Discussion 4: Scheduling

October 04, 2023

Contents

1 Scheduling **2**
1.1 Round Robin T/F 3
1.2 Life Ain't Fair 4
1.3 Bitcoin Mining 5

1 Scheduling

Scheduling is the process of deciding which threads are given access to resources from moment to moment. Usually, scheduling pertains to CPU access times, but it can encompass any type of resource like network bandwidth or disk access.

When discussing scheduling, we typically make some assumptions for simplification. Each user is assumed to have one single-threaded program where each program is independent of each other.

Goals and Criteria

The goal of scheduling is to dole out CPU time to optimize some desired parameters of the system. Generally speaking, scheduling policies focus on the following goals and criteria.

Minimize Completion Time

Completion time is the combination of the waiting time plus the run time of a process (e.g. time to compile a program, time to echo a keystroke in an editor). Minimizing completion time is crucial for time-sensitive tasks (e.g. I/O).

Maximize Throughput

Throughput is the rate at which tasks are completed. While throughput is related to completion time, they are not the same. Maximizing throughput involves minimizing overhead (e.g. context switching) and efficient use of resources.

Maintain Fairness

Fairness refers to sharing resources in some equitable manner. Evidently, fairness is not very well defined unless specific parameters are specified. Maintaining fairness will usually contradict minimizing completion time.

First Come First Serve (FCFS)

First come first serve (FCFS) schedules tasks in the order in which they arrive. It's simple to implement and is good for throughput since it minimizes the overhead of context switching. However, average completion time under FCFS can very significantly according to arrival order. Additionally, FCFS suffers from the **Convoy effect** where short tasks get stuck behind long tasks.

Shortest Job First (SJF) / Shortest Remaining Time First (SRTF)

Shortest job first (SJF) schedules the shortest task first. **Shortest remaining time first (SRTF)** is a preemptive version of SJF. If a task arrives and has a shorter time to completion than the current running task, the resource will be preempted. SJF and SRTF are provably optimal for minimizing average completion time among non-preemptive and preemptive schedulers, respectively. However, both of them involve the impossible idea of knowing how long a task is going to take.

Moreover, they do not maintain fairness with regards to the amount of time spent using a resource. In fact, SJF and SRTF can lead to **starvation** which is the continued lack of progress for one task due to other tasks being scheduled over it. If small tasks keep arriving, larger tasks will never get to run.

Round Robin (RR)

Round robin (RR) schedules tasks such that each of them takes turns using a resource for a small unit of time known as the **time quantum** (q). After q expires, the task is preempted and added to the end of the ready queue. When q is large, RR resembles FCFS, becoming identical if q is larger than the length of any task. When q is small, RR resembles SJF Generally, q must be large with respect to the cost of context switching to avoid the overhead being too high.

If there are n tasks, each task gets $1/n$ amount of resources, ensuring fairness in terms of sharing resources. No task will wait for more than $(n-1)q$ time units.

In general, with RR, small scheduling quantum decreases response time but increases completion time, due to the extra context switching overhead and the fact that longer jobs get "stretched out".

Lottery

Lottery gives each task a certain number of lottery tickets. On each time slice, a ticket is randomly picked. On expectation, each task will be given time proportional to the number tickets it was originally given. When distributing tickets, it's important to make sure that every task gets at least one ticket to avoid starvation. If SRTF was to be approximated, shorter tasks would simply get more tickets than longer tasks.

Multi-Level Feedback Queue (MLFQ)

Multi-level feedback queue (MLFQ) uses multiple queues which each have different priority. Each queue has its own scheduling algorithm. Higher-priority queues (foreground) will typically use RR while lower-priority queues (background) might use FCFS.

A task will start at the highest-priority queue. If the task uses up all the resources (e.g. q for RR), it will get pushed one level down as a penalty. If the task does not use up all the resources (e.g. less than q for RR), it will get pushed one level up as a reward. This ensures that long running tasks (e.g. CPU bound) don't hog all resources and get demoted into low priority queues while short running tasks (e.g. I/O bound) will remain at the higher-priority queues.

1.1 Round Robin T/F

1. The average wait time is less than that of FCFS for the same workload.

   False. This is generally not true when the time quantum is small. Consider tasks A, B, and C (arriving in this order) which each take 3 time units. With FCFS, the average wait time is 3. With RR ($q = 1$), average wait time is 5.

2. It requires preemption to maintain uniform quanta.

   True. Without preemption, a task would just run forever without adhering to the quantum.

3. If a quantum is constantly updated to become the number of cpu ticks since boot, Round Robin becomes FCFS.

   True. Quantum never gets used for any task since it always increases as the task progresses.

4. Cache performance is likely to improve relative to FCFS.

   False. RR usually results in more context switches when compared to FCFS, meaning the cache will have more misses.

5. 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.

False. There exists context switching overhead.

1.2 Life Ain't Fair

Suppose the following threads (**priorities given in parantheses**) 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 (prio = 7)
1
2   Kevin (prio = 1)
3   Neil (prio = 3)
4
5   Akshat (prio = 5)
6
7   William (prio = 11)
8
9   Alina (prio = 14)
```

Determine the order and time allocations of execution for each given scheduler scenario. 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.

1. Round robin with time quantum 3

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 just before Taj. 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

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

2. SRTF with preemptions

Since each thread takes 5 ticks, a thread will never be preempted once begun. As a result, each thread will just execute in order of arrival (i.e. FIFO).

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

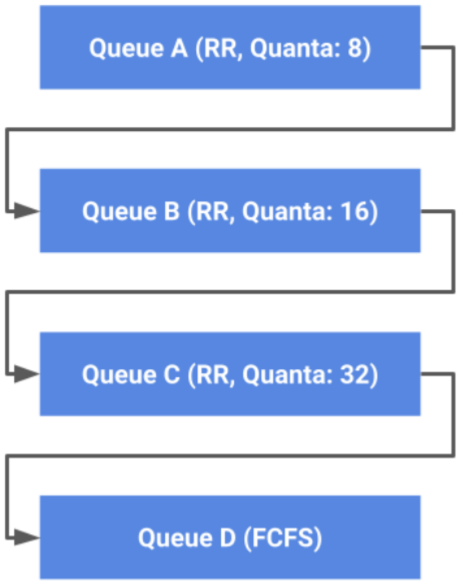
3. Preemptive priority

Even though Kevin and Neil arrive while Taj is running, Taj has higher priority, so it runs to completion. Just as Taj finishes, Akshat arrives with a higher priority than Kevin and Neil who are waiting, so Akshat runs for two clock ticks until William arrives with a greater priority. Similarly, William runs for two clock ticks until Alina arrives with a greater priority. Alina runs to completion, then the remaining threads each run to completion based on priority.

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

1.3 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.



1. 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) {
    _______________
    _______________
    _______________
  }
}
```

bitcoin_mine.c

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

bitcoin_mine_sol.c

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.

2. 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.