Section 6: Synchronization, Scheduling

July 13, 2020

Contents

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.

2 Synchronization

2.1 test_and_set

In the following code, we use test_and_set to emulate locks.

```
int value = 0;
int hello = 0;

void print_hello() {
    while (test_and_set(&value));
    hello += 1;
    printf("Child thread: %d\n", hello);
    value = 0;
    pthread_exit(0);
}

void main() {
    pthread_t thread1;
    pthread_t thread2;
    pthread_create(&thread1, NULL, (void *) &print_hello, NULL);
    pthread_create(&thread2, NULL, (void *) &print_hello, NULL);
    while (test_and_set(&value));
    printf("Parent thread: %d\n", hello);
    value = 0;
}
```

Assume the following sequence of events:
1. Main starts running and creates both threads and is then context switched right after
2. Thread2 is scheduled and run until after it increments hello and is context switched
3. Thread1 runs until it is context switched
4. The thread running main resumes and runs until it get context switched
5. Thread2 runs to completion
6. The thread running main runs to completion (but doesn't exit yet)
7. Thread1 runs to completion

Is this sequence of events possible? Why or why not?

Yes. In steps 3 and 4, the main thread and thread1 make no progress. They can only run to completion after thread2 resets the value to 0.

At each step where `test_and_set(&value)` is called, what value(s) does it return?

1. No call to test_and_set
2. 0
3. 1, 1, ..., 1
4. 1, 1, ..., 1
5. No call to test_and_set
6. 0
7. 0

Given this sequence of events, what will C print?

```
Child thread: 1
Parent thread: 1
Child thread: 2
```

Is this implementation better than using locks? Explain your rationale.

No, this involves a ton of busy waiting.

3 Scheduling

3.1 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. Easy to find counter example. 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.

3.2 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   Sam : 7
1
2   Kevin : 1
3   Bobby: 3
4
5   Jonathan : 5
6
7   William: 11
8
9   Alex: 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 Sam 3 units at first looks like:

```
0   Sam
1   Sam
2   Sam
```

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

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

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

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

```
RR:
0   Sam
1   Sam
2   Sam
3   Kevin
4   Kevin
5   Kevin
6   Bobby
7   Bobby
8   Bobby
9   Sam
10  Sam
11  Jonathan
12  Jonathan
13  Jonathan
14  Kevin
15  Kevin
16  William
17  William
18  William
19  Alex
20  Alex
21  Alex
22  Bobby
23  Bobby
24  Jonathan
25  Jonathan
26  William
27  William
28  Alex
29  Alex
```

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

Preemptive Priority
0   Sam
1   Sam
2   Sam
3   Sam
4   Sam
5   Jonathan
6   Jonathan
7   William
8   William
9   Alex
10  Alex
11  Alex
12  Alex
13  Alex
14  William
15  William
16  William
17  Jonathan
18  Jonathan
19  Jonathan
20  Bobby
21  Bobby
22  Bobby
23  Bobby
24  Bobby
25 - 29 Kevin
```

3.3 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?

The entire first 50 cycles is spent running `Kiki`.

```
@12 : a = 1, b = 1
@23 : a = 1, b = 2
@34 : a = 2, b = 3
@45 : a = 3, b = 5
```

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

`a = 3, b = 5`

There was deadlock! After part (a), the last instruction Thread 1 ran (cycle 50) is line 6, `lock_a.acquire()`. Thread 2 can run until it has acquired lock b, but blocks on `lock_a.acquire()` as Thread 1 is currently holding lock a. Thread 3 can run until `lock_a.acquire()`, then it will block as well. When Thread 1 is switched back in, it cannot run the line `lock_b.acquire()` because Thread 2 is holding lock b.

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?

```
@12 : a = 1, b = 1                   @59 : T2 blocked on line 6
@23 : a = 1, b = 2                   @61 : T1 context switched on
@34 : a = 2, b = 3                   @67 : a = 5, b = 8
@45 : a = 3, b = 5                   @79 : a = 8, b = 13
@51 : T1 pre-empted after line 6     @91 : a = 13, b = 21
@53 : T2 context switched on.        @100 : a = 21, b = 34
```

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?

```
@50 thread1 has acquired lock A
@53 thread 2 begins to run
After 58, context switch back to thread 1
After 64, thread 1 releases lock_a and thread 2 runs
After 67, thread 2 acquires lock_a
@68 thread 2 attempts to acquire lock_b
@71 thread 1 releases lock_b
After 73, lock_b is acquired by thread 2
a = 13, b = 21
```