Operating System

Phase 2 Report

Spring 2015 - CS 3243 - Section 001

April 29, 2015

Group Members:

Aman Bhimani

Bradley Richey

Darvius Vu

Donald "Evan" Ross

Mike Gwyn

Table of Contents

0. Introduction

0.1 Goals of Phase II

The main goals of the second phase are as follows:

1. Using the same modules, run the same processes in Shortest Job First (SJF) algorithm
2. Run all 30 jobs with the same algorithms, using a memory management technique called paging - with interrupts on I/O and page faults.

The first requirement is to run a different kind of algorithm, SJF, with 1-CPU and N-CPU operating systems. There are very few changes in the operating system from Phase I to complete this requirement.

The next requirement involves a longer process of paging the hard disk, and framing the memory system (explained in detail later on). This makes the memory (RAM) non-contiguous, and you are able to fit more jobs in memory at the same time. Using this new paging system, we must run all three algorithms on N-CPU and 1-CPU. This will give us enough data to compare to the Phase 1 data and see if the paging system makes sense to implement in our operating system.

After running new algorithms and incorporating them in to our project, and updating our memory management techniques, we must compare the data from Phase 1 and Phase 2 to see how the new memory management system affects our Operating System.

0.2 Hypothesis

a. Step 1: Shortest Job First (SJF)

After discussing with all group members and thinking about the problem, we have hypothesized that the SJF algorithm will not be faster in execution. Since the operating system is the same, and the CPU is the same, and the instructions in the jobs are the same, the execution times will remain similar to other algorithm techniques discussed in Phase 1. However, we do think that the average waiting time for the 30 jobs given will improve. This is because the queues are sorted by their duration of run time, so each job has to wait less.

b. Step 2: Memory Management System (Memory Paging)

We believed that the paging system will not make much of a difference when it comes to waiting times. The wait times for each job, or the average waiting time for all jobs depends on the algorithm used by the Short Term Scheduler.

However, paging system may create some extra overhead during execution time because pages have to be brought into the memory before several instructions. This uses extra time during execution because the memory is full of pages that are of other jobs.

1. Design

1.1 Design Updates

Addition of Paging for single and multiple CPU(s) and incorporation of the SJF.

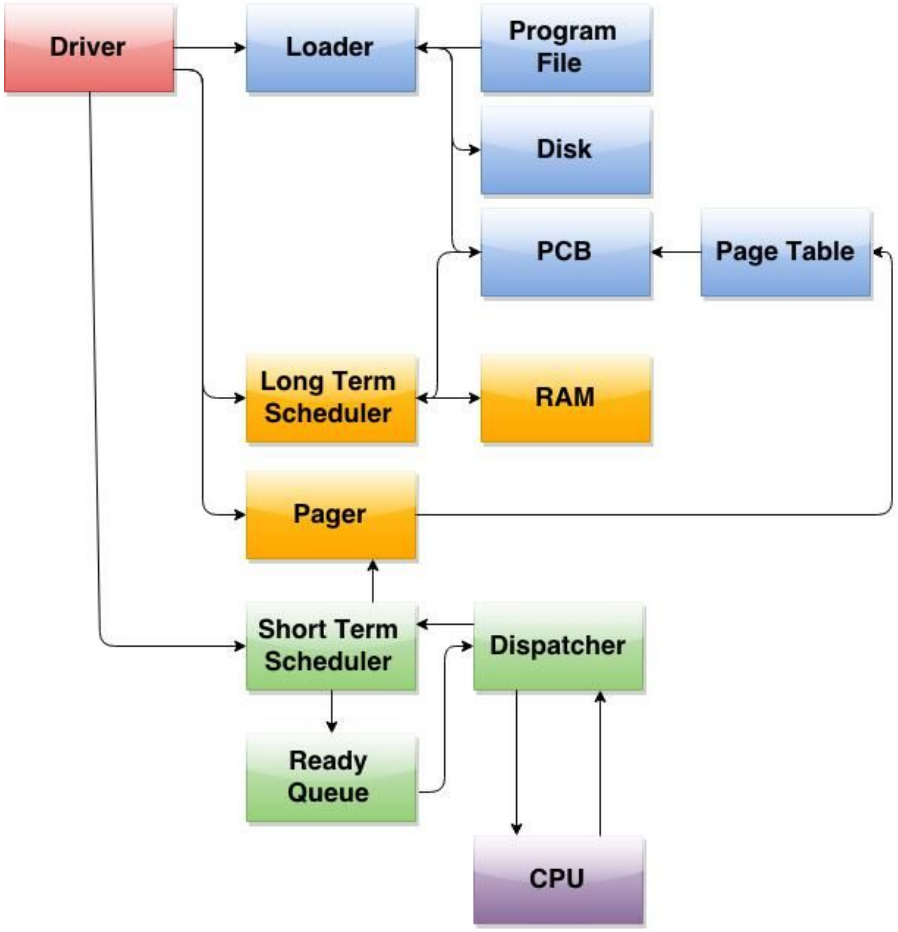
a. Shortest Job First (SJF)

The Shortest Job First is implemented just like the other algorithms in Phase 1. It is run after job priority algorithm. Before we run the Shortest Job First algorithm, we make sure to clear the RAM and important information in PCB.

b. Paging

The paging system is designed with three classes: page, pager, and page table. The page will be a part of the page table, and the page table will be a part of the PCB already implemented in Phase 1.

The Pager class is responsible for managing the page tables and pages in each page table. The responsibility includes things like transferring pages into frames (RAM) and returning the memory address for an instruction.



2. Implementation of Modules

2.1. New Modules in Phase 2

a. Page

This module is an object that is used for all jobs, and is stored in the Page Table in the PCB for each job. Our pages consist of 4 words each. This means that each page has addresses for four words in either disk or memory. Data is not moved around in the disk, but the memory (RAM) addresses need to be updated frequently.

Each page consists of:

1. An array of size 4 for RAM Addresses (integers)
2. An array of size 4 for Disk Addresses (integers)
3. A binary semaphore to keep track of if the page is in memory
4. A long number to keep track of page service fault times (in milliseconds)

When a page is put in memory after a page fault, the page is updated with the correct memory address, and the semaphore is flagged as true to be in memory. When a page is taken out of memory, the RAM addresses are cleared and the semaphore is flagged as false. The page fault service time is also recorded after the page is put into memory; which is the time it takes for the page to be placed into memory.

The page has methods to update and retrieve RAM addresses, disk addresses, page fault service times, and print out the page to an output for debugging purposes.

b. Page Table

The page table resides in the Process Control Block for each process and it contains all the pages for that specific process. No other process is able to get into the page table, and any methods that modify the Page Table or the Pages inside need access to the specific job it is trying to edit.

In the constructor of the Page Table, a new array of the required size is created and the corresponding Disk address array is filled with the correct addresses from the Disk.

There is also a method to return the number of pages in a given Page Table and a method to print the whole page table for debugging purposes.

The way the size of the page table is determine by the size of the job on disk, and the word count. Our word count per page is 4 words, so the page table size would be:

page table size: ***(n / 4) + 1***

Adding 1 depends on if the instruction counts are divisible by 4 or not.

c. Pager

The pager class is a separate entity that manages the page tables for all jobs. The CPU and the Pager interact with each other once the job is in execution. The Pager is responsible for looking up the memory address for each page requested by the CPU and transferring pages into frames. The pager class checks the RAM for free frames, and full frames which are counting semaphores, to see where a page can be put into RAM.

Initially, the pager puts 4 pages per job into memory as frames. The CPU calls a method that returns the memory address for an instruction if it is in a frame. Otherwise, it first puts the page into RAM as a frame, and then returns the memory address for RAM.

Any changes made to memory frames are recorded in the page table by the pager, and other modules can now access the page table from the PCB if needed.

2.2. Modified Modules from Phase 1

a. Driver

There are several changes to the Driver depending on which OS we are running. For example, if we are running 1-cpu with paging system, then we only need to make one CPU. If we are using n-cpu, then we need to create and initialize n number of cpus.

In the beginning of our driver, we have several global variables that can be accessed through the operating system with correct securities. Next, it starts the First In First Out algorithm, then the job priority algorithm, and then the shortest job first algorithm.

Each Algorithm consists of:

1. Clearing the status of PCB of each job, to clear any left overs from previous runs of the operating systems
2. Framing the first four pages from each job into the memory
3. Calling the Short Term Scheduler to start the algorithm and pass jobs to CPU(s)
4. After all jobs are completed, printing the data such as waiting times, execution times to the debugger.

b. PCB Object

From phase one, we have added a process status which includes: new, ready, waiting, running, finished, and error. This keeps track of the process that other modules can refer to when looking at the job. For example, when the job is in the ready queue, the status is changed to “ready” and when the process is on the CPU, the status is changed to “running”.

We have also added the page table, as previously discussed, into the PCB. This table keeps track of all of the instructions and where they are on disk, and whether or not they are in memory. This also includes methods that relate to the page table, such as getting a certain page, and recording page fault service times.

c. PCB

From phase 1, the only thing that has changed in the PCB is the incorporation of the shortest job first algorithm, and modifications to the method that clears status of the all PCB Objects.

There are new variables in the PCB Object, so more variables have to be cleared before all the jobs can run another time with another algorithm - as the Driver requires it to.

d. Long Term Scheduler

The long term scheduler was changed in order to incorporate the shortest job first algorithm. This was only for the contiguous non-paging system for RAM. It can sort the jobs with the one that has the shortest size first, and then put it on the ready queue for the short term scheduler.

e. Short Term Scheduler

Since phase 1, the short term scheduler has changed in order to incorporate the shortest job first algorithm. There were new arraylists made for this algorithm to keep track of waiting and execution times.

We have also modified methods that print waiting and execution times to the console for debugging purposes.

f. Dispatcher

The dispatcher, part of the short term scheduler, has changed in order to incorporate the shortest job first algorithm. It can now use the ready queue, and the waiting and execution time lists appropriately for shortest job first.

g. DMA Channel

The DMA Channel was modified from phase one to include calls to the Pager to make sure that the correct instructions or data are being retrieved from RAM. It passes the job number and the instruction number to the Pager. The Pager then returns the address of the instruction or data. This address can be used to read from the RAM at that location or write to the RAM at that location.

h. CPU

The CPU was changed so that it calls the Pager when it needs to retrieve the next instruction. It passes the job number and the instruction number. (the number corresponding to where the instruction is in the sequence of instructions in Disk) The Pager then returns the address of the instruction and the CPU reads the RAM at that location to make sure that the correct instruction is next.

i. RAM

RAM has significantly changed since phase 1. During phase 1, RAM was only an array of strings, which could hold the instructions or the data. In phase 2, the array of strings has been changed to an array of objects - Ram Slots.

Each Ram Slot consists of:

1. String of data
2. Semaphore for a “dirty” bit
3. semaphore for an “empty” bit
4. Page number where the instruction came from
5. job number where the instruction came from

There are some methods changed in order to accommodate for the changes in the RAM. For example, reading and writing to RAM now has to change the data inside the object of a ram slot.

We also have a new stack data structure, and a new queue data structure. This contains integers from 0 to 255 that could be frames that are empty, or frames that are full. When a page is put into a frame, the stack is popped and the frame number is put into the queue. When the stack is empty, the first item in the queue is removed since it is the oldest frame.

2.3. Unchanged Modules from Phase 1

There are two modules from phase 1 that have not been changed - these are the disk and the loader. The job of the loader is still to put data into the disk, and the disk is still an array of strings. The pages of disk are kept track of in the page table.

3. Simulation of the OS

3.1. Step 1 - Shortest Job First

Step one involved processing the jobs in Shortest job First (SJF) algorithm on 1-CPU and N-CPUs, in our case, 4-CPUs. This step was done without the paging system to be implemented in the future.

All of the algorithms start with the Driver which contains our main method. First, all of the contents of the PCB are cleared that are crucial to execution. Next, the start time is set to the current system time so the start time is set to exactly when the algorithm begins scheduling the jobs. The long term scheduler now loads the jobs according to the shortest job first into the RAM. It cannot load all the jobs into RAM, because RAM is has much less storage than the Disk. It makes a ready queue for the jobs that are ready to be executed. Next, the Short Term Scheduler takes jobs out of the ready queue as they are sorted by the job size, and passes the PCB Object to the CPU.

If the Operating System is 4-CPU architecture there is a switch statement with a count variable that determines which of the CPU's to run the job on. Once the CPU is selected whether it is 4-CPU's or 1-CPU the Dispatcher is called and passes the job to the CPU. It reads all of the instructions from RAM and stores them into a local cache. The Program counter is then set to 0 corresponding with the first instruction stored in the Cache. The CPU then fetches the instruction from the Cache and converts it into a binary string. The binary string is then decoded into multiple substrings corresponding to the Opcode, destination register, source register. Once the Opcode is obtained the instruction is then executed based on the Opcode. This loop is run until the last instruction is reached. The CPU then returns that amount of time that the job was running on the CPU. Since in the 4-CPU architecture the CPU's are clones of each other there is no difference between the 1-CPU and 4-CPU classes.

3.2. Step 2 - Memory Paging (1 CPU and N-CPU)

The memory paging system only affects how the processes are brought into the memory, and how they work during execution. Before the Short Term Scheduler puts each job in the CPU, the initial pages have to be moved into frames in the RAM. This is done by the Driver calling the Pager class. The pager puts 4 frames per job initially into RAM and then passes the control to the Short Term Scheduler. Here, it acts the same way as if it were a non-paging system. The CPU knows which job is on hand. The only difference is that when it is trying to retrieve an instruction or data slot in the RAM, it asks the Pager class for the job and instruction it is trying to get to.

The pager class looks at the PCB and determines if the page is in memory or not. If it is in memory, it retrieves the memory address and passes it on to the CPU. If the page is not in memory as a frame, then this creates a page fault. The page fault is resolved by calling a method in the Pager class that takes a job number, a page number, and a boolean value for debugging purposes. Once it knows which page it has to move into a frame in RAM, it does so, and records the time it takes to do this job.

Now the page requested by the CPU is in memory, and the memory address for the particular instruction is passed to the CPU. During this whole time, the CPU is at halt and it is not processing a job. This creates a huge overhead and a lot of the time, CPU may not be processing a job, but waiting for the page to be in the job. This will definitely increase the execution time for each job, compared to a non-paging system.

For N-CPUs, the only difference is that there are multiple CPUs that take one job each at a time. They do not run simultaneously, so there are no conflicts between the multiple CPUs.

4. Data Results

4.1. Step 1 - Shortest Job First (No Memory Paging)

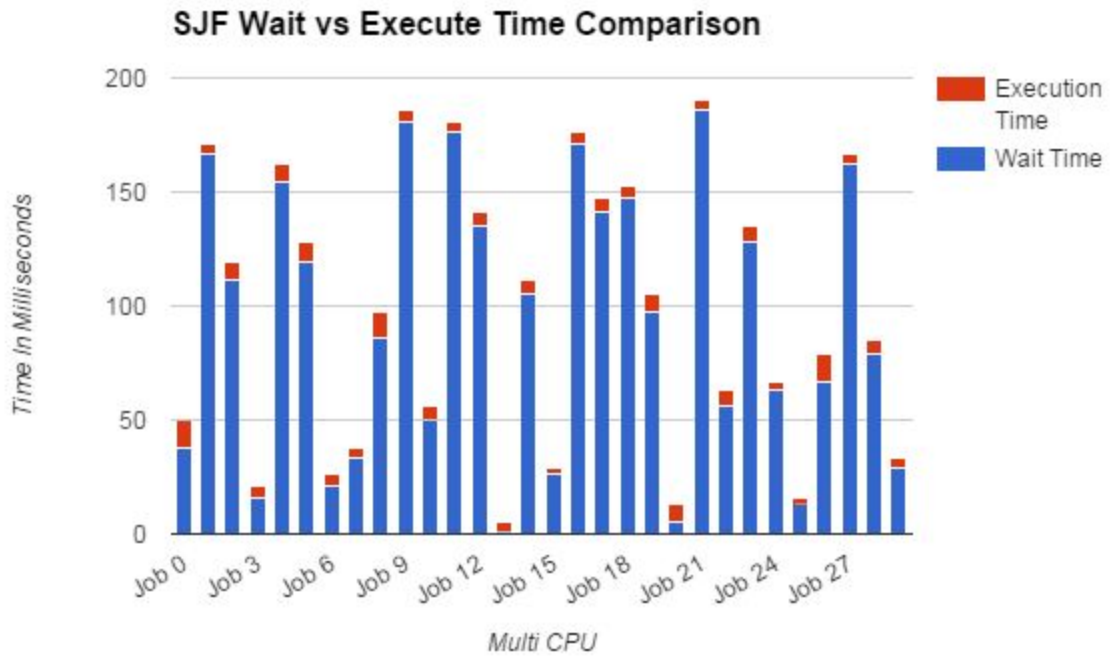
4.1.1. Shortest Job first (1-CPU)

Shortest Job First(SJF) 1 CPU				
job	Wait	Execute	Wait + Execute (ms)	IO Count
1	53	15	68	12
2	210	10	220	12
3	128	10	138	12
4	1	10	11	12
5	220	8	228	12
6	138	9	147	12
7	11	6	17	12
8	17	5	22	12
9	147	7	154	12
10	230	6	236	12
11	68	11	79	12
12	236	10	246	12
13	154	8	162	12
14	23	5	28	12
15	162	8	170	10
16	28	3	31	11
17	246	7	253	5
18	170	9	179	12
19	253	7	260	12
20	179	9	188	10
21	31	6	37	12
22	260	6	266	13
23	79	34	113	12
24	188	14	202	11
25	113	5	118	12
26	37	3	40	7
27	118	10	128	8
28	266	5	271	12
29	202	8	210	12
30	40	13	53	11

4.2.2. Shortest Job First (N-CPU)

Shortest Job First(SJF) N CPU				
Job	Wait	Execute	Wait + Execute (ms)	IO Count
1	38	12	50	12
2	167	4	171	12
3	111	8	119	12
4	16	5	21	12
5	154	8	162	12
6	119	9	128	12
7	21	5	26	12
8	33	5	38	12
9	86	11	97	12
10	181	5	186	12
11	50	6	56	12
12	176	5	181	12
13	135	6	141	12
14	1	4	5	12
15	105	6	111	10
16	26	3	29	11
17	171	5	176	5
18	141	6	147	12
19	147	6	153	12

20	97	8	105	10
21	5	8	13	12
22	186	4	190	13
23	56	7	63	12
24	128	7	135	11
25	63	4	67	12
26	13	3	16	7
27	67	12	79	8
28	162	5	167	12
29	79	6	85	12
30	29	4	33	11

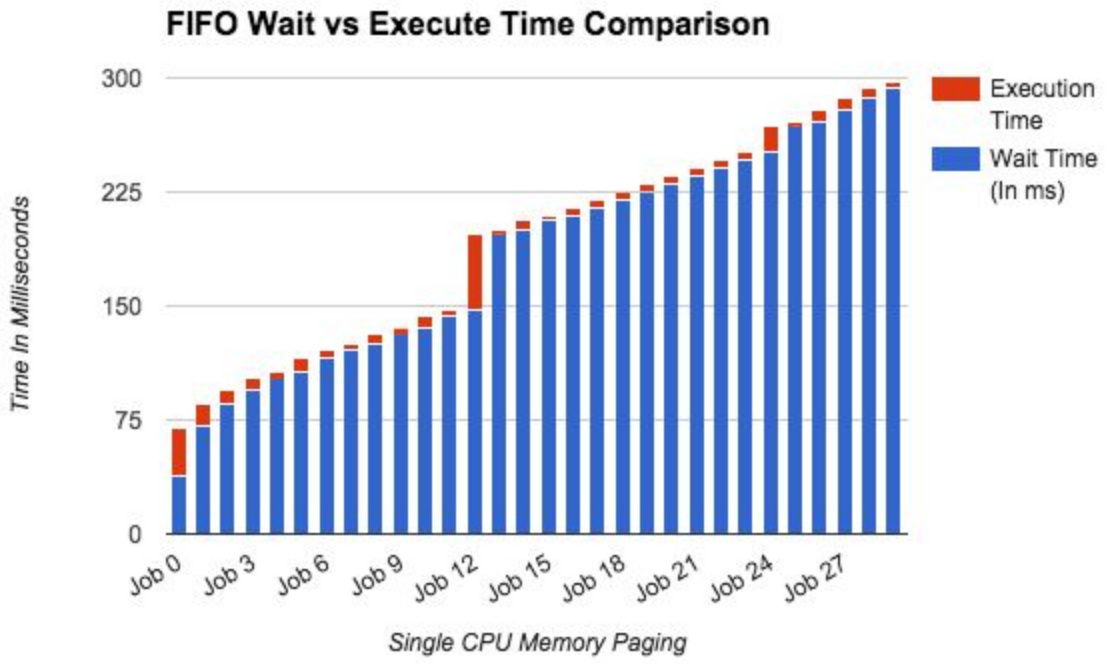


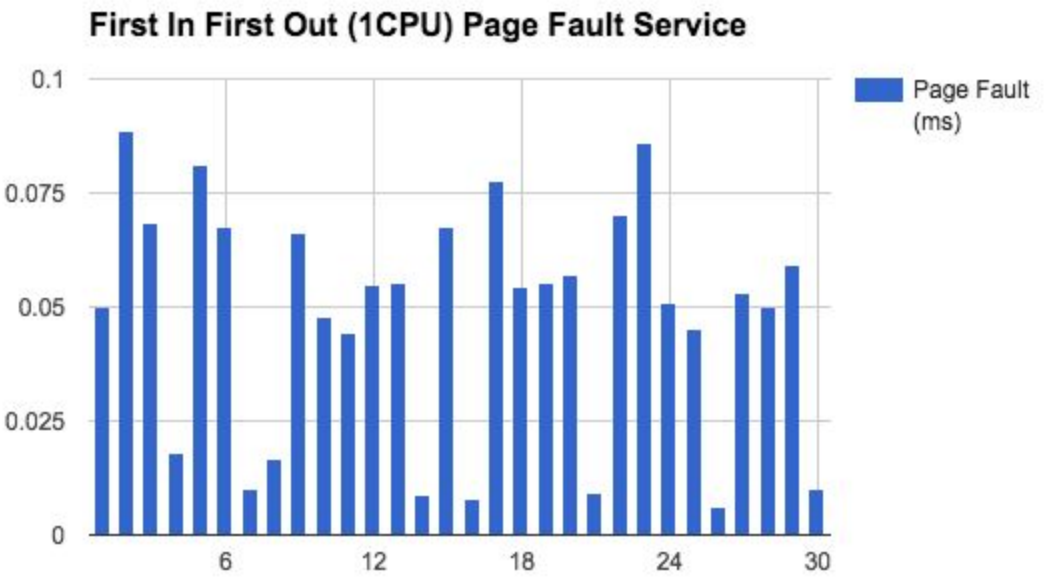
4.2. Step 2 - Memory Paging System (3 Algorithms)

4.2.1. First In First Out (1-CPU)

First in First Out (FIFO) 1 CPU						
Job	**Page Fault (ms)**	**Page Faults**	**Wait (ms)**	**Execute (ms)**	**Wait +Execute (ms)**	**IO Count**
1	0.049914	3	38	32	70	12
2	0.088437	5	71	15	86	12
3	0.06848	4	86	9	95	12
4	0.017766	1	95	7	102	12
5	0.081002	5	102	5	107	12

6	0.067402	4	107	9	116	12
7	0.010095	1	116	5	121	12
8	0.016675	1	121	4	125	12
9	0.06617	4	125	6	131	12
10	0.047646	5	131	5	136	12
11	0.044196	3	136	7	143	12
12	0.054971	5	143	4	147	12
13	0.055246	4	147	50	197	12
14	0.008912	1	197	3	200	12
15	0.067592	4	200	6	206	10
16	0.00788	1	206	3	209	11
17	0.077802	5	209	5	214	5
18	0.054317	4	214	6	220	12
19	0.05536	5	220	5	225	12
20	0.056987	4	225	5	230	10
21	0.00935	1	230	6	236	12
22	0.070212	5	236	5	241	13
23	0.086109	3	241	5	246	12
24	0.050764	4	246	5	251	11
25	0.044992	3	251	17	268	12
26	0.006104	1	268	3	271	7
27	0.053107	3	271	8	279	8
28	0.050114	5	279	8	287	12
29	0.059154	4	287	6	293	12
30	0.010168	1	293	4	297	11

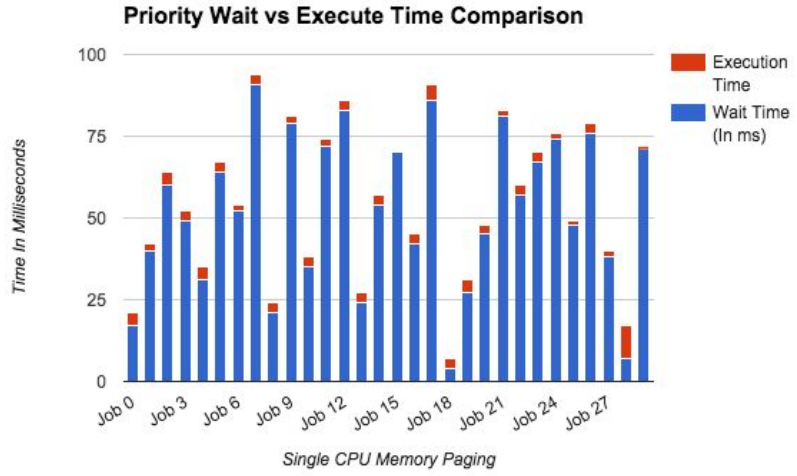


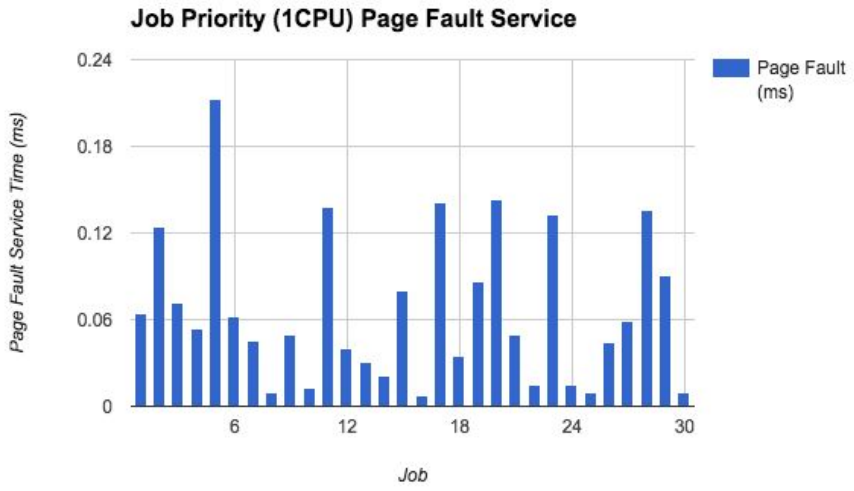


4.2.2. Job Priority (1-CPU)

Priority 1 CPU						
Job	Page Fault (ms)	Page Faults	Wait (ms)	Execute (ms)	Wait +Execute (ms)	IO Count
1	0.064343	7	17	4	21	12
2	0.123871	9	40	2	42	12
3	0.071303	8	60	4	64	12
4	0.0541	5	49	3	52	12
5	0.213063	9	31	4	35	12
6	0.061789	8	64	3	67	12
7	0.045029	5	52	2	54	12
8	0.009347	5	91	3	94	12
9	0.049294	8	21	3	24	12
10	0.012556	9	79	2	81	12
11	0.13751	7	35	3	38	12
12	0.040368	9	72	2	74	12
13	0.030891	8	83	3	86	12
14	0.020899	5	24	3	27	12
15	0.079875	8	54	3	57	10
16	0.006884	5	70	0	70	11
17	0.140939	9	42	3	45	5
18	0.034413	11	86	5	91	12
19	0.086073	9	4	3	7	12
20	0.143309	8	27	4	31	10

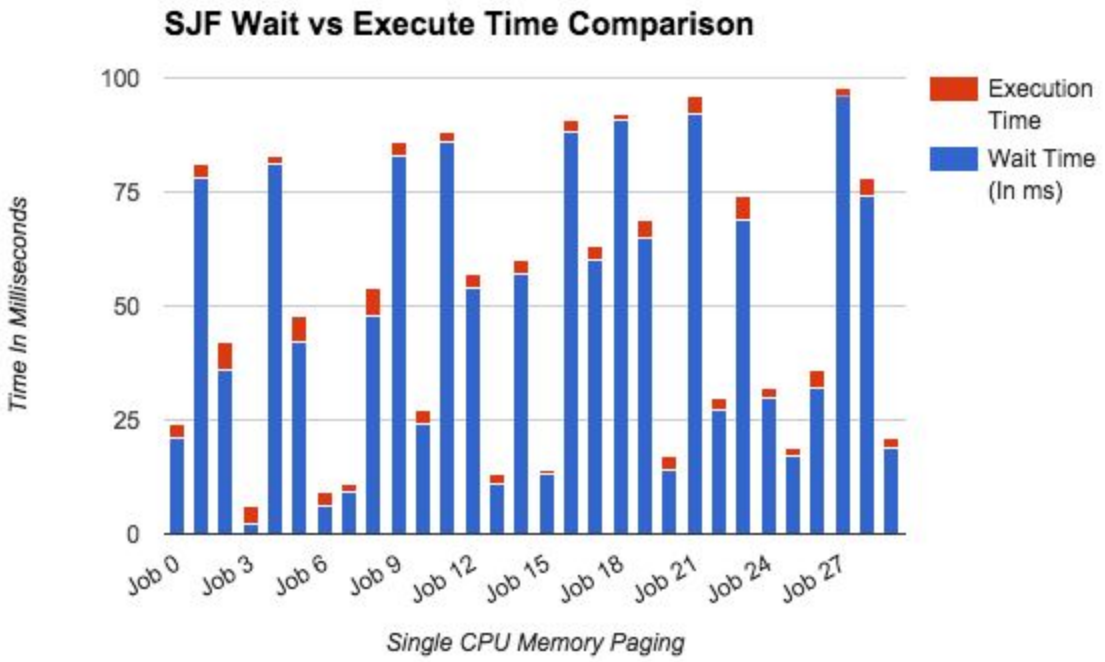
21	0.049054	5	45	3	48	12
22	0.014873	9	81	2	83	13
23	0.133127	7	57	3	60	12
24	0.014532	8	67	3	70	11
25	0.009095	7	74	2	76	12
26	0.044052	5	48	1	49	7
27	0.059014	7	76	3	79	8
28	0.136063	9	38	2	40	12
29	0.090418	4	7	10	17	12
30	0.009024	5	71	1	72	11

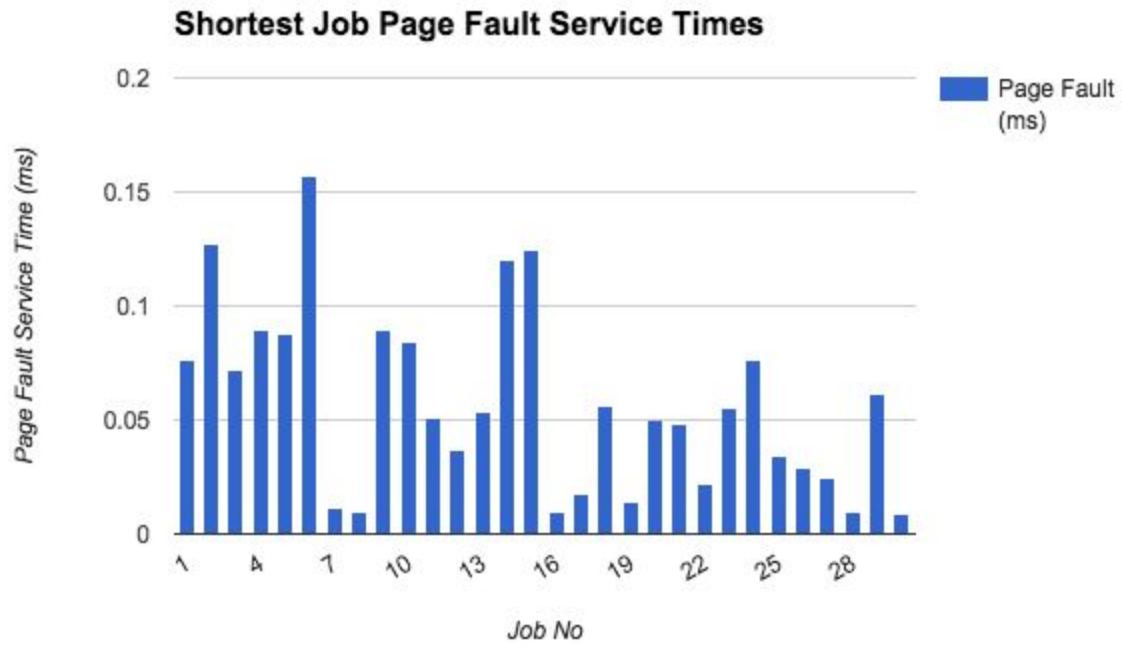




4.2.3 Shortest Job first (1-CPU)

Shortest Job First (SJF) 1 CPU						
Job	Page Fault (ms)	Page Faults	Wait (ms)	Execute (ms)	Wait + Execute (ms)	IO Count
1	0.075946	7	21	3	24	12
2	0.127237	9	78	3	81	12
3	0.071734	8	36	6	42	12
4	0.089253	1	2	4	6	12
5	0.087392	9	81	2	83	12
6	0.156857	8	42	6	48	12
7	0.011118	1	6	3	9	12
8	0.009656	1	9	2	11	12
9	0.089117	8	48	6	54	12
10	0.083893	9	83	3	86	12
11	0.050452	7	24	3	27	12
12	0.037181	9	86	2	88	12
13	0.053385	4	54	3	57	12
14	0.12047	5	11	2	13	12
15	0.124817	8	57	3	60	10
16	0.00995	1	13	1	14	11
17	0.017818	9	88	3	91	5
18	0.056479	4	60	3	63	12
19	0.014122	5	91	1	92	12
20	0.050436	4	65	4	69	10
21	0.04848	5	14	3	17	12
22	0.021943	9	92	4	96	13
23	0.055272	7	27	3	30	12
24	0.076445	8	69	5	74	11
25	0.033945	3	30	2	32	12
26	0.028833	5	17	2	19	7
27	0.024923	3	32	4	36	8
28	0.009763	5	96	2	98	12
29	0.061529	4	74	4	78	12
30	0.009034	1	19	2	21	11

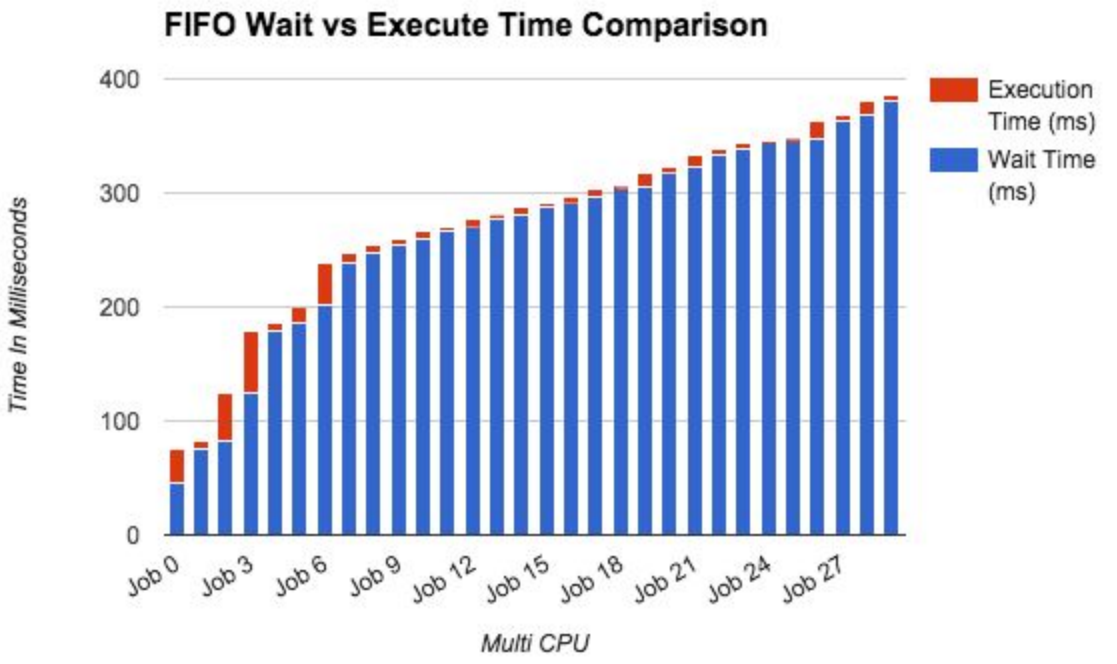

SJF Wait vs Execute Time Comparison
Time In Milliseconds
100
75
50
25
0
Job 0
Job 3
Job 6
Job 9
Job 12
Job 15
Job 18
Job 21
Job 24
Job 27
Single CPU Memory Paging
Execution Time
Wait Time (In ms)

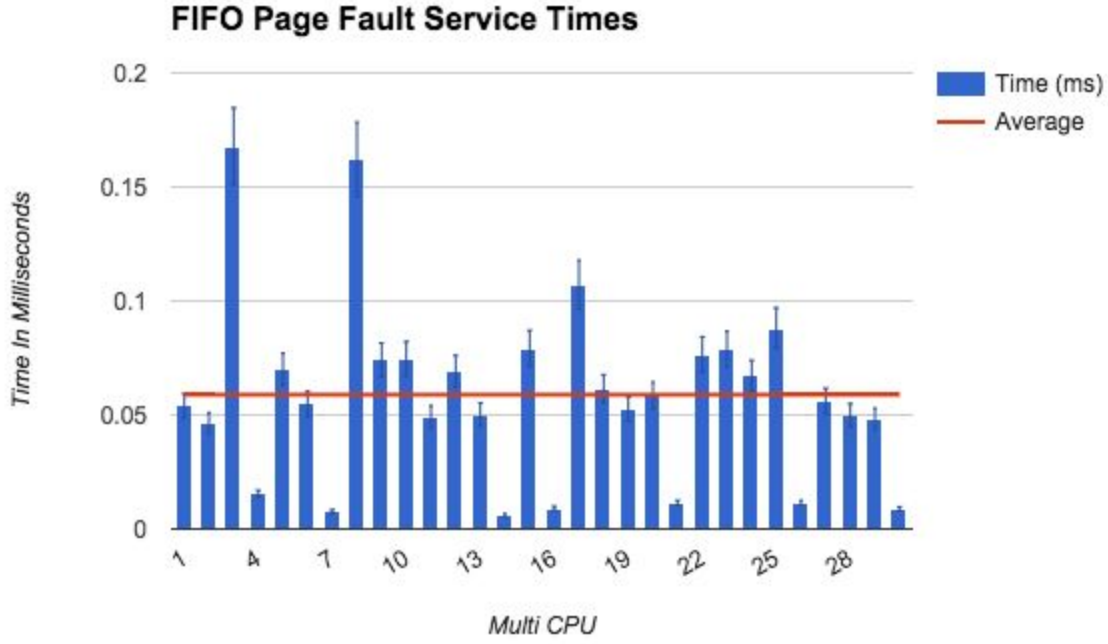

Shortest Job Page Fault Service Times
Page Fault Service Time (ms)
0.2
0.15
0.1
0.05
0
1
4
7
10
13
16
19
22
25
28
Job No
Page Fault (ms)

‘

4.2.4. First In First Out (N-CPU)

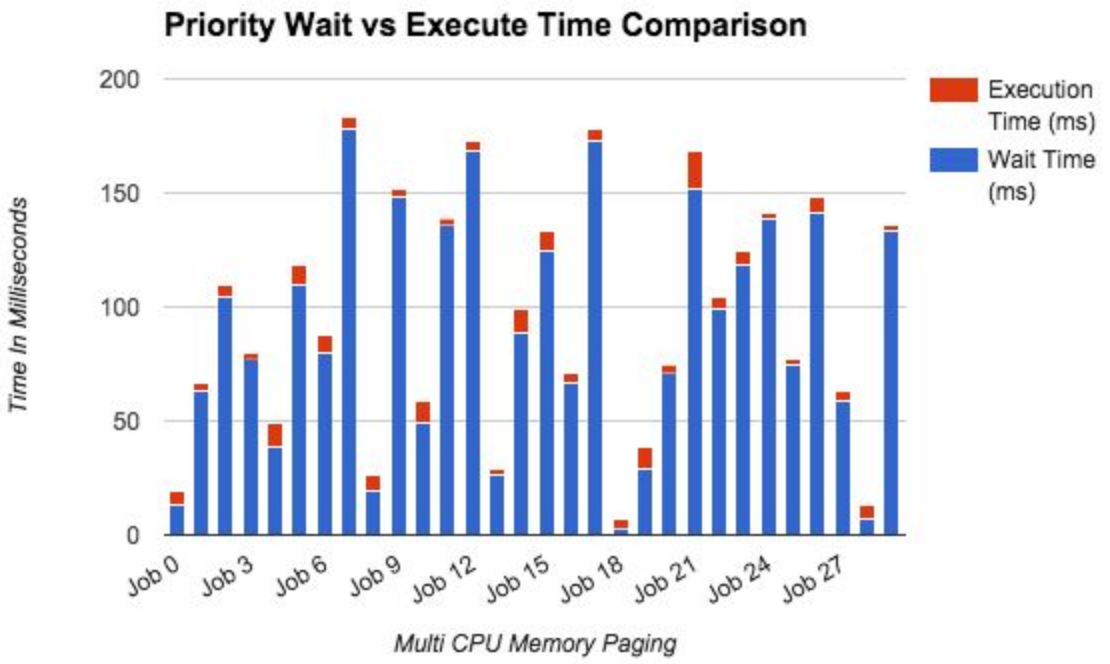
First in First Out (FIFO) N CPU						
Job	Page Fault (ms)	Page Faults (#)	Wait (ms)	Execute (ms)	Wait +Execute (ms)	IO Count
1	0.053975	3	45	30	75	12
2	0.046403	5	76	6	82	12
3	0.167917	4	82	43	125	12
4	0.015426	1	125	54	179	12
5	0.070065	5	179	7	186	12
6	0.054852	4	186	14	200	12
7	0.007667	1	201	38	239	12
8	0.162229	1	239	8	247	12
9	0.074172	4	247	8	255	12
10	0.074701	5	255	4	259	12
11	0.049121	3	259	7	266	12
12	0.069208	5	266	4	270	12
13	0.050272	4	270	8	278	12
14	0.006056	1	278	3	281	12
15	0.079181	4	281	6	287	10
16	0.009074	1	287	4	291	11
17	0.107155	5	291	6	297	5
18	0.061516	4	297	6	303	12
19	0.052561	5	303	3	306	12
20	0.058611	4	306	12	318	10
21	0.011397	1	318	5	323	12
22	0.076636	5	323	10	333	13
23	0.078835	3	333	5	338	12
24	0.067242	4	338	5	343	11
25	0.088143	3	343	2	345	12
26	0.011416	1	345	3	348	7
27	0.056112	3	348	16	364	8
28	0.049948	5	364	4	368	12
29	0.048161	4	368	13	381	12
30	0.008852	1	381	5	386	11

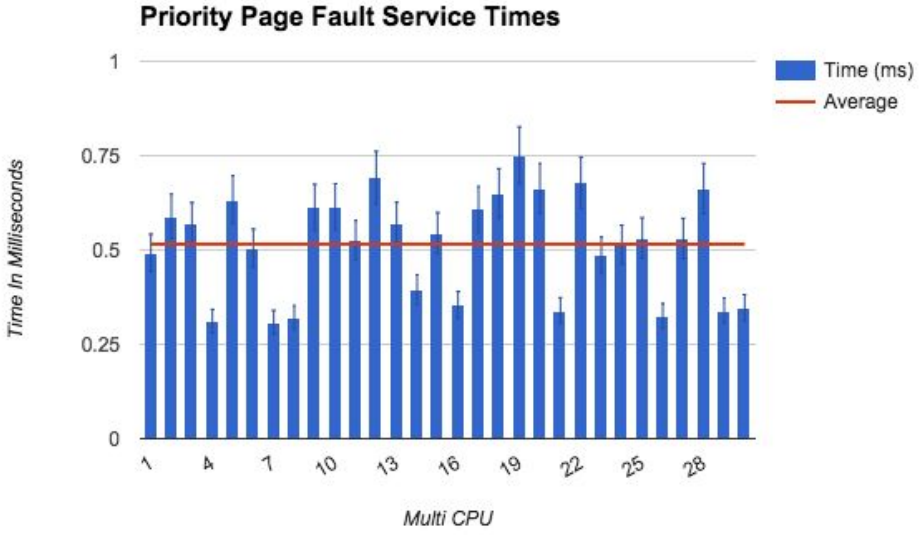

FIFO Wait vs Execute Time Comparison
400
300
200
100
0
Time In Milliseconds
Job 0
Job 3
Job 6
Job 9
Job 12
Job 15
Job 18
Job 21
Job 24
Job 27
Multi CPU
Execution Time (ms)
Wait Time (ms)


FIFO Page Fault Service Times
0.2
0.15
0.1
0.05
0
Time In Milliseconds
1
4
7
10
13
16
19
22
25
28
Multi CPU
Time (ms)
Average

4.2.5. Job Priority (N-CPU)

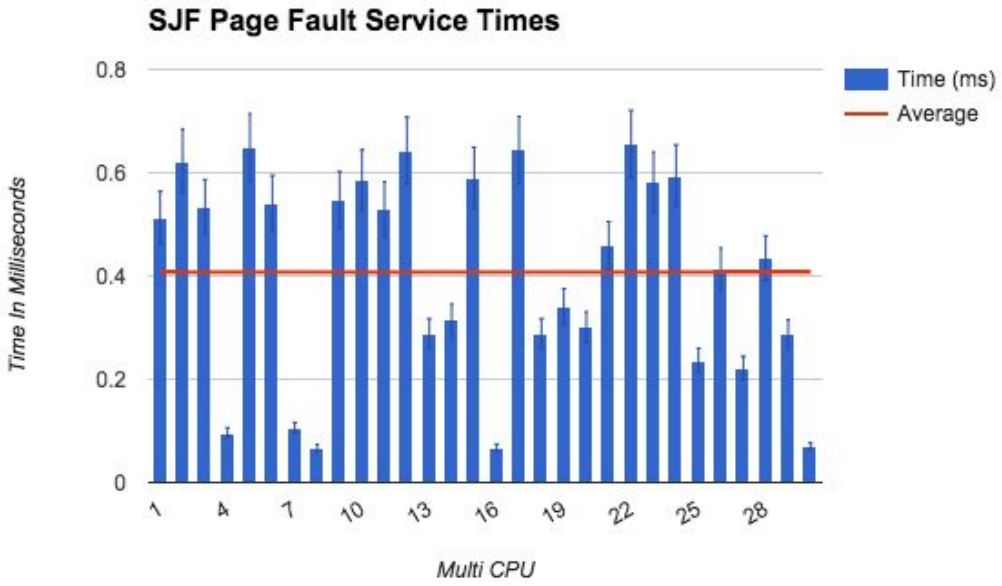
Priority N CPU						
Job	Page Fault (ms)	Page Faults	Wait (ms)	Execute (ms)	Wait + Execute (ms)	IO Count
1	0.49273	7	13	6	19	12
2	0.589065	9	63	4	67	12
3	0.568904	8	104	6	110	12
4	0.311345	5	77	3	80	12
5	0.633635	9	39	10	49	12
6	0.505056	8	110	8	118	12
7	0.309099	5	80	8	88	12
8	0.320857	5	178	5	183	12
9	0.613044	8	19	7	26	12
10	0.614074	9	148	4	152	12
11	0.52543	7	49	10	59	12
12	0.692254	9	136	3	139	12
13	0.569581	8	168	5	173	12
14	0.394919	5	26	3	29	12
15	0.544576	8	89	10	99	10
16	0.354473	5	125	8	133	11
17	0.607732	9	67	4	71	5
18	0.650483	11	173	5	178	12
19	0.751243	9	3	4	7	12
20	0.663297	8	29	10	39	10
21	0.339437	5	71	4	75	12
22	0.67825	9	152	16	168	13
23	0.485988	7	99	5	104	12
24	0.514507	8	118	7	125	11
25	0.532264	7	139	2	141	12
26	0.32558	5	75	2	77	7
27	0.530868	7	141	7	148	8
28	0.66295	9	59	4	63	12
29	0.338859	4	7	6	13	12
30	0.346928	5	133	3	136	11

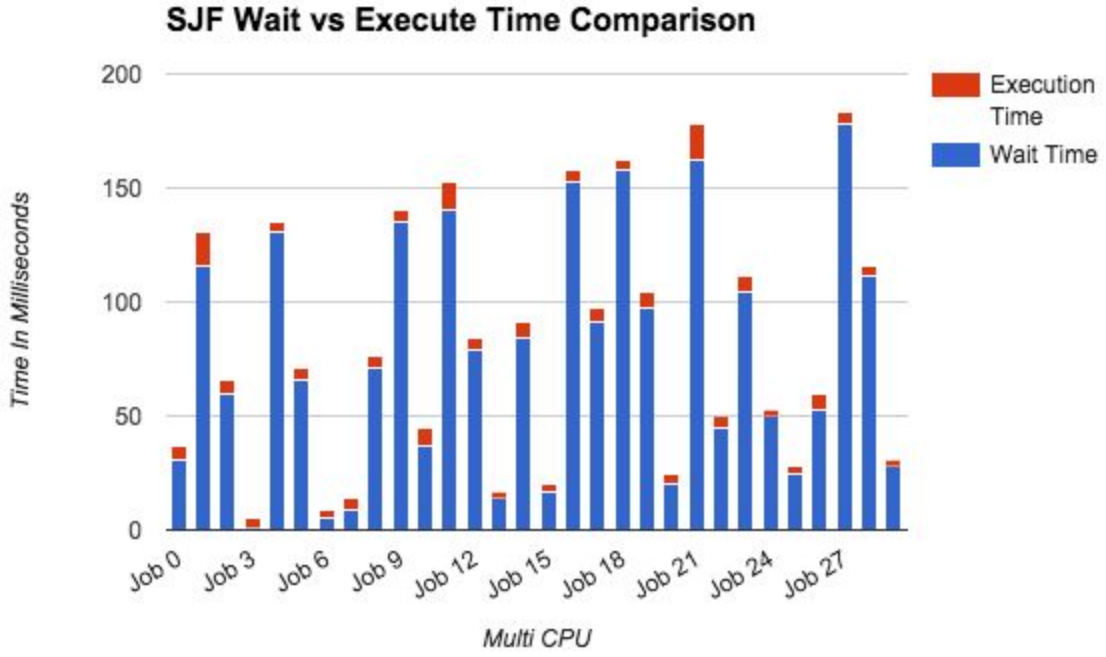

Priority Wait vs Execute Time Comparison
200
150
100
50
0
Time In Milliseconds
Job 0
Job 3
Job 6
Job 9
Job 12
Job 15
Job 18
Job 21
Job 24
Job 27
Multi CPU Memory Paging
Execution Time (ms)
Wait Time (ms)


Priority Page Fault Service Times
1
0.75
0.5
0.25
0
Time In Milliseconds
1
4
7
10
13
16
19
22
25
28
Multi CPU
Time (ms)
Average

4.2.6. Shortest Job First (N-CPU)

Shortest Job First (SJF) N CPU						
Job	Page Fault (ms)	Page Faults	Wait (ms)	Execute (ms)	Wait + Execute (ms)	IO Count
1	0.512752	7	31	6	37	12
2	0.622694	9	116	15	131	12
3	0.533301	8	60	6	66	12
4	0.096425	1	1	4	5	12
5	0.649276	9	131	4	135	12
6	0.540606	8	66	5	71	12
7	0.105697	1	5	4	9	12
8	0.067077	1	9	5	14	12
9	0.547489	8	71	5	76	12
10	0.586029	9	135	5	140	12
11	0.529527	7	37	8	45	12
12	0.643415	9	140	13	153	12
13	0.288554	4	79	5	84	12
14	0.314245	5	14	3	17	12
15	0.590341	8	84	7	91	10
16	0.067594	1	17	3	20	11
17	0.64436	9	153	5	158	5
18	0.28842	4	91	6	97	12
19	0.341503	5	158	4	162	12
20	0.300816	4	97	7	104	10
21	0.459275	5	20	5	25	12
22	0.655278	9	162	16	178	13
23	0.581896	7	45	5	50	12
24	0.594522	8	104	7	111	11
25	0.236394	3	50	3	53	12
26	0.413213	5	25	3	28	7
27	0.222057	3	53	7	60	8
28	0.434246	5	178	5	183	12
29	0.286227	4	111	5	116	12
30	0.069713	1	28	3	31	11


SJF Page Fault Service Times
0.8
0.6
0.4
0.2
0
Time In Milliseconds
1
4
7
10
13
16
19
22
25
28
Multi CPU
Time (ms)
Average


SJF Wait vs Execute Time Comparison
200
150
100
50
0
Time In Milliseconds
Job 0
Job 3
Job 6
Job 9
Job 12
Job 15
Job 18
Job 21
Job 24
Job 27
Multi CPU
Execution Time
Wait Time

4.2.7. Summary of All Averages

Averages						
Scheduling Policy	**CPU**	**Memory Paging (Y/N)**	**Wait (ms)**	**Execute (ms)**	**Page Fault (ms)**	**Total**
Shortest Job First(SJF)	1	No	159.600	8.900	N/A	168.500
Shortest Job First(SJF)	N	No	92.100	6.233	N/A	98.333
First in First Out (FIFO)	1	Yes	183.033	8.600	1.437	193.070
Priority	1	Yes	52.167	2.967	0.066	55.200
Shortest Job First(SJF)	1	Yes	47.500	3.133	0.057	50.690
First in First Out (FIFO)	N	Yes	264.467	11.300	0.059	275.826
Priority	N	Yes	89.667	5.967	0.516	96.149
Shortest Job First(SJF)	N	Yes	75.700	5.967	0.407	82.074

4.2.8. RAM Usage

SJF (1-CPU): 95.21%
SJF (N-CPU): 98.93%

RAM Usage:	**1-CPU (Paging)**	**N-CPU (Paging)**
FIFO	92.58%	92.58%
Priority	100%	100%
SJF	100%	100%

4.2.9. Total Page Faults

Page Faults	**1-CPU (Paging)**	**N-CPU (Paging)**
FIFO	99	99
Priority	218	218
SJF	167	167

4.2.10. Jobs per CPU

FIFO	
CPU	Job Count
CPU 1	8
CPU 2	8
CPU 3	7
CPU 4	7

Priority	
CPU	Job Count
CPU 1	8
CPU 2	8
CPU 3	7
CPU 4	7

SJF	
CPU	Job Count
CPU 1	8
CPU 2	8
CPU 3	7
CPU 4	7

5. Conclusions

5.1 Step 1 - Shortest Job First non Paging

SJF is the fastest of the available scheduling algorithms, because it takes into account the actual size of the job and makes sure that the shortest is running first. The First Come First Serve algorithm simply takes the jobs in the order that they appear in but, this order could lead to a very inefficient order of running the jobs. The Priority scheduling runs the jobs based on the Priority number which is determined before runtime. Because of this we have no way of knowing if the order that Priority scheduling will run the jobs in is the most efficient, and based on the data we see this is true.

5.2 Step 2 - Paging vs Continuous

In Step two of Phase two we tested the efficiency of adding a paging system to our Operating System. We measured the efficiency by the time it took for the jobs to execute and wait, and by the percentage RAM usage of each of the Memory Systems.

The first metric used was the time required for each for the jobs to execute and the amount of time these jobs had to wait to execute. In certain cases, depending on the algorithm, the paged system was slower to execute jobs and in some cases it was faster. We have come to the conclusion that the Java Virtual Machine was responsible for the ambiguous results in our execution times.

The second metric we used to determine whether or not a paged memory system is more efficient than a contiguous memory system is the percentage of RAM allocated to each of the jobs. In a contiguous memory system less RAM is used per job compared to a paged memory system. In a paged memory system Instead of clearing the RAM when 100% is being utilized, the oldest page is swapped with the page that is going to be added. By doing this we ensure that 100% of the RAM is being utilized as often as possible.

Based on these two metrics we have determined that the Paged Memory System is more efficient than the Contiguous one. While it is not generally faster, it allows more efficiency in memory usage.

6. Appendix (Phase 1 Fixes)

1. Design of the Operating System

- Received all points on Design Portion of the Report.

2. Implementation of Modules

2.1. The Driver

One thing we missed talking about in the in the Driver class is the position of the start time of the OS. We chose having the startTime begin on O.S start, because the user has to wait for the O.S to boot, to run the first job. This would mean that there was some time involved in waiting for the first program to run, but not extra time spent on the ready queue.

The Driver class is responsible for initializing variables to be used during the OS. These include things like short term loader, long term loader, and start time. It then proceeds to clear all data in the PCB and RAM to start the first algorithm. Each algorithm goes through this process:

1. RAM is cleared
2. Long-term loader is called to load processes into RAM
3. Short-term loader is called to start the algorithm
4. Dispatcher is called to dispatch jobs onto CPU
5. CPU Runs all jobs as required by the Short Term Loader and Dispatcher

It runs FIFO first, then proceeds to run jobs by their priority. It is also responsible for printing RAM Usage, disk usage, and execution, and wait times.

2.2 Loader Class

No further clarifications were needed for this module

2.3 Long Term Scheduler

The long term scheduler has two methods: one that loads the jobs in the ready queue and memory and a method that puts a job in memory. The driver calls the long term scheduler and gives it the algorithm that it needs to run as an argument. It then loads jobs into RAM and Ready Queue based on that algorithm.

The second method is used to put jobs into memory. For example, if the next job in the ready queue is not in memory, then this method is called in order to put the job into memory.

2.4 Short Term Scheduler

The Short Term Scheduler makes sure that the CPU executes each and every job based on the chosen scheduling algorithm. It is also responsible for putting each job in a ready queue. and then from the ready queue. The Dispatcher method is then called which allow the programs to run on the CPU. The Short Term Scheduler is also responsible for printing the waiting times, and execute times for each job. This is done through objects called “waitTimes”, and “executeTimes” which has the attributes - “jobNumber” and “waitTime”. The wait time is a long value that holds the time in milliseconds. The two times are added together to get the completion time for each job.

2.5 Dispatcher (Not discussed in Phase 1 Report)

The Dispatcher Method is the method that the Short Term Scheduler calls to run each job on the CPU or many CPU's. It calls the loadCPU method and adds the returned executeTime into an ArrayList of executeTimes. The order that the jobs are run in is determined by the chosen scheduling algorithm, and it calls a method to sort the ready queue. Once the ready queue is sorted the jobs are taken off one by one and run on the CPU for the one CPU O.S. For the 4-CPU O.S a count variable is used to determine which of the four CPU's to run the job on. The Dispatcher is the only Module that required significant change to allow multiple CPU's to run.

2.6 CPU

The CPU class is split up into four methods and one Default Constructor. It also sets up an array of 16 registers to be used for storing and manipulating values in the other methods of the CPU class. The first method of the CPU class is the loadCpu method. This method starts by loading the CPU's Cache with all of the needed instructions and data from RAM.

The CPU has the following methods

1. loadCpu - loading the jobs initially into CPU
2. effectiveAddress - returns the effective address of the instruction
3. fetch - returns the instruction
4. decode - decodes the instruction so the execute method can recognize it
5. execute - executes the instruction

2.7 DMA Channel (Not discussed in Phase 1 Report)

The DMA Channel module allows the CPU to access RAM to write data to it, and read data from the Cache. It is split into two methods Read, and Write. Read, reads from the Cache at the address passed into it. The Write methods writes to the address in RAM passed to the DMA Channel.

3. Simulation

Step one involved processing the jobs in First Come First Serve, and Priority algorithms on 1-CPU and N-CPUs, in our case, 4-CPUs. All of the algorithms start with the Driver which contains our main method. First, all of the contents of the PCB are cleared that are crucial to execution.

Next, the start time is set to the current system time so the start time is set to exactly when the algorithms begin scheduling the jobs. The Long Term Scheduler now loads the jobs according to the order the jobs are entered in the disk, or based on the priority that are included with the jobs themselves. It cannot load all the jobs into RAM, because RAM is has much less storage than the Disk. It makes a ready queue for the jobs that are ready to be executed.

Next, the Short Term Scheduler takes jobs out of the ready queue as they are sorted FCFS, or Priority and passes the PCB Object to the CPU. If the Operating System is 4-CPU architecture there is a switch statement with a count variable that determines which of the CPU's to run the job on.

Once the CPU is selected whether it is 4-CPU's or 1-CPU the Dispatcher is called and passes the job to the CPU. It reads all of the instructions from RAM and stores them into a local cache. The Program counter is then set to 0 corresponding with the first instruction stored in the Cache. The CPU then fetches the instruction from the Cache and converts it into a binary string. The binary string is then decoded into multiple substrings corresponding to the Opcode, destination register, source register. Once the Opcode is obtained the instruction is then executed based on the Opcode. This loop is run until the last instruction is reached.

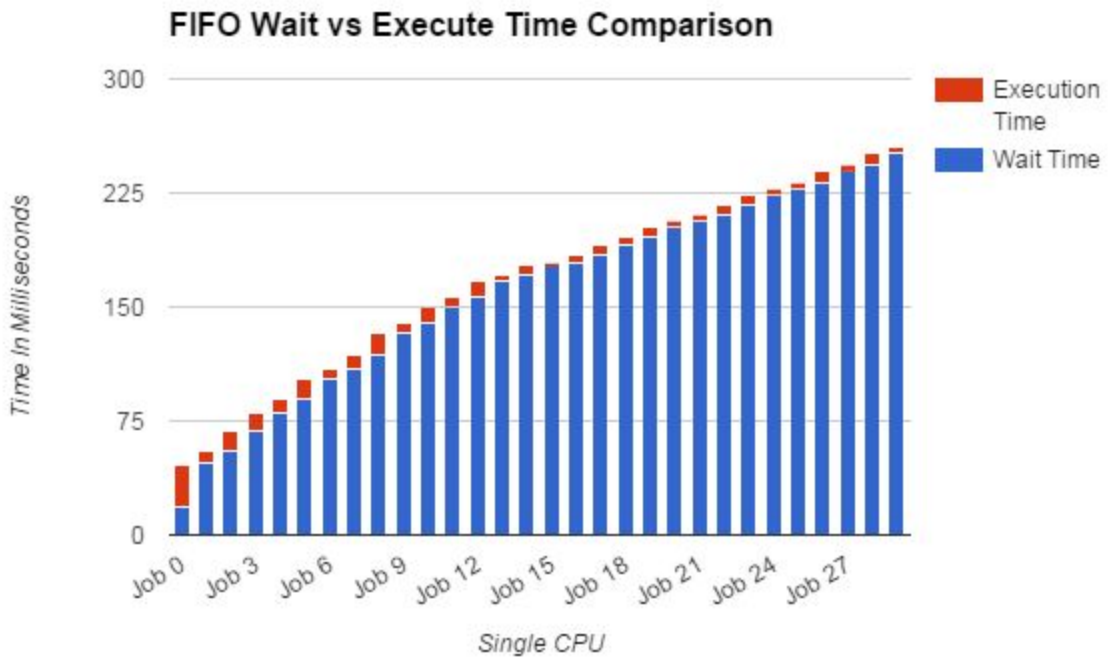
The CPU then returns that amount of time that the job was running on the CPU. Since in the 4-CPU architecture the CPU's are clones of each other there is no difference between the 1-CPU and 4-CPU classes.

4. Data

4.1 First In First Out (1-CPU)

First in First Out(FIFO) 1 CPU				
Job	**Wait**	**Execute**	**Wait + Execute (ms)**	**IO Count**
1	19	27	46	12
2	47	8	55	12
3	55	13	68	12
4	68	12	80	12
5	80	9	89	12
6	89	13	102	12
7	102	7	109	12
8	109	9	118	12
9	118	15	133	12
10	133	7	140	12
11	140	10	150	12
12	150	7	157	12
13	157	10	167	12
14	167	4	171	12
15	171	6	177	10
16	177	2	179	11
17	179	5	184	5
18	184	7	191	12
19	191	5	196	12
20	196	6	202	10
21	203	4	207	12
22	207	4	211	13
23	211	6	217	12
24	217	7	224	11
25	224	4	228	12
26	228	4	232	7
27	232	7	239	8

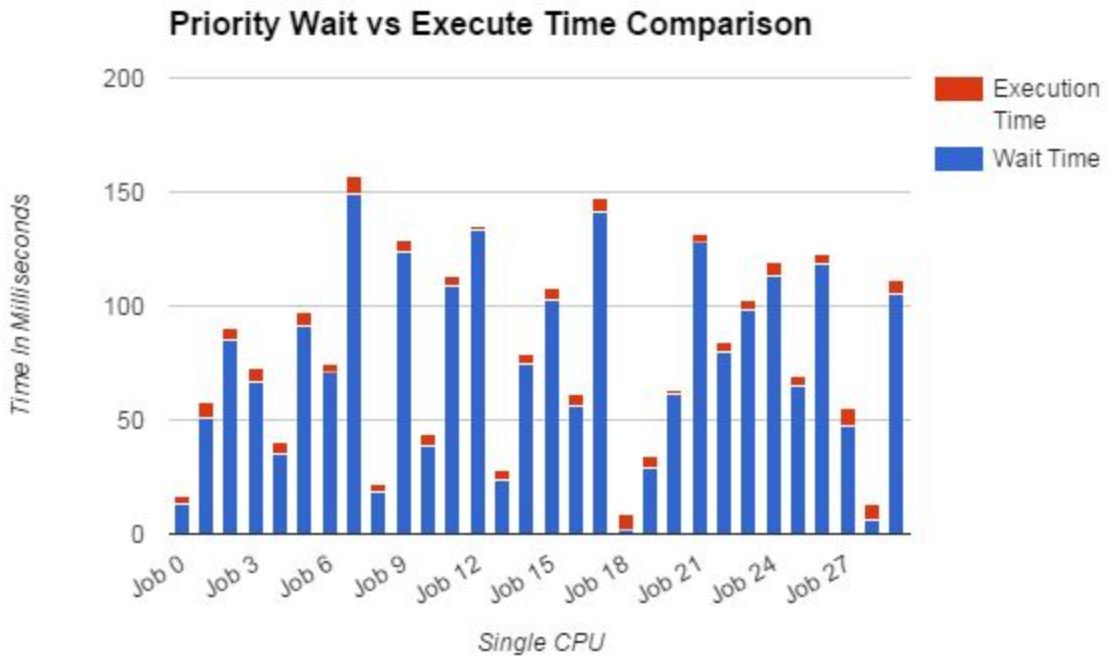
28	239	5	244	12
29	244	7	251	12
30	251	4	255	11



4.2 Job Priority (1-CPU)

Priority 1 CPU				
Job	**Wait**	**Execute**	**Wait + Execute (ms)**	**IO Count**
1	13	4	17	12
2	51	7	58	12
3	85	5	90	12
4	67	6	73	12
5	35	5	40	12
6	91	6	97	12
7	71	4	75	12
8	149	8	157	12
9	18	4	22	12
10	124	5	129	12
11	39	5	44	12
12	109	4	113	12
13	133	2	135	12
14	24	4	28	12
15	75	4	79	10
16	103	5	108	11

17	56	5	61	5
18	141	6	147	12
19	2	7	9	12
20	29	5	34	10
21	61	2	63	12
22	128	4	132	13
23	80	4	84	12
24	98	5	103	11
25	113	6	119	12
26	65	4	69	7
27	118	5	123	8
28	47	8	55	12
29	6	7	13	12
30	105	6	111	11



5. Summary of Findings and Conclusions

After analyzing the data obtained by running our O.S simulation we have come to several conclusions.

The first of these conclusions was that the fastest of the scheduling algorithms was Priority. We believe this is the case because in First Come First Serve the jobs are run in the order that they are loaded into the Disk in, but Priority runs them based on a Priority number that is assigned to them before they enter the Disk. We are not sure what the Priority number indicates (besides a desired run order) but this order is obviously faster than the FCFS order.

Another conclusion we came to was that running the O.S with 4 CPU's is faster than running it with 1 CPU. We believe this is the case, because Dispatcher is able to call the jobs to be Executed by the CPU in a much more efficient way than with one CPU.

In conclusion our major finding where:

- Priority was the fastest Scheduling Algorithm
- 4 CPU's Executed the jobs faster than one CPU