```
/*****************************************************************************
Authors : Manuella,Evan      , Box     , <manuella@grinnell.edu>
        : Albert Owusu-Asare , Box 4497, <owususasa@grinnell.edu>

Date    : Fri Sep 19 22:02:11 CDT 2014

This document contains answers to questions 1 - 9 of Part A of
the lab on Scheduling Simulation :

http://www.cs.grinnell.edu/~weinman/courses/CSC213/2014F/labs/scheduling.html

*****************************************************************************/
```

PartA: Understanding the code
---

4. The macros __EVENTQ_H and __SCHEDULER_H__ are used to make sure that the contents of "eventq.h" and "scheduler.h" are not redifined when they are encountered numerous times during the compilation and linking process. This prevents us from redefining anything that we've defined before.

5
a) the ready queue

Code in"scheduler_simulation.c" can change the ready queue structure because it can:
i) Initialise the queue
ii) add jobs to the queue,
iii)remove jobs from the queue

b) the event queue

Code in"scheduler_simulation.c" can change the event queue structure because it can:
i) it can initialise the queue
ii) delete an event from the queue
iii) add an event to the queue

c) the performance statistic structure

Code in "scheduler_simulation.c" can only initialise the structure it calls **stats_print**(**double** sim_time) however the call does not change the underlying structure.

6. Say flag -1 initially corresponds to the scheduler handler and flag -2 initially corresponds to the register_job handler,**if for** some reason later we decide to replace say register_job with some other handler, all we need is to change remap flag -2 to this new handler and map register_job with something **else**. Our code will still work as expected because its implementation is with the flag and not what lies at the other end of the flag

However flags are extra work and perhaps there is some overhead in the work done because, the code has to first try to preprocess what each flag means and then perhaps copy the underlying handler implementation whereever the flag was called. This is extra work and perhaps innefficient management of resources as we could have simply pointed to where in memory our particular handler lies instead of making copies everytime.

7. eventq_next_event_time returns the time field of the next event in our event queue.

When called in "scheduler_simulation.c" :
It is used to set the 'sim_time' (time **for** the simulation) by adding the **return** of 'event_next_event_time' to sim_time to have an idea of how **long** the simulation might run.

8. This simulation assumes a constant overhead per sheduler run(0.35). This might not neccessarily reflect the actual overhead and so might not accurately represent scheduler overhead. However we believe that using an approximation might give us an insight into how fast each scheduler is in relation to other schedulers- even **if** the outcome does not reflect reality.

9. Machine/system failures can have an effect on job completion time. System failures can be caused by many factores: **for** instance external such as power-shortages, overheating, and other factors such as hardware/software failures. We are not in direct control over some of these factors hence it is reasonable they were left out of our model. Even though we could try to fit such models in our simulation it might make our overall simulation model extra complex.

PART B: Non-preemptive scheduling

4.
a) Overall the average waiting times(response times) and turnaround times **for** the SJN algorithm is lower per each overhead time, than the average waiting time and turnaround times **for** the FCFS algorithm. The throughput of both the FCFS and SJN algorithm stayed relatively the same.

Conclusion:
When response time is critical to us in our design,(high I/O bound number of jobs), SJN might be a better choice. Withing a set time more jobs will be completed with the SJN.Hence overall it seems the SJN algorithm is the more effective among the two

b) Examining the data, we found no relative differences in the derivatives of response time versus overhead or turn around time versus overhead between sjn and fcf. We found no significant differences in the derivate of throughputs with respect to time (both almost 0).

PART C: Simple preemptive scheduling

4.

a)
average Wait time:

Overall the average wait time **for** the RR algorithm was lesser in magnitude per overhead than the average wait time **for** the SJN.

Based solely on response time, preemptive scheduling is more effective than non preemptive scheduling.

average Turnaround time:

Overall the average turnaround time **for** the RR algorithm was higher in magnitude per overhead than the average turnaround time **for** the SJN algorithm

Based solely on turnaround time, non preemptive scheduling is more effective than preemptive scheduling.

throughputs:
Throughputs **for** the RR algorithm were realatively lower than throughputs of the SJN algorithm.

Based solely on throughputs, non preemptive scheduling is more effective than preemptive scheduling.

b) The derivative of response time in respects to overhead stat increases

linearly in round robin. This is because as we have more jobs, the time between the same job being scheduled twice equals the number of
jobs * (overhead + quantom). In the SJN scheduling algorithm, overhead does not have the same amount of impact, because overhead only happens when we switch jobs, and SJN minimizes occurences of job switches.

There is a similiar effect on turn-around times.

Over head time takes a significanlty larger toll on round robin compared to SJN, because round robin switches jobs more often.

c. Quantom has a significant effect on round robin. As expected, it increased the response time, because it takes longer to finish a quantom, increasing the time between responses.

Increasing quantom also decreases the average turn-around time. This is because there is reduced **overhead** (from reduced context switches).

PART D

We could not work out the particulars of the implentation, but we wanted to explain our idea. Our idea was to have the code in insert_queue insert a job into a corresponding priority queue. There would be x, x being the number of priorities, number of priority queues that correspond to priorities. When queue_select is called, it would return the first job from the highest priority queue. Quantum would be set, and the rest of the jobs in this priority queue would run in round robin. Our implementation failed because we needed to move to queues in arrays, instead of a switch statement of priority queues.