Technology Fundamentals - Assignment 2

CPU Scheduler / Sorting Algorithms

Peter O'Reilly D12123601

Part A)

Use your own Heapsort implementation in Java to sort and print the following job-queue from lowest to highest priority job ascending.

{ TIME-CRITICAL(100), NORMAL(50), BACKGROUND(1000), BACKGROUND(900), NORMAL(100), TIME-CRITICAL(1000), TIME-CRITICAL(1), NORMAL(1000), TIME-CRITICAL(1) }

Should sort to

{ BACKGROUND(1000), BACKGROUND(900), NORMAL(1000), NORMAL(100), NORMAL(50), TIME-CRITICAL(1000),TIME-CRITICAL(100), TIME-CRITICAL(1), TIME-CRITICAL(1) }

A demonstration of this implementation is visible as a unit test, *Test 20*, in the *ie.dit.scheduler.priorityQueueTest* package in the test directory.

(Note: Process times are set using a setTimeStamp() method. In the implementation of test 20 the return value is the process name and its process runtime. However it is NOT sorting on process runtime, but as required on i) process class, then ii) process timestamp values. The time stamps are being set in the test in the required order that would simulate the above initial list of jobs and their respective orders of class and timestamp values.)

Part B)

Extend your max-heap from a) to implement a max-priority queue based on the priorities of a process

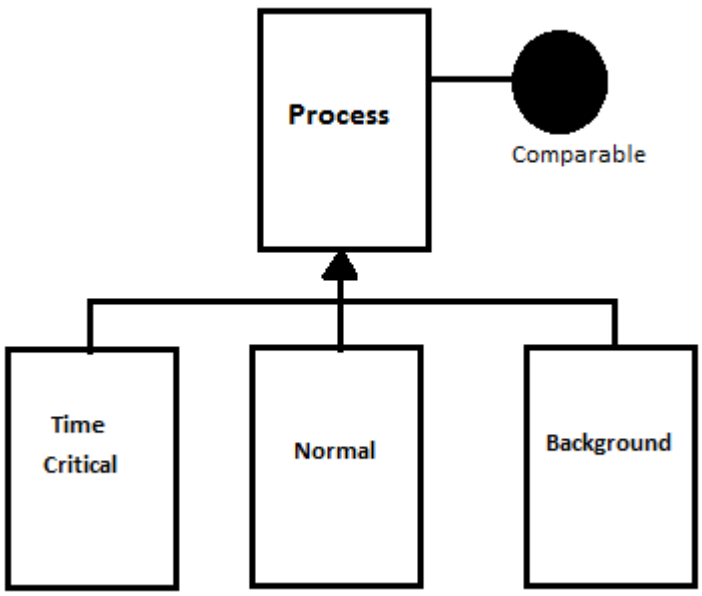
I. *Create the class Process which implements java.lang.Comparable<Process> which encapsulating relative priority.*

II. *Create the class PriorityQueue which implements the Queue contract on page 3 as a priority queue based on the max-heap process.*

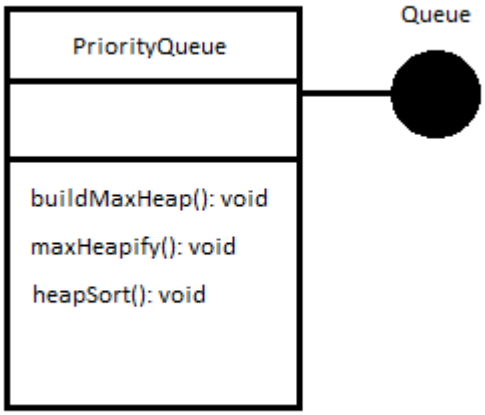
i) The class Process (implementing the Comparable Interface) is implemented in the **src** folder package name***, ie.dit.scheduler.priorityQueue***.

> ***Test number 24* in *ie.dit.scheduler.priorityQueueTest,***
> **illustrates the implementation of the compateTo() method using the Arrays.sort(someArray) method.**

The process class is used as a super class in this application. Implementing the Comparable interface and all different process classes i.e. Background, Normal, Time-Critical extend this parent class. In each sub classes constructor on instantiation defines its process class in a call to super.



ii) PriorityQueue class is in the **src** folder, package, ***ie.dit.scheduler.priorityQueue*** , which implements the Queue interface. There is significant test coverage in the, ***dit.scheduler.priorityQueueTest***. The PriorityQueue implements the Max-Heap process to ensure classes of highest priority are at the first index of the PriorityQueue's Process[]. Only Arrays were used in the implementation of this class.



PriorityQueue

Part C)

Further implement

i) A producer that can create y jobs of random running time between 0 and 60 seconds and of random class priority.

ii) A ready queue

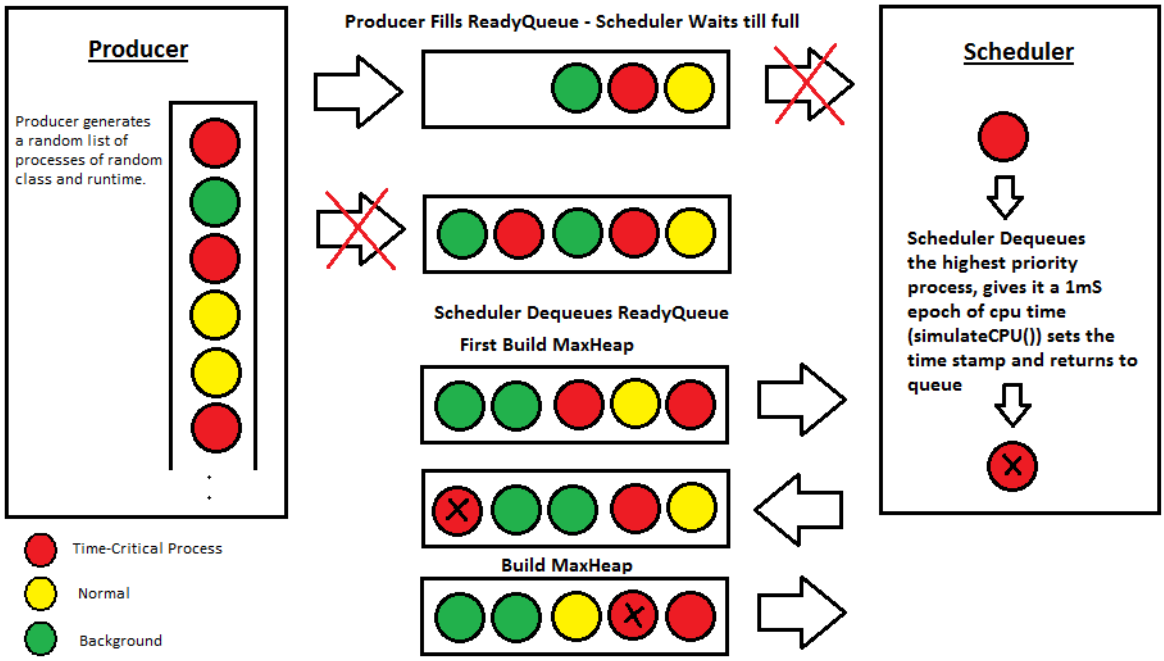
iii) A scheduler (consumer) that processes jobs in class-first, then LRU order until they are complete and uses a 0.1 second epoch for time-based pre-emptive scheduling.

i) The Producer class is implemented in the package, ***ie.dit.scheduler.producerConsumer***. It will generate a random number of processes or different classes and of different runtime.

This is illustrated by running the JUnit test class *ProducerTest.java* in the *ie.dit.scheduler.priorirtyQueueTest* package which will on every run will print to console a random number of processes of different class and runtime (displayed in milliseconds).

ii) The ReadyQueue is in the **i*e.dit.scheduler.producerConsumer*** package. The ReadyQueue extends the PriorityQueue and is used as a buffer for the producer – consumer i.e. Producer class and the Scheduler Class.

iii) The Scheduler class exists in the same package as the above two classes. A test class (QueingTest.java in the ***ie.dit.scheduler.producerConsumerTest*** package) illustrates the process as well As Runner.java which contains the main method. See below for instructions.



Process Outline:

The above process has been implemented in the following manner. Firstly the Producer will produce a random number of processes (in this application between 10-30 processes) of random class and runtime. The method to generate these processes is called in the Producer's constructor.

Once generated and the Producer and Scheduler Threads are started, the producer enqueues the ReadyQueue. If the ReadyQueue is not full the scheduler waits. Once filled the Scheduler is notified where it begins the process of dequeueing the ReadyQueue. Before the process is dequeued a buildMaxHeap method is called to ensure that the process at the head of the queue is of the highest priority.

When the Scheduler dequeues the Process it sets the Process' timestamp and decrements the process runtime by one mS before returning it to the readyQueue. It is arbitrarily placed back in the readyQueue as its relative position is not important as everytime the head() method is called on the queue it ensures that the highest priority processes is at the first index.

This process is continued until all processes are terminated and the ReadyQueue is empty upon which condition the Producer is notified and the remainder of Ready Processes are placed on the readyQueue.

(Note: This implementation of the Producer – Consumer only notifies the Producer to begin producing once the queue is empty. The Consumer is only notified it i) the ReadyQueue is full or, ii) if the Queue is not full and the Producer is finished.)

Instructions for Part C

There are two demonstrations of the implementation of part C. One as a unit test in the ***QueingTest.java*** class (***ie.dit.scheduler.producerConsumerTest*** package) which statically creates 12 "random" processes and runs the application

OR

In the Runner.java file is the main method which also runs the application using the randomly generated number of processes.

The running application firstly prints to console the processes that have been created, their class (name) and process runtime. Once the Scheduler starts removing the processes and simulating the CPU burst, it will print to console when it (the Scheduler) is enqueuing the process back to the ReadyQueue.

Because of how quickly the processes are processed the number of randomly generated processes is not easy to identify, for that purpose there is a method which will allow a frame to be displayed with the total number of processes in the main method.

Because of this it is recommended that to illustrate that the application will not end in a deadlock situation that ***QueingTest.java*** class (***ie.dit.scheduler.producerConsumerTest*** package) be run to illustrate that the application will not end in a Thread waiting.

The **Runner.java** in **ie.dit.scheduler.producerConsumer** will illustrate process starvation by only displaying processes that are receiving simulated cpu time. This can be illustrated be comparing the processes that are visible in the frame versus those that are being displayed on the console.