers
Semaphore mutex(1); // to
protect rcount variable
```

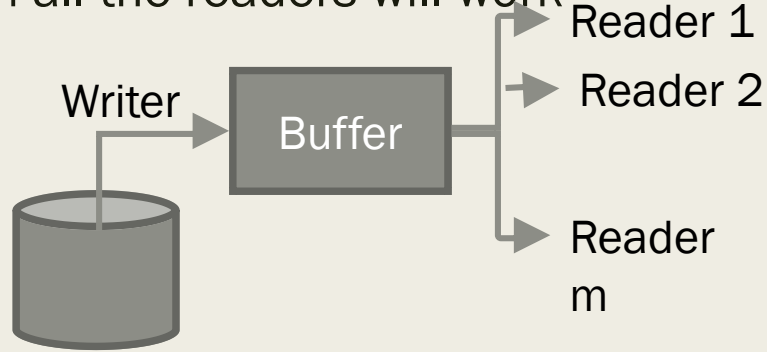
```
void Writer () {
    do {
        wr1.P();
        /*writing is performed*/
        wr1.V();
    }while(true);
}
```

```
void Reader ()
{
    do {
        //maintain rcount, but safely
        mutex.P();
        rcount++;
        if (rcount == 1)
            wr1.P(); // wait for writer
        mutex.V();
        //now going in with the lock
        at hand, no writer here
        /*reading is performed*/
        // maintain rcount, but
        safely
        mutex.P();
        rcount--;
        if (rcount == 0)
            wr1.V(); //last reader, so
            give up lock
        mutex.V();
    }while(true);
}
```

Readers-Writers Notes

- If there is a writer
 - *First reader blocks on **wrl***
 - *Other readers block on **mutex***
- Once a writer exists, all readers get to go through
 - *Which reader gets in first?*
- The last reader to exit signals a writer
 - *If no writer, then readers can continue*
- If readers and writers waiting on **wrl**, and writer exits
 - *Who gets to go in first?*
- Why doesn't a writer need to use **mutex**?

How about a more Specific Reader-Writer problem?

- **Problem:** One writer thread in charge of reading from an external source (e.g., disk or network) and putting data in a buffer
 - m reader threads who consume from the buffer
 - First, the writer will populate data, then all the readers will work
 - Does this problem look familiar?
 - BoundedBuffer with buffer size 1
- 
- The diagram illustrates the Reader-Writer problem setup. On the left, a cylinder represents an external source. An arrow labeled 'Writer' points from this source to a rectangular box labeled 'Buffer'. From the right side of the 'Buffer' box, three arrows branch out to the right, labeled 'Reader 1', 'Reader 2', and 'Reader m' respectively, representing multiple reader threads consuming data from the buffer.
- Let us see some similar examples in action

Monkeys-Crossing-River Problem

- Several monkeys trying to cross a river on a thin rope
 - *The rope can carry at most M monkeys at a time*
- You have to fill-up the following functions used by each monkey:

```
void monkey(int monkeyid) {  
    WaitUntilSafe();           // define  
    CrossRiver(monkeyid);      // given  
    DoneWithCrossing();        // define  
}
```

- Will 1 semaphore suffice?

Follow-Up

- What if now complicate the problem a little bit
- New Restrictions:
 - *There are 2 directions the monkeys are moving (east and west)*
 - *All monkeys executing in CrossRiver() are heading in the same direction.*
- Now, the function looks like the following:

```
void monkey(int monkeyid, int dir) {  
    WaitUntilSafe(dir);           // define  
    CrossRavine(monkeyid, dir);   // given  
    DoneWithCrossing(dir);        // define  
}
```

Monkeys Crossing with Directions

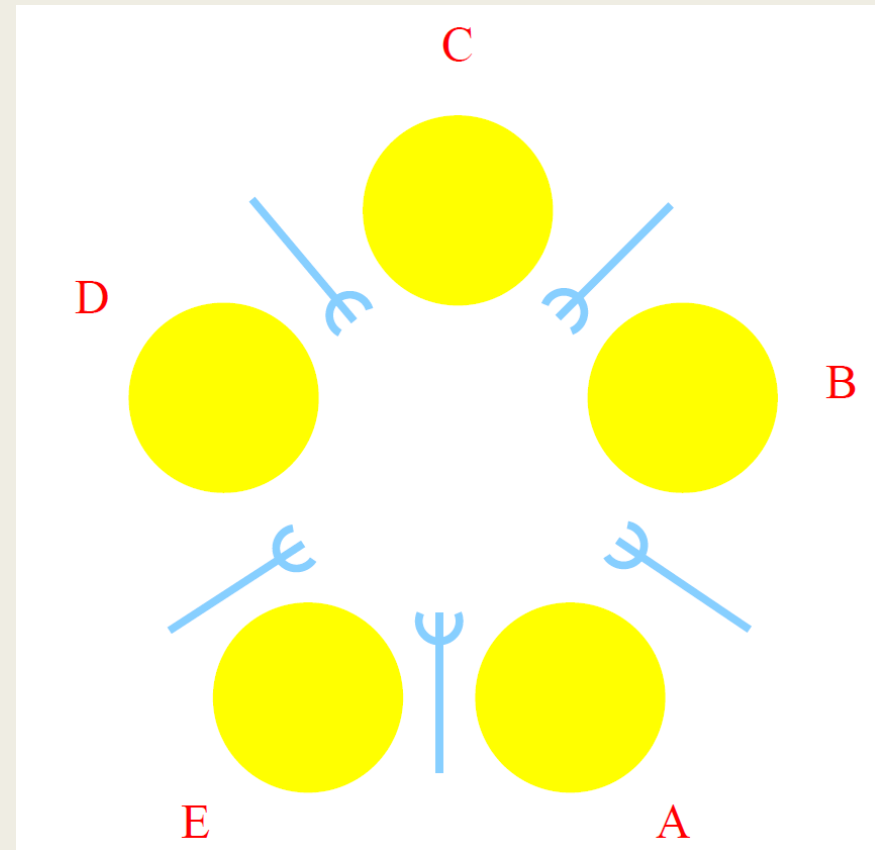
```
int monkey_count[2] = {0,0}; //counter for each direction
Semaphore mutex[2] = {1,1}; //mutexes to protect counters
Semaphore capacity(M); //ensure maximum M monkeys on rope
Semaphore dirmtx(1); // direction mutex used by the first
monkey from each direction
```

```
void WaitUntilSafe (int dir)
{
    mutex[dir].P();
    monkey_count[dir]++;
    if (monkey_count[dir]==1)
        dirmtx.P();
    mutex[dir].V();
    capacity.P();
}
```

```
void DoneWithCrossing(int dir)
{
    mutex[dir].P();
    monkey_count[dir]--;
    if (monkey_count[dir] == 0)
        //last mon:release dirmtx
        dirmtx.V();
    mutex[dir].V();
    capacity.V();
}
```


Dining-Philosopher Problem

- There are a number of philosophers, who either 1) think or 2) eat spaghetti
- Each philosopher needs 2 forks to eat
- Have to avoid deadlock or starvation
 - *Every philosopher grabbed 1 fork to the left (or right)*
- This is left for your own study



Definitions and Quick Recap

- **Synchronization:** using atomic operations to ensure cooperation between threads
 - *For now, only loads and stores are atomic*
- **Critical Section:** piece of code that only one thread can execute at once
- **Mutual Exclusion:** ensuring that only one thread executes critical section
 - *One thread excludes the other while doing its task*
 - *Critical section and mutual exclusion are two ways of describing the same thing*

More Definitions

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



Implementing a Lock

- How can we build multi-instruction atomic operations?
 - *Recall: dispatcher gets control in two ways.*
 - Internal: Thread does something to relinquish the CPU
 - External: Interrupts cause dispatcher to take CPU
 - *On a uniprocessor, can avoid context-switching by:*
 - Avoiding internal events
 - Preventing external events by disabling interrupts

- Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }
```

```
LockRelease { enable Ints; }
```

Naïve use of Interrupt Enable/Disable: Problems

- Can't let user do this! Consider following:

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

- Real-Time system—no guarantees on timing!
 - *Critical Sections might be arbitrarily long*
- What happens with I/O or other important events?
 - *“Reactor about to meltdown. Help?”*



Better Implementation of Locks

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

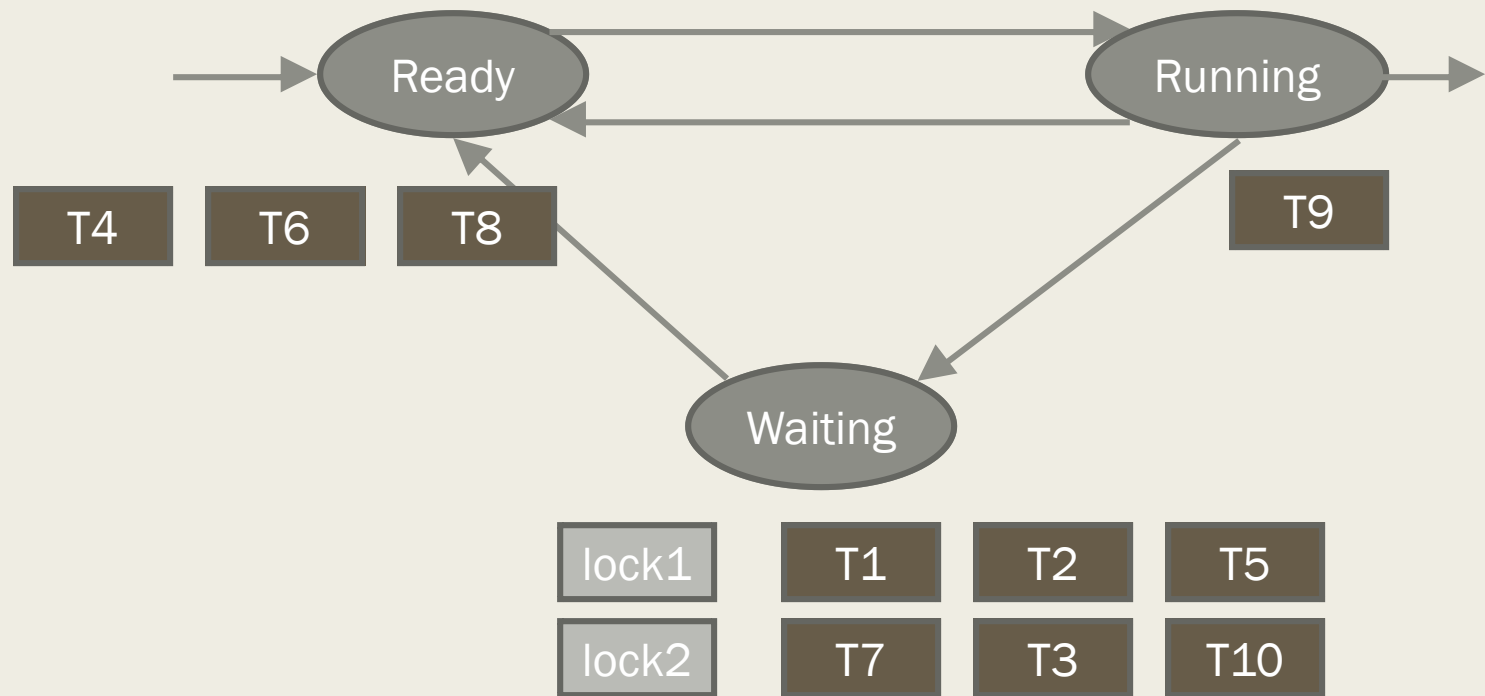
```
int value = FREE;
```



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

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

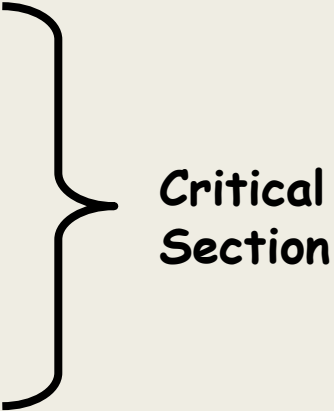
Over-All Idea



New Lock Discussion

- Disable interrupts: avoid interrupting between checking lock value (in Acquire()) and setting it (in Release())
 - *Otherwise two threads could think that they both have lock*

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

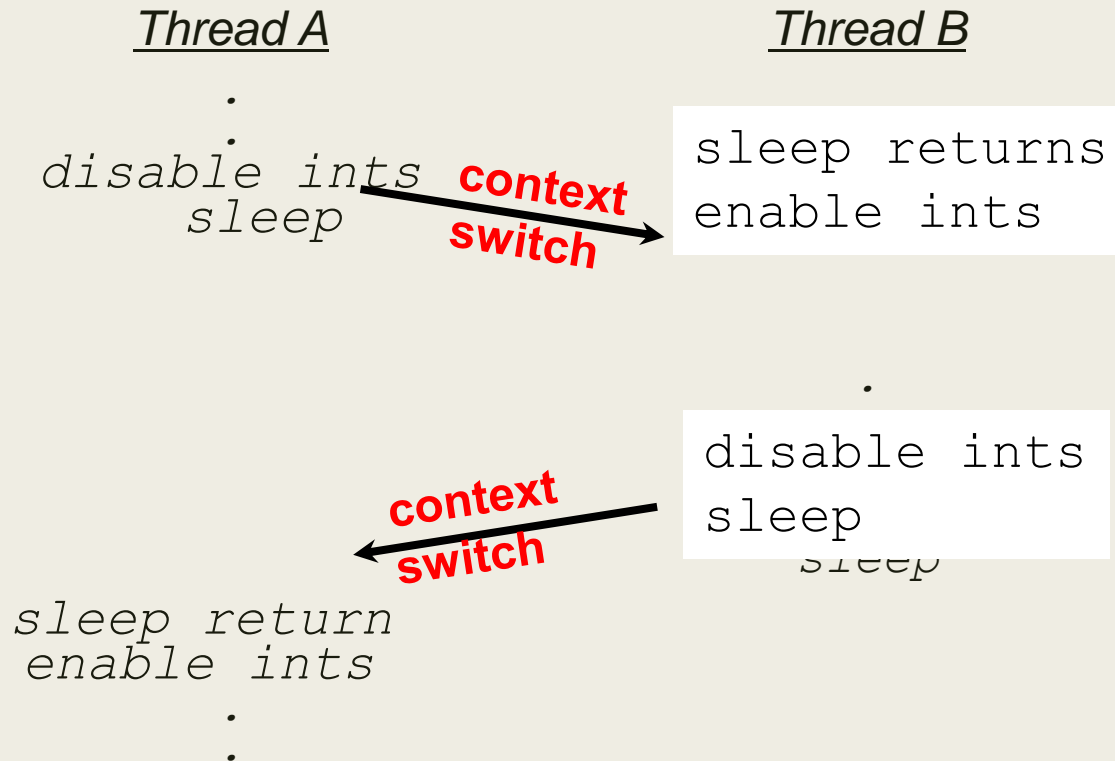


Critical
Section

- Note: unlike previous solution, critical section very short
 - *User of lock can take as long as they like in their own critical section*
 - *Critical interrupts taken in time*

Who Enables Interrupts then?

- 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



An Execution Trace

T1	T2	T3	Interrupt	Wait Queue for this lock	Ready Queue
Acquire()			Enabled		
	Acquire()		Disabled	T2	
CS of T1			Disabled	T2	
CS of T1			Disabled	T2	
Release()	Acquire() returns		Enabled		T2
		Acquire()	Disabled	T3	
	CS of T2		Disabled	T3	
	Release()		Enabled		T3

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;  
}
```

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

Enable Position
Enable Position



- Before putting thread on the wait queue?
 - Thread switch to Release, which checks the queue does not find anybody and not wake up thread until next lock acquire/release
- After putting the thread on the wait queue
 - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
 - Misses wakeup and still holds lock (deadlock!) while sleeping

Summary So Far

- Locks showed so far require Interrupts enable/disable
 - *Separate lock variable reduces Interrupt Disabled duration*
 - *But that duration can still be intolerable*
- We need a better lock implementation
- Things are going towards the “usual suspect” – We are going to need **hardware support**

Atomic Read-Modify-Write instructions

- Problems with interrupt-based lock solution:
 - *Can miss critical Interrupts while INT disabled*
 - *Can't give lock implementation to users*
 - *Doesn't work well on multiprocessor*
 - *Disabling interrupts on all processors requires messages and would be very time consuming*
- Alternative: atomic instructions added to the instruction set
 - *These instructions read a value from memory and write a new value atomically*
 - *Hardware is responsible for implementing this correctly*
 - on uniprocessors (not too hard)
 - and multiprocessors (requires help from cache coherence protocol)

Examples of Read-Modify-Write

- `test&set (&address){/*most architectures*/
 result = M[address];
 M[address] = 1;
 return result;
}`
- `swap (&address, register) { /* x86 */
 temp = M[address];
 M[address] = register;
 register = temp;
}`

Implementing Locks with test&set

```
int value = 0; // Free
Acquire() {
    while (test&set(value)); // while busy
}
Release() {
    value = 0;
}

test&set (&address) {
    result = M[address];
    M[address] = 1;
    return result;
}
```

Explanation:

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

Problem: Busy-Waiting for Lock

- Positives for this solution
 - *Machine can receive interrupts*
 - *User code can use this lock*
 - *Works on a multiprocessor*
- Negatives
 - *Inefficient: busy-waiting thread will consume cycles*
 - *Waiting thread may take cycles away from thread holding lock!*
 - **Priority Inversion:** *If busy-waiting thread has higher priority than thread holding lock \Rightarrow no progress!*
- Priority Inversion problem with original Martian rover



Better Locks using test&set

- Can we build test&set locks without busy-waiting?
 - *Can't entirely, but can minimize!*
 - *Idea: only busy-wait to atomically check lock value*

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

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

Note: sleep has to be sure to reset the guard variable

Locks using test&set vs. Interrupts

- Compare to “disable interrupt” solution

```
int value = FREE;
```

```
Acquire() {
```

```
    disable interrupts;
```

```
    if (value == BUSY) {
```

```
        put thread on wait queue;
```

```
        Go to sleep();
```

```
        // Enable interrupts?
```

```
    } else {
```

```
        value = BUSY;
```

```
    }
```

```
    enable interrupts;
```

```
}
```

```
Release() {
```

```
    disable interrupts;
```

```
    if (anyone on wait queue) {
```

```
        take thread off wait queue
```

```
        Place on ready queue;
```

```
    } else {
```

```
        value = FREE;
```

```
    }
```

```
    enable interrupts;
```

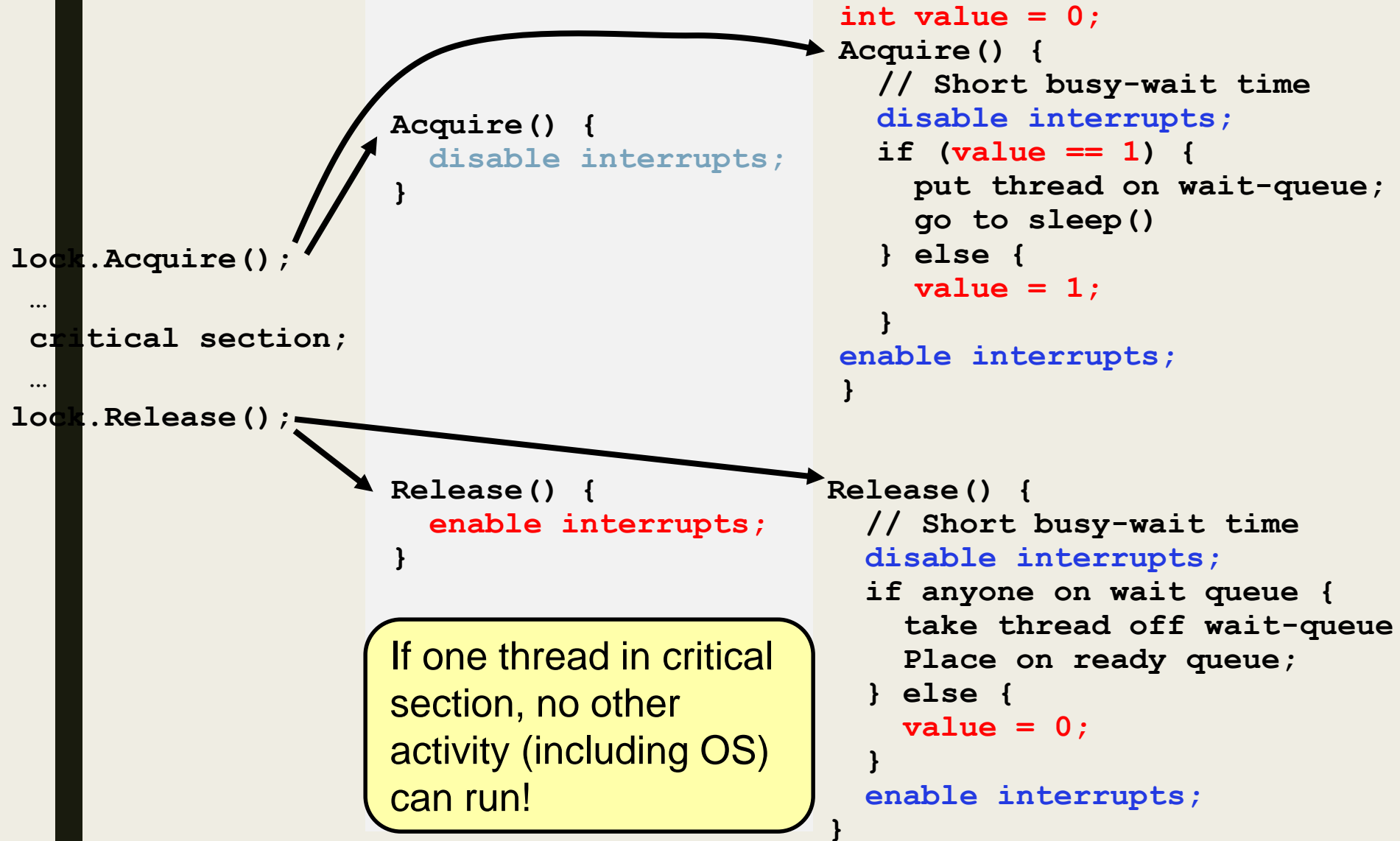
```
}
```



- Basically replace

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

Recap: Locks



Recap: Locks

