Section 5: Scheduling

CS 162

October 1, 2021

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 . . . 7

3 MLFQS **10**
3.1 Bitcoin Mining . . . 10

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

2,3
1. False. Counterexample by setting a very small quanta. 2. True. 3. True. 4. False. Not a requirement. 5. False. More context switches means worse cache performance. 6. Trick question. There is some overhead. 7. Trick question. Needs definition of fair.

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.

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

Kevin is the next person to run from t=3 to t=6. At t=5, Akshat gets added to the run queue, which consists of at this point: Neil, Taj, Akshat

At t=6, Kevin gets preempted and Neil gets to run since he is next. Kevin gets added to the back of the queue, which consists of: Taj, Akshat, Kevin.

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

```
RR:
0   Taj
1   Taj
2   Taj
3   Kevin
4   Kevin
```

```
5   Kevin
6   Neil
7   Neil
8   Neil
9   Taj
10  Taj
11  Akshat
12  Akshat
13  Akshat
14  Kevin
15  Kevin
16  William
17  William
18  William
19  Alina
20  Alina
21  Alina
22  Neil
23  Neil
24  Akshat
25  Akshat
26  William
27  William
28  Alina
29  Alina

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

Preemptive Priority
0   Taj
1   Taj
2   Taj
3   Taj
4   Taj
5   Akshat
6   Akshat
7   William
8   William
9   Alina
10  Alina
11  Alina
12  Alina
13  Alina
```

```
14  William
15  William
16  William
17  Akshat
18  Akshat
19  Akshat
20  Neil
21  Neil
22  Neil
23  Neil
24  Neil
25 - 29 Kevin
```

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.

```
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
11   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);
      thread_exit();
}
```

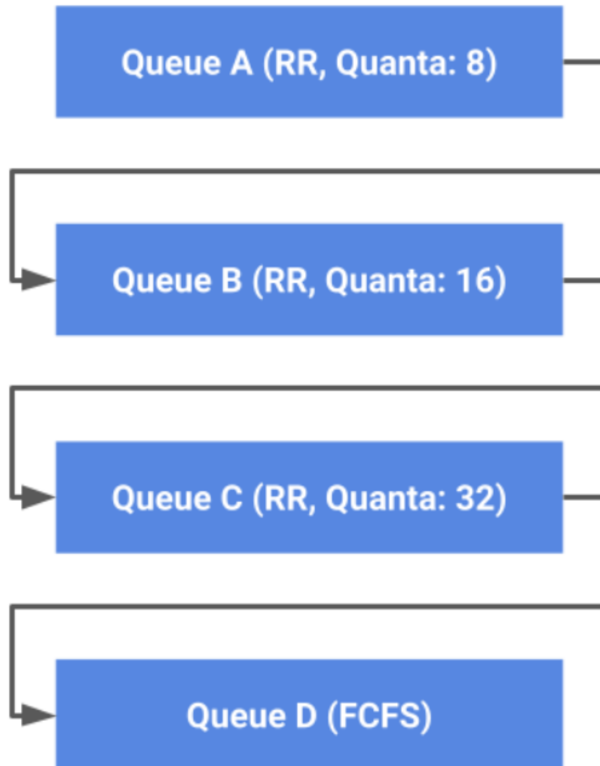
```
It turns out that Gandalf generally does mean well. Donations will make
Gandalf allow you to pass.
At some point Gandalf will sleep on a timer and leave Tyrion alone in the
ready queue.
Tyrion will run even though he has a lower priority (Gandalf has a 5
donated to him)
Tyrion then sets his priority to 12 and chain-donates to Gandalf. Gandalf
breaks his loop.
Gandalf releases the lock and his priority reverts back to 3.
Ned, now the highest priority thread, is preempted and starts to run.
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 MLFQS

3.1 Bitcoin Mining

You are a Bitcoin miner, and you've developed an algorithm that can run on an unsuspecting machine and mine Bitcoin. You now need to write a program that will run your mining algorithm forever. While you want your mining job to be scheduled often, you also don't want to attract too much suspicion from system users or administrators. Fortunately, you know that the machines you're targeting use a MLFQS algorithm to schedule jobs, outlined below:



You decide that the best strategy is to guarantee that your mining job will *always* be placed on Queues B and C.

Assume that the CPU-intensive mining algorithm you've developed can be run in 10 tick intervals. Implement your mining program, and explain your design. The only functions you should use are 'mine()' (which runs for 10 ticks) and 'printf()'. Assume that your job is initially placed on Queue B.

```
void mine_forever() {
    while(1) {
        for (int i = 0; i < 4; i++)
            mine();
        printf(\Not a Bitcoin miner!!!");
    }
}
```

The program needs to exceed the Queue B quanta so that the quanta expires and the job is placed on Queue C. Once placed on Queue C, the job needs to voluntarily yield so that it can be placed on Queue B once again. By running the mining algorithm 4 times, the first 2 iterations will cause the Queue B quanta to expire. After the remaining 2 iterations, the job should voluntarily yield (e.g. print to stdout). Note: Once the job is placed on Queue C, it will still run the 2nd iteration of the mining algorithm for 4 ticks.

Explain why, regardless of how you implement your mining program, your job will never be placed on Queue A twice in a row.

Since the mining algorithm can only be run in 10 tick intervals, any implementation will always exceed the Queue A quanta before the CPU can be voluntarily yielded. This will cause the job to be placed on Queue B, since the Queue A quanta expired.