



# COS40003 Concurrent Programming

## Lecture 6: Lock II

# Outline

- Review of Lock
  - Review basic idea of Lock
  - Review lock struct
  - Review C Implementation of Lock
  - Review Java Implementation of Lock
    - Using ReentrantLock
    - Using Synchronized methods
    - Using Synchronized statements
  - Discuss when to use which lock
- How lock is implemented

# Basic Idea: Lock, a struct associated with fields and methods

```
1 lock_t mutex;  
// lock_t: lock type, a struct;  
mutex: lock variable;  
  
2 lock(&mutex);  
3 balance = balance + 1;  
4 unlock(&mutex);  
// lock, unlock: lock methods
```

# Basic Idea: Lock, a struct associated with fields and methods

```
lock_t mutex;
```

- Information stored in Lock struct `lock_t`
  - Which thread holds the lock
  - A queue for ordering lock acquisition in future
  - etc. (we will see later)

# POSIX C implementation of Lock

```
#include <pthread.h>
```

```
1 pthread_mutex_t lock =  
  PTHREAD_MUTEX_INITIALIZER;
```

```
2 Pthread_mutex_lock(&lock);
```

```
3 balance = balance + 1;  
// critical section
```

```
4 Pthread_mutex_unlock(&lock);
```

# Java Implementation of Lock

- **Using Lock object**
- **Using “Synchronized” keyword**
  - synchronized methods
  - synchronized statements

# Using ReentrantLock Class

```
public class MyClass {  
    private int shared_variable;  
    private ReentrantLock lock;  
    //ReentrantLock class implements Lock interface  
    public void myFunction() {  
        lock.lock();  
        try {  
  
            shared_variable++;  
            // critical section  
  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

# Using **synchronized** keyword

```
public synchronized void MyFunction() {
```

```
    // critical section
```

```
}
```



```
synchronized(shared_object) {  
    //synchronized(this) {
```

```
        // critical section
```

```
}
```





# Example: Synchronized statements

```
public void addName(String name) {  
  
    // other work goes here  
  
    synchronized(this) {  
  
        lastName = name;  
        nameCount++;  
  
    }  
  
    // other work goes here  
  
}
```

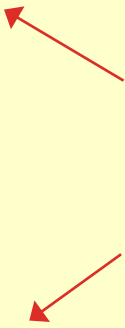
# When we should use intrinsic lock, when we should use object's lock?

- Example
  - One critical section in one instance
    - `Synchronized(this)`
  - More than one critical section
    - `Synchronized(object1){ c1++ }`
    - `Synchronized(object2){ c2++ }`
- Guideline: instead of one big lock that is used any time in any critical section, one can protect different data and data structures with different locks

# Example: Synchronized statements

```
public class MsLunch {  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        synchronized(lock1) {  
            c1++;  
        }  
    }  
  
    public void inc2() {  
        synchronized(lock2) {  
            c2++;  
        }  
    }  
}
```

Using an object that  
provides intrinsic lock



# How to protect static variable?

- Static variable does not belong to a particular instance
- Using the intrinsic lock of the instance does not guard mutually exclusive accesses to a static variable!

# Example 3 (Recall)

```
3 public class IncrementTest implements Runnable{
4
5     static int classData = 0;
6     int instanceData = 0;
7
8     @Override
9     public void run() {
10         int localData = 0;
11
12         while (localData < 100000000) {
13             localData++;
14             instanceData++;
15             classData++;
16         }
17
18         System.out.println("localData: " + localData +
19                             "\tinstanceData: " + instanceData +
20                             "\tclassData: " + classData);
21     }
22
23
24     public static void main(String[] args) {
25         // TODO Auto-generated method stub
26         IncrementTest instance = new IncrementTest();
27
28         Thread t1 = new Thread(instance);
29         Thread t2 = new Thread(instance);
30
31         t1.start();
32         t2.start();
33     }
34 }
35
36 13
37 }
```

is shared between all instances of this class

# Exercises

- Try: create thread t2 with another instance and see the result:

```
IncrementTest instance = new IncrementTest();  
IncrementTest instance2 = new IncrementTest();  
Thread t1 = new Thread(instance);  
Thread t2 = new Thread(instance2);
```

# How to protect static variable?

- Static variable does not belong to a particular instance
- Using the intrinsic lock of the instance does not guard mutually exclusive accesses to a static variable!
- Solution:
  - Synchronized(ClassName.class){}
  - Use a Static ReentrantLock

# Example: protect static variable?

-----Method 1-----

```
synchronized (IncrementTest.class) {  
    classData++;  
}
```

-----Method 2-----

```
static ReentrantLock data_lock = new  
ReentrantLock();
```

```
    data_lock.lock();  
    try{  
        classData++;  
    }  
    finally {  
        data_lock.unlock();  
    }
```



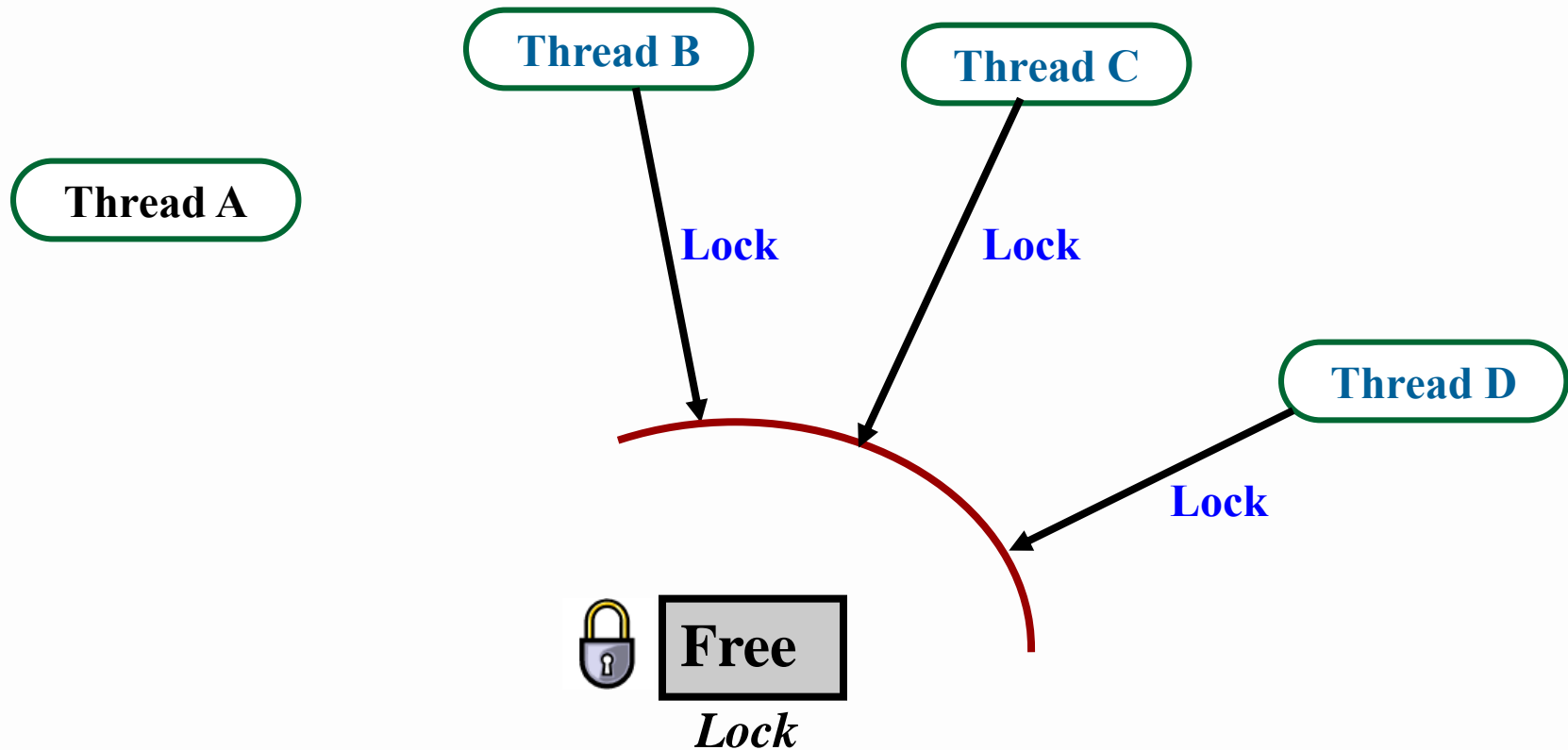
# Outline

- Review of Lock
  - Review basic idea of Lock
  - Review lock struct
  - Review C Implementation of Lock
  - Review Java Implementation of Lock
    - Using ReentrantLock
    - Using Synchronized methods
    - Using Synchronized statements
  - Discuss when to use which lock
- How lock is implemented

# Outline

- How lock is implemented
  - Controlling interrupts;
  - Loads/Stores
  - Test-and-Set;
  - Compare-and-Swap;
  - Fetch-and-Add;
  - Yield();
  - Using queues
- Requirement: be able to read and understand the C code. Not required to write.

# Who will get lock next ?



# To build a lock

- Hardware Support
- Operating System Support
- Good lock criteria
  - Mutual exclusion
    - preventing multiple threads from entering a critical section
  - Fairness
    - Each thread contending for the lock get a fair shot at acquiring it once it is free, ie. **never starve**
  - Performance
    - Low time overhead added by using the lock

# Outline

- How lock is implemented
  - Controlling interrupts;
  - Loads/Stores
  - Test-and-Set;
  - Compare-and-Swap;
  - Fetch-and-Add;
  - Yield();
  - Using queues
- Requirement: be able to read and understand the C code. Not required to write.

# A bad implementation of lock: Controlling Interrupts

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

- 1. Requires too much trust in applications.
- 2. Does not work on multiprocessors
- 3. Turning off interrupts for extended periods of time can lead to serious systems problems
- 4. This approach can be inefficient (least important)

# A bad implementation of lock: Controlling Interrupts

- 1. Requires too much trust in applications.
  - a greedy program could call `lock()` at the beginning of its execution and thus monopolize the processor; worse, a malicious program could call `lock()` and go into an endless loop
- 2. Does not work on multiprocessors
  - it does not matter whether interrupts are disabled on the current processor; threads will be able to run on other processors

# A bad implementation of lock: Controlling Interrupts

- 3. Turning off interrupts for extended periods of time can lead to serious systems problems
  - Eg., if the CPU missed the fact that a disk device has finished a read request. How will the OS know to wake the process waiting for the read?
- 4. This approach can be inefficient (least important)



# A bad implementation of lock: Controlling Interrupts

- Conclusion: turning off interrupts is only used in limited contexts
  - eg, in some cases an operating system itself will use interrupt masking to guarantee atomicity when accessing its own data structures, or at least to prevent certain messy interrupt handling situations from arising.
  - This usage makes sense, as the trust issue disappears inside the OS, which always trusts itself to perform privileged operations

# Outline

- How lock is implemented
  - Controlling interrupts;
  - Loads/Stores
  - Test-and-Set;
  - Compare-and-Swap;
  - Fetch-and-Add;
  - Yield();
  - Using queues
- Requirement: be able to read and understand the C code. Not required to write.

# A Failed Attempt: Just Using Loads/Stores

- The idea is quite simple:
  - use a simple variable (flag) to indicate whether some thread has possession of a lock.
- The first thread that enters the critical section will call `lock()`, which 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.
- 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/Stores

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1); // TEST the flag
10     // spin-wait (do nothing)
11     mutex->flag = 1; // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

# A Failed Attempt: Just Using Loads/Stores

- **Problem 1:** incorrect, no mutual exclusion

## Thread 1

```
call lock ()  
while (flag == 1)  
interrupt: switch to Thread 2
```

```
flag = 1; // set flag to 1 (too!)
```

## Thread 2

```
call lock ()  
while (flag == 1)  
flag = 1;  
interrupt: switch to Thread 1
```

# A Failed Attempt: Just Using Loads/Stores

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1); // TEST the flag
10     // spin-wait (do nothing)
11     mutex->flag = 1; // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

# A Failed Attempt: Just Using Loads/Stores

- **Problem 2:** the performance problem
  - it endlessly checks the value of flag, a technique called **spin-waiting**.
  - Spin-waiting wastes time waiting for another thread to release a lock. The waste is exceptionally high on a uniprocessor

# Outline

- How lock is implemented
  - Controlling interrupts;
  - Loads/Stores
  - Test-and-Set;
  - Compare-and-Swap;
  - Fetch-and-Add;
  - Yield();
  - Using queues
- Requirement: be able to read and understand the C code. Not required to write.



# A Failed Attempt: Just Using Loads/Stores

- **Problem 1:** incorrect, no mutual exclusion

## Thread 1

```
call lock ()
```

```
while (flag == 1)
```

```
interrupt: switch to Thread 2
```

```
flag = 1; // set flag to 1 (too!)
```

## Thread 2

```
call lock ()
```

```
while (flag == 1)
```

```
flag = 1;
```

```
interrupt: switch to Thread 1
```

Reason: test and set not atomic!

# Building Working Spin Locks with Test-And-Set

- Hardware support of a **test-and-set instruction** (atomic)

```
1 int TestAndSet(int *old_ptr, int new) {  
  
2   int old = *old_ptr;  
   // fetch old value at old_ptr  
  
3   *old_ptr = new;  
   // store 'new' into old_ptr  
  
4   return old; // return the old value  
5 }
```

# Building Working Spin Locks with Test-And-Set

- 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

# Building Working Spin Locks with Test-And-Set

```
1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0 indicates that lock is available, 1 that it is held
7     lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Compare with  
Loads/Stores:

```
while (mutex->flag == 1);
// spin-wait (do nothing)
mutex->flag = 1;
```

# Building Working Spin Locks with Test-And-Set

- By making both the **test** (of the old lock value) and **set** (of the new value) a single atomic operation, we ensure that only one thread acquires the lock. And that's how to build a working mutual exclusion primitive!
- Question:
  - Can it work on a non-preemptive processor?
  - No - will always spin

# Evaluating Test-And-Set Spin Locks

- **Correctness:**
  - Yes
- **Fairness**
  - No fairness guarantee
- **Performance**
  - Single processor:
    - Bad, 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
  - Multiple processor:
    - all right

# Evaluating Test-And-Set Spin Locks

- **Performance**

- Single processor:
  - Bad
- Multiple processor:
  - **All right**, 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

# Outline

- How lock is implemented
  - Controlling interrupts;
  - Loads/Stores
  - Test-and-Set;
  - Compare-and-Swap;
  - Fetch-and-Add;
  - Yield();
  - Using queues
- Requirement: be able to read and understand the C code. Not required to write.



# Building Working Spin Locks with Compare-And-Swap

- Hardware support of a **Compare-And-Swap instruction** (atomic)

```
1 int CompareAndSwap(int *ptr, int expected, int  
new) {  
2     int actual = *ptr;  
3     if (actual == expected)  
4         *ptr = new;  
5     return actual;  
6 }
```

- Test-and-Set

```
1 int TestAndSet(int *old_ptr, int new) {  
2     int old = *old_ptr;  
3     *old_ptr = new;  
4     return old;  
}
```

- Compare-and-Swap

```
1 int CompareAndSwap(int *ptr, int expected, int  
new) {  
2     int actual = *ptr;  
3     if (actual == expected)  
4         *ptr = new;  
5     return actual;  
6 }
```

# Building Working Spin Locks with Compare-And-Swap

- With the compare-and-swap instruction, we can build a lock in a manner quite similar to that with test-and-set.

```
1 void lock(lock_t *lock) {  
2     while (CompareAndSwap(&lock->flag, 0, 1) == 1)  
3         ; // spin  
4 }
```

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

# Outline

- How lock is implemented
  - Controlling interrupts;
  - Loads/Stores
  - Test-and-Set;
  - Compare-and-Swap;
  - Fetch-and-Add;
  - Yield();
  - Using queues
- Requirement: be able to read and understand the C code. Not required to write.

# Building Working Spin Ticket Locks with Fetch-And-Add

- Hardware support of a **Fetch-And-Add instruction** (atomic)

```
1 int FetchAndAdd(int *ptr) {  
2     int old = *ptr;  
3     *ptr = old + 1;  
4     return old;  
5 }
```

- Instead of a single value, this solution uses a **ticket** and **turn** variable in combination to build a lock.

```
1 typedef struct __lock_t {
2     int ticket;
3     int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7     lock->ticket = 0;
8     lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15 }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
```

# Building Working Spin Ticket Locks with Fetch-And-Add

- Correctness:
  - Yes
- Fairness
  - Yes
  - Note one important difference with this solution versus our previous attempts: it ensures progress for all threads.
- Performance
  - Still bad on single processor

# Outline

- How lock is implemented
  - Controlling interrupts;
  - Loads/Stores
  - Test-and-Set;
  - Compare-and-Swap;
  - Fetch-and-Add;
  - Yield();
  - Using queues
- Requirement: be able to read and understand the C code. Not required to write.



# Too Much Spinning: What Now?

- Hardware support got us pretty far: **working locks**, and even (as with the case of the ticket lock) **fairness** in lock acquisition.
- However, we still have a problem: **what to do when a context switch occurs in a critical section, and threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again?**

# A Simple Approach: “Just Yield, Baby”



- Photo from: <https://www.quickanddirtytips.com/health-fitness/mental-health/7-ways-to-feel-good-about-giving-up>

# A Simple Approach: “Just Yield, Baby”

- We assume there is an operating system primitive `yield()` which a thread can call when it wants to give up the CPU and let another thread run.
- A thread can be in one of three states (running, ready, or blocked); `yield` is simply a system call that moves the caller from the **running** state to the **ready** state, and thus promotes another thread to running. Thus, the yielding process essentially **deschedules** itself.

# A Simple Approach: “Just Yield, Baby”

```
1 void init() {  
2     flag = 0;  
3 }  
4  
5 void lock() {  
6     while (TestAndSet(&flag, 1) == 1)  
7         yield(); // give up the CPU  
8 }  
9  
10 void unlock() {  
11     flag = 0;  
12 }
```

# Outline

- How lock is implemented
  - Controlling interrupts;
  - Loads/Stores
  - Test-and-Set;
  - Compare-and-Swap;
  - Fetch-and-Add;
  - Yield();
  - Using queues
- Requirement: be able to read and understand the C code. Not required to write.

# Still problem

- Consider the case where there are many threads (say 100) contending for a lock repeatedly.
- In this case, if one thread acquires the lock and is preempted before releasing it, the other 99 will each call `lock()`, find the lock held, and yield the CPU.
- Assuming some kind of round-robin scheduler, each of the 99 will execute this run-and-yield pattern before the thread holding the lock gets to run again. While better than our spinning approach (which would waste 99 time slices spinning), this approach is still costly; the cost of a context switch can be substantial, and there is thus plenty of waste.

# Using Queues: Sleeping Instead Of Spinning

- We want some control over which thread next gets to acquire the lock after the current holder releases it.
- To do this, we will need
  - a little more OS support,
  - as well as a queue to keep track of which threads are waiting to acquire the lock.

# Using Queues: Sleeping Instead Of Spinning

- calls:
  - `park()` to put a calling thread to sleep, and
  - `unpark(threadID)` to wake a particular thread as designated by `threadID`.
- These two routines can be used in tandem to build a lock that puts a caller to sleep if it tries to acquire a held lock and wakes it when the lock is free.



# Using Queues: Sleeping Instead Of Spinning

- Code not posted in slides
- Read Chapter 28 of Operating Systems: Three Easy Pieces for details

# Acknowledgement

- Chapter 28
  - Operating Systems: Three Easy Pieces
- Java documentation
  - <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>

# Questions?