CPU Scheduler Simulation Report

Written by Michael Greenberg

March 21, 2009

Table of Contents

Introduction

To properly illustrate the functionality of a CPU scheduling algorithm and the effect each algorithm has on the execution of processes, a CPU scheduling simulator was written using PHP and the results of three algorithms were collected and compared. The type of algorithm used often impacts the apparent speed at which the processes perform and complete.

In the following report, a comparison of results using the “First Come First Serve” and “Multilevel Feedback” algorithms are compared with pre-generated results of the “Shortest Job First” algorithm. We identify the strengths and weaknesses of each algorithm and discuss potential applications where each algorithm may be best suited. Additionally, we will consider the efficiency of the simulation project and discuss pitfalls of its implementation and potential areas of improvement for future iterations.

Discussion and Method

Designing this experiment to work in a controlled fashion was extremely important to getting results for each algorithm which were relevant to each other. We started with a set of predefined bursts (Table 1: Burst lengths used in each processTable 1 in the Appendix) for each process in the simulation. In designing the simulator, it was important to each of the processes as similarly as possible, only including variations which were specific to the algorithm being implemented. Originally, it was intended to use a base Scheduler class which had all of the common functions such as addProcess() and executeTick(). From this base class, all variations of scheduler would be derived implementing only the specialty functions which are needed for that algorithm. Unfortunately, in my haste and eagerness, I ended up implementing the entire “First Come First Serve” algorithm into the base Scheduler class and instead decided to keep the work. To implement the “Multilevel Feedback Queue,” the original class was copied and modified as necessary. Still, many of the original function were left untouched so reimplementation should be inconsequential.

While stepping through the code and outlining the logic used within the scheduling algorithms, I found some suboptimal steps which might affect the final results. Ideally, all process state changes should be resolved early in each cycle and at one time. This should reduce complications in overall process management and isolate CPU resource management. An easy solution might be to keep

all process state management code in location within the simulator. This implementation did not follow that idea and created a problem when pre-emption was being implemented. In review of the logic, there are two points when the state of the process is considered and action is taken. At the scheduler level, each process is considered and its state is changed as necessary according the actions the algorithm was going to take in the upcoming tick. After each process is queried by the scheduler, then each process is allowed to do its own internal work by running through a portion of its burst, or updating itself to the next state upon completion of a burst.

A more appropriate procedure might be to consider state changes necessary before the tick executes and make those changes. At this point, the process would complete whatever work it needed to accomplish internally and the cycle would complete. The next cycle would start again considering last cycle's actions and making appropriate state changes for the upcoming cycle. This ensures only one context change per cycle and pre-emption need only be considered at the beginning of the cycle before the CPU is "used" for that cycle, whereas before, pre-emption required removing the process from the CPU which may have just been assigned there.

Due to time constraints and the previously mentioned shortcoming in the original design of the simulator, pre-emption for the "Multilevel Feedback Queue" was omitted. This will obviously have a large impact on the results for that algorithm, but redesigning the flow for the scheduler would be required and a hacked solution which resembles a pre-empting action could not be found. For the purpose of analyzing the data, we will make the assumption that pre-emption has a negligible impact on the results of the simulation.

In review of the data, each algorithm has clear advantages and disadvantages. "First Come First Serve" seems to cater primarily to CPU utilization. Intuitively, CPU utilization should be only primarily affected by an effort in the scheduler to "look ahead" and factor the entire stack of bursts the process has prepared rather than looking only at the upcoming burst when deciding CPU usage. Balancing usage between IO resources and CPU resources seem to be the key to maximizing CPU utilization. Additionally, despite lack of pre-emption in the "Multilevel Feedback Queue," it is unlikely this would impact CPU utilization.

On the other hand, it should be extremely obvious how effective the "Multilevel Feedback Queue" algorithm should be in reducing Response Time for processes. Considering that no new process will be using the CPU longer than seven cycles and then get reduced to a lower priority, this will make sure that any new process will get a small amount of time on the CPU when it is first added

to the scheduler. Optimizing the process for small bursts would effectively reduce the process turnaround to an absolute minimum. Many of the bursts for the processes in this simulation were longer, otherwise there would be a stronger evidence of this in the results.

As far as the average Wait time for each process, it is unsurprising the see the smaller values for the “Shortest Job First” algorithm. This algorithm should push processes through more quickly which will take up the least amount of time on the CPU up front where slower processes would usually monopolize the CPU while shorter burst processes would normally wait. This is also evident when looking at the Turnaround time for “Shortest Job First” as it completes more processes as quickly as it can.

Overall, the “Multilevel Feedback Queue” provides the best results with the fewest drawbacks. It provides by far the best response time of any of the algorithms due to is “red carpet treatment” for new processes. The efficient processes are pushed through quickly in much the same way the “Shortest Job First” algorithm might accomplish by lowering the priority of processes with longer, drawn-out bursts. Additionally, the added overhead for the logic of “Shortest Job First” would likely slow the apparent response of the algorithm where the “Multilevel Feedback Queue” algorithm applies standard rules to each process and the results are a byproduct this standard process handling.

Appendix

Data and Results

Table 1: Burst lengths used in each process

P0 = (17,52,16,61,15,53,16,52,17,64,18,78,16,71,17)

P1 = (5,44,6,42,4,38,6,41,5,45,5,44,6)

P2 = (8,24,6,31,7,26,10,24,11,27,9,28,7)

P3 = (11,21,12,20,13,20,11,19,12,17,10,18,12)

P4 = (7,25,8,15,7,16,9,14,7,14,8,16,9,18,8)

P5 = (8,14,5,12,6,10,3,11,7,10,6,11,5,10,6,12,7)

P6 = (21,33,17,31,19,32,17,33,20,31,18,32,19,33,17)

Table 2: Simulation Results

	First Come First Served			Multilevel Feedback Queue			Shortest Job First		
	Wait	Turnaround	Response	Wait	Turnaround	Response	Wait	Turnaround	Response
P0	133	696	0	228	791	0	284	847	243
P1	179	470	17	29	320	7	27	318	0
P2	219	437	22	109	327	12	94	312	12
P3	253	449	30	143	339	19	134	330	28
P4	341	522	41	172	353	26	50	231	5
P5	353	496	48	235	378	33	64	207	20
P6	166	539	56	305	678	40	309	682	284
Average Values	**234.86**	**515.57**	**30.57**	**174.43**	**455.14**	**19.57**	**137.43**	**418.14**	**84.57**
CPU Utilization	**74.43%**			**72.31%**			**67.53%**		

Logic Flowchart

Figure 1: Logic Flowchart Page 1

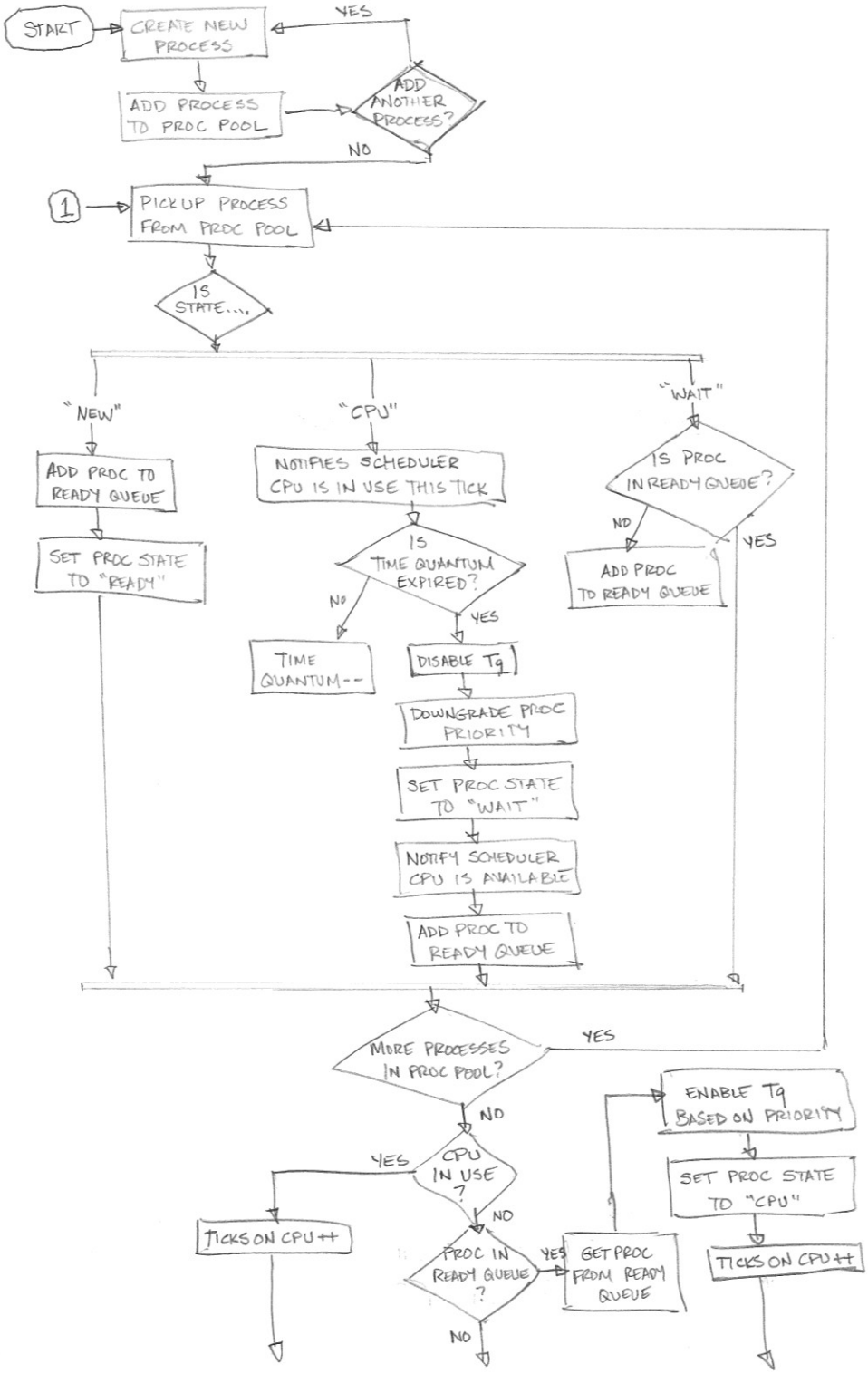
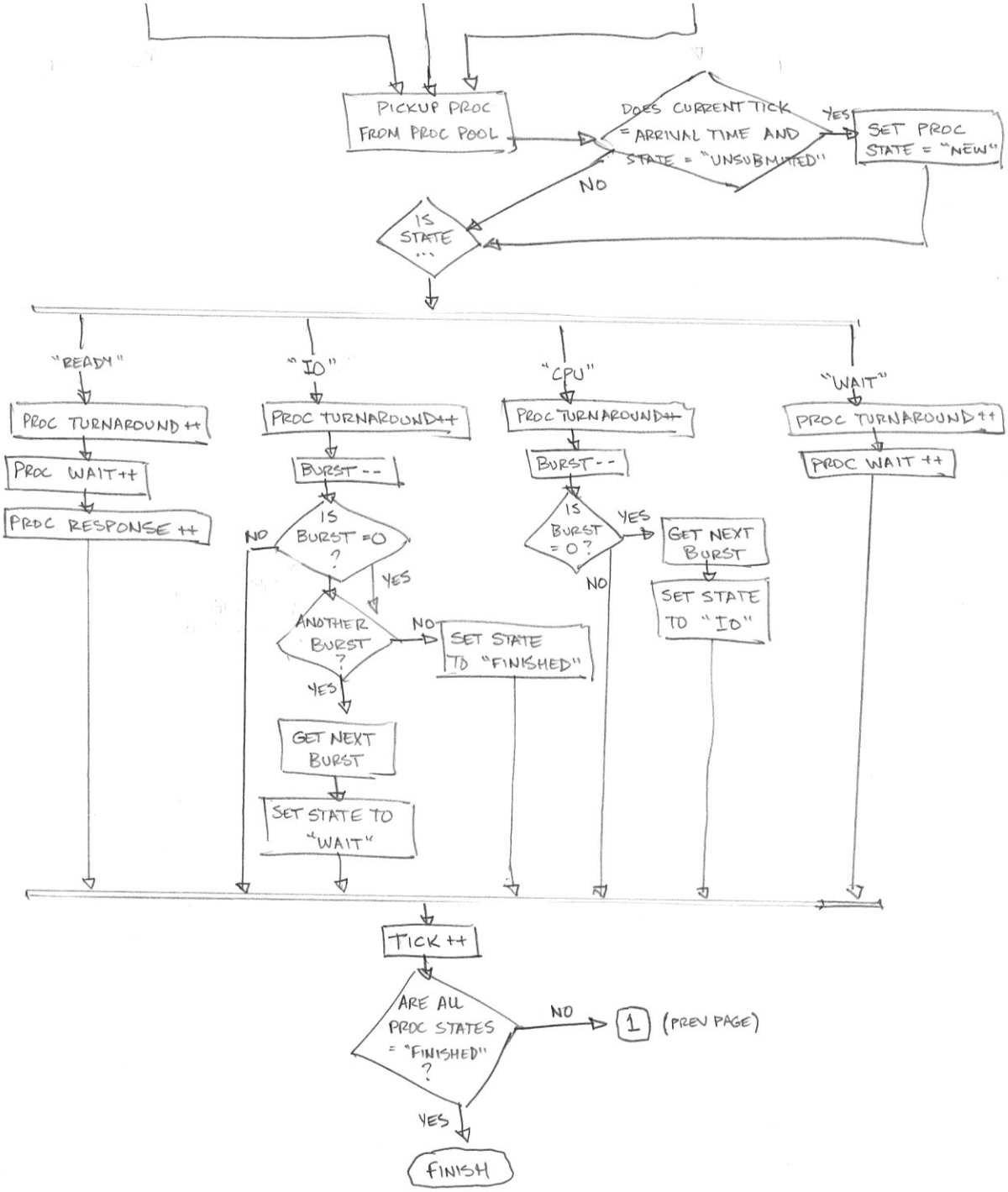


Figure 2: Logic Flowchart Page 2



Simulation Source Code

Index.php

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
  <meta http-equiv="content-type" content="text/html; charset=windows-1250">
  <title>cpuScheduler // nobulb</title>
  </head>
  <body>
<?php

  require_once('scheduler.class.php');
  require_once('scheduler_MLFQ.class.php');

  function error_handler ($type, $msg, $file, $line) {
    echo "<p>Error in $file [$line]<br />";
    echo "Message: $msg<br /></p>";
  }

  set_error_handler('error_handler');
  if (isset($_GET['v'])) {
    if ($_GET['v'] == 'fcfs') {
       $cpu = new Scheduler;
    } elseif ($_GET['v'] == 'mlfb') {
      $cpu = new MLFQ_Scheduler;
    }
    $p1Bursts = array(17,52,16,61,15,53,16,52,17,64,18,78,16,71,17);
    $p2Bursts = array(5,44,6,42,4,38,6,41,5,45,5,44,6);
    $p3Bursts = array(8,24,6,31,7,26,10,24,11,27,9,28,7);
    $p4Bursts = array(11,21,12,20,13,20,11,19,12,17,10,18,12);
    $p5Bursts = array(7,25,8,15,7,16,9,14,7,14,8,16,9,18,8);
    $p6Bursts = array(8,14,5,12,6,10,3,11,7,10,6,11,5,10,6,12,7);
    $p7Bursts = array(21,33,17,31,19,32,17,33,20,31,18,32,19,33,17);

    $p1 = new Process($p1Bursts);
    $p2 = new Process($p2Bursts);
    $p3 = new Process($p3Bursts);
    $p4 = new Process($p4Bursts);
    $p5 = new Process($p5Bursts);
    $p6 = new Process($p6Bursts);
    $p7 = new Process($p7Bursts);

    echo '<div style="font-family:monospace;">';

    $cpu->addProc($p1);
    $cpu->addProc($p2);
    $cpu->addProc($p3);
    $cpu->addProc($p4);
    $cpu->addProc($p5);
    $cpu->addProc($p6);
    $cpu->addProc($p7);
    $continueSim = true;

    while($cpu->tick()) { //tick() returns false when complete
    }

    $cpu->printHistory();
    echo '</div>';
  } else {
  ?>
  </body>
</html>
```

Process.class.php

```
<?php

  /*
  *  process.class.php
  *  Written by: Michael Greenberg
  *  Date: March 20, 2009
  *  ============================
  *  This work is licensed under the Creative Commons Attribution-Noncommercial-
  *  Share Alike 3.0 United States License. To view a copy of this license,
  *  visit http://creativecommons.org/licenses/by-nc-sa/3.0/us/ or send a letter
  *  to Creative Commons, 171 Second Street, Suite 300, San Francisco,
  *  California, 94105, USA.
  *  ============================
  *
  *  public function __construct ($inputBursts, $arrTime=0)
  *     Constuctor function.
  *       $inputBursts is an array with an odd number of values.
  *       $arrTime is the time which the process becomes "new".
  *  public function getStats ()
  *     Returns an associative array with the following indices:
  *     'responseTime', 'waitTime', 'turnaroundTime'
  *  public function getState()
  *     Returns a string identifing the state of the process. Can be any of the
  *     following values:
  *     'unsubmitted', 'new', 'ready', 'cpu', 'io', 'wait', 'finished'
  *  public function setState ($toState)
  *     Sets the state of the process.
  *       $toState is a string with the following values:
  *       'new', 'ready', 'cpu', 'io', 'wait', 'finished'
  *  public function getPriority ()
  *     Returns an integer value identifying the priority of the process.
  *     Lower value has higher priority.
  *  public function setPriority ($priority)
  *     Sets the priority of the process.
  *       $priority is an integer >= 0.
  *  public function getID()
  *     Returns a int value representing the Process ID.
  *  public function setID($name)
  *     Sets the process ID of the process.
  *       $name is an integer >=0.
  *  public function getArrival ()
  *     Returns an integer value which is the time the process "new".
  *  public function getCurrentBurst()
  *     Returns an integer value which identifies the remaining burst for the
  *     current process state.
  *  public function updateProc()
  *     Pushes the process through a tick and changes internal variables
  *     according to their state.
  *  public function printStatus ($format="full")
  *     Outputs the current status of the process.
  *       $format defaults 'full' but can also be set to 'short' or 'line'.
  *  public function printHistory()
  *     Outputs one line which shows the change of states from one tick to the
  *     next using the first letter of the state name.
  */

class Process {

  private $tick;
  private $process_id;
  private $priority;
  private $state;
  private $arrivalTime;
  private $responseTime;
  private $waitTime;
  private $turnAroundTime;
```

```
  private $burstStack;
  private $currentBurst;
  private $history;


  public function __construct ($inputBursts, $arrTime=0) {
    if ((count($inputBursts)%2) != 1) {
      trigger_error("Must use an odd number of input bursts!",E_USER_ERROR);
      die();
    }
    $this->tick = 0;
    $this->process_id = '';
    $this->priority = 0;
    if ($arrTime == 0) {
      $this->state = 'new';
    } else {
      $this->state = 'unsubmitted';
    }
    $this->arrivalTime = $arrTime;
    $this->responseTime = 0;
    $this->waitTime = 0;
    $this->turnaroundTime = 0;
    $this->burstStack = array();
    $this->burstStack = $inputBursts;
    $this->currentBurst = array_shift($this->burstStack);
    $this->history = array();
  }

  public function getStats () {
    return array(
      'responseTime' => $this->responseTime,
      'waitTime' => $this->waitTime,
      'turnaroundTime' => $this->turnaroundTime
    );
  }

  public function getState() {
    return $this->state;
  }

  public function setState ($toState) {
    switch ($toState) {
      case 'new':
        $this->state = 'new';
      return true;
      case 'ready':
        $this->state = 'ready';
      return true;
      case 'cpu':
        $this->state = 'cpu';
      return true;
      case 'io':
        $this->state = 'io';
      return true;
      case 'wait':
        $this->state = 'wait';
      return true;
      case 'finished':
        $this->state = 'finished';
        echo "Process P".$this->getID()." has completed!<br />";
      return true;
      default:
        trigger_error("Undefined state switch to '$toState' on Process '$this-
>process_id'",E_USER_NOTICE);
      return false;
    }
  }

  public function getPriority () {
```

```
    return $this->priority;
  }

  public function setPriority ($priority) {
    if (is_int($priority) && $priority >= 0) {
      $this->priority = $priority;
      return true;
    } else {
      echo "Priority is $priority <br />";
      trigger_error("Priority must be set with an integer value greater than or equal to
zero.",E_USER_ERROR);
      return false;
    }
  }

  public function getID() {
    return $this->process_id;
  }

  public function setID($name) {
    if (is_int($name) && $name >= 0) {
      $this->process_id = $name;
      return true;
    } else {
      trigger_error("Process ID must be set with an integer value greater than or equal to
zero.",E_USER_ERROR);
      return false;
    }
  }

  public function getArrival () {
    return $this->arrivalTime;
  }

  public function getCurrentBurst() {
    return $this->currentBurst;
  }

  private function updateBurstStack () {
    $this->currentBurst--;
    if ($this->currentBurst == 0) {
      $this->currentBurst = array_shift($this->burstStack);
      if ($this->currentBurst == false) {
        return -1; // burstStack is empty.
      }
      return 0; // burstStack is switching states.
    }
    return 1; // burstStack has decreased by one.
  }

  public function updateProc() {
    // Pre ppdate
    if($this->tick == $this->arrivalTime && $this->getState() == 'unsubmitted') {
      $this->setState('new');
    }

    // Update
    if($this->getState() == 'unsubmitted') {
      $this->history[$this->tick] = 'U';
    }elseif($this->getState() == 'new') {
      $this->history[$this->tick] = 'N';
    }elseif($this->getState() == 'ready') {
      $this->history[$this->tick] = 'R';
      $this->turnaroundTime++;
      $this->waitTime++;
      $this->responseTime++;
    }elseif($this->getState() == 'cpu') {
      $this->history[$this->tick] = 'C';
      $this->turnaroundTime++;
```

```
      $status = $this->updateBurstStack();
    }elseif($this->getState() == 'io') {
      $this->history[$this->tick] = 'I';
      $this->turnaroundTime++;
      $status = $this->updateBurstStack();
    }elseif($this->getState() == 'wait') {
      $this->history[$this->tick] = 'W';
      $this->turnaroundTime++;
      $this->waitTime++;
    }elseif($this->getState() == 'finished') {
      $this->history[$this->tick] = 'F';
    }
    //$this->printStatus();

    // Post update
    if (isset($status)) {
      if($status == -1)
        $this->setState('finished');
      elseif($status == 0 && $this->state == 'cpu')
        $this->setState('io');
      elseif($status == 0 && $this->state == 'io')
        $this->setState('wait');
    }
    $this->tick++;
    return $this->tick - 1;
  }

  public function printStatus ($format="full") {
    switch ($format) {
      case 'short':
        echo "Process P$this->process_id (Priority: ".$this->getPriority().")".
         " Next Burst: $this->currentBurst".
            " Wait Total: $this->waitTime<br />";
      break;
      case 'line':
        echo "<p>P$this->process_id ($this->state): ".
                  "Tr: $this->responseTime Twait: $this->waitTime Ttar: $this->turnaroundTime <br
/>";

      break;
      case 'full':
        echo "<p>P$this->process_id ($this->state): At: $this->arrivalTime CurrBurst: $this-
>currentBurst ".
             "Rt: $this->responseTime Wt: $this->waitTime Tt: $this->turnaroundTime ";
        echo '<br /> Burst Stack: ';
        print_r($this->burstStack);
        echo '</p>';
        break;
      default:
        trigger_error("Invalid format. Please use 'full', 'short', or 'line'.",E_USER_WARNING);
        printStatus();
        break;
    }
  }

  public function printHistory() {
    printf('P%-5u',$this->getID());
    foreach ($this->history as $action) {
      printf('%4s',$action);
    }
  }
}
?>
```

Scheduler.class.php

```
<?php

  /*
  *  scheduler.class.php
  *  Written by: Michael Greenberg
  *  Date: March 20, 2009
  *  ============================
  *  This work is licensed under the Creative Commons Attribution-Noncommercial-
  *  Share Alike 3.0 United States License. To view a copy of this license,
  *  visit http://creativecommons.org/licenses/by-nc-sa/3.0/us/ or send a letter
  *  to Creative Commons, 171 Second Street, Suite 300, San Francisco,
  *  California, 94105, USA.
  *  ============================
  *
  *  public function __construct()
  *     Constructor function.
  *  public function addProc (Process &$proc)
  *     Adds a process to the scheduler. Sets the process ID to the next integer
  *       $proc is a Process instance.
  *  public function tick()
  *     The primary logic of the CPU. Iterates through the added processes and
  *     simulates the actions the CPU might take according to a First-Come-
  *     First-Serve schema. Each execution of this function simulates one tick.
  *  public function printQueue()
  *     Outputs the Ready Queue of the CPU.
  *  public function printHistory()
  *     Outputs the history of each of the processes managed by the system.
  *
  */

require_once('process.class.php');

class Scheduler {
  private $tick;
  private $procQueue;
  private $procPool;
  private $procOnCPU;
  private $ticksOnCPU;

  public function __construct() {
    $this->tick = 0;
    $this->cpuInUse = false;
    $this->procQueue = array();
    $this->procPool = array();
  }

  private function processPool () {
    $cpuInUse = false;
    $procUnsubmitted = false;
    foreach ($this->procPool as $proc_key => &$proc) {
      $procState = $proc->getState();
      switch ($procState) {
        case 'unsubmitted':
          $procUnsubmitted = true;
        break;
        case 'new':
          array_push($this->procQueue,$proc->getID());
          $proc->setState('ready');
        break;
        case 'cpu':
          $cpuInUse = $proc_key;
        break;
        case 'wait':
          $thisProc = $proc->getID();
          if (array_search($thisProc,$this->procQueue) === false ) {
            array_push($this->procQueue,$proc->getID());
```

```
        }
        break;
      }
    }
    if ($cpuInUse === false) {
      $nextProc = array_shift($this->procQueue);
      if ($nextProc != NULL || $nextProc === 0) {
        $this->procOnCPU = $nextProc;

        // New process on CPU. Print dynamic execution details.
        echo "=============<br/>";
        echo "Tick #$this->tick<br />";
        echo "=============<br/>";
        echo "New Process (P$this->procOnCPU) on CPU! <br />";
        echo "Remaining Burst: ".$this->procPool[$nextProc]->getCurrentBurst()."<br /><br />";
             $this->printQueue();
        echo "<br/>";
        echo "I/O Pool ProcessNum (Remaining I/O):<br/>";
        foreach ($this->procPool as &$proc) {
          if ($proc->getState() == 'io') {
            echo "P".$proc->getID()." (".$proc->getCurrentBurst().")<br />";
          }
        }
        echo "<p />";

        $this->procPool[$nextProc]->setState('cpu');
      } else {
        // No process to put on CPU.
      }
    } else {
      $this->ticksOnCPU++;
    }
  }

  public function addProc (Process &$proc) {
    $arrayNum = array_push($this->procPool,$proc);
    $proc->setID(($arrayNum - 1));
    return $arrayNum;
  }

  public function tick() {
    // Prepare for tick
    $this->processPool();

    // During tick
    $continueSim = false;
    foreach ($this->procPool as &$proc) {
      $procTick = $proc->updateProc();
      if ($procTick != $this->tick) {
        trigger_error("Process $proc->getID() is not synched. ProcTick: $procTick SchedulerTick:
$this->tick",E_USER_NOTICE);
      }
      $procState = $proc->getState();
      if ($procState !== 'finished') {
        $continueSim = true;
      }
    }

    // Post tick
    $this->tick++;
    return $continueSim;
  }

  public function printQueue() {
    echo "Ready Queue:<br />";
    foreach ($this->procQueue as $proc) {
      $this->procPool[$proc]->printStatus('short');
    }
  }
```

```
  public function printHistory() {
    echo "<pre>Tick  ";
    for($i=0;$i<$this->tick;$i++) {
      printf('%4u',$i);
    }
    echo " |<br />";
    foreach ($this->procPool as &$proc) {
      $proc->printHistory();
      echo " |<br />";
    }
    echo "</pre>";
    printf("Took %d ticks to complete everything.<br /> CPU Utilization was %01.2f%%.<br />",$this-
>tick,($this->ticksOnCPU/$this->tick*100));
    $i=0;
    $TtarAvg=0;
    $TrAvg=0;
    $TwaitAvg=0;
    foreach ($this->procPool as &$proc) {
      $i++;
      $proc->printStatus('line');
      $stats = $proc->getStats();
      $TrAvg += $stats['responseTime'];
      $TtarAvg += $stats['turnaroundTime'];
      $TwaitAvg += $stats['waitTime'];
    }
    printf("Avg Tr: %2.2f Avg Twait: %2.2f Avg Ttar: %2.2f<br
/>",($TrAvg/$i),($TwaitAvg/$i),($TtarAvg/$i));
  }

}
?>
```

Scheduler_MLFQ.class.php

```
<?php

  /*
  *  scheduler_MLFQ.class.php
  *  Written by: Michael Greenberg
  *  Date: March 20, 2009
  *  ============================
  *  This work is licensed under the Creative Commons Attribution-Noncommercial-
  *  Share Alike 3.0 United States License. To view a copy of this license,
  *  visit http://creativecommons.org/licenses/by-nc-sa/3.0/us/ or send a letter
  *  to Creative Commons, 171 Second Street, Suite 300, San Francisco,
  *  California, 94105, USA.
  *  ============================
  *
  *  public function __construct()
  *     Constructor function.
  *  public function addProc (Process &$proc)
  *     Adds a process to the scheduler. Sets the process ID to the next integer
  *       $proc is a Process instance.
  *  public function tick()
  *     The primary logic of the CPU. Iterates through the added processes and
  *     simulates the actions the CPU might take according to a First-Come-
  *     First-Serve schema. Each execution of this function simulates one tick.
  *  public function printQueue()
  *     Outputs the Ready Queue of the CPU.
  *  public function printHistory()
  *     Outputs the history of each of the processes managed by the system.
  *
  */

require_once('process.class.php');
```

```
class MLFQ_Scheduler {
  private $tick;
  private $procQueue;
  private $procPool;
  private $procOnCPU;
  private $ticksOnCPU;
  private $timeQ;

  public function __construct() {
    $this->tick = 0;
    $this->cpuInUse = false;
    $this->procQueue = array(0=>array(),1=>array(),2=>array());
    $this->procPool = array();
    $this->timeQ = -1;
  }

  private function processPool () {
    $cpuInUse = false;
    $procUnsubmitted = false;
    foreach ($this->procPool as $proc_key => &$proc) {
      $procState = $proc->getState();
      switch ($procState) {
        case 'unsubmitted':
          $procUnsubmitted = true;
        break;
        case 'new':
          array_push($this->procQueue[$proc->getPriority()],$proc->getID());
          $proc->setState('ready');
        break;
        case 'cpu':
          $cpuInUse = $proc_key;
          if ($this->timeQ > 0) {
            $this->timeQ -= 1;
          } else if ($this->timeQ == 0) {
            $this->timeQ = -1;
            $newP = $proc->getPriority() + 1;
            settype($newP,'int');
            echo "New Priority on P".$proc->getID().": $newP <br />";
            $proc->setPriority($newP);
            $proc->setState('wait');
            $cpuInUse = false;
            $this->addToQueue($proc->getID());
          }
        break;
        case 'wait':
          $thisProc = $proc->getID();
          if ($this->findInQueue($proc->getID()) === false) {
            $this->addToQueue($proc->getID());
          }
        break;
      }
    }
    if ($cpuInUse === false) {
      $this->procOnCPU = $this->getNextProc();
      if ($this->procOnCPU !== false) {

        // New process on CPU. Print dynamic execution details.
        echo "=============<br/>";
        echo "Tick #$this->tick<br />";
        echo "=============<br/>";
        echo "New Process (P$this->procOnCPU) on CPU! <br />";
        echo "Priority ".$this->procPool[$this->procOnCPU]->getPriority()." <br />";
        echo "Remaining Burst: ".$this->procPool[$this->procOnCPU]->getCurrentBurst()."<br /><br
/>";
             $this->printQueue();
        echo "<br/>";
        echo "I/O Pool ProcessNum (Remaining I/O):<br/>";
        foreach ($this->procPool as &$proc) {
```

```
          if ($proc->getState() == 'io') {
            echo "P".$proc->getID()." (".$proc->getCurrentBurst().")<br />";
          }
        }
        echo "<p />";

        $procPriority = $this->procPool[$this->procOnCPU]->getPriority();
        switch ($procPriority) {
          case 0:
            $this->timeQ = 6;
            break;
          case 1:
            $this->timeQ = 13;
            break;
          case 2:
            $this->timeQ = -1;
            break;
        }
        $this->procPool[$this->procOnCPU]->setState('cpu');
        $this->ticksOnCPU++;
      } else {
        // No process to put on CPU.
      }
    } else {
      $this->ticksOnCPU++;
    }
  }

  public function addProc (Process &$proc) {
    $arrayNum = array_push($this->procPool,$proc);
    $proc->setID(($arrayNum - 1));
    return $arrayNum;
  }

  public function tick() {
    // Prepare for tick
    $this->processPool();

    // During tick
    $continueSim = false;
    foreach ($this->procPool as &$proc) {
      $procTick = $proc->updateProc();
      if ($procTick != $this->tick) {
        trigger_error("Process $proc->getID() is not synched. ProcTick: $procTick SchedulerTick:
$this->tick",E_USER_NOTICE);
      }
      $procState = $proc->getState();
      if ($procState !== 'finished') {
        $continueSim = true;
      }
    }

    // Post tick
    $this->tick++;
    return $continueSim;
  }

  private function getNextProc() {
    foreach ($this->procQueue as &$procs) {
      $nextProc = array_shift($procs);
      if ($nextProc !== NULL) {
        return $nextProc;
      }
    }
    return false;
  }

  static private function sortSJF($a, $b) {
    if ($a['burst'] == $b['burst']) {
```

```
      return 0;
    }
    return ($a['burst'] < $b['burst']) ? -1 : 1;
  }

  private function addToQueue($procID) {
    $oldProcPriority = $this->procPool[$this->procOnCPU]->getPriority();
    $procPriority = $this->procPool[$procID]->getPriority();
    /*
    if ($this->procOnCPU !== false && $procPriority < $oldProcPriority)
    {
      // Move old process off the CPU and back into Queue
      $oldProc = $this->procPool[$this->procOnCPU]->getID();
      $this->procPool[$oldProc]->setState('wait');
      array_unshift($this->procQueue[$oldProcPriority],$oldProc);

      // Move preempting process directly onto CPU
      $this->procOnCPU = $procID;
      $this->procPool[$this->procOnCPU]->setState('cpu');

     // New process on CPU. Print dynamic execution details.
      echo "INTERRUPTED!<br />";
      echo "=============<br/>";
      echo "Tick #$this->tick<br />";
      echo "=============<br/>";
      echo "New Process (P$this->procOnCPU) on CPU! <br />";
      echo "Priority ".$this->procPool[$this->procOnCPU]->getPriority()." <br />";
      echo "Remaining Burst: ".$this->procPool[$this->procOnCPU]->getCurrentBurst()."<br /><br />";
           $this->printQueue();
      echo "<br/>";
      echo "I/O Pool ProcessNum (Remaining I/O):<br/>";
      foreach ($this->procPool as &$proc) {
        if ($proc->getState() == 'io') {
          echo "P".$proc->getID()." (".$proc->getCurrentBurst().")<br />";
        }
      }
      echo "<p />";
      return;
    } */
    switch ($procPriority) {
      case 0:
        array_push($this->procQueue[0],$this->procPool[$procID]->getID());
        break;
      case 1:
        array_push($this->procQueue[1],$this->procPool[$procID]->getID());
        break;
      case 2:
        array_push($this->procQueue[2],$this->procPool[$procID]->getID());
        $spareQueue = array();
        $procBurst = -1;
        $lastBurst = -1;
        foreach ($this->procQueue[2] as $procB) {
          $procBurst = $this->procPool[$procB]->getCurrentBurst();
          $spareQueue[] = array('procID' => $procB, 'burst' => $procBurst);
        }
        usort($spareQueue,array("MLFQ_Scheduler","sortSJF"));
        $this->procQueue[2] = array();
        foreach ($spareQueue as $procA) {
          array_push($this->procQueue[2],$procA['procID']);
        }
        unset($spareQueue);
        break;
    }
    return;
  }

  private function findInQueue($procID) {
    $procInQueue = false;
    foreach ($this->procQueue as $qLvl=>$qProcs) {
```

```
      if (array_search($procID,$qProcs) !== false) {
        $procInQueue = true;
      }
    }
    return $procInQueue;
  }

  public function printQueue() {
    echo "Ready Queue:<br />";
    foreach ($this->procQueue as $priority=>$procLvl) {
      foreach ($procLvl as $proc) {
        $this->procPool[$proc]->printStatus('short');
      }
    }
  }

  public function printHistory() {
    echo "<pre>Tick  ";
    for($i=0;$i<$this->tick;$i++) {
      printf('%4u',$i);
    }
    echo " |<br />";
    foreach ($this->procPool as &$proc) {
      $proc->printHistory();
      echo " |<br />";
    }
    echo "</pre>";
    printf("Took %d ticks to complete everything.<br /> CPU Utilization was %01.2f%%.<br />",$this-
>tick,($this->ticksOnCPU/$this->tick*100));
    $i=0;
    $TtarAvg=0;
    $TrAvg=0;
    $TwaitAvg=0;
    foreach ($this->procPool as &$proc) {
      $i++;
      $proc->printStatus('line');
      $stats = $proc->getStats();
      $TrAvg += $stats['responseTime'];
      $TtarAvg += $stats['turnaroundTime'];
      $TwaitAvg += $stats['waitTime'];
    }
    printf("Avg Tr: %2.2f Avg Twait: %2.2f Avg Ttar: %2.2f<br
/>",($TrAvg/$i),($TwaitAvg/$i),($TtarAvg/$i));
  }
}

?>
```