Operating Systems Assignment

ASSIGNMENT REPORT

Name: Jason Tan Thong Shen

STUDENT ID: 20060534

[THIS SUBMITTED ASSIGNMENT IS MY OWN WORK]

Report length: 47 pages

Report Contents:

(i) Source code for all programs from “PP”, “STRF”, “MultiThreading” folders.
(ii) README.txt file location. (compiling instructions)
(iii) Discussion on how any mutual exclusion is achieved and what processes/threads access the shared resources.
(iv) Description of any cases for which your program is not working correctly or how you test your program that make you believe it works perfectly.
(v) Sample inputs and outputs from your running programs. (on terminal)

(i) Source code for all programs from “PP”, “STRF”, “MultiThreading” folders.

Folder: PP (has pp.c, pp.h)

[pp.c]----------

```
#include "pp.h"


int main()
{
   char fileName[10]; /*file name no longer that 10 characters*/

   do
   {
      printf("exclude \".txt\" from file name\nPP simulation: ");
      scanf("%s", fileName);
      if( strcmp(fileName, QUIT) != 0 )
      {
         readFile( fileName ); /*read file when received file name*/
      }
   }while( strcmp(fileName, QUIT) != 0 );

   return 0;
}




void readFile( char fileName[] )
{
   Task task; /*to declare the Task struct for usage*/
   int time, burst_t, priority, task_count, pid;
```

```
   FILE *f;

   strcat(fileName, ".txt");
   f = fopen(fileName, "r");

   if( f == NULL )
   {
      perror("Error opening file");
   }
   else
   {
      task_count = 0;
      pid = 'A';
      while( fscanf(f, "%d %d %d", &time, &burst_t, &priority) !=EOF )
      {
         task[task_count].pid = pid;
         task[task_count].time = time;
         task[task_count].burst_t = burst_t;
         task[task_count].priority = priority;
         task[task_count].rt = burst_t;
         task[task_count].ct = NONE;
         task[task_count].tat = NONE;
         task[task_count].wt = NONE;

         task_count++;
         pid++;
      }
      if( ferror(f) )
      {
         perror("Error reading file");
      }
      sortArrivalTime( task_count, task );
      ppStart( task_count, task ); /*start the pp scheduling*/
      displayDetails( task_count, task ); /*display table & results*/
   }
   fclose(f);
}




/*sorting arrival time from 0 to n (increasing order)*/
void sortArrivalTime( int task_count, Task task )
```

```
{
   Task tempTask;
   int i, j;

   i = 0;
   while( i < task_count - 1 )
   {
      j = i + 1;
      while( j < task_count )
      {
         if( task[i].time > task[j].time )
         {/*swap places if time j is smaller than time i*/
            tempTask[0] = task[i];
            task[i] = task[j];
            task[j] = tempTask[0];
         }
         j++;
      }
      i++;
   }
}




void displayDetails( int task_count, Task task )
{
   int i;
   double avgWT, avgTAT, totalWT, totalTAT;
   printf("Number of Task: %d\n\n", task_count);
   printf("pid\ttime\tburst\tpriority   rt\tct\ttat\twt\n");
   printf("---\t----\t-----\t--------   --\t--\t---\t--\n");

   totalWT = 0;
   totalTAT = 0;
   for( i = 0; i < task_count; i++ )/*display tasks data*/
   {
      printf("%c\t", task[i].pid);
      printf("%d\t", task[i].time);
      printf("%d\t", task[i].burst_t);
      printf("%d\t", task[i].priority);
      printf("   %d\t", task[i].rt);
```

```
        printf("%d\t", task[i].ct);
        printf("%d\t", task[i].tat);
        printf("%d\n", task[i].wt);

        /*calculate total waiting time and turnaround time*/
        totalWT += task[i].wt;
        totalTAT += task[i].tat;
    }
    /*calculte average waiting time and turnaround time*/
    avgWT = totalWT / task_count;
    avgTAT = totalTAT / task_count;

    printf("\nAVERAGE WAITING TIME: %f\n", avgWT);
    printf("\nAVERAGE TURNAROUND TIME: %f\n", avgTAT);
}




void ppStart( int task_count, Task task )
{
    Task readyQueue, running;
    int curTime, i, ready_count, burst_count;
    curTime = 0;
    running[0].empty = true;

    /*find total burst time*/
    burst_count = findTotalBurst( task_count, task );
    printf("Total burst time: %d\n", burst_count);

    /*displaying Gantt Chart-------------------------------------------*/
    printf("|");
    ready_count = 0;
    do
    {
        /*1. find the new task on current time*/
        ready_count = findNewTask( task_count, task, readyQueue, curTime,
                                   ready_count );
        /*2. find the highest task*/
        ready_count = findHighest( readyQueue, running, ready_count,
                                   &burst_count );
        /*3. decrement the remaining time and/or removing the running to
            mark it as empty when running is done*/
```

```
      ready_count = decrementRT( task_count, task, readyQueue, running,
                           curTime, ready_count );
      printf("|");
      curTime++; /*go to next time*/
   }while( curTime < burst_count ); /*loop through each time*/
   printf("\n");

   for( i = 0; i <= burst_count; i++ ) /*for Gantt Chart time interval*/
      printf("%d ", i );
   printf("\n");
   /*----------------------------------------------------------------*/
}




int findTotalBurst( int task_count, Task task )
{
   /*find total burst time*/
   int i;
   int burst_count = 0;
   for( i = 0; i < task_count; i++ )
   {
      burst_count += task[i].burst_t;
   }
   return burst_count;
}




int findNewTask( int task_count, Task task, Task readyQueue, int curTime,
              int ready_count )
{
   int i;
   for( i = 0; i < task_count; i++ )
   {
      if( task[i].time == curTime ) /*find new task for current time*/
      {
         readyQueue[ready_count] = task[i]; /*insert to ready queue*/
         readyQueue[ready_count].empty = false;
```

```
            ready_count++;
        }
    }
    return ready_count;
}
```

```
int findHighest( Task readyQueue, Task running, int ready_count,
                int *burst_count )
{
    int i, highPrioNum, highPrioIdx;

    /*now check which process in the queue is highest*/
    if( ready_count >= ONE )/*have multiple task to choose from*/
    {
        highPrioNum = readyQueue[0].priority; /*initial priority NUMBER*/
        highPrioIdx = 0; /*initial priority INDEX*/
        for( i = 1; i < ready_count; i++ )
        {
            /*if found another new higher priority*/
            if( highPrioNum > readyQueue[i].priority )
            {
                /*new higher priority found with different pid*/
                highPrioNum = readyQueue[i].priority;
                highPrioIdx = i;
            }
        }
        /*if found same priority, no need preemption, because
         first come first serve basis, so the current running
         continue running without changing to another process
         of same priority*/
        if( !running[0].empty )
        {
            /*only assign process when have higher priority*/
            if( running[0].priority > readyQueue[highPrioIdx].priority )
            {
                ready_count = determineRun( readyQueue, running,
                                ready_count, highPrioIdx );
            }
            else
```

```
          {
          }
        }
        else if( running[0].empty )
        {/*only assign process if running is empty*/
           ready_count = determineRun( readyQueue, running,
                                ready_count, highPrioIdx );
        }
     }
     else if( ready_count == EMPTY && running[0].empty )
     {
         *burst_count = *burst_count + 1;
     }
     /*if ready_count is empty means no ready queue allowed to enter
       running process, because empty*/
    return ready_count;
}




int determineRun( Task readyQueue, Task running, int ready_count,
                int highPrioIdx )
{
    int i;
    /*determine how processes should run*/
     if( !running[0].empty )
     {/*have running process, need to save progress first*/
       /*means need to preemp the current running process to make way
        for the higher priority task*/

        /*1. save progress by adding to ready queue*/
        readyQueue[ready_count] = running[0];
        ready_count++;

        /*2. change to another process*/
        running[0] = readyQueue[highPrioIdx];
        running[0].highPrioIdx = highPrioIdx;
        running[0].empty = false;

        /*3. delete the "assigned ready queue to running" from the
            queue and shift the array forward like a queue*/
```

```
      for( i = running[0].highPrioIdx; i < ready_count - 1; i++ )
      {
         readyQueue[i] = readyQueue[i+1];
      }
      ready_count--;
    }
    else if( running[0].empty )
    {/*no process, just assign process*/

      /*1. assign a process*/
      running[0] = readyQueue[highPrioIdx];
      running[0].highPrioIdx = highPrioIdx;
      running[0].empty = false; /*means no more empty*/

      /*2. delete the "assigned ready queue to running" from the queue
        and shift the array forward like a queue*/
      for( i = running[0].highPrioIdx; i < ready_count - 1; i++ )
      {
         readyQueue[i] = readyQueue[i+1];
      }
      ready_count--;
   }
   return ready_count;
}
```

```
int decrementRT( int task_count, Task task, Task readyQueue,
            Task running, int curTime, int ready_count )
{
   int count;
   if( !running[0].empty )
   {
      /*if still have remaining time*/
      if( running[0].rt > EMPTY )
      {
         running[0].rt--; /*decrement remaining time of running*/
         printf("%c", running[0].pid);

        /*if reaches zero after decrement, save data, & straight away
         set running to empty for other process to use*/
```

```
        if( running[0].rt == EMPTY )/*if no more remaining time*/
        {
           /*1. calculate completion, turnaround, waiting time*/
           running[0].ct = curTime + 1; /*because get next time*/
           running[0].tat = running[0].ct - running[0].time;
           running[0].wt = running[0].tat - running[0].burst_t;

           /*2. find original task (not running task or ready queue)
               to update their rt, ct, tat, wt*/
           for( count = 0; count < task_count; count++ )
           {
              /*found the running procoess's original/initial task*/
              if( running[0].pid == task[count].pid )
              {
                 /*3. transfer the info to original task from
                     running process*/
                 task[count].rt = running[0].rt;
                 task[count].ct = running[0].ct;
                 task[count].tat = running[0].tat;
                 task[count].wt = running[0].wt;
                 running[0].empty = true;/*4. set running to empty*/
              }
           }
        }
     }
  }
  return ready_count;
}
```

[pp.h]----------

```
#ifndef PP_H
#define PP_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>

#define NONE -1
#define EMPTY 0
#define QUIT "QUIT"
#define ONE 1
```

```
typedef struct Task{
   int time, burst_t, priority;
   char pid;
   int rt, ct, tat, wt;
   bool empty; /*only for running process*/
   int highPrioIdx; /*only for decrementRT to use for deleting ready
                queue array index*/
}Task[50]; /*maximum number of task is 50*/


void readFile( char fileName[] );
void sortArrivalTime( int task_count, Task task );
void displayDetails( int task_count, Task task );
void ppStart( int task_count, Task task );
int findTotalBurst( int task_count, Task task );
int findNewTask( int task_count, Task task, Task readyQueue, int curTime,
             int ready_count );
int findHighest( Task readyQueue, Task running, int ready_count,
            int *burst_count );
int determineRun( Task readyQueue, Task running, int ready_count,
             int highPrioIdx );
int decrementRT( int task_count, Task task, Task readyQueue,
             Task running, int curTime, int ready_count );
#endif
```

Folder: STRF (has srtf.c, srtf.h)

[srtf.c]----------

```
#include "srtf.h"


int main()
{
   char fileName[10]; /*file name no longer that 10 characters*/

   do
   {
      printf("Exclude \".txt\" from file name\nSRTF simulation: ");
      scanf("%s", fileName);
```

```
    if( strcmp(fileName, QUIT) != 0 )
    {
      readFile( fileName ); /*read file when received file name*/
    }
  }while( strcmp(fileName, QUIT) != 0 );

  return 0;
}




void readFile( char fileName[] )
{
  Task task; /*to declare the Task struct for usage*/
  int time, burst_t, priority, task_count, pid;
  FILE *f;

  strcat(fileName, ".txt");

  f = fopen(fileName, "r");

  if( f == NULL )
  {
    perror("Error opening file");
  }
  else
  {
    task_count = 0;
    pid = 'A';
    while( fscanf(f, "%d %d %d", &time, &burst_t, &priority) !=EOF )
    {
      task[task_count].pid = pid;
      task[task_count].time = time;
      task[task_count].burst_t = burst_t;
      task[task_count].priority = priority;
      task[task_count].rt = burst_t;
      task[task_count].ct = NONE;
      task[task_count].tat = NONE;
      task[task_count].wt = NONE;

      task_count++;
```

```
            pid++;
        }
        if( ferror(f) )
        {
            perror("Error reading file");
        }
        sortArrivalTime( task_count, task );
        srtfStart( task_count, task ); /*start the srtf scheduling*/
        displayDetails( task_count, task ); /*display table & results*/
    }
    fclose(f);
}




/*sorting arrival time from 0 to n (increasing order)*/
void sortArrivalTime( int task_count, Task task )
{
    Task tempTask;
    int i, j;

    i = 0;
    while( i < task_count - 1 )
    {
        j = i + 1;
        while( j < task_count )
        {
            if( task[i].time > task[j].time )
            {/*swap places if time j is smaller than time i*/
                tempTask[0] = task[i];
                task[i] = task[j];
                task[j] = tempTask[0];
            }
            j++;
        }
        i++;
    }
}
```

```
void displayDetails( int task_count, Task task )
{
   int i;
   double avgWT, avgTAT, totalWT, totalTAT;
   printf("Number of Task: %d\n\n", task_count);
   printf("pid\ttime\tburst\tpriority   rt\tct\ttat\twt\n");
   printf("---\t----\t-----\t--------   --\t--\t---\t--\n");

   totalWT = 0;
   totalTAT = 0;
   for( i = 0; i < task_count; i++ )/*display tasks data*/
   {
      printf("%c\t", task[i].pid);
      printf("%d\t", task[i].time);
      printf("%d\t", task[i].burst_t);
      printf("%d\t", task[i].priority);
      printf("   %d\t", task[i].rt);
      printf("%d\t", task[i].ct);
      printf("%d\t", task[i].tat);
      printf("%d\n", task[i].wt);

      /*calculate total waiting time and turnaround time*/
      totalWT += task[i].wt;
      totalTAT += task[i].tat;
   }
   /*calculte average waiting time and turnaround time*/
   avgWT = totalWT / task_count;
   avgTAT = totalTAT / task_count;

   printf("\nAVERAGE WAITING TIME: %f\n", avgWT);
   printf("\nAVERAGE TURNAROUND TIME: %f\n", avgTAT);
}
```

```
void srtfStart( int task_count, Task task )
{
   Task readyQueue, running;
   int curTime, i, ready_count, burst_count;
   curTime = 0;
```

```
   running[0].empty = true;

   /*find total burst time*/
   burst_count = findTotalBurst( task_count, task );
   printf("Total burst time: %d\n", burst_count);

   /*displaying Gantt Chart-----------------------------------------*/
   printf("|");
   ready_count = 0;
   do
   {
      /*1. find the new task on current time*/
      ready_count = findNewTask( task_count, task, readyQueue, curTime,
                        ready_count );
      /*2. find the highest task*/
      ready_count = findHighest( readyQueue, running, ready_count,
                        &burst_count );
      /*3. decrement the remaining time and/or removing the running to
          mark it as empty when running is done*/
      ready_count = decrementRT( task_count, task, readyQueue, running,
                        curTime, ready_count );
      printf("|");
      curTime++; /*go to next time*/
   }while( curTime < burst_count ); /*loop through each time*/
   printf("\n");

   for( i = 0; i <= burst_count; i++ ) /*for Gantt Chart time interval*/
      printf("%d ", i );
   printf("\n");
   /*-----------------------------------------------------------------*/
}




int findTotalBurst( int task_count, Task task )
{
   /*find total burst time*/
   int i;
   int burst_count = 0;
   for( i = 0; i < task_count; i++ )
   {
```

```
      burst_count += task[i].burst_t;
   }
   return burst_count;
}
```

```
int findNewTask( int task_count, Task task, Task readyQueue, int curTime,
             int ready_count )
{
   int i;
   for( i = 0; i < task_count; i++ )
   {
      if( task[i].time == curTime ) /*find new task for current time*/
      {
         readyQueue[ready_count] = task[i]; /*insert to ready queue*/
         readyQueue[ready_count].empty = false;
         ready_count++;
      }
   }
   return ready_count;
}
```

```
int findHighest( Task readyQueue, Task running, int ready_count,
             int *burst_count )
{
    int i, highPrioRT, highPrioIdx;

    /*now check which process in the queue is highest*/
    if( ready_count >= ONE )/*have multiple task to choose from*/
    {
       highPrioRT = readyQueue[0].rt; /*initial priority NUMBER*/
       highPrioIdx = 0; /*initial priority INDEX*/
       for( i = 1; i < ready_count; i++ )
       {
          /*if found another new higher priority*/
          if( highPrioRT > readyQueue[i].rt )
```

```
            {
               /*new higher priority found with different pid*/
               highPrioRT = readyQueue[i].rt;
               highPrioIdx = i;
            }
         }
         /*if found same priority, no need preemption, because
          first come first serve basis, so the current running
          continue running without changing to another process
          of same priority*/
         if( !running[0].empty )
         {
            /*only assign process when have higher priority*/
            if( running[0].rt > readyQueue[highPrioIdx].rt )
            {
               ready_count = determineRun( readyQueue, running,
                                 ready_count, highPrioIdx );
            }
            else
            {
            }
         }
         else if( running[0].empty )
         {/*only assign process if running is empty*/
            ready_count = determineRun( readyQueue, running,
                               ready_count, highPrioIdx );
         }
      }
      else if( ready_count == EMPTY && running[0].empty )
      {
          *burst_count = *burst_count + 1;
      }
      /*if ready_count is empty means no ready queue allowed to enter
        running process, because empty*/
     return ready_count;
}




int determineRun( Task readyQueue, Task running, int ready_count,
                  int highPrioIdx )
```

```
{
   int i;
   /*determine how processes should run*/
    if( !running[0].empty )
    {/*have running process, need to save progress first*/
      /*means need to preemp the current running process to make way
       for the higher priority task*/

       /*1. save progress by adding to ready queue*/
       readyQueue[ready_count] = running[0];
       ready_count++;

       /*2. change to another process*/
       running[0] = readyQueue[highPrioIdx];
       running[0].highPrioIdx = highPrioIdx;
       running[0].empty = false;

       /*3. delete the "assigned ready queue to running" from the
            queue and shift the array forward like a queue*/
       for( i = running[0].highPrioIdx; i < ready_count - 1; i++ )
       {
          readyQueue[i] = readyQueue[i+1];
       }
       ready_count--;
    }
    else if( running[0].empty )
    {/*no process, just assign process*/

       /*1. assign a process*/
       running[0] = readyQueue[highPrioIdx];
       running[0].highPrioIdx = highPrioIdx;
       running[0].empty = false; /*means no more empty*/

       /*2. delete the "assigned ready queue to running" from the queue
         and shift the array forward like a queue*/
       for( i = running[0].highPrioIdx; i < ready_count - 1; i++ )
       {
          readyQueue[i] = readyQueue[i+1];
       }
       ready_count--;
   }
   return ready_count;
}
```

```
int decrementRT( int task_count, Task task, Task readyQueue,
            Task running, int curTime, int ready_count )
{
   int count;
   if( !running[0].empty )
   {
      /*if still have remaining time*/
      if( running[0].rt > EMPTY )
      {
         running[0].rt--; /*decrement remaining time of running*/
         printf("%c", running[0].pid);/*print for gantt chart*/

         /*1. find original task (not running task or ready queue)
             to update their rt, ct, tat, wt*/
         for( count = 0; count < task_count; count++ )
         {
            /*found the running procoess's original/initial task*/
            if( running[0].pid == task[count].pid )
            {
              /*2. constantly update remaining time into task after
                  every decrement*/
               task[count].rt = running[0].rt;

               /*if reaches zero after decrement, save data, &
                 straight away set running to empty for other process
                 to use*/
               if( running[0].rt == EMPTY )/*if no more RT*/
               {
               /*A. calculate completion, turnaround, waiting time*/
                  running[0].ct = curTime + 1;
                  /*because get next time ^^^*/
                  running[0].tat = running[0].ct - running[0].time;
                  running[0].wt = running[0].tat -
                              running[0].burst_t;

                  /*B. transfer the info to original task from
                      running process*/
                  task[count].ct = running[0].ct;
                  task[count].tat = running[0].tat;
```

```
                task[count].wt = running[0].wt;
                running[0].empty = true;/*C. set running to empty*/
              }
            }
          }
        }
    }
    return ready_count;
}
```

[srtf.h]----------

```
#ifndef SRTF_H
#define SRTF_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>

#define NONE -1
#define EMPTY 0
#define QUIT "QUIT"
#define ONE 1

typedef struct Task{
    int time, burst_t, priority;
    char pid;
    int rt, ct, tat, wt;
    bool empty; /*only for running process*/
    int highPrioIdx; /*only for decrementRT to use for deleting ready
                queue array index*/
}Task[50]; /*maximum number of task is 50*/


void readFile( char fileName[] );
void sortArrivalTime( int task_count, Task task );
void displayDetails( int task_count, Task task );
void srtfStart( int task_count, Task task );
int findTotalBurst( int task_count, Task task );
int findNewTask( int task_count, Task task, Task readyQueue, int curTime,
            int ready_count );
int findHighest( Task readyQueue, Task running, int ready_count,
```

```
            int *burst_count );
int determineRun( Task readyQueue, Task running, int ready_count,
            int highPrioIdx );
int decrementRT( int task_count, Task task, Task readyQueue,
            Task running, int curTime, int ready_count );
#endif
```

Folder: MultiThreading (has multiThread.c, multiThread.h, pp.c, pp.h, srtf.c, srtf.h)

[multiThread.c]----------

```
#include "multiThread.h"


/*shared memory*/
int countB1, countB2; /*for the buffer*/
char fileName[10]; /*for the writer method*/
Buffer buffer1;
Buffer buffer2;

pthread_mutex_t keyOne = PTHREAD_MUTEX_INITIALIZER;/*mutex for buffer1*/
pthread_mutex_t keyTwo = PTHREAD_MUTEX_INITIALIZER;/*mutex for buffer2*/
pthread_mutex_t uiKey = PTHREAD_MUTEX_INITIALIZER;/*mutex for UI*/

pthread_cond_t isFull = PTHREAD_COND_INITIALIZER;/*if full buffer1*/
pthread_cond_t isEmpty = PTHREAD_COND_INITIALIZER;/*if empty buffer1*/

pthread_cond_t isFullBTwo = PTHREAD_COND_INITIALIZER;/*full buffer2*/
pthread_cond_t isEmptyBTwo = PTHREAD_COND_INITIALIZER;/*empty buffer2*/



int main()
{
   int i; /*for loop*/
   /*two threads*/
   pthread_t ppThread;
   pthread_t srtfThread;

   countB1 = 0; /*initial zero file name in buffer*/
   countB2 = 0;
```

```
do
{
   /*user input with uiKey*/
   pthread_mutex_lock( &uiKey );
   printf("\nInclude \".txt\" after file name\nfile: ");
   scanf("%s", fileName);
   pthread_mutex_unlock( &uiKey );
   /*----------------------------*/

   /*producer for buffer1**************************************/
   pthread_mutex_lock( &keyOne );
   /*means full, need to wait for isEmpty from 2 consumers*/
   while( countB1 == ONE ) /*for condition variable*/
   {
      pthread_cond_wait( &isEmpty, &keyOne );
      /*equivilent to:
        pthread_mutex_unlock( &keyOne );
        then wait on signal "isEmpty" then call again
        pthread_mutex_lock( &keyOne );*/
   }
   /*critical section*/
   strcpy(buffer1[BUFFER_SIZE-1].fileName, fileName);
   countB1 = 1;
   pthread_cond_signal( &isFull );/*signal consumer isFull*/
   pthread_mutex_unlock( &keyOne );
   /*************************************************************/

   if( strcmp(fileName, QUIT) != ZERO )
   {
      /*creating child threads: ppThread and srtfThread*/
      /*consumers for buffer1*/
      /*& producers for buffer2*/
      pthread_create(&ppThread, NULL, ppFunc, NULL );
      pthread_create(&srtfThread, NULL, srtfFunc, NULL );
   }
   else if( strcmp(fileName, QUIT) == ZERO )
   {
      /*wait until the thread is done before we exit*/
      if( pthread_join( ppThread, NULL ) == ZERO )
      {
         printf("PP: terminate.\n");
      }
      if( pthread_join( srtfThread, NULL) == ZERO )
      {
```

```
            printf("SRTF: terminate.\n");
        }
        pthread_exit(NULL);
        /*forces main function to wait for the threats created to
          complete its work, if not main function will exit too fast*/
      }
      /******get AVG time from buffer2 [consumer]*****************/
      for( i = 0; i < 2; i++ )
      {
        sleep(1);/*to allow time for other sections in the child thread
                to acquire lock before this section in the parent
                thread*/
        pthread_mutex_lock( &uiKey );
        pthread_mutex_lock( &keyTwo );
        /*means empty, need to wait for isFullBTwo from producer*/
        while( countB2 == ZERO )
        {
          pthread_cond_wait( &isFullBTwo, &keyTwo);
        }
        if( buffer2[BUFFER_SIZE-1].type == PP )
        {
          printf("PP: ");
        }
        else if( buffer2[BUFFER_SIZE-1].type == SRTF )
        {
          printf("SRTF: ");
        }
        /*critical seciton*/
        printf("the average turnaround time = %f, ",
              buffer2[BUFFER_SIZE-1].avgTAT);
        printf("the average waiting time = %f\n",
              buffer2[BUFFER_SIZE-1].avgWT);
        countB2--;
        pthread_cond_signal( &isEmptyBTwo ); /*signal producer
                                    isEmptyBTwo*/
        pthread_mutex_unlock( &keyTwo );
        pthread_mutex_unlock( &uiKey );
      }
      /*********************************************************/
    }while( strcmp(fileName, QUIT) != ZERO );
    return 0;
}
```

```
void *ppFunc(void *none)
{
   /*consumer for buffer1*************************************/
   pthread_mutex_lock( &keyOne );
   /*means empty, need to wait for isFull from producer*/
   while( countB1 == NONE ) /*for condition variable*/
   {
      pthread_cond_wait( &isFull, &keyOne);
      /*equivilent to:
        pthread_mutex_unlock( &keyOne );
        then wait on signal "isFull" then call again
        pthread_mutex_lock( &keyOne );*/
   }
   /*critical section*/
   srtfReadFile( buffer1[BUFFER_SIZE-1].fileName );
   countB1--;
   pthread_cond_signal( &isEmpty ); /*signal producer isEmpty*/
   pthread_mutex_unlock( &keyOne );
   /*********************************************************/
   return NULL;
}
```

```
void *srtfFunc(void *none)
{
   /*consumer for buffer1*************************************/
   pthread_mutex_lock( &keyOne );
   /*means empty, need to wait for isFull from producer*/
   while( countB1 == NONE )
   {
      pthread_cond_wait( &isFull, &keyOne );
      /*equivilent to:
        pthread_mutex_unlock( &keyOne );
        then wait on signal "isFull" then call again
        pthread_mutex_lock( &keyOne );*/
   }
```

```
   /*critical section*/
   ppReadFile( buffer1[BUFFER_SIZE-1].fileName );
   countB1--;
   pthread_cond_signal( &isEmpty ); /*signal producer isEmpty*/
   pthread_mutex_unlock( &keyOne );
   /*********************************************************/
   return NULL;
}
```

[multiThread.h]----------

```
#ifndef MULTITHREAD_H
#define MULTITHREAD_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <pthread.h>
#include "pp.h"
#include "srtf.h"

/*shared memory*/
#define BUFFER_SIZE 1
#define NONE -1
#define EMPTY 0
#define QUIT "QUIT"
#define ONE 1
#define ZERO 0
#define PP 3425
#define SRTF 1059

typedef struct Buffer{
   char fileName[10]; /*maximum 10 characters including ".txt"*/
   int type; /*either PP or SRTF scheduling*/
   double avgWT, avgTAT; /*for buffer2*/
}Buffer[BUFFER_SIZE];


void *ppFunc(void *none);
void *srtfFunc(void *none);
```

```
#endif
```

[pp.c]----------

```
#include "pp.h"

extern int countB2; /*for buffer2*/
extern Buffer buffer2;
extern pthread_mutex_t keyTwo; /*mutex lock for buffer 2*/
extern pthread_mutex_t uiKey; /*mutex lock for user interface*/
extern pthread_cond_t isFullBTwo; /*conditional variable for buffer2*/
extern pthread_cond_t isEmptyBTwo; /*conditional variable for buffer2*/



void ppReadFile( char fileName[] )
{
   PPTask task; /*to declare the PPTask struct for usage*/
   int time, burst_t, priority, task_count, pid;
   FILE *f;

   f = fopen(fileName, "r");

   if( f == NULL )
   {
      perror("Error opening file");
   }
   else
   {
      task_count = 0;
      pid = 'A';
      while( fscanf(f, "%d %d %d", &time, &burst_t, &priority) !=EOF )
      {
         task[task_count].pid = pid;
         task[task_count].time = time;
         task[task_count].burst_t = burst_t;
         task[task_count].priority = priority;
         task[task_count].rt = burst_t;
         task[task_count].ct = NONE;
         task[task_count].tat = NONE;
         task[task_count].wt = NONE;

         task_count++;
         pid++;
      }
```

```
      if( ferror(f) )
      {
         perror("Error reading file");
      }
      ppSortArrivalTime( task_count, task );
      ppStart( task_count, task ); /*start the pp scheduling*/
      ppGetDetails( task_count, task );
   }
   fclose(f);
}




/*sorting arrival time from 0 to n (increasing order)*/
void ppSortArrivalTime( int task_count, PPTask task )
{
   PPTask tempTask;
   int i, j;

   i = 0;
   while( i < task_count - 1 )
   {
      j = i + 1;
      while( j < task_count )
      {
         if( task[i].time > task[j].time )
         {/*swap places if time j is smaller than time i*/
            tempTask[0] = task[i];
            task[i] = task[j];
            task[j] = tempTask[0];
         }
         j++;
      }
      i++;
   }
}
```

```
void ppGetDetails( int task_count, PPTask task )
{
   int i;
   double avgWT, avgTAT, totalWT, totalTAT;

   totalWT = 0;
   totalTAT = 0;
   for( i = 0; i < task_count; i++ )/*display tasks data*/
   {
      /*calculate total waiting time and turnaround time*/
      totalWT += task[i].wt;
      totalTAT += task[i].tat;
   }
   /*calculte average waiting time and turnaround time*/
   avgWT = totalWT / task_count;
   avgTAT = totalTAT / task_count;

   /**********producer for buffer2*******************/
   pthread_mutex_lock( &keyTwo );
   /*means full, need to wait for isEmptyBTwo from consumer*/
   while( countB2 == ONE )
   {
      pthread_cond_wait( &isEmptyBTwo, &keyTwo );
   }
   /*critical section*/
   buffer2[BUFFER_SIZE-1].type = PP;
   buffer2[BUFFER_SIZE-1].avgWT = avgWT;
   buffer2[BUFFER_SIZE-1].avgTAT = avgTAT;
   countB2 = 1;
   pthread_cond_signal( &isFullBTwo ); /*signal consumer isFullBTwo*/
   pthread_mutex_unlock( &keyTwo );
   /*********************************************/
}
```

```
void ppStart( int task_count, PPTask task )
{
   PPTask readyQueue, running;
   int curTime, i, ready_count, burst_count;
   curTime = 0;
   running[0].empty = true;
```

```
   /*find total burst time*/
   burst_count = ppFindTotalBurst( task_count, task );

   /*displaying Gantt Chart-----------------------------------------*/

   pthread_mutex_lock( &uiKey );
   printf("PP:\n|");
   ready_count = 0;
   do
   {
      /*1. find the new task on current time*/
      ready_count = ppFindNewTask( task_count, task, readyQueue, curTime,
                        ready_count );
      /*2. find the highest task*/
      ready_count = ppFindHighest( readyQueue, running, ready_count,
                        &burst_count );
      /*3. decrement the remaining time and/or removing the running to
          mark it as empty when running is done*/
      ready_count = ppDecrementRT( task_count, task, readyQueue, running,
                        curTime, ready_count );
      printf("|");
      curTime++; /*go to next time*/
   }while( curTime < burst_count ); /*loop through each time*/

   printf("\n");
   for( i = 0; i <= burst_count; i++ ) /*for Gantt Chart time interval*/
      printf("%d ", i );
   printf("\n");
   pthread_mutex_unlock( &uiKey );
   /*---------------------------------------------------------------*/
}




int ppFindTotalBurst( int task_count, PPTask task )
{
   /*find total burst time*/
   int i;
   int burst_count = 0;
   for( i = 0; i < task_count; i++ )
```

```
   {
      burst_count += task[i].burst_t;
   }
   return burst_count;
}
```

```
int ppFindNewTask( int task_count, PPTask task, PPTask readyQueue,
                int curTime, int ready_count )
{
   int i;
   for( i = 0; i < task_count; i++ )
   {
      if( task[i].time == curTime ) /*find new task for current time*/
      {
         readyQueue[ready_count] = task[i]; /*insert to ready queue*/
         readyQueue[ready_count].empty = false;
         ready_count++;
      }
   }
   return ready_count;
}
```

```
int ppFindHighest( PPTask readyQueue, PPTask running, int ready_count,
              int *burst_count )
{
    int i, highPrioNum, highPrioIdx;

    /*now check which process in the queue is highest*/
    if( ready_count >= ONE )/*have multiple task to choose from*/
    {
       highPrioNum = readyQueue[0].priority; /*initial priority NUMBER*/
       highPrioIdx = 0; /*initial priority INDEX*/
       for( i = 1; i < ready_count; i++ )
       {
          /*if found another new higher priority*/
```

```
            if( highPrioNum > readyQueue[i].priority )
            {
               /*new higher priority found with different pid*/
               highPrioNum = readyQueue[i].priority;
               highPrioIdx = i;
            }
         }
         /*if found same priority, no need preemption, because
          first come first serve basis, so the current running
          continue running without changing to another process
          of same priority*/
         if( !running[0].empty )
         {
            /*only assign process when have higher priority*/
            if( running[0].priority > readyQueue[highPrioIdx].priority )
            {
               ready_count = ppDetermineRun( readyQueue, running,
                                 ready_count, highPrioIdx );
            }
            else
            {
            }
         }
         else if( running[0].empty )
         {/*only assign process if running is empty*/
            ready_count = ppDetermineRun( readyQueue, running,
                                 ready_count, highPrioIdx );
         }
      }
      else if( ready_count == EMPTY && running[0].empty )
      {
          *burst_count = *burst_count + 1;
      }
      /*if ready_count is empty means no ready queue allowed to enter
        running process, because empty*/
     return ready_count;
}
```

```
int ppDetermineRun( PPTask readyQueue, PPTask running, int ready_count,
```

```
            int highPrioIdx )
{
   int i;
   /*determine how processes should run*/
    if( !running[0].empty )
    {/*have running process, need to save progress first*/
      /*means need to preemp the current running process to make way
       for the higher priority task*/

       /*1. save progress by adding to ready queue*/
       readyQueue[ready_count] = running[0];
       ready_count++;

       /*2. change to another process*/
       running[0] = readyQueue[highPrioIdx];
       running[0].highPrioIdx = highPrioIdx;
       running[0].empty = false;

       /*3. delete the "assigned ready queue to running" from the
            queue and shift the array forward like a queue*/
       for( i = running[0].highPrioIdx; i < ready_count - 1; i++ )
       {
          readyQueue[i] = readyQueue[i+1];
       }
       ready_count--;
    }
    else if( running[0].empty )
    {/*no process, just assign process*/

       /*1. assign a process*/
       running[0] = readyQueue[highPrioIdx];
       running[0].highPrioIdx = highPrioIdx;
       running[0].empty = false; /*means no more empty*/

       /*2. delete the "assigned ready queue to running" from the queue
         and shift the array forward like a queue*/
       for( i = running[0].highPrioIdx; i < ready_count - 1; i++ )
       {
          readyQueue[i] = readyQueue[i+1];
       }
       ready_count--;
   }
   return ready_count;
}
```

```
int ppDecrementRT( int task_count, PPTask task, PPTask readyQueue,
            PPTask running, int curTime, int ready_count )
{
   int count;
   if( !running[0].empty )
   {
      /*if still have remaining time*/
      if( running[0].rt > EMPTY )
      {
         running[0].rt--; /*decrement remaining time of running*/
         printf("%c", running[0].pid);

        /*if reaches zero after decrement, save data, & straight away
         set running to empty for other process to use*/
         if( running[0].rt == EMPTY )/*if no more remaining time*/
         {
            /*1. calculate completion, turnaround, waiting time*/
            running[0].ct = curTime + 1; /*because get next time*/
            running[0].tat = running[0].ct - running[0].time;
            running[0].wt = running[0].tat - running[0].burst_t;

            /*2. find original task (not running task or ready queue)
                to update their rt, ct, tat, wt*/
            for( count = 0; count < task_count; count++ )
            {
               /*found the running procoess's original/initial task*/
               if( running[0].pid == task[count].pid )
               {
                  /*3. transfer the info to original task from
                      running process*/
                  task[count].rt = running[0].rt;
                  task[count].ct = running[0].ct;
                  task[count].tat = running[0].tat;
                  task[count].wt = running[0].wt;
                  running[0].empty = true;/*4. set running to empty*/
               }
            }
         }
      }
   }
```

```
   return ready_count;
}
```
[pp.h]----------

```
#ifndef PP_H
#define PP_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include "multiThread.h"

typedef struct PPTask{
   int time, burst_t, priority;
   char pid;
   int rt, ct, tat, wt;
   bool empty; /*only for running process*/
   int highPrioIdx; /*only for decrementRT to use for deleting ready
                queue array index*/
}PPTask[50]; /*maximum number of task is 50*/

void ppReadFile( char fileName[] );
void ppSortArrivalTime( int task_count, PPTask task );
void ppGetDetails( int task_count, PPTask task );
void ppStart( int task_count, PPTask task );
int ppFindTotalBurst( int task_count, PPTask task );
int ppFindNewTask( int task_count, PPTask task, PPTask readyQueue, int curTime,
                           int ready_count );
int ppFindHighest( PPTask readyQueue, PPTask running, int ready_count,
                          int *burst_count );
int ppDetermineRun( PPTask readyQueue, PPTask running, int ready_count,
                            int highPrioIdx );
int ppDecrementRT( int task_count, PPTask task, PPTask readyQueue,
                           PPTask running, int curTime, int ready_count );
#endif
```

[srtf.c]----------

```
#include "srtf.h"

extern int countB2; /*for buffer2*/
extern Buffer buffer2;
extern pthread_mutex_t keyTwo; /*mutex lock for buffer 2*/
extern pthread_mutex_t uiKey; /*mutex lock for user interface*/
extern pthread_cond_t isFullBTwo; /*conditional variable for buffer2*/
extern pthread_cond_t isEmptyBTwo; /*conditional variable for buffer2*/


void srtfReadFile( char fileName[] )
{
   SRTFTask task; /*to declare the SRTFTask struct for usage*/
   int time, burst_t, priority, task_count, pid;
   FILE *f;

   f = fopen(fileName, "r");
   if( f == NULL )
   {
      perror("Error opening file");
   }
   else
   {
      task_count = 0;
      pid = 'A';
      while( fscanf(f, "%d %d %d", &time, &burst_t, &priority) !=EOF )
      {
         task[task_count].pid = pid;
         task[task_count].time = time;
         task[task_count].burst_t = burst_t;
         task[task_count].priority = priority;
         task[task_count].rt = burst_t;
         task[task_count].ct = NONE;
         task[task_count].tat = NONE;
         task[task_count].wt = NONE;

         task_count++;
         pid++;
      }
      if( ferror(f) )
      {
```

```
            perror("Error reading file");
        }
        srtfSortArrivalTime( task_count, task );
        srtfStart( task_count, task ); /*start the srtf scheduling*/
        srtfGetDetails( task_count, task );

    }
    fclose(f);
}




/*sorting arrival time from 0 to n (increasing order)*/
void srtfSortArrivalTime( int task_count, SRTFTask task )
{
    SRTFTask tempTask;
    int i, j;

    i = 0;
    while( i < task_count - 1 )
    {
        j = i + 1;
        while( j < task_count )
        {
            if( task[i].time > task[j].time )
            {/*swap places if time j is smaller than time i*/
                tempTask[0] = task[i];
                task[i] = task[j];
                task[j] = tempTask[0];
            }
            j++;
        }
        i++;
    }
}




void srtfGetDetails( int task_count, SRTFTask task )
```

```
{
   int i;
   double avgWT, avgTAT, totalWT, totalTAT;

   totalWT = 0;
   totalTAT = 0;
   for( i = 0; i < task_count; i++ )/*display tasks data*/
   {
      /*calculate total waiting time and turnaround time*/
      totalWT += task[i].wt;
      totalTAT += task[i].tat;
   }
   /*calculte average waiting time and turnaround time*/
   avgWT = totalWT / task_count;
   avgTAT = totalTAT / task_count;

   /**********producer for buffer2*******************/
   pthread_mutex_lock( &keyTwo );
   /*means full, need to wait for isEmptyBTwo from consumer*/
   while( countB2 == ONE )
   {
      pthread_cond_wait( &isEmptyBTwo, &keyTwo );
   }
   /*critical section*/
   buffer2[BUFFER_SIZE-1].type = SRTF;
   buffer2[BUFFER_SIZE-1].avgWT = avgWT;
   buffer2[BUFFER_SIZE-1].avgTAT = avgTAT;
   countB2 = 1;
   pthread_cond_signal( &isFullBTwo ); /*signal consumer isFullBTwo*/
   pthread_mutex_unlock( &keyTwo );
   /************************************************/
}




void srtfStart( int task_count, SRTFTask task )
{
   SRTFTask readyQueue, running;
   int curTime, i, ready_count, burst_count;
   curTime = 0;
   running[0].empty = true;
```

```
   /*find total burst time*/
   burst_count = srtfFindTotalBurst( task_count, task );

   /*displaying Gantt Chart-----------------------------------------*/

   pthread_mutex_lock( &uiKey );
   printf("SRTF:\n|");
   ready_count = 0;
   do
   {
      /*1. find the new task on current time*/
      ready_count = srtfFindNewTask( task_count, task, readyQueue,
                              curTime, ready_count );
      /*2. find the highest task*/
      ready_count = srtfFindHighest( readyQueue, running, ready_count,
                           &burst_count );
      /*3. decrement the remaining time and/or removing the running to
          mark it as empty when running is done*/
      ready_count = srtfDecrementRT( task_count, task, readyQueue,
                              running, curTime, ready_count );
      printf("|");
      curTime++; /*go to next time*/
   }while( curTime < burst_count ); /*loop through each time*/

   printf("\n");
   for( i = 0; i <= burst_count; i++ ) /*for Gantt Chart time interval*/
      printf("%d ", i );
   printf("\n");
   pthread_mutex_unlock( &uiKey );
   /*---------------------------------------------------------------*/
}




int srtfFindTotalBurst( int task_count, SRTFTask task )
{
   /*find total burst time*/
   int i;
   int burst_count = 0;
   for( i = 0; i < task_count; i++ )
   {
```

```
      burst_count += task[i].burst_t;
   }
   return burst_count;
}
```

```
int srtfFindNewTask( int task_count, SRTFTask task, SRTFTask readyQueue,
                int curTime, int ready_count )
{
   int i;
   for( i = 0; i < task_count; i++ )
   {
      if( task[i].time == curTime ) /*find new task for current time*/
      {
         readyQueue[ready_count] = task[i]; /*insert to ready queue*/
         readyQueue[ready_count].empty = false;
         ready_count++;
      }
   }
   return ready_count;
}
```

```
int srtfFindHighest( SRTFTask readyQueue, SRTFTask running,
                int ready_count, int *burst_count )
{
    int i, highPrioRT, highPrioIdx;

    /*now check which process in the queue is highest*/
    if( ready_count >= ONE )/*have multiple task to choose from*/
    {
       highPrioRT = readyQueue[0].rt; /*initial priority NUMBER*/
       highPrioIdx = 0; /*initial priority INDEX*/
       for( i = 1; i < ready_count; i++ )
       {
          /*if found another new higher priority*/
          if( highPrioRT > readyQueue[i].rt )
```

```
            {
               /*new higher priority found with different pid*/
               highPrioRT = readyQueue[i].rt;
               highPrioIdx = i;
            }
         }
         /*if found same priority, no need preemption, because
          first come first serve basis, so the current running
          continue running without changing to another process
          of same priority*/
         if( !running[0].empty )
         {
            /*only assign process when have higher priority*/
            if( running[0].rt > readyQueue[highPrioIdx].rt )
            {
               ready_count = srtfDetermineRun( readyQueue, running,
                                 ready_count, highPrioIdx );
            }
            else
            {
            }
         }
         else if( running[0].empty )
         {/*only assign process if running is empty*/
            ready_count = srtfDetermineRun( readyQueue, running,
                               ready_count, highPrioIdx );
         }
      }
      else if( ready_count == EMPTY && running[0].empty )
      {
          *burst_count = *burst_count + 1;
      }
      /*if ready_count is empty means no ready queue allowed to enter
        running process, because empty*/
     return ready_count;
}




int srtfDetermineRun( SRTFTask readyQueue, SRTFTask running,
                  int ready_count, int highPrioIdx )
```

```
{
   int i;
   /*determine how processes should run*/
    if( !running[0].empty )
    {/*have running process, need to save progress first*/
      /*means need to preemp the current running process to make way
       for the higher priority task*/

       /*1. save progress by adding to ready queue*/
       readyQueue[ready_count] = running[0];
       ready_count++;

       /*2. change to another process*/
       running[0] = readyQueue[highPrioIdx];
       running[0].highPrioIdx = highPrioIdx;
       running[0].empty = false;

       /*3. delete the "assigned ready queue to running" from the
            queue and shift the array forward like a queue*/
       for( i = running[0].highPrioIdx; i < ready_count - 1; i++ )
       {
          readyQueue[i] = readyQueue[i+1];
       }
       ready_count--;
    }
    else if( running[0].empty )
    {/*no process, just assign process*/

       /*1. assign a process*/
       running[0] = readyQueue[highPrioIdx];
       running[0].highPrioIdx = highPrioIdx;
       running[0].empty = false; /*means no more empty*/

       /*2. delete the "assigned ready queue to running" from the queue
         and shift the array forward like a queue*/
       for( i = running[0].highPrioIdx; i < ready_count - 1; i++ )
       {
          readyQueue[i] = readyQueue[i+1];
       }
       ready_count--;
   }
   return ready_count;
}
```

```
int srtfDecrementRT( int task_count, SRTFTask task, SRTFTask readyQueue,
                SRTFTask running, int curTime, int ready_count )
{
   int count;
   if( !running[0].empty )
   {
      /*if still have remaining time*/
      if( running[0].rt > EMPTY )
      {
         running[0].rt--; /*decrement remaining time of running*/
         printf("%c", running[0].pid);/*print for gantt chart*/

         /*1. find original task (not running task or ready queue)
             to update their rt, ct, tat, wt*/
         for( count = 0; count < task_count; count++ )
         {
            /*found the running procoess's original/initial task*/
            if( running[0].pid == task[count].pid )
            {
              /*2. constantly update remaining time into task after
                  every decrement*/
              task[count].rt = running[0].rt;

              /*if reaches zero after decrement, save data, &
                straight away set running to empty for other process
                to use*/
              if( running[0].rt == EMPTY )/*if no more RT*/
              {
              /*A. calculate completion, turnaround, waiting time*/
                 running[0].ct = curTime + 1;
                 /*because get next time ^^^*/
                 running[0].tat = running[0].ct - running[0].time;
                 running[0].wt = running[0].tat - running[0].burst_t;

                 /*B. transfer the info to original task from
                     running process*/
                 task[count].ct = running[0].ct;
                 task[count].tat = running[0].tat;
                 task[count].wt = running[0].wt;
                 running[0].empty = true;/*C. set running to empty*/
              }
```

```
            }
          }
        }
    }
    return ready_count;
}
```

[srtf.h]----------

```
#ifndef SRTF_H
#define SRTF_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include "multiThread.h"

typedef struct SRTFTask{
    int time, burst_t, priority;
    char pid;
    int rt, ct, tat, wt;
    bool empty; /*only for running process*/
    int highPrioIdx; /*only for decrementRT to use for deleting ready
                queue array index*/
}SRTFTask[50]; /*maximum number of task is 50*/

void srtfReadFile( char fileName[] );
void srtfSortArrivalTime( int task_count, SRTFTask task );
void srtfGetDetails( int task_count, SRTFTask task );
void srtfStart( int task_count, SRTFTask task );
int srtfFindTotalBurst( int task_count, SRTFTask task );
int srtfFindNewTask( int task_count, SRTFTask task, SRTFTask readyQueue, int curTime,
                            int ready_count );
int srtfFindHighest( SRTFTask readyQueue, SRTFTask running, int ready_count,
                           int *burst_count );
int srtfDetermineRun( SRTFTask readyQueue, SRTFTask running, int ready_count,
                              int highPrioIdx );
int srtfDecrementRT( int task_count, SRTFTask task, SRTFTask readyQueue,
                            SRTFTask running, int curTime, int ready_count );
#endif
```

(ii) README.txt file location. (compiling instructions)

The README.txt is located in the assignment folder before the “PP”, “SRTF”, “MultiThreading” folders.

(iii) Discussion on how any mutual exclusion is achieved and what processes/threads access the shared resources.

Mutual exclusion ensures threads or section of code do not enter critical section to modify the shared memory at the same time. If a section of the code in a thread is using the critical section to modify data, the other sections in current thread or other threads cannot enter critical section to modify the data until it is completed and returned the key. Consumer can only consume data from the critical section after the Producer entered critical section.

For buffer1, parent thread in the main function is the producer whereas “ppFunc” and “srtfFunc” methods are the consumer for buffer1. In this case, the producer produces the file name before consumers consume. The two consumers reads the file name from buffer1 at the same time because is doesn’t modify the file name and just read the file name, after that consumers will signal isEmpty to producer to produce another file name

For buffer2, parent thread in the main function is the consumer of buffer2 whereas “ppGetDetails” and “srtfGetDetails” methods are the producers of buffer2. Here, producer must insert a pair of average waiting time and average turnaround time into buffer2 before consumer consumes. Mutual exclusion helps facilitate the administration of these sections of code that wants to enter critical section to ensure only one section of the current code can enter critical section at a time to produce or consume data.

(iv) Description of any cases for which your program is not working correctly or how you test your program that makes you believe it works perfectly.

The gantt chart visualization isn’t properly aligned with the numbers below. But, it is in correct order with the correct number of burst time.

Other than that, my code works just fine. The program’s results are accurate to my manual calculated results for PP and SRTF scheduling.

(v) Sample inputs and outputs from your running programs. (on terminal)

rt = remaining time
ct = completion time
tat = turnaround time
wt = waiting time

"PP" folder input and output on terminal

[highlighted in blue is your user input, the rest are output from the program]

remember to exclude ".txt" for PP program.

Result:

exclude ".txt" from file name
PP simulation: input2
Total burst time: 20
|A|A|A|A|B|B|C|C|C|C|C|C|C|B|E|E|D|D|D|D|
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Number of Task: 5

pid	time	burst	priority	rt	ct	tat	wt
A	0	4	1	0	4	4	0
B	0	3	2	0	14	14	11
C	6	7	1	0	13	7	0
D	11	4	3	0	20	9	5
E	12	2	2	0	16	4	2

AVERAGE WAITING TIME: 3.600000

AVERAGE TURNAROUND TIME: 7.600000
exclude ".txt" from file name
PP simulation: QUIT

“SRTF” folder input and output on terminal

[highlighted in blue is your user input, the rest are output from the program]

remember to exclude “.txt” for SRTF program.

Result:

Exclude ".txt" from file name
SRTF simulation: input6
Total burst time: 23
|D|D|D|A|E|B|B|E|E|E|A|A|A|A|C|C|C|C|C|C|C|C|
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Number of Task: 5

pid	time	burst	priority	rt	ct	tat	wt
D	0	3	7	0	3	3	0
C	1	8	3	0	23	22	14
A	2	6	1	0	15	13	7
E	4	4	2	0	10	6	2
B	5	2	4	0	7	2	0

AVERAGE WAITING TIME: 4.600000

AVERAGE TURNAROUND TIME: 9.200000
Exclude ".txt" from file name
SRTF simulation: QUIT

“MultiThreading” folder input and output on terminal

[highlighted in blue is your user input, the rest are output from the program]

remember to include “.txt” for multiThread program.

Result:

Include ".txt" after file name
file: input2.txt
PP:
|A|A|A|A|B|B|C|C|C|C|C|C|C|B|E|E|D|D|D|D|
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
SRTF:
|B|B|B|A|A|A|A|C|C|C|C|C|C|C|E|E|D|D|D|D|
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
PP: the average turnaround time = 7.600000, the average waiting time = 3.600000
SRTF: the average turnaround time = 6.200000, the average waiting time = 2.200000

Include ".txt" after file name
file: input6.txt
PP:
|D|C|A|A|A|A|A|A|E|E|E|E|C|C|C|C|C|C|C|B|B|D|D|
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
SRTF:
|D|D|D|A|E|B|B|E|E|E|A|A|A|A|A|C|C|C|C|C|C|C|C|
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
PP: the average turnaround time = 14.200000, the average waiting time = 9.600000
SRTF: the average turnaround time = 9.200000, the average waiting time = 4.600000

Include ".txt" after file name
file: QUIT
PP: terminate.
SRTF: terminate.