Zhenyu Yan

**Report 4**

In the report, I conclude the algorithm and implementation of non-aggressive multi-level feedback scheduler , aggressive multi-level feedback scheduler, Preemptive shortest job first scheduler, and lottery scheduler. Also discuss the Correctness Testing , Efficiency testing and compare the policy I implemented.

**Non-****aggressive Multi-level Feedback Scheduler and aggressive Multi-level Feedback Scheduler**

I implement non-aggressive by following this algorithm:

1. Jobs are scheduled according to priority, which ranges from 0 (highest priority) to 7 (lowest priority)
2. When a job starts a burst (that is, when it becomes ready either because it has just started or because it has finished doing I/O), it is assigned  priority 0 .
3. The scheduler maintains a (FIFO) queue of jobs for each priority level.  The scheduler will always run the first job of the highest priority level available (i.e. lowest-numbered non-empty queue). For example, if queues 0 and 1 are empty but queue 2 is not, the scheduler will run the first job in queue 2.
4. When a job is run, it is assigned a slice, which is a number of quanta based on the priority of the job. A job at priority level 0 has a time slice length of 1 quantum, a job at level 1 has a time slice of 2 quanta, a job at level 2 has a time slice of 4 quanta, and so on. In general, a job with priority i has a time slice of  2***i*** quanta.
5. If a job with priority i uses up its time slice without blocking for I/O or terminating, the scheduler stops it, lowers its priority to i+1, and adds it at the tail of queue i+1, and selects a job as in rule (3).  However if the job is already in the lowest priority queue, its priority is unchanged and it returns to the end of the same queue.  While it is possible that the same job will be selected again--for example, if it is the only ready job--normally a different job will be given the opportunity to run.
6. This policy is non-aggressive in the following sense:   If a job becomes ready while another job is running, it is added to the tail of queue 0, but the running job is not stopped until it terminates, blocks for I/O, or uses up its time slice.

I implement aggressive by following this algorithm:

This version is a modified version of your first version. In this version jobs arriving at the CPU scheduler can preempt running jobs, and the priority of a job is ``remembered'' from one burst until the next. In more detail, rules (2) and (5) are modified as follows:

1. (2)'  When a job becomes ready because it has finished doing I/O, it is given priority i-1, where i is the priority it had when it blocked for I/O.  There is no level -1, so if a job finishes a burst at priority 0, it stays at priority 0. Newly created jobs are assigned priority 0.

2. (5)'  This policy is **aggressively preemptive**in the following sense:  If a job becomes ready while another job is running, it is added  to the tail of the appropriate queue as defined by rule (2'), the running job is stopped and has 1 subtracted from its priority (unless it is already at priority 0), it is added to the tail of the appropriate queue, and another job is selected to run as in rule (2).

The major different for aggressive multi-level feedback and non-aggressive multi-level feedback is aggressive multi-level feedback can preempt the current process if priority is higher, and non-aggressive multi-level feedback cannot preempt and must wait till the current process finish.

Below is the code to determine if the scheduler can preempt or not:

int scheduler\_can\_preempt\_multilevel**(** process\_t**\*** p **){**

**if(**SCHEDULER\_AGRESSIVE**){**

**for(**int i**=**0**;** i **<** p**->**priority**;** i**++){**

**if(**ready\_queue**[**i**]->**count **!=** 0**){**

**return** 1**;**

**}**

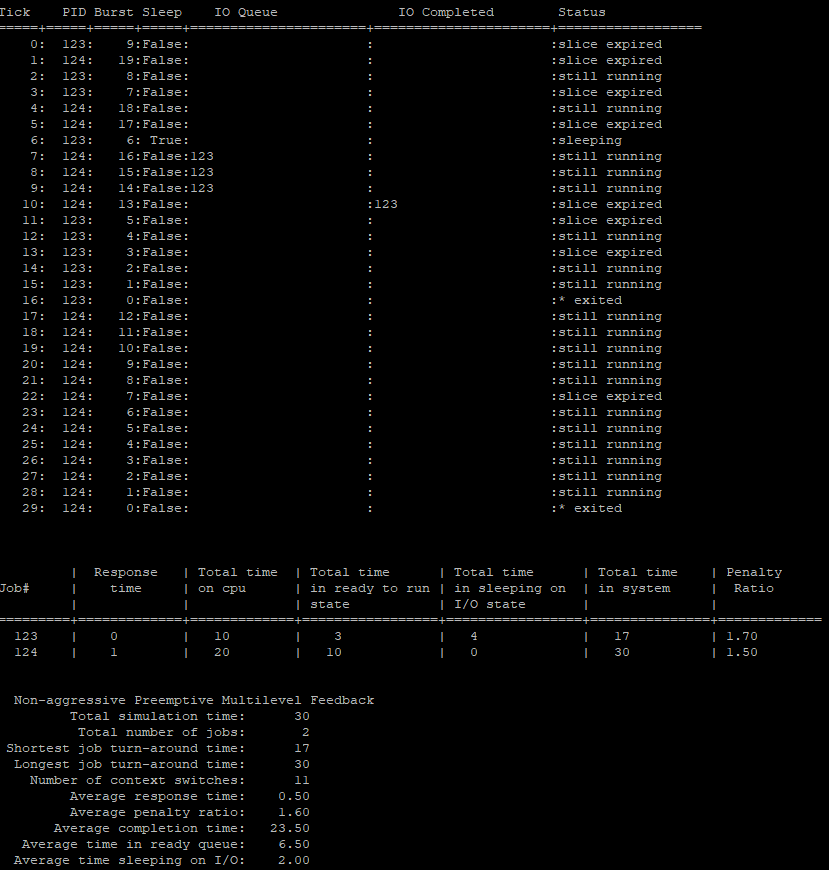
**}**

**}**

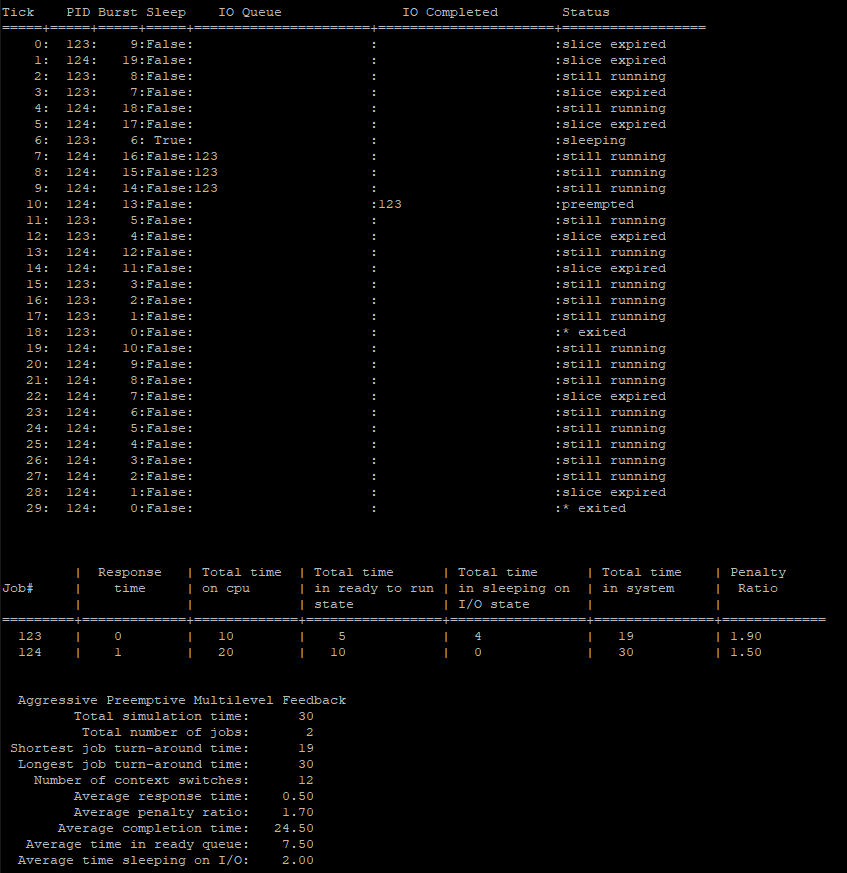
**return** 0**;**

**}**

The result for non – aggressive:



The result for non – aggressive:

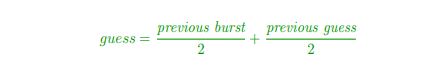


compare to the sample output, there is one more switch on the aggressive more than the sample.

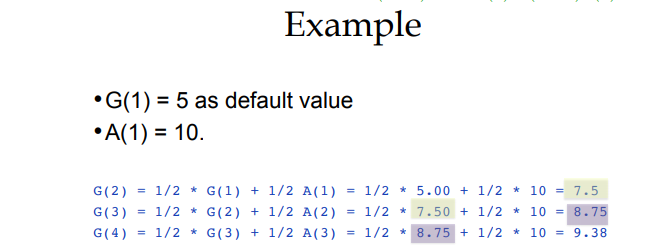
**Preemptive shortest job first scheduler**

I implement sjf scheduler by following this algorithm:

In this strategy the ready queue will consist of one queue ordered according to the time that the scheduler 'thinks' the job needs on the CPU. You will need to calculate this "guess" using exponential averaging (p. 269 in textbook). The weight of the most current value is w and the default weight is 1/2. Suggest using 5 as the default for the initial guess G(1)=5, as we did in lecture.

The most important part is to how to calculate the exponential averaging. In the book, the fomular is : 

The example in lecture:

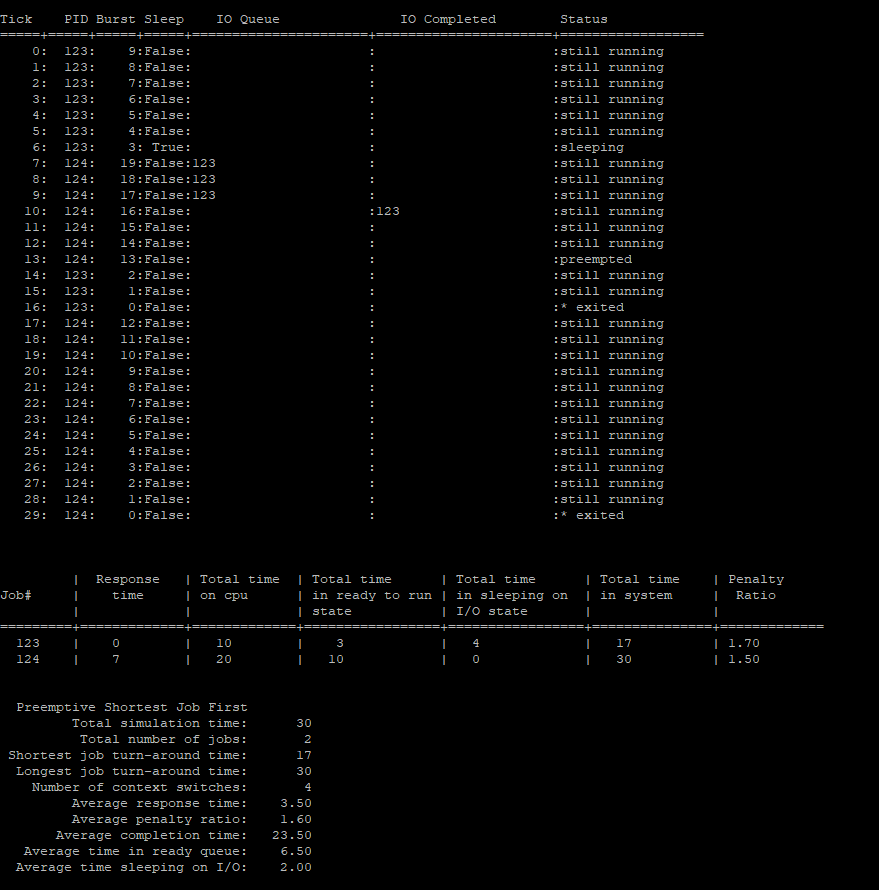


As the fomular and the example in our lecture, we can come up the implementation for the brust guess below:

p**->**sjf\_burst **=** 0.5 **\*** p**->**sjf\_burst **+** 0.5**\*(**clock - p**->**timeslice\_started**);**

(remember, the initial guess is set to 5.0)

result:



Compare to the sample, my result is very close excpet there is one less number of context switches and one more seconds in the total time in ready to run states which effect the total time in system by one second too.

**Lottery Scheduler**

I implement lottery scheduler by following this algorithm:

statistically guarantees a variable fraction of processor time to each runnable process. The concept is much like a lottery. At each scheduling decision, each runnable process is given a number of "lottery tickets". Then a random number is generated, corresponding to a specific ticket. The process with that ticket gets the quantum.

The implementation is below, as you can see, I sum up all the tickets and generate a random number between 0 and the total tickets +1. Then I find which jobs own that particular ticket.(below)

process\_t**\*** scheduler\_get\_job\_lottery**()**

**{**

process\_t**\*** job **=** **NULL;**

**if(**queue\_index **==** 0**){**

**return** 0**;**

**}**

int totalTickets **=** 0**;**

**for(**int i **=** 0**;** i **<** queue\_index**;** i**++){**

totalTickets **+=** queue**[**i**]->**lottery\_tickets**;**

**}**

int bingo **=** **(**mrand**()** **%** totalTickets**)+**1**;**

int currentTicket **=** 0**;**

**for(**int i **=** 0**;** i **<** queue\_index**;** i**++){**

currentTicket **+=** queue**[**i**]->**lottery\_tickets**;**

**if(**currentTicket **>=** bingo**){**

job **=** queue**[**i**];**

queue\_index**--;**

queue**[**i**]** **=** queue**[**queue\_index**];**

job**->**timeslice\_started **=** clock**;**

**break;**

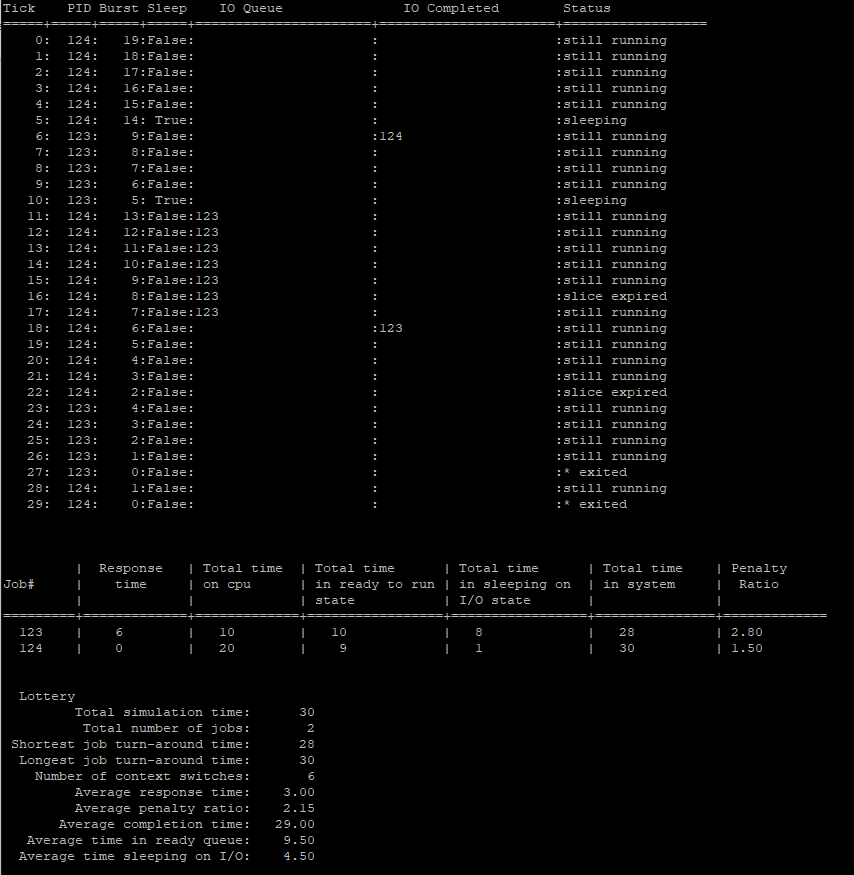
**}**

**}**

**return** job**;**

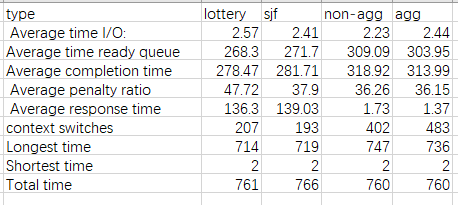
**}**

Result:



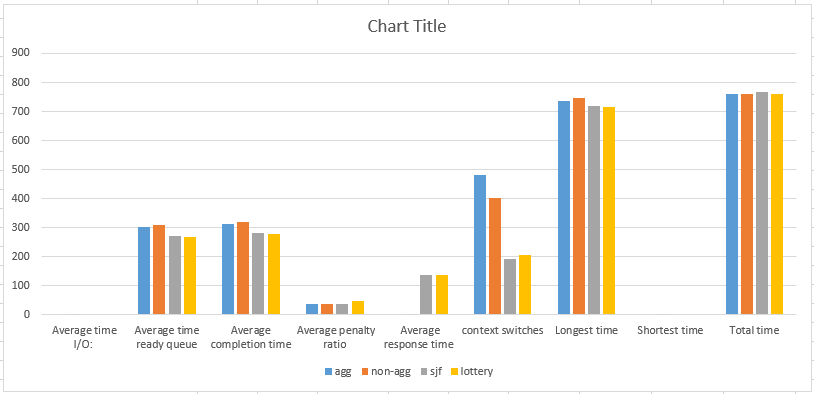
(because there is no lottery scheduler in the binary test, I can not compare)

**Comparesion and effciency**



In comparsion of above table , aggressive and non aggressive number of context switch is much higher than sjf and lottery, but much faster for the response time. Also they ar slighterly higher in ready queue and completion.

**Conclustion:**



In conclusion, we inplement fou different scheduler and the coordinator to compare their perfromance in different circumstance.

1. Aggressive is better than non aggressive but it require more context switches.
2. Sjf and lottery is faster than aggressitve and non aggressive has lower context switch but it reuqire more response time.
3. Aggreesive require the most context switches.
4. Non-aggressive has the longest system time for job.

(For intrustion, read “README.txt” file and it include an youtube demo video link in it.)