HW5
Adrian Law

1. Lambda = r/sec
   Ts = 0.008 sec

   Little's Law:
   P = r * 0.008
   Q = (0.008r)/(1-0.008r)
   Tq = 0.008 / (1-0.008r)

   Average Q = 0/5 * r * 0.008/(1-0.008r)
   = 0.004/(1-0.008r)
2.
   a. Non-preemptive HSN is unfair to I/O bound jobs:

Arrival Time	Process Name and Type	Service Time	Service Start	Slowdown
0	P1, CPU	1000	0	1
1	P2, I/O	10	1000	99.9

   In this example, a CPU job enters the system at t=0 before an I/O job, and the scheduler assigns the CPU task first. The I/O which arrived at t=1 has to wait for an inordinate amount of time, which can cause slowdown to grow very large. Thus if the schedule ever only sees CPU bound jobs and picks it, it can spell disaster for any I/O jobs that arrive during the processing period of the CPU job.

   b. HSN preemptive to job arrivals can still be unfair to I/O bound jobs:

Arrival Time	Process Name and Type	Service Time	Service Start	Slowdown @ Start
0	P1, I/O	10	0,1002	1 ,100.1
1	P2, CPU	1000	2	1.001
2	P3	-	-	-

   In this example, the I/O job enters before a CPU job. At t=0, the I/O job initially gets serviced, and even after the arrival of a CPU job, assuming the schedule continues to pick the I/O job (Both P1 and P2 have slowdowns of 1 since 9/9 = 1000/1000). However, the arrival of a new process P3 at t=2 causes the schedule to revaluate the slowdowns of all jobs. Since only P2, the CPU job, has a slowdown greater than 1, it will get assigned (The type and service time of P3 does not matter in this example, as it is a new arrival it can't have a slowdown greater than 1). Without any further arrivals, I/O is again waiting for the CPU job to finish, causing slowdown to grow very large.

This scheduler does improve if there are more job arrivals, however, since more jobs means that the scheduler more frequently revaluates all slowdowns and calculates that I/O jobs are massively slowed own. However, in a sequence similar to the example, if a I/O job arrives before a lull in arrivals, it may have to wait very long before being serviced.

c. HSN preemptive to a fixed timeout is better, but not ideal for I/O bound jobs:

Arrival Time	Process Name and Type	Service Time	Service Start	Slowdown
0	P1, CPU	1000	0	1
1	P2, I/O	10	11	2

With fixed timeouts, the size of slowdown grows more slowly for I/O jobs compared to the previous schedulers. In this example, the timeout size is 11. CPU job arrives at t=0 and is serviced for the first 11 seconds. The I/O job arrives at t=1 and waits for an additional 10 seconds before its selected by the scheduler, which by then already has a timeout of 2.

Thus there is somewhat of a "bound" on the slowdown of I/O jobs before they are chosen. This bound grows as the load increases, but it will grow at a slower rate than the other forms of HSN mentioned above. This scheduler is also limited in that the fixed timeout should be at least as big as the longest I/O job, and more frequent timeouts can cause costs in overhead.

d. HSN can be improved by changing how slowdown is calculated in a scheduler preemptive to job arrivals. By adding a constant factor C to slowdown = (t – arrival time + expected job length + C) / expected job length, the scheduler can determine how "fast" slowdown grows for each job, and continue to pick the shortest jobs first without starving CPU jobs.

Arrival Time	Process Name and Type	Service Time	Service Start	Slowdown @ Start
0	P1, I/O	10	0	2 (1), 1.9 (1)
1	P2, CPU	1000	10	1.01 (1)
2	P3	-	-	-

Using a factor C = 10 and with the same example from B), the I/O job is chosen even after the arrival of a CPU job at t = 1. The constant factor increases the slowdown of P1 at t=0 to 2 from 1, and at t=2 to 1.9 from 1. This allows HSN to pick I/O jobs first as their slowdowns grow much more quickly than CPU bound jobs, and still takes advantage of the fact that CPU jobs cannot starve in a HSN scheduler.

3.

a. In this example, it is possible for the cache to starve out a request:

   Cache stores files f1 to f10
   Q1: f1, f1, f1, f2, f7, f8, f2, f4, f10, f2, f3, f5…
   Q2: f11

   If requests in Q1 keep increasing, then Q2 will never be served and new files won't be cached from disk.

b. It is similar in concept to an N-Batch scan; you are servicing N requests as a group at a time.

c. Large values of N are more efficient since they are better suited to taking advantage of locality, as similar requests are more likely to be received together in a system of high locality.

d. Small values of N are fairer since it reduces the bound on the max number of out-of-order requests that are served before the current request.

e.
   i. 1, light loads means that even though there's strong locality, there are not enough requests building up in batches that a FIFO system wouldn't take up any more time than the overhead of a N-batch system would take.
   ii. 10, moderate load means that there will be multiple batches building up for a small size N, but not enough to justify increasing efficiency by a higher magnitude of N. Since there is strong locality, 10 will still be adequate for serving these specific files by exploiting that efficiency.
   iii. 1, since there is no advantage of an N-batch system compared to FIFO, it is better to remove the overhead all together and go with FIFO.
   iv. 100, heavy load means that N should be large to maximize efficiency in a tradeoff for fairness.

4.

a. Yes, if B arrives quite frequently to the point where the CPU is preempted for A and B requests, then C won't be served by the CPU and will be starved out.

b. Assuming that when B class requests are being selected the scheduler doesn't take into account which arrived first, maximum response times (Tq = Tw + Ts) for:

   A = 0 (top priority) + 5 (service time) = 5
   B = 5 (waiting for A) + 4.5 (waiting for other Bs to finish due to backup by A) + 0.5 (service time) = 10
   C = 25 (waiting for A and B) + 20 (service time) = 45

c. $(5/60) + (0.5/1) + (c/500) = 1$
   $c = 125$
   If C takes more than 125 seconds out of 5 minutes to finish, then the response times of C will grow infinitely as the CPU can't enter steady state.

d. If C arrives right before A, then its possible that A will then have to wait for C to finish as it can't use any resource till R frees up. This is exacerbated if B arrives, as then B blocks C from using the CPU and thus A is now waiting for C and for B.

e. A waiting for C (waiting for B to finish) = 40

f. Similar to the case in D, priority inversion occurred on the Mars Rover when a low priority process, the meteorological data collection, uses the non-preemptable data bus, and the high priority process, the bus management, arrives. The medium priority process, communications, is very long running and blocks the data collection from finishing, thus the bus management process has to wait for lower priority cases to finish before running. On the rover, this causes system resets as it detects that the bus hasn't been used after a fixed timeout

g. Priority inheritance occurs when a non-preemptable resources like the bus is held by any process. That specific process "inherits" a high priority status when it is using that resource, allowing it to more quickly finish using the bus and freeing it up for other processes.

h. A waiting for C only = 20