Locks

ICS432 - Fall 2008
Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

"Step on toes"?

- In previous lectures we've mentioned the notion that threads could *step on each other's toes*
- This does not happen for independent threads
  - they don't share variables/memory in read/write fashion
  - they don't wait for each other
- While independent threads are useful, typical concurrent applications need threads to share memory
- For instance, our application thread and the swing event dispatcher thread share memory
  - They can reference the same objects
- In this lecture we'll talk about the basic concepts for thread synchronization: Locks
- This and the next lecture are not Java-specific, but rather present general concepts
- We will see how Java deals with locks later

Step on Toes Example

- Consider two threads that both increment a variable x
  - Thread #1: ...; increment x; ...
  - Thread #2: ...; increment x; ...
- If you think of this in some low-level code, like assembly or byte code, the codes of the two threads are:

Thread #1	Thread #2
...	...
Load x into Register R	Load x into Register S
R = R + 1	S = S + 1
Store R into x	Store S into x
...	...

Step on Toes Example

- Problem: Threads can be context-switched at will by the O/S / JVM
- In principle: one can have an arbitrary interleaving of instructions
- Example:

Thread #1	Interleaving	Thread #2
	...	
...	Load x into Register R	...
Load x into Register R	Load x into Register S	Load x into Register S
R = R + 1	S = S + 1	S = S + 1
Store R into x	Store S into x	Store S into x
...	R = R + 1	...
	Store R into x	
	...	

Resulting computation: x+=1 as opposed to x+=2!

Likely Interleaving?

- The error in the previous slide is called "lost update"
- On a single-proc/single-core computer, with false concurrency, the odds that bad interleaving happens could be low
  - The O/S does not context-switch threads that fast
  - However, in a previous lecture we showed that the JVM's event-dispatch thread did in fact get its toes stepped on, so it can happen
  - And as usual, a bug that happens only once a month is terribly annoying
- On a multi-proc/multi-core system, i.e., when we have true concurrency, bad interleaving is much more likely

Race Condition

- The behavior of our example is non-deterministic
  - The end value of variable x could be either 1 or 2
- There is no way to know in advance what the result will be as it depends on
  - The architecture
  - The O/S, JVM
  - The load and state of the computer
- This lost update problem is an example of a race condition
  - The final result depends on the interleaving of the threads' instructions
  - Threads are "racing" to "get there first" and one cannot tell in advance which thread will win

Atomicity and mutual exclusion

- What we need is a mechanism that makes the updating of shared variable x atomic
  - Atomic: Whenever the update is initiated, we are guaranteed that it will go uninterrupted/undisturbed by other updates
- One can implement atomic updates to variable x by enforcing mutual exclusion
  - If one thread is updating variable x then **no** other thread can initiate an update of variable x
- This is a great idea, but how can we specify this in a program?
- Answer: with critical sections
- A critical section is a section of code in which only one thread is allowed at a time
- This is the most common and simplest form of synchronization for multi-threaded programs

Critical Sections

- One would like to write code that looks like this:

```
while(true) {
  enter_CS
  x++
  leave_CS
}
```

- We would like to have the following properties
  - Mutual exclusion: only one thread can be inside the CS
  - No deadlocks: one of the competing threads enters the CS
  - No unnecessary delays: a thread enters the CS immediately if no other thread is competing for it
  - Eventual entry: a thread that tries to enter the critical CS will enter it at some point
- We will see that these come both from:
  - the way the CS is implemented by the language+system
  - the way in which one writes concurrent applications

Critical Sections with Locks

- The concept of a critical section is binary
  - Either 0 threads are in the critical section
  - Or 1 thread is in the critical section
- Therefore, the critical section can be “controlled” with a boolean variable
- This variable is called a **lock**

```
while(true) {
    try to acquire lock  // wait if can't and keep trying
    x++
    release lock
}
```

- Just like going to the toilet in an airplane
  - While the lock is “red” wait
  - Then go in and set the lock to “red”
  - Then set the lock to “green” and leave

Locks

- Different languages have different ways to declare/use locks
  - We’ll see the syntax in Java and in C
- Let’s see the use of locks on several examples using a C-like syntax
  - Declaration: lock_t lock
  - Locking: lock(&lock)
  - Unlocking: unlock(&lock)

Locks for Data Structures

- A classical use of locks is to protect updates of linked data structures
- Example: Queue and threads
  - Consider a program that maintains a queue (of ints >0)
  - Thread #1 (Producer) adds elements to the queue
  - Thread #2 (Consumer) removes elements from the queue

Thread #1

```
int x;
while(1) {
  x = generate();
  insert(list,x);
}
```

Thread #2

```
int x;
while(1) {
  x = remove(list);
  process(x);
}
```

Queue Implementation

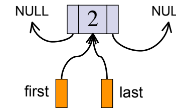
```
void insert (queue_t q, int x) {
  queue_item_t *item = (queue_item_t)
  calloc(1,sizeof(queue_item_t));
  item->value = x;
  item->next = q->first;
  if (item->next)
    item->next->prev = item;
  q->first = item;
  if (! q->last)  q->last = item;
}
```

Queue Implementation

```
int remove (queue_t q) {
    queue_item_t *item;
    int x;
    if (! q->last) return -1;
    x = q->last->value;
    item = q->last->prev;
    free(q->last);
    if (item) {
      item->next = NULL;
    }
    q->last = item;
    if (q->last == NULL) {
      q->first = NULL;
    }
    return x;
}
```

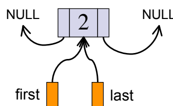
What bad thing could happen?

- Consider the following linked list



What bad thing could happen?

- Consider the following linked list

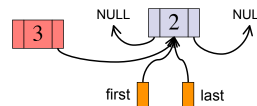


- The Producer calls insert(3)

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

What bad thing could happen?

- Consider the following linked list



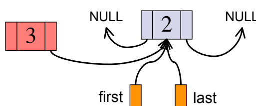
- The Producer calls insert(3)

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

context switch

What bad thing could happen?

- Consider the following linked list



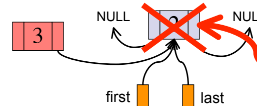
- The Consumer calls remove

```
...
item = q->last->prev;   // returns NULL
free(q->last);
if (item) {
. . .
```

context switch

What bad thing could happen?

- Consider the following linked list



Freed Memory

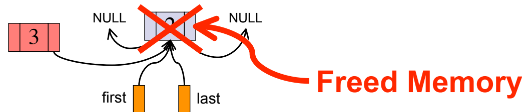
- The Consumer calls remove

```
...
item = q->last->prev;   // returns NULL
free(q->last);
if (item) {
. . .
```

context switch

What bad thing could happen?

- Consider the following linked list



- The Producer resumes

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
   item->next->prev = item;
q->first = item;
if (! q->last)
   q->last = item;
```

Freed Memory Access

So what?

- In this example, the producer updates memory that has been de-allocated
- In Java we would get an exception once in a while
- C doesn't zero out or track freed memory and we would get a segmentation fault once in a while
  - A third thread could have done a malloc and be given the memory that has been de-allocated
  - Then the producer could modify the memory used by that third thread
  - This could cause a bug in that third thread that could be very difficult to track
  - Basically, if you have threads and you get unexplained segmentation faults, you may have a race condition
    - Even if the segmentation fault occurs in a part of the code that has nothing to do with the relevant part of the code!
- Let's use locks and fix it

Simple Solution

```
lock_t lock;  // global variable

void producer() {
  int x;
  while(1) {
    x = generate();
    lock(lock);
    insert(list,x);
    unlock(lock);
  }
}

void consumer() {
  int x;
  while(1) {
    lock(lock);
    x = remove(list);
    unlock(lock);
    process(x);
  }
}
```

Simple Solution

- Important: we use a single lock that is referenced/used by both threads
- The solution is simple: place the lock around all calls that manipulate the queue
  - Sometimes determining what calls and code segments modify a structure requires some thought
- The critical section is then the whole queue implementation
- This is the typical strategy when using a non-thread-safe implementation of the queue abstract data type
- To produce a thread-safe implementation, one needs to create critical sections in the queue implementation

Thread Safe Queue

```
void insert (queue_t q, int x) {
  lock(q.lock);  // each queue has its lock
  queue_item_t *item = (queue_item_t)
  calloc(1,sizeof(queue_item_t));
  item->value = x;
  item->next = q->first;
  if (item->next)
    item->next->prev = item;
  q->first = item;
  if (! q->last)  q->last = item;
  unlock(q.lock);
}
```

Thread-Safe Queue

```
int remove (queue_t q) {
    queue_item_t *item;
    int x;
    lock(q.lock);
    if (! q->last) return -1;
    x = q->last->value;
    item = q->last->prev;
    free(q->last);
    if (item) {  item->next = NULL;  }
    q->last = item;
    if (q->last == NULL) {
      q->first = NULL;
    }
    unlock(q.lock);
    return x;
}
```

Critical Sections and Performance

- An easy way to make code thread safe is to put a lot of things in critical sections
  - "I don't really know what these functions do, I'll use a single lock and put all calls in a critical section"
- Problem: Critical sections reduce concurrency
  - In a critical section there can be only one thread
- At the extreme, the code becomes purely sequential
- For better concurrency one should have short critical sections:
  - Use many locks whenever possible to generate many shorter independent critical sections rather than a few longer ones
    - Two threads can of course be in two different critical sections at the same time
    - In our queue example, a producer can modify queue q1 while a consumer can modify queue q2 (we used "q.lock")
  - Look at the code carefully and put only what's necessary between calls to lock() and unlock()

Better Thread Safe Queue

```
void insert (queue_t q, int x) {
   // lock(q.lock);
   queue_item_t *item = (queue_item_t)
   calloc(1,sizeof(queue_item_t));
   item->value = x;
   lock(q.lock);
   item->next = q->first;
   if (item->next)
    item->next->prev = item;
   q->first = item;
   if (! q->last)  q->last = item;
  unlock(q.lock);
}
```

taken outside of the CS

A consumer can operate on the queue while a producer is allocating memory for a new element

➡ more concurrency

Good General Principles

- Try to make critical sections as short as possible
- Try to use different locks for different data items
  - See our queue example in which we attached a different lock to each queue
  - One concern is that with many locks there is more memory consumption and there may be more overhead
    - Typically ok given the benefits of more concurrency
- Try to avoid critical sections by replicating or splitting shared data whenever possible
  - It may be that data structures can be reorganized so that threads don't step on each others' toes
  - e.g., use two separate counters to avoid the "lost update" problem in our first simple example, and sum them up when both threads have completed

DeadLocks

- Deadlock: a common problem when synchronizing threads with *multiple* locks
  - you write your program with many threads and shared variables/structures protected by locks
  - you run it
  - at some point, it's stuck
- Deadlock can happen with nested critical sections
- Classic example (which may deadlock but perhaps not always):

```
. . .
lock(lock1)
. . .
lock(lock2)
. . .
unlock(lock2)
. . .
unlock(lock1)
. . .
```

```
. . .
lock(lock2)
. . .
lock(lock1)
. . .
unlock(lock1)
. . .
unlock(lock2)
. . .
```

Deadlocks

- The previous example is trivial
- But in practice code can become complex and deadlocks do happen and require careful debugging
- We will discuss a case-study in a couple of lectures that will highlight many thread synchronization problems:
  - deadlocks
  - starvation
- Clever coding techniques and implementation strategies make it possible to obtain live and fair executions
  - The program always makes progress when it can
  - No thread is starving and all threads get to run equally

Implementing Lock

- At this point we have some idea of how to use lock() and unlock() to create critical sections and ensure safe concurrency
- Question: how does one implement lock???
  - Granted, you will probably not need to as languages/systems provide them
  - But it's interesting to have some idea of how things work
  - And it will be our first attempts at reasoning about concurrency
- There are two kinds of lock implementations
  - software solutions
  - hardware solutions

Software Spin Locks: v0

- A Spin Lock is simply a boolean variable
- unlock()
  - simply set the variable to 0
- lock()
  - check whether the variable is equal to 0
  - if it is equal to 1 check again
  - if it is equal to 0, set it to 1 and enter and continue
- Simple implementation?

```
void unlock(int *lock) {
  *lock = 0;
}
```

```
void lock(int *lock) {
  while (*lock) yield();  // spin
  *lock = 1;
}
```

- Note the use of yield(), which is probably better but not mandatory in a system that implements time-slicing
- What's wrong with this implementation?

Software Spin Locks: v0

```
void lock(int *lock) {
  while (*lock) yield(); // spin
  *lock = 1;
}
```

- Code for lock() in assembly pseudo-code

```
spin:   LD R1, <lock>
        BNEZ R1, spin
        SDI #1, <lock>
```

Thread #1 **Thread #2**

```
spin:   LD R1, <lock>
```

Software Spin Locks: v0

```
void lock(int *lock) {
  while (*lock) yield(); // spin
  *lock = 1;
}
```

- Code for lock() in assembly pseudo-code

```
spin:   LD R1, <lock>
        BNEZ R1, spin
        SDI #1, <lock>
```

Thread #1

```
spin:   LD R1, <lock>
```

Thread #2

```
spin:   LD R1, <lock>
        BNEZ R1, spin
        SDI #1, <lock>
```

Software Spin Locks: v0

```
void lock(int *lock) {
  while (*lock) yield(); // spin
  *lock = 1;
}
```

- Code for lock() in assembly pseudo-code

```
spin:   LD R1, <lock>
        BNEZ R1, spin
        SDI #1, <lock>
```

Thread #1

```
spin:   LD R1, <lock>
        BNEZ R1, spin
        SDI #1, <lock>
```

Thread #2

```
spin:   LD R1, <lock>
        BNEZ R1, spin
        SDI #1, <lock>
```

Both threads are now in the critical section!!

Software Spin Lock

- There is a race condition in the lock() function on the boolean lock variable itself!
  - Adding another lock on the lock would only push the problem down one level, and so on...
- One possible solution would be to used a “turn-based” system
  - A variable alternates between 0 and 1
  - A value of 0 indicates that Thread #1 should get access to the critical section
  - A value of 1 indicates that Thread #2 should get access to the critical section
  - Initially the value is (arbitrarily) set to 0
- Let's look at the code

Software Spin Lock: v1

```
void unlock(int *lock, int id) {
  *lock = 1 - id;
}
```

```
void lock(int *lock, int id) {
  while (*lock != id) yield();  // spin
  *lock = id;
}
```

- Thread #1 calls the functions passing 0 as an argument, and thread #2 calls the functions passing 1 as an argument
- The code above solves the problem of the previous implementation
  - the two threads cannot enter the critical section because only a single thread can have its id equal to the lock
- What is the problem?

Software Spin Lock: v1

```
void unlock(int *lock, int id) {
  *lock = 1 - id;
}
```

```
void lock(int *lock, int id) {
  while (*lock != id) yield();  // spin
  *lock = id;
}
```

- The problem is starvation
- Consider the following sequence of locks and unlocks:
  - Thread #1: lock(0);
  - Thread #1: unlock(0);
  - Thread #1: lock(0);   // blocks!
- Thread #1 is blocked until Thread #2 goes into the critical section
- Threads have to alternate in the critical section
  - Because it's turn-based
- This goes against the principle of "no unnecessary delays"

Software Spin Lock: v2

```
void unlock(lock_t lock, int id) {
  lock.flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  while (lock.flag[1-id] == true) yield();
  lock.flag[id] = true;
}
```

- The idea here is to use two variables inside the lock:

```
typedef struct {
  boolean flag[2];
} lock_t;
```

  - Initialize at {false, false}
- This way, we avoid the alternating problem of the previous implementation
- Is it correct?

Software Spin Lock: v2

```
void unlock(lock_t lock, int id) {
  lock.flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  while (lock.flag[1-id] == true) yield();
  lock.flag[id] = true;
}
```

- The idea here is to use two variables inside the lock:

```
typedef struct {
  boolean flag[2];
} lock_t;
```

  - Initialize at {false, false}
- This way, we avoid the alternating problem of the previous implementation
- Is it correct?
- Nope:
  - The two threads enter lock() "at the same time"
  - They both see the other's flag set to false and proceed
  - We now have two threads in the critical section!

Software Spin Lock: v3

```
void unlock(lock_t lock, int id) {
  lock.flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  lock.flag[id] = true;
  while (lock.flag[1-id] == true) yield();
}
```

- To avoid the problem from before we swapped the two statements in function lock()
  - There is no interleaving of the executions that can lead to both threads entering the critical section simultaneously

```
lock.flag[0] = true;                 lock.flag[1] = true;
while(lock[1] == true) yield();      while(lock[0] == true) yield();
```

- But now we have a new problem

Software Spin Lock: v3

```
void unlock(lock_t lock, int id) {
  lock.flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  lock.flag[id] = true;
  while (lock.flag[1-id] == true) yield();
}
```

- To avoid the problem from before we swapped the two statements in function lock()
  - There is no interleaving of the executions that can lead to both threads entering the critical section simultaneously

```
lock.flag[0] = true;                 lock.flag[1] = true;
while(lock[1] == true) yield();      while(lock[0] == true) yield();
```

- But now we have a new problem: deadlock!
  - Both threads set their variable to true
  - Then they both spin forever
- Again, unlikely but possible, esp. with true concurrency

Software Spin Lock: v4

```
void unlock(lock_t lock, int id) {
  lock.flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  lock.flag[id] = true;
  while (lock.flag[1-id] == true) {
    lock.flag[id] = false;
    yield();
    lock.flag[id] = true;
  }
}
```

- The idea here is to fix the problem from v3 by having threads back off when they realize they're both entering the function at the same time
  - If the other's flag is set to true, I set mine to false, let the other run for a while, and set mine to true again and check on the other's flag
- There is STILL a problem here!

Software Spin Lock: v4

```
void unlock(lock_t lock, int id) {
  lock.flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  lock.flag[id] = true;
  while (lock.flag[1-id] == true) {
    lock.flag[id] = false;
    yield();
    lock.flag[id] = true;
  }
}
```

- The problem is livelock!
  - A kind of deadlock in which threads are in an infinite (or very long) sequence of blocking and unblocking
- Threads could be in locked step
  - They both set their flags to true
  - They both set their flags to false
  - They both set their flags to true
  - . . .
- With false concurrency, this is virtually impossible
- With true concurrency, the _livelock_ could last a long time

Software Spin Lock: v5

```
void unlock(lock_t lock, int id) {
  lock.flag[id] = false;
  lock.turn = 1-id;
}
```

```
void lock(lock_t lock, int id) {
  lock.flag[id] = true;
  while (lock.flag[1-id] == true) {
    if (turn != id) {
      lock.flag[id] = false;
      while (turn != id) yield();
      lock.flag[id] = true;
    }
  }
}
```

- We add a “turn” variable to the lock structure

```
typedef struct {
  boolean flag[2];
  int turn;
}
```

- The threads take turns backing off!
- This is a very good solution [Dekker, 1960’s]
  - But it does allow starvation in some situations

Software Spin Lock: v6

- In 1981 Peterson came up with a complete *and simpler* solution:

```
typedef struct {
  boolean flag[2];
  int last;
} lock_t;
```

- The last field tracks which thread last tried to enter the CS
- This is the thread that is delayed if both threads compete
  - Remove the starvation problem of v5

```
void unlock(lock_t lock, int id) {
  lock.flag[id] = false;
}
```

```
void lock(lock_t lock, int id) {
  lock.flag[id] = true;
  lock.last = id;
  while (lock.flag[1-id] == true && last == id)
    yield();
}
```

Exponential Back off

- One problem with spin locks is that they consume CPU cycles
  - A thread is in an infinite loop trying to acquire the lock
- A fix (a “hack” really) is to have threads back off for exponentially increasing (but bounded) periods of time
  - Reduces responsiveness
  - But saves CPU cycles which may benefit the thread in the critical section
- No matter what, it’s always a good idea to have very short critical sections so that threads spend very little time in lock()

Software Locks: Conclusion

- It turns out that having a good solution required some thought
- Thanks to Peterson we have one
  - Note that formally proving that it is a correct solution is not easy
  - But in this course we won’t touch on the theory
  - Just know that detecting race conditions, deadlocks and starvations by looking at the code is NP-hard
- But what about more than two threads?
- Turns out things get much more complicated
- Example: the bakery algorithm (by Lamport)
  - Analogous to a bakery with a machine dispensing tickets to customers
  - Cleverly designed to avoid all the problems we have seen with v1, v2, v3 and v4
  - Accommodates an arbitrary number of threads

Hardware Solutions

- The software solutions are interesting
  - Especially because the same principles and reasoning applies when writing concurrent applications that use locks
- But they could be time/memory consuming
- As usual, a hardware solution can solve many of the problems of a software solution in a way that’s simpler and faster
  - At least simple for software developers
- There are two possible options
  - Disabling interrupts
  - Atomic instructions

Disabling Interrupts

- This is extremely simple: to avoid badly timed context switches, just disable context switches!
- Context switches are implemented via interrupts
- All systems have instructions to disable and enable interrupts
  - But typically these instructions are reserved for the O/S running in kernel mode
- It could be very dangerous
  - e.g., disable interrupts and go into an infinite loop in the critical section
- Therefore, this is not really an acceptable solution
- Furthermore, it wouldn't work on a multi-CPU or multi-core architecture
  - Interrupts are local to a processor
- So although attractive because of simplicity it can only be used in the code of the O/S

Atomic instructions

- Let's look at our first naive implementation

```
void lock(int *lock) {
  while (*lock) yield(); // spin
  *lock = 0;
}
```

- The assembly was

```
spin:    LD R1, <lock>
         BNEZ R1, spin
         SDI #0, <lock>
```

- Therefore, between the *loading*, the *testing* and the *setting* the value may have changed
- If we had an atomic "test and set" instruction, we could be sure that the if is done correctly

Test&Set Instruction

- Most processors provide atomic instructions that do multiple things at once
- Example: T&S R1, 0(R2)
  - Equivalent to:
    - Load memory cell 0(R2) into R1
    - If R1 is 0, store 1 into memory cell 0(R2)
  - Can be implemented by locking the memory bus so that no other memory access can occur in between the load, test, and store
- One can then write the assembly for lock():

```
Lock: T&S   R1, <lock>
      BNZ   R1, Lock
      RET
```

Conclusion

- Race conditions are common bugs in concurrent programs
  - non-deterministic and thus difficult to detect
- To prevent race conditions one needs locks
  - can be implemented in software or in hardware
- New issues arise
  - Deadlocks
  - Livelocks
  - Starvation
- We discussed these issues in the context of implementing locks themselves
- But they arise in applications as well
  - More on this in later lectures