Threading Intro, part 2

1 Synchronization

So, at this point we know that threaded code can interfere with itselfin unpleasant ways. Race conditions should be avoided, so code thatmanipulates shared data should be organized into critical sections.Fine, but how do you actually prevent other threads from executingtheir critical sections when your thread is in the middle of itscritical section? As often in systems, there are several ways toaccomplish this, and each with its own advantages and disadvantages.

> **Conditional** - The thread waits until some conditionbecomes true.
>
> **Mutual Exclusion** - Critical sections are guarded bymutex's, and only the thread that holds the mutex may execute thecritical section (all other threads must wait).

We've already encountered examples of conditional synchronization.When a process asks the OS for a file to be read off disk, the OS hasthe process wait until the file data is available in memory. Mutualexclusion is trickier. A thread must claim a mutex in order to executea critical section, so the protocol for acquiring and releasingmutexes must be interleaving-proof (i.e. it must either not beinterrupted, or immune to interference in the presence ofinterruptions).

At a high level of abstraction, a thread with a critical section lookslike:

```
...
NCS1   //non-critical code
Entry1 //start of critical section 1
CS1    //actual code of critical section
Exit1  //leaving critical section
NCS2   //non-critical code
Entry2
CS2    //Second critical section
Exit2
NCS3
...
```

A mutex must ensure that only one thread is executing `CS1` at any given time, and a seperate mutex must ensure that only one thread is executing `CS2` at any given time. That is absolutely essential. However, it is desireable for a mutex to be:

> **Efficient** Entry and Exit should be fast, and acquiring a mutex for `CS` 1 or 2 shouldn't impact other threads executing the non-critical code (i.e. only code that's executing critical sections should pay the cost of acquiring a mutex).
>
> **Progress** If multiple threads are attempting to execute a critical section, at least one of them gets to (i.e. some work is always getting done).

No Starvation Every contenting thread eventually executes its critical section (i.e. no thread is starved of mutexes).

The challenge, of course, is implementing a mutual exclusion primitive that is correct, efficient, progress-ensuring and fair all at the same time. This is, as you may imagine, quite difficult.

2 An algorithmic example: Doubly Linked Lists

Here's a high level example of thread interferance. Let's just look at a doubly-linked list of integers. The list is sorted (least to greatest), and insertions into the list should respect the order. If we assume that a node is defined thus:

```
struct Node{
  struct Node* prev;
  int val;
  struct Node* next;
};
```

Then, the find method would look something like:

```
struct Node* find(int key, struct Node* head){
  struct Node* current = head;

  while((current != NULL) && (current->val < key)){
    current = current->next;
  }

  //return NULL at end-of-list or the next highest node
  return current;
}
```

And, insert would look something like:

```
struct Node* insert(int key, struct Node* head){
  struct Node* current = head;
  struct Node* last = head;
  struct Node* new_node = malloc(sizeof(struct Node));
  new_node->next = new_node->prev = NULL;
  new_node->val = key;

  //STEP 1: finding the spot in the list
```

```
  while((current != NULL) && (current->val < key)){
    last = current;
    current = current->next;
  }

  //STEP 2: stitching the new node into the list, part 1
  new_node->prev = last;
  if(last != NULL){
    last->next = new_node;
  }

  //STEP 3: stitching the new node into the list, part 2
  new_node->next = current;
  if(current != NULL){
    current->prev = new_node;
  }

  if(last == NULL){ //List was empty
    return new_node;
  }else{
    return head;
  }
}
```

Now consider two threads. Thread 1 is executing an insert with a key of 17 into a two element list (with keys 1 and 765), and Thread 2 is executing a find on the same list with a key of 765. Now, if the interleaving is such that thread 1 completes the insertion before thread 2 starts the find, then there is no conflict. Additionally, if thread 2 completes the find before thread 1 starts manipulating pointers in the list (which starts at step 2), then there is no conflict (thread 2 will find the last node). But, if the threads are interleaved so that thread 2 starts navigating the list after `STEP 2` but before `STEP 3`, then there can be a problem.

Consider the following diagram, which illustrates the state of the list right after thread 1 has completed step 1:

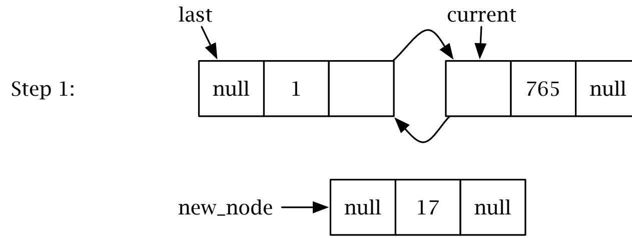

Figure 1: Step 1 Completed

At this point, a find can still navigate the entire list without difficulty (thread 2 will still find 765).

Now consider the following diagram, which illustrates the state of the list right after thread 1 has completed step 3:

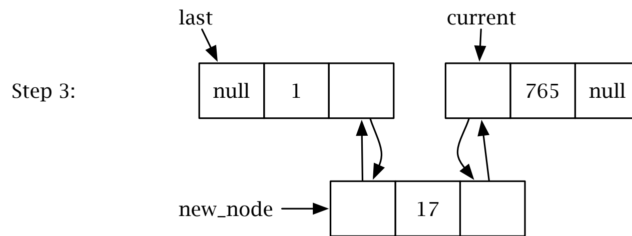

Figure 2: Step 3 Completed

Here, too, find will still succeed. The list is one node longer than in the previous case, but still fully navigable (i.e. the doubly-linked list invariants are still holding). Now, consider the following diagram, which illustrates the list just after thread 1 has completed step 2 (but before step 3):

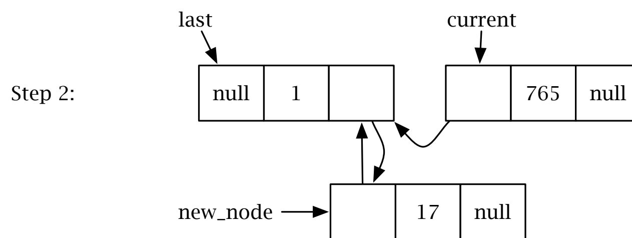

Figure 3: Step 2 Completed

What happens if thread 2 just happens to be scheduled right at this moment? Thread 2 will be able to navigate the partial list, but will actually not be able to reach the last node (765) because

the "stitching in" of the new node is not complete. In fact, find will see the NULL value in the new node's next pointer and assume that the end of the list has been reached. This is bad, when depending on the interleaving, a thread can "lose" a datum in a data structure.

2.1 Fixing the problem

How do we fix this? Well, mutual exclusion should work. It seems like the list-manipulation section of `insert`'s code is a critical section. Ok, let's say we have a mutex labeled `list_mutex`, then if we insert a lock just before step 2 and an unlock just before the return we should be good right? The answer is no. To understand why, think about which code will call the locking code. At this point, only insert calls will try and lock the mutex. So, find will not be prevented from navigating the list while another thread is executing the critical section within the insert method.

So, we obviously need to synchronize threads running inserts and threads running finds. If we inserted a lock call as the first line in find, and an unlock just before the return (and the lock/unlock calls try to lock/unlock the `list_mutex` mutex). Then only one thread can be inserting and/or find-ing at the same time. At this point, the code is preventing the kind of interferance between insertion and finding that we saw before. But here are some questions: **Is this efficient?**, **Is this completely correct? What about two insertions running simultaneously?**

2.2 Fixing the problem without locks

Here's another question: do you have to use locks? The answer is actually no. Imagine if the code to set the next and prev pointers in the new node were moved before step 2, like so:

```
//STEP 1: finding the spot in the list
while((current != NULL) && (current->val < key)){
  last = current;
  current = current->next;
}

new_node->prev = last;
new_node->next = current;

//STEP 2: stitching the new node into the list, part 1
if(last != NULL){
  last->next = new_node;
}

//STEP 3: stitching the new node into the list, part 2
if(current != NULL){
  current->prev = new_node;
}
```

Now, even if the inserting thread is interrupted between steps 2 and 3, a finding thread can still successfully navigate the list. This situation is illustrated by the following diagram:

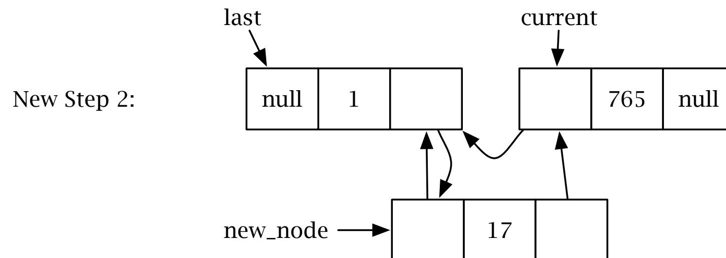

Figure 4: Step 2, Completed new order of operations

This is one of those situations where re-ordering the internal operations of a method can fix a threading conflict. But now here's a new question: **Does this work for all cases?** (hint: think about two inserts running concurrently).

3 Mutual Exclusion Mechanisms

The challenge is implementing something fast and correct in terms of the hardware primtives provided by a machine, and then exporting that to the programmer as a language primitive or a function call. To illustrate the difficulties this presents, we'll cover some partial (read: broken) solutions to mutual exclusion

3.1 Idea 1: Flags

Here, the basic idea is to associate a flag with a critical section. If a thread is executing that critical section, the flag is set and other threads will have to wait until the flag gets lowered again. Here's some C-like pseudo-code to illustrate the idea:

```
entry:
 test: while(in_cs1){
         //do nothing
       }
 set:  in_cs1 = true;
...
exit:  in_cs1 = false;
```

Assume, for the moment that the `test` and `set` portions of `entry` are atomic. Let's test this implementation. Assume that there are two threads, A and B that are executing this code. Here's a possible interleaving (assuming that the flag is false initially):

```
A: test
A: set
A: some CS instructions
CONTEXT SWITCH
B: test //fails, so loops
CONTEXT SWITCH
A: finish critical section
A: exit
CONTEXT SWITCH
B: test //loop exits
B: set
B: critical section
B: exit
```

In this case, the mutex works. Thread A tests and sets the flag before B does and so A excludes B. You can easily see that if you reversed the order of A and B, then B would execute first. But what would happen if there was a context switch between A's test and A's set?

```
A: test //succeeds
CONTEXT SWITCH
B: test //succeeds
B: set
B: some critical section code
CONTEXT SWITCH
A: set
A: CS instructions
A: exit
CONTEXT SWITCH
B: finish critical section
B: exit
```

Whoops! If you interrupt the entry code between the test and the set, you can violate the correctness property for mutual exclusion (namely, that only one thread execute a critical section at once). Obviously, flags implemented this way won't hack it.

3.2 Idea 2: Taking turns

This mechanism assigns an integer "my_turn" to each thread, ensuring that each thread will "wait its turn". The critical section would then have a turn variable associated with it, set to the `my_turn` value of the thread currently executing it. Consider the following code:

```
entry:
 test: while(turn <= (my_turn - 1)){
         //do nothing
```

```
        }
...
exit:  turn = my_turn;
```

In this case, A's my_turn would be 1 and B's would be 2. Initially, turn would be 0 (i.e. no thread executing it). In this case, there's no problem if A or B is interrupted right after reading turn. B can only execute the critical section after A has executed it. However, what happens if B is scheduled before A? B will have to wait for A to enter, execute and exit the critical section before B can. Even if B wins the race to the entry code. B will just have to sit and spin at enter until A finally gets around to executing. This violates the efficiency criteria (a thread is blocked from executing a critical section even though no other thread is executing it). In addition, it barely satisfies the progress criteria. This is becaue taking turns establishes an arbitrary order on threads that may not have any relation to scheduling order. Taking turns is correct, but inefficient.

4 Hardware Mechanisms

As the previous examples showed, mutex implementation may require multiple machine instructions. This is bad because hardware can only ensure that individual instructions execute atomically. The flag example illustrated that the problem is particularly acute if a thread is interrupted between reading a shared variable and writing it. Modern hardware comes equipped with special 'interlock' instructions that perform conditional read/write operations atomically.

> **Test and Set** - A simple primitive, tests a memory value against a local (register) value and sets the memory location to a predefined value (usually 0, 1, or all 1s).
>
> **Load-linked Store Conditional** - A two instruction combo, the linked load loads a value into a register, and the store conditional succeeds if no other thread has modified or read the memory value since the last linked load. This can result in spurious failure.
>
> **Compare and Swap** - The classic atomic instruction. Takes three values: a memory address, a test value, and a set value. If the memory address contains the test value, it is set to the set value. If the memory address does not contain the test value, CAS fails.

With these specialized concurrency instructions, it is possible to implement lightweight flag-style mutexes without sacrificing correctness. For example, the flag code from above could be reworked thus (assuming that compare and swap was available as a function
`int CAS(int* addr, int test, int val)`:

```
entry:
 test: while(!CAS(&in_cs1, 0, 1)){
          //spin
        }
...
exit:  in_cs1 = 0;
```

In this case, `in_cs1` holds the flag. Assuming CAS returns 1 on success, this code will loop (the technical term is spin) waiting for the flag to drop. When the flag becomes false (0), then CAS will succeed. And because CAS is atomic, the thread won't be interrupted between reading `in_cs1` and writing out the new value. Therefore this code is actually correct! Note too, that the exit code is unchanged. Because we can assume that `in_cs1` is 1, we don't have to set it with another CAS.

Is this code perfect? No. This can cause starvation as threads are now racing to the entry point for the critical section. Because of this race, fast or "lucky" threads will get to execute the critical section first. This code doesn't reward threads who've been waiting longer for the flag, and is unfair in that sense.

How do you make it fair? One solution is to have the threads sleep for exponentially increasing amounts of time (so-called exponential backoff). This is statistically fair, but it can be inefficient. So, how do you make this fair without using exponential backoff? By adding queues of course! Abstractly, what you'd like to do is attach a list of waiting threads to the lock (which is actually what the flag has become). When a thread goes to acquire the lock and finds it unavailable, rather than just spinning and wasting cycles, the thread will append a record to the list of waiting threads and then sleep. Then, when the thread executing the critical section exits, it will wake one of the sleeping waiters and pass the lock onto it. In pseudo-code:

```
entry:
 test: if(!CAS(&in_cs1, 0, 1)){
         appendToQueue();
         sleep();
       }
...
exit: if(waitersInQueue()){
        ThreadRecord t = dequeueThread();
        handLockToThread(t, &in_cs1);
        wake(t);
      }else{
        in_cs1 = 0;
      }
```

This code is intentionally vague, because the specifics of this kind of system will dictate how precisely the code is written. But the basic idea is the same. Rather than just reseting the flag on exit, if waiting threads are present, the thread that has been waiting the longest is handed the lock and then woken up. Note that this thread will resume execution immediately after the sleep statement in the entry code. Note also that the `handLockToThread` code doesn't reset the flag to 0. If it did, that would allow a "lucky" thread to come and grab the lock (via a CAS) out from under the recently awoken thread. This is bad because: A) now you can potentially have two threads executing a critical section simultaneously, B) it is definitely unfair.

4.1 A Real System Example

In practice, locking schemes like this are implemented with a lot of low-level trickery. The scheme I describe here is a simplification of a standard trick. First, a lock is an integer in memory somewhere. The actual lock flag is one of the two low bits in that integer. The upper 30 bits, however are a pointer to the waiting queue for that lock. When the lock is acquired, the low bit is set to 1, but the upper 30 bits still point to the waiting queue. That way, if a thread fails to acquire the lock, it can just mask off the lower 2 bits and find the waiting queue to append itself to. This is illustrated below:

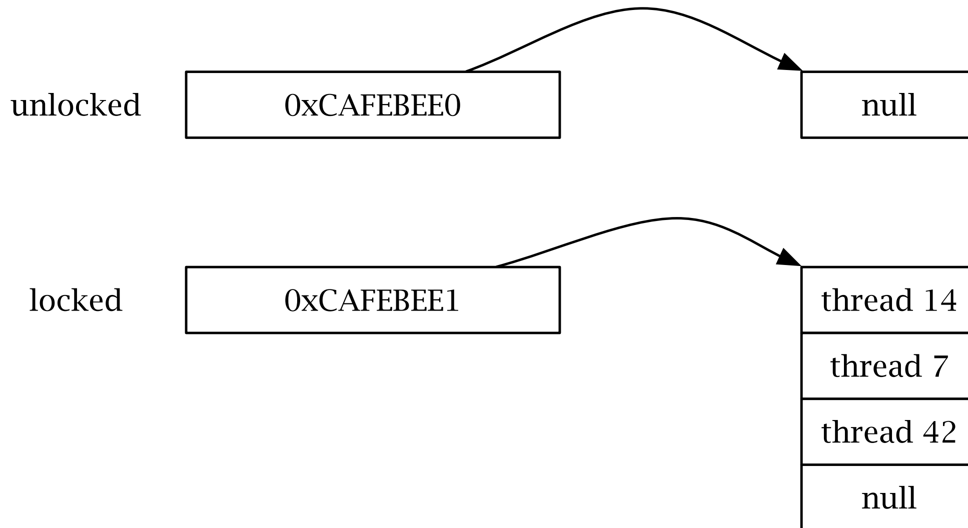

Figure 5: A fancy, low-level lock

There's actually a problem with this simple scheme. The problem is that the wait queue itself is a data structure and that threads inserting themselves into it may be interrupted half-way through. This is bad. There are actually several different solutions to this problem. The simplest and least elegant is to gaurd this queue with an unfair, simple flag-style spinlock. A more elegant and efficient solution is to implement the waiting queue in a lock-free fashion. Lock-free, in this case, means that the modifications to the data structure are all done with atomic swap instructions (CAS, in this case), that are carefully ordered so that no destructive interferance occurs. This type of coding is intensely tricky, however, and way too much to get into in an introductory course.

Assuming that all the queue details are handled in the `appendToQueue` and `dequeueThread` functions, the code would look something like:

```
entry:
 test: int free_val = in_cs1 & 0xFFFFFFFC; //mask off lower 2 bits
       int locked_val = free_val | 0x1;
       if(!CAS(&in_cs1, free_val, locked_val)){
```

```
          appendToQueue((void*)free_val);
          sleep();
        }
...
exit: ThreadRecord t;
      if(t = dequeueThread()){
        //at this point, there were waiters in the queue
        // and we just got the one that had been waiting the longest
        wake(t);
      }else{
        in_cs1 = 0;
      }
```

In this case, we read in the lock value first. Because we know that the low-order bit is 0 in the unlocked state, we can just mask off the lower bits (thereby setting them to 0). We then take the calculated value for the free lock and calculate the locked value from it. So now, instead of CAS-ing a boolean flag, we test for the free value and try and set it to the locked value. Note, too, the change to the exit code. `dequeueThread()` now will return NULL if the queue is empty, this avoids a problem in the previous code (if the exiting thread was interrupted just after the `waitersInQueue` call, but before the `dequeueThread()` call and that the wait queue had been emptied by some other thread).

Industrial systems, such as JVMs use tricks similar to this one to keep locking overhead low. Why? Because the cheaper locks are to use, the better multi-threaded code will scale.

4.2 Issues with Hardware Exclusion

There are some issues with hardware interlock primitives. The main issue is that a CAS-style instruction can play havoc with the hardware memory system. Most memory controllers (like microprocessors), reorder non-conflicting memory requests in order to improve throughput. However a CAS acts as a barrier to reordering (for correctness reasons). Additionally, on a multiprocessor system, CASes can tie up the memory system on all processors, as the system has to make sure that CAS has unfettered access to main memory. This is the reason why some systems choose to use alternative primitives, such as load-linked/store-conditional. However, that primitive suffers from so-called spurious failure. When a CAS fails, you can be certain that the value in memory was not what you claimed it should be. When a store-conditional fails, it could simply be because another thread loaded the value. And not all reads to shared data are interfering.