

Section 5: Scheduling

CS 162

October 2, 2020

Contents

1	Vocabulary	2
2	Scheduling	3
2.1	Round Robin Scheduling	3
2.2	Life Ain't Fair	4
2.3	All Threads Must Die	6
3	Fall 2019 Practice Midterm	8
3.1	Optional Practice: Delivery Service	8

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.
- **Preemption** - The process of interrupting a running thread to allow for the scheduler to decide which thread runs next.
- **FIFO Scheduling** - First-In-First-Out (aka First-Come-First-Serve) scheduling runs jobs as they arrive. Turnaround time can degrade if short jobs get stuck behind long ones (convoy effect);
- **round-robin Scheduling** - Round-Robin scheduling runs each job in fixed-length time slices (quanta). The scheduler preempts a job that exceeds its quantum and moves on, cycling through the jobs. It avoids starvation and is good for short jobs, but context switching overhead can become important depending on quanta length;
- **Shortest Time Remaining First Scheduling** - A scheduling algorithm where the thread that runs is the one with the least time remaining. This is ideal for throughput but also must be approximated in practice.
- **Linux CFS** - Linux scheduling algorithm designed to optimize for fairness. It gives each thread a weighted share of some target latency and then ensures that each thread receives that much virtual CPU time in its scheduling decisions.
- **Earliest Deadline First** - Scheduling algorithm used in real time systems. It attempts to meet deadlines of threads that must be processed in real time by selecting the thread that has the closest deadline to complete first.
- **Multi-Level Feedback Queue Scheduling** - MLFQS uses multiple queues with priorities, dropping CPU-bound jobs that consume their entire quanta into lower-priority queues.
- **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.

2 Scheduling

2.1 Round Robin Scheduling

Which of the following are true about Round Robin Scheduling?

1. The average wait time is less than that of FCFS for the same workload.
2. It requires pre-emption to maintain uniform quanta.
3. If quanta is constantly updated to become the # of cpu ticks since boot, Round Robin becomes FIFO.
4. If all threads in the system have the same priority, Priority Schedulers **must** behave like round robin.
5. Cache performance is likely to improve relative to FCFS.
6. If no new threads are entering the system all threads will get a chance to run in the cpu every $QUANTA * SECONDS_PER_TICK * NUMTHREADS$ seconds. (Assuming QUANTA is in ticks).
7. It is the fairest scheduler

2.2 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  Taj : 7
1
2  Kevin : 1
3  Neil: 3
4
5  Akshat : 5
6
7  William: 11
8
9  Alina: 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 Taj 3 units at first looks like:

```

0  Taj
1  Taj
2  Taj

```

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



2.3 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();
}

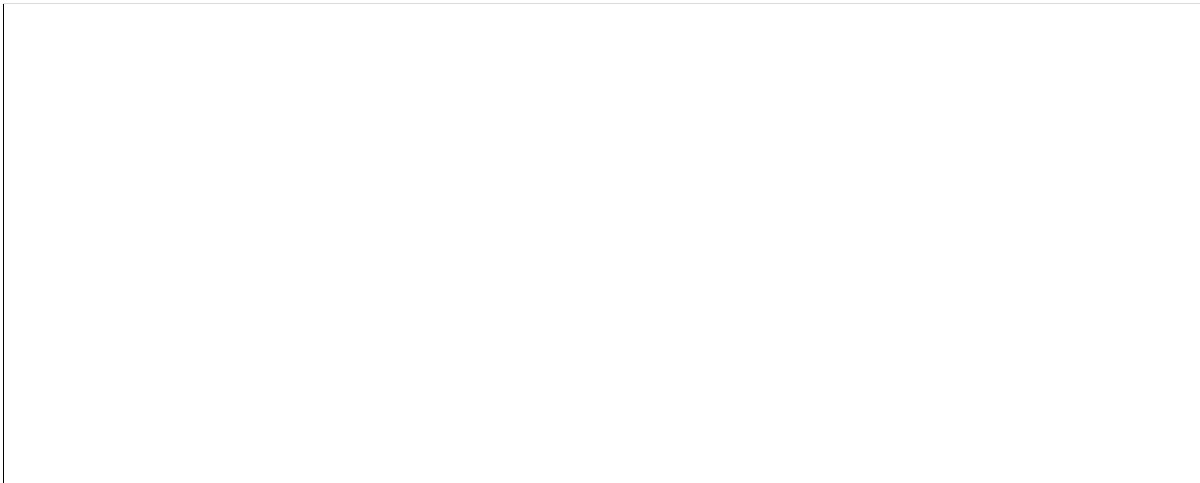
void ned(){
    lock_acquire(&midTerm);
    lock_acquire(&isComing);
    list_remove(list_head(braceYourself));
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

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.



3 Fall 2019 Practice Midterm

3.1 Optional Practice: Delivery Service

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?

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

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?

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?