CS-323: OS Exercise Week 4 (Fall 2023)

October 11, 2023

Last name, first name: ____

1 Context switch

(a) What are four causes for a context switch?

- The running process completes/exits.
- The running process executes a system call causing it to be blocked (e.g. synchronously loading data from the disk) and the OS scheduler switches to another task that is ready.
- The hardware requires OS help and causes an interrupt, an example could be a key press on the keyboard.
- The OS scheduler is woken up by the timer interrupt and decides to preempt the running process and switches to another.

(b) Why might a process not yield the CPU in a cooperative multi-tasking environment (2 reasons)?

- The process might have encountered a bug (i.e. entered an infinite loop).
- The process might be maliciously trying to keep all the system resources to itself.

(c) How can the OS mitigate the risk of cooperative multi-tasking and regain control from misbehaving processes?

The OS sets up a regular timer interrupt for the scheduler. Every time the timer expires, a hardware interrupt is triggered and switches to the OS scheduler, which will decide whether to continue running the currently running process or switch it out. The length of time between these interrupts is known as the scheduler tick. Typical values include 10-100ms.

(d) Which scheduler metrics (utilization, turnaround time, response time, fairness, and progress) can be badly impacted by setting an extremely small tick?

Context switching is not free and this means that the time spent executing useful programs (utilization) will be lower. As is often the case in operating systems, there is a trade-off as a longer tick time can have a negative impact on the other metrics (fairness, response-time, turnaround-time) depending on the typical job length .

(e) Why is it important for a scheduler to distinguish between CPU- and I/O-bound tasks?

CPU-bound tasks tend to be longer running jobs that rarely perform I/O and might block shorter running tasks run for too long. On the other hand, I/O-bound tasks will often request the CPU for short period of time before going back to waiting on I/O. Delaying these short bursts of CPU activity will slow down the tasks while executing them have little impact on overall CPU consumption.

The most basic examples would be a simple text editor being an I/O bound task and compilation being CPU bound. The text editor mainly uses the CPU when keys are being pressed and is idling

Name: ____________________________

waiting for key presses otherwise. Compilation will only do a little I/O for reading in source code at the start and outputting compiled code at the end, the whole time in between will be spent on the CPU analyzing source code and performing optimizations. We want I/O bound processes to have low latency to keep their responsiveness acceptable.

(f) You have a CPU-bound task that requires 1 second of CPU time. The overhead of a timer interrupt, scheduling and the context switching to the OS and back all together require 500μs. What is the efficiency/utilization of the system when the tick is 1ms, 10ms and 100ms respectively?

For a particular tick, the number of ticks required to complete the task: `task_length / ticks`.

For 1ms, the task requires $1,000$ ticks. After each tick, the OS interrupts the task, incurring 500 μs overhead. In total, this setup incurs $1000 * 500\mu s = 500$ms overhead. Overall, efficiency $= 1s/(1s + 500ms) = 0.66$. 34% of the processor time is lost due to kernel overhead. Therefore, 1ms tick is wasteful for this task.

For 10ms, the task requires 100 ticks. After each tick, the OS interrupts the task, incurring 500μs overhead. In total, this setup incurs $100 * 500\mu s = 50$ms overhead. Overall, efficiency $= 1s/(1s + 50ms) = 0.95$. 5% of the processor time is wasted on overheads, which is already much better than the first case.

For 100ms, the task requires 10 ticks. After each tick, the OS interrupts the task, incurring 500 μs overhead. In total, this setup incurs $10*500\mu s = 5$ms overhead. Overall, efficiency $= 1s/(1s+5ms) = 0.995$. Only 0.5% of the processor time goes to OS overheads. This setup is very efficient for throughput of CPU bound tasks.

Assuming the task length is divisible by the tick length, the long-term utilization calculation is simply `tick_length / (tick_length + switch_overhead)`.

(g) You have a CPU-bound task A running along with an I/O bound task B (eg. typing in a word processor). The word processor uses polling and reads a single character from the keyboard every time it runs and then yields. With ticks of 1ms, 10ms and 100ms, what is the max typing speed the word processor can run at? Assume that the scheduling is round-robin and a timeslice of one tick.

The schedule of tasks will be: Task A runs for one tick, Task B reads one character and blocks, Task A runs for another tick, Task B reads another character, and so on and so forth.

Effectively, task B is allowed to run again after one tick of task A running. Let us ignore the cost of scheduling, interrupts and context switching. Also, we assume that the task B runtime to poll for a character and process it is negligible, since it is a very short operation.

With 1ms ticks, task B can run approximately $1,000$ times per second, which is fast enough to support even the fastest typer.

With 10ms ticks, task B can run approximately 100 times per second, which is again fast enough to support even the fastest typer.

With 100ms ticks, task B can run approximately 10 times per second, which is slower than some touch typists. They will encounter noticeable delay while typing. Further, for applications like e-sport games, the 100ms delay between inputs will make the game unplayable.

This, along with the previous question, illustrates the tradeoff between utilization and latency (response-time) for schedulers.

2 FIFO scheduler

(a) What is one example of a FIFO scheduler in daily life?

The checkout register at grocery stores.

(b) Consider a FIFO scheduler as described in the lectures with the following arrival pattern:

Name: ______________________

Arrival time	Task length
0	10
1	2
5	20
11	5

(a) When will the CPU be done with executing all the tasks?

10 + 2 + 20 + 5 = 37 time units

(b) What is the turnaround time?

The first task arrives at 0s and finishes at 10s. Turnaround time = 10s.

The second task arrives at 1s and finishes at 12s. Turnaround time = 11s.

The third task arrives at 5s and finishes at 32s. Turnaround time = 27s.

The fourth task arrives at 11s and finishes at 37s. Turnaround time = 26s.

(10 + 11 + 27 + 26) / 4 = 18.5s is the average turnaround time.

(c) What is the average response time?

The first task arrives at 0s and starts at 0s. Response time = 0s.

The second task arrives at 1s and starts at 10s. Response time = 9s.

The third task arrives at 5s and starts at 12s. Response time = 7s.

The fourth task arrives at 11s and starts at 32s. Response time = 21s.

(0 + 9 + 7 + 21) / 4 = 9.25s is the average response time.

(d) If we applied a Shortest Job First scheduler, which metrics would be impacted and in which way?

With this sequence of jobs, both the average response time and turnaround time would decrease as the third and fourth task would be inverted.

The first two tasks are the same as before.

The fourth task arrives at 11s, starts at 12s and finishes at 17s. Turnaround time = 6s, response time = 1s.

The third tasks arrives at 5s, starts at 17s and finishes at 37s. Turnaround time = 32s, response time = 12s.

Average turnaround time: (10 + 11 + 6 + 32) / 4 = 14.75s
Average response time: (0 + 9 + 1 + 12) / 4 = 5.5s

3 MLFQ scheduler

Consider a Multi-Level Feedback Queue scheduler with 3 queues and a time slice of length 1 tick, 2 ticks, and 4 ticks for priority high to low.
Every 10 ticks, the scheduler will boost the priority of unexecuted tasks by one level.

(a) For each tick, note which task is executed and its priority level.

Task ID	Arrival time	Task length
0	0	7
1	0	3
2	3	6
3	3	0.99 + I/0 for 16 (blocked) + 5
4	4	7
5	5	10

Assuming 2 as the highest priority and 0 as lowest. Remember tasks enter from the highest priority. Also, tasks will go down if they use up their whome time slice.

Name: ______________________________

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Task ID	0	1	0	2	3	4	5	0	1	1	2	2	4	4	5	5	0	0	0	0
Priority	2	2	1	2	2	2	2	1	1	1	1	1	1	1	1	1	0	0	0	0

Time	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
Task ID	2	3	3	3	4	4	5	5	2	2	3	3	4	4	5	5	5	5	5
Priority	0	2	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0

Task 0's time slice is remembered between 2 and 7. This is because MLFQ continues to keep track of the running time even if runtime is interrupted. See section 8.4 in OSTEP. A priority boost occurs at 24, as task 4 was waiting for the last 10 time units or more.
A priority boost occurs at 26, as task 5 was waiting for the last 10 time units or more.
Task 3 reenters the ready queue at time 21 and is scheduled with the highest priority as it did not consume its whole previous time slice ($0.99 < 1$). The effect this 0.01s has on very slightly shifting the schedule earlier is ignored.

4 More advanced schedulers

You might need to consult additional material for answer some of these questions.

(a) A desirable property for schedulers is fairness. What does this mean for a scheduler?

Fairness represents how close a scheduler is, over a period of time, to assigning for each task its proportional share of execution time. As an example, if we had 5 tasks and were examining 10 seconds, the fairest situation would be each of them having gotten the CPU for 2 seconds.

(b) A simple fair scheduler is lottery scheduling. How does it work? Why is it fair?

Every now and then, the CPU runs a lottery where it randomly picks a number below a fixed value. Beforehand, it assigns each number (or more simply, ranges) to a specific tasks. The task which holds the number drawn will be executed next.
As long as each task is given a fair number of tickets and over a big enough period, the execution time of each task will correspond to the ticker repartition and therefore be fair.

(c) In addition to a fair scheduler, Linux also has the concept of process `niceness`. Why is it useful? Who can adjust the niceness of each task?

Linux allows the user to specify if their task is important via niceness. Users (programs) are the only ones that know what priority a process should have and adjust accordingly, and the scheduler itself does not adjust niceness. The Linux scheduler (CFS) then uses this value to slightly modify task runtimes: a nice process will seem like it has run more than it actually has, while a less nice process (high priority) will seem like it has run less than it actually has (so it should run more!). `niceness` in this case allows for well-behaved programs to hint the scheduler about un/important tasks.

(d) Can you find a real-world example where a process in a "scheduler" uses niceness?

Someone with a full grocery cart that lets you pass at a grocery store register to pay your single item.

(e) Some operating systems (e.g., Real-Time Operating System (RTOS)) allows program to specify deadlines for each task. Why is it interesting for systems dealing with a lot of real-time interaction to have the concept of deadlines? How would you change a simple SJF scheduler for a RTOS?

Dealing with the world in real-time means that we should act quickly too. For example, a robot should be responsive when interacting with a human. Additionally, it needs to process data coming from the real-world. In certain cases, newer data might make older data obsolete. By adding deadlines, we allow a process to specify at which point a task should at the latest finish running for its results not to be obsolete. An important application comes in automobiles and airlines. The input from the brake pedal *must* be processed within a certain deadline, or there is danger. At the same time, the multimedia system can tolerate a small delay of a few seconds in the music.

Name: ____________________________

We can make a simple modification to SJF: Instead of running the shortest job first, we can have our scheduler run the nearest deadline job first.