Operating Systems CMPSCI 377 Spring 2020

Synchronization and Locks

Clicker Question

What is fastest to context switch?

(A) Two threads of the same process

(B) Two threads of different processes

(C) Neither, it is the same

(D) It depends on the balance of I/O and CPU in each thread

Answer on Next Slide

Last Time

- Concurrency: An Introduction
  - Why use threads?
  - Thread Examples
  - Shared Data
  - Uncontrolled Scheduling
  - Atomicity
  - Waiting for others

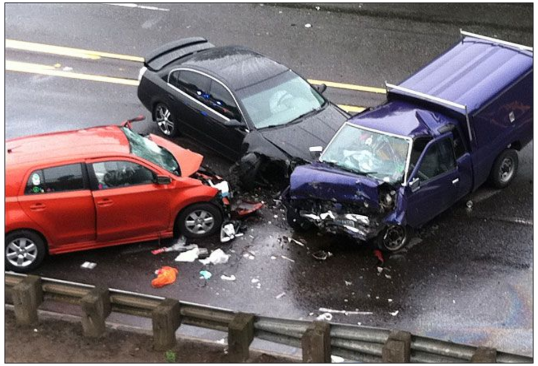
Today's Class

- Synchronization and Locks
    - Atomic operations
    - Test-and-set
    - Compare-and-swap
    - Spin locks

Problem: Threads and Critical Sections

From the introduction to concurrency, we saw one of the fundamental problems in concurrent programming:

We would like to execute a series of instructions atomically, but due to the presence of interrupts on a single processor (or multiple threads executing on multiple processors concurrently), we couldn't.



Problem: Threads and Critical Sections

We thus attack this problem directly.

To do this we introduce something referred to as a lock.

Programmers annotate source code with locks, putting them around critical sections, and thus ensure that any such critical section executes as if it were a single atomic instruction.

Locks: The Basic Idea

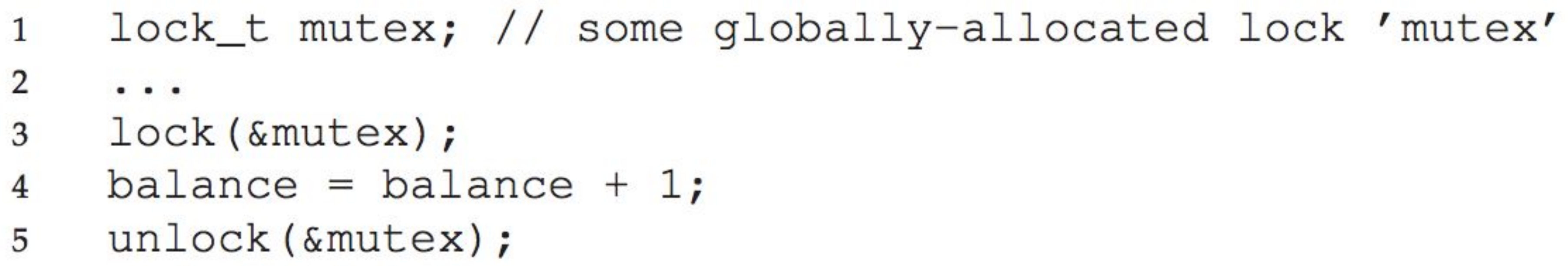
As an example, assume our critical section looks like this, the canonical update of a shared variable:

```
balance = balance + 1;
```

Of course, other critical sections are possible, such as adding an element to a linked list or other more complex updates to shared structures, but we’ll just keep to this simple example for now.

Locks: The Basic Idea

To use a lock, we add some code around the critical section like this:



```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks: The Basic Idea

To use a lock, we add some code around the critical section like this:

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

A lock is just a variable.

You must declare a lock variable of some kind (such as mutex above).

This lock variable (or just “lock” for short) holds the state of the lock at any instant in time.

Locks: The Basic Idea

To use a lock, we add some code around the critical section like this:

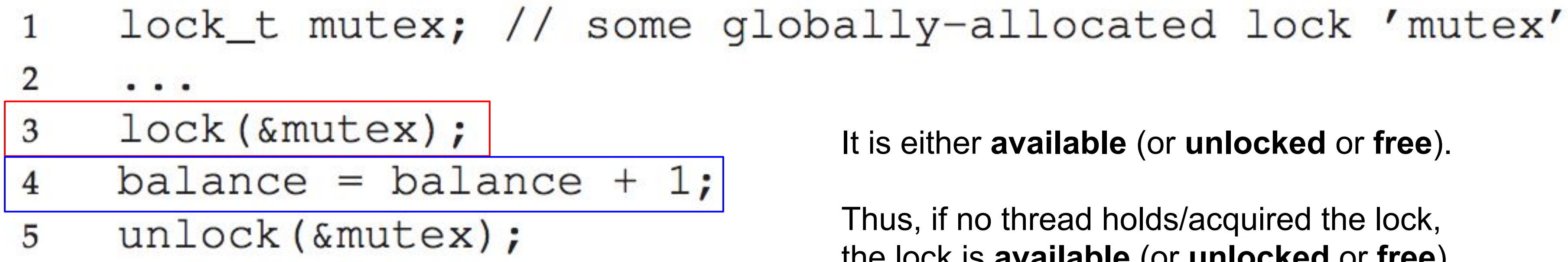
```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

It is either **available** (or **unlocked** or **free**).

Thus, if no thread holds/acquired the lock, the lock is **available** (or **unlocked** or **free**).

Locks: The Basic Idea

To use a lock, we add some code around the critical section like this:



```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

It is either **available** (or **unlocked** or **free**).

Thus, if no thread holds/acquired the lock, the lock is **available** (or **unlocked** or **free**).

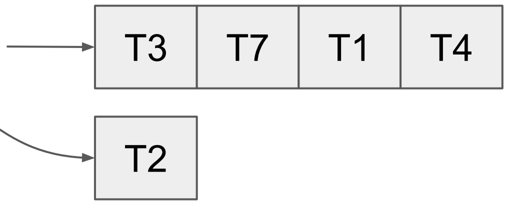
If the lock is acquired, exactly one thread holds the lock and presumably is about to enter or is currently in a critical section.

Locks: The Basic Idea

To use a lock, we add some code around the critical section like this:

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

```
struct lock_t {
  Queue<pthread> waiting;
  pthread owner;
};
```



We could (and will) store other information in the data type as well, such as:

- Which thread holds the lock
- Or a queue for ordering lock acquisition

Information like that is hidden from the *user* of the lock.

Lock Semantics

Calling the routine lock():

```
void lock(lock_t *mutex) {
  While (true):
    If no other thread holds the lock:
      Current thread will acquire the lock.
      Return.
    Else:
      Queue the current thread in the waiting queue.
      Put the thread in the "blocked" state.
}
```

Lock Semantics

Calling the routine lock():

```
void lock(lock_t *mutex) {
  While (true):
    If no other thread holds the lock:
      Current thread will acquire the lock.
      Return.
    Else:
      Queue the current thread in the waiting queue.
      Put the thread in the "blocked" state.
}
```

Keep looping until the current thread can acquire the lock.

Lock Semantics

Calling the routine lock():

```
void lock(lock_t *mutex) {
  While (true):
    If no other thread holds the lock:
      Current thread will acquire the lock.
      Return.
    Else:
      Queue the current thread in the waiting queue.
      Put the thread in the "blocked" state.
}
```

If the lock is free, the current thread acquires the lock and returns.

Lock Semantics

Calling the routine lock():

```
void lock(lock_t *mutex) {
  While (true):
    If no other thread holds the lock:
      Current thread will acquire the lock.
      Return.
    Else:
      Queue the current thread in the waiting queue.
      Put the thread in the "blocked" state.
}
```

If the lock is held by another thread, the current thread is queued on the waiting list.

The current thread is then blocked and will move into the running state later to check if the lock is available.

Clicker Question

This is guaranteed to work (TRUE/FALSE):

```
If no other thread holds the lock:
      Current thread will acquire the lock.
      Return.
```

(A) TRUE

(B) FALSE

Answer on Next Slide

Lock Semantics - Huh?

Calling the routine lock():

```
void lock(lock_t *mutex) {
  While (true):
    If no other thread holds the lock:
      Current thread will acquire the lock.
      Return.
    Else:
      Queue the current thread in the waiting queue.
      Put the thread in the "blocked" state.
}
```

Huh?

I thought this is an atomic operation?

This does not look atomic.

We will come back to this shortly...

Unlock Semantics

Calling the routine unlock():

```
void unlock(lock_t *mutex) {
  Release the lock if current thread is the owner
  If there are waiting threads:
    Wake up all threads.
    Return.
}
```

Once the owner of the lock calls unlock(), the lock is now available (free) again.

Unlock Semantics

Calling the routine unlock():

```
void unlock(lock_t *mutex) {
  Release the lock if current thread is the owner
  If there are waiting threads:
    Wake up all threads.
    Return.
}
```

If no other threads are waiting for the lock, the state of the lock is simply changed to free.

If there are waiting threads (stuck in lock()), one of them will (eventually) notice (or be informed of) this change of the lock's state, acquire the lock, and enter the critical section.

Power to the Threads

Locks provide some minimal amount of control over scheduling to programmers.

In general, we view threads as entities created by the programmer but scheduled by the OS, in any fashion that the OS chooses.

Power to the Threads

Locks provide some minimal amount of control over scheduling to programmers.

In general, we view threads as entities created by the programmer but scheduled by the OS, in any fashion that the OS chooses.

Locks yield some of that control back to the programmer!

Power to the Threads

Locks provide some minimal amount of control over scheduling to programmers.

In general, we view threads as entities created by the programmer but scheduled by the OS, in any fashion that the OS chooses.

Locks yield some of that control back to the programmer!

By putting a lock around a section of code, the programmer can guarantee that no more than a single thread can ever be active within that code.

Power to the Threads

Locks provide some minimal amount of control over scheduling to programmers.

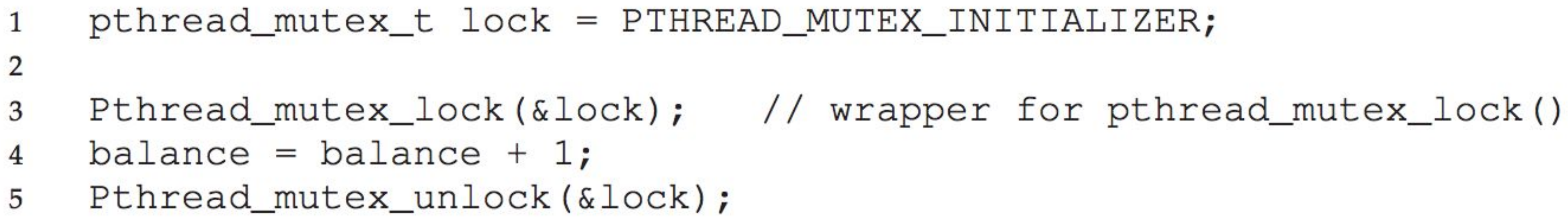
In general, we view threads as entities created by the programmer but scheduled by the OS, in any fashion that the OS chooses.

Locks yield some of that control back to the programmer!

By putting a lock around a section of code, the programmer can guarantee that no more than a single thread can ever be active within that code.

Thus locks help transform the chaos that is traditional OS scheduling into a more controlled activity.

Pthread Locks



```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock);    // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

The name that the POSIX library uses for a lock is a mutex, as it is used to provide mutual exclusion between threads.

Pthread Locks

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock);    // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

You might also notice here that the POSIX version passes a variable to lock and unlock, as we may be using different locks to protect different variables. Doing so can increase concurrency:

- Coarse-grained locking strategy
- Fine-grained locking strategy

Pthread Locks

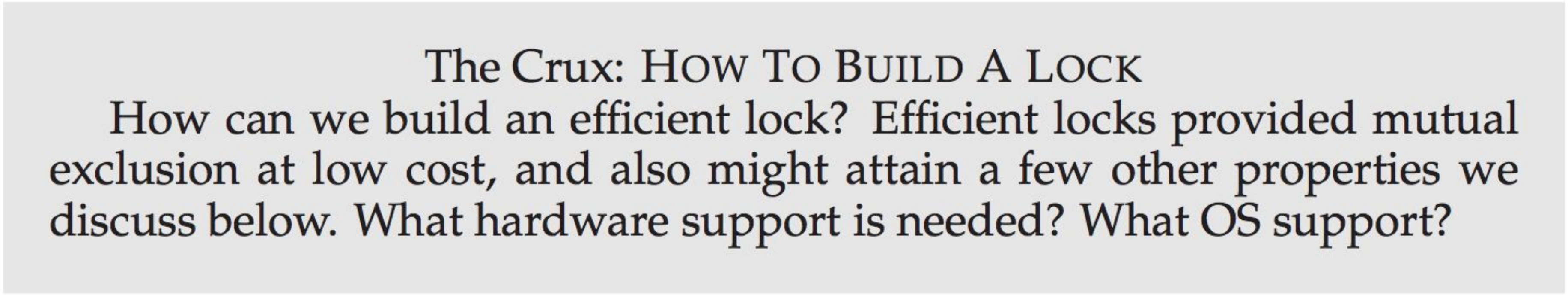
```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock);    // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

You might also notice here that the POSIX version passes a variable to lock and unlock, as we may be using different locks to protect different variables. Doing so can increase concurrency:

- Coarse-grained locking strategy
- Fine-grained locking strategy

Can you think of an example?

Building a Lock



The Crux: HOW TO BUILD A LOCK

How can we build an efficient lock? Efficient locks provided mutual exclusion at low cost, and also might attain a few other properties we discuss below. What hardware support is needed? What OS support?

Evaluating Locks

Goals:

- Mutual exclusion
- Fairness (do not want to starve threads)
- Performance

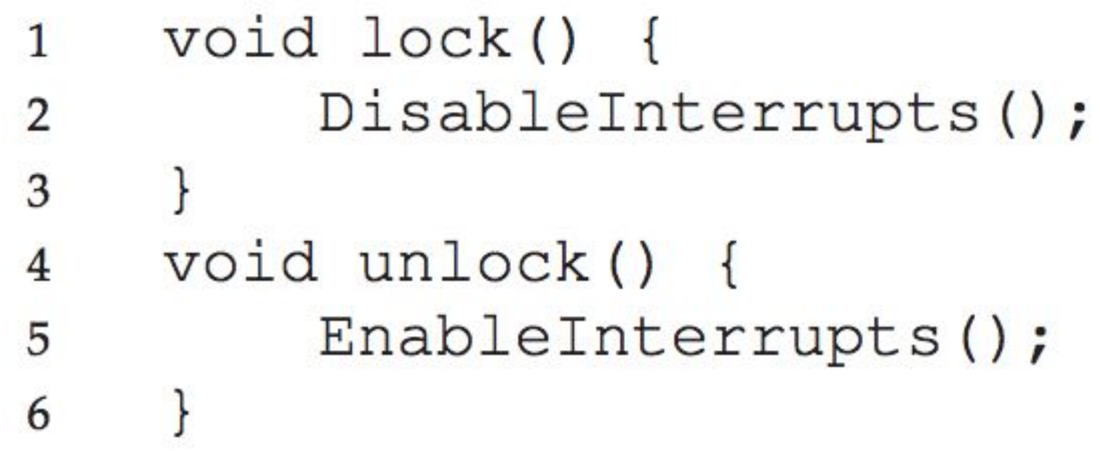
Controlling Interrupts

One of the earliest solutions used to provide mutual exclusion was to disable interrupts for critical sections; this solution was invented for single-processor systems. The code would look like this:

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Controlling Interrupts

What are the PROS of this approach?

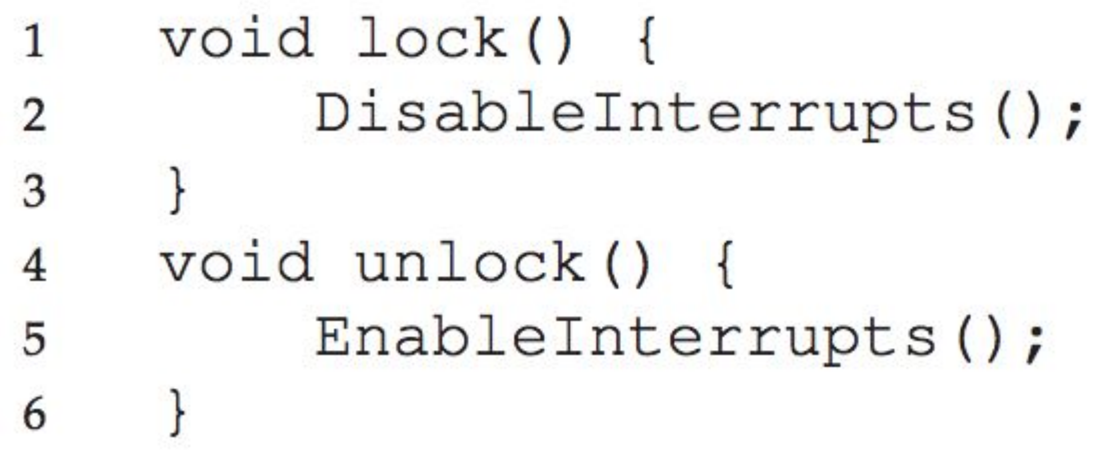


```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Controlling Interrupts

What are the PROS of this approach?

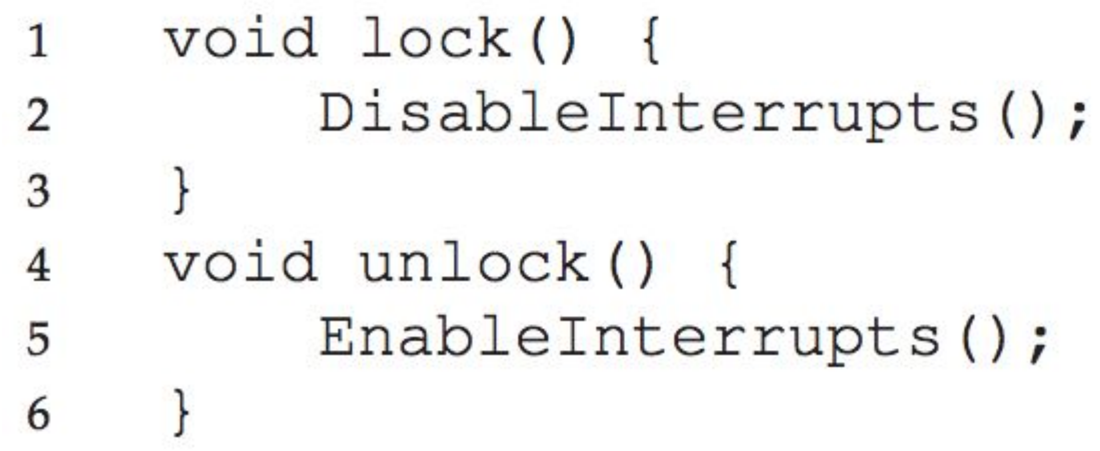
It is certainly very simple!



```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Controlling Interrupts

What are the CONS of this approach?

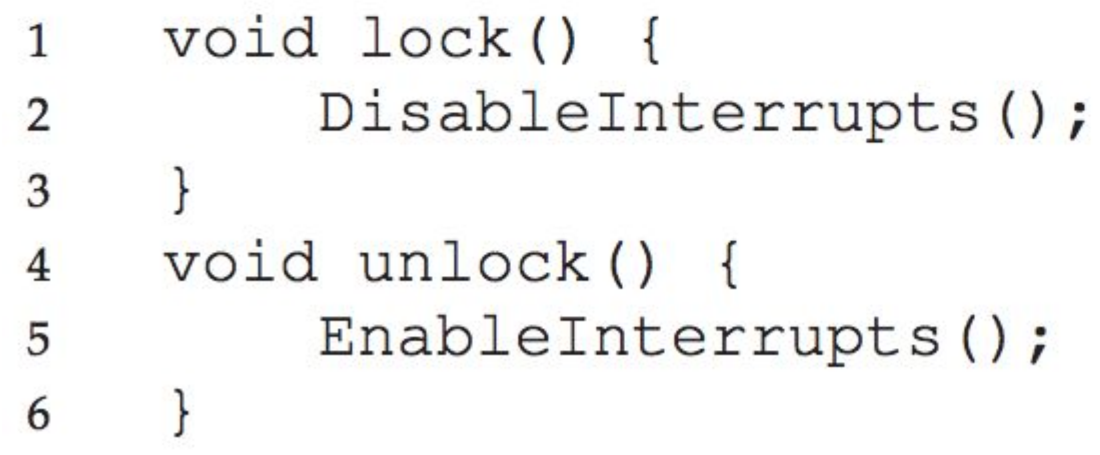


```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Controlling Interrupts

What are the CONS of this approach?

#1: Disabling/Enabling of interrupts is a privileged operation.

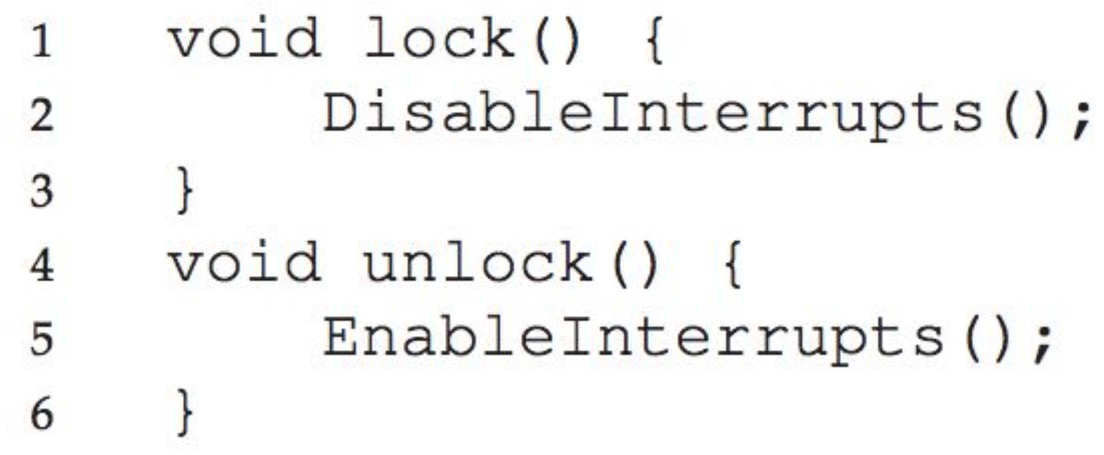


```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Controlling Interrupts

What are the CONS of this approach?

#2: A greedy program could monopolize the CPU.



```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Controlling Interrupts

What are the CONS of this approach?

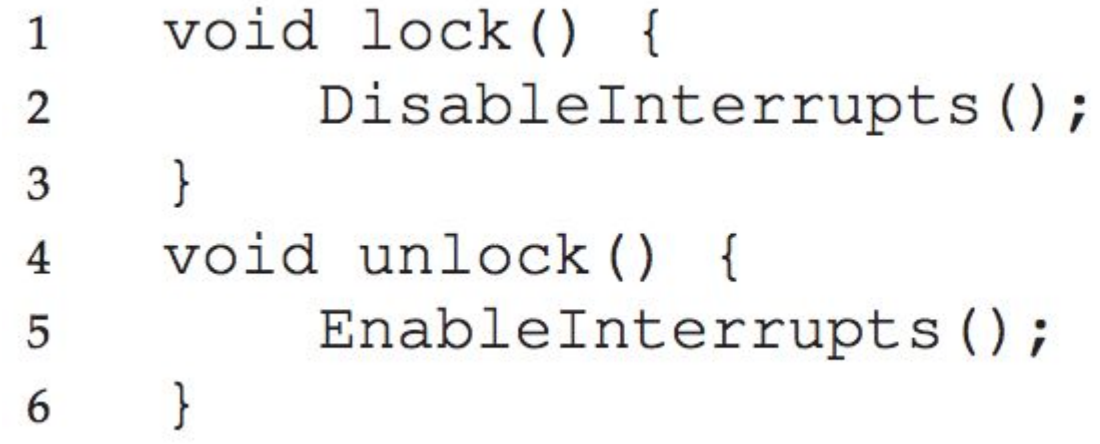
#3: A malicious program could take over the machine (how?)

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Controlling Interrupts

What are the CONS of this approach?

#4: Doesn't work on multiple processors (why is that?)

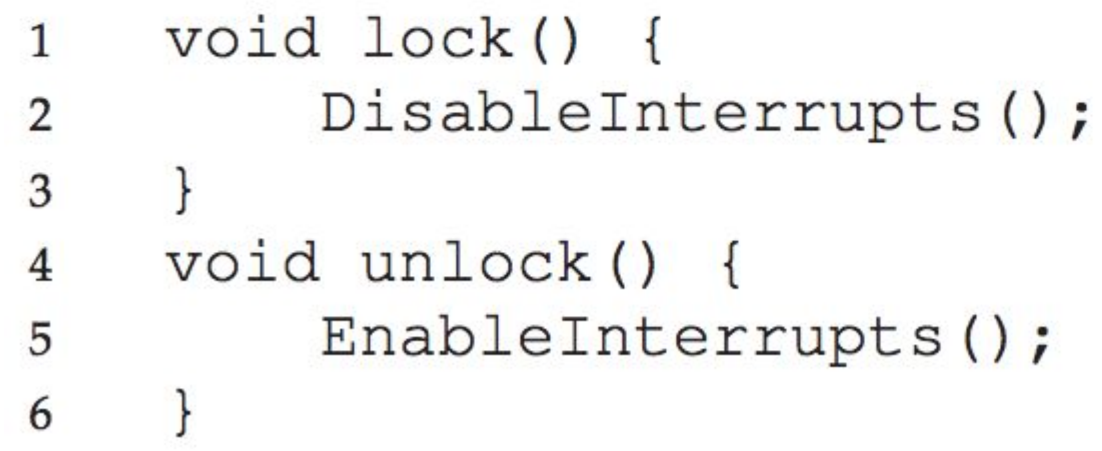


```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Controlling Interrupts

What are the CONS of this approach?

#5: Can lead to interrupts being lost.

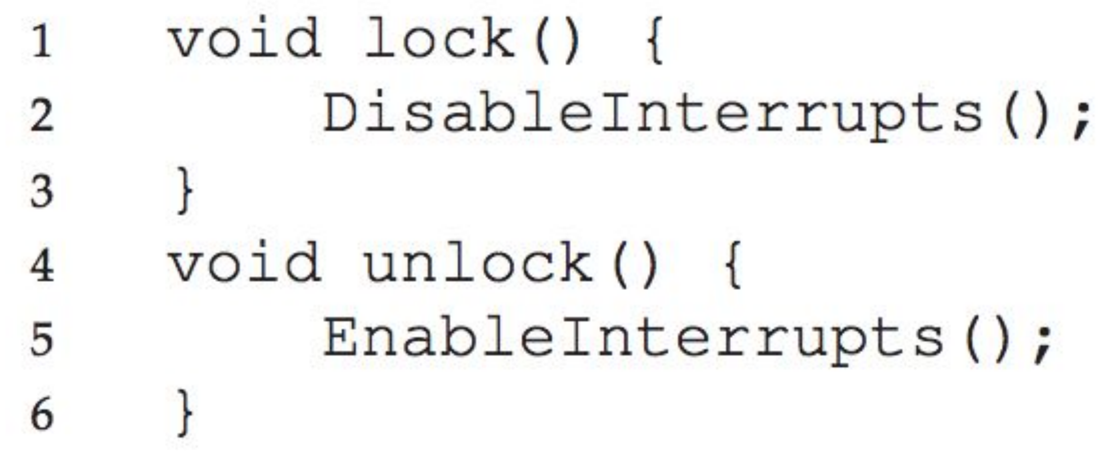


```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Controlling Interrupts

What are the CONS of this approach?

#6: Poor performance.



```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

A Failed Attempt: Just Using Loads and Stores

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Use a simple variable (flag) to indicate whether some thread has possession of a lock.

A Failed Attempt: Just Using Loads and Stores

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

The lock is initialized as expected.

0: lock is available

1: lock is held

A Failed Attempt: Just Using Loads and Stores

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

The first thread that enters the critical section will call lock().

This tests whether the flag is equal to 1 (in this case, it is not), and then sets the flag to 1 to indicate that the thread now holds the lock.

A Failed Attempt: Just Using Loads and Stores

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

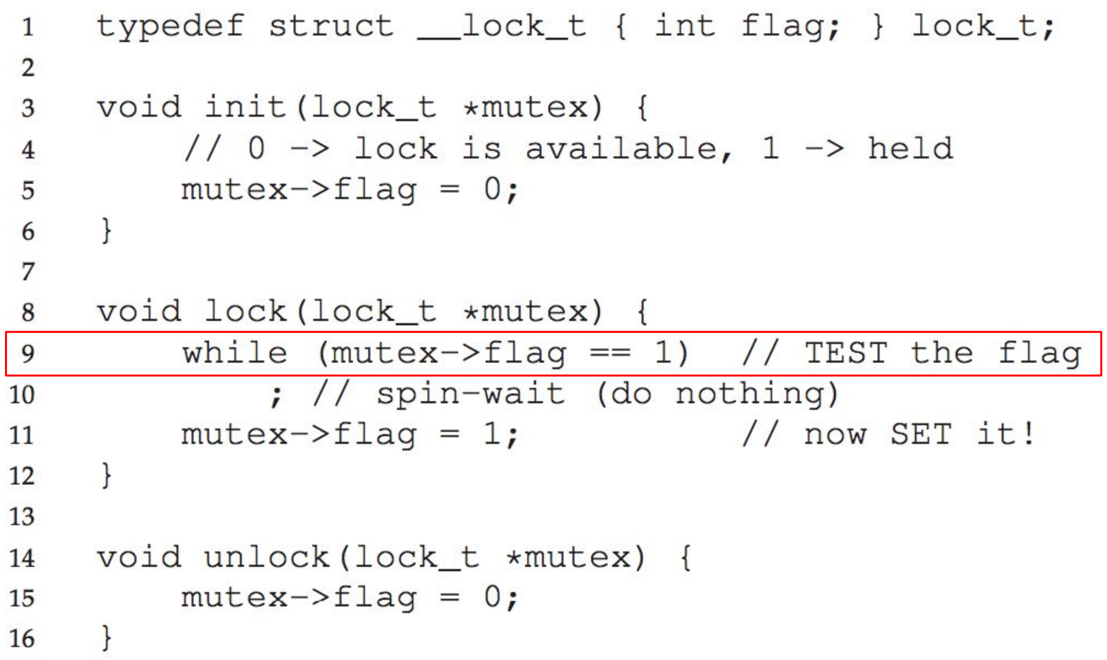
When finished with the critical section, the thread calls unlock() and clears the flag, thus indicating that the lock is no longer held.

A Failed Attempt: Just Using Loads and Stores

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

What is the problem?

A Failed Attempt: Just Using Loads and Stores

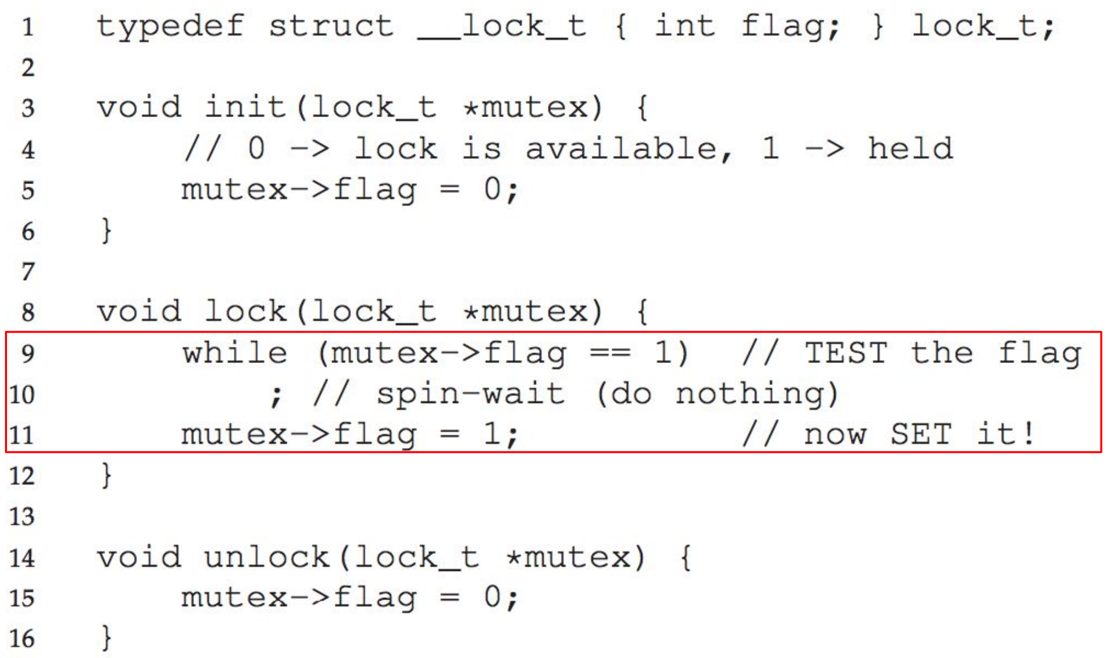


```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

What is the problem?

#1: Performance.

A Failed Attempt: Just Using Loads and Stores



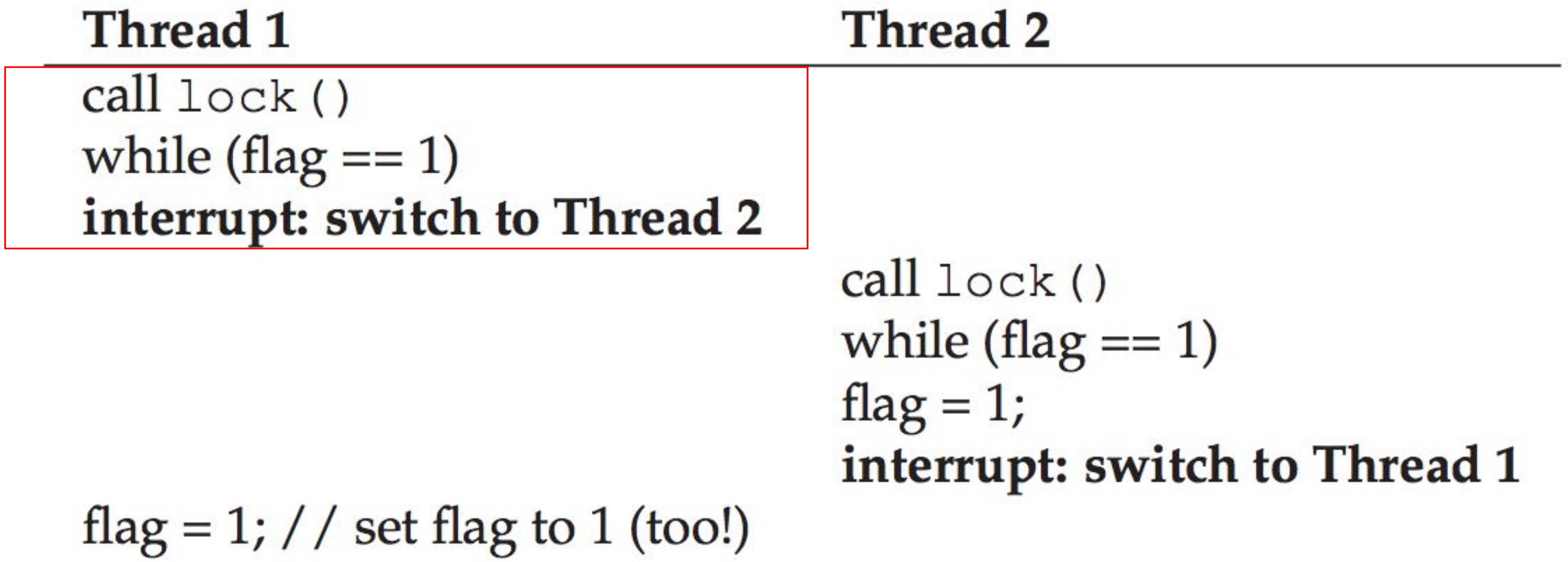
```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

What is the problem?

#1: Performance.

#2: Correctness.

A Failed Attempt: Just Using Loads and Stores

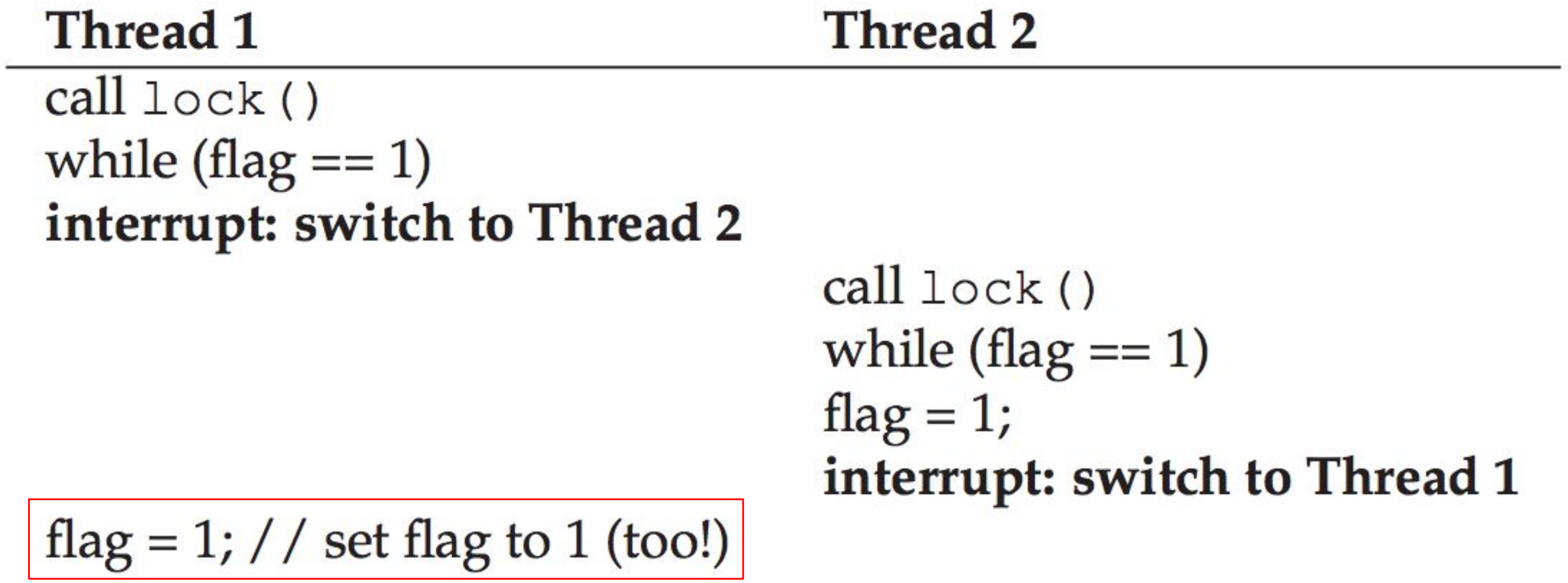


Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

A Failed Attempt: Just Using Loads and Stores

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

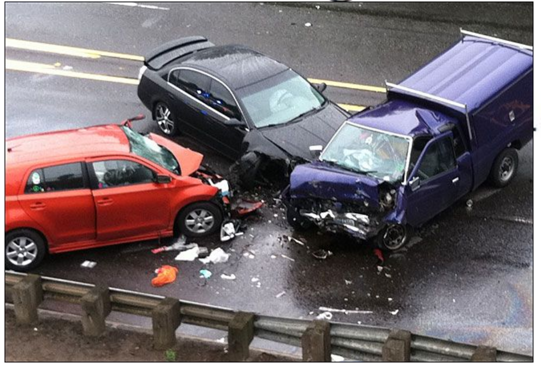
A Failed Attempt: Just Using Loads and Stores



Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

A Failed Attempt: Just Using Loads and Stores

Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	to Thread 1
flag = 1; // set flag to 1 (too!)	



Building Working Spin Locks with **test-and-set**

Disabling interrupts does not work - plenty of problems with that approach.

Building Working Spin Locks with **test-and-set**

Disabling interrupts does not work - plenty of problems with that approach.

Simple approaches using loads and stores don’t work either!

Building Working Spin Locks with **test-and-set**

Disabling interrupts does not work - plenty of problems with that approach.

Simple approaches using loads and stores don’t work either!

To overcome these problems, system designers invented hardware support for locking.

Building Working Spin Locks with **test-and-set**

Disabling interrupts does not work - plenty of problems with that approach.

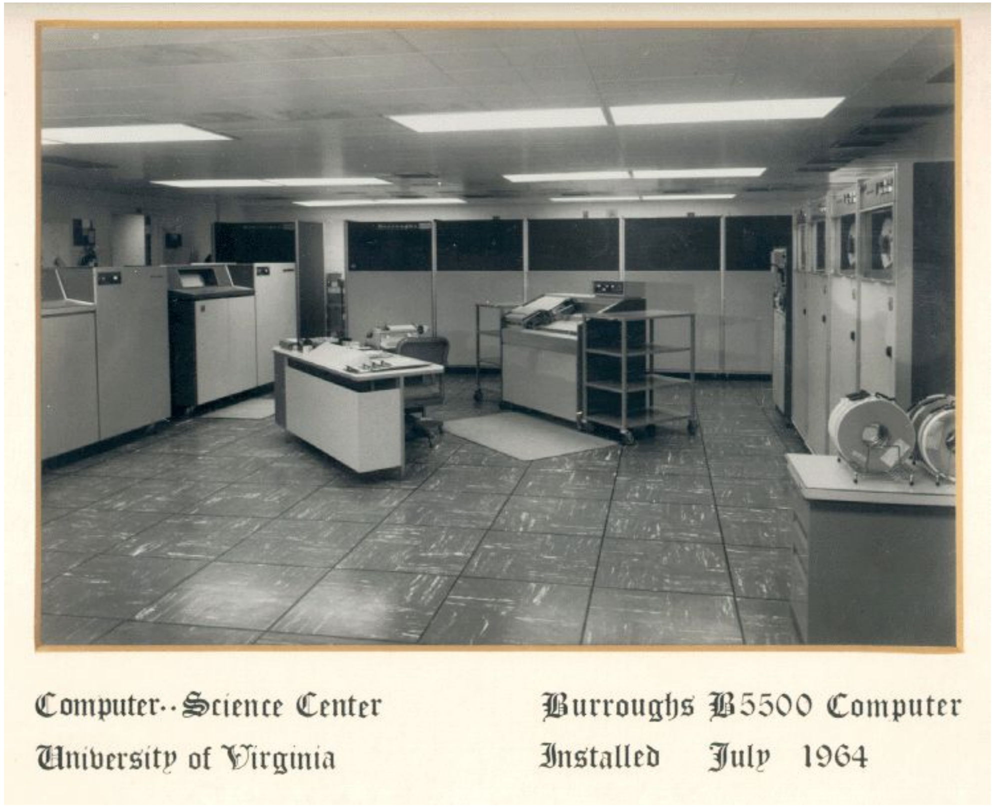
Simple approaches using loads and stores don’t work either!

To overcome these problems, system designers invented hardware support for locking.

The earliest multiprocessor systems, such as the Burroughs B5000 in the early 1960’s, had such support.

<u>Today all systems provide this type of support, even for single CPU systems.</u>

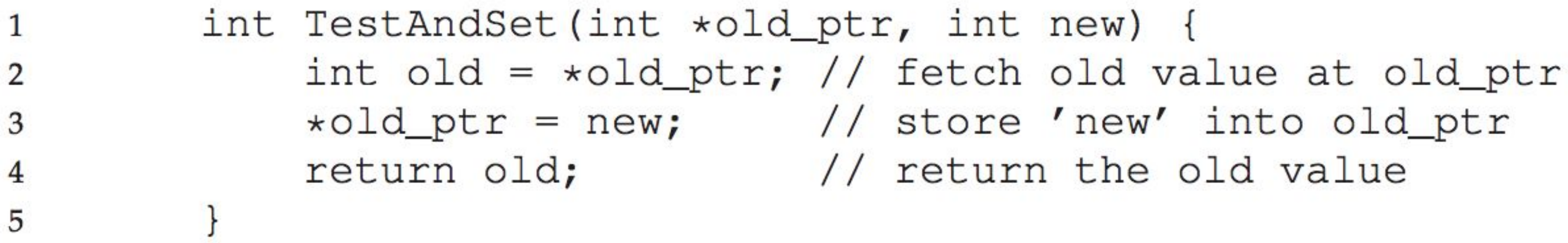
Building Working Spin Locks with **test-and-set**


Computer--Science Center
University of Virginia
Burroughs B5500 Computer
Installed July 1964

The **test-and-set** instruction

The simplest bit of hardware support to understand is what is known as a **test-and-set** instruction.

We define what the **test-and-set** instruction does via the following C code snippet:

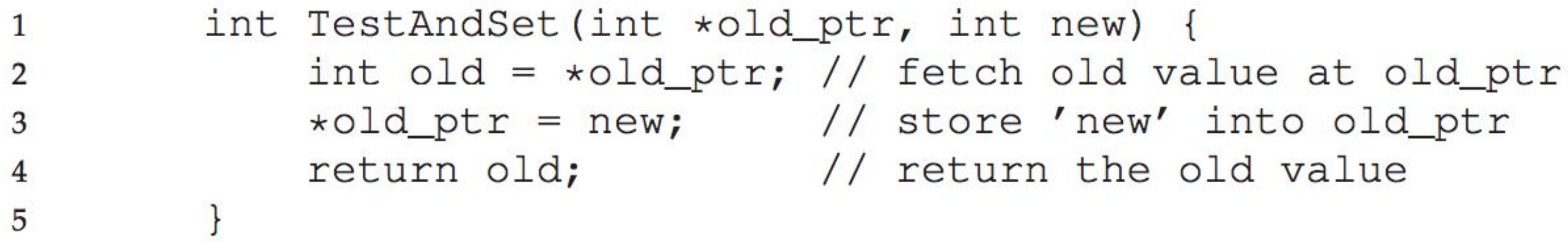


```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

The **test-and-set** instruction

test-and-set returns the old value pointed to by the ptr, and simultaneously updates said value to new.

The key, of course, is that this sequence of operations is performed atomically.



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

The **test-and-set** instruction

The reason it is called “test and set” is that it enables you to **test** the old value (which is what is returned) while simultaneously **setting** the memory location to a new value.

This slightly more powerful instruction is enough to build a simple spin lock.

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

How do we use **test-and-set**?

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Using the TestAndSet C definition, how might you use it in this code fragment to safely fix the load/store issue?

```
int TestAndSet(int *old_ptr,
               int new) {
  int old = *old_ptr;
  *old_ptr = new;
  return old;
}
```

How do we use **test-and-set**?

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Using the TestAndSet C definition, how might you use it in this code fragment to safely fix the load/store issue?

```
int TestAndSet(int *old_ptr,
               int new) {
  int old = *old_ptr;
  *old_ptr = new;
  return old;
}
```

Evaluating Spin Locks: Correctness

Given our basic spin lock, we can now evaluate how effective it is along our previously described metrics.

The most important aspect of a lock is **correctness**:
	does it provide mutual exclusion?

The answer here is yes:
	the spin lock only allows a single thread to enter the critical section at a time.
	Thus, we have a correct lock.

Evaluating Spin Locks: Fairness

The next metric is **fairness**.

How fair is a spin lock to a waiting thread?
Can you guarantee that a waiting thread will ever enter the critical section?

The answer here, unfortunately, is bad news:

spin locks don’t provide any fairness guarantees. Indeed, a thread spinning may spin forever, under contention. Simple spin locks (as discussed thus far) are not fair and may **lead to starvation**.

Evaluating Spin Locks: Performance

The final metric is **performance**.

What are the costs of using a spin lock?

To analyze this more carefully, think about a few different cases.

#1: Imagine threads competing for the lock on a single processor.

#2: Consider threads spread out across many CPUs.

Evaluating Spin Locks: Performance

#1: Imagine threads competing for the lock on a single processor.

In the single CPU case, performance overheads can be quite painful!

Imagine the case where the thread holding the lock is pre-empted within a critical section. The scheduler might then run every other thread (imagine there are N − 1 others), each of which tries to acquire the lock. In this case, each of those threads will spin for the duration of a time slice before giving up the CPU, a waste of CPU cycles.

Evaluating Spin Locks: Performance

#2: Consider threads spread out across many CPUs.

On multiple CPUs, spin locks work reasonably well (if the number of threads roughly equals the number of CPUs).

Imagine Thread A on CPU 1 and Thread B on CPU 2, both contending for a lock.

If Thread A (CPU 1) grabs the lock, and then Thread B tries to, B will spin (on CPU 2). However, presumably the critical section is short, and thus soon the lock becomes available, and is acquired by Thread B. Spinning to wait for a lock held on another processor doesn't waste many cycles in this case, and thus can be effective

Another Approach: Compare and Swap

Another hardware primitive that some systems provide is known as the **compare-and-swap** instruction.

The C pseudocode for this single instruction looks like this:

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Another Approach: Compare and Swap

The basic idea is for compare-and-swap to test whether the value at the address specified by ptr is equal to expected; if so, update the memory location pointed to by ptr with the new value. If not, do nothing. In either case, return the actual value at that memory location, thus allowing the code calling compare-and-swap to know whether it succeeded or not.

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Clicker Question

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Which of these is right for locking?

(A) while (CompareAndSwap(&lock->flag, 0, 1) == 0);

(B) while (CompareAndSwap(&lock->flag, 1, 0) == 0);

(C) while (CompareAndSwap(&lock->flag, 0, 1) == 1);

(D) while (CompareAndSwap(&lock->flag, 1, 0) == 1);

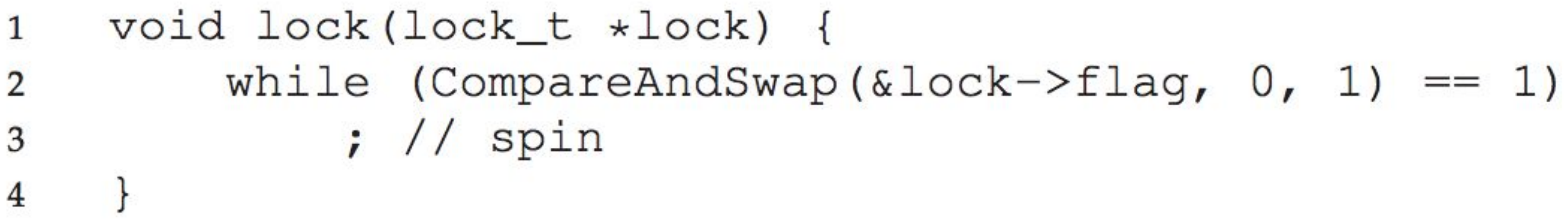
(E) while (CompareAndSwap(&lock->flag, 1, 1) == 1);

Answer on Next Slide

Another Approach: Compare and Swap

compare-and-swap is a more powerful instruction than test-and-set.

However, if we just build a simple spin lock with it, its behavior is identical to the spin lock we analyzed for test-and-set.



```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

What to do?

There are many approaches to implementing atomic instructions in hardware.

We have looked at test-and-set and compare-and-swap.

There are other approaches:

- Load-Linked and Store-Conditional
- Fetch-and-Add / Ticket Locks
- And many more…

There is still a fundamental problem with the efficiency of a spin lock.

Yielding a Thread

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Yielding a Thread

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

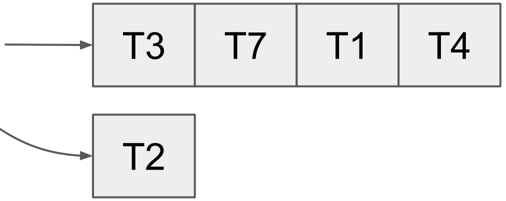
What about starvation?

Locks: The Basic Idea

To use a lock, we add some code around the critical section like this:

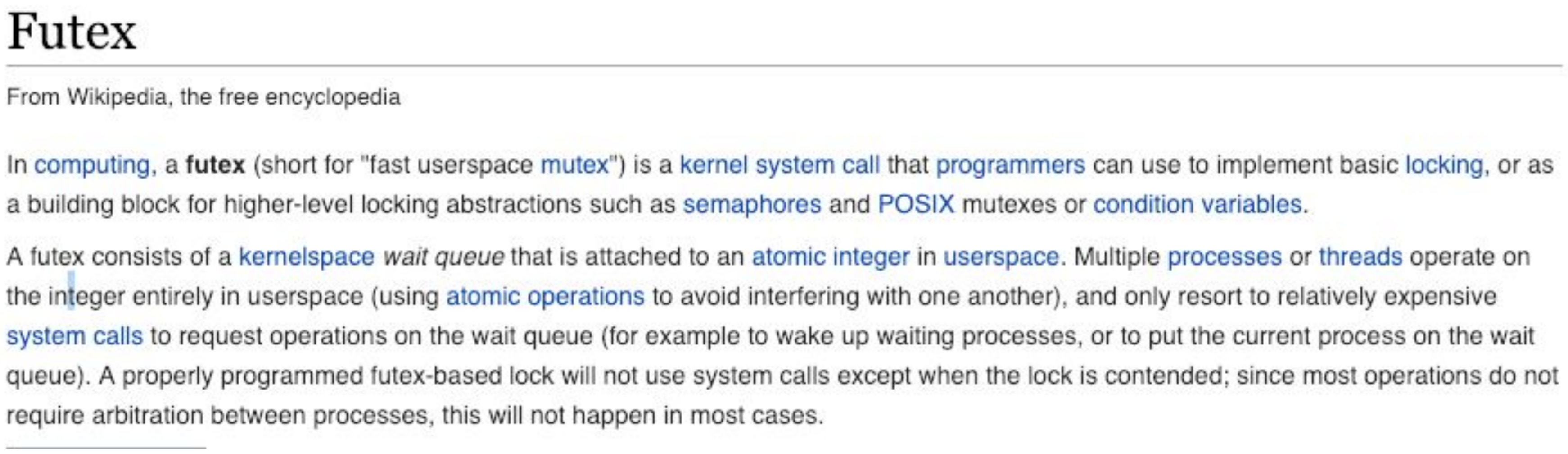
```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

```
struct lock_t {
  Queue<pthread> waiting;
  pthread owner;
};
```



To help starvation the OS provides sophisticated scheduling techniques, like the ones we have covered, to minimize busy-waiting (spin lock) and improve fairness.

Case Study: Linux locks



Futex

From Wikipedia, the free encyclopedia

In computing, a **futex** (short for "fast userspace mutex") is a kernel system call that programmers can use to implement basic locking, or as a building block for higher-level locking abstractions such as semaphores and POSIX mutexes or condition variables.

A futex consists of a kernelspace *wait queue* that is attached to an atomic integer in userspace. Multiple processes or threads operate on the integer entirely in userspace (using atomic operations to avoid interfering with one another), and only resort to relatively expensive system calls to request operations on the wait queue (for example to wake up waiting processes, or to put the current process on the wait queue). A properly programmed futex-based lock will not use system calls except when the lock is contended; since most operations do not require arbitration between processes, this will not happen in most cases.

Next Tuesday

Exam 1 - in class.

multiple choice

true/false

2-3 open response questions