CS 350: Homework #5

Due 3PM, Thursday, March 17

Kristel Tan (ktan@bu.edu)

March 1, 2016

Question 1

$$\lambda = 100 \; requests/\sec$$

$$mean\ service\ time = T_S = \frac{1}{2}*(0+19) = 9.5\ ms$$

$$\rho = \lambda * T_S = 0.1*9.5 = 0.95$$

$$\sigma_{Ts}^2 = \frac{(19-0)^2}{12} \approx 30.08$$

$$\sigma_{Ts} = \sqrt{30.08} \approx 5.48$$

$$A = \frac{1}{2}\left[1+\left(\frac{\sigma_{Ts}}{Ts}\right)^2\right] = \frac{1}{2}\left[1+\left(\frac{5.48}{9.5}\right)^2\right] \approx 0.67$$

$$q = \frac{\rho^2 A}{1-\rho} + \rho = \frac{0.95^2 * 0.67}{1-0.95} + 0.95 \approx 13.04$$

$$T_q = \frac{q}{\lambda} = \frac{13.04}{0.1} = 130.4\ ms$$

# requests arriving after t could be serviced before R in the worst case

$$= \frac{1}{2} * \lambda * T_q = \frac{1}{2} * 0.1 * 130.4 = 6.52$$

Question 3

Part a

The system described in this problem services requests in a comparable manner to that of a Shortest-Seek First (SSF) scheduler, which particularly causes starvation under heavy loads. In the instance that requests for files in the cache continuously keep coming in, the server may never get around to servicing requests for files in the disk found in Q2. This happens because Q1 (storing requests for files in the cache) essentially has priority over Q2. For this reason, the system may experience the "head stickiness" phenomenon in which the server becomes "head stuck" for requests in Q1. This leads to starvation of requests in Q2.

Part b

My friend is thinking of an N-Batched SCAN scheduling algorithm when suggesting the above fix.

Part c

A larger N is better for improving the cache performance because it minimizes head movement by the server. This behavior is modeled my an F-SCAN scheduler. Because no new requests are permitted to come in when serving a single batch, the server can first serve all requests in Q1, accessing the cache and then serve the lower-priority queue, Q2.

Part d

A smaller N is better for fairness because the smaller that N gets, the closer the system resembles a FIFO scheduler. By nature, FIFO is one of the fairest scheduling systems because requests are serviced in order. Similarly in this web server, the less number of requests there are, the more likely they will be served in order of arrival as well.

Part e

- Given that the load on the system is light and there is a strong locality of reference, I would set the value of N to 1. Because the load is light, efficiency is not a high priority for the system and servicing each request one at a time will not be prone to starvation. A smaller N will also make for a fairer system.
- Given that the load on the system is moderate and there is a strong locality of reference, I would set the value of N to 10. At this rate, we will want to start taking into account the efficiency of the system. As we know from part (c), a larger N can help improve performance by minimizing head movement. This will also in turn preserve the strong locality of reference.
- Given that the load on the system is high and there is no locality of reference, I would still set the value of N to 10. Under heavy loads, we know that this system can be prone to "head stickiness". To eliminate that, we can serve the requests 10 at a time, making sure that every request in each queue gets served. The reason that we don't want to make N too large yet at this point is that we don't want to sacrifice too much fairness.
- Given that the load on the system is extremely high and there is a strong locality of reference, I would set the value of N to 100. However, because the load is extremely high, we may also want to process the batches with a larger N so as to keep the system's performance efficient.

Question 4

Part a

Yes, starvation is possible in this system because it does not treat jobs fairly enough. Hence why jobs of class C are most susceptible to starvation because they are of lowest possible priority. Higher priority jobs preempt jobs of class C, always being served first. In the case that there are many jobs of class A or class B coming in, class C jobs will starve.

Part b

- The worst-case response time for jobs of class A is 5.0 seconds because it has the highest priority.
- The worst-case response time for jobs of class B is 5.5 seconds because it must wait for class A jobs and requires the CPU for 0.5 seconds.
- The worst-case response time for jobs of class C is is 50 (5.0 + 5.5 + 20.0 + 19.5) seconds. What happens is that a class A job takes 5 seconds. During this time, five more class B jobs enter, taking 5.5

seconds total until a class C job can finally enter at 10.5 seconds. It will take 20 seconds total for the class C job to execute. However, a total of 39 more class B jobs will interrupt and execute during C's attempt at execution, totaling 39 * 0.5 sec = 19.5 seconds. This totals 50 seconds worst-case response time for jobs of class C.

Part c

$$\frac{\lambda_A}{Ts_A} + \frac{\lambda_B}{Ts_B} + \frac{\lambda_C}{Ts_C} \leq \rho$$

$$\frac{5}{60} + \frac{0.5}{1} + \frac{c}{300} \leq 1$$

$$\frac{c}{300} \leq 1 - 0.583$$

$$\frac{c}{300} \leq 0.417$$

$$c = 125.1$$

maximum CPU time = 125.1 sec

Part d

Priority inversion is possible for the priority scheme devised in this problem in the instance that a class C job gets a hold of resource R first. If a class A job then enters the system during this time, it would not be able to get a hold of resource R because C has it and also cannot compete for CPU. In this case, although the class A job is presumably of higher priority, it would end up having to wait for the class C job of lower priority.

Part e

The worst case response time for jobs of class A occurs when a class A job must wait for a class C job holding the resource R and CPU. When a class C job holds R, the class A job cannot use it until it is released. However as the class C job is executing, 39 class B jobs will interrupt it again as well, totaling 19.5 seconds. Then, in the case that jobs of class C still take 20 seconds to utilize both resources and jobs of class A take 5 seconds, the worst case response time for jobs of class A would be 44.5 seconds (20.0 + 19.5 + 5.0).

Part f

Priority inversion refers to a phenomenon in which a higher priority process is preempted by a lower priority process when sharing a resource. In other words, a higher priority process must wait for a lower priority process to complete before it can utilize the resource. An example of this in the article is when a high priority information bus thread was sometimes blocked waiting for a low priority meteorological data thread.

Part g

Priority inheritance refers to the attribute of a low priority process when it "inherits" or acquires the highest priority. This may happen when processes of different priorities share a resource with each other, but because a low priority process is holding a resource first, other processes coming after it must wait. That low priority process holding the resource has now inherited a higher priority than it normally would have.

Part h

The worst-case response time for jobs of class A occurs when a job of class C has inherited priority. In this case, a class A job must wait for 20 seconds while the class C job uses the CPU. Then A can finally execute for 5 seconds, totaling 25 seconds for the job to finish.