Section 6: Deadlock and Midterm Review

October 9-11

Contents

1 Vocabulary

- **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.
- **Deadlock** - Situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.
- **Banker's Algorithm** - A resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, before deciding whether allocation should be allowed to continue.

2 Scheduling

2.1 All Threads Must Die

You have three threads with the associated priorities shown below. They each run the functions with their respective names. Assume upon execution all threads are initially 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.

Tyrion : 4
Ned: 5
Gandalf: 11

Note: The following uses references to Pintos locks and data structures.

```
struct list braceYourself;    // pintos list. Assume it's already initialized and populated.
struct lock midTerm;          // pintos lock. Already initialized.
struct lock isComing;
void tyrion(){
    thread_set_priority(12);
    lock_acquire(&midTerm);
    lock_release(&midTerm);
    thread_exit();

}
void ned(){
    lock_acquire(&midTerm);
    lock_acquire(&isComing);
```

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

1. 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(&isComing);  //blocks
    list_remove(list_head(braceYourself));
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

void gandalf(){
1    lock_acquire(&isComing);
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(&isComing);
    thread_exit();
}

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

2. 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(&isComing);  //blocks
12   list_remove(list_head(braceYourself));  //KERNEL PANIC
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

void gandalf(){
1    lock_acquire(&isComing);
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(&isComing);
11    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 exits.
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.
```

3 Practice Exam

3.1 File I/O [Question 6]

1. Implement the function `read162` that reads the specified number of bytes from a file, and only returns short if the end of file is reached or an error occurs.

```
int read162(int fd, char *buf, size_t nbyte) {
    int bytes_read = 0, total_read = 0;
    while (( bytes_read = read(fd, &buf[total_read], nbyte - total_read)) > 0) {
        total_read += bytes_read;
    }
    return total_read;
}
```

2. Suppose you have a `cs162_file_t` struct, similar to `FILE`, that looks like this:

```
typedef struct {
    int fd;
    char read_buf[BUF_SIZE];
    int buf_pos;
    int buf_max;
    pthread_mutex_t lock;
} cs162_file_t;
```

Implement a thread-safe function `char fgetc(cs162_file_t *f)` that returns the next character in the file or EOF if end of file is reached. You may call `read162` from above.

```
char fgetc(cs162_file_t *f) {
    char ret = EOF;
    pthread_mutex_lock(&f->lock);
    if (f->buf_pos >= f->buf_max) {
        bytes_read = read162(f->fd, f->read_buf, BUF_SIZE);
        f->buf_pos = 0;
        f->buf_max = bytes_read;
    }
    if (f->buf_max != 0) {
        ret = f->read_buf[f->buf_pos];
        f->buf_pos += 1;
    }
    pthread_mutex_unlock(&f->lock);
    return ret;
}
```

3.2 Scheduling [Question 7]

Assume each numbered line of code takes 1 CPU cycle to run, and that a context switch takes 2 CPU cycles. Hardware preemption occurs every 50 CPU cycles and takes 1 CPU cycle. The scheduler is run after every hardware preemption and takes 0 time. Finally, the currently running thread does not change until the end of a context switch.

```
Lock lock_a, lock_b; // Assume these locks are already initialized and unlocked.
int a = 0;
int b = 1;
bool run = true;

    Kiki() {
1.      bool cond = run;
2.      while (cond) {
3.          int x = a;
4.          int y = b;
5.          int sum = x + y;
6.          lock_a.acquire();
7.          lock_b.acquire();
8.          a = y;
9.          b = sum;
10.         lock_a.release();
11.         lock_b.release();
12.         cond = run;
        }
    }

    Jiji() {
1.      bool cond = run;
2.      while (cond) {
3.          int x = b;
4.          int sum = a + b;
5.          lock_b.acquire();
6.          lock_a.acquire();
7.          b = sum;
8.          a = x;
9.          lock_b.release();
10.         lock_a.release();
11.         cond = run;
        }
    }

    Tombo() {
1.      while (true) {
2.          lock_a.acquire()
3.          a = 0;
4.          lock_a.release()
5.          lock_b.acquire()
6.          b = 1;
7.          lock_b.release()
    }   }
```

Thread 1 runs `Kiki`, Thread 2 runs `Jiji`, and Thread 3 runs `Tombo`. Assuming round robin scheduling (threads are initially scheduled in numerical order):

1. What are the values of `a` and `b` after 50 CPU cycles?

   The entire first 50 cycles is spent running `Kiki`.

   ```
   @12 : a = 1, b = 1
   @23 : a = 1, b = 2
   @34 : a = 2, b = 3
   @45 : a = 3, b = 5
   ```

2. What are the values of `a` and `b` after 200 CPU cycles? What line is the program counter on for each thread?

   The entire first 50 cycles is spent running `Kiki`. `a = 3, b = 5`

   There was deadlock! After part (a), the last instruction Thread 1 ran (cycle 50) is line 6, `lock_a.acquire()`. Thread 2 can run until it has acquired lock b, but it will block on `lock_a.acquire()`, since Thread 1 currently has it. Thread 3 can run until `lock_a.acquire()` as well. When it gets switched back to Thread 1, it cannot `lock_b.acquire()` because Thread 2 has it.

Now assume we use the same round robin scheduler but we just run 2 instances of `Kiki`.

3. What are the values of `a` and `b` after 100 cycles?

   ```
   @12 : a = 1, b = 1                  @58 : T2 blocked on line 6
   @23 : a = 1, b = 2                  @60 : T1 context switched on
   @34 : a = 2, b = 3                  @66 : a = 5, b = 8
   @45 : a = 3, b = 5                  @78 : a = 8, b = 13
   @50 : T1 pre-empted after line 6    @90 : a = 13, b = 21
   @52 : T2 context switched on.       @100 : a = 21, b = 34
   ```

Now we replace our scheduler with a CFS-like scheduler. In particular, this scheduler is invoked on every call to `lock_acquire` or `lock_release`. We still have 2 threads running `Kiki`, but this time thread 1 runs for 50 cycles before thread 2 starts.

4. What are the values of `a` and `b` at the end of the 73rd cycle?

   ```
   @50 thread1 has acquired lock A
   @53 thread 2 begins to run
   After 58, context switch back to thread 1
   After 64, thread 1 releases lock_a and thread 2 runs
   After 67, thread 2 acquires lock_a
   @68 thread 2 attempts to acquire lock_b
   @71 thread 1 releases lock_b
   After 73, lock_b is acquired by thread 2
   a = 13, b = 21
   ```

4 Deadlock

4.1 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;
}
```

1. 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 Alice's balance to 0, adds 5 to Bob's balance, but before storing 10 in Bob's balance, the next call comes in and executes to completion, decrementing Bob's balance to 0 and making Alice's balance 5. Finally we return to the first call, which just has to store 10 into Bob's 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 donor's 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).

2. 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.

4.2 Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows:

Total		
A	B	C
7	8	9

	Current			Max		
T/R	A	B	C	A	B	C
T1	0	2	2	4	3	3
T2	2	2	1	3	6	9
T3	3	0	4	3	1	5
T4	1	3	1	3	3	4

1. Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

Yes, the system is in a safe state.
To find a safe sequence of executions, we need to first calculate the available resources and the needed resources for each thread. To find the available resources, we sum up the currently held resources from each thread and subtract that from the total resources:

Available		
A	B	C
1	1	1

To find the needed resources for each thread, we subtract the resources they currently have from the maximum they need:

Needed			
	A	B	C
T1	4	1	1
T2	1	4	8
T3	0	1	1
T4	2	0	3

From these, we see that we must run T3 first, as that is the only thread for which all needed resources are currently available. After T3 runs, it returns its held resources to the resource pool, so the available resource pool is now as follows:

Available		
A	B	C
4	1	5

We can now run either T1 or T4, and following the same process, we can arrive at a possible execution sequence of either $T3 \rightarrow T1 \rightarrow T4 \rightarrow T2$ or $T3 \rightarrow T4 \rightarrow T1 \rightarrow T2$.

2. Repeat the previous question if the total number of C instances is 8 instead of 9.

Following the same procedure from the previous question, we see that there are 0 instances of C available at the start of this execution. However, every thread needs at least 1 instance of C to run, so we are unable to run any threads and thus the system is not in a safe state.