# A8: Scheduling Algorithms

## *General Instructions:*

1- Create a C project in eclipse called A8. Download A8.zip file and extract the contents to the A8 project.
2- Rename: “A8_template.txt” to “A8.c”. Enter your credentials on top of the file.
3- Copy the following files into the A8 project:
    a. process.h
    b. process.c
    c. data.h
    d. data.c
    e. pqueue.h
    f. pqueue.c
4- Do not change any of the files other than the data.c and A8.c files.
5- Use the file “A8_test.c” to test your solution. Use the validator to validate your solution.
6- When you are done, you need to submit only your “A8.c” file. Do not export the project or upload a .zip file.

## *General Instructions*

In this assignment, you are allowed to access and update the data members of a process struct. For example, you can use p.time and p.arrival to check or update the process time or arrival.

## *Task 1: Dispatcher*

Change the `compare_data` function in the `data.c` file, such that processes are compared based on: arrival time, then service time then PID. (Note in the previous

assignment, compare_data was part of the pqueue implementation. You can copy the function to the data.c file and then edit it. )

In the A8.c file, implement the function:

```
void schedule(char *type, Data *processes, const int size);
```

The function receives a string represent a scheduler `type`. The function acts as a dispatcher, calling the proper scheduler based on the given type as follows:

- If the given type is “FIFO” the function calls FIFO scheduler.
- If the given type is “SJF” the function calls SJF scheduler
- If the given type is “RRx” the function calls the RR scheduler.
- If the given type is any other than the above three, the function prints an error message and returns.

The function also receives an array of processes along with its size. A priority queue is to be created and all elements of the array are to be inserted into the pqueue. It is one of your design decisions to pick the ‘H’ or ‘L’ priority modes, and to determine to use the sorted or unsorted insertion.

When creating the pqueue you can use any capacity ranging from size to size*2.

For the RR scheduler, an extra step is needed to extract the period of the Round Robin Scheduler. The period can be found in the trailer of the type string. For example, if the type is RR8, the period is 8 and if the type is RR12 the period is 12.

## *Task 2: FIFO Scheduler:*

Implement the function `void schedule_FIFO(pQueue *q)` which receives a priority queue containing processes. The function schedules the given processes using the FIFO policy.

The advantage of using the priority queue is that, using the updated version of the compare_data function, processes will be sorted based on their arrival time, regards of when they have been added to the pqueue.

The FIFO scheduler function prints a start and end statements declaring the start and closing of the scheduler.

The function also maintains a timer that starts at time 0 and ends when all processes in the queue have been processed. The timer is a simple integer that increments with every loop cycle.

At any specific instance, i.e. value of the timer equals x, the scheduler peeks into the queue to see if there are any processes that have arrived. This can be done by inspecting the arrival of the process at the front of the queue and comparing it to the timer. If no process has arrived yet, the scheduler is considered 'idle', i.e. not serving any process. The scheduler simply increments the timer when it is idle and performs no other action.

If there is a process awaiting to be served, then the scheduler fetch it and serve it depending on the value of ***process.time***. During this time, the timer will be incremented depending on how much time the process is being served.

The above is repeated until all processes in the queue have been served.

The scheduler prints a log of its status at each tick (increment) of the timer, in the following format:

```
[Timer:<timer_value>]: <process>
```

If there is no process being served, the log will print:

```
[Timer:<timer_value>]: idle
```

To fully understand this task inspect both: A8_output.txt and one of the processes.txt files, and observe how the scheduler is expected to execute.

## *Task 3: SJF Scheduling:*

In real-environment, processes are rarely served using the FIFO policy. There are many scheduling policies that use some sort of priority beyond the arrival time. Examples include: shortest job first (SJF), Longest job first (LJF), Longest Waiting Time (LWT), …etcs. In this task, you will inspect the SJF policy.

Implement the function `void schedule_SJF(pQueue *q)` which receives a list of Process structs within a priority queue and schedule them using the SJF policy. The overall structure of the scheduler is similar to the FIFO scheduler developed in Task 2, except that SJF is used instead of FIFO.

In SJF, the scheduler inspects all processes that have arrived and then picks the one with the shortest service time, i.e. lowest value of *process.time*. The main advantage of this policy is that short processes do not have to wait long int the queue awaiting the very long processes to finish.

Similar to Task 2, using a priority Queue to store your processes will sort them based on their arrival time. However, it is not enough to fetch the front of the queue. Instead you should inspect all processes that have arrived (you need to use peek/remove/insert methods not pqueue.array), and fetch the process with the smallest service time.

One solution to the above is to remove all processes that have arrived and store it in a temporary buffer/cache (which is simply a list). The scheduler picks the shortest job, fetch it for processing and insert the others back to the queue. The power of priority queue will be manifested here, as the queue will sort the queue again based on the arrival time.

## *Task 4: RR Scheduling:*

FIFO is a good policy in terms of fairness. However, if there is a long process ahead of several short processes, the overall average waiting time will be high. On the other hand, SJF has a great average waiting time but it has a problem which we call starvation. Starvation happen when short processes continually come to the queue and serving the long processes keep to be deferred. This can be indefinite making long processes “starve”.

A scheduling policy that balances fairness, good average waiting time and does not have starvation is called Round Robin.

Round Robin means that you allocate a fixed specific time (called period) and serve all processes available in the queue, one by one, but only a service time equal to period. If a process is not done been served, then it is pushed to the back of the queue.

Let us take an example:

Assume the queue has the following three processes:

```
([1](1000010000,7),[3](2000020000,6),[7](3000030000,5))
```

Let the Round Robin period be 4. Then scheduling will work as follows:

At timer [0]:
the scheduler starts, and since there are no processes that have arrived, the scheduler stays idle.

At timer [1]:
the scheduler sees that a process has arrived, and fetch it. The scheduler only serve it an amount equal to 4. Therefore, when the timer was at [1],[2],[3] and [4], it was serving Process 1000010000.
However, since there are three more time slots required to serve it, it update the process to become: [5](1000010000,3) and push it to the back of the queue.
Notice how the arrival time is now changed to 5, because that is the time it is been added to the queue. Also, notice how the service time has changed from 7 to 3.
After this step, the contents of the queue will become:

```
([3](2000020000,6), ([5](1000010000,3),[7](3000030000,5))
```

Notice how the process was added ahead of Process 3000030000, because it has an earlier arrival time.

At timer [5]:

the scheduler peeks and fetches Process 200020000 because it had arrive at [3]. Again it serve it for a time slot equal to 4. It updates the Process to [9](20002000,2) and push it to the back of the queue. The queue becomes:

```
([5](1000010000,3), [7](3000030000,5), [9](2000020000,2))
```

At timer [9]:

the scheduler peeks and fetches Process 100010000. This is the second time, this process has been served. It grants the process a time slot of 4, but since the process only requires 3 slots, it only serve it for 3 slots and remove it from the queue. The queue becomes:

```
([7](3000030000,5), [9](2000020000,2))
```

At timer [12]:

the scheduler peeks and fetches Process 300030000. It grants the process a time slot of 4, update to [16](3000030000,4) and push back to the queue, which becomes:

```
([9](2000020000,2), [16](3000030000,1))
```

At timer [16]:

the scheduler peeks and fetches Process 200020000. It serves the process for 2 time slots and remove it. The queue becomes.

```
([16](3000030000,1))
```

At timer [18]:

the scheduler peeks and fetches Process 300030000. It serves the process for 1 time slot and remove the process. The queue is now empty.

At timer [19]:

The scheduler is closed, because there are no more items in the queue.

Implement the function `void schedule_RR(pQueue *q, int period)`. The first parameter is a priority queue containing processes, and the second is an integer representing the Round Robin period.