

Project 3: AUBatch Report

Jordan Sosnowski

2020-3-9

Contents

Introduction	3
I. Problem Description	3
II. Background	3
Central Processing Unit	3
First Come, First Served	3
Shortest Job First	4
Priority Based	4
Design and Implementation	4
I. Dataflow Diagram	4
II. AUBatch	6
III. Commandline	7
IV. Modules	15
Performance Metrics	27
Performance Evaluation	30
Instant Arrival	30
First Come First Served, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-10	30
Shortest Job First, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-10	32
Priority Based, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-10	34
Two Second Arrival	36
First Come First Served, 5 Jobs, Arrival Time 2, Priority Range 0-5, CPU Burst Range 0-10	36
Shortest Job First, 5 Jobs, Arrival Time 2, Priority Range 0-5, CPU Burst Range 0-10	38
Priority Based, 5 Jobs, Arrival Time 2, Priority Range 0-5, CPU Burst Range 0-10	40
Max Burst < Arrival Time	42
First Come First Served, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-3	42
Shortest Job First, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-3	44
Priority Based, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-3	46
Lessons Learned	47
Conclusion	47
References	48

Introduction

I. Problem Description

Central processing units are the core of any computer. Any program that has to run has to go through the CPU, as without the CPU the program cannot be executed. However, one single program cannot fully utilize a CPU; therefore, if we were to leave a single program on the CPU until it is finished executing we would be wasting valuable time. Imagine if you could only run one program at a time per CPU on a computer, that would be horrendous. Therefore, it is important to keep a CPU as active as possible. For example, if one program is busing doing I/O it should probably be booted off the CPU so a program that can use the CPU's resources can be loaded. But which program should be loaded next? AUBatch is a simulation that looks into process, or job, scheduling. We look into three algorithms: first come, first served, shortest job first, and priority-based.

Additionally, we assume all of our algorithms are non-preemptive, so once a process is loaded onto the CPU it is there until it is completed. Preemptive algorithms are extremely popular and efficient, as state earlier, but to implement one is out of scope for this current project.

II. Background

To fully understand some of the algorithms and technologies discussed in this paper, a background in these methodologies needs to be established.

Central Processing Unit

A central processing unit (CPU)¹ is hardware that executes instructions that make up a computer program. Also referred to as the brain of the computer, without it the computer would not be able to operate. Most modern CPUs have multiple cores, each core can load a single thread of execution. Therefore a CPU with two cores can run two parallel processes.

First Come, First Served

First come, first served (FCFS)² is a scheduling algorithm that loads processes onto the CPU as they arrive. Therefore, if three processes arrive in the following order A, C, B they will execute in the same order.

Shortest Job First

Shortest job first (SJF)³, also known as shortest job next, loads processes based on the remaining CPU burst time. This scheduler minimizes response time as jobs are usually loaded faster.

Priority Based

Priority based scheduling is similar to SJF. Instead of sorting by remaining CPU burst, it will sort based on priority (highest priority first, lowest last). Our implementation of priority based is non-preemptive, most are preemptive.

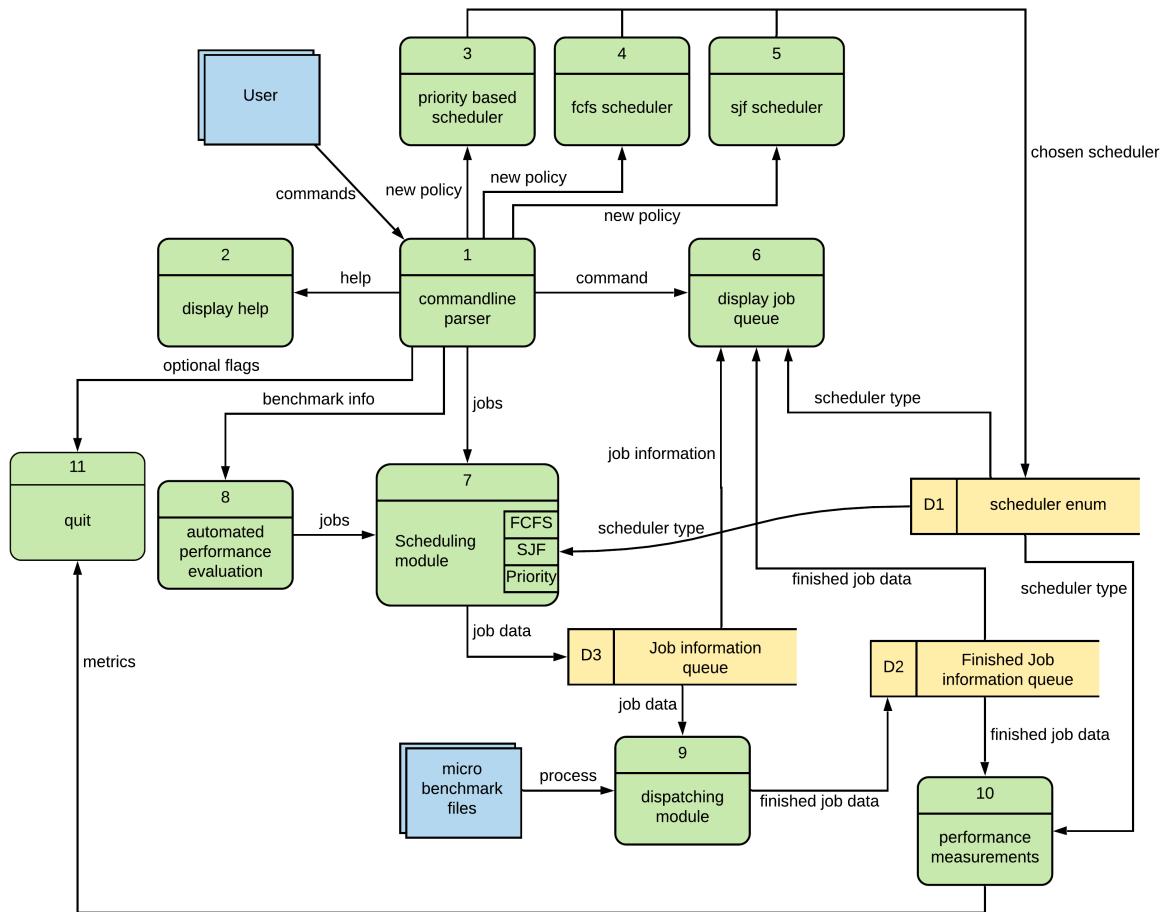
Design and Implementation

I. Dataflow Diagram

Here we have the dataflow diagram for the AUBatch framework. We have two external entities, user, and microbenchmark files. There are 10 processes and 3 data stores. Our `user` interacts with the `commandline parser` which will call a function depending on the input.

We can provide different scheduling types, help / h / ?, list/ls, run ..., test ..., and quit.

- `run` will call the `scheduling module` which will, in turn, load the job into the `job information queue` which is used by the `dispatching module` which when finished will load the finished job onto the `finished job information queue`.
- `list` will call the `display job queue` which pulls information from the `scheduler enum`, the `finished job information queue`, and the `job information queue`.
- `test` will call the `automated performance evaluation process` which sends jobs to the `scheduling module`.
- `fcfs / sjf / priority` will change the current scheduling algorithm by loading it into the `scheduler enum`.
- `quit` can take two optional flags `-i` or `-d`. `-i` will wait until the current job finished, `-d` will wait until all the jobs finish. If no flag is provided it will simply quit immediately.



II. AUBatch

I will say here before I get into the code discussion for any function that I was not familiar with I used [cplusplus](#)⁵ as reference material as it is an unofficial C/C++ API. I also used [GeeksforGeeks](#)⁶ for some fundamental implementations such as how to use `qsort`. Additionally, I used Dr. Qin's sample code as a base for most of this project.

The first module we will discuss is `aubatch.c`. It is the driver for the whole framework and the only module with a main function.

We first include `commandline.h` and `modules.h` so we can gain the functionality of those two files.

```
1 #include "commandline.h"
2 #include "modules.h"
```

After the include statement, we flesh out main. Within main we print the welcome message, and we declare and instantiate a multitude of variables used throughout `commandline.c` and `modules.c`. Following this, we create two threads, one that calls `commandline` located in `commandline.c` and another that calls `dispatcher` which is located in `modules.c`.

`count` is used to determine the number of jobs in the waiting queue, plus the job that is currently on the CPU. `buf_head` and `buf_tail` point to spots within `process_buffer` an array of running and waiting processes. `buf_head` points to the next available spot in the array and `buf_tail` points to the next process that should be loaded, as specified by the scheduling algorithm, onto the CPU. `finished_head` points to the next available spot in the `finished_process_buffer` an array of processes that have finished execution. `batch` is used as a flag to denote whether we are adding jobs in batch mode or not. This is only set to 1 when we do a benchmark with an arrival rate of 0. This means we are assuming all the jobs are arriving at the same time.

Following this, we wait for the threads to join, and if they have return values we print them out.

```
1 int main(int argc, char **argv)
2 {
3     printf("Welcome to Jordan Sosnowski's batch job scheduler Version
4           1.0.\nType 'help' to find more about AUBatch commands.\n");
5     pthread_t executor_thread, dispatcher_thread; /* Two concurrent
6           threads */
7
8     int iret1, iret2;
9
10    /* Initialize count, three buffer pointers */
11    count = 0;
12    buf_head = 0;
13    buf_tail = 0;
```

```

14     finished_head = 0;
15     batch = 0;
16
17     /* Create two independent threads: executor and dispatcher */
18
19     iret1 = pthread_create(&executor_thread, NULL, commandline, (void
20                           *)NULL);
20     iret2 = pthread_create(&dispatcher_thread, NULL, dispatcher, (void
21                           *)NULL);
21
22     /* Initialize the lock the two condition variables */
23     pthread_mutex_init(&cmd_queue_lock, NULL);
24     pthread_cond_init(&cmd_buf_not_full, NULL);
25     pthread_cond_init(&cmd_buf_not_empty, NULL);
26
27     /* Wait till threads are complete before main continues. Unless we
28      */
28     /* wait we run the risk of executing an exit which will terminate
29      */
29     /* the process and all threads before the threads have completed.
30      */
30     pthread_join(executor_thread, NULL);
31     pthread_join(dispatcher_thread, NULL);
32
33     if (iret1)
34         printf("executor_thread returns: %d\n", iret1);
35     if (iret2)
36         printf("dispatcher_thread returns: %d\n", iret1);
37     return 0;
38 }
```

III. Commandline

Within `commandline.c` we include `modules.h` and `commandline.h`.

```

1 #include "commandline.h"
2 #include "modules.h"
```

Following this, we declare an array of strings that define the different values help should print out.

```

1 static const char *helpmenu[] = {
2     "run <job> <time> <pri>: submit a job named <job>, execution
3       time is <time>, priority is <pri>",
4     "list: display the job status",
5     "help: Print help menu",
6     "fcfs: change the scheduling policy to FCFS",
7     "sjf: changes the scheduling policy to SJF",
8     "priority: changes the scheduling policy to priority",
```

```

8     "test: <benchmark> <policy> <num_of_jobs> <priority_levels> <
9      min_CPU_time> <max_CPU_time>",
10    "quit: Exit AUbatch | -i quits after current job finishes | -d
       quits after all jobs finish",
11    NULL};
```

Next, we define a custom type `cmd` which is a struct that houses a string and a function. After that, we define an array of `cmds`. This will be used by `cmd_dispatch` to help decide which function to call based on the input value.

```

1 typedef struct
2 {
3     const char *name;
4     int (*func)(int nargs, char **args);
5 } cmd;
6
7 // array of cmds to be used by the command line
8 static const cmd cmdtable[] = {
9     {"?", cmd_helpmenu},
10    {"h", cmd_helpmenu},
11    {"help", cmd_helpmenu},
12    {"r", cmd_run},
13    {"run", cmd_run},
14    {"q", cmd_quit},
15    {"quit", cmd_quit},
16    {"fcfs", cmd_fcfs},
17    {"sjf", cmd_sjf},
18    {"priority", cmd_priority},
19    {"list", cmd_list},
20    {"ls", cmd_list},
21    {"test", cmd_test},
22    {NULL, NULL}};
```

After this, we hit `commandline` which is called by the executor thread back in `aubatch.c`. This is where the command line gets input from the user to then determine what to do based on said input.

```

1 void *commandline(void *ptr)
2 {
3
4     char *buffer;
5
6     buffer = (char *)malloc(MAX_CMD_LEN * sizeof(char));
7     if (buffer == NULL)
8     {
9         perror("Unable to malloc buffer");
10        exit(1);
11    }
12
13    while (1)
14    {
```

```

15     printf("> [? for menu]: ");
16     fgets(buffer, MAX_CMD_LEN, stdin);
17     remove_newline(buffer);
18     cmd_dispatch(buffer);
19 }
20 return (void *)NULL;
21 }
```

Next we will look into `cmd_dispatch` this is the controller of `commandline` as it helps determine code flow. Within this function we first determine the number of arguments, we assume arguments are space delimited. To determine the number of arguments we use `strtok` to tokenize the string. If we send in a command that has more than 8 arguments we remind the user that they have provided more than the tool can handle. If we provided `test bench1 fcfs 5 0 6 1 10` the `args` array would be as follows: `args[0] = test, args[1] = bench, args[2] = fcfs, args[3] = 5, args[4] = 0, args[5] = 6, args[6] = 1, and args[7] = 10.`

After that we loop through `cmdtable` to see if `args[0]` equals a function. For example if we provided `run ./microbatch.out 10`, `args[0]` would equal `run` which matches with `cmd_run` in `cmdtable`. Once we find the correct function we call it and pass `args` and `nargs` as arguments.

```

1 int cmd_dispatch(char *cmd)
2 {
3     char *args[MAXMENUARGS];
4     int nargs = 0;
5     char *word;
6     char *context;
7     int i, result;
8
9     for (word = strtok_r(cmd, " ", &context);
10         word != NULL;
11         word = strtok_r(NULL, " ", &context))
12     {
13
14         if (nargs >= MAXMENUARGS)
15         {
16             printf("Command line has too many words\n");
17             return E2BIG;
18         }
19         args[nargs++] = word;
20     }
21
22     if (nargs == 0)
23     {
24         return 0;
25     }
26
27     for (i = 0; cmdtable[i].name; i++)
28     {
```

```

1   if (*cmdtable[i].name && !strcmp(args[0], cmdtable[i].name))
2   {
3       assert(cmdtable[i].func != NULL);
4       result = cmdtable[i].func(nargs, args);
5       return result;
6   }
7 }
8 printf("%s: Command not found\n", args[0]);
9 return EINVAL;
10 }
```

What follows next are the implementations for each cmd function.

`cmd_quit`, which is called via `quit` or `q`, will first check and see if you passed any flags with it. If you pass `-i` aubatch will wait for the current process on the CPU to finish. If you pass `-d` aubatch will wait for all processes to finish.

After waiting, or not waiting, it will call `report_metrics` and then exit.

```

1 int cmd_quit(int nargs, char **args)
2 {
3     if (nargs == 2)
4     {
5         if (!strcmp(args[1], "-i")) // wait for current job to finish
6             running
7         {
8             int cur_count = count;
9             printf("Waiting for current job to finish ... \n");
10            if (count)
11            {
12                while (cur_count == count)
13                {
14                }
15            }
16        }
17        else if (!strcmp(args[1], "-d")) // wait for all jobs to finish
18        {
19            printf("Waiting for all jobs to finish...\n");
20            while (count)
21            {
22            }
23        }
24    }
25    printf("Quiting AUBatch... \n");
26
27    report_metrics();
28
29    exit(0);
```

30 }

`cmd_helpmenu`, which can be called with `help`, `h`, or `?`, will loop through each `helpmenu` array element and print to the screen for the user.

```

1 int cmd_helpmenu(int n, char **a)
2 {
3
4     printf("\n");
5     printf("AUbatch help menu\n");
6
7     int i = 0;
8     while (1)
9     {
10         if (helpmenu[i] == NULL)
11         {
12             break;
13         }
14         printf("%s\n", helpmenu[i]);
15         i++;
16     }
17     printf("\n");
18     return 0;
19 }
```

`cmd_priority`, which is called with `priority`, will change the current scheduling policy to priority based.

```

1 int cmd_priority()
2 {
3     policy = PRIORITY;
4     change_scheduler();
5     return 0;
6 }
```

`cmd_sjf`, which is called with `sjf`, will change the current scheduling policy to the shortest job first.

```

1 int cmd_sjf()
2 {
3     policy = SJF;
4     change_scheduler();
5     return 0;
6 }
```

`cmd_fcfs`, which is called with `fcfs`, will change the current scheduling policy to first come, first served.

1 int cmd_sjf()

```

2 {
3     policy = FCFS;
4     change_scheduler();
5     return 0;
6 }
```

Each change in scheduling algorithm will also call `change_scheduler` which will print out some information to the screen for the user. It will also resort to the buffer to ensure the processes are in the correct order for the new scheduler.

```

1 void change_scheduler()
2 {
3     const char *str_policy = get_policy_string();
4     printf("Scheduling policy is switched to %s. All the %d waiting
            jobs have been rescheduled.\n", str_policy, buf_head - buf_tail)
            ;
5     sort_buffer(process_buffer);
6 }
```

`cmd_list`, which is called with `ls` or `list`, will list the running process, and all the finished and waiting processes and relevant information about them. If you have no processes waiting, running, or finished it will notify you.

```

1 int cmd_list()
2 {
3     if (finished_head || count)
4     {
5         printf("Name          CPU_Time Pri Arrival_time
                Progress\n");
6         for (int i = 0; i < finished_head; i++)
7         {
8
9             finished_process_p process = finished_process_buffer[i];
10            char *status = "finished";
11
12            char *time = convert_time(process->arrival_time);
13            remove_newline(time);
14            printf("%-18s %-8d %-3d %s %s\n",
15                   process->cmd,
16                   process->cpu_burst,
17                   process->priority,
18                   time,
19                   status);
20        }
21
22        for (int i = 0; i < buf_head; i++)
23        {
24
25            process_p process = process_buffer[i];
26            char *status = "-----";
```

```

1   if (process->cpu_remaining_burst == 0)
2   {
3       continue;
4   }
5   else if (process->first_time_on_cpu > 0 && process->
6           cpu_remaining_burst > 0)
7   {
8       status = "running ";
9   }
10
11
12     char *time = convert_time(process->arrival_time);
13     remove_newline(time);
14     printf("%-18s %-8d %-3d %s %s\n",
15             process->cmd,
16             process->cpu_burst,
17             process->priority,
18             time,
19             status);
20
21     }
22     printf("\n");
23 }
24
25 else
26     printf("No processes loaded yet!\n");
27 return 0;
28 }
```

`cmd_test`, which is called with `test`, is the benchmark function. It takes 7 parameters: `benchmark_name`, `policy`, `num_of_jobs`, `arrival_rate`, `priority`, `min_cpu_burst`, and `max_cpu_burst`. If the user does not provide the right number of arguments or provides logical fallacies such as `min_cpu_burst > max_cpu_burst` the user will be notified. Additionally, for `test` to work, it assumes no other jobs have been run or are currently running. It assumes that because if there are jobs currently on the CPU or jobs that need to be loaded it would mess with the metrics for the benchmark.

It calls `test_scheduler` which at a high level creates all the jobs needed for the benchmark and notifies `dispatcher` when appropriate. After the jobs finish, we report the metrics, free all the jobs from the finished buffer, and reset the head and tail variables.

We use `while(count){}` to see if any jobs are waiting to run. For each job that is loaded onto `process_buffer` `count` is incremented and for each job that is taken off `count` is decremented. Therefore, once `count` is 0 we know there are no longer any jobs.

I could have used a conditional variable for this but decided not to.

```

1 int cmd_test(int nargs, char **argv)
2 {
3
4     srand(0); // ensure seed is set to the same value each time to make
```

```
      same jobs created
5   if (nargs != 8)
6   {
7       printf("Usage: test <benchmark> <policy> <num_of_jobs> <
           arrival_rate> <priority_levels> <min_CPU_time> <max_CPU_time
           >\n");
8       return EINVAL;
9   }
10  else if (count || finished_head)
11  {
12      printf("Error: Jobs current in queue / on CPU, no jobs should
           have ran if doing benchmark...\n");
13      return EINVAL;
14  }
15  char *benchmark = argv[1];
16  char *str_policy = argv[2];
17  int num_of_jobs = atoi(argv[3]);
18  int arrival_rate = atoi(argv[4]);
19  int priority_levels = atoi(argv[5]);
20  int min_cpu_burst = atoi(argv[6]);
21  int max_cpu_burst = atoi(argv[7]);
22
23  if (min_cpu_burst >= max_cpu_burst)
24  {
25      printf("<min_CPU_time> cannot be greater than or equal
           to <max_CPU_time>\n");
26      return EINVAL;
27  }
28  else if (num_of_jobs < 0 || min_cpu_burst < 0 || max_cpu_burst < 0
29           || priority_levels < 0 || arrival_rate < 0)
30  {
31      printf("<num_of_jobs> <min_CPU_time> <max_CPU_time> <
           arrival_rate> and <priority_levels> must be greater than 0\n
           ");
32      return EINVAL;
33  }
34  if (!strcmp(str_policy, "fcfs"))
35  {
36      policy = FCFS;
37  }
38  else if (!strcmp(str_policy, "sjf"))
39  {
40      policy = SJF;
41  }
42  else if (!strcmp(str_policy, "priority"))
43  {
44      policy = PRIORITY;
45  }
46  else
47  {
```

```

48         printf("Error: <policy> must be either fcfs, sjf, or priority\n"
49                 );
50     }
51
52     test_scheduler(benchmark, num_of_jobs, arrival_rate,
53                     priority_levels, min_cpu_burst, max_cpu_burst);
53     printf("Benchmark is running please wait...\n");
54     while (count)
55     {
56
57         report_metrics();
58
59         // clear process queue and finished queue
60         // ensures that the metrics aren't reported when quitting aubatch
61         // also ensures if running metrics again that the prior jobs will
62         // not interfere
63         for (int i = 0; i < finished_head; i++)
64         {
65             free(finished_process_buffer[i]);
66         }
67         finished_head = 0;
68         buf_head = 0;
69         buf_tail = 0;
70
71         return 0;
72     }

```

IV. Modules

`modules.c` contain the schedulers and dispatcher, without this file the process specified in `run` or `test` would not be able to `run`. `commandline` is simply an interface that interacts with the functions within `modules.c`.

Therefore, within `modules.c` we only import `modules.h`.

```
1 #include "modules.h"
```

First we have `test_scheduler`, which is called by `cmd_test` in `commandline.c`. This function will use mutexes and condition variables to ensure there are no issues among threads.

```

1 pthread_mutex_lock(&cmd_queue_lock);
2
3 while (count == CMD_BUF_SIZE)
4 {
5     pthread_cond_wait(&cmd_buf_not_full, &cmd_queue_lock);
6 }
```

```

7
8 pthread_mutex_unlock(&cmd_queue_lock);

```

After ensuring there are no issues and it is `test_schedulers` time to run we will enter a `for` loop. We will loop for `num_of_jobs`, this will allow us to create the number of jobs specified by `cmd_test`. We create `process` which is of type `process_p`; a pointer to `process_t` which is a custom type of a struct that holds all the information we need about a process. For `test_scheduler` we assume that each cmd to be run by the process will be `./microbatch.out` which is a file that simply sleeps for `n` seconds. This will allow us to get a more accurate CPU burst time. We randomly generate the priority and `cpu_burst` based on the seed set in `cmd_test`. We set the seed to 0 for each time we call `cmd_test` to make sure each time we call benchmark the same jobs are created. If we do not ensure this we cannot accurately compare runs across different scheduling policies. If we provide an `arrival_rate > 0` we will need to load the jobs and notify the dispatcher each time to ensure more accurate metrics. After we notify the dispatcher we will need to sleep for `arrival_rate` seconds. If `arrival_rate` is 0 we assume all the processes arrive at the same time, therefore, we create them all and load them onto the buffer and only once they are all created will we notify `dispatcher`.

If our loop increment becomes larger than `CMD_BUF_SIZE` we need to notify the dispatcher ahead of time to ensure our queue does not overflow.

```

1 if (!arrival_rate)
2     batch = 1;
3
4 // create jobs based on num_of_jobs
5 for (int i = 0; i < num_of_jobs; i++)
6 {
7     /* lock the shared command queue */
8     pthread_mutex_lock(&cmd_queue_lock);
9
10    while (count == CMD_BUF_SIZE)
11    {
12        pthread_cond_wait(&cmd_buf_not_full, &cmd_queue_lock);
13    }
14
15    pthread_mutex_unlock(&cmd_queue_lock);
16
17    int priority = (rand() % (priority_levels + 1)) + 1;
18    int cpu_burst = (rand() % (max_CPU_time + 1)) + min_CPU_time;
19    process_p process = malloc(sizeof(process_t));
20    strcpy(process->cmd, "./microbatch.out");
21    process->arrival_time = time(NULL);
22    process->cpu_burst = cpu_burst;
23    process->cpu_remaining_burst = cpu_burst;
24    process->priority = priority;
25    process->interruptions = 0;
26    process->first_time_on_cpu = 0;
27

```

```
28     if (i >= CMD_BUF_SIZE) // if i is larger than cmd_buff we need to
        notify dispatcher earlier
29     // without this we would be stuck forever
30     {
31         pthread_mutex_lock(&cmd_queue_lock);
32         while (count == CMD_BUF_SIZE)
33         {
34             pthread_cond_wait(&cmd_buf_not_full, &cmd_queue_lock);
35         }
36         pthread_mutex_unlock(&cmd_queue_lock);
37         pthread_mutex_lock(&cmd_queue_lock);
38         process_buffer[buf_head] = process;
39         count++;
40
41         /* Move buf_head forward, this is a circular queue */
42         buf_head++;
43         sort_buffer(process_buffer);
44         buf_head %= CMD_BUF_SIZE;
45         /* Unlock the shared command queue */
46
47         pthread_cond_signal(&cmd_buf_not_empty);
48         pthread_mutex_unlock(&cmd_queue_lock);
49     }
50     else if (arrival_rate) // if there is an arrival rate, notify
        dispatcher immediately and then sleep for arrival_rate
51     {
52
53         pthread_mutex_lock(&cmd_queue_lock);
54         process_buffer[buf_head] = process;
55         count++;
56         /* Move buf_head forward, this is a circular queue */
57         buf_head++;
58
59         sort_buffer(process_buffer);
60         buf_head %= CMD_BUF_SIZE;
61         /* Unlock the shared command queue */
62
63         pthread_cond_signal(&cmd_buf_not_empty);
64         pthread_mutex_unlock(&cmd_queue_lock);
65         sleep(arrival_rate); // wait for the arrival rate
66     }
67     else
68     {
69         pthread_mutex_lock(&cmd_queue_lock);
70         process_buffer[buf_head] = process;
71         count++;
72
73         /* Move buf_head forward, this is a circular queue */
74         buf_head++;
75         sort_buffer(process_buffer);
76         buf_head %= CMD_BUF_SIZE;
```

```

77         pthread_mutex_unlock(&cmd_queue_lock);
78     }
79 }
80 if (!arrival_rate) // if arrival rate is 0, load all the jobs and then
81     notify dispatcher
82 {
83     pthread_mutex_lock(&cmd_queue_lock);
84
85     sort_buffer(process_buffer);
86
87     /* Unlock the shared command queue */
88
89     pthread_cond_signal(&cmd_buf_not_empty);
90     pthread_mutex_unlock(&cmd_queue_lock);
91 }
```

`scheduler` is called by `cmd_run` unlike `test_scheduler` this only loads one job at a time. Immediately after loading the job it always notifies dispatcher so the job can immediately start processing.

We pass `get_process argv` which simply transforms the provides user arguments into `process_p` a pointer to `process_t` which is a custom type of a struct that holds all the information we need about a process. After this, we call `submit_job` which simply prints some useful information to the screen for the user. After that, we load the process onto the buffer, increment count and `buf_head`, and sort the buffer with `sort_buffer`.

After all this, we notify `dispatcher` with `pthread_cond_signal`.

```

1 void scheduler(int argc, char **argv)
2 {
3     /* lock the shared command queue */
4     pthread_mutex_lock(&cmd_queue_lock);
5
6     while (count == CMD_BUF_SIZE)
7     {
8         pthread_cond_wait(&cmd_buf_not_full, &cmd_queue_lock);
9     }
10
11    pthread_mutex_unlock(&cmd_queue_lock);
12    process_p process = get_process(argv);
13
14    // print information about job
15    submit_job(process->cmd);
16
17    process_buffer[buf_head] = process;
18    pthread_mutex_lock(&cmd_queue_lock);
19
20    count++;
21
22    /* Move buf_head forward, this is a circular queue */
```

```

23     buf_head++;
24     buf_head %= CMD_BUF_SIZE;
25
26     // ensure buffer is in accordance to current policy
27     sort_buffer(process_buffer);
28
29     /* Unlock the shared command queue */
30     pthread_cond_signal(&cmd_buf_not_empty);
31     pthread_mutex_unlock(&cmd_queue_lock);
32 }
```

dispatcher will grab the process from `buf_tail` as the sorting algorithm places the next process to be scheduled at the bottom of the queue. After we grab it off we call `complete_process` which runs the process and then loads up a completed process type and loads it onto another buffer.

After we return we decrement the count and move our tail forward. We will also set our `running_process` to `NULL`.

```

1 void *dispatcher(void *ptr)
2 {
3
4     while (1)
5     {
6
7         /* lock and unlock for the shared process queue */
8         pthread_mutex_lock(&cmd_queue_lock);
9
10        // printf("In dispatcher: count = %d\n", count);
11
12        while (count == 0)
13        {
14            pthread_cond_wait(&cmd_buf_not_empty, &cmd_queue_lock);
15        }
16        running_process = process_buffer[buf_tail];
17
18        pthread_cond_signal(&cmd_buf_not_full);
19        /* Unlock the shared command queue */
20        pthread_mutex_unlock(&cmd_queue_lock);
21
22        complete_process(running_process);
23        /* Run the command scheduled in the queue */
24        count--;
25
26        // printf("In dispatcher: process_buffer[%d] = %s\n", buf_tail,
27        // process_buffer[buf_tail]->cmd);
28
29        /* Move buf_tail forward, this is a circular queue */
30        buf_tail++;
31        buf_tail %= CMD_BUF_SIZE;
```

```

32         running_process = NULL;
33     }
34     return (void *)NULL;
35 }
```

Within `calculate_wait` which is called by `submit_job`, we estimate the wait time for the newest process. By wait time we mean the amount of time it will have to wait before it is loaded onto the CPU.

```

1 int calculate_wait()
2 {
3     int wait = 0;
4     for (int i = buf_tail; i < buf_head; i++)
5     {
6         wait += process_buffer[i]->cpu_remaining_burst;
7     }
8     return wait;
9 }
```

`get_process` which is called by `scheduler` takes in a string array and loads up the process's variables.

```

1 process_p get_process(char **argv)
2 {
3     process_p process = malloc(sizeof(process_t));
4     remove_newline(argv[3]);
5
6     // load process structure
7     strcpy(process->cmd, argv[1]);
8     process->arrival_time = time(NULL);
9     process->cpu_burst = atoi(argv[2]);
10    process->cpu_remaining_burst = process->cpu_burst;
11    process->priority = atoi(argv[3]);
12    process->interruptions = 0;
13    process->first_time_on_cpu = 0;
14    return process;
15 }
```

`complete_process` which is called in `dispatcher` performs all the commands needed when finishing a process. First, we will run the process, if the process's cmd is `./microbatch.out` we will append the burst time to it and call system. `microbatch.out` expects an additional command and uses that to determine how long to sleep for.

If we don't provide that program we will simply just run it using `system`. However, we will run the command but pipe its output to `/dev/null` this unclutters our view. For example, if we were to run `/bin/ls` when it lists the files of the current directory it sends that to `/dev/null` instead of standard output.

After that, we create `finished_process`, of type `finished_process_p` a pointer to `finished_process_t`. This is similar to `process_t` except it has variables that are related to metrics.

After we create this variable we load its fields and then increment `finished_head` and free the original processes' memory.

```

1 void complete_process(process_p process)
2 {
3     char cmd[MAX_CMD_LEN * 2];
4     if (!strcmp(process->cmd, "./microbatch.out"))
5         sprintf(cmd, "%s %d", process->cmd, process->
6                 cpu_remaining_burst);
7     else
8         sprintf(cmd, "%s > /dev/null", process->cmd);
9
10    if (process->first_time_on_cpu == 0)
11        process->first_time_on_cpu = time(NULL);
12
13    system(cmd);
14
15    process->cpu_remaining_burst = 0;
16
17    finished_process_p finished_process = malloc(sizeof(
18        finished_process_t));
19    finished_process->finish_time = time(NULL);
20
21    //allows more accurate cpu burst, if we run ls 10 1, ls wont
22    //actually run for 10 seconds, therefore we need to update its
23    //burst time
24    process->cpu_burst = (int)(finished_process->finish_time - process
25                               ->first_time_on_cpu);
26
27    strcpy(finished_process->cmd, process->cmd);
28    finished_process->arrival_time = process->arrival_time;
29    finished_process->cpu_burst = process->cpu_burst;
30    finished_process->interruptions = process->interruptions;
31    finished_process->priority = process->priority;
32    finished_process->first_time_on_cpu = process->first_time_on_cpu;
33    finished_process->turnaround_time = finished_process->finish_time -
34                                         finished_process->arrival_time;
35    if (finished_process->turnaround_time)
36        finished_process->waiting_time = finished_process->
37                                         turnaround_time - finished_process->cpu_burst;
38    else
39        finished_process->waiting_time = 0;
40
41    finished_process->response_time = finished_process->
42                                     first_time_on_cpu - finished_process->arrival_time;
43
44    finished_process_buffer[finished_head] = finished_process;
45    finished_head++;

```

```

38
39     free(process);
40 }
```

Next, we will discuss `report_metrics`. This function is called by `commandline` whenever we quit. It is a lot of code, but all it does is iterate through each finished process and print relative metrics.

It also takes note of specific metrics through each process so it can average them or get the min/max at the end.

```

1 void report_metrics()
2 {
3     if (!finished_head)
4     {
5         printf("No jobs completed!\n");
6         return;
7     }
8     int total_waiting_time = 0;
9     int total_turnaround_time = 0;
10    int total_response_time = 0;
11    int total_cpu_burst = 0;
12
13    int max_waiting_time = INT_MIN;
14    int min_waiting_time = INT_MAX;
15    int max_response_time = INT_MIN;
16    int min_response_time = INT_MAX;
17    int max_turnaround_time = INT_MIN;
18    int min_turnaround_time = INT_MAX;
19    int max_cpu_burst = INT_MIN;
20    int min_cpu_burst = INT_MAX;
21
22    printf("\n==== Reporting Metrics for %s ====\n\n", get_policy_string
23          ());
23    finished_process_p finished_process;
24    int i = 0;
25    for ( ; i < finished_head; i++)
26    {
27        finished_process = finished_process_buffer[i];
28
29        printf("Metrics for job %s:\n", finished_process->cmd);
30        printf("\tCPU Burst:      %d seconds\n", finished_process
31              ->cpu_burst);
31        printf("\tInterruptions:   %d times\n", finished_process->
32              interruptions);
32        printf("\tPriority:        %d\n", finished_process->
33              priority);
33
34        printf("\tArrival Time:    %s", convert_time(
35            finished_process->arrival_time));
35        printf("\tFirst Time on CPU: %s", convert_time(
            finished_process->first_time_on_cpu));
```

```
36     printf("\tFinish Time:      %s", convert_time(
37         finished_process->finish_time));
38     printf("\tTurnaround Time:    %d seconds\n", finished_process
39         ->turnaround_time);
40     printf("\tWaiting Time:       %d seconds\n", finished_process
41         ->waiting_time);
42     printf("\tResponse Time:      %d seconds\n", finished_process
43         ->response_time);
44     printf("\n");
45
46     if (finished_process->waiting_time < min_waiting_time)
47         min_waiting_time = finished_process->waiting_time;
48     if (finished_process->turnaround_time < min_turnaround_time)
49         min_turnaround_time = finished_process->turnaround_time;
50     if (finished_process->response_time < min_response_time)
51         min_response_time = finished_process->response_time;
52     if (finished_process->cpu_burst < min_cpu_burst)
53         min_cpu_burst = finished_process->cpu_burst;
54
55     if (finished_process->waiting_time > max_waiting_time)
56         max_waiting_time = finished_process->waiting_time;
57     if (finished_process->turnaround_time > max_response_time)
58         max_response_time = finished_process->turnaround_time;
59     if (finished_process->response_time > max_response_time)
60         max_response_time = finished_process->response_time;
61     if (finished_process->cpu_burst > max_cpu_burst)
62         max_cpu_burst = finished_process->cpu_burst;
63
64     total_response_time += finished_process->response_time;
65     total_waiting_time += finished_process->waiting_time;
66     total_turnaround_time += finished_process->turnaround_time;
67     total_cpu_burst += finished_process->cpu_burst;
68 }
69
70 printf("Overall Metrics for Batch:\n");
71 printf("\tTotal Number of Jobs Completed: %d\n", finished_head);
72 printf("\tTotal Number of Jobs Submitted: %d\n", finished_head + (
73     buf_head - buf_tail));
74 printf("\tAverage Turnaround Time:        %.3f seconds\n",
75     total_turnaround_time / (float)i);
76 printf("\tAverage Waiting Time:          %.3f seconds\n",
77     total_waiting_time / (float)i);
78 printf("\tAverage Response Time:         %.3f seconds\n",
79     total_response_time / (float)i);
80 printf("\tAverage CPU Burst:            %.3f seconds\n",
81     total_cpu_burst / (float)i);
82 printf("\tTotal CPU Burst:              %d seconds\n",
83     total_cpu_burst);
84 printf("\tThroughput:                  %.3f No./second\n", 1 / (
85     total_turnaround_time / (float)i));
```

```

76
77     printf("\tMax Turnaround Time:           %d seconds\n",
78         max_turnaround_time);
79     printf("\tMin Turnaround Time:           %d seconds\n\n",
80         min_turnaround_time);
81
82     printf("\tMax Waiting Time:             %d seconds\n",
83         max_waiting_time);
84     printf("\tMin Waiting Time:             %d seconds\n\n",
85         min_waiting_time);
86
87     printf("\tMax Response Time:            %d seconds\n",
88         max_response_time);
89     printf("\tMin Response Time:            %d seconds\n\n",
90         min_response_time);
91
92     printf("\tMax CPU Burst:                %d seconds\n",
93         max_cpu_burst);
94     printf("\tMin CPU Burst:                %d seconds\n\n",
95         min_cpu_burst);
96 }

```

`sort_buffer` is the implementation of the scheduling policy. We first determine which policy we are running, we do this to determine at run-time which sorting algorithm to run.

After this, we use `qsort` to sort the `process_buffer`. Note we do something weird with `process_buffer`. We get the element at `buf_tail`, this ensures we get the current process that is not on the CPU. Then we get the address of this and pass it to `qsort`. We also only run for `buf_head - buf_tail` iterations. It is for the same reason as the prior command. If we were to remove the process of the buffer once it is run this would solve this but this workaround works.

```

1 void sort_buffer(process_p *process_buffer)
2 {
3     void *sort;
4     switch (policy)
5     {
6         case FCFS:
7             sort = fcfs_scheduler;
8             break;
9         case SJF:
10            sort = sjf_scheduler;
11            break;
12         case PRIORITY:
13             sort = priority_scheduler;
14     }
15
16     int index;
17
18     // if we are doing a batch job, aka arrival rate is not 0 then add

```

```

19         1 to buf_tail
20     // if we sort ahead of buf_tail for a batch job we will all the
21     // processes even tho
22     // none are currently on the CPU
23     if (!batch)
24         index = buf_tail + 1;
25     else
26         index = buf_tail;
27     qsort(&process_buffer[index], buf_head - index, sizeof(process_p),
28           sort);
29 }
```

What follows next is implementations for the sorting algorithms.

First, we have `sjf_scheduler` this sorts based on `cpu_remaining_burst`.

```

1 int sjf_scheduler(const void *a, const void *b)
2 {
3
4     process_p process_a = *(process_p *)a;
5     process_p process_b = *(process_p *)b;
6
7     return (process_a->cpu_remaining_burst - process_b->
8             cpu_remaining_burst);
```

Next, we have `fcfs_scheduler` which sorts based on arrival time.

```

1 int fcfs_scheduler(const void *a, const void *b)
2 {
3
4     process_p process_a = *(process_p *)a;
5     process_p process_b = *(process_p *)b;
6
7     return (process_a->arrival_time - process_b->arrival_time);
```

Finally, we have `priority_scheduler` which sorts based on priority. Note the final calculation is swapped. This is because we are sorting with the highest priority goes first and the lowest priority goes last.

```

1 int priority_scheduler(const void *a, const void *b)
2 {
3
4     process_p process_a = *(process_p *)a;
5     process_p process_b = *(process_p *)b;
6
7     return (-process_a->priority + process_b->priority);
```

What comes next are some utility functions that just help with ease of use.

First, we have a function that takes in a buffer and removes a trailing newline.

```

1 void remove_newline(char *buffer)
2 {
3     int string_length = strlen(buffer);
4     if (buffer[string_length - 1] == '\n')
5     {
6         buffer[string_length - 1] = '\0';
7     }
8 }
```

Next, we have a function that takes in time and returns the human-readable string version of it.

```

1 char *convert_time(time_t time)
2 {
3     return asctime(localtime(&time));
4 }
```

`get_policy_string` will return the human readable string of the current policy.

```

1 char *get_policy_string()
2 {
3     switch (policy)
4     {
5         case FCFS:
6             return "FCFS";
7
8         case SJF:
9             return "SJF";
10
11        case PRIORITY:
12            return "Priority";
13
14        default:
15            return "Unknown";
16    }
17 }
```

`submit_job` will print out useful information for the user when submitting a job. This includes the name of the job, the number of jobs in the queue, the expected waiting time, and the scheduling policy.

```

1 void submit_job(const char *cmd)
2 {
3     const char *str_policy = get_policy_string();
4     printf("Job %s was submitted.\n", cmd);
5     printf("Total number of jobs in the queue: %d\n", count + 1);
6     printf("Expected waiting time: %d\n",
7           calculate_wait());
```

```
8     printf("Scheduling Policy: %s.\n", str_policy);  
9 }
```

Recommendations

Performance Metrics

```
1 ./aubatch.out  
2 Welcome to Jordan Sosnowski's batch job scheduler Version 1.0.  
3 Type ''help' to find more about AUBatch commands.  
4 > [? for menu]: help  
5  
6 AUBatch help menu  
7 run <job> <time> <pri>: submit a job named <job>, execution time  
     is <time>, priority is <pr>  
8 list: display the job status  
9 help: Print help menu  
10 fcfs: change the scheduling policy to FCFS  
11 sjf: changes the scheduling policy to SJF  
12 priority: changes the scheduling policy to priority  
13 test: <benchmark> <policy> <num_of_jobs> <pri_levels> <  
        min_CPU_time> <max_CPU_time>  
14 quit: Exit AUBatch | -i quits after current job finishes | -d quits  
       after all jobs finish  
15  
16 > [? for menu]: test bench1 fcfs 5 0 5 0 10  
17 Benchmark is running please wait...  
18  
19 === Reporting Metrics for FCFS ===  
20  
21 Metrics for job ./microbatch.out:  
22   CPU Burst:      4 seconds  
23   Interruptions: 0 times  
24   Priority:      1  
25   Arrival Time:  Sun Mar 8 20:06:21 2020  
26   First Time on CPU: Sun Mar 8 20:06:21 2020  
27   Finish Time:   Sun Mar 8 20:06:25 2020  
28   Turnaround Time: 4 seconds  
29   Waiting Time:  0 seconds  
30   Response Time: 0 seconds  
31  
32 Metrics for job ./microbatch.out:  
33   CPU Burst:      5 seconds  
34   Interruptions: 0 times  
35   Priority:      6  
36   Arrival Time:  Sun Mar 8 20:06:21 2020  
37   First Time on CPU: Sun Mar 8 20:06:25 2020  
38   Finish Time:   Sun Mar 8 20:06:30 2020
```

```
39     Turnaround Time:      9 seconds
40     Waiting Time:        4 seconds
41     Response Time:       4 seconds
42
43 Metrics for job ./microbatch.out:
44     CPU Burst:           6 seconds
45     Interruptions:       0 times
46     Priority:            2
47     Arrival Time:        Sun Mar  8 20:06:21 2020
48     First Time on CPU:   Sun Mar  8 20:06:30 2020
49     Finish Time:         Sun Mar  8 20:06:36 2020
50     Turnaround Time:    15 seconds
51     Waiting Time:        9 seconds
52     Response Time:       9 seconds
53
54 Metrics for job ./microbatch.out:
55     CPU Burst:           0 seconds
56     Interruptions:       0 times
57     Priority:             5
58     Arrival Time:        Sun Mar  8 20:06:21 2020
59     First Time on CPU:   Sun Mar  8 20:06:36 2020
60     Finish Time:         Sun Mar  8 20:06:36 2020
61     Turnaround Time:    15 seconds
62     Waiting Time:        15 seconds
63     Response Time:       15 seconds
64
65 Metrics for job ./microbatch.out:
66     CPU Burst:           8 seconds
67     Interruptions:       0 times
68     Priority:             3
69     Arrival Time:        Sun Mar  8 20:06:21 2020
70     First Time on CPU:   Sun Mar  8 20:06:36 2020
71     Finish Time:         Sun Mar  8 20:06:44 2020
72     Turnaround Time:    23 seconds
73     Waiting Time:        15 seconds
74     Response Time:       15 seconds
75
76 Overall Metrics for Batch:
77     Total Number of Jobs Completed: 5
78     Total Number of Jobs Submitted: 5
79     Average Turnaround Time:        13.200 seconds
80     Average Waiting Time:          8.600 seconds
81     Average Response Time:        8.600 seconds
82     Average CPU Burst:            4.600 seconds
83     Total CPU Burst:              23 seconds
84     Throughput:                  0.076 No./second
85     Max Turnaround Time:         23 seconds
86     Min Turnaround Time:         4 seconds
87
88     Max Waiting Time:            15 seconds
89     Min Waiting Time:           0 seconds
```

```
90  
91      Max Response Time:          15 seconds  
92      Min Response Time:          0 seconds  
93  
94      Max CPU Burst:            8 seconds  
95      Min CPU Burst:            0 seconds
```

Performance Evaluation

Instant Arrival

First Come First Served, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-10

```
1 > [? for menu]: test bench1 fcfs 5 0 5 0 10
2 Benchmark is running please wait...
3
4 === Reporting Metrics for FCFS ===
5
6 Metrics for job ./microbatch.out:
7   CPU Burst:        4 seconds
8   Interruptions:    0 times
9   Priority:         1
10  Arrival Time:    Sun Mar  8 20:06:21 2020
11  First Time on CPU: Sun Mar  8 20:06:21 2020
12  Finish Time:     Sun Mar  8 20:06:25 2020
13  Turnaround Time: 4 seconds
14  Waiting Time:    0 seconds
15  Response Time:   0 seconds
16
17 Metrics for job ./microbatch.out:
18   CPU Burst:        5 seconds
19   Interruptions:    0 times
20   Priority:         6
21   Arrival Time:    Sun Mar  8 20:06:21 2020
22   First Time on CPU: Sun Mar  8 20:06:25 2020
23   Finish Time:     Sun Mar  8 20:06:30 2020
24   Turnaround Time: 9 seconds
25   Waiting Time:    4 seconds
26   Response Time:   4 seconds
27
28 Metrics for job ./microbatch.out:
29   CPU Burst:        6 seconds
30   Interruptions:    0 times
31   Priority:         2
32   Arrival Time:    Sun Mar  8 20:06:21 2020
33   First Time on CPU: Sun Mar  8 20:06:30 2020
34   Finish Time:     Sun Mar  8 20:06:36 2020
35   Turnaround Time: 15 seconds
36   Waiting Time:    9 seconds
37   Response Time:   9 seconds
38
39 Metrics for job ./microbatch.out:
40   CPU Burst:        0 seconds
41   Interruptions:    0 times
42   Priority:         5
43   Arrival Time:    Sun Mar  8 20:06:21 2020
```

```
44      First Time on CPU: Sun Mar  8 20:06:36 2020
45      Finish Time:     Sun Mar  8 20:06:36 2020
46      Turnaround Time: 15 seconds
47      Waiting Time:    15 seconds
48      Response Time:   15 seconds
49
50 Metrics for job ./microbatch.out:
51     CPU Burst:        8 seconds
52     Interruptions:   0 times
53     Priority:         3
54     Arrival Time:    Sun Mar  8 20:06:21 2020
55     First Time on CPU: Sun Mar  8 20:06:36 2020
56     Finish Time:     Sun Mar  8 20:06:44 2020
57     Turnaround Time: 23 seconds
58     Waiting Time:    15 seconds
59     Response Time:   15 seconds
60
61 Overall Metrics for Batch:
62     Total Number of Jobs Completed: 5
63     Total Number of Jobs Submitted: 5
64     Average Turnaround Time:       13.200 seconds
65     Average Waiting Time:        8.600 seconds
66     Average Response Time:       8.600 seconds
67     Average CPU Burst:          4.600 seconds
68     Total CPU Burst:            23 seconds
69     Throughput:                0.076 No./second
70     Max Turnaround Time:       23 seconds
71     Min Turnaround Time:      4 seconds
72
73     Max Waiting Time:         15 seconds
74     Min Waiting Time:        0 seconds
75
76     Max Response Time:       15 seconds
77     Min Response Time:      0 seconds
78
79     Max CPU Burst:           8 seconds
80     Min CPU Burst:          0 seconds
```

Shortest Job First, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-10

```
1 > [? for menu]: test bench2 sjf 5 0 5 0 10
2 Benchmark is running please wait...
3
4 === Reporting Metrics for SJF ===
5
6 Metrics for job ./microbatch.out:
7   CPU Burst:          0 seconds
8   Interruptions:     0 times
9   Priority:          5
10  Arrival Time:      Sun Mar  8 20:10:20 2020
11  First Time on CPU: Sun Mar  8 20:10:20 2020
12  Finish Time:       Sun Mar  8 20:10:20 2020
13  Turnaround Time:   0 seconds
14  Waiting Time:      0 seconds
15  Response Time:     0 seconds
16
17 Metrics for job ./microbatch.out:
18   CPU Burst:          3 seconds
19   Interruptions:     0 times
20   Priority:          1
21   Arrival Time:      Sun Mar  8 20:10:20 2020
22   First Time on CPU: Sun Mar  8 20:10:20 2020
23   Finish Time:       Sun Mar  8 20:10:23 2020
24   Turnaround Time:   3 seconds
25   Waiting Time:      0 seconds
26   Response Time:     0 seconds
27
28 Metrics for job ./microbatch.out:
29   CPU Burst:          5 seconds
30   Interruptions:     0 times
31   Priority:          6
32   Arrival Time:      Sun Mar  8 20:10:20 2020
33   First Time on CPU: Sun Mar  8 20:10:23 2020
34   Finish Time:       Sun Mar  8 20:10:28 2020
35   Turnaround Time:   8 seconds
36   Waiting Time:      3 seconds
37   Response Time:     3 seconds
38
39 Metrics for job ./microbatch.out:
40   CPU Burst:          6 seconds
41   Interruptions:     0 times
42   Priority:          2
43   Arrival Time:      Sun Mar  8 20:10:20 2020
44   First Time on CPU: Sun Mar  8 20:10:28 2020
45   Finish Time:       Sun Mar  8 20:10:34 2020
46   Turnaround Time:   14 seconds
47   Waiting Time:      8 seconds
48   Response Time:     8 seconds
49
```

```
50 Metrics for job ./microbatch.out:
51   CPU Burst:          8 seconds
52   Interruptions:     0 times
53   Priority:          3
54   Arrival Time:      Sun Mar  8 20:10:20 2020
55   First Time on CPU: Sun Mar  8 20:10:34 2020
56   Finish Time:       Sun Mar  8 20:10:42 2020
57   Turnaround Time:   22 seconds
58   Waiting Time:      14 seconds
59   Response Time:     14 seconds
60
61 Overall Metrics for Batch:
62   Total Number of Jobs Completed: 5
63   Total Number of Jobs Submitted: 5
64   Average Turnaround Time:        9.400 seconds
65   Average Waiting Time:          5.000 seconds
66   Average Response Time:         5.000 seconds
67   Average CPU Burst:            4.400 seconds
68   Total CPU Burst:              22 seconds
69   Throughput:                  0.106 No./second
70   Max Turnaround Time:         22 seconds
71   Min Turnaround Time:         0 seconds
72
73   Max Waiting Time:           14 seconds
74   Min Waiting Time:           0 seconds
75
76   Max Response Time:          14 seconds
77   Min Response Time:          0 seconds
78
79   Max CPU Burst:              8 seconds
80   Min CPU Burst:              0 seconds
```

Priority Based, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-10

```
1 > [? for menu]: test bench3 priority 5 0 5 0 10
2 Benchmark is running please wait...
3
4 === Reporting Metrics for Priority ===
5
6 Metrics for job ./microbatch.out:
7   CPU Burst:      5 seconds
8   Interruptions: 0 times
9   Priority:       6
10  Arrival Time:  Sun Mar 8 20:13:07 2020
11  First Time on CPU: Sun Mar 8 20:13:07 2020
12  Finish Time:   Sun Mar 8 20:13:12 2020
13  Turnaround Time: 5 seconds
14  Waiting Time:   0 seconds
15  Response Time:  0 seconds
16
17 Metrics for job ./microbatch.out:
18   CPU Burst:      0 seconds
19   Interruptions: 0 times
20   Priority:       5
21   Arrival Time:  Sun Mar 8 20:13:07 2020
22   First Time on CPU: Sun Mar 8 20:13:12 2020
23   Finish Time:   Sun Mar 8 20:13:12 2020
24   Turnaround Time: 5 seconds
25   Waiting Time:   5 seconds
26   Response Time:  5 seconds
27
28 Metrics for job ./microbatch.out:
29   CPU Burst:      8 seconds
30   Interruptions: 0 times
31   Priority:       3
32   Arrival Time:  Sun Mar 8 20:13:07 2020
33   First Time on CPU: Sun Mar 8 20:13:12 2020
34   Finish Time:   Sun Mar 8 20:13:20 2020
35   Turnaround Time: 13 seconds
36   Waiting Time:   5 seconds
37   Response Time:  5 seconds
38
39 Metrics for job ./microbatch.out:
40   CPU Burst:      6 seconds
41   Interruptions: 0 times
42   Priority:       2
43   Arrival Time:  Sun Mar 8 20:13:07 2020
44   First Time on CPU: Sun Mar 8 20:13:20 2020
45   Finish Time:   Sun Mar 8 20:13:26 2020
46   Turnaround Time: 19 seconds
47   Waiting Time:   13 seconds
48   Response Time:  13 seconds
49
```

```
50 Metrics for job ./microbatch.out:  
51   CPU Burst:            3 seconds  
52   Interruptions:        0 times  
53   Priority:             1  
54   Arrival Time:         Sun Mar  8 20:13:07 2020  
55   First Time on CPU:    Sun Mar  8 20:13:26 2020  
56   Finish Time:          Sun Mar  8 20:13:29 2020  
57   Turnaround Time:      22 seconds  
58   Waiting Time:         19 seconds  
59   Response Time:        19 seconds  
60  
61 Overall Metrics for Batch:  
62   Total Number of Jobs Completed: 5  
63   Total Number of Jobs Submitted: 5  
64   Average Turnaround Time:        12.800 seconds  
65   Average Waiting Time:          8.400 seconds  
66   Average Response Time:        8.400 seconds  
67   Average CPU Burst:            4.400 seconds  
68   Total CPU Burst:              22 seconds  
69   Throughput:                 0.078 No./second  
70   Max Turnaround Time:         22 seconds  
71   Min Turnaround Time:         5 seconds  
72  
73   Max Waiting Time:           19 seconds  
74   Min Waiting Time:          0 seconds  
75  
76   Max Response Time:         19 seconds  
77   Min Response Time:         0 seconds  
78  
79   Max CPU Burst:              8 seconds  
80   Min CPU Burst:              0 seconds
```

Two Second Arrival

First Come First Served, 5 Jobs, Arrival Time 2, Priority Range 0-5, CPU Burst Range 0-10

```
1 > [? for menu]: test bench1 fcfs 5 2 5 0 10
2 Benchmark is running please wait...
3
4 === Reporting Metrics for FCFS ===
5
6 Metrics for job ./microbatch.out:
7   CPU Burst:      3 seconds
8   Interruptions: 0 times
9   Priority:      1
10  Arrival Time:  Sun Mar  8 20:14:50 2020
11  First Time on CPU: Sun Mar  8 20:14:50 2020
12  Finish Time:   Sun Mar  8 20:14:53 2020
13  Turnaround Time: 3 seconds
14  Waiting Time:  0 seconds
15  Response Time: 0 seconds
16
17 Metrics for job ./microbatch.out:
18   CPU Burst:      5 seconds
19   Interruptions: 0 times
20   Priority:      6
21   Arrival Time:  Sun Mar  8 20:14:52 2020
22   First Time on CPU: Sun Mar  8 20:14:53 2020
23   Finish Time:   Sun Mar  8 20:14:58 2020
24   Turnaround Time: 6 seconds
25   Waiting Time:  1 seconds
26   Response Time: 1 seconds
27
28 Metrics for job ./microbatch.out:
29   CPU Burst:      6 seconds
30   Interruptions: 0 times
31   Priority:      2
32   Arrival Time:  Sun Mar  8 20:14:54 2020
33   First Time on CPU: Sun Mar  8 20:14:58 2020
34   Finish Time:   Sun Mar  8 20:15:04 2020
35   Turnaround Time: 10 seconds
36   Waiting Time:  4 seconds
37   Response Time: 4 seconds
38
39 Metrics for job ./microbatch.out:
40   CPU Burst:      0 seconds
41   Interruptions: 0 times
42   Priority:      5
43   Arrival Time:  Sun Mar  8 20:14:56 2020
44   First Time on CPU: Sun Mar  8 20:15:04 2020
45   Finish Time:   Sun Mar  8 20:15:04 2020
46   Turnaround Time: 8 seconds
```

```
47 Waiting Time:      8 seconds
48 Response Time:     8 seconds
49
50 Metrics for job ./microbatch.out:
51   CPU Burst:        8 seconds
52   Interruptions:    0 times
53   Priority:         3
54   Arrival Time:    Sun Mar  8 20:14:58 2020
55   First Time on CPU: Sun Mar  8 20:15:04 2020
56   Finish Time:     Sun Mar  8 20:15:12 2020
57   Turnaround Time: 14 seconds
58   Waiting Time:    6 seconds
59   Response Time:   6 seconds
60
61 Overall Metrics for Batch:
62   Total Number of Jobs Completed: 5
63   Total Number of Jobs Submitted: 5
64   Average Turnaround Time:       8.200 seconds
65   Average Waiting Time:         3.800 seconds
66   Average Response Time:       3.800 seconds
67   Average CPU Burst:           4.400 seconds
68   Total CPU Burst:             22 seconds
69   Throughput:                 0.122 No./second
70   Max Turnaround Time:        14 seconds
71   Min Turnaround Time:        3 seconds
72
73   Max Waiting Time:          8 seconds
74   Min Waiting Time:          0 seconds
75
76   Max Response Time:         8 seconds
77   Min Response Time:         0 seconds
78
79   Max CPU Burst:             8 seconds
80   Min CPU Burst:              0 seconds
```

Shortest Job First, 5 Jobs, Arrival Time 2, Priority Range 0-5, CPU Burst Range 0-10

```
1 > [? for menu]: test bench2 sjf 5 2 5 0 10
2 Benchmark is running please wait...
3
4 === Reporting Metrics for SJF ===
5
6 Metrics for job ./microbatch.out:
7     CPU Burst:      3 seconds
8     Interruptions: 0 times
9     Priority:      1
10    Arrival Time:  Mon Mar 9 11:33:48 2020
11    First Time on CPU: Mon Mar 9 11:33:48 2020
12    Finish Time:   Mon Mar 9 11:33:51 2020
13    Turnaround Time: 3 seconds
14    Waiting Time:   0 seconds
15    Response Time:  0 seconds
16
17 Metrics for job ./microbatch.out:
18     CPU Burst:      5 seconds
19     Interruptions: 0 times
20     Priority:      6
21    Arrival Time:  Mon Mar 9 11:33:50 2020
22    First Time on CPU: Mon Mar 9 11:33:51 2020
23    Finish Time:   Mon Mar 9 11:33:56 2020
24    Turnaround Time: 6 seconds
25    Waiting Time:   1 seconds
26    Response Time:  1 seconds
27
28 Metrics for job ./microbatch.out:
29     CPU Burst:      0 seconds
30     Interruptions: 0 times
31     Priority:      5
32    Arrival Time:  Mon Mar 9 11:33:54 2020
33    First Time on CPU: Mon Mar 9 11:33:56 2020
34    Finish Time:   Mon Mar 9 11:33:56 2020
35    Turnaround Time: 2 seconds
36    Waiting Time:   2 seconds
37    Response Time:  2 seconds
38
39 Metrics for job ./microbatch.out:
40     CPU Burst:      6 seconds
41     Interruptions: 0 times
42     Priority:      2
43    Arrival Time:  Mon Mar 9 11:33:52 2020
44    First Time on CPU: Mon Mar 9 11:33:56 2020
45    Finish Time:   Mon Mar 9 11:34:02 2020
46    Turnaround Time: 10 seconds
47    Waiting Time:   4 seconds
48    Response Time:  4 seconds
49
```

```
50 Metrics for job ./microbatch.out:  
51     CPU Burst:          8 seconds  
52     Interruptions:      0 times  
53     Priority:           3  
54     Arrival Time:       Mon Mar  9 11:33:56 2020  
55     First Time on CPU:  Mon Mar  9 11:34:02 2020  
56     Finish Time:        Mon Mar  9 11:34:10 2020  
57     Turnaround Time:    14 seconds  
58     Waiting Time:       6 seconds  
59     Response Time:      6 seconds  
60  
61 Overall Metrics for Batch:  
62     Total Number of Jobs Completed: 5  
63     Total Number of Jobs Submitted: 5  
64     Average Turnaround Time:        7.000 seconds  
65     Average Waiting Time:         2.600 seconds  
66     Average Response Time:        2.600 seconds  
67     Average CPU Burst:           4.400 seconds  
68     Total CPU Burst:             22 seconds  
69     Throughput:                 0.143 No./second  
70     Max Turnaround Time:        14 seconds  
71     Min Turnaround Time:        2 seconds  
72  
73     Max Waiting Time:          6 seconds  
74     Min Waiting Time:          0 seconds  
75  
76     Max Response Time:         6 seconds  
77     Min Response Time:         0 seconds  
78  
79     Max CPU Burst:            8 seconds  
80     Min CPU Burst:             0 seconds
```

Priority Based, 5 Jobs, Arrival Time 2, Priority Range 0-5, CPU Burst Range 0-10

```
1 > [? for menu]: test bench3 priority 5 2 5 0 10
2 Benchmark is running please wait...
3
4 === Reporting Metrics for Priority ===
5
6 Metrics for job ./microbatch.out:
7     CPU Burst:      3 seconds
8     Interruptions: 0 times
9     Priority:      1
10    Arrival Time:  Mon Mar 9 11:34:57 2020
11    First Time on CPU: Mon Mar 9 11:34:57 2020
12    Finish Time:   Mon Mar 9 11:35:00 2020
13    Turnaround Time: 3 seconds
14    Waiting Time:  0 seconds
15    Response Time: 0 seconds
16
17 Metrics for job ./microbatch.out:
18     CPU Burst:      5 seconds
19     Interruptions: 0 times
20     Priority:      6
21    Arrival Time:  Mon Mar 9 11:34:59 2020
22    First Time on CPU: Mon Mar 9 11:35:00 2020
23    Finish Time:   Mon Mar 9 11:35:05 2020
24    Turnaround Time: 6 seconds
25    Waiting Time:  1 seconds
26    Response Time: 1 seconds
27
28 Metrics for job ./microbatch.out:
29     CPU Burst:      0 seconds
30     Interruptions: 0 times
31     Priority:      5
32    Arrival Time:  Mon Mar 9 11:35:03 2020
33    First Time on CPU: Mon Mar 9 11:35:05 2020
34    Finish Time:   Mon Mar 9 11:35:05 2020
35    Turnaround Time: 2 seconds
36    Waiting Time:  2 seconds
37    Response Time: 2 seconds
38
39 Metrics for job ./microbatch.out:
40     CPU Burst:      8 seconds
41     Interruptions: 0 times
42     Priority:      3
43    Arrival Time:  Mon Mar 9 11:35:05 2020
44    First Time on CPU: Mon Mar 9 11:35:05 2020
45    Finish Time:   Mon Mar 9 11:35:13 2020
46    Turnaround Time: 8 seconds
47    Waiting Time:  0 seconds
48    Response Time: 0 seconds
49
```

```
50 Metrics for job ./microbatch.out:  
51     CPU Burst:          6 seconds  
52     Interruptions:      0 times  
53     Priority:           2  
54     Arrival Time:       Mon Mar  9 11:35:01 2020  
55     First Time on CPU:  Mon Mar  9 11:35:13 2020  
56     Finish Time:        Mon Mar  9 11:35:19 2020  
57     Turnaround Time:    18 seconds  
58     Waiting Time:       12 seconds  
59     Response Time:      12 seconds  
60  
61 Overall Metrics for Batch:  
62     Total Number of Jobs Completed: 5  
63     Total Number of Jobs Submitted: 5  
64     Average Turnaround Time:        7.400 seconds  
65     Average Waiting Time:         3.000 seconds  
66     Average Response Time:        3.000 seconds  
67     Average CPU Burst:            4.400 seconds  
68     Total CPU Burst:              22 seconds  
69     Throughput:                  0.135 No./second  
70     Max Turnaround Time:         18 seconds  
71     Min Turnaround Time:         2 seconds  
72  
73     Max Waiting Time:           12 seconds  
74     Min Waiting Time:          0 seconds  
75  
76     Max Response Time:          12 seconds  
77     Min Response Time:         0 seconds  
78  
79     Max CPU Burst:              8 seconds  
80     Min CPU Burst:              0 seconds
```

Max Burst < Arrival Time

First Come First Served, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-3

```
1 > [? for menu]: test bench1 fcfs 5 2 5 0 3
2 Benchmark is running please wait...
3
4 === Reporting Metrics for FCFS ===
5
6 Metrics for job ./microbatch.out:
7     CPU Burst:      3 seconds
8     Interruptions: 0 times
9     Priority:       1
10    Arrival Time:   Mon Mar 9 11:37:56 2020
11    First Time on CPU: Mon Mar 9 11:37:56 2020
12    Finish Time:    Mon Mar 9 11:37:59 2020
13    Turnaround Time: 3 seconds
14    Waiting Time:   0 seconds
15    Response Time:  0 seconds
16
17 Metrics for job ./microbatch.out:
18     CPU Burst:      0 seconds
19     Interruptions: 0 times
20     Priority:       6
21    Arrival Time:   Mon Mar 9 11:37:58 2020
22    First Time on CPU: Mon Mar 9 11:37:59 2020
23    Finish Time:    Mon Mar 9 11:37:59 2020
24    Turnaround Time: 1 seconds
25    Waiting Time:   1 seconds
26    Response Time:  1 seconds
27
28 Metrics for job ./microbatch.out:
29     CPU Burst:      1 seconds
30     Interruptions: 0 times
31     Priority:       2
32    Arrival Time:   Mon Mar 9 11:38:00 2020
33    First Time on CPU: Mon Mar 9 11:38:00 2020
34    Finish Time:    Mon Mar 9 11:38:01 2020
35    Turnaround Time: 1 seconds
36    Waiting Time:   0 seconds
37    Response Time:  0 seconds
38
39 Metrics for job ./microbatch.out:
40     CPU Burst:      1 seconds
41     Interruptions: 0 times
42     Priority:       5
43    Arrival Time:   Mon Mar 9 11:38:02 2020
44    First Time on CPU: Mon Mar 9 11:38:02 2020
45    Finish Time:    Mon Mar 9 11:38:03 2020
46    Turnaround Time: 1 seconds
```

```
47      Waiting Time:      0 seconds
48      Response Time:    0 seconds
49
50 Metrics for job ./microbatch.out:
51     CPU Burst:        1 seconds
52     Interruptions:   0 times
53     Priority:         3
54     Arrival Time:    Mon Mar  9 11:38:04 2020
55     First Time on CPU: Mon Mar  9 11:38:04 2020
56     Finish Time:     Mon Mar  9 11:38:05 2020
57     Turnaround Time: 1 seconds
58     Waiting Time:    0 seconds
59     Response Time:   0 seconds
60
61 Overall Metrics for Batch:
62     Total Number of Jobs Completed: 5
63     Total Number of Jobs Submitted: 5
64     Average Turnaround Time:       1.400 seconds
65     Average Waiting Time:        0.200 seconds
66     Average Response Time:      0.200 seconds
67     Average CPU Burst:          1.200 seconds
68     Total CPU Burst:            6 seconds
69     Throughput:                0.714 No./second
70     Max Turnaround Time:       1 seconds
71     Min Turnaround Time:       1 seconds
72
73     Max Waiting Time:         1 seconds
74     Min Waiting Time:        0 seconds
75
76     Max Response Time:       1 seconds
77     Min Response Time:      0 seconds
78
79     Max CPU Burst:           3 seconds
80     Min CPU Burst:          0 seconds
```

Shortest Job First, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-3

```
1 > [? for menu]: test bench3 sjf 5 2 5 0 3
2 Benchmark is running please wait...
3
4 === Reporting Metrics for SJF ===
5
6 Metrics for job ./microbatch.out:
7     CPU Burst:      3 seconds
8     Interruptions: 0 times
9     Priority:      1
10    Arrival Time:  Mon Mar 9 11:38:58 2020
11    First Time on CPU: Mon Mar 9 11:38:58 2020
12    Finish Time:   Mon Mar 9 11:39:01 2020
13    Turnaround Time: 3 seconds
14    Waiting Time:   0 seconds
15    Response Time:  0 seconds
16
17 Metrics for job ./microbatch.out:
18     CPU Burst:      0 seconds
19     Interruptions: 0 times
20     Priority:      6
21    Arrival Time:  Mon Mar 9 11:39:00 2020
22    First Time on CPU: Mon Mar 9 11:39:01 2020
23    Finish Time:   Mon Mar 9 11:39:01 2020
24    Turnaround Time: 1 seconds
25    Waiting Time:   1 seconds
26    Response Time:  1 seconds
27
28 Metrics for job ./microbatch.out:
29     CPU Burst:      1 seconds
30     Interruptions: 0 times
31     Priority:      2
32    Arrival Time:  Mon Mar 9 11:39:02 2020
33    First Time on CPU: Mon Mar 9 11:39:02 2020
34    Finish Time:   Mon Mar 9 11:39:03 2020
35    Turnaround Time: 1 seconds
36    Waiting Time:   0 seconds
37    Response Time:  0 seconds
38
39 Metrics for job ./microbatch.out:
40     CPU Burst:      1 seconds
41     Interruptions: 0 times
42     Priority:      5
43    Arrival Time:  Mon Mar 9 11:39:04 2020
44    First Time on CPU: Mon Mar 9 11:39:04 2020
45    Finish Time:   Mon Mar 9 11:39:05 2020
46    Turnaround Time: 1 seconds
47    Waiting Time:   0 seconds
48    Response Time:  0 seconds
49
```

```
50 Metrics for job ./microbatch.out:  
51     CPU Burst:          1 seconds  
52     Interruptions:      0 times  
53     Priority:           3  
54     Arrival Time:       Mon Mar  9 11:39:06 2020  
55     First Time on CPU:  Mon Mar  9 11:39:06 2020  
56     Finish Time:        Mon Mar  9 11:39:07 2020  
57     Turnaround Time:    1 seconds  
58     Waiting Time:       0 seconds  
59     Response Time:      0 seconds  
60  
61 Overall Metrics for Batch:  
62     Total Number of Jobs Completed: 5  
63     Total Number of Jobs Submitted: 5  
64     Average Turnaround Time:        1.400 seconds  
65     Average Waiting Time:         0.200 seconds  
66     Average Response Time:        0.200 seconds  
67     Average CPU Burst:           1.200 seconds  
68     Total CPU Burst:             6 seconds  
69     Throughput:                 0.714 No./second  
70     Max Turnaround Time:        1 seconds  
71     Min Turnaround Time:        1 seconds  
72  
73     Max Waiting Time:          1 seconds  
74     Min Waiting Time:          0 seconds  
75  
76     Max Response Time:         1 seconds  
77     Min Response Time:         0 seconds  
78  
79     Max CPU Burst:            3 seconds  
80     Min CPU Burst:             0 seconds
```

Priority Based, 5 Jobs, Arrival Time 0, Priority Range 0-5, CPU Burst Range 0-3

```
1 > [? for menu]: test bench3 priority 5 2 5 0 3
2 Benchmark is running please wait...
3
4 === Reporting Metrics for Priority ===
5
6 Metrics for job ./microbatch.out:
7     CPU Burst:      3 seconds
8     Interruptions: 0 times
9     Priority:      1
10    Arrival Time:   Mon Mar 9 11:36:36 2020
11    First Time on CPU: Mon Mar 9 11:36:36 2020
12    Finish Time:    Mon Mar 9 11:36:39 2020
13    Turnaround Time: 3 seconds
14    Waiting Time:   0 seconds
15    Response Time:  0 seconds
16
17 Metrics for job ./microbatch.out:
18     CPU Burst:      0 seconds
19     Interruptions: 0 times
20     Priority:      6
21    Arrival Time:   Mon Mar 9 11:36:38 2020
22    First Time on CPU: Mon Mar 9 11:36:39 2020
23    Finish Time:    Mon Mar 9 11:36:39 2020
24    Turnaround Time: 1 seconds
25    Waiting Time:   1 seconds
26    Response Time:  1 seconds
27
28 Metrics for job ./microbatch.out:
29     CPU Burst:      1 seconds
30     Interruptions: 0 times
31     Priority:      2
32    Arrival Time:   Mon Mar 9 11:36:40 2020
33    First Time on CPU: Mon Mar 9 11:36:40 2020
34    Finish Time:    Mon Mar 9 11:36:41 2020
35    Turnaround Time: 1 seconds
36    Waiting Time:   0 seconds
37    Response Time:  0 seconds
38
39 Metrics for job ./microbatch.out:
40     CPU Burst:      1 seconds
41     Interruptions: 0 times
42     Priority:      5
43    Arrival Time:   Mon Mar 9 11:36:42 2020
44    First Time on CPU: Mon Mar 9 11:36:42 2020
45    Finish Time:    Mon Mar 9 11:36:43 2020
46    Turnaround Time: 1 seconds
47    Waiting Time:   0 seconds
48    Response Time:  0 seconds
49
```

```

50 Metrics for job ./microbatch.out:
51     CPU Burst:          1 seconds
52     Interruptions:      0 times
53     Priority:           3
54     Arrival Time:       Mon Mar  9 11:36:44 2020
55     First Time on CPU: Mon Mar  9 11:36:44 2020
56     Finish Time:        Mon Mar  9 11:36:45 2020
57     Turnaround Time:   1 seconds
58     Waiting Time:       0 seconds
59     Response Time:      0 seconds
60
61 Overall Metrics for Batch:
62     Total Number of Jobs Completed: 5
63     Total Number of Jobs Submitted: 5
64     Average Turnaround Time:        1.400 seconds
65     Average Waiting Time:          0.200 seconds
66     Average Response Time:         0.200 seconds
67     Average CPU Burst:            1.200 seconds
68     Total CPU Burst:              6 seconds
69     Throughput:                  0.714 No./second
70     Max Turnaround Time:         1 seconds
71     Min Turnaround Time:         1 seconds
72
73     Max Waiting Time:           1 seconds
74     Min Waiting Time:           0 seconds
75
76     Max Response Time:          1 seconds
77     Min Response Time:          0 seconds
78
79     Max CPU Burst:              3 seconds
80     Min CPU Burst:              0 seconds

```

Lessons Learned

Before this project, I had “okay” proficiency with the C language. I had never dealt with programming with mutexes, conditional variables or threads so those new additions were a challenge. Dr. Qin’s source code aided in the creation process as a base to go off of. I feel though after this project my C language understanding has doubled, if not tripled. The biggest hurdle I had was dealing with double pointers, my custom type `process_p` and `process_t` and dealing with thread synchronization. For example the day of submission I realized I had a large bug with the benchmark code. I was not ensuring a small edge case code was locking the mutex to ensure it was synced with the other thread.

Conclusion

References

- 1: https://en.wikipedia.org/wiki/Central_processing_unit
- 2: [https://en.wikipedia.org/wiki/Scheduling_\(computing\)#First_come,_first_served](https://en.wikipedia.org/wiki/Scheduling_(computing)#First_come,_first_served)
- 3: https://en.wikipedia.org/wiki/Shortest_job_next
- 4: [https://en.wikipedia.org/wiki/Scheduling_\(computing\)#Fixed_priority_pre-emptive_scheduling](https://en.wikipedia.org/wiki/Scheduling_(computing)#Fixed_priority_pre-emptive_scheduling)
- 5: <http://www.cplusplus.com/>
- 6: <https://www.geeksforgeeks.org/>