# ROUND ROBIN


## BACKGROUND

Whenever there is a single resource that needs to be used (or shared) by many individuals, a scheduler is used to determine who gets to use the resource and for how long. This exercise asks you to write a round-robin scheduler to assign processes to a CPU and determine when each process is able to finish.

## ROUND ROBIN SCHEDULING

Consider living in an ancient near-east city such as Babylon or Jerusalem. It was common to have an entire city share access to a single well from which water was drawn. Imagine that at various times, individuals would bring some container (each of varying sizes) and seek to fill their container with water from the single well (the shared resource). Imagine that the city hired a well-manager to control which of the potentially numerous individuals that came for water received access to the well and for how long.

The well-manager could choose many strategies; perhaps giving the most politically powerful person preference above all others such that the most powerful person filled their container entirely before the next-most-important person was allowed access. Our round-robin well-manager, however, adopts a more equitable strategy of sharing. The round-robin manager ensures that there is a single line at all times. Each individual desiring access to the well must first be added to the end of the line. The manager also has a stop-watch that counts, or ticks, once every so often. The well-manager allows the person at the front of the queue to have ALLOTMENT ticks of time at the well. If the individual's container is completely filled within those ALLOTMENT ticks, the individual returns home as soon as their container is full. If the individual's container is not filled within those ALLOTMENT ticks, that individual is sent back to the end of the line to await another turn. Each individual receives the same ALLOTMENT of ticks.

The basic idea is simple, however some details are relevant. Consider, for example, the case of an individual who is the first to get in line (the line is of size 1). That individual receives ALLOTMENT ticks but their container is not full (they have not completed their task). Since nobody else is in line, the manager allows that user to continue to use the well. However, within a short time another individual gets in line. The manager will immediately send the first user (since they have been using the well for consecutive ticks that exceed ALLOTMENT) to the end of the line and allow the newcomer access to the well.

## DETAILS

For this assignment, you will write a Queue class (the line), a Scheduler(the round-robin manager), and a Process (an individual needing access to the shared resource for some number of time units). Your code will then be used to compute how long a collection of Processes must take for them all to complete their tasks.

### QUEUE

You must implement the Queue<E> interface shown below. Your class must be a generic class named LinkedQueue. The LinkedQueue class must have exactly two attributes: a doubly-linked-list node named "sentinel" and an int named "size". No other attributes are allowed. ***Each method must run in constant-time***!

```
public interface Queue<E> {
        public boolean enqueue(E e);
        public E dequeue() throws NoSuchElementException;
        public int size();
        public boolean isEmpty();
        public E peek() throws NoSuchElementException;
}
```

## SCHEDULABLE AND PROCESS

A Schedulable object is any object that can be scheduled. You must write a class named Process that implements the Schedulable interface shown below.

```
public interface Schedulable {
        public void takeTurn( int atTime );
        public boolean isCompleted();
        public int getTurnsTaken();
        public int getTicksRequiredToComplete();
        public void setStartOfTimeInScheduler( int time );
        public int getStartOfTimeInScheduler( );
        public int getTimeToComplete();
        public int getTimeCompleted();
}
```

Method specifications

- Process( int ticksRequiredToComplete ) : constructs a Process with the specified number of ticks required before it completes. The ticks-required-to-complete is analogous to the size of the water-container in the well analogy.
- takeTurn( int atTime ) : the process gets a single tick of time with the shared resource. The tick of time occurred at the global-time atTime.
- setStartOfTimeInScheduler( int time ) : time is the global-time at which the Process got into the end of the line.
- getTimeToComplete() : returns the number of ticks between the time-completed and the start-of-time-in-scheduler.
- getTimeCompleted() : returns the global-time at which the Process finally completed.

## SCHEDULER

A Scheduler is able to control access to a single shared resource such that only objects of Schedulable type are able to gain access to the shared resource. You must implement the Scheduler interface shown below using a round-robin scheduling strategy. Your implementation must be named RoundRobinScheduler. The scheduler keeps track of global time by 'ticking' whenever the 'tick' method is called. Each clock 'tick' represents one time-unit such that one Schedulable object (the one who's turn it is, if any) will have received one 'tick' of time with the shared resource. The global time of the RoundRobinScheduler will be zero when initially constructed.

```
public interface Scheduler {
        public void add( Schedulable s );
        public Schedulable tick( );
```

```
        public int getTime( );
        public int size();
        public int getAllotment();
        public void setAllotment( int allotment );
}
```

Method specifications

- add( Schedulable s ): element s is added to the scheduler. It will be given access to the shared resource in accordance with the schedulers strategy.
- getTime() : returns the time in terms of 'ticks' executed by the scheduler.
- size() : returns the number of Schedulable elements that are being managed by the Scheduler but are not yet complete
- setAllotment( int allotment ) : sets the allotment.
- getAllotment() : returns the allotment
- E tick() : advances the global time clock by one tick. Returns the element the completed at the conclusion of this tick. If no element completed, this method returns null.

## SIMULATION

You must also write a program named Simulation that uses the scheduler to determine the completion time of a collection of processes, each of which has different start times and different ticks-required-to-complete. Your program must read an input file (given as the only command-line argument) and print the results to the terminal window. The input file format is described below.

### *INPUT FORMAT*

The input is a text file. The first line contains a non-negative integer denoting the ALLOTMENT. Each subsequent line contains information related to a single process. Each line conforms to the following format.

**<process-name> <ticks-required-to-complete> <global-start-of-time-in-scheduler>**

### *OUTPUT FORMAT*

Print each process of the input file on a single line in the order in which the processes complete. Each line must conform to the following format.

**<process-name> <global-start-of-time-in-scheduler> <global-end-of-time> <ticks-required-to-complete> <time-to-complete>**