|  |
| --- |
|  |
| Java Implementation Of RRScheduler |
| 3BB4 Assignment 4 |
|  |
| **Jenell Hogg, Student no 1226606** |
| **Hyuna Loy, Student no 1214931** |

[Module: RRScheduler 3](#_Toc415479619)

[FSP Translation 3](#_Toc415479620)

[Design Documentation 3](#_Toc415479621)

[Extends: Thread 3](#_Toc415479622)

[Field Variables: 3](#_Toc415479623)

[INTERFACE: 3](#_Toc415479624)

[Module: Generator 4](#_Toc415479625)

[FSP Translation 4](#_Toc415479626)

[COMPARISONS 4](#_Toc415479627)

[DESIGN DOCUMENTATION 4](#_Toc415479628)

[FIeld Variables: 4](#_Toc415479629)

[INTERFACE: 5](#_Toc415479630)

[IMPLEMENTATION: 5](#_Toc415479631)

[Module: ReadyQueue 6](#_Toc415479632)

[FSP Translation 6](#_Toc415479633)

[COMPARISONS 6](#_Toc415479634)

[Design Documentation 7](#_Toc415479635)

[Field Variables: 7](#_Toc415479636)

[INTERFACE: 7](#_Toc415479637)

[IMPLEMENTATION: 7](#_Toc415479638)

[Module: GrimReaper 9](#_Toc415479639)

[FSP Translation 9](#_Toc415479640)

[DESIGN DOCUMENTATION 9](#_Toc415479641)

[Field Variables: 9](#_Toc415479642)

[INTERFACE: 9](#_Toc415479643)

[IMPLEMENTATION 10](#_Toc415479644)

[Module: Dispatcher 11](#_Toc415479645)

[FSP Translation 11](#_Toc415479646)

[COMPARISONS 11](#_Toc415479647)

[DESIGN DOCUMENTATION 11](#_Toc415479648)

[Field Variables 11](#_Toc415479649)

[INTERFACE: 11](#_Toc415479650)

[IMPLEMENTATION: 12](#_Toc415479651)

[Module: CPU 13](#_Toc415479652)

[FSP TRANSLATION 13](#_Toc415479653)

[COMPARISONS 13](#_Toc415479654)

[DESIGN DOCUMENTATION 13](#_Toc415479655)

[Field Variables: 13](#_Toc415479656)

[INTERFACE: 13](#_Toc415479657)

[IMPLEMENTATION: 14](#_Toc415479658)

[Module: Process 15](#_Toc415479659)

[FSP Translation 15](#_Toc415479660)

[DESIGN DOCUMETATION 15](#_Toc415479661)

[Field Variables: 15](#_Toc415479662)

[INTERFACE: 15](#_Toc415479663)

[IMPLEMENTATION: 16](#_Toc415479664)

|  |
| --- |
|  |

# Module: RRScheduler

## FSP Translation

This class is the implementation of the RR\_SCHEDULER process. In the FSP model, RR\_Scheduler is the concurrent execution of the processees QUEUE, GENERATOR, GRIMREAPER, DISPATCHER, and CPU.

The RRScheduler is where the Dispatcher, ReadyQueue, and Generator objects are initialized. (Representing QUEUE, MONITOR, and DISPATCHER, respectively)

The ReadyQueue object is initialized and stored into a static reference so it may act as a monitor for the Dispatcher, Generator, and GrimReaper. (Please see ReadyQueue Module for more details)

The Dispatcher and Generator are ran with Java's start() method, so they execute concurrently.

It should be noted the CPU process shares the action select with the DISPATCHER and the GRIMREAPER shares the action checkTime with the CPU. The result of these shared actions in the model is that the Dispatcher thread is responsible for starting the CPU thread and the CPU thread is responsible for starting the GrimReaper thread. Therefore, there is no need to start the Dispatcher and CPU threads in the RRScheduler. This implementation stays true to our model as it represents how the concurrent execution of the processes in behave.

## Design Documentation

### Extends: Thread

### Field Variables:

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Meaning |
| readyQ | ReadyQueue | A static reference to the ReadyQueue. |
| messageTime | int | The time threads should wait before continuing after printing a message (This is so the output generated is more friendly to the human observer- so it doesn't run too fast) |

### INTERFACE:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| main | String args[] | - | The main method of the project.  Initializes the ReadyQueue, Generator and Dispatcher.  Starts the Generator and Dispatcher threads so they execute concurrently. | readyQueue=new ReadyQueue();  messageTime=500; | Generator.Generator  Dispatcher.Dispatcher  Thread.join() |

# Module: Generator

## FSP Translation

The Generator class is the implementation of the GENERATOR process. The Generator is responsible for adding threads to the queue. It waits until the queue is not full and adds threads to the queue. Just as in the FSP model, the Generator depends on the ReadyQueue monitor to regulate when it is able to add things into the threadQueue. (Notice how the GENERATOR process shares the state variable waiting with QUEUE as well as the actions add\_Q and remove\_Q.

The Generator thread will be blocked by the ReadyQueue class when it attempts to enqueue a process when the threadQueue is full. Please see the ReadyQueue class for more information.

### COMPARISONS

|  |  |  |
| --- | --- | --- |
| ACTIONS USED IN LTSA MODEL | METHODS USED IN JAVA IMPLEMENTATION | COMPARISON |
| add\_Q | enqueue() | Both add\_Q and enqueue() are shared by the QUEUE/ReadyQueue which holds the threads waiting to be executed. |
| remove\_Q | none | This action in the model was used to keep track of the number of threads in the QUEUE. It is represented by the ReadyQueue class which acts as a monitor to decide when a process may be added. |

## DESIGN DOCUMENTATION

Extends: Thread

### FIeld Variables:

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Meaning |
| processCount | private int | represents how many processes have been generated. This variable is assigned as the ID for new processed generated. |

### INTERFACE:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| Generator | - | - | Constructor | INITIAL STATE:  processCount=0 | checkQueue() |
| run() | - | - | Overrided run method from Thread  Continually creates new processes and attempts to add them to the global  ReadyQueue object generated by the RRScheduler class.  The ReadyQueue monitor is responsible for regulating when the thread should be blocked and when it may proceed.  This run() method executes indefinitely. |  | this.enqueue() |

### IMPLEMENTATION:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| enqueue() | void | void | Creates a new Process and adds it to the readyQueue | generated=new Process(processCount)  RRScheduler.readyQ.enqueue(generated)  processCount++ | Process.Process (Constructor)  this.generateMessage()  RRScheduler.readyQ.enqueue(); |
| generateMessage() |  | none | Generates a message of the form  "Process # loaded into ready Queue) |  |  |

# Module: ReadyQueue

## FSP Translation

### COMPARISONS

|  |  |  |
| --- | --- | --- |
| ACTIONS USED IN LTSA MODEL | METHODS USED IN JAVA IMPLEMENTATION | COMPARISON |
| select | select() | Both actions/methods indicate the selection of a thread for execution. Both methods are shared between the ReadyQueue/QUEUE and the Dispatcher/DISPATCHER class/process. |
| add\_Q | enqueue(Process p) | Both actions/methods indicate the addition of a thread to the queue. Both actions/methods are dependant on whether the queue is full. |
| remove\_Q | dequeue() | Both actions/methods indicate the removal of a thread from the queue.  Both actions/methods are shared in the Generator/GENERATOR, and GrimReaper/GRIMREAPER class/process. In the QUEUE process this action is one of the possible actions as a result of WHATNOW. The java implementation does not require a WHATNOW process to decide to the methods being joint to the GrimReaper class. The java implementation through a checkTime() method in the GrimReaper class which will decide the result. |
| backInQueue | backInQueue(Process p) | Both actions/methods indicate the addition of a previously terminated thread back into the Queue. Both actions/methods are shared in the GrimReadper/GRIMREAPER class/process. The backInQueue action is chosen as a result of the checkTime action in the GRIMREAPER. Those actions are represented in the implementation through the method checkTime() in the GrimReaper class. |

The ReadyQueue class is the implementation of the QUEUE process found in the model.

The ReadyQueue class is a monitor. The QUEUE process allowed execution of certain actions based upon its state variable waiting, which represents the number of Processes currently in the queue its modeling. The ReadyQueue class also has a state variable waiting- an integer that represents how many non-null entries there are in the threadQueue (An array of Process Objects).

The ReadyQueue class methods are accessed by the Generator, Dispatcher, and GrimReaper classes- this reflects how in the FSP model the actions in the alphabet of the QUEUE process are shared with the DISPATCHER, GENERATOR, and GRIMREAPER processes.

The ReadyQueue will block the Dispatcher thread if the threadQueue is empty (Just like how the QUEUE process cannot engage in the select action if waiting==0)

The ReadyQueue will block the Generator if the threadQueue is full or if a process is loaded onto the CPU (Our model does not allow the Generator to engage in the add\_Q action if the select action has been executed but the backInQueue or remove\_Q actions have not been reached yet)

The ReadyQueue will allow the GrimReaper thread to remove a Process from the threadQueue or move a Process to the back of the queue. After doing either of these actions, the ReadyQueue will notify any threads waiting on it, allowing for another state transition. If the GrimReaper decided to dequeue(), waiting is decremented. (Just as our QUEUE process engages in another QUEUE process with updated state variables)

## Design Documentation

### Field Variables:

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Meaning |
| waiting | int | The number of threads waiting in the Queue |
| threadQueue | Process[5] | The queue of processes waiting to be executed. Our queue is arbitrarily set to a maximum size of 5. |
| isLoaded | boolean | A boolean representing whether or not a process should be added to the queue. Corresponds to whether a process has been selected. |

### INTERFACE:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| ReadyQueue | void | ReadyQueue | Constructor | INITIAL STATE:  waiting=0  threadQueue=new Process[5] |  |
| enqueue(Process p) | A Process representing the new thread to be put into the threadQueue | void | Waits for the thread to have space and for the CPU to be clear(isLoaded==false), then adds the given process to the threadQueue | waiting++  threadQueue[lastEmptyIndex]=p | this.isFull()  this.findEmptyIndex()  notify()  wait() |
| dequeue() | void | void | Takes a thread out of the queue.  Notifies any thread waiting on this class's monitor | waiting--  for(i=1,i<lastNonEmptyIndex;i++){  threadQueue[i-1]=threadQueue[i]  threadQueue[lastNonEmptyIndex]=null  isLoaded=false | findNonEmptyIndex()  wait()  notifyAll() |
| backInQueue(Process p) | A Process p that represents the modified thread at the front of the queue  (This thread was taken by the dispatcher and is now being returned by the Grim Reaper) | void | Puts the first item in the threadQueue in the last available spot, then notifies any thread that's waiting. | placeHolder=threadQueue[0]  for(i=1<, i<lastNonEmptyIndex;i++){  threadQueue[i-1]=threadQueue[i]  threadQueue[lastNonEmptyIndex]=placeHolder  isLoaded=false; | findNonEmptyIndex  notifyAll() |
| select() | void | The process at the head of the threadQueue | Waits until the thread is not empty, then returns the thread at the head of the Queue | isLoaded=true; | isEmpty()  wait() |

### IMPLEMENTATION:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| findEmptyIndex | void | int | Finds the index of the first null element in the threadQueue and returns it | - | - |
| findNonEmptyIndex | void | int | Finds the index the the last element in the threadQueue | - | - |
| isFull() |  | boolean | Returns a boolean representing whether the queue is full or not | - | - |
| isEmpty() | void | boolean | Returns a boolean representing whether the thread is empty or not | - | - |

# Module: GrimReaper

## FSP Translation

The GrimReaper class implements the termination or re-addition of a thread into the queue. The CPU calls the GrimReaper in order to determine the fate of a given thread. It then calls the ReadyQueue class to update the queue.

In the model, checkTime represents the decision of whether to terminate a process or add it back into the queue. The actions in WHATNOWGR, remove\_Q and backInQueue, represent those decisions and are present in the QUEUE. The implementation adheres to the model with checkTime() representing checkTime->WHATNOW, and the shared actions remove\_Q and backInQueue replaced by dequeue() and backInQueue() respectively. The methods dequeue() and backInqueue() call the ReadyQueue’s methods dequeue(currentThread) and backInQueue(currentThread).

#### COMPARISONS

|  |  |  |
| --- | --- | --- |
| ACTIONS USED IN LTSA MODEL | METHODS USED IN JAVA IMPLEMENTATION | COMPARISON |
| checkTime | checkTime() | Both methods/actions represent a decision of whether the process should be terminated or put back in the queue. In the model, this process is guarded by the actions execute in the CPU. This is represented in the implementation through the run() method which calls the execute() method before unload() (unload() calls the reaper which evaluates the thread and decides whether it should be put back or terminated. |
| remove\_Q | dequeue() | Both methods/actions represent the termination of a process. As a result of checkTime/checkTime() action/method being called, both may call remove\_Q/dequeue() which are shared between the QUEUE/ReadyQueue. |
| backInQueue | backInQueue(currentThread) | Both methods/actions represent the re-addition of a process back into the queue. As a result of checkTime/checkTime() action/method being called, both may call backInQueue/backInQueue() which are shared between the QUEUE/ReadyQueue. |

## DESIGN DOCUMENTATION

Extends: Thread

### Field Variables:

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Meaning |
| currentThread | Process | Current thread to have its fate decided |

### INTERFACE:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| GrimReaper() | - | GrimReaper | Constructor | INITIAL STATE:  currentThread=null; |  |
| run() | - | void | Overrided method from Thread  The GrimReaper decides whether or not to send the currentThread back into the queue or to kill it.  The GrimReaper is not blocked by any threads once it is ran. It access the ReadyQueue monitor either through its dequeue or backInQueue method, which may notify other threads waiting on the ReadyQueue monitor. | See checkTime | this.checkTime() |
| give(Process p) | A process that's to be judged by the GrimReaper when it runs. | void | Setter | currentThread=p |  |

### IMPLEMENTATION

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| checkTime() | - | void |  | int timeLeft=currentThread.getExecutionTime()  if(timeLeft>0){  backInQueue(currentThread)  else{  dequeue();  currentThread=null;  } |  |
| Overloaded methods:  generateMessage(id,x)  generateMessage(id) | int id  int x  x represents how many seconds a process with id number id has executed  id represents a processID | void | Generates a message of the form either  "Process id finished executing"  "Process id recycled into ready queue with x seconds remaining" |  | Generates a message and prints it  Integer.toString() |

# Module: Dispatcher

## FSP Translation

The Dispatcher thread is the implementation of the DISPATCHER process. The Dispatcher selects Processes from the queue in the ReadyQueue class via the method select(). (Please see the ReadyQueue documentation for more information on how the Dispatcher thread is monitored)

The Dispatcher thread will be called to wait() on the ReadyQueue monitor if the queue is empty. This represents the guarded action in the QUEUE process (waiting==0) that is guarding the action select that is shared between the DISPATCHER and QUEUE.

The DISPATCHER process shares the action select with the CPU process which guards the execute action. In the implementation, this is represented by the CPU class only being called to run in the Dispatcher after the Dispatcher has obtained a selected thread from the ReadyQueue class. After starting the CPU thread, the Dispatcher joins the thread via Java's join() method. This effectively implements the guarded action execute as the select() process in Dispatcher will always be called before the CPU's execute() method.

### COMPARISONS

|  |  |  |
| --- | --- | --- |
| ACTIONS USED IN LTSA MODEL | METHODS USED IN JAVA IMPLEMENTATION | COMPARISON |
| select | select() | Both methods/actions represent the dispatcher selecting a process for execution. In the model, the DISPATCHER guards the execute action for the CPU. In the implementation, the Dispatcher’s select() method calls the CPU class which corresponds with the order of the model. The DISPATCHER cannot select a thread when the QUEUE is empty. This guarded action is represented by the select() function in the ReadyQueue which is called in the DISPATCHER to confirm whether the queue is empty or not. |
| execute |  | The select() method calls the CPU class for execution. This corresponds well since the CPU process in the model is responsible for execution of processes and execute is a shared action between them. |

## DESIGN DOCUMENTATION

### Field Variables

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Meaning |
| processSelected | Process | Represents the process that is to be sent to the CPU |
| cpu | CPU | Represents a CPU process |

### INTERFACE:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| Dispatcher |  | Dispatcher | Constructor | INITIAL STATE:  processSelected=null |  |
| run | void | void | Overrided method from Thread class  The dispatcher class continually tries to select a Process from the static ReadyQueue object generated by the RRScheduler class.  The Dispatcher's progress is regulated by the ReadyQueue object. (It is blocked and notified by said monitor on when it is appropriate to select a Process)  After it selects a Process, the Dispatcher allows the CPU process  The Dispatcher process then gives the selected Process to the CPU thread and starts it- then it waits to finish by using the  The dispatcher generates a message each time processSelected changes. | See select | this.select()  Thread.join() |

### IMPLEMENTATION:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| select | void | void | Takes a Process from the RRScheduler.readyQueue  Generate a message indicating a Process was selected | processSelected=RRScheduler.readyQ.select()  cpu=new CPU(5000)  cpu.load(processSelected)  cpu.start()  cpu.join() | ReadyQueue.select()  CPU.CPU  CPU.load()  Thread.start()  Thread.join() |
| generateMessage() |  | none | Generates a message of the form  "Process # loaded into CPU) |  |  |

# Module: CPU

FSP TRANSLATION:

The CPU class is the implementation of the CPU process. The CPU process in our model shares the action select with the process dispatcher- this is to represent how the action execute is blocked by the action select in the DISPATCHER process. In the implementation, the CPU thread is called to run only by the Dispatcher, ensuring the method execute() is guarded by the method select() in the Dispatcher class.

The CPU process also shares the action checkTime with the GRIMREAPER process. This is to represent how the action execute guards the checkTime action (That is, a process must be executed before the GrimReaper can do anything with it). The GrimReaper class is called to run only within the CPU thread, after which the CPU thread joins with the GrimReaper (Using the join() method after starting it to wait until it finishes executing) This implementation represents the guarded actions of execute->checkTime in our model.

### COMPARISONS

|  |  |  |
| --- | --- | --- |
| ACTIONS USED IN LTSA MODEL | METHODS USED IN JAVAIMPLEMENTATION | COMPARISON |
| execute | execute() | In the model, execute represents the CPU executing the process. This is represented in the implementation with the execute() method which runs the process for the required time. |
| checkTime |  | In the model, checkTime was used as a shared action for the CPU and the GRIMREAPER as a guarded action. In the implementation, the CPU class communicates with the GrimReaper class via the unload method which sends a thread that has finished executing to the GrimReaper. The GrimReaper then executes a checkTime method. In the run() method execute() is called before unload(). In this way, the order the actions take place is preserved. (execute->checkTime ) |

## DESIGN DOCUMENTATION

Extends: Thread

### Field Variables:

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Meaning |
| maxExecutionTime | int long | represents the max executionTime a Process may run |
| reaper | GrimReaper | represents GrimReaper process |
| loadedThread | Process | The current thread on the CPU |

### INTERFACE:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| CPU | (int long maxExecution, GrimReaper reaper) |  |  | maxExecutionTime=maxExectution  loadedThread=null; |  |
| run | void | void | Runs the loadedThread  Updates the amount of time left on the thread  Sends the loadedThread to the GrimReaper, then finishes executing (And thereby "freeing" the Dispatcher) | See execute, unload | this.execute()  this.unload() |
| load | Process p | void |  | loadedThread=p; |  |

### IMPLEMENTATION:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| unload() | Represents a Process being unloaded from the CPU |  | Passes the Process object to the GrimReaper. Starts the GrimReaper, then waits for it to execute. | reaper=new Reaper()  reaper.give(loadedThread)  reaper.start()  reaper.join() | GrimReaper.give()  GrimReaper.GrimReaper  Thread.start()  Thread.join()  this.generateMessage |
| execute() |  |  | Runs the Process loaded into the CPU and then updates the remaining execution time in said process | loadedThread.updateTime(maxExecutionTime)  loadedThread.run() | Process.updateTime()  Process.run() |
| generateMessage() |  | none | Generates a message of the form  "Process # executed for x seconds" |  |  |

# Module: Process

Extends: Thread

## FSP Translation

Process holds general information about the threads such as the ID, amount of time needed for execution, and how much time it has run for. It is used in the other classes to represent the current thread in use.

## DESIGN DOCUMETATION

### Field Variables:

|  |  |  |
| --- | --- | --- |
| Field Name | Type | Meaning |
| remainingExecutionTime | int | Represents the remaining execution time this process has left to execute before finishing. |
| id | int | Represents the id number of the Process |
| runTime | int | Represents the execution time the process has been given by the CPU during its next execution. |

### INTERFACE:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| run() | void | void | Overridden method from Thread  The thread sleeps for the given runTime- representing"work" done by the thread. |  | sleep() |
| Process(int id) | An integer to be assigned as the id number of the Process | Process | Constructor  Assigns a random number between 1000 and 7000 as the execution time for the Process | this.id=id  executionTime=randInt(1000,7000) | this.randInt() |
| updateTime(int Long l) | A long integer representing how long the Process ran for (CPU maxRunTime) | integer Long  how long the thread ran |  | int newTime=executionTime-l  int r = executionTime;  if (newTime>0){  executionTime=newTime;  return newTime  }  else{  executionTime=0;  return r;  } |  |
| checkTime() | - | An integer representing the remaining execution time the Process has left to execute before finishing. | Getter (returns remainingExecutionTime) | - | - |
| getid() | - | Returns the Process's assigned id number | Getter (returns id) | - | - |
| checkRunTime() | - | Returns the assigned run time | Getter (returns runTime) | - | - |
| updateTime(int cpuRunTime) | An integer representing the maxExecutionTime of the CPU. | void | Given the maximum execution time the process is allowed to run, this method calculates the amount of time the Thread will run upon its next execution, as well as the remaining execution time it will have after the next execution. | difference=(remainingExecutionTime-cpuRunTime);    if(difference>0): remainingExecutionTime=newTime;  runTime=cpuRunTime;  else: runTime=remainingExecutionTime; remainingExecutionTime=0; | - |

### IMPLEMENTATION:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method Name | Input | Output | Description | State Transitions | Dependencies |
| randInt(int min, int max) | An integer min, representing the minimum of the range.  An integer max, representing the maximum value of the range. |  | Generates a random integer in the range of [min, max] |  | java.util.Random |

# Testing

Notes: In order to view a trace, simply compile the program on a java compiler of your choice, and the program will generate messages indicating it’s current actions.

|  |  |
| --- | --- |
| Testing Trace | Observations |
| Process 0 loaded into ready Queue  Process 1 loaded into ready Queue  Process 0 loaded into the CPU  Process 0 executed for 2 seconds  Process 0 finished executing  Process 2 loaded into ready Queue  Process 1 loaded into the CPU  Process 1 executed for 5 seconds  Process 1 finished executing  Process 3 loaded into ready Queue  Process 2 loaded into the CPU  Process 2 executed for 5 seconds  Process 2 recycled into ready queue with 1 seconds remaining}  Process 4 loaded into ready Queue  Process 2 loaded into the CPU  Process 2 executed for 1 seconds  Process 2 finished executing  Process 5 loaded into ready Queue  Process 6 loaded into ready Queue  Process 3 loaded into the CPU  Process 3 executed for 3 seconds  Process 3 finished executing  Process 7 loaded into ready Queue  Process 4 loaded into the CPU  Process 4 executed for 5 seconds  Process 4 recycled into ready queue with 1 seconds remaining}  Process 8 loaded into ready Queue  Process 5 loaded into the CPU  Process 5 executed for 3 seconds  Process 5 finished executing  Process 9 loaded into ready Queue  Process 6 loaded into the CPU  Process 6 executed for 5 seconds  Process 6 recycled into ready queue with 1 seconds remaining}  Process 6 loaded into the CPU  Process 6 executed for 1 seconds  Process 6 finished executing  Process 10 loaded into ready Queue  Process 7 loaded into the CPU  Process 7 executed for 3 seconds  Process 7 finished executing  Process 11 loaded into ready Queue | 1. Processes 0-11: Exhibits requirement 1 a) Many processes are generated.    1. Many processes are loaded in to the ready queue, executed and removed efficiently. 2. Requirement 1 b) The whole system is deadlock free.    1. The system does not appear to have deadlock. This was tested by running the program for an extended period of time (10 minutes) and waiting for it to stop. Deadlock is avoided due to preemption. A process is forced to leave the CPU after the maximum execution time has passed for another thread to use. 3. Process 2: Exhibits requirement 1 c) No process with remaining execution time is destroyed.    1. This process executes for a total of 6 seconds. It is loaded into the CPU and executed twice. The first time it executes for 5 seconds. Five seconds is the maximum time that a CPU can execute a thread for as specified in the implementation. It is then loaded back into the queue and sent to the CPU for execution again for the remaining second. 4. Processes 0-7: Exhibits requirement 1 d) No process uses the CPU more than its allowed time.    1. The max CPU time is set to 5 seconds, none of the processes execute for more than five seconds at one time. 5. Process 0-7: Exhibits requirement 1 e) No process with no-remaining time is scheduled for execution.    1. None of these processes have an execution time equal to 0. |