Section 4: Scheduling and Synchronization

Isaac Tian

February 16, 2015

Contents

1 Warmup

1.1 Who's Your Daddy

What does C print in the following code? Assume the new thread's ID is 0x4c554b45. Assume Round Robin Scheduling and no other threads/processes are running.

```
pthread_t thread_id = 0x0;

void print_hello_world() {
    thread_id = pthread_self();  //like getpid but for threads
    printf("You deallocated my father!\n");
    pthread_exit(0);
}

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, (void *) &print_hello_world, NULL);
    pthread_yield();
    printf("No, thread %#010x, I AM your father!!"\n, thread_id);
}
```

```
No, thread 0x4c554b45, I AM your father!!
You deallocated my father!

OR

You deallocated my father!
No, thread 0x4c554b45, I AM your father!!

OR

No, thread 0x0, I AM your father!!
You deallocated my father!

OR

garbage / other combinations possible if store on thread_id is interrupted at
assembly language level (x86).

Round robin wait queues are FIFO so you'll at least force the new thread
to start the global variable store before the parent prints. But without
synchronization or any notion of the time quantum the store is not guaranteed
to finish. It could be forced to yield the CPU at any time again, which may
create a race condition on thread_id.

Note that the following is INCORRECT:
You deallocated my father!
No, thread 0x0, I AM your father!
(impossible for parent to print 0x0 after child reached print statement
because execution in the same thread is still serial)
```

2 Vocabulary

- **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.
- **Condition Variable** - Synchronization variables that provide serialization. A condition variable (CV) is defined by an associated lock, a condition, and a wait queue. A thread must hold the lock in order to access any CV functions. If the condition is false, the thread blocks. The wait queue keeps track of all waiting threads and signals one or all threads to unblock should the condition become true. Note that this is not generally equivalent to passing control directly to the waking thread (See Mesa semantics).
- **pthread_yield** - Equivalent to thread_yield() in Pintos. Causes the calling thread to vacate the CPU and go back into the ready queue without blocking. The calling thread is able to be scheduled again immediately. This is not the same as an interrupt and will succeed in Pintos even if interrupts are disabled.
- **Race Condition** - A state of execution that causes multiple threads to access the same shared variable (heap or global data segment) with at least one thread attempting to execute a write without enforcing mutual exclusion. The result is not necessarily garbage but is treated as being undefined since there is no guarantee as to what will actually happen. Note that multiple reads do not need to be mutexed.
- **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.
- **Hoare Semantics** - In a 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 a condition variable, wake a blocked thread when the condition is true with no guarantee on when that thread will actually execute. (The newly woken thread simply gets put on the ready queue and is subject to the same scheduling semantics as any other thread.) The implications of this mean 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.

3 Problems

3.1 These Are The Locks You're Looking For

What does C print in the following code? You may not assume anything about the scheduler other than that it behaves with Mesa semantics. (i.e. could be RR, FIFO, priority) In general, user programs should not depend on the scheduler and should run correctly regardless of the scheduler used.

```
int ben = 0;

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (ben==1) printf("These are not the droids you are looking for. ben = %d\n", ben);
    else printf("These are the droids you are looking for! ben = %d\n", ben);
    exit(0);
}

void *helper(void* arg) {
    ben+=1;
    pthread_exit(0);
}
```

The output of this program is undefined because there is a race condition on the global variable ben. The value of ben could change while it is being read by main, or it could change after the value is checked but before the print is executed. Yields have no control over scheduling.

Declare a lock and use it to guarantee the print message of this program. Pseudocode is fine.

```
int ben = 0;
//LOCK L

void main() {
    pthread_t thread;
    //ACQUIRE L
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (ben==1) printf("These are not the droids you are looking for.\n");
    else printf("These are the droids you are looking for!\n");
    //RELEASE L
    exit(0);
}

void *helper(void* arg) {
    //ACQUIRE L
    ben+=1;
    //RELEASE L
    pthread_exit(0);
}
```

First you must declare the lock as a global variable (so it goes in the data segment and can be accessed by all threads.) Then with appropriate locking, the best you can do in this case is making it always print "These are the droids you are looking for!"

Locks do not guarantee ordering, only mutual exclusion. If we acquire the lock after the new thread is created, we guarantee that there is no race condition on the variable 'ben' but not its actual value, as the scheduler could preempt the main thread and run the new thread even before a yield is called.

Suppose we did the following instead to attempt to force serialization. What will this program print? Does it add any extra synchronization protection?

```
int ben = 0;

void main() {
    //INTR_DISABLE()
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (ben==1) printf("These are not the droids you are looking for.\n");
    else printf("These are the droids you are looking for!\n");
    //INTR_ENABLE()
    exit(0);
}

void *helper(void* arg) {
    ben+=1;
    pthread_exit(0);
}
```

We still don't know for sure because disabling interrupts does not prevent you from forcing your current thread to yield. (Note that yielding isn't an interrupt.)

If the scheduler schedules the new thread after the yield,
it will always print "These are not the droids you are looking for."
If the scheduler schedules the main thread after the yield,
it will always print "These are the droids you are looking for!"

Because interrupts are disabled, the race condition on ben is gone but like with locks the order of execution is still undefined.

3.2 Only A Sith Deals In Absolute Conditions

Consider the same block of code. How do you ensure that you always print out the canonically correct line? Assume the scheduler behaves with Mesa semantics. (Pseudocode is OK) You may only add lines, so the trivial answer of not checking the value of ben before printing is not correct.

```
int ben = 0;

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (ben==1) printf("These are not the droids you are looking for.\n");
    else printf("These are the droids you are looking for!\n");
    exit(0);
}

void *helper(void* arg) {
    ben+=1;
    pthread_exit(0);
}
```

```
int ben = 0;
//LOCK = L
//CONDVAR = C

void main() {
    pthread_t thread;
    //LOCK L ACQUIRE
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    //WHILE BEN != 1
        //CONDVAR C WAIT
    if (ben==1) printf("These are not the droids you are looking for.\n");
    //SHOULD ALWAYS BE TRUE
    else printf("These are the droids you are looking for!);
    //LOCK L RELEASE
    exit(0);
}

void *helper(void* arg) {
    //LOCK L ACQUIRE
    ben+=1;
    //CONDVAR C SIGNAL
    //LOCK L RELEASE
    pthread_exit(0);
}

(Did not bother to use syntax for POSIX locks and condvars since it isn't in PintOS)
```

3.3 Life Ain't Fair

Suppose the following threads denoted by THREADNAME : PRIORITY pairs arrive in the ready queue at the clock ticks shown. Assume all threads arrive unblocked and that each takes 5 clock ticks to finish executing. Assume threads arrive in the queue at the beginning of the time slices shown and are ready to be scheduled in that same clock tick. (This means you update the ready queue with the arrival before you schedule/execute that clock tick.) Assume you only have one physical CPU.

```
0   Vaishaal : 7
1
2   Caleb : 1
3   Stanley: 3
4
5   Arka : 5
6
7   Will: 11
8
9   George: 14
```

Determine the order and time allocations of execution for the following scheduler scenarios:

- Round Robin with time slice 3
- Shortest Time Remaining First (SRTF/SJF) WITH preemptions
- Preemptive priority (higher is more important)

Write answers in the form of vertical columns with one name per row, each denoting one clock tick of execution. For example, allowing Vaishaal 3 units at first looks like:

```
0   Vaishaal
1   Vaishaal
2   Vaishaal
```

It will probably help you to draw a diagram of the ready queue at each tick for this problem.

We're assuming that threads that arrive always get scheduled earlier than threads that have already been running or have just finished.

Explanation for RR:

From t=0 to t=3, Vaishaal gets to run since there is initially no one else on the run queue. At t=3, Vaishaal gets preempted since the time slice is 3. Caleb is selected as the next person to run, and Stanley gets added to the run queue (t=2.9999999) just before Vaishaal (t=3).

Caleb is the next person to run from t=3 to t=6. At t=5, Arka gets added to the run queue, which consists of at this point: Stanley, Vaishaal, Arka

At t=6, Caleb gets preempted and Stanley gets to run since he is next. Caleb gets added to the back of the queue, which consists of: Vaishaal, Arka, Caleb.

From t=6 to t=9, Stanley gets to run and then is preempted. Vaishaal gets to run again from t=9 to t=10, and then finishes executing. Arka gets to run next and this pattern continues until everyone has completed running.

```
RR:
0   Vaishaal
1   Vaishaal
2   Vaishaal
3   Caleb
4   Caleb
```

```
5   Caleb
6   Stanley
7   Stanley
8   Stanley
9   Vaishaal
10  Vaishaal
11  Arka
12  Arka
13  Arka
14  Caleb
15  Caleb
16  Will
17  Will
18  Will
19  George
20  George
21  George
22  Stanley
23  Stanley
24  Arka
25  Arka
26  Will
27  Will
28  George
29  George

Preemptive SRTF
0   Vaishaal
1   Vaishaal
2   Vaishaal
3   Vaishaal
4   Vaishaal
5   Caleb
6   Caleb
7   Caleb
8   Caleb
9   Caleb
...
(Pretty much just like FIFO since every thread takes 5 ticks)

Preemptive Priority
0   Vaishaal
1   Vaishaal
2   Vaishaal
3   Vaishaal
4   Vaishaal
5   Arka
6   Arka
7   Will
8   Will
9   George
10  George
11  George
12  George
```

```
13  George
14  Will
15  Will
16  Will
17  Arka
18  Arka
19  Arka
20  Stanley
21  Stanley
22  Stanley
23  Stanley
24  Stanley
25 - 29 Caleb
```

3.4 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();
}
```

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.
```

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.
```