1. **Source Code**:
   - README.txt

```
# Bharath Sukesh Scheduling Simulator -- README

Operating Systems Assignment 2021 S1 -- Curtin University

### Functionality:

To compile, run 'make clean' then 'make' in the respective directory folder, and then to run -- ./schedsim

E.g:

    In Assignment\Priority: 'make', then './pp'

    In Assignment\SRTF: 'make', then './srtf'

    In Assignment\Threading: 'make', then './schedsim'

================================================================
```

I: Header file (used by all three programs) & Makefile

- task.h

```c
#ifndef TASK_H
#define TASK_H


    /* Defined macros*/
    #define TRUE 1
    #define FALSE 0

    typedef struct
    {
        int pid;
        int arrival_time, burst_time, priority; // Read in from file.
        int waiting_time, turnaround_time; // What we need to calculate.
        int time_left, complete_time; // What we maintain throughout the program.
        int status;
    }Task;

    // Forward declarations
    int readFile(char* argv, Task* taskArray);

#endif
```

For scheduling simulation program, additional forward declarations are present:

```c
// Forward declarations
int readFile(char* argv, Task* taskArray);
void* pp();
void* srtf();
```

Makefiles:

```
# Makefile Variables
# AUTHOR: Bharath Sukesh - 19982634
# PURPOSE: Makefile to compile
scheduler program

CC              = gcc
EXEC     = srtf
CFLAGS = -Wall -ansi -pedantic -pthread
-g -std=c99
OBJ      = srtf.o
$(EXEC) : $(OBJ)
        $(CC) $(OBJ) -o $(EXEC) -g

srtf.o : srtf.c task.h
        $(CC) $(CFLAGS) -c srtf.c

clean:
        rm -f $(EXEC) $(OBJ)
```

```
# Makefile Variables
# AUTHOR: Bharath Sukesh - 19982634
# PURPOSE: Makefile to compile
scheduler program

CC              = gcc
EXEC     = pp
CFLAGS = -Wall -ansi -pedantic -pthread
-g -std=c99
OBJ      = pp.o
$(EXEC) : $(OBJ)
        $(CC) $(OBJ) -o $(EXEC) -g

pp.o : pp.c task.h
        $(CC) $(CFLAGS) -c pp.c

clean:
        rm -f $(EXEC) $(OBJ)
```

```
# Makefile Variables
# AUTHOR: Bharath Sukesh - 19982634
# PURPOSE: Makefile to compile
threading program
CC              = gcc
EXEC     = schedsim
CFLAGS = -Wall -ansi -pedantic -pthread
-g -std=c99
OBJ      = schedsim.o
$(EXEC) : $(OBJ)
        $(CC) $(OBJ) -o $(EXEC) -g -
pthread

schedsim.o : schedsim.c task.h
        $(CC) $(CFLAGS) -c schedsim.c

clean:
        rm -f $(EXEC) $(OBJ)
```

II: Scheduler programs

- **readFile method** – present in both scheduling algorithm files (both srtf.c & pp.c) – pasting once for readability/convenience.

```
// ================== File IO ==================
int readFile(char* argv, Task* taskArray)
{
    FILE* file; /* File pointer to the settings file stream */

    int counter = 0;
    /* Opens file with a name specified by user in the command line */
    file = fopen(argv, "r");

    if(file == NULL) /* If file cannot open */
    {
        printf("Error - could not open file. ");
    }
    else // Read line by line.
    {
        while(!feof(file)) /* && (error == FALSE)) */
        {
            // Format: {Arrival} <space> {Burst} <space> {Priority}

            fscanf(file, "%d %d %d", &(taskArray[counter].arrival_time), (&taskArray[counter].burst_time),
(&taskArray[counter].priority));
            /* Read in the file and assort each part of the file to its according part in our taskArray*/
            taskArray[counter].pid = counter + 1;
            taskArray[counter].time_left = taskArray[counter].burst_time; // Burst = Orig burst time (constant),
timeleft changes upon bursts.
            tot_burst += taskArray[counter].burst_time;
            counter++;
            numProc++;
        }
        if(ferror(file))
        {
            perror("Error reading from file. \n");
        }
        fclose(file);
    }
}
```

- **srtf.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Task Struct
#include "task.h"

// Global variables
int current_time = 0;
int numProc = 0;
int tot_burst = 0;

// ============= Main =================
int main(int argc, char* argv[])
{
    int currentIdx = 0;
    char menuStr[] = "SRTF Simulation\n"
                     "Enter Filename: \n";
    // Start operation:
    char filename[20];
    Task taskArray[50];
    while(1)
    {
        printf("\n%s", menuStr);
        scanf("%s", filename);
        if(strcmp(filename, "QUIT") == 0)
        {
            break;
        }
        else
        {
            readFile(filename, taskArray);
            for(int ii=0;ii<numProc;ii++)
            {
                printf("%d %d %d \n", taskArray[ii].arrival_time,taskArray[ii].burst_time, taskArray[ii].priority);
            }

            // Perform scheduling
            int maxIdx = -5;
            while(current_time != tot_burst)
            {
                maxIdx = -5; // Reset max, find new max.
                        taskArray[maxIdx].time_left = 9999;
                for(int ii=0;ii<numProc;ii++)
                { // Check if time_left is lower.
```

```
                    if(taskArray[ii].arrival_time <= current_time && taskArray[ii].time_left != 0 &&
taskArray[ii].time_left < taskArray[maxIdx].time_left)
                    {
                        maxIdx = ii;
                    }
                }

                // Max/Current should be initialised per timestep.
                printf("P%d|", taskArray[maxIdx].pid);
                // If a task was found i.e. if maxIdx != -5
                if(maxIdx != -5)
                {
                    // Decrement remaining time.
                    taskArray[maxIdx].time_left--;
                    // Increment total time
                    current_time++;
                    if(taskArray[maxIdx].time_left == 0)
                    {
                        //add CT
                        taskArray[maxIdx].complete_time = current_time;

                        //calc and store TT/WT
                        // TT = CT - AT
                        taskArray[maxIdx].turnaround_time = taskArray[maxIdx].complete_time -
taskArray[maxIdx].arrival_time;
                        // WT = TT - BT
                        taskArray[maxIdx].waiting_time = taskArray[maxIdx].turnaround_time -
taskArray[maxIdx].burst_time;
                    }
                }
                else
                {
                    current_time++;
                }
            }
            printf("\n\n");

            int maxWT = 0;
            int maxTT = 0;
            for(int ii=0;ii<numProc;ii++)
            {
                maxWT += taskArray[ii].waiting_time;
                maxTT += taskArray[ii].turnaround_time;
            }

            float avgWT=0,avgTT=0;
            avgTT = (float) maxTT / numProc;
```

```
            avgWT  = (float) maxWT / numProc;

            printf("avgWT = %f\n", avgWT);
            printf("avgTT = %f\n", avgTT);
            current_time = 0;
            numProc = 0;
            tot_burst = 0;
        }
    }
}
```

- **pp.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Task Struct
#include "task.h"

// Global variables
int current_time = 0;
int numProc = 0;
int tot_burst = 0;

// ============= Main =================
int main(int argc, char* argv[])
{
    int currentIdx = 0;
    char menuStr[] = "PP Simulation\n"
                     "Enter Filename: \n";
    // Start operation:
    char filename[20];
    Task taskArray[50];
    while(1)
    {
        printf("\n%s", menuStr);
        scanf("%s", filename);
        if(strcmp(filename, "QUIT") == 0)
        {
            break;
        }
        else
        {
```

```
            readFile(filename, taskArray);
            for(int ii=0;ii<numProc;ii++)
            {
                printf("%d %d %d \n", taskArray[ii].arrival_time,taskArray[ii].burst_time, taskArray[ii].priority);
            }

            // Perform scheduling
            int maxIdx = -5;
            while(current_time != tot_burst)
            {
                maxIdx = -5; // Reset max, find new max.
                taskArray[maxIdx].priority = 9999;
                for(int ii=0;ii<numProc;ii++)
                { // Check if priority is lower.
                    if(taskArray[ii].arrival_time <= current_time && taskArray[ii].time_left != 0 &&
taskArray[ii].priority < taskArray[maxIdx].priority)
                    {
                        maxIdx = ii;
                    }
                }

                // Max/Current should be initialised per timestep.
                printf("P%d|", taskArray[maxIdx].pid);
                // If a task was found i.e. if maxIdx != -5
                if(maxIdx != -5)
                {
                    // Decrement remaining time.
                    taskArray[maxIdx].time_left--;
                    // Increment total time
                    current_time++;
                    if(taskArray[maxIdx].time_left == 0)
                    {
                        //add CT
                        taskArray[maxIdx].complete_time = current_time;

                        //calc and store TT/WT
                        // TT = CT - AT
                        taskArray[maxIdx].turnaround_time = taskArray[maxIdx].complete_time -
taskArray[maxIdx].arrival_time;
                        // WT = TT - BT
                        taskArray[maxIdx].waiting_time = taskArray[maxIdx].turnaround_time -
taskArray[maxIdx].burst_time;
                    }
                }
                else
                {
                    current_time++;
```

```
                }
            }
            printf("\n\n");

            int maxWT = 0;
            int maxTT = 0;
            for(int ii=0;ii<numProc;ii++)
            {
                maxWT += taskArray[ii].waiting_time;
                maxTT += taskArray[ii].turnaround_time;
            }

            float avgWT=0,avgTT=0;
            avgTT = (float) maxTT / numProc;
            avgWT  = (float) maxWT / numProc;

            printf("avgWT = %f\n", avgWT);
            printf("avgTT = %f\n", avgTT);
            current_time = 0;
            numProc = 0;
            tot_burst = 0;
        }
    }
}
```

III: Threading simulation

- **schedsim.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
// ====================== Task Struct ============
#include "task.h"

// Global variables for Scheduling.
int current_time = 0;
int numProc = 0;
int tot_burst = 0;

//Global variables for memory shared among threads
int bufferCount = 0;
int done = 0;
char buffer1[11];
char buffer2[11];
pthread_mutex_t lock;
pthread_mutex_t readingMutex;
pthread_mutex_t receivingMutex;
pthread_cond_t calculated = PTHREAD_COND_INITIALIZER;
pthread_cond_t read = PTHREAD_COND_INITIALIZER;

// ============= Main // Parent Thread =================
int main(int argc, char* argv[])
{
    char outputArr[50];
    pthread_t srtfT;
    pthread_t ppT;
    int tally = 0;
    int done = 0;
    //initialise lock
    pthread_mutex_init(&lock, NULL);
    pthread_mutex_init(&readingMutex, NULL);
    pthread_mutex_init(&receivingMutex, NULL);


    do
    {
        if(pthread_create(&srtfT, NULL, &srtf, NULL) != 0)
        {
            return NULL;
```

```
        }

        if(pthread_create(&ppT, NULL, &pp, NULL) != 0)
        {
            return NULL;
        }
        printf("Enter Filename: \n");
        scanf("%s", buffer1);

        if(strcmp(buffer1, "QUIT") == 0)
        {
            done = 1;
            printf("SRTF terminating.....\n");
            printf("PP terminating.....\n");
            exit(0);
        }
        else if(buffer1[0] != '0/')
        {
            pthread_cond_broadcast(&read); //--- Kickstarts reading process - signals A & B to read.
        }

        // ========================================================================
        while(bufferCount == 0 && done == 0)
        {
            pthread_cond_wait(&calculated, &readingMutex);
        }

        sprintf(outputArr,"%s",buffer2);

        tally++;
        bufferCount = 0;
        buffer2[0] = '\0'; // Empties the buffer.
        while(bufferCount == 0 && done == 0)
        {
            pthread_cond_wait(&calculated, &readingMutex);
        }
        strcat(outputArr,buffer2);
        tally++;
        if(tally == 2 && done == 0)
        {
            printf("\n%s\n", outputArr);
        }
        bufferCount = 0;
        buffer2[0] = '\0'; // Empties the buffer.
        buffer1[0] = '\0'; // Empties the buffer.
        tally = 0; // Reset tally.
    } while(done == 0);
```

```
    // =============
    if(pthread_join(srtfT, NULL) != 0)
    {
        return NULL;
    }

    if(pthread_join(ppT, NULL) != 0)
    {
        return NULL;
    }

    printf("SRTF terminating.....\n");
    printf("PP terminating.....\n");

    pthread_mutex_destroy(&lock);
    pthread_mutex_destroy(&readingMutex);
    pthread_mutex_destroy(&receivingMutex);

    return 0;
}

// ================== Thread A: PP  ==================

void* pp()
{
    char output[] = "PP Simulation\n";
    // Start operation:
    Task taskArray[50];
    pthread_mutex_lock(&lock);

    while(buffer1[0] == '\0')
    {
        pthread_cond_wait(&read, &lock); // This gets unlocked when filename received.
    }


    readFile(buffer1, taskArray);
    printf("%s", output);

    // Perform scheduling
    int maxIdx = -5;
    while(current_time != tot_burst)
    {
        maxIdx = -5; // Reset max, find new max.
        taskArray[maxIdx].priority = 9999;
        for(int ii=0;ii<numProc;ii++)
```

```
        {
            if(taskArray[ii].arrival_time <= current_time && taskArray[ii].time_left != 0 && taskArray[ii].priority <
taskArray[maxIdx].priority) // Check if priority is lower.
            {
                maxIdx = ii;
            }
        }

        // Max/Current should be initialised per timestep.
        printf("P%d|", taskArray[maxIdx].pid);
        // If a task was found i.e. if maxIdx != -5
        if(maxIdx != -5)
        {
            // Decrement remaining time.
            taskArray[maxIdx].time_left--;
            // Increment total time
            current_time++;
            if(taskArray[maxIdx].time_left == 0)
            {
                //add CT
                taskArray[maxIdx].complete_time = current_time;

                //calc and store TT/WT
                // TT = CT - AT
                taskArray[maxIdx].turnaround_time = taskArray[maxIdx].complete_time - taskArray[maxIdx].arrival_time;
                // WT = TT - BT
                taskArray[maxIdx].waiting_time = taskArray[maxIdx].turnaround_time - taskArray[maxIdx].burst_time;
            }
        }
        else
        {
            current_time++;
        }
    }
    printf("\n\n");

    int maxWT = 0;
    int maxTT = 0;
    for(int ii=0;ii<numProc;ii++)
    {
        maxWT += taskArray[ii].waiting_time;
        maxTT += taskArray[ii].turnaround_time;
    }

    float avgWT=0,avgTT=0;
    avgTT = (float) maxTT / numProc;
    avgWT  = (float) maxWT / numProc;
```

```
    sprintf(buffer2, "PP: avgTT = %f | avgWT = %f\n", avgTT, avgWT);
    // Reset variables.
    current_time = 0;
    numProc = 0;
    tot_burst = 0;
    bufferCount++;
    // Signal that thread has written to buffer.
    pthread_cond_signal(&calculated);
    pthread_mutex_unlock(&lock);
    return NULL;
}

// ================== Thread B: SRTF ==================
void* srtf()
{
    char output[] = "SRTF Simulation: \n";
    // Start operation:
    Task taskArray[50];
    pthread_mutex_lock(&lock);

    while(buffer1[0] == '\0' && done == 0)
    {
        pthread_cond_wait(&read, &lock); // This gets unlocked when filename received.
    }

    readFile(buffer1, taskArray);
    printf("%s", output);

    // Perform scheduling
    int maxIdx = -5;
    while(current_time != tot_burst)
    {
        maxIdx = -5; // Reset max, find new max.
        taskArray[maxIdx].time_left = 9999;
        for(int ii=0;ii<numProc;ii++)
        {

            if(taskArray[ii].arrival_time <= current_time && taskArray[ii].time_left != 0 && taskArray[ii].time_left <
taskArray[maxIdx].time_left)
            {
                maxIdx = ii;
            }
        }

        // Max/Current should be initialised per timestep.
        printf("P%d|", taskArray[maxIdx].pid);
```

```
        // If a task was found i.e. if maxIdx != -5
        if(maxIdx != -5)
        {
            // Decrement remaining time.
            taskArray[maxIdx].time_left--;
            // Increment total time
            current_time++;
            if(taskArray[maxIdx].time_left == 0)
            {
                //add CT
                taskArray[maxIdx].complete_time = current_time;

                //calc and store TT/WT
                // TT = CT - AT
                taskArray[maxIdx].turnaround_time = taskArray[maxIdx].complete_time - taskArray[maxIdx].arrival_time;
                // WT = TT - BT
                taskArray[maxIdx].waiting_time = taskArray[maxIdx].turnaround_time - taskArray[maxIdx].burst_time;
            }
        }
        else
        {
            current_time++;
        }
    }
    printf("\n\n");

    int maxWT = 0;
    int maxTT = 0;
    for(int ii=0;ii<numProc;ii++)
    {
        maxWT += taskArray[ii].waiting_time;
        maxTT += taskArray[ii].turnaround_time;
    }

    float avgWT=0,avgTT=0;
    avgTT = (float) maxTT / numProc;
    avgWT  = (float) maxWT / numProc;

    sprintf(buffer2, "SRTF: avgTT = %f | avgWT = %f\n", avgTT, avgWT);
    // Reset variables.
    current_time = 0;
    numProc = 0;
    tot_burst = 0;
    bufferCount++;
    // Signal that thread has written to buffer.
    pthread_cond_signal(&calculated);
    // Unlock mutex.
```

```
    pthread_mutex_unlock(&lock);
    return NULL;
}

// ================== File IO ==================
int readFile(char* argv, Task* taskArray)
{
    FILE* file; /* File pointer to the file stream */

    int counter = 0;
    // ==============
    //Opens file with a name specified by user in the command line
    file = fopen(argv, "r");

    if(file == NULL) /* If file cannot open */
    {
        printf("Error - could not open file. ");
    }
    else // Read line by line.
    {
        while(!feof(file)) /* && (error == FALSE)) */
        {
            // Reading Format: {Arrival} <space> {Burst} <space> {Priority}
            fscanf(file, "%d %d %d", &(taskArray[counter].arrival_time), (&taskArray[counter].burst_time),
(&taskArray[counter].priority));
            /* Read in the file and assort each part of the file to its according part in our taskArray - array of
structs */
            taskArray[counter].pid = counter + 1;
            taskArray[counter].time_left = taskArray[counter].burst_time; // Burst = Orig burst time (constant),
timeleft changes upon bursts.
            tot_burst += taskArray[counter].burst_time;
            counter++;
            numProc++;
        }
        if(ferror(file))
        {
            perror("Error reading from file. \n");
        }
        fclose(file);
    }
    return NULL;
}
```

2. Mutual Exclusion:

Mutual Exclusion was achieved by using the POSIX Pthreads Library, specifically by utilising mutexes and condition variables. These mutex and condition variables are global to the program. I use mutexes when performing the scheduling process from beginning to end, to ensure that these operations are atomic, meaning that these as well as for halting the program in the necessary places to ensure our requirement for buffers storing only one pair of calculated values at a time. All these variabels were initialised before thread creation and destroyed after thread completion in the main method/parent thread.

Going further in detail with this, the mutex 'lock' was used to guarantee the atomicity of the scheduling; done by performing a pthread_mutex_lock with the lock mutex before the main scheduling 'critical section' and pthread_mutex_unlock with lock afterwards. Between the locking and unlocking of the 'lock' mutex is my "critical section" which involves important code such as calculating the scheduler values, printing to the terminal and modifying shared resources like global variables and Buffer 2. The same mutex was used for both Priority Preemptive and SRTF schedulers, meaning that both schedulers can run in unison, without being to be preempted, whilst they alter shared resources and calculate values.

As for the two buffers in the program, buffer1 stores the filename whilst buffer2 stores one set of calculated values at a time. Synchronisation between the user and the two threads were handled via condition variables; altered via the pthread_cond methods. The threads are to block until the user enters a filename; thus the threads perform a pthread_cond_wait() on a condition variable, *until* a filename is received. The threads will know when a filename is received, when the parent thread changes the condition variable via pthread_cond_broadcast(), which alerts all threads listening on that condition variable, that a change has occurred; indicating they can stop blocking, and can proceed to reading in the file, and performing the critical section (as aforementioned, atomically). Once it broadcasts, the parent thread will perform pthread_cond_wait() until buffer2 is populated by one of the schedulers i.e. waits on a condition variable. Buffer2 can only store one pair of calculated values at a time; thus, once one thread finishes calculating its pair of values and stores to the (global) buffer2, it signals the condition variable that parent waits on, meaning the parent stops waiting and stores the result from the buffer. This process is repeated once again (parent waits once more), and the second pair of values are stored into the buffer. These operations are all atomic meaning race conditions are accounted for, and a process can't write to the buffer twice or not write to the buffer at all.

Shared resources:

- Buffer1 which stores filename, & Buffer2 which stores one pair of calculated values; used by the parent thread as well as both scheduling threads. Same goes for bufferCount.

- The two scheduling methods share three variables among them which are used to synchronise the number of total processes, the total burst and current time within the scheduler method. This could've been kept non-global, but I made it global to ensure that these variables are synchronised properly between the two schedulers running concurrently, and to make sure that they run atomically (otherwise for e.g. if they weren't atomic, the current time would increment to a crazy high number as the two threads access a shared variable at the same time).

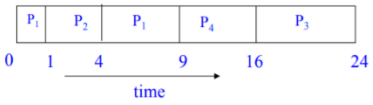
3. Known Issues:
   - As part of my assumptions, I assume we don't need to perform any error handling like other computing units, so I don't account for handling any invalid data. The assignment specification doesn't specify the need to account for validating input/files unlike other units.

4. Sample Cases:

Example #1: SRTF Program (reference – Lecture Slides):

Process	Arrival time	Burst time
1	0	6
2	1	3
3	2	8
4	3	7

The Gantt chart for the schedule:

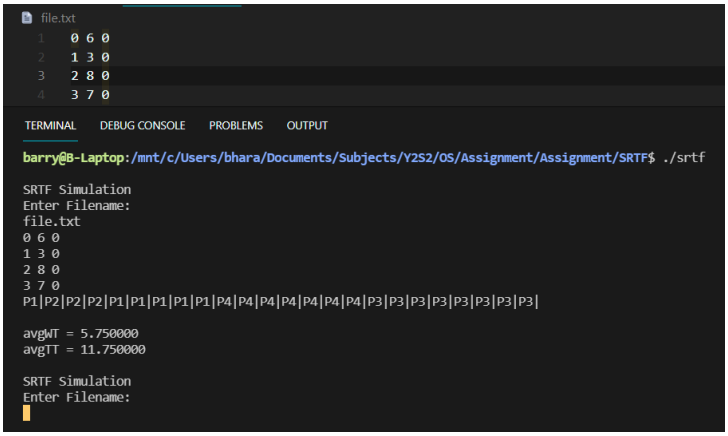


Waiting time for P1 = 4-1; P2 = 1-1; P3 = 16-2; P4 = 9-3

Average waiting time = $\frac{3+0+14+6}{4} = \frac{23}{4} = 5.75$

Average turnaround time: $\frac{(9-0)+(4-1)+(24-2)+(16-3)}{4} = \frac{47}{4} = 11.75$

Note: waiting time of a process is its turnaround time *minus* its burst time.

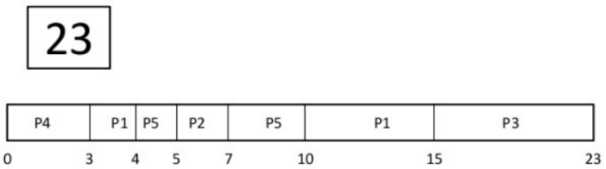


Example #2: SRTF Program – Online resource – guru99.com

Consider the following five process:

Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

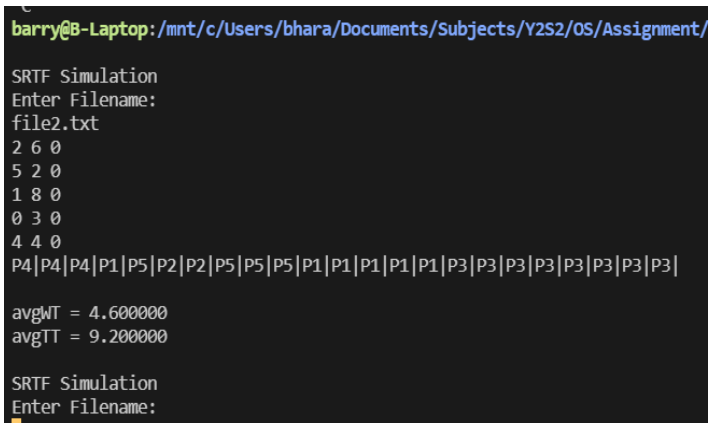
Step 10) At time =23, P3 finishes its execution.



Step 11) Let's calculate the average waiting time for above example.

```
Wait time
P4= 0-0=0
P1=  (3-2) + 6 =7
P2= 5-5 = 0
P5= 4-4+2 =2
P3= 15-1 = 14
```

```
Average Waiting Time = 0+7+0+2+14/5 = 23/5 =4.6
```

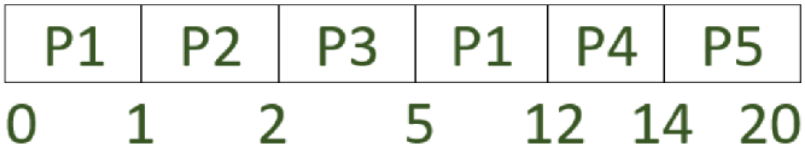


Example #3: Priority Preemptive Program – Geeks4Geeks problem:

Process	Arrival Time	Burst Time	Priority
P1	0	8	3
P2	1	1	1
P3	2	3	2
P4	3	2	3
P5	4	6	4

2. Priority Preemptive scheduling :

The Gantt chart will look like:



Average waiting time (AWT),

```
= ((5-1) + (1-1) + (2-2) + (12-3) + (14-4)) / 5
= 23/5
= 4.6
```

Average turnaround time (TAT),

```
= ((12-0) + (2-1) + (5-2) + (14-3) + (20-4)) / 5
= 43 / 5
= 8.5
```

NOTE: Average Turnaround Time should be 8.6, as 43/5 = 8.6 (not 8.5).

```
PP Simulation
Enter Filename:
file.txt
0 8 3
1 1 1
2 3 2
3 2 3
4 6 4
P1|P2|P3|P3|P3|P1|P1|P1|P1|P1|P1|P1|P4|P4|P5|P5|P5|P5|P5|P5|

avgWT = 4.600000
avgTT = 8.600000

PP Simulation
Enter Filename:
```

Example #4: Thread synchronisation:

```
barry@B-Laptop:/mnt/c/Users/bhara/Documents/Subjects/Y2S2/OS/Assignment/Assignment/Threading$ ./schedsim
Enter Filename:
d
SRTF Simulation:
P1|P2|P3|P3|P3|P4|P4|P5|P5|P5|P5|P5|P5|P1|P1|P1|P1|P1|P1|P1|

PP Simulation
P1|P2|P3|P3|P3|P1|P1|P1|P1|P1|P1|P1|P4|P4|P5|P5|P5|P5|P5|P5|


SRTF: avgTT = 7.400000 | avgWT = 3.400000
PP: avgTT = 8.600000 | avgWT = 4.600000

Enter Filename:
QUIT
SRTF terminating.....
PP terminating.....
```

Above has SRTF then PP (i.e. SRTF acquires mutex lock first); below is an example of the opposite:

```
Enter Filename:
d
PP Simulation
P1|P2|P3|P3|P3|P1|P1|P1|P1|P1|P1|P1|P4|P4|P5|P5|P5|P5|P5|P5|

SRTF Simulation:
P1|P2|P3|P3|P3|P4|P4|P5|P5|P5|P5|P5|P5|P1|P1|P1|P1|P1|P1|P1|


PP: avgTT = 8.600000 | avgWT = 4.600000
SRTF: avgTT = 7.400000 | avgWT = 3.400000

Enter Filename:
```