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);
}
```

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);
}
```

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

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);
}
```

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);
}
```

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.

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.

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.