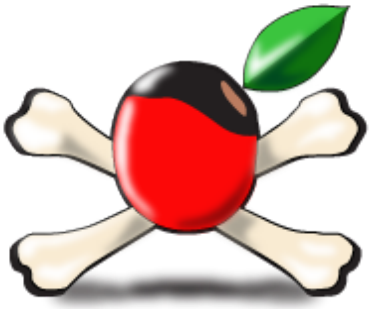


# JequirityOS

**Poison Dev Team**

Chris Koronkowski, Jordan Madah-Amiri, Ethan Mitchell, Stephen Strickland

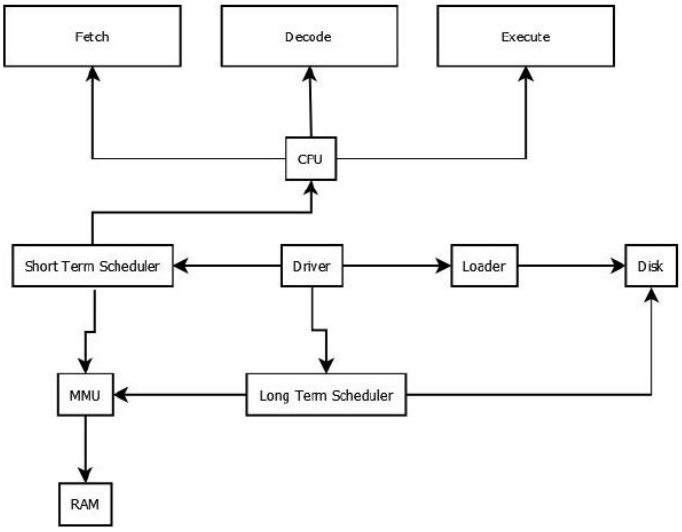
Operating Systems Section 1 Patrick Otoo Bobbie April 25, 2016

# Table of Contents

# JequirityOS Phase 2 Report

## Design Approach

Firstly as a group, we examined the project specification block diagram to get a better understanding of how the system as a whole fit together in the more common practice. After examining the diagram, we read through the phase 1 specifications. We derived a basic concept of how the system flowed in a more simplified view.

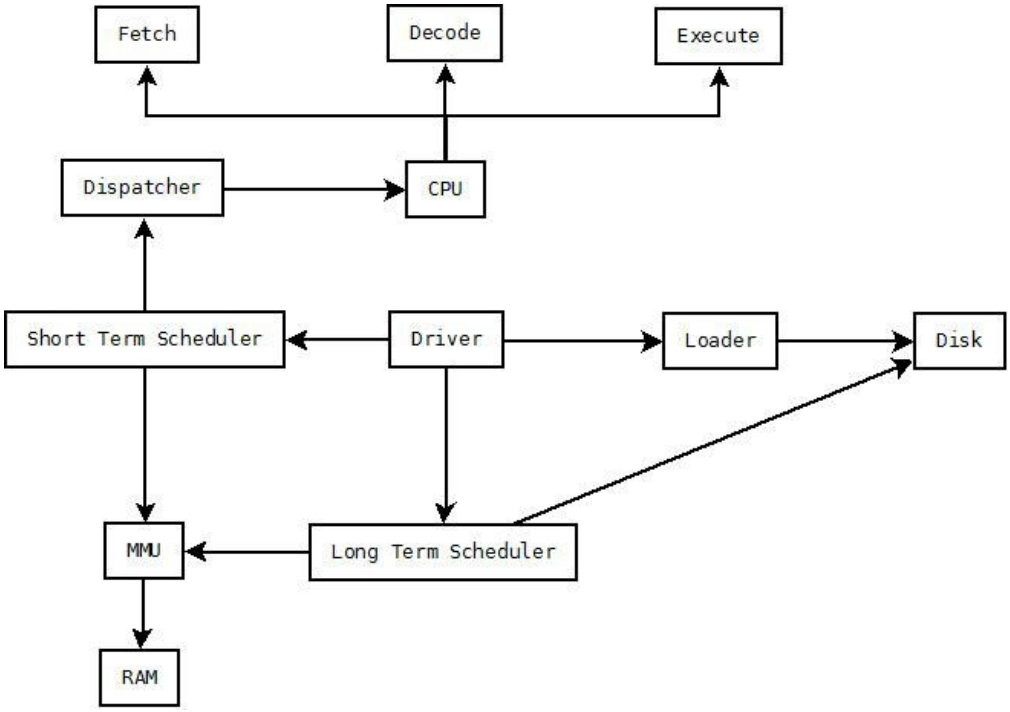



The main components that we pulled out were the CPU, the Short Term Scheduler, the Long Term Scheduler, the MMU, the RAM, the Loader, the Disk, and the Driver. We established the driver as the “engine” of the overall process. From there, we ordered the other components in a flow that takes the information provided in the instruction set and loads them onto the disk through the loader component. From there the disk will be accessed by the Long Term Scheduler and jobs will be loaded to RAM. After jobs have been loaded to RAM, the MMU, referred to in the Phase 1 Document as Memory, accesses the jobs needed to be scheduled by the Short Term Scheduler which then in turn utilizes the dispatcher to send jobs to the CPU. The CPU then loops a three part cycle of fetch, decode, and execute until it has run out of instructions at which point, the OS terminates.

For the multiprocessing portion of our OS, we needed to look at three areas: making support for multiple cores, memory management amongst the processors, and the cache of each processor. In our design approach we took note of this and decided to implement the CPU module to handle all of these items as a single processor and then using threads to handle

multiple cores. This approach makes it much easier to scale the CPU cores when appropriate.

For phase 2, we went back and reevaluated the design we had for phase 1. In doing this we addressed areas of concern for the parts that we would implement in phase 2. The general architecture idea hasn't really changed from phase 1. The Major changes came in the form of separating the Dispatcher into its own individual unit and placing it between the ShortTermScheduler and CPU modules, and the implementation of the cache and paging systems.



## Implementation Modules

The modules below have been structured and implemented in a way which will perform the simulation of an operating system running on various hardware components. Each module has its own  individual functions and properties that contribute to the system as a whole just as in a non-simulated system. Below can be found the implemented modules and a description of their major functionalities as pertaining to the system.

# Central Processing Unit

## CPU Module

As in phase 1, the CPU contains several key methods integral to the OS. The Fetch method fetches the instruction needed and then increments the program counter value by 1, so as to retrieve the next instruction to be decoded upon the next fetch call. The Decoder converts the hexadecimal instruction set into 32 bit binary sequence, separates the instructions into the proper types, and stages each in the proper place in the system for the execution method. The Execute method uses a switch loop to perform an action based on what it has received as being the decoded instruction set. Upon successful completion of the execution method will then increment the program counter to the next necessary location. The CPU module contains the Effective Address method under the title of read(), which handles all the address translation. A difference in phase 2 from phase 1 is that the CPU module can be cloned and several CPU modules can be emulated at once to emulate multicore processing in our simulated OS. JequirityOS is capable of N-CPU and has been tested at up to 2048 emulated cores, however, due to the size limitation of our RAM, only 12 CPUs are used based on the size of the jobs given in the instruction set.

CPU load

```
309     public void load(PCB job){
310
311         System.out.println("cpu.loading job: " + job.getJobNumber());
312         jobNumber = job.getJobNumber();
313         printFlags("PRE LOADING");
314        // int instructionCounter = 1;
315         jobMemoryStartAddress=job.getJobMemoryAddress();
316         start = System.currentTimeMillis();
317         j = job;
318         IOCount = 0;
319         tempBuffer = j.getTemporaryBuffer();
320         inputBuffer = j.getInputBuffer();
321         outputBuffer = j.getOutputBuffer();
322         jobCounter = j.getJobInstructionCount();
323         cacheSize = tempBuffer + inputBuffer + outputBuffer + jobCounter;
324         cache = new String[cacheSize];
325         instructionCache = new String[jobCounter];
326         VMA = job.getAllocatedVirtualPages();
327         for (int i = 0; i < cacheSize; i++)
328         {
329             cache[i] = read(i);
330 //            instructionAddress++;
331         }
332
333         programCounter = j.getProgramCounter();
334         setIdleFlag(false);
335         setTerminateFlag(false);
336         setJobLoaded(true);
337         System.out.println("cpu.loading job: " + job.getJobNumber() + " complete");
338         updateMetrics(CPUMetrics.CPU_STATE.LOADING, "N/A");
339         sleep(3);
340         updateOsMetrics();
341         printFlags("POST LOADING");
342     }
343
```

CPU run

```
344     public void run()
345     {
346         Driver.jobMetricses[currentJobNumber()-1].setStartRunTime(System.currentTimeMillis());
347         Driver.updateJobMetrics(Driver.jobMetricses[currentJobNumber()-1]);
348         printFlags("PRE RUNNING");
349         setRunning(true);
350         threadID = Thread.currentThread().getId();
351         Driver.jobsRan.add("\nRUNNING JOB: " + currentJobNumber() + "\tON THREAD: " + threadID);
352         System.out.println("\nRUNNING JOB: " + currentJobNumber() + "\tON THREAD: " + threadID);
353         while(programCounter < jobCounter)
354         {
355             String instruction = fetch(programCounter);
356             int opCode = decode(instruction);
357             executeInstruction(opCode);
358 
359             if(!jump) {
360                 programCounter++;
361             }
362             else {
363                 jump = false;
364             }
365             sleep();
366             updateMetrics(CPUMetrics.CPU_STATE.RUNNING, instruction);
367         }
368 
369         setIdleFlag(true);
370         setTerminateFlag(true);
371         setUnload(true);
372         setRunning(false);
373         Driver.jobMetricses[currentJobNumber()-1].setEndRunTime(System.currentTimeMillis());
374         Driver.updateJobMetrics(Driver.jobMetricses[currentJobNumber()-1]);
375         printFlags("POST RUNNING");
376     }
```

CPU decode

```
38  public int decode(String instruction) {
39      String binInstr = Helpers.convertFromHexStringToBinaryString(instruction.substring(2));
40      String tmpInstr = binInstr;
41      instType=Integer.parseInt(tmpInstr.substring(0,2));
42      opCode=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(2, 8));
43 
44      switch(instType) {
45          case 00: {
46              //arithmetic or logical operation
47              tmpSReg1=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(8,12));
48              tmpSReg2=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(12, 16));
49              tmpDstReg=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(16, 20));
50              break;
51          }
52          case 01: {
53              //conditional jump
54              tmpBReg=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(8, 12));
55              tmpDstReg=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(12, 16));
56              tmpAddress=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(16));
57              break;
58          }
59          case 10: {
60              //unconditional jump
61              tmpAddress=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(8));
62              break;
63          }
64          case 11: {
65              //IO operation
66              tmpReg1=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(8, 12));
67              tmpReg2=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(12, 16));
68              tmpAddress=Helpers.convertFromBinaryStringToDecimalInteger(tmpInstr.substring(16));
69              break;
70          }
71          default: {
72              System.out.println("EXCEPTION: Invalid instruction type");
73          }
74      }
75 
76      return opCode;
77  }
```

## DMA Module

This module receives the instructions that are for I/O reading or writing and call a method for a module that is not the CPU and will be able to handle the reading and writing to and from different simulated I/O devices while the Driver handles the “compute only” functionalities. This process frees the CPU of the time consuming process of transferring files leaving it more time for computational instructions that can be handled many times faster.

DMA Read/Write

```
78      public void executeInstruction(int instruction)
79      {
80          int opcode=instruction;
81          switch(opcode)
82          {
83              case 0: //0 RD
84              {   //IO read
85                  if(tmpReg2 > 0)
86                  {
87                      registers[tmpReg1] = Helpers.convertFromHexToDecimal(cache[registers[tmpReg2]/4].substring(2));
88                  }
89                  else
90                  {   //dma read for the tmpAddress
91                      registers[tmpReg1] = Helpers.convertFromHexToDecimal( cache[tmpAddress/4].substring(2));
92                  }
93                  break;
94              }
95              case 1: //1 WR
96              {
97                  //IO write
98                  if(tmpReg2 > 0)
99                  {
100                     registers[tmpReg2] =registers[tmpReg1];
101                 }
102                 else
103                 {
104                     //dmawrite with tmpAddress
105                     cache[tmpAddress/4] = "0x" + Helpers.convertFromDecimalToHex(registers[tmpReg1]);
106                 }
107                 break;
108             }
```

## Scheduling

### LongTermScheduler Module

The LongTermScheduler loads jobs into RAM. It performs the majority of its task by consulting the PCBManager for data and then sends everything off to be written to RAM. When jobs from RAM have been completed and removed and there are still more jobs on disk to be completed, the LongTermScheduler will send more jobs from disk to be written to RAM as necessary.

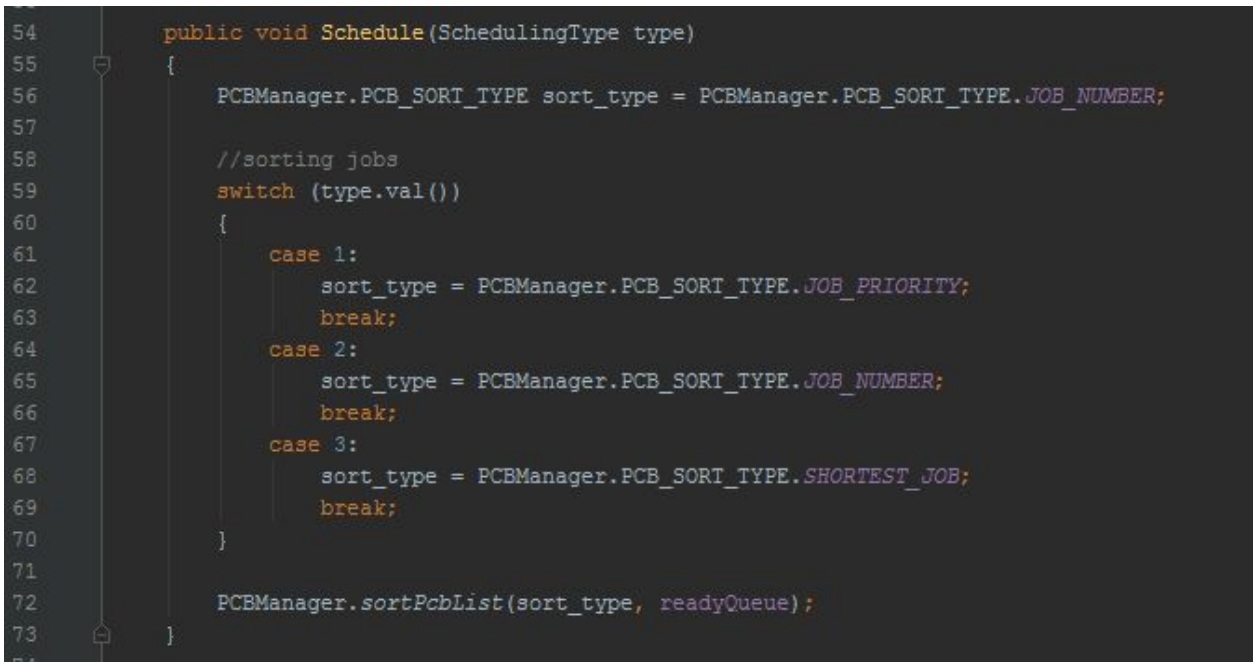
LongTermScheduler loadJobToRAM

```
27      public void loadJobToRAM(PCB block) {
28          int jobNo = block.getJobNumber();
29          int dataCardSize = block.getInputBuffer() + block.getOutputBuffer() + block.getTemporaryBuffer(); //44 or 40% of (5)
30          int memory = PCBManager.getPCB(jobNo).getJobInstructionCount() + dataCardSize;
31          int numPages= (int)Math.ceil((double)memory/(double)RAM.getPageSize());
32          int startAddress=block.getJobDiskAddress();
33          int currentDiskAddress=startAddress;
34          int physPageNo;
35          int virtualPageNo = RAM.getNextAvailableVirtualPageNumber();
36          String[] chunk;
37          ArrayList<Integer> virtualAllocatedPages = RAM.allocate(numPages);
38          if(virtualAllocatedPages.size() != 0)
39          {
40              System.out.println(jobNo);
41              PCBManager.getPCB(jobNo).setJobInMemory(true);
42              PCBManager.getPCB(jobNo).setJobMemoryAddress( virtualPageNo * RAM.getPageSize());
43              PCBManager.getPCB(jobNo).setProcessStatus(PCB.PROCESS_STATUS.READY);
44              PCBManager.getPCB(jobNo).setPagesNeeded(numPages);
45              PCBManager.getPCB(jobNo).setAllocatedVirtualPages(virtualAllocatedPages);
46              int n = RAM.getNextAvailableVirtualPageNumber();
47              for (int j=0;j<virtualAllocatedPages.size(); j++)
48              {
49                  physPageNo=RAM.getPhysicalPageNumber(virtualAllocatedPages.get(j));
50                  chunk=Disk.getChunk(currentDiskAddress, calulateChunkSize(block, currentDiskAddress));
51                  try {
52                      RAM.fillPage(physPageNo, chunk);
53                  }
54                  catch (Exception e)
55                  {
56                      e.printStackTrace();
57                      System.exit(1);
58                  }
59                  currentDiskAddress+=RAM.getPageSize();
60              }
61              JobMetrics metrics = new JobMetrics();
62              metrics.setTimestamp(System.currentTimeMillis());
63              metrics.setJobNumber(jobNo);
64              metrics.setStartWaitTime(metrics.getTimestamp());
65              metrics.setBlocksUsed(memory);
66              Driver.jobMetricses[jobNo-1].update(metrics);
67              Driver.ShortTermScheduler.addToReadyQueue(PCBManager.getPCB(jobNo));
68          }
69      }
```

## ShortTermScheduler Module

The ShortTermScheduler is responsible for scheduling the jobs for execution based on one of the desired scheduling methods: First In First Out (FIFO), Priority Scheduling, or Shortest Job First (SJF). the scheduling method changes the order in which the dispatcher calls the jobs from RAM to the CPU. This module loads jobs into the ready queue for the dispatcher. The STS calls the dispatcher according to the scheduling choice, which then loads the jobs onto the CPU.

ShortTermScheduler Schedule



```
54      public void Schedule(SchedulingType type)
55      {
56          PCBManager.PCB_SORT_TYPE sort_type = PCBManager.PCB_SORT_TYPE.JOB_NUMBER;
57
58          //sorting jobs
59          switch (type.val())
60          {
61              case 1:
62                  sort_type = PCBManager.PCB_SORT_TYPE.JOB_PRIORITY;
63                  break;
64              case 2:
65                  sort_type = PCBManager.PCB_SORT_TYPE.JOB_NUMBER;
66                  break;
67              case 3:
68                  sort_type = PCBManager.PCB_SORT_TYPE.SHORTEST_JOB;
69                  break;
70          }
71
72          PCBManager.sortPcbList(sort_type, readyQueue);
73      }
```

## Dispatcher Module

The Dispatcher module is responsible for loading jobs onto the CPU. The dispatcher has been modified for phase 2 to support job dispatching to multiple CPU cores. The dispatcher is responsible for setting jobs that have been successfully completed to a terminated state, load jobs onto currently idle CPU's, and to send the "end run, all jobs complete" signal to the OS. In order to accomplish this, the dispatcher runs through the PCB and loads the necessary jobs onto the CPU while gathering the metric data for each job.

Dispatcher loadJobs()

```
60      public static void loadJobs()
61      {
62          for (int i = 0; i < Driver.CPUs.length; i++) {
63              if (Driver.CPUs[i].isIdle() && Driver.ShortTermScheduler.readyQueue.size() > 0)
64              {
65                  PCB pcb  = Driver.ShortTermScheduler.readyQueue.pop();
66                  if(Driver.CPUs[i].shouldUnload())
67                  {
68                      System.out.println(String.format("Job:%sCPU:%sTrapped!",Driver.CPUs[i].currentJobNumber(), i));
69                      terminateJobs();
70                  }
71                  Driver.jobMetricses[pcb.getJobNumber()-1].setTimestamp(System.currentTimeMillis());
72                  Driver.jobMetricses[pcb.getJobNumber()-1].setJobNumber(pcb.getJobNumber());
73                  Driver.jobMetricses[pcb.getJobNumber()-1].setEndWaitTime(System.currentTimeMillis());
74                  Driver.jobMetricses[pcb.getJobNumber()-1].setCpuNo(i+1);
75                  Driver.updateJobMetrics(Driver.jobMetricses[pcb.getJobNumber()-1]);
76                  Driver.commands[pcb.getJobNumber()] += "loading job: " + pcb.getJobNumber() + " cpu: " + i;
77                  Driver.CPUs[i].load(pcb);
78
79              }
80          }
81      }
82  }
```

Dispatcher terminateJobs()

```
32      public static void terminateJobs()
33      {
34          for (int i = 0; i < Driver.CPUs.length; i++)
35          {
36              int currentJobNumber = Driver.CPUs[i].currentJobNumber();
37              try {
38              if (Driver.CPUs[i].isIdle() && Driver.CPUs[i].shouldTerminate()) {
39                  System.out.println(String.format("TERMINATING " + currentJobNumber + ": is idle:%s shouldTerminate:%s cpu loaded:%s shouldUnload:%s",
40                          Driver.CPUs[i].isIdle(), Driver.CPUs[i].shouldTerminate(), Driver.CPUs[i].isJobLoaded(), Driver.CPUs[i].shouldUnload()));
41
42                  System.out.println("TERMINATING JOB " + currentJobNumber + " ON CPU: " + i);
43                  Driver.commands[currentJobNumber] += "TERMINATING JOB " + currentJobNumber + " ON CPU: " + i;
44                  Driver.CPUs[i].unload(PCBManager.getPCB(currentJobNumber));
45                  MMU.synchronizeCache(PCBManager.getPCB(currentJobNumber));
46                  RAM.deallocatePcb(PCBManager.getPCB(currentJobNumber));
47                  System.out.println(String.format("TERMINATED " + currentJobNumber + ": is idle:%s shouldTerminate:%s cpu loaded:%s shouldUnload:%s",
48                          Driver.CPUs[i].isIdle(), Driver.CPUs[i].shouldTerminate(), Driver.CPUs[i].isJobLoaded(), Driver.CPUs[i].shouldUnload()));
49                  if(PCBManager.getPCB(currentJobNumber).getProcessStatus() == PCB.PROCESS_STATUS.TERMINATE)
50                      Driver.completedJobs++;
51                  Driver.updateOsMetric();
52
53              }
54              }
55              catch (ArrayIndexOutOfBoundsException ex)
56              {
57                  throw new ArrayIndexOutOfBoundsException("EXCEPTION TERMINATING JOB " + currentJobNumber + " ON CPU: " + i + " " + ex.toString());
58              }
59          }
60      }
```

# Memory System

## MMU Module

This is the memory management unit. This module facilitates the communication between other modules and the RAM.

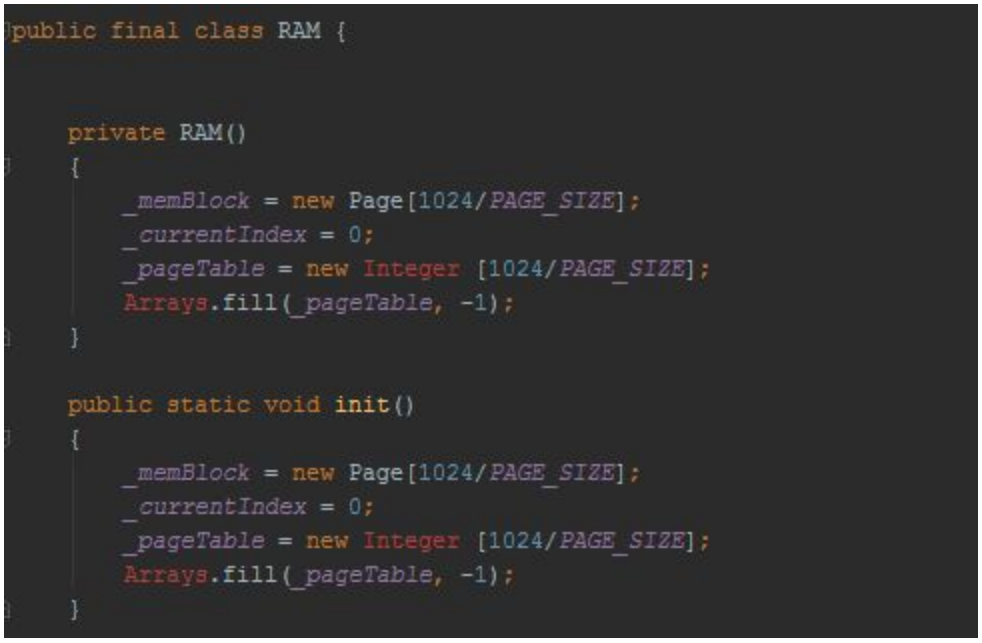
MMU read/write

```
19      public static String read(int virtualAddress)
20      {
21          int virtualPageNum = getVirtualPageNumber(virtualAddress);
22          int offset = getOffset(virtualAddress);
23
24          return readFromPhysical(getPhysicalPageNumber(virtualPageNum), offset);
25      }
26
27      public static void write(int virtualAddress, String value)
28      {
29          int virtualPageNum = getVirtualPageNumber(virtualAddress);
30          int offset = getOffset(virtualAddress);
31
32          writeToPhysical(getPhysicalPageNumber(virtualPageNum), offset, value);
33      }
34
35      public static void writeToPhysical(int physicalPageNumber, int offset, String value )
36      {
37          RAM.writeRam(physicalPageNumber, offset, value);
38      }
39
40      public static String readFromPhysical(int physicalPageNumber, int offset)
41      {
42          return RAM.readRam(physicalPageNumber, offset);
43      }
```

## RAM Module

The virtual version of our RAM that processes are loaded into. The RAM module utilizes a similar structure to the Disk module, which is an array of strings to represent 4 bit words. The RAM module is wrapped by the MMU such that no other modules interact directly with it.

RAM init



```
public final class RAM {


    private RAM()
    {
        _memBlock = new Page[1024/PAGE_SIZE];
        _currentIndex = 0;
        _pageTable = new Integer [1024/PAGE_SIZE];
        Arrays.fill(_pageTable, -1);
    }

    public static void init()
    {
        _memBlock = new Page[1024/PAGE_SIZE];
        _currentIndex = 0;
        _pageTable = new Integer [1024/PAGE_SIZE];
        Arrays.fill(_pageTable, -1);
    }
```

## Disk Module

Is the virtual version of our hard drive that holds all the info. Our disk utilizes a simple array of strings to represent the data on a hard drive.

```
7   public final class Disk {
8
9       private Disk() { _diskBlock = new String[4096]; }
13
14      public static void init() { _diskBlock = new String[4096]; }
18
19      private static String _diskBlock[] = new String[4096];
20
21  @   public static String readDisk(int index){return _diskBlock[index];}
22
23      public static void writeDisk(String val, int index){_diskBlock[index] = val;}
24
25      /**
26       * used by the LongTermScheduler when loading a job into memory. The chunk
27       * returned should fit within a page of memory.
28       * @param index
29       * @param size
30       * @return array of hex strings from disk.
31       */
32  @   public static String[] getChunk(int index, int size)
33      {
34          String[] temp = new String[size];
35          int j = 0;
36          for (int i  = index; i < index+size; i++)
37          {
38              temp[j] = _diskBlock[i];
39              j++;
40          }
41          return temp;
42      }
43  }
```

## Page Module

The page module constructs the pages that are used by the system for paging. It contains sizing information, the information the pages hold, and functions to write to the page

## Driver Module

Functions as the main() of the operating system. This module is where all other modules are called upon to function meaningfully together. First it calls the loader module, then it calls the schedulers, which in turn call the dispatcher. The driver loops until no more instructions remain in the ready queue. This functionality is under the run() method, which would be the equivalent of compute only.

Driver run()

```
95          public static void run() throws IOException
96          {
97
98              RAM.init();
99              Disk.init();
100             PCBManager.init();
101             osStartTime = System.currentTimeMillis();
102
103             Loader loader = new Loader(System.getProperty("user.dir") + "/src/ProgramFile.txt");
104             loader.Start();
105
106             jobMetricses = new JobMetrics[PCBManager.getJobListSize()];
107             for (int i = 0; i < jobMetricses.length; i++)
108                 jobMetricses[i] = new JobMetrics();
109
110
111             String s = Disk.readDisk(2);
112             System.out.println("DISK TEST: " + s);
113
114             while (custardStands())
115             {
116                 ready();
117                 aim((schedulingType != null)? schedulingType : SchedulingType.FIFO);
118                 fire();
119             }
120             executorService.shutdown();
121
122         }
```

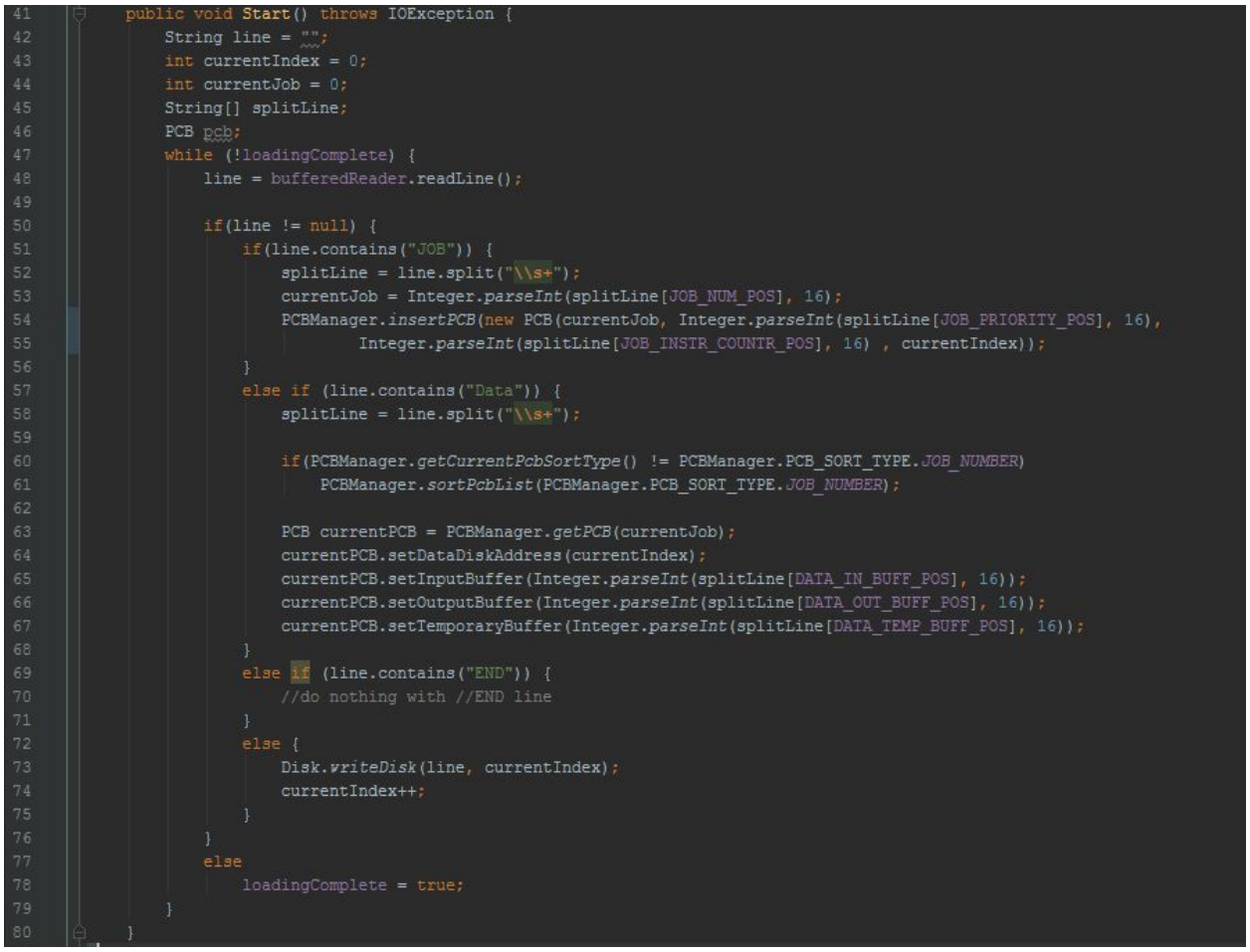
Driver computeOnly()

```
124     public static boolean custardStands() { return  !PCBManager.allJobsDone() && !isOSComplete; }
128
129     public static void ready() { LongTermScheduler.Schedule(); }
133
134     public static void aim(SchedulingType type) { ShortTermScheduler.Schedule(type); }
138
139     public static void fire()
140     {
141         Dispatcher.dispatch();
142         for (int i  = 0; i <CPUs.length && custardStands(); i++)
143         {
144             if(cpuFutures[i] != null && cpuFutures[i].isCancelled())
145                 System.out.println("XXXXXXX CANCELLED XXXXXX" + i);
146             try {
147
148             if(cpuFutures[i] != null  && cpuFutures[i].isDone())
149                 cpuFutures[i].get();
150             }
151             catch (Exception ex)
152             {
153                 System.out.println("<<FUTURE FINISHED BUT EXCEPTION WAS THROWN!!!!>>\n" + ex.toString());
154             }
155             if(((cpuFutures[i] == null || cpuFutures[i].isDone() || cpuFutures[i].isCancelled()) && CPUs[i].isJobLoaded() && !CPUs[i].shouldUnload()))
156 //                  || (!CPUs[i].isRunning() && !CPUs[i].isIdle() && !CPUs[i].shouldUnload()))
157             {
158                 cpuFutures[i] = executorService.submit(CPUs[i]);
159                 commands[CPUs[i].currentJobNumber()] += "\nRUNNING JOB: " + CPUs[i].currentJobNumber() + "\tON CPU: " + i;
160             }
161         }
```

## Loader Module

Opens the programfile and loads data onto the disk according to the format of the file. The loader module parses the data and places the jobs into the PCBManager with the appropriate information, and inserts instructions into RAM.

Loader Start()



```
41      public void Start() throws IOException {
42          String line = "";
43          int currentIndex = 0;
44          int currentJob = 0;
45          String[] splitLine;
46          PCB pcb;
47          while (!loadingComplete) {
48              line = bufferedReader.readLine();
49
50              if(line != null) {
51                  if(line.contains("JOB")) {
52                      splitLine = line.split("\\s+");
53                      currentJob = Integer.parseInt(splitLine[JOB_NUM_POS], 16);
54                      PCBManager.insertPCB(new PCB(currentJob, Integer.parseInt(splitLine[JOB_PRIORITY_POS], 16),
55                              Integer.parseInt(splitLine[JOB_INSTR_COUNTR_POS], 16) , currentIndex));
56                  }
57                  else if (line.contains("Data")) {
58                      splitLine = line.split("\\s+");
59
60                      if(PCBManager.getCurrentPcbSortType() != PCBManager.PCB_SORT_TYPE.JOB_NUMBER)
61                          PCBManager.sortPcbList(PCBManager.PCB_SORT_TYPE.JOB_NUMBER);
62
63                      PCB currentPCB = PCBManager.getPCB(currentJob);
64                      currentPCB.setDataDiskAddress(currentIndex);
65                      currentPCB.setInputBuffer(Integer.parseInt(splitLine[DATA_IN_BUFF_POS], 16));
66                      currentPCB.setOutputBuffer(Integer.parseInt(splitLine[DATA_OUT_BUFF_POS], 16));
67                      currentPCB.setTemporaryBuffer(Integer.parseInt(splitLine[DATA_TEMP_BUFF_POS], 16));
68                  }
69                  else if (line.contains("END")) {
70                      //do nothing with //END line
71                  }
72                  else {
73                      Disk.writeDisk(line, currentIndex);
74                      currentIndex++;
75                  }
76              }
77              else
78                  loadingComplete = true;
79          }
80      }
```

## Process Control Block

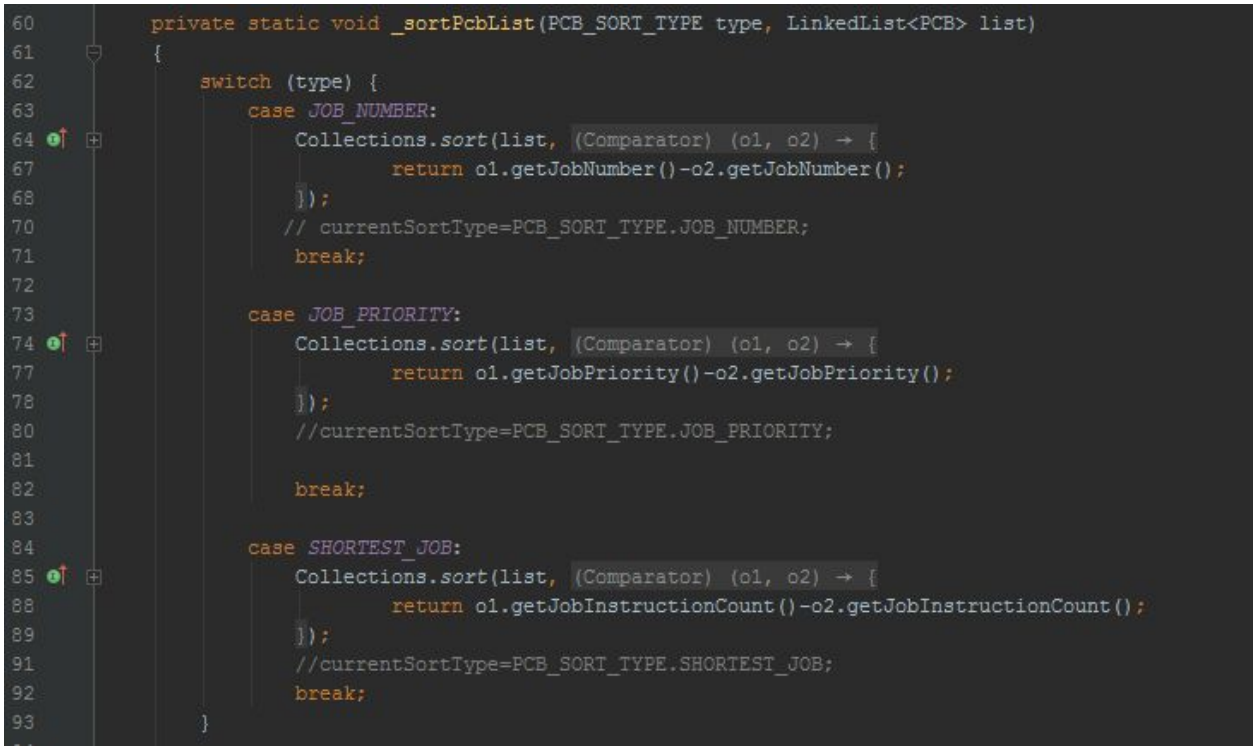
### PCB Module

This module contains all of the information pertinent to a specific job such as the job number, the priority, the disk address, etc, acting as our data structure for the PCB objects. The PCB module is not actually the entire PCB, but only a singular object which is contained by the PCBManager module. The PCB objects represent individual jobs and should not be confused with the PCBManager.

### PCBManager Module

The PCB list. Our implementation utilizes a linked list structure and contains the methods necessary to manipulate the PCB list and gather information from it for other modules. Beyond the PCB objects, the PCBManager also contains the current sort type for the schedulers: First In First Out (FIFO), Priority Scheduling, and Shortest Job First (SJF).

PCBManager sort



## Helpers Module

Helps with processes that may be required of multiple other modules. These processes are small operations here and there that perform a variety of tasks that assist other modules in completing their task. This module performs actions such as converting a hexadecimal number to a decimal number and vice versa.

## GUI Implementation

The following modules were implemented upon completion of the project's primary goal. The below modules build off of each other to facilitate the ability to better view metrics put out by the emulated OS while at the same time giving the OS a unique look and feel that performs both aesthetic and metrology related functionalities. The below modules wrap the driver module and serve as an extension to the main OS modules.

### MainFrame

This module is responsible creating the GUI for the OS. This creates  a usable interface that provides a unique look for our simulated OS. Everything is run from within the MainFrame module.

### ConsoleConstructor Module

The ConsoleConstructor module is responsible for creating a console like output from the system. This in essence creates a console unit that functions as a commandline interface with the system.

### ConsoleStream Module

The ConsoleStream module is used to provide the ConsoleConstructor with its function and appearance.

### CPUMetricsPanel Module

This module is responsible for reading in the process events and then displaying results as it performs measurements against the metrics required in the phase report specifications.

### OSMetricsPanel Module

This module creates the display for the metrics taken by the OS as it runs. These metrics

include total jobs, total jobs completed, jobs in progress, average wait time, and average run time.

### JobTable Module

The job table is a visual representation of the jobs and their current metrics. It updates the table as each job is ran and displays the job's timestamp, job number, assigned CPU, wait time, run time, and blocks used.

### MetricsDialogue Module

This module extends some of the other GUI modules and uses them to create a metrics display that is viewable during runtime.

## Simulation and Data Analysis

The OS is simulated, so the system runs in virtual space. The system begins by asking the user to enter the scheduling type that they wish to use, then the system asks the user the amount of CPU cores desired during the simulation. The Simulation then starts from the Driver. The Driver creates the simulated CPU, RAM, Disk, Long Term Scheduler, Short Term Scheduler, PCB Manager, and Loader needed by the system. Once the system starts, the Loader loads everything from the program file into the simulated Disk while creating a PCB for each process as it is loaded in. After everything is loaded in and the scheduling type and number of CPU cores has been passed into the system, the Drive starts our "compute only" equivalent. This is a 3 part cycle of loading jobs into RAM , scheduling the jobs in RAM according to the prefered scheduling method, and then dispatching the jobs to the CPU to be run. This process runs continuously until all of the jobs to be completed have been run. The first step of the cycle relies on the LongTermScheduler module to load jobs from the disk into RAM. As the jobs on ram are completed, the LongTermScheduler will add more jobs from Disk to RAM to be run. The second part of the cycle is scheduling the jobs in ram to be dispatched. This is done with the ShortTermScheduler module. The ShortTermScheduler uses the desired scheduling algorithm taken in from the user and schedules the job order in ram based on that input. The third part of the cycle is dispatching these jobs from RAM to the CPU. This is accomplished with the Dispatcher module. The LongTermScheduler will read in jobs from disk until everything has been read in. At that point the LongTermScheduler sends a signal that indicates that all jobs have been read in from Disk. Once the CPU finishes running the jobs currently in RAM the cycle is ended and the system has completed its running cycle.

For the simulation of our OS we ran the tests and gathered the metrics under 2 running conditions. The first running condition uses 1 CPU core, and the second runs under the N-CPU core condition. As mentioned earlier, JequirityOS is capable of N-CPU and has been tested at

up to 2048 emulated cores, but because of the limitation of our RAM, only 12 CPUs are ever used based on the size of the jobs given in the instruction set. For our N-CPU condition we chose to run the OS with 8 simulated cores.

The simulation was run on three scheduling algorithms: Priority Scheduling, First In First Out(FIFO), and Shortest Job First(SJF). During each run, our simulation captured a number of metrics to be used in data analysis of the CPU performance. The items gathered were: the System time at which the job started, the job number, which CPU core the job ran on, the wait time, the run time, the number of pages used for each job, and the number of I/O operations performed by each job.

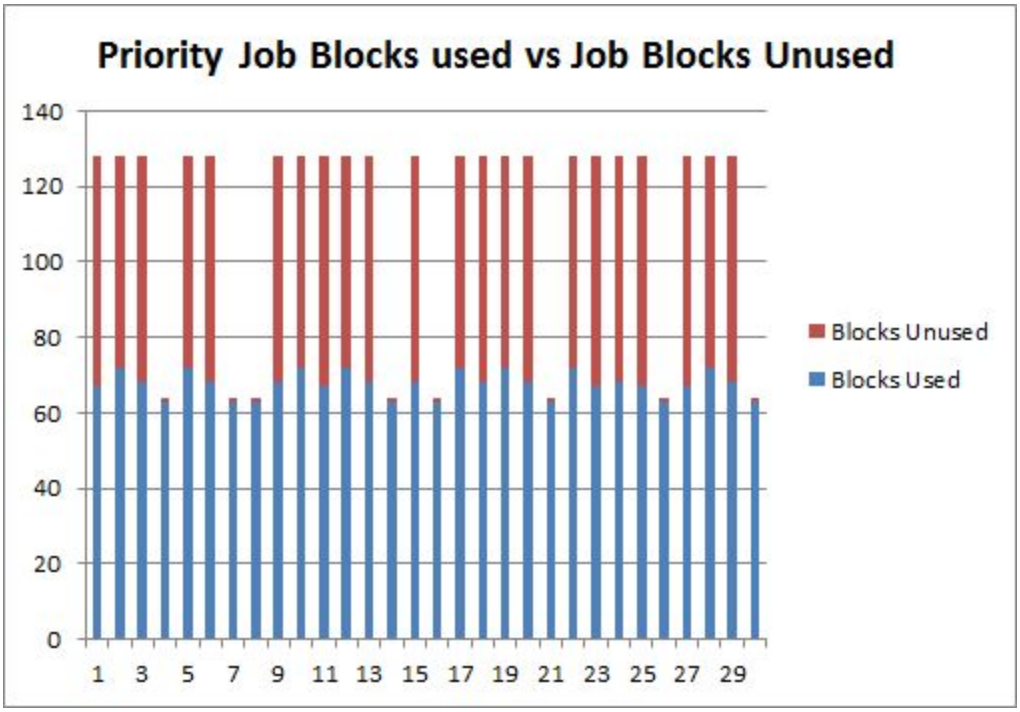
The results for each run can be found below in its appropriate category:

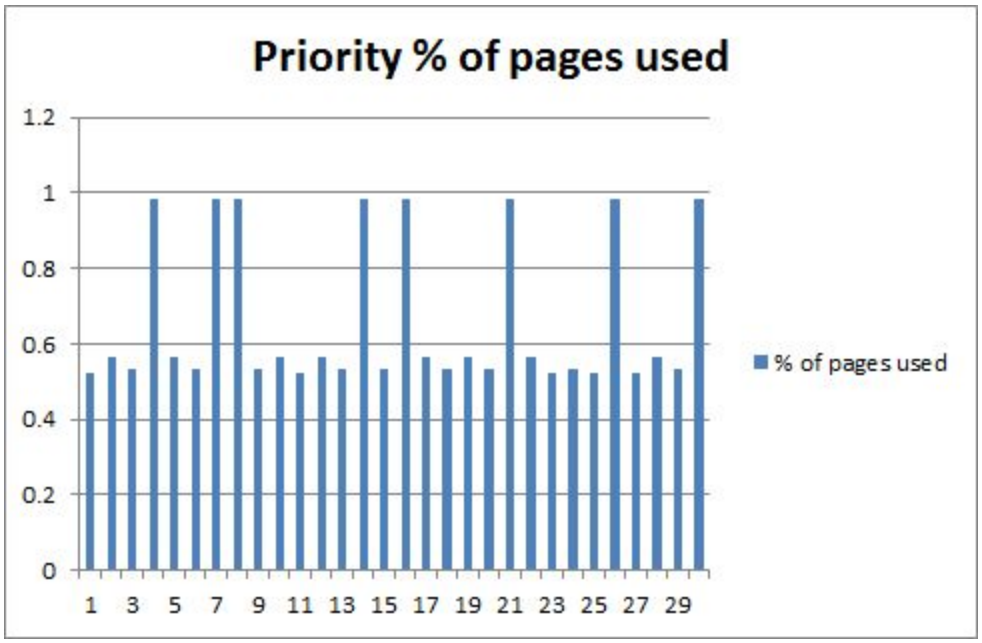
## 1 CPU Core simulation and Data

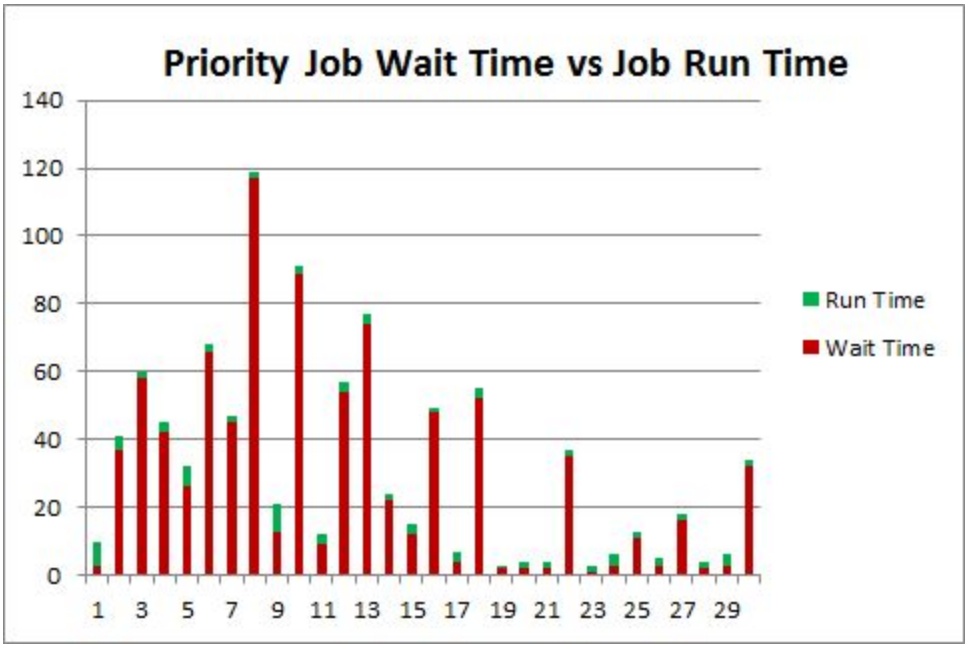
Priority Scheduling Results:

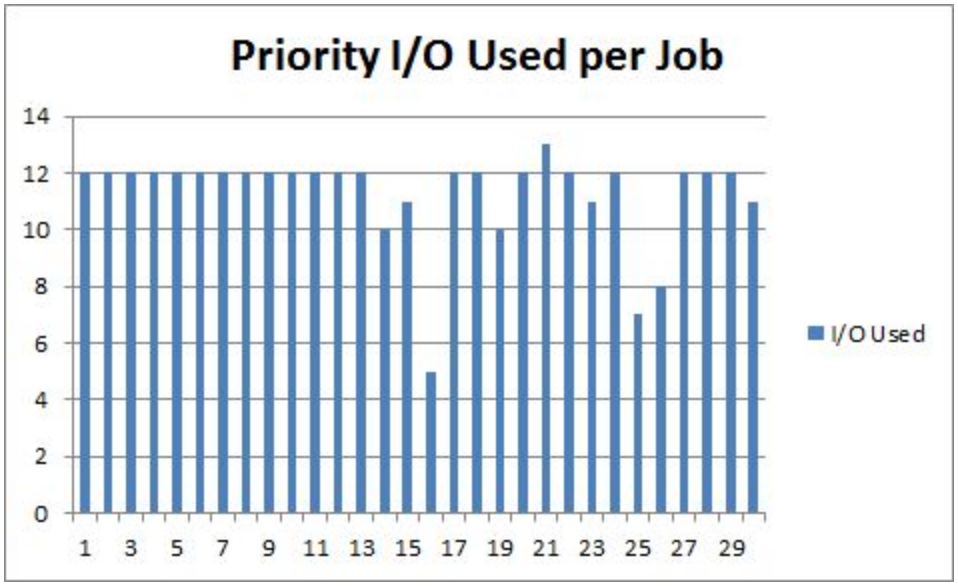
| SYSTEM TIME | JOB # | WAIT TIME | RUN TIME | BLOCKS USED / 64 = pages used | I/O Used |
|---|---|---|---|---|---|
| **PRIO** | | | | | |
| 1461184722200 | 1 | 3 | 7 | 67 | 12 |
| 1461184722211 | 9 | 13 | 8 | 68 | 12 |
| 1461184722220 | 14 | 22 | 2 | 63 | 10 |
| 1461184722223 | 5 | 26 | 6 | 72 | 12 |
| 1461184722229 | 11 | 9 | 3 | 67 | 12 |
| 1461184722234 | 2 | 37 | 4 | 72 | 12 |
| 1461184722239 | 4 | 42 | 3 | 63 | 12 |
| 1461184722242 | 7 | 45 | 2 | 63 | 12 |
| 1461184722245 | 21 | 2 | 2 | 63 | 13 |
| 1461184722249 | 26 | 3 | 2 | 63 | 8 |
| 1461184722251 | 15 | 12 | 3 | 68 | 11 |
| 1461184722255 | 3 | 58 | 2 | 68 | 12 |
| 1461184722260 | 17 | 4 | 3 | 72 | 12 |

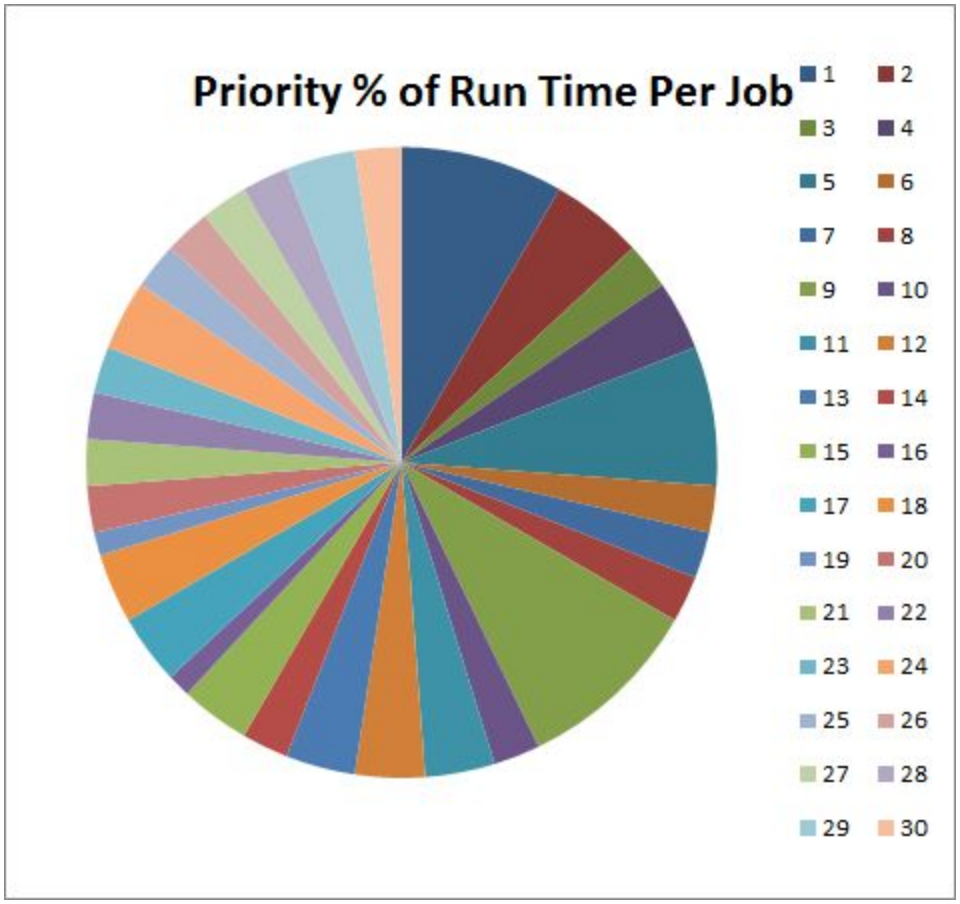
| | | | | | |
|---|---|---|---|---|---|
| 1461184722263 | 6 | 66 | 2 | 68 | 12 |
| 1461184722266 | 19 | 2 | 1 | 72 | 10 |
| 1461184722269 | 20 | 2 | 2 | 68 | 12 |
| 1461184722272 | 16 | 48 | 1 | 63 | 5 |
| 1461184722273 | 23 | 1 | 2 | 67 | 11 |
| 1461184722277 | 24 | 3 | 3 | 68 | 12 |
| 1461184722281 | 30 | 32 | 2 | 63 | 11 |
| 1461184722284 | 12 | 54 | 3 | 72 | 12 |
| 1461184722288 | 25 | 11 | 2 | 67 | 7 |
| 1461184722290 | 28 | 2 | 2 | 72 | 12 |
| 1461184722294 | 29 | 3 | 3 | 68 | 12 |
| 1461184722297 | 27 | 16 | 2 | 67 | 12 |
| 1461184722301 | 10 | 89 | 2 | 72 | 12 |
| 1461184722304 | 22 | 35 | 2 | 72 | 12 |
| 1461184722308 | 13 | 74 | 3 | 68 | 12 |
| 1461184722312 | 18 | 52 | 3 | 68 | 12 |
| 1461184722315 | 8 | 117 | 2 | 63 | 12 |


Priority Job Blocks used vs Job Blocks Unused
140
120
100
80
60
40
20
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Blocks Unused
Blocks Used


Priority % of pages used
1.2
1
0.8
0.6
0.4
0.2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
% of pages used


Priority Job Wait Time vs Job Run Time
140
120
100
80
60
40
20
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Run Time
Wait Time

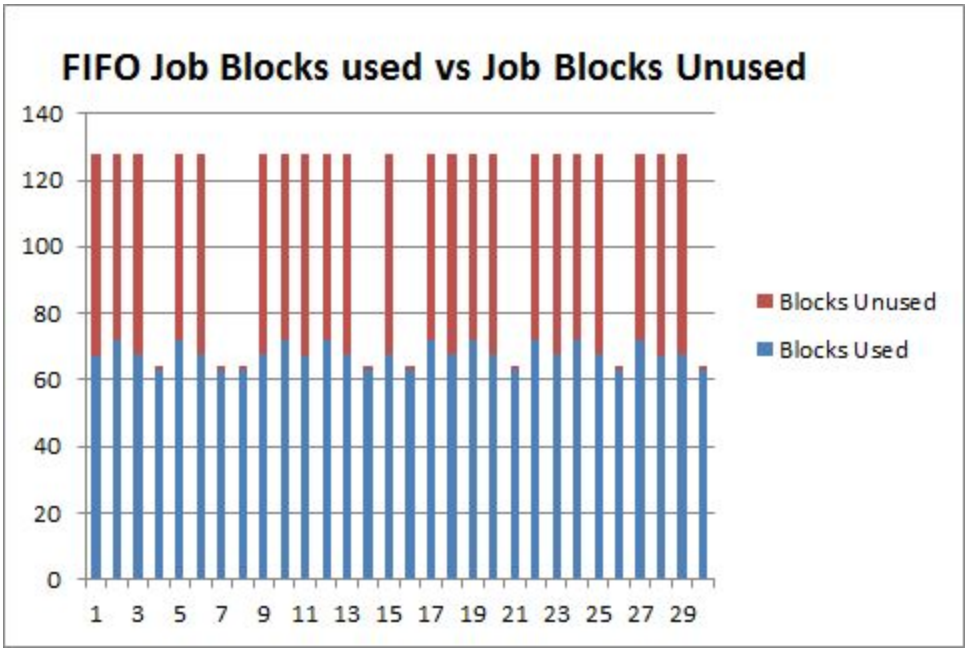

Priority I/O Used per Job
14
12
10
8
6
4
2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
I/O Used

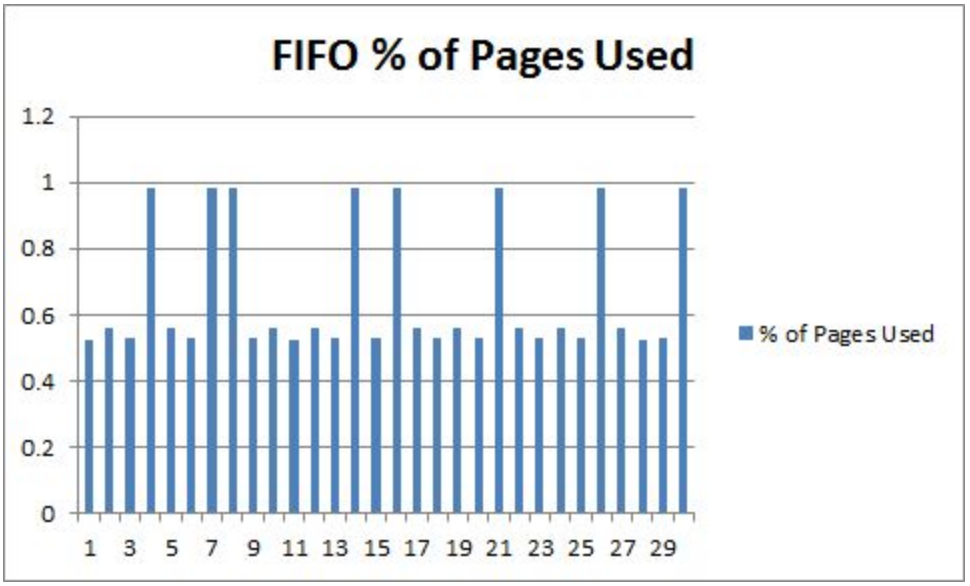


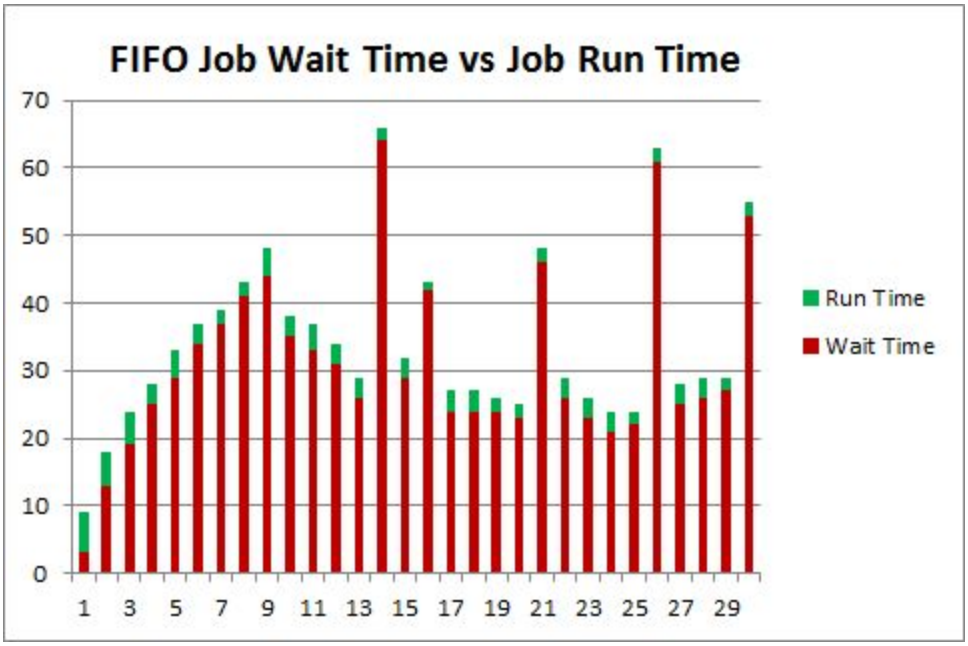

First In First Out(FIFO) Scheduling Results:

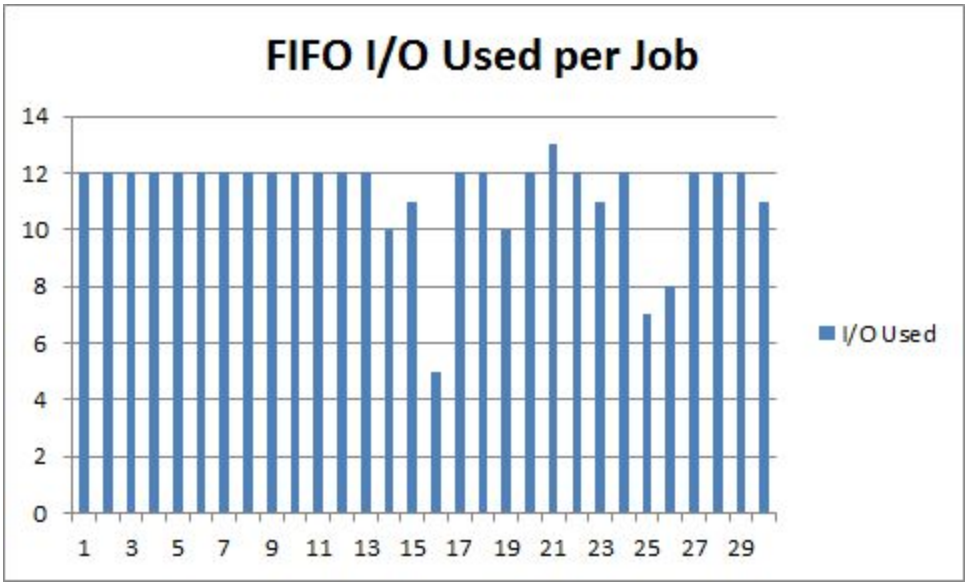
| SYSTEM TIME | JOB # | WAIT TIME | RUN TIME | BLOCKS USED / 64 = pages used | I/O Used |
|---|---|---|---|---|---|
| **FIFO** | | | | | |
| 1461184785147 | 1 | 3 | 6 | 67 | 12 |
| 1461184785157 | 2 | 13 | 5 | 72 | 12 |
| 1461184785164 | 3 | 19 | 5 | 68 | 12 |
| 1461184785170 | 4 | 25 | 3 | 63 | 12 |
| 1461184785174 | 5 | 29 | 4 | 72 | 12 |
| 1461184785179 | 6 | 34 | 3 | 68 | 12 |
| 1461184785182 | 7 | 37 | 2 | 63 | 12 |
| 1461184785186 | 8 | 41 | 2 | 63 | 12 |

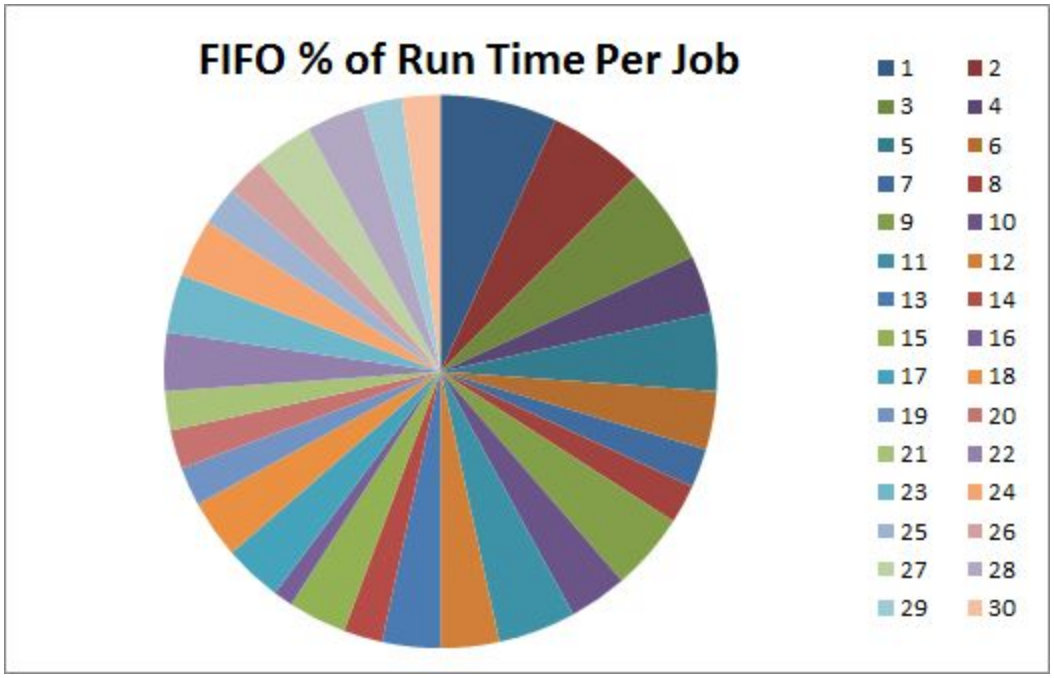
| | | | | | |
|---|---|---|---|---|---|
| 1461184785189 | 9 | 44 | 4 | 68 | 12 |
| 1461184785193 | 10 | 35 | 3 | 72 | 12 |
| 1461184785197 | 11 | 33 | 4 | 67 | 12 |
| 1461184785201 | 12 | 31 | 3 | 72 | 12 |
| 1461184785205 | 13 | 26 | 3 | 68 | 12 |
| 1461184785209 | 14 | 64 | 2 | 63 | 10 |
| 1461184785212 | 15 | 29 | 3 | 68 | 11 |
| 1461184785216 | 16 | 42 | 1 | 63 | 5 |
| 1461184785218 | 17 | 24 | 3 | 72 | 12 |
| 1461184785222 | 18 | 24 | 3 | 68 | 12 |
| 1461184785225 | 19 | 24 | 2 | 72 | 10 |
| 1461184785228 | 20 | 23 | 2 | 68 | 12 |
| 1461184785232 | 21 | 46 | 2 | 63 | 13 |
| 1461184785235 | 22 | 26 | 3 | 72 | 12 |
| 1461184785239 | 23 | 23 | 3 | 67 | 11 |
| 1461184785243 | 24 | 21 | 3 | 68 | 12 |
| 1461184785248 | 25 | 22 | 2 | 67 | 7 |
| 1461184785251 | 26 | 61 | 2 | 63 | 8 |
| 1461184785254 | 27 | 25 | 3 | 67 | 12 |
| 1461184785258 | 28 | 26 | 3 | 72 | 12 |
| 1461184785262 | 29 | 27 | 2 | 68 | 12 |
| 1461184785265 | 30 | 53 | 2 | 63 | 11 |


FIFO Job Blocks used vs Job Blocks Unused
140
120
100
80
60
40
20
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Blocks Unused
Blocks Used


FIFO % of Pages Used
1.2
1
0.8
0.6
0.4
0.2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
% of Pages Used


FIFO Job Wait Time vs Job Run Time
70
60
50
40
30
20
10
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Run Time
Wait Time

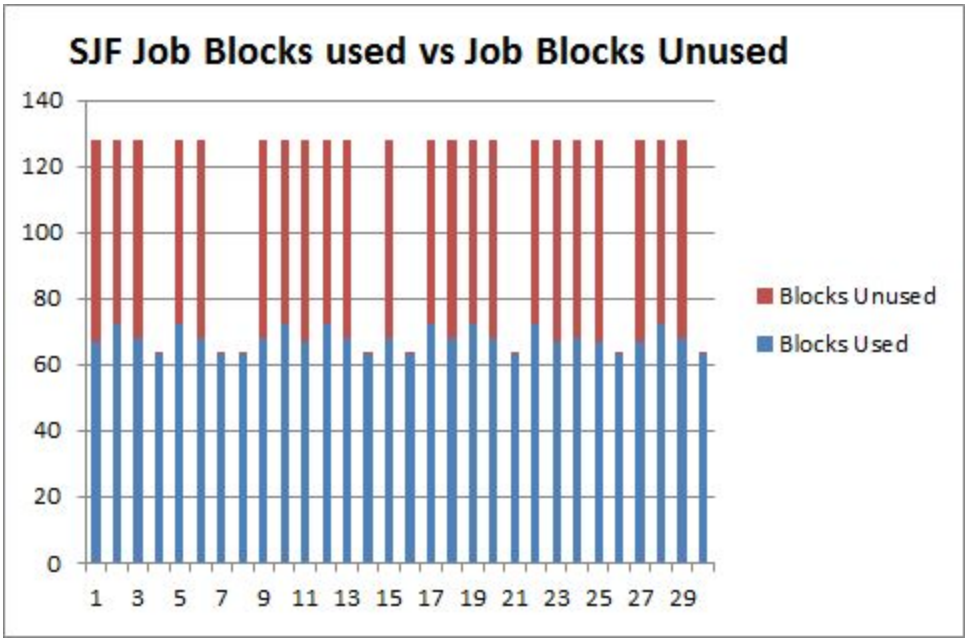

FIFO I/O Used per Job
14
12
10
8
6
4
2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
I/O Used

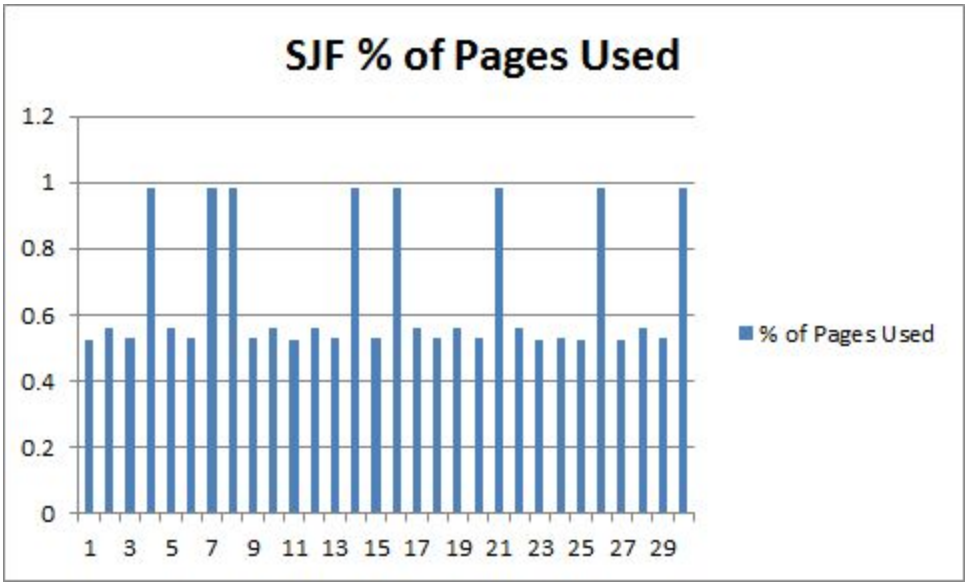


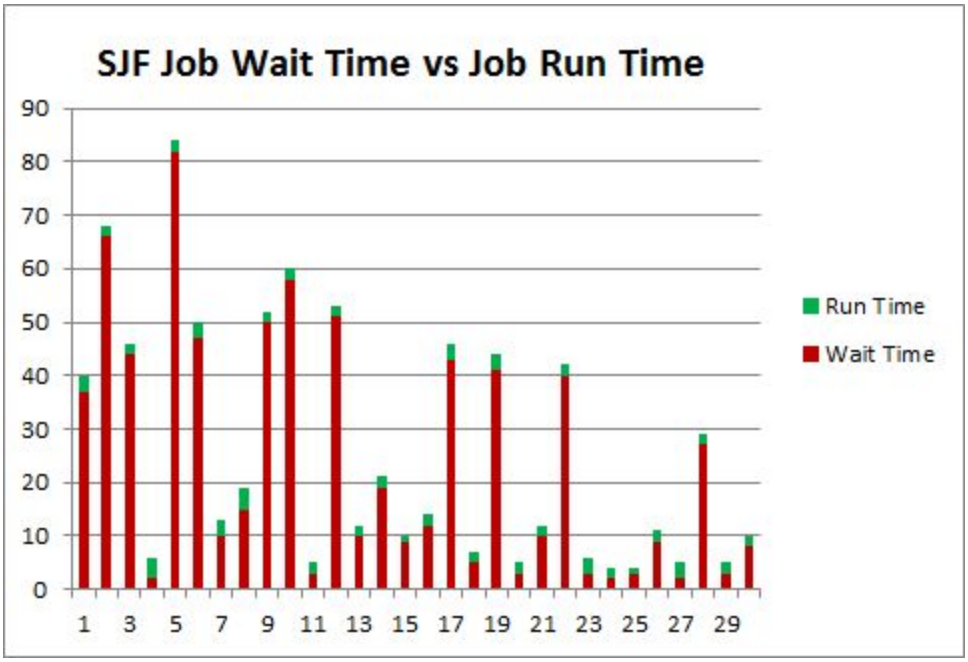

Shortest Job First Scheduling Results:

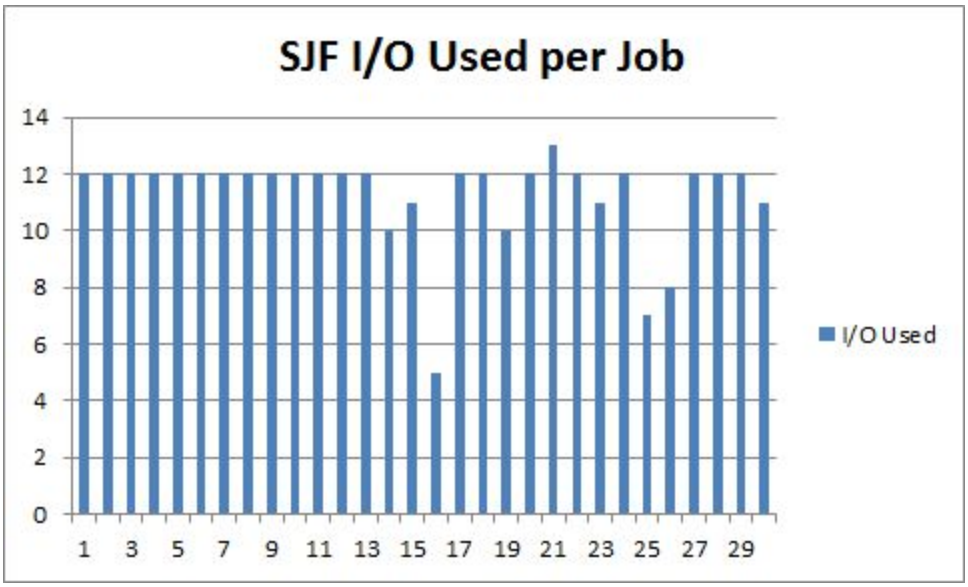
| SYSTEM TIME | JOB # | WAIT TIME | RUN TIME | BLOCKS USED / 64 = pages used | I/O Used |
|---|---|---|---|---|---|
| **SJF** | | | | | |
| 1461184818325 | 4 | 2 | 4 | 63 | 12 |
| 1461184818334 | 7 | 10 | 3 | 63 | 12 |
| 1461184818339 | 8 | 15 | 4 | 63 | 12 |
| 1461184818343 | 14 | 19 | 2 | 63 | 10 |
| 1461184818347 | 16 | 12 | 2 | 63 | 5 |
| 1461184818349 | 21 | 10 | 2 | 63 | 13 |
| 1461184818353 | 26 | 9 | 2 | 63 | 8 |
| 1461184818356 | 30 | 8 | 2 | 63 | 11 |
| 1461184818360 | 1 | 37 | 3 | 67 | 12 |
| 1461184818363 | 11 | 3 | 2 | 67 | 12 |

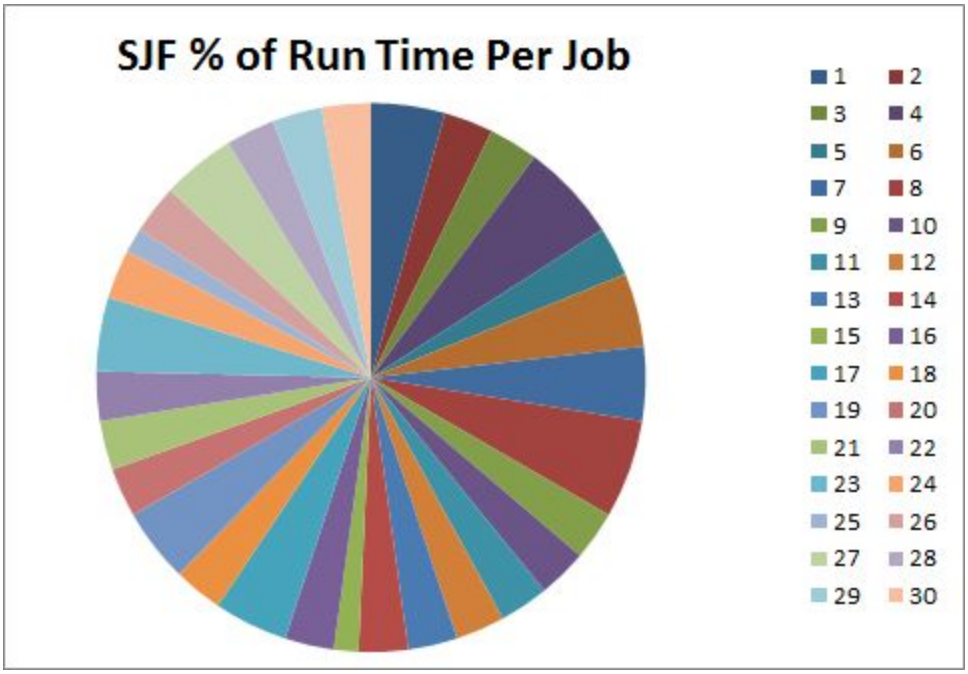
| | | | | | |
|---|---|---|---|---|---|
| 1461184818367 | 3 | 44 | 2 | 68 | 12 |
| 1461184818371 | 6 | 47 | 3 | 68 | 12 |
| 1461184818374 | 9 | 50 | 2 | 68 | 12 |
| 1461184818378 | 13 | 10 | 2 | 68 | 12 |
| 1461184818380 | 15 | 9 | 1 | 68 | 11 |
| 1461184818383 | 18 | 5 | 2 | 68 | 12 |
| 1461184818386 | 20 | 3 | 2 | 68 | 12 |
| 1461184818389 | 2 | 66 | 2 | 72 | 12 |
| 1461184818392 | 23 | 3 | 3 | 67 | 11 |
| 1461184818395 | 24 | 2 | 2 | 68 | 12 |
| 1461184818399 | 25 | 3 | 1 | 67 | 7 |
| 1461184818401 | 27 | 2 | 3 | 67 | 12 |
| 1461184818405 | 5 | 82 | 2 | 72 | 12 |
| 1461184818408 | 29 | 3 | 2 | 68 | 12 |
| 1461184818412 | 10 | 58 | 2 | 72 | 12 |
| 1461184818415 | 12 | 51 | 2 | 72 | 12 |
| 1461184818418 | 17 | 43 | 3 | 72 | 12 |
| 1461184818422 | 19 | 41 | 3 | 72 | 10 |
| 1461184818426 | 22 | 40 | 2 | 72 | 12 |
| 1461184818428 | 28 | 27 | 2 | 72 | 12 |


SJF Job Blocks used vs Job Blocks Unused
140
120
100
80
60
40
20
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Blocks Unused
Blocks Used


SJF % of Pages Used
1.2
1
0.8
0.6
0.4
0.2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
% of Pages Used


SJF Job Wait Time vs Job Run Time
90
80
70
60
50
40
30
20
10
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Run Time
Wait Time


SJF I/O Used per Job
14
12
10
8
6
4
2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
I/O Used

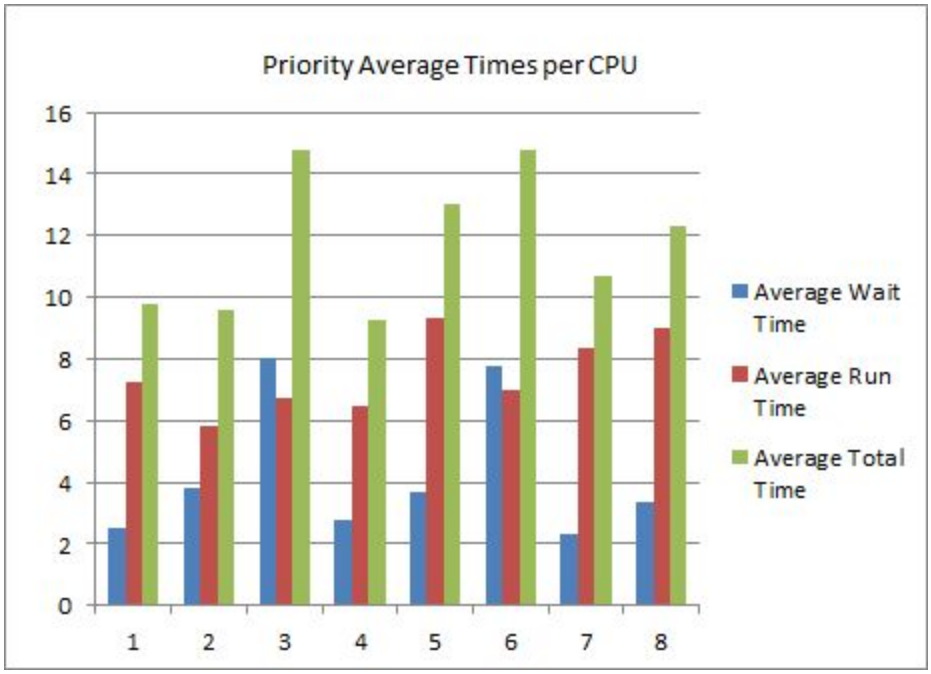


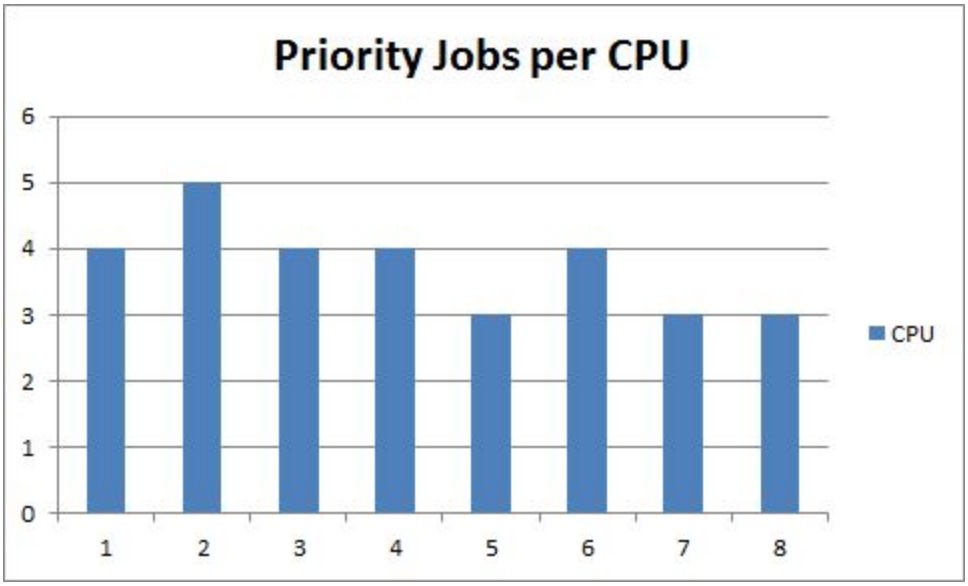

### N-CPU Cores Where N = 8 CPU Cores

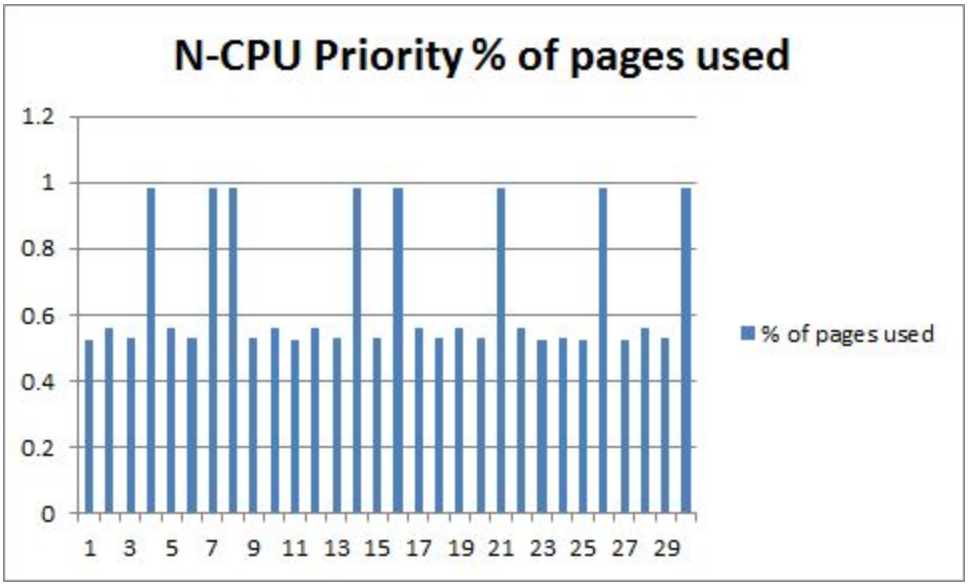
Priority Scheduling 8 CPU Cores Results:

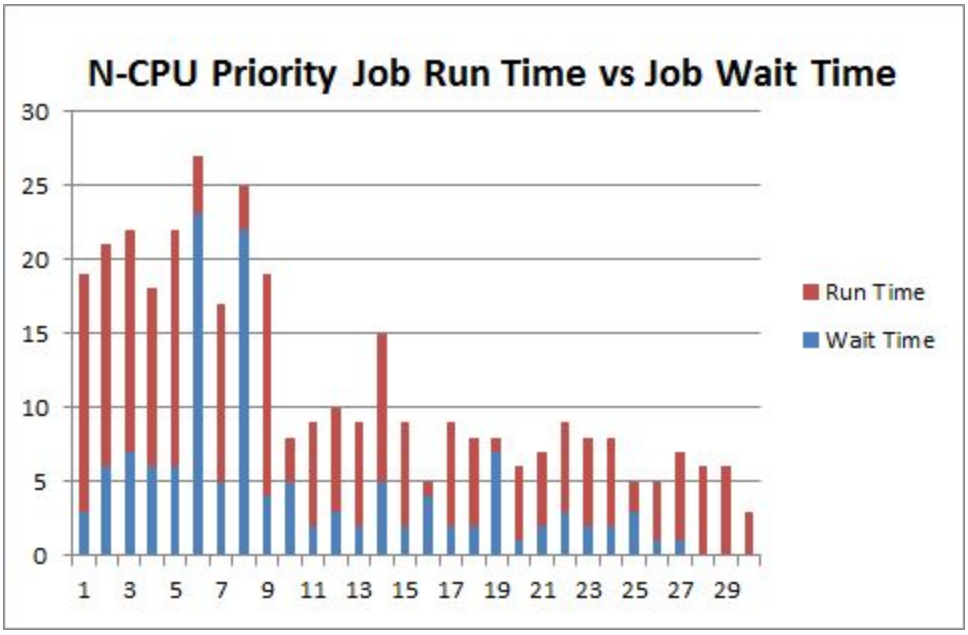
| SYSTEM TIME | JOB # | CPU # | WAIT TIME | RUN TIME | BLOCKS USED / 64 = pages used | I/Os Used |
|---|---|---|---|---|---|---|
| PRIO | | | | | | |
| 1461184594806 | 1 | 1 | 3 | 16 | 67 | 12 |
| 1461184594808 | 9 | 2 | 4 | 15 | 68 | 12 |
| 1461184594809 | 14 | 3 | 5 | 10 | 63 | 10 |
| 1461184594809 | 5 | 4 | 6 | 16 | 72 | 12 |
| 1461184594809 | 2 | 5 | 6 | 15 | 72 | 12 |
| 1461184594809 | 4 | 6 | 6 | 12 | 63 | 12 |
| 1461184594809 | 7 | 7 | 5 | 12 | 63 | 12 |
| 1461184594810 | 3 | 8 | 7 | 15 | 68 | 12 |
| 1461184594826 | 6 | 3 | 23 | 4 | 68 | 12 |
| 1461184594826 | 8 | 6 | 22 | 3 | 63 | 12 |

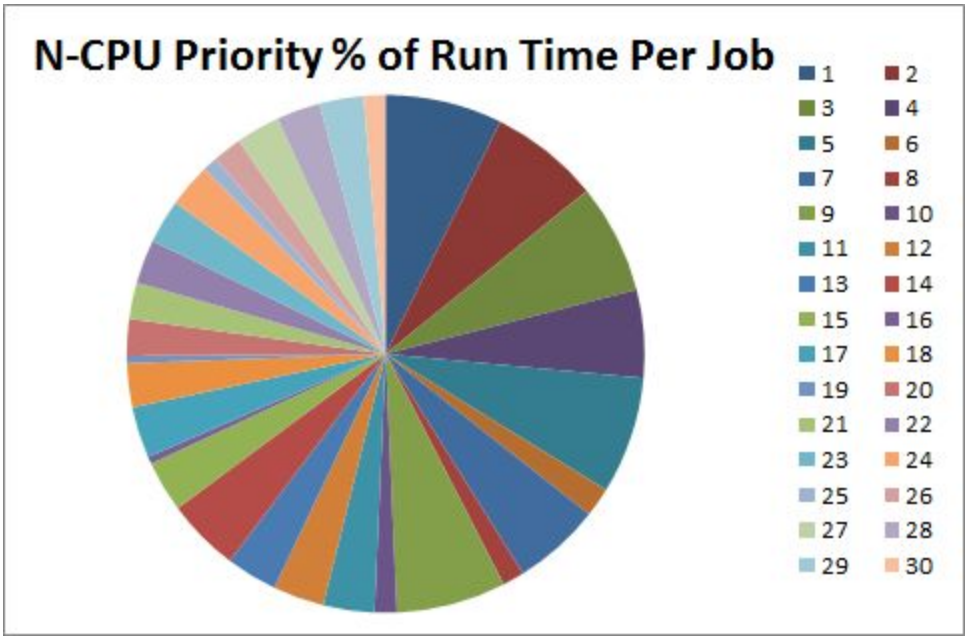
| | | | | | | |
|---|---|---|---|---|---|---|
| 1461184594831 | 16 | 1 | 4 | 1 | 63 | 5 |
| 1461184594832 | 10 | 2 | 5 | 3 | 72 | 12 |
| 1461184594834 | 11 | 1 | 2 | 7 | 67 | 12 |
| 1461184594835 | 17 | 3 | 2 | 7 | 72 | 12 |
| 1461184594835 | 21 | 4 | 2 | 5 | 63 | 13 |
| 1461184594835 | 15 | 5 | 2 | 7 | 68 | 11 |
| 1461184594835 | 12 | 6 | 3 | 7 | 72 | 12 |
| 1461184594835 | 13 | 7 | 2 | 7 | 68 | 12 |
| 1461184594837 | 26 | 2 | 1 | 4 | 63 | 8 |
| 1461184594838 | 18 | 8 | 2 | 6 | 68 | 12 |
| 1461184594845 | 19 | 2 | 7 | 1 | 72 | 10 |
| 1461184594847 | 20 | 1 | 1 | 5 | 68 | 12 |
| 1461184594848 | 23 | 2 | 2 | 6 | 67 | 11 |
| 1461184594848 | 24 | 3 | 2 | 6 | 68 | 12 |
| 1461184594849 | 25 | 4 | 3 | 2 | 67 | 7 |
| 1461184594849 | 22 | 5 | 3 | 6 | 72 | 12 |
| 1461184594850 | 29 | 6 | 0 | 6 | 68 | 12 |
| 1461184594850 | 28 | 7 | 0 | 6 | 72 | 12 |
| 1461184594850 | 27 | 8 | 1 | 6 | 67 | 12 |
| 1461184594853 | 30 | 4 | 0 | 3 | 63 | 11 |


Priority Average Times per CPU
16
14
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
Average Wait Time
Average Run Time
Average Total Time


Priority Jobs per CPU
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
CPU


N-CPU Priority % of pages used
1.2
1
0.8
0.6
0.4
0.2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
% of pages used

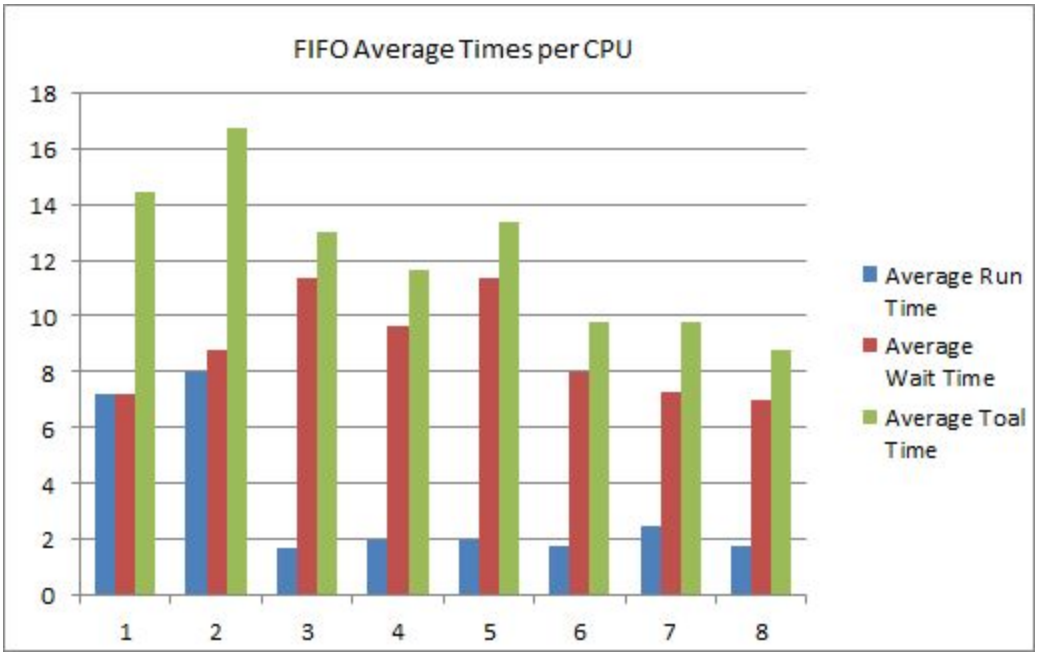

N-CPU Priority Job Run Time vs Job Wait Time
30
25
20
15
10
5
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Run Time
Wait Time

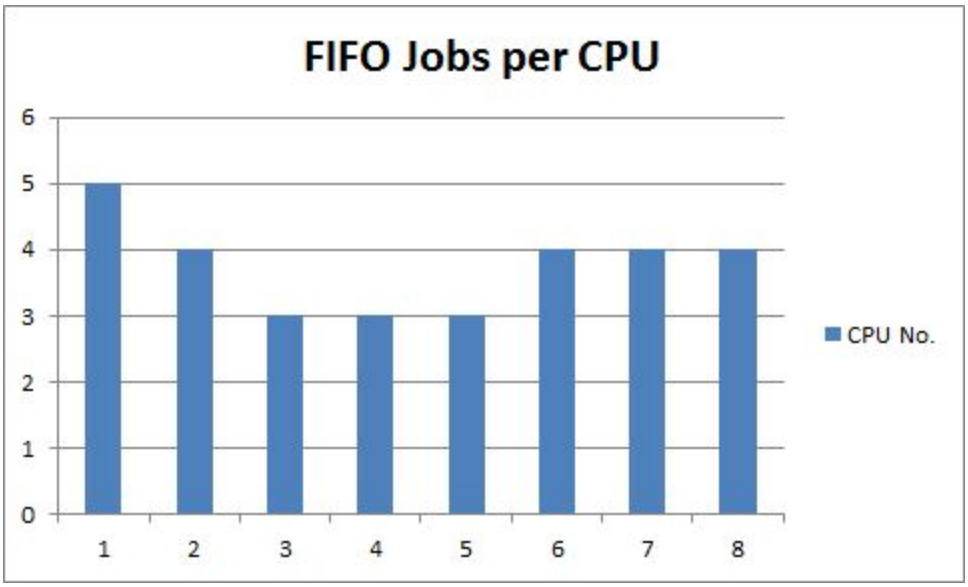


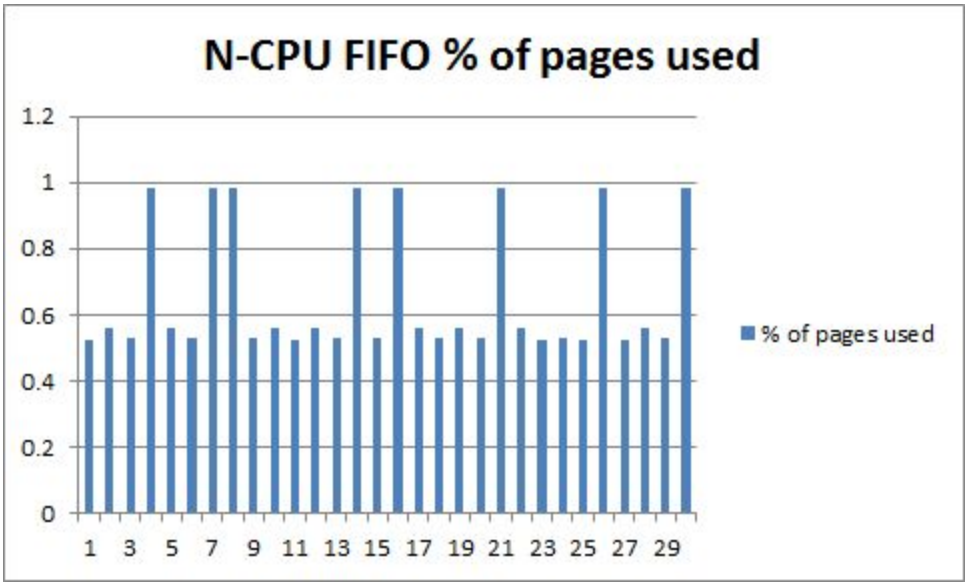

First In First Out(FIFO) Scheduling 8 CPU Cores Results:

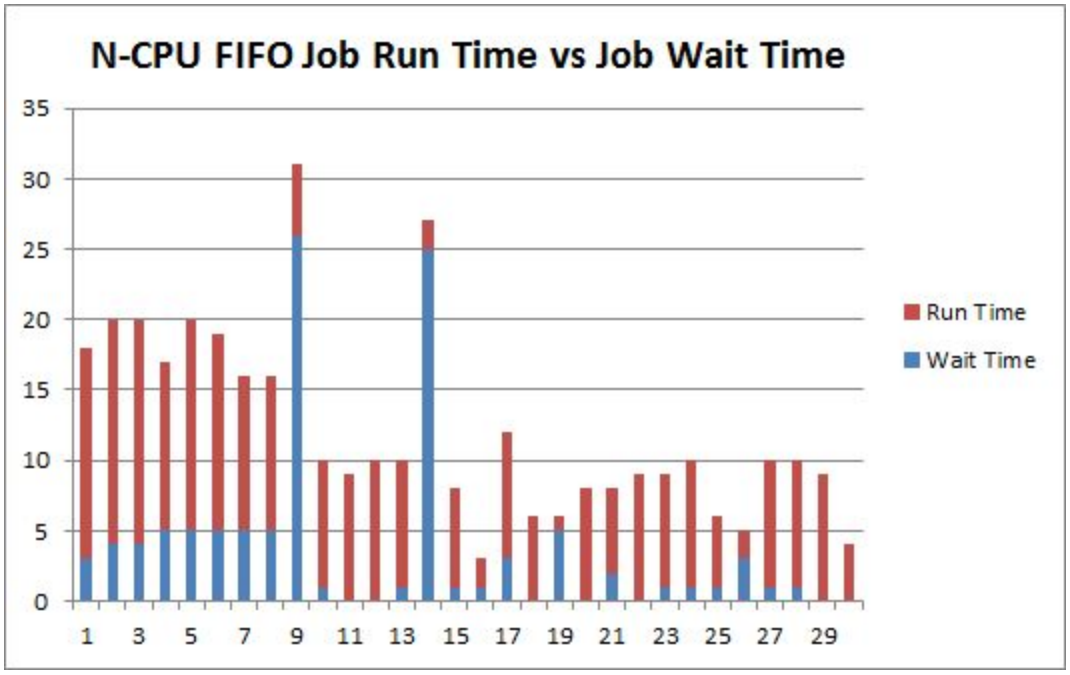
| SYSTEM TIME | JOB # | CPU # | WAIT TIME | RUN TIME | BLOCKS USED / 64 = pages used | I/Os Used |
|---|---|---|---|---|---|---|
| **FIFO** | | | | | | |
| 1461184653796 | 1 | 1 | 3 | 15 | 67 | 12 |
| 1461184653797 | 2 | 2 | 4 | 16 | 72 | 12 |
| 1461184653797 | 3 | 3 | 4 | 16 | 68 | 12 |
| 1461184653798 | 4 | 4 | 5 | 12 | 63 | 12 |
| 1461184653798 | 5 | 5 | 5 | 15 | 72 | 12 |
| 1461184653798 | 6 | 6 | 5 | 14 | 68 | 12 |
| 1461184653798 | 7 | 7 | 5 | 11 | 63 | 12 |
| 1461184653798 | 8 | 8 | 5 | 11 | 63 | 12 |
| 1461184653819 | 9 | 1 | 26 | 5 | 68 | 12 |
| 1461184653819 | 14 | 2 | 25 | 2 | 63 | 10 |
| 1461184653820 | 10 | 3 | 1 | 9 | 72 | 12 |
| 1461184653820 | 11 | 4 | 0 | 9 | 67 | 12 |

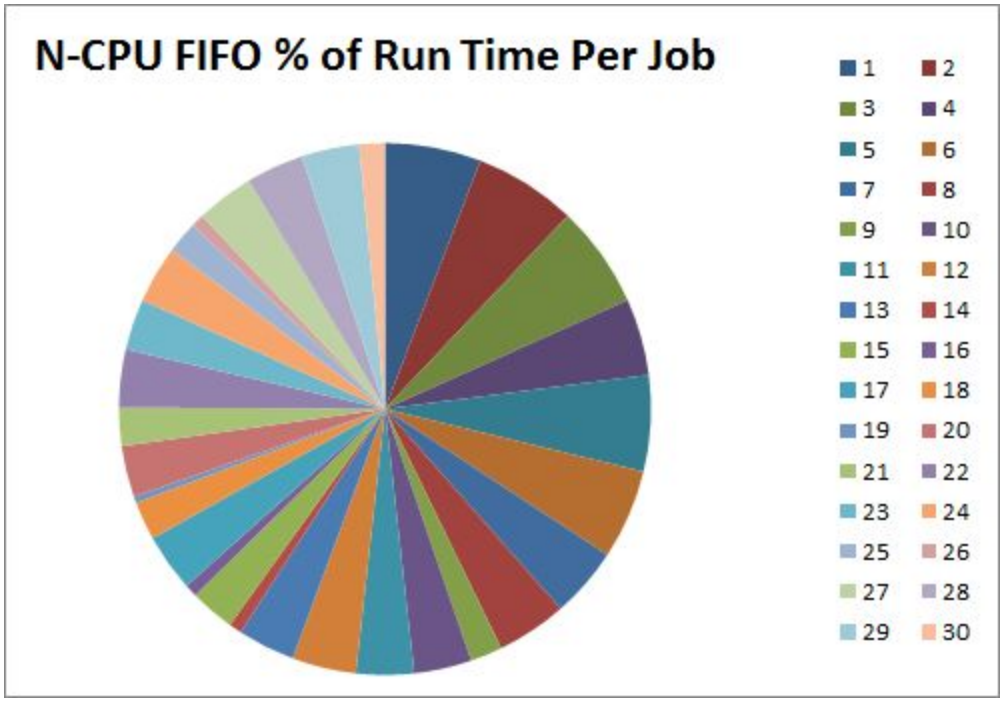
| | | | | | | |
|---|---|---|---|---|---|---|
| 1461184653820 | 12 | 5 | 0 | 10 | 72 | 12 |
| 1461184653821 | 13 | 6 | 1 | 9 | 68 | 12 |
| 1461184653821 | 15 | 7 | 1 | 7 | 68 | 11 |
| 1461184653821 | 16 | 8 | 1 | 2 | 63 | 5 |
| 1461184653823 | 17 | 2 | 3 | 9 | 72 | 12 |
| 1461184653826 | 21 | 1 | 2 | 6 | 63 | 13 |
| 1461184653827 | 18 | 8 | 0 | 6 | 68 | 12 |
| 1461184653830 | 26 | 7 | 3 | 2 | 63 | 8 |
| 1461184653835 | 19 | 1 | 5 | 1 | 72 | 10 |
| 1461184653836 | 20 | 2 | 0 | 8 | 68 | 12 |
| 1461184653836 | 22 | 3 | 0 | 9 | 72 | 12 |
| 1461184653837 | 23 | 4 | 1 | 8 | 67 | 11 |
| 1461184653837 | 24 | 5 | 1 | 9 | 68 | 12 |
| 1461184653837 | 25 | 6 | 1 | 5 | 67 | 7 |
| 1461184653837 | 27 | 7 | 1 | 9 | 67 | 12 |
| 1461184653837 | 28 | 8 | 1 | 9 | 72 | 12 |
| 1461184653839 | 29 | 1 | 0 | 9 | 68 | 12 |
| 1461184653844 | 30 | 6 | 0 | 4 | 63 | 11 |


FIFO Average Times per CPU
18
16
14
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
Average Run Time
Average Wait Time
Average Toal Time


FIFO Jobs per CPU
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
CPU No.


N-CPU FIFO % of pages used
1.2
1
0.8
0.6
0.4
0.2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
% of pages used

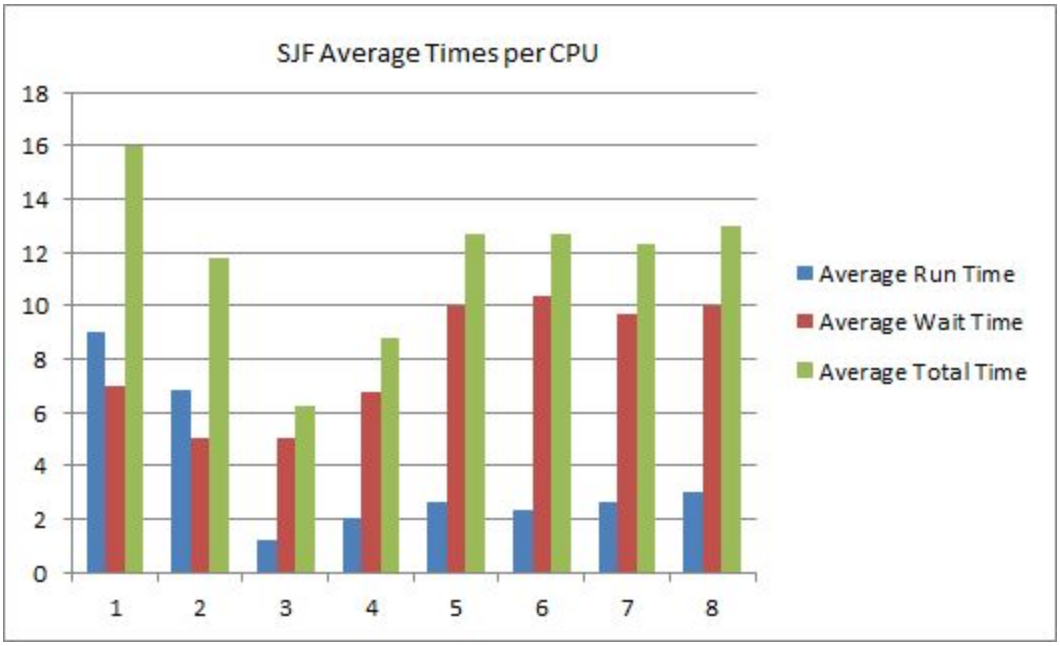

N-CPU FIFO Job Run Time vs Job Wait Time
35
30
25
20
15
10
5
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Run Time
Wait Time

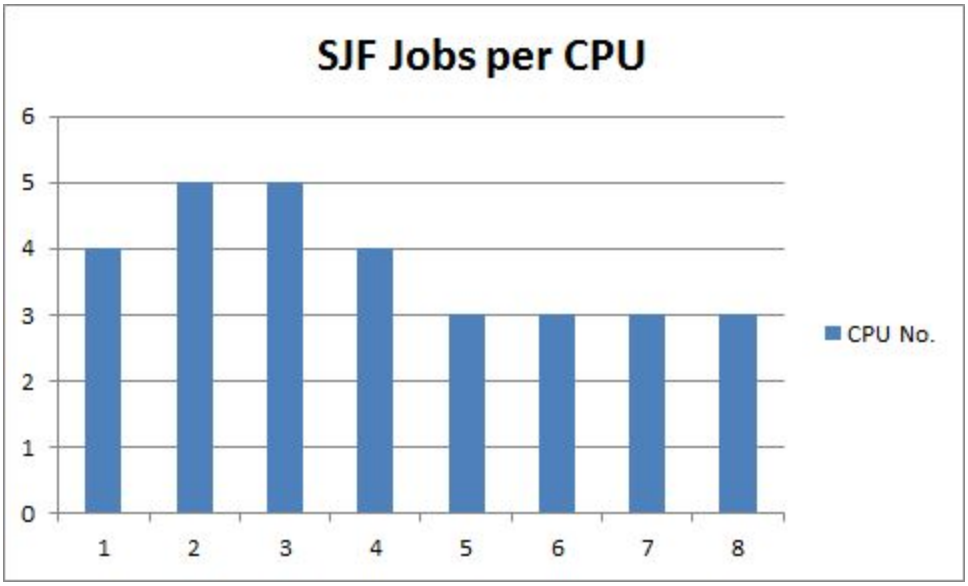


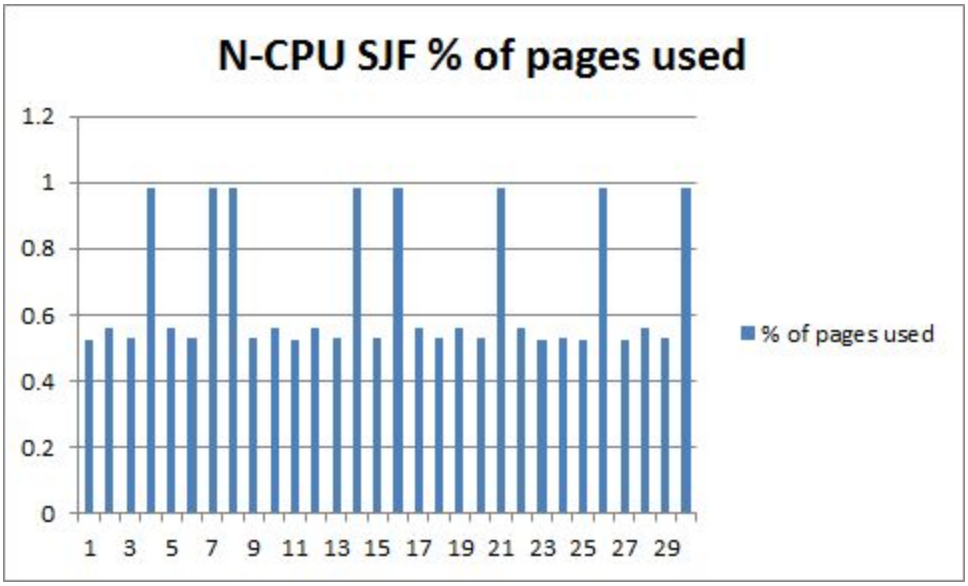

Shortest Job First() Scheduling 8 Core Results:

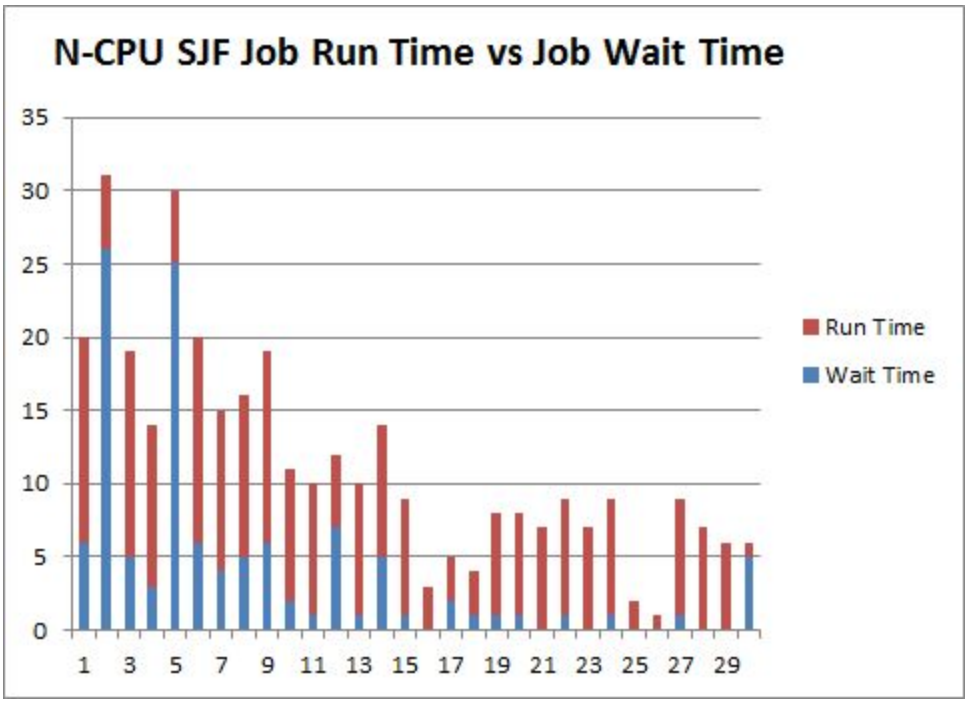
| SYSTEM TIME | JOB # | CPU # | WAIT TIME | RUN TIME | BLOCKS USED / 64 = pages used | I/Os Used |
|---|---|---|---|---|---|---|
| **SJF** | | | | | | |
| 1461184694730 | 4 | 1 | 3 | 11 | 63 | 12 |
| 1461184694731 | 7 | 2 | 4 | 11 | 63 | 12 |
| 1461184694732 | 8 | 3 | 5 | 11 | 63 | 12 |
| 1461184694732 | 14 | 4 | 5 | 9 | 63 | 10 |
| 1461184694732 | 1 | 5 | 6 | 14 | 67 | 12 |
| 1461184694732 | 3 | 6 | 5 | 14 | 68 | 12 |
| 1461184694733 | 6 | 7 | 6 | 14 | 68 | 12 |
| 1461184694733 | 9 | 8 | 6 | 13 | 68 | 12 |
| 1461184694752 | 2 | 1 | 26 | 5 | 72 | 12 |
| 1461184694752 | 5 | 2 | 25 | 5 | 72 | 12 |
| 1461184694753 | 16 | 3 | 0 | 3 | 63 | 5 |

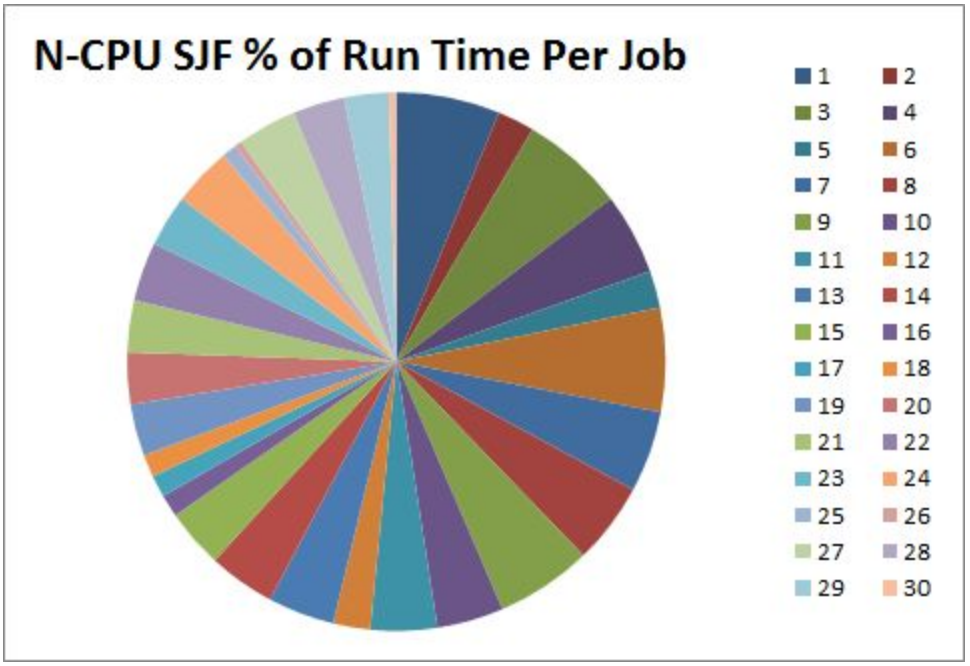
| | | | | | | |
|---|---|---|---|---|---|---|
| 1461184694753 | 21 | 4 | 0 | 7 | 63 | 13 |
| 1461184694753 | 11 | 5 | 1 | 9 | 67 | 12 |
| 1461184694753 | 13 | 6 | 1 | 9 | 68 | 12 |
| 1461184694754 | 15 | 7 | 1 | 8 | 68 | 11 |
| 1461184694754 | 10 | 8 | 2 | 9 | 72 | 12 |
| 1461184694759 | 12 | 1 | 7 | 5 | 72 | 12 |
| 1461184694760 | 26 | 2 | 0 | 1 | 63 | 8 |
| 1461184694761 | 18 | 3 | 1 | 3 | 68 | 12 |
| 1461184694762 | 17 | 4 | 2 | 3 | 72 | 12 |
| 1461184694768 | 30 | 2 | 5 | 1 | 63 | 11 |
| 1461184694769 | 23 | 1 | 0 | 7 | 67 | 11 |
| 1461184694769 | 25 | 3 | 0 | 2 | 67 | 7 |
| 1461184694770 | 27 | 4 | 1 | 8 | 67 | 12 |
| 1461184694770 | 20 | 5 | 1 | 7 | 68 | 12 |
| 1461184694770 | 24 | 6 | 1 | 8 | 68 | 12 |
| 1461184694770 | 19 | 7 | 1 | 7 | 72 | 10 |
| 1461184694770 | 22 | 8 | 1 | 8 | 72 | 12 |
| 1461184694772 | 28 | 2 | 0 | 7 | 72 | 12 |
| 1461184694775 | 29 | 3 | 0 | 6 | 68 | 12 |


SJF Average Times per CPU
18
16
14
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
Average Run Time
Average Wait Time
Average Total Time


SJF Jobs per CPU
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
CPU No.


N-CPU SJF % of pages used
1.2
1
0.8
0.6
0.4
0.2
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
% of pages used


N-CPU SJF Job Run Time vs Job Wait Time
35
30
25
20
15
10
5
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
Run Time
Wait Time




### 1 CPU Core vs. 8 CPU Cores(N-CPU)

Job Time Averages Based on Scheduling Type and CPU Core Amount

| Scheduling Type | CPU Core Amount | Wait Time(ms) | Run Time(ms) | Total Time(ms) |
|---|---|---|---|---|
| Priority | 1 | 29.43 | 2.80 | 32.23 |
| Priority | N | 4.37 | 7.30 | 11.67 |
| FIFO | 1 | 31.00 | 2.93 | 33.93 |
| FIFO | N | 3.63 | 8.57 | 12.20 |
| SJF | 1 | 23.80 | 2.30 | 26.10 |
| SJF | N | 3.87 | 7.50 | 11.37 |

Total Job Time Averages

| Core Type | CPU Core Amount | Wait Time(ms) | Run Time(ms) | Total Time(ms) |
|---|---|---|---|---|
| 1 CPU Core | 1 | 28.07 | 2.67 | 30.74 |
| N-CPU Cores | 8 | 3.96 | 7.79 | 11.75 |

# Conclusions

In terms of scheduling algorithms, we tested three: Priority Scheduling, First In First Out(FIFO), and Shortest Job First(SJF). When comparing the speed of the algorithms, SJF took the least amount of time for jobs to run. The wait time, run time, and total time for the SJF algorithm were faster than both FIFO and Priority on all accounts in our measurements. The next fastest scheduling algorithm was Priority Scheduling, which had the second fastest measurements on all accounts when compared to the other scheduling algorithms except under wait time in the N-CPU condition. FIFO had the second shortest wait time when compared to the other algorithms when looking at the N-CPU cases. FIFO scored as the worst scheduling type when looking at the three we implemented. Overall, the SJF algorithm performed the fastest on average, due to its performance over multiple CPUs.

When comparing the results from running the instruction set with 1 CPU core and the N-CPU cores, the wait time dropped dramatically with extra cores. For all of the wait times across the scheduling algorithms, when run with 8 CPU cores, the time was at most one sixth the wait time of the results found in the single CPU equivalent, a significant margin. Although the wait times were cut by such a large proportion, the run times increased by a factor of at least 2.5. Even though this jump in run time occurs when increasing the amount of cores, the greatly reduced wait time balances the total job time to a fraction of the job time when run on 1 CPU. The best number of CPUs for the OS is around 8 cores, more could increase the speed, but due to the size of our RAM, only 12 CPUs would be loaded at a time. With this in mind, we could run several more CPU cores, but for our allotted RAM this becomes less practical.

Jobs with less I/O ran slightly faster than those with more, but the difference was negligible in the given scale of the project.

To summarize, the fastest scheduling type was Shortest Job First, and the optimal number of CPUs was 8, but any amount up to 12 was about as efficient.