Section 5: Thread Synchronization

CS162

February 22, 2019

Contents

1 Vocabulary

- **Thread** - a thread of execution is the smallest unit of sequential instructions that can be scheduled for execution by the operating system. Multiple threads can share the same address space, but each thread independently operates using its own program counter.
- **Atomic operation** - An operation that appears to be indivisible to observers. Atomic operations must execute to completion or not at all.
- **Critical section** - A section of code that accesses a shared resource and must not be concurrently run by more than a single thread.
- **Race condition** - A situation whose outcome is dependent on the sequence of execution of multiple threads running simultaneously.
- **test_and_set** - An atomic operation implemented in hardware. Often used to implement locks and other synchronization primitives. In this handout, assume the following implementation.

```
int test_and_set(int *value) {
    int result = *value;
    *value = 1;
    return result;
}
```

  This is more expensive than most other instructions, and it is not preferable to repeatedly execute this instruction.
- **Lock** - Synchronization variables that provide mutual exclusion. Threads may acquire or release a lock. Only one thread may hold a lock at a time. If a thread attempts to acquire a lock that is held by some other thread, it will block at that line of code until the lock is released and it successfully acquires it. Implementations can vary.
- **Scheduler** - Routine in the kernel that picks which thread to run next given a vacant CPU and a ready queue of unblocked threads. See next_thread_to_run() in Pintos.
- **Priority Inversion** - If a higher priority thread is blocking on a resource (a lock, as far as you're concerned but it could be the Disk or other I/O device in practice) that a lower priority thread holds exclusive access to, the priorities are said to be inverted. The higher priority thread cannot continue until the lower priority thread releases the resource. This can be amended by implementing priority donation.
- **Priority Donation** - If a thread attempts to acquire a resource (lock) that is currently being held, it donates its effective priority to the holder of that resource. This must be done recursively until a thread holding no locks is found, even if the current thread has a lower priority than the current resource holder. (Think about what would happen if you didn't do this and a third thread with higher priority than either of the two current ones donates to the original donor.) Each thread's effective priority becomes the max of all donated priorities and its original priority.
- **Condition Variable** - A synchronization variable that provides serialization (ensuring that events occur in a certain order). A condition variable is associated with:
  - a lock (a condition variable + its lock are known together as a **monitor**)
  - some boolean condition (e.g. `hello < 1`)
  - a queue of threads waiting for the condition to be true

In order to access any CV functions **OR** to change the truthfulness of the condition, a thread must/should hold the lock. Condition variables offer the following methods:

– **cv_wait(cv, lock)** - Atomically unlocks the lock, adds the current thread to **cv**'s thread queue, and puts this thread to sleep.
– **cv_notify(cv)** - Removes one thread from **cv**'s queue, and puts it in the ready state.
– **cv_broadcast(cv)** - Removes all threads from **cv**'s queue, and puts them all in the ready state.

When a **wait()**ing thread is notified and put back in the ready state, it also re-acquires the lock before the **wait()** function returns.
When a thread runs code that may potentially make the condition true, it should acquire the lock, modify the condition however it needs to, call notify() or broadcast() on the condition's CV, so waiting threads can be notified, and finally release the lock.
Why do we need a lock anyway? Well, consider a race condition where thread 1 evaluates the condition C as false, then thread 2 makes condition C true and calls **cv.notify**, then 1 calls **cv.wait** and goes to sleep. Thread 1 might never wake up, since it went to sleep too late.

- **Hoare Semantics** - (In terms of condition variable) Wake a blocked thread when the condition is true and transfer control of the CPU and ownership of the lock to that thread immediately. This is difficult to implement in practice and generally not used despite being conceptually easier to deal with.

- **Mesa Semantics** - (In terms of condition variable) Wake a blocked thread when the condition is true, with no guarantee that the thread will execute immediately. The newly woken thread simply gets put on the ready queue and is subject to the same scheduling mechanisms as any other thread. The implication of this is that **you must check the condition with a while loop instead of an if statement because it is possible for the condition to change to false between the time the thread was unblocked and the time it takes over the CPU**.

2 Problems

2.1 The Central Galactic Floopy Corporation

It's the year 3162. Floopies are the widely recognized galactic currency. Floopies are represented in digital form only, at the Central Galactic Floopy Corporation (CGFC).

You receive some inside intel from the CGFC that they have a Galaxynet server running on some old OS called x86 Ubuntu 14.04 LTS. Anyone can send requests to it. Upon receiving a request, the server forks a POSIX thread to handle the request. In particular, you are told that sending a transfer request will create a thread that will run the following function immediately, for speedy service.

```
void transfer(account_t *donor, account_t *recipient, float amount) {
  assert (donor != recipient); // Thanks CS161

  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

Assume that there is some struct with a member `balance` that is `typedef`-ed as `account_t`. Describe how a malicious user might exploit some unintended behavior.

There are multiple race conditions here.

Suppose Alice and Bob have 5 floopies each. We send two quick requests: transfer(&alice, &bob, 5) and transfer(&bob, &alice, 5). The first call decrements Alices balance to 0, adds 5 to Bobs balance, but before storing 10 in Bobs balance, the next call comes in and executes to completion, decrementing Bobs balance to 0 and making Alices balance 5. Finally we return to the first call, which just has to store 10 into Bobs balance. In the end, Alice has 5, but Bob now has 10. We have effectively duplicated 5 floopies.

Graphically:

Thread 1	**Thread 2**
temp1 = Alice's balance (== 5)	
temp1 = temp1 - 5 (== 0)	
Alice's balance = temp1 (== 0)	
temp1 = Bob's balance (== 5)	
temp1 = temp1 + 5 (== 10)	
INTERRUPTED BY THREAD 2	
	temp2 = Bob's balance (== 5)
	temp2 = temp2 - 5 (== 0)
	Bob's balance = temp2 (== 0)
	temp2 = Alice's balance (== 0)
	temp2 = temp2 + 5 (== 5)
	Alice's balance = temp2 (== 5)
	THREAD 2 COMPLETE
RESUME THREAD 1	
Bob's balance = temp1 (== 10)	
THREAD 1 COMPLETE	

It is also possible to achieve a negative balance. Suppose at the beginning of the function, the donor has enough money to participate in the transfer, so we pass the conditional check for sufficient funds. Immediately after that, the donors balance is reduced below the required amount by some other running thread. Then the transfer will go through, resulting in a negative balance for the donor.

Sending two identical transfer(&alice, &bob, 2) may also cause unintended behavior, since the increment/decrement operations are not atomic (though it is arguably harder to exploit for profit).

Since you're a good person who wouldn't steal floopies from a galactic corporation, what changes would you suggest to the CGFC to defend against this exploit?

The entire function must be made atomic. One could do this by disabling interrupts for that period of time (if there is a single processor), or by acquiring a lock beforehand and releasing the lock afterwards. Alternatively, you could have a lock for each account. In order to prevent deadlocks, you will have to acquire locks in some predetermined order, such as lowerst account number first.

2.2 test_and_set

In the following code, we use test_and_set to emulate locks.

```
int value = 0;
int hello = 0;
```

```
void print_hello() {
    while (test_and_set(&value));
    hello += 1;
    printf("Child thread: %d\n", hello);
    value = 0;
    pthread_exit(0);
}

void main() {
    pthread_t thread1;
    pthread_t thread2;
    pthread_create(&thread1, NULL, (void *) &print_hello, NULL);
    pthread_create(&thread2, NULL, (void *) &print_hello, NULL);
    while (test_and_set(&value));
    printf("Parent thread: %d\n", hello);
    value = 0;
}
```

Assume the following sequence of events:
1. Main starts running and creates both threads and is then context switched right after
2. Thread2 is scheduled and run until after it increments hello and is context switched
3. Thread1 runs until it is context switched
4. The thread running main resumes and runs until it get context switched
5. Thread2 runs to completion
6. The thread running main runs to completion (but doesn't exit yet)
7. Thread1 runs to completion

Is this sequence of events possible? Why or why not?

Yes. In steps 3 and 4, the main thread and thread1 make no progress. They can only run to completion after thread2 resets the value to 0.

At each step where `test_and_set(&value)` is called, what value(s) does it return?

1. No call to test_and_set
2. 0
3. 1, 1, ..., 1
4. 1, 1, ..., 1
5. No call to test_and_set
6. 0
7. 0

Given this sequence of events, what will C print?

```
Child thread: 1
Parent thread: 1
Child thread: 2
```

Is this implementation better than using locks? Explain your rationale.

No, this involves a ton of busy waiting.

2.3 Hello World

This code compiles (given a sprinkling of `#include`s etc.) but doesn't work properly. Why?

```
pthread_mutex_t lock;
pthread_cond_t cv;
int hello = 0;

void* print_hello(void* arg) {
    hello += 1;
    printf("First line (hello=%d)\n", hello);
    pthread_cond_signal(&cv);
    pthread_exit(0);
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, print_hello, NULL);
    while (hello < 1) {
      pthread_cond_wait(&cv, &lock);
    }
    printf("Second line (hello=%d)\n", hello);
    return 0;
}
```

This won't work because the main thread should have locked the lock before calling pthread_cond_wait, and the child thread should have locked the lock before calling pthread_cond_signal. (Also, we never initialized the lock and cv.)

Add in the necessary code to the above problem to make it work correctly.

Acquire a lock before the cv is used and release it afterwards.

```
void* print_hello(void* arg) {
    pthread_mutex_lock(&lock);
    hello += 1;
    printf("First line (hello=%d)\n", hello);
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&lock);
    pthread_exit(0);
}

int main() {
    pthread_t thread;
    pthread_mutex_init(&lock, 0);
    pthread_cond_init(&cv, 0);

    pthread_create(&thread, NULL, print_hello, NULL);

    pthread_mutex_lock(&lock);
    while (hello < 1) {
        pthread_cond_wait(&cv, &lock);
```

```
    }
    pthread_mutex_unlock(&lock);

    printf("Second line (hello=%d)\n", hello);
    return 0;
}
```

2.4 SpaceX Problems

Consider this program.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

int n = 3;

void* counter(void* arg) {
  pthread_mutex_lock(&lock);
  for (n = 3; n > 0; n--)
    printf("%d\n", n);
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&lock);
}

void* announcer(void* arg) {
  while (n != 0) {
    pthread_mutex_lock(&lock);
    pthread_cond_wait(&cv, &lock);
    pthread_mutex_unlock(&lock);
  }
  printf("FALCON HEAVY TOUCH DOWN!\n");
}

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, counter, NULL);
  pthread_create(&t2, NULL, announcer, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return 0;
}
```

What is wrong with this code?

The lock in announcer() should be outside of the while loop. Or else, the announcer thread might never wake up.

2.5 All Threads Must Die

You have three Pintos threads with the associated priorities shown below. They each run the functions with their respective names.

Tyrion : 4
Ned: 5
Gandalf: 11

Assume upon execution that all threads are unblocked and begin at the top of their code blocks. The operating system runs with a preemptive priority scheduler. You may assume that set_priority commands are atomic. (Note: The following uses references to Pintos locks and data structures.)

```
struct list brace_yourself;   // pintos list. Assume it's already initialized and populated.
struct lock midterm;          // pintos lock. Already initialized.
struct lock is_coming;

void tyrion(){
    thread_set_priority(12);
    lock_acquire(&midterm);
    lock_release(&midterm);
    thread_exit();
}

void ned(){
    lock_acquire(&midterm);
    lock_acquire(&is_coming);
    list_remove(list_head(brace_yourself));
    lock_release(&midterm);
    lock_release(&is_coming);
    thread_exit();
}

void gandalf(){
    lock_acquire(&is_coming);
    thread_set_priority(3);
    while (thread_get_priority() < 11) {
        printf("YOU .. SHALL NOT .. PAAASS!!!!!!);
        timer_sleep(20);
    }
    lock_release(&is_coming);
    thread_exit();
}
```

What is the output of this program when there is no priority donation? Trace the program execution and number the lines in the order in which they are executed.

```
void tyrion(){
5    thread_set_priority(12);
6    lock_acquire(&midterm); //blocks
    lock_release(&midterm);
    thread_exit();
```

```
}

void ned(){
3    lock_acquire(&midterm);
4    lock_acquire(&is_coming);  //blocks
    list_remove(list_head(brace_yourself));
    lock_release(&midterm);
    lock_release(&is_coming);
    thread_exit();
}

void gandalf(){
1    lock_acquire(&is_coming);
2    thread_set_priority(3);
7    while (thread_get_priority() < 11) {
8        printf("YOU .. SHALL NOT .. PAAASS!!!!!!); //repeat till infinity
9        timer_sleep(20);
    }
    lock_release(&is_coming);
    thread_exit();
}

Gandalf, as you might expect, endlessly prints "YOU SHALL NOT PASS!!" every 20 clock ticks or so.\\
```

What is the output and order of line execution if priority donation was implemented? Draw a diagram of the three threads and two locks that shows how you would use data structures and struct members (variables and pointers, etc) to implement priority donation for this example.

```
void tyrion(){
8    thread_set_priority(12);
9    lock_acquire(&midterm); //blocks
    lock_release(&midterm);
    thread_exit();

}

void ned(){
3    lock_acquire(&midterm);
4    lock_acquire(&is_coming);  //blocks
11   list_remove(list_head(brace_yourself));  //KERNEL PANIC
    lock_release(&midterm);
    lock_release(&is_coming);
    thread_exit();
}

void gandalf(){
1    lock_acquire(&is_coming);
2    thread_set_priority(3);
5    while (thread_get_priority() < 11) {   //priority is 5 first, but 12 at some later loop
6        printf("YOU .. SHALL NOT .. PAAASS!!!!!!);
7        timer_sleep(20);
    }
10    lock_release(&is_coming);
```

```
    thread_exit();
}

It turns out that Gandalf generally does mean well. Donations will make
Gandalf allow you to pass.
At some point Gandalf will sleep on a timer and leave Tyrion alone in the
ready queue.
Tyrion will run even though he has a lower priority (Gandalf has a 5
donated to him)
Tyrion then sets his priority to 12 and chain-donates to Gandalf. Gandalf
breaks his loop.
Ned unblocks after Gandalf releases the is_coming lock.
However, allowing Ned to remove the head of a list will trigger an ASSERT
failure in lib/kernel/list.c.

Gandalf will print YOU SHALL NOT PASS at least once.
Then Ned will get beheaded and cause a kernel panic that crashes Pintos.
```