Table of Contents

Software solution....1
Scheduler C implementation....1
file I/O C implementation....7
Queue C implementation....10
Header files....13
Make file....15
Extras....16
Task_file generator: gentask.sh....16
Queue test harness....16
How to run....17
Achieving Mutual Exclusion....18
Are there any issues with the program?....20
There are no known issues with the program such as leaks or other errors....20
Sample input and output....21

Software solution

Scheduler C implementation

```c
/**
 *Task Scheduler simulator OS assignment
 *AUTHOR: Kei Sum Wang, student id: 19126089
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>
#include "scheduler.h"

/***GLOBAL VARIABLES***/
float num_tasks = 0.0, total_waiting_time = 0.0, total_turnaround_time = 0.0;

int bufferLimit = 0, fileSize = 0;

pthread_cond_t full, empty;//conditions for wait and signaling
pthread_mutex_t mutex;//to help setup mutual exclusion when entering crit section

//shared buffers
Queue* readyQueue;//buffer to be accessed by threads
Queue* fromFileQueue;//task_file content are stored in queue

pthread_t task_thread, cpu1, cpu2, cpu3;

char *name1 = "CPU-1", *name2 = "CPU-2", *name3 = "CPU-3"; //names of cpus
char *simLog = "simulation_log";//output file

/*
 *MAIN
 *IMPORT: take in a count for # args, filename and size for ready queue
 */
int main(int argc, char *argv[])
{
    bufferLimit = atoi(argv[2]);

    if(argc < 3 || (bufferLimit < 1 || bufferLimit > 10))
    {
        printf("Incorrect arguments\nCheck readme for argument details\n");
        return -1;
    }

    //initilize containers
    fromFileQueue = createQueue();
    readyQueue = createQueue();

    if(readFile(argv[1], fromFileQueue) != 0)
    {
            freeQueue(fromFileQueue, 0);
            freeQueue(readyQueue, 0);
            return -1;//unsuccessful read should end
    }

    //total number of tasks from task_file global var
    fileSize = fromFileQueue->length;

    //wipe previous log to prevent appending
    wipeLog(simLog);

    //initialise mutex and condition vars
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&full, NULL);
    pthread_cond_init(&empty, NULL);
```

```
    //task thread creation
    pthread_create(&task_thread, NULL, &task, NULL);

    //cpu threads creation
    pthread_create( &cpu1, NULL, &cpu, (void*) name1);
    pthread_create(&cpu2, NULL, &cpu, (void*) name2);
    pthread_create(&cpu3, NULL, &cpu, (void*) name3);

    //start threads
    pthread_join(task_thread, NULL);
    pthread_join(cpu1, NULL);
    pthread_join(cpu2, NULL);
    pthread_join(cpu3, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&full);
    pthread_cond_destroy(&empty);

    //free containers from memory
    freeQueue(fromFileQueue, 0);
    freeQueue(readyQueue, 0);

    //output end stats the num_task, waiting and turn around stats to screen and file
    taskTimeStats();

    return 0;
}

/**
 *task method
 *PURPPOSE: used by the task thread to add tasks to ready queue
 */
void* task(void *arg)
{
    int now_hr = 0, now_min = 0, now_sec = 0, i, spaceLeft, concat;
    char summary[200];

    process *task1, *task2 ;

    while(!queueEmpty(fromFileQueue))
    {

        pthread_mutex_lock(&mutex);//lock thread to prevent to incorrect processing
        //task thread is the only thread able access ready queue until unlock

        task1 = dequeue(fromFileQueue);

        //wait for cpu to consume if buffer limit reached
        while(readyQueue->length == bufferLimit)
        {
            printf("task waiting for cpus to consume %d length \n\n", readyQueue->length);
            pthread_cond_wait(&empty, &mutex);
        }

        printf("*******Task thread inputting*******\n\n");

        //critical section for queueing, task can only access ready when lock cpus cannot
        getTime(&(task1->arrive_hr), &(task1->arrive_min), &(task1->arrive_sec));//get current time
        enqueue(readyQueue, task1);

        //output task statistics to screen and file
        taskThreadStats(task1);

        //calculate the space left in ready queue
        spaceLeft = bufferLimit - readyQueue->length;
        /*buffer size must be greater 1 in order to get 2 tasks
         *file queue cannot be empty after task1 dequeue
         *space left must be greater than 0 otherwise ready queue 1 task above buffer limit
         */
        if(bufferLimit > 1 && !queueEmpty(fromFileQueue) && spaceLeft > 0)
        {
            printf("*******adding second task*******\n");
            task2 = dequeue(fromFileQueue);
```

```
            getTime(&(task2->arrive_hr), &(task2->arrive_min), &(task2->arrive_sec));//get current time
            enqueue(readyQueue, task2);

            //output task threads stat for queueing of 2nd task
            taskThreadStats(task2);
        }

        printf("current ready queue size after task thread put tasks: %d\n\n", readyQueue->length);

        pthread_cond_signal(&full);//signal that processes are available in readyqueue
        pthread_mutex_unlock(&mutex);//unlock mutex after done enqueueing

    }

    /*Since there are 3 cpus, 1 cpu may block forever if there less than 3 tasks from file
     *task thread cannot signal in critical section once done, so loop to release any cpu blocked
     *This is to prevent deadlock
     */
    if(fileSize < 3)
    {
            for(i = 0; i < 3; i++)
            {
                    pthread_cond_signal(&full);
            }
    }

    //concat string ready to write task thread summary to file and print to screen
    concat = sprintf(summary, "Number of tasks put into Ready-Queue: %d\n", fileSize);
    getTime(&now_hr, &now_min, &now_sec);
    concat += sprintf(summary + concat, "Terminate at time: %d:%d:%d\n\n",  now_hr, now_min, now_sec);
    printf("%s", summary);
    writeToFile(simLog, summary);
    return NULL;

}

/**
 *cpu
 *PURPOSE: To perform CPU operations
 */
void* cpu(void *arg)
{
    int task_counter;
    char summary[50];
    char *cpu_name = (char*)arg;//get cpu thread name
    process* task;

    task_counter = 0;

    //**check criticalnotEmpty() to see the mutex lock**
    while(criticalNotEmpty(cpu_name))
    {
        /*if criticalNotEmpty() is true allow 1 cpu that is waiting to
         *enter crit section of accessing ready queue
         */
        printf("*******%s removing task from ready queue*******\n", cpu_name);
        printf("current %d file queue length\n", fromFileQueue->length);
        printf("current %d ready queue length\n", readyQueue->length);

        //critical section
        //at least 1 item should be available to dequeue other this loop should've not execute
        task = dequeue(readyQueue);

        printf("*******ready queue size is %d after %s dequeues*******\n", readyQueue->length, cpu_name);
        //calculate the wait and turn around time
        getTime(&(task->serv_hr), &(task->serv_min), &(task->serv_sec));
        calcWaitTime(task);
        calcTurnAround(task);

        //output cpu serving task stats
        cpuThreadStats(task, cpu_name, SERVING);

        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);//release lock now, cpu will now consume
```

```
                        //no longer need lock since task out of ready queue

        //remainder section aka consuming
        printf("*******%s is consuming*******\n\n", cpu_name);
        sleep(task->cpu_burst);
        getTime(&(task->comp_hr), &(task->comp_min), &(task->comp_sec));
        task_counter++;//count task for current thread
        num_tasks++;//count task for global

        //output cpu completion task stats
        cpuThreadStats(task, cpu_name, DONE);

        free(task);
    }

    printf("*******CPU Termination********\n");
    sprintf(summary,"%s terminates after servicing %d tasks\n\n", cpu_name, task_counter);
    printf("%s", summary);
    writeToFile(simLog, summary);

    pthread_mutex_unlock(&mutex);//unlock lock set by criticalNotEmpty(), no tasks at this point.

    return NULL;
}

/*
 *criticalNotEmpty function checks if cpu can enter the critical section
 *only if ready queue is not empty and file queue still has task
 *IMPORT: cpu's name
 *EXPORT: integer flag indicating ready and file queue is empty(0) or not(1)
 */
int criticalNotEmpty(char *nameCpu)
{
    int notEmpty = 0;

    pthread_mutex_lock(&mutex);    //one cpu hold lock at a time, enforce mutual exclusion
    /*this is critical section for cpu thread, must ensure that each cpu (one at a time)
     *checks the empty status of ready queue and file queue before consuming
     *this prevent cpu from entering loop and dequeing when previously another
     *cpu task has just dequeued (RACE CONDITION)
     */

    //ready queue is empty but file queue still has tasks wait for task thread
    while(queueEmpty(readyQueue) && !queueEmpty(fromFileQueue))
    {
        printf("*******%s waiting for ready queue to have tasks********\n", nameCpu);
        pthread_cond_wait(&full, &mutex);
    }

    //if this is true(1) then task can continue to critcal section to consume
    //we don't need to check if file queue is empty due to wait above
    //task thread would queue more tasks if ready queue is empty but there are still more tasks
    notEmpty = (!queueEmpty(readyQueue));

    return notEmpty;
}

/*
 *function to get time on system. Uses C time standard library
 *IMPORTS: takes in pointers to integer for hours, minutess and seconds
 */
void getTime(int *hr, int *min, int *sec)
{
    time_t now;

    time(&now);

    struct tm *curTime = localtime(&now);

    *hr = curTime->tm_hour;
    *min = curTime->tm_min;
    *sec = curTime->tm_sec;
}
```

```c
/*
 *function to output arrival from task thread
 *IMPORT: a task
 */
void taskThreadStats(process *task)
{
    int j = 0;
    char str[200];

    //string concatanation
    j = sprintf(str, "#: %d | Burst: %d\n", task->task_id, task->cpu_burst);
    j += sprintf(str + j, "Arrival time: %d:%d:%d\n\n", task->arrive_hr, task->arrive_min, task->arrive_sec);

    //print to terminal
    printf("%s", str);

    //write to file using func from fileIO.c
    writeToFile(simLog, str);
}

/*
 *function to output service and completion stats of tasks of cpu thread
 *IMPORT: a task, name of cpu and status(this is a preprocessor directive SERVING or DONE)
 *       find the preprocessor directive in scheduler.h
 */
void cpuThreadStats(process *task, char *cpu_name, int status)
{
    int j = 0;
    char str[200];

    //string concatanation
    j = sprintf(str, "Statistics for %s:\n", cpu_name);
    j += sprintf(str + j, "#: %d | Burst: %d\n", task->task_id, task->cpu_burst);
    j += sprintf(str + j, "Arrival time: %d:%d:%d\n", task->arrive_hr, task->arrive_min, task->arrive_sec);

    if(status == SERVING)
    {
            sprintf(str + j, "Service time: %d:%d:%d\n\n", task->serv_hr, task->serv_min, task->serv_sec);
    }
    else if(status == DONE)
    {
            sprintf(str + j, "Completion time: %d:%d:%d\n\n", task->comp_hr, task->comp_min, task->comp_sec);
    }

    //print to screen
    printf("%s", str);

    //write to log using func from fileIO.c
    writeToFile(simLog, str);
}

/*
 *function to calculate total wait time in seconds
 *IMPORTS: task, hour, minutes and seconds
 */
void calcWaitTime(process *inTask)
{
    float arrival = 0.0, serv_time = 0.0;

    //convert all parts of time to seconds for wait time calculations
    serv_time = (float)((inTask->serv_hr * 3600) + (inTask->serv_min * 60) + inTask->serv_sec);
    arrival = (float)((inTask->arrive_hr * 3600) + (inTask->arrive_min * 60) + inTask->arrive_sec);

    total_waiting_time += serv_time - arrival; //global var total waiting time in seconds
}
```

```
/*
 *function to calculate total turn around time in seconds
 *IMPORTS: task, hour, minutes and seconds
 */
void calcTurnAround(process *inTask)
{
    float arrival = 0.0, comp_time = 0.0, result = 0.0;

    //All time measurements in seconds for turn around time calculations
    comp_time = (float)((inTask->serv_hr * 3600) + (inTask->serv_min * 60) + inTask->serv_sec) + (float)(inTask-
>cpu_burst);
    arrival = (float)((inTask->arrive_hr * 3600) + (inTask->arrive_min * 60) + inTask->arrive_sec);

    result = comp_time - arrival;
    total_turnaround_time += result; //global var total waiting time in seconds
}

/*
 *function to print out overall task time statistics after scheduler done
 */
void taskTimeStats()
{
    float avg_wait = 0.0, avg_TAT = 0.0;

    avg_wait = total_waiting_time / num_tasks;
    avg_TAT = total_turnaround_time / num_tasks;

    printf("Number of tasks: %d\n", (int)num_tasks);
    printf("Average waiting time: %.2f seconds\n", avg_wait);
    printf("Average turn around time: %.2f seconds\n", avg_TAT);

    //fileIO.c write stats to file
    writeTaskTimeStats(simLog, num_tasks, avg_wait, avg_TAT);
}
```

file I/O C implementation

```c
/*
 * file dealing with file io
 * AUTHOR: Kei Sum Wang, student id: 19126089
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "fileIO.h"

#define MAX_CHARS 100


/*
 * processFile
 * IMPORTS: string fileName, and a queue.
 * PURPOSE: reads in task file and stores tasks in a struct, adds tasks into a queue
 */
int readFile(char *fileName, Queue *q)
{
   FILE *file = fopen(fileName, "r");
   char str[MAX_CHARS];
   int check1, check2;
   process* task = NULL;

   if (file == NULL)
   {
      printf("\n");
      perror("Error reading");
      return -1;
   }
   else
   {
      while (fgets(str, MAX_CHARS, file) != NULL)//read line by line
      {
         sscanf(str, "%d %d", &check1, &check2);
         if(check1 > 0 && check2 > 0)//id and burst are (+)ve
         {
            task = (process*)malloc(sizeof(process));//alloc memory for task
            sscanf(str, "%d %d", &(task->task_id), &(task->cpu_burst));/*get task id and cpu burst*/

            enqueue(q, task);/*insert last into queue for ascending order*/
         }
         else
         {
           printf("Error reading: id and burst time must be positive");
           fclose(file);
           return -1;
         }

      }

      if (ferror(file))/*check if any error when read*/
      {
           printf("Error reading file\n");
      }

      fclose(file);
   }

   return 0;
}
```

```c
/*
 * writeToFile
 * IMPORTS: string fileName
 * PURPOSE: appends str to file
 */
int writeToFile(char *fileName, char *str)
{
    FILE *file = fopen(fileName, "a");

    if (file == NULL)
    {
        printf("\n");
        perror("Error writing to file");
        return -1;
    }
    else
    {

        fprintf(file, "%s", str);

        if (ferror(file))
        {
            printf("Error writing to file");/*if error found, print error*/
        }

        fclose(file);
    }
    return 0;
}

/*
 * writeTaskStats
 * IMPORTS: string fileName, num of task, avg wait time and avg turn around
 * PURPOSE: appends time statistics of process simulation
 */
int writeTaskTimeStats(char *fileName, float num_tasks, float avg_wait, float avg_TAT)
{
    FILE *file = fopen(fileName, "a");

    if (file == NULL)
    {
        printf("\n");
        perror("Error writing to file");
        return -1;
    }
    else
    {

        fprintf(file,"Number of tasks: %d\n", (int)num_tasks);
        fprintf(file, "Average waiting time: %.2f seconds\n", avg_wait);
        fprintf(file, "Average turn around time: %.2f seconds\n", avg_TAT);

        if (ferror(file))
        {
            printf("Error writing to file");/*if error found, print error*/
        }
        else
        {
            printf("\nWrite to file: %s success\n", fileName);/*print success if no error found when writing*/
        }

        fclose(file);
    }

    return 0;
}
```

```c
/*
 *function to clear contents of file
 *IMPORT: file name
 */
int wipeLog(char *fileName)
{
    fclose(fopen(fileName, "w"));

    return 0;
}
```

Queue C implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include "Queue.h"

/*
 * Queue implementation
 *AUTHOR: Kei Sum Wang    student id: 19126089
 *REFERENCE: this is based off linked list from the unit Unix and C programming
 */

/*
 *Initialisation of queue
 *EXPORT: an initialised queue ready to use
 */
Queue* createQueue()
{
   Queue* q;

   q = (Queue*)malloc(sizeof(Queue));
   q->head = NULL;
   q->tail = NULL;
   q->length = 0;

   return q;
}

/*
 * enqueue
 *PURPOSE: inserts data into queue. each data inserted at end of queue
 *IMPORT: queue and a data of type void pointer
 */
void enqueue(Queue* q, void* data)
{
   Qnode* newNode;

   newNode = (Qnode*)malloc(sizeof(Qnode));
   newNode->next = NULL;

   newNode->data = data;

   if(q->head == NULL)//if queue empty
   {
      q->head = newNode;
   }
   else
   {
      q->tail->next = newNode;
   }

   q->tail = newNode;

   q->length++;
}
```

```c
/*
```

```
 * dequeue
 *PURPOSE: remove data from queue. each data remove from front of queue
 *IMPORT: queue
 *EXPORTS: void pointer
 */
void* dequeue(Queue* q)
{
   Qnode* currHead;
   void* value;

   currHead = q->head;

   if(q->length == 0)
   {
      value = NULL;
      printf("List is Empty: nothing to dequeue\n");
   }
   else
   {
      value = currHead->data;
      if(currHead->next == NULL)
      {
         q->tail = NULL;
      }

      q->head = currHead->next;

      q->length--;
   }
   free(currHead);

   return value;
}

/*
 *Check if queue is empty
 *IMPORT: queue
 *EXPORT: integer indicating empty or nor
 */
int queueEmpty(Queue* q)
{
   int isEmpty = 0;

   //nothing at head then queue is empty
   if(q->head == NULL)
   {
      isEmpty = 1;
   }

   return isEmpty;
}

/*
 * peek at front of queue
 *IMPORT: queue
 *EXPORT: data of type void pointer
 */
void* peekFirst(Queue *q)
{
   void* value;

   if (q->length == 0)
   {
      value = NULL;
   }
   else
   {
      value = q->head->data;
   }

   return value;
}

/*
```

```
 * peek at end of queue
 *IMPORT: queue
 *EXPORT: data of type void pointer
 */
void* peekLast(Queue *q)
{
   void* value;

   if (q->length == 0)
   {
      value = NULL;
   }
   else
   {
      value = q->tail->data;
   }

   return value;
}

/*
 * free queue and its content from memory if needed
 *IMPORT: queue, integer indicating to free data or not
 */
void freeQueue(Queue* q, int freeOrNot)
{
   Qnode* node, *nextNode;

   node = q->head;/*start at front of q*/
   while (node != NULL)
   {
      nextNode = node->next;/*get next node*/
      if (freeOrNot == 1)/*if freeOrNot is 1
                           then free the data.
                           Otherwise don't free data*/
      {
         free(node->data);/*free malloced data*/
      }
      free(node);
      node = nextNode;/*make sure current is the next node*/
   }

   free(q);
}
```

Header files

```c
/**
 *Scheduler head file
 **/

#pragma once

#define SERVING 1
#define DONE 2

#include "fileIO.h"
#include "Queue.h"

//process struct containing info
typedef struct
{
        int task_id;
        int cpu_burst;
        int arrive_hr, arrive_min, arrive_sec;
        int serv_hr, serv_min, serv_sec;
        int comp_hr, comp_min, comp_sec;
}process;

void* task(void* arg);
void* cpu(void *arg);
void getTime(int *hr, int *min, int *sec);
void calcWaitTime(process *inTask);
void calcTurnAround(process *inTask);
void taskThreadStats(process *task);
void cpuThreadStats(process *task, char *cpu_name, int status);
void taskTimeStats();
int criticalNotEmpty(char *nameCpu);



/*
 *Queue header file
 */
#pragma once

typedef struct node
{
   void* data;
   struct node* next;
}Qnode;

typedef struct
{
   Qnode* head;
   Qnode* tail;
   int length;
}Queue;

Queue* createQueue();
void enqueue(Queue* q, void* data);
void* dequeue(Queue* q);
int queueEmpty(Queue* q);
void* peekFirst(Queue *q);
void* peekLast(Queue *q);
void freeQueue(Queue* q, int freeOrNot);
```

```c
/*
 * fileIO header file
 */
#pragma once

#include "Queue.h"
#include "scheduler.h"

int readFile(char *fileName, Queue *q);
int writeToFile(char *fileName, char *str);
int writeTaskTimeStats(char *fileName, float num_tasks, float avg_wait, float avg_TAT);
int wipeLog(char *fileName);
```

Make file

```
EXEC = scheduler
UNITTEST = qUnitTest
OBJTEST = UnitTestQueue.o Queue.o
OBJ = scheduler.o fileIO.o Queue.o
CFLAGS = -Wall -g -lpthread
CC = gcc

all : $(UNITTEST) $(EXEC)

$(EXEC) : $(OBJ)
	$(CC) $(OBJ) -o $(EXEC) $(CFLAGS)

$(UNITTEST) : $(OBJTEST)
	$(CC) $(OBJTEST) -o $(UNITTEST) $(CFLAGS)

scheduler.o : scheduler.c scheduler.h fileIO.h Queue.h
	$(CC) -c scheduler.c $(CFLAGS)

Queue.o : Queue.c Queue.h
	$(CC) -c Queue.c $(CFLAGS)

fileIO.o : fileIO.c fileIO.h scheduler.h
	$(CC) -c fileIO.c $(CFLAGS)

UnitTestQueue.o : UnitTestQueue.c Queue.h
	$(CC) -c UnitTestQueue.c $(CFLAGS)

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

Extras

Task_file generator: gentask.sh

```
#!/bin/bash

size=$1
count=1

if [ "$size" -gt 0 ]; then
    while [ $count -le $size ]; do
        echo $count $(((RANDOM % 50) + 1))
       count=$(($count + 1))
    done > task_file
    echo "Done task_file"
else
    echo "Command arg must be greater than 0"
fi
```

Queue test harness

```
/*
 * Unit test for linked q
 *
 * by Kei Sum Wang, student id: 19126089
 * REFERENCE: this is from the unit Unix and C programming
 */

#include <stdio.h>
#include <stdlib.h>
#include "Queue.h"

void printFormat(void* data);
void testLastOps(Queue *q);


int main()
{
    Queue *q = createQueue();

    printf("====TEST HARNESS FOR Queue=======\n");

    printf("\n====Test creation of Queue=======\n");
    if (q->head == NULL && q->tail == NULL && q->length == 0)
    {
        printf("q creation success\n");
    }
    else
    {
        printf("creation of q failed\n");
    }

    printf("\n==========TESTING LAST OPERATIONS=============\n");
    testLastOps(q);

    freeQueue(q, 0);

    return 0;
}

void testLastOps(Queue *q)
{
    int d = 4, e = 5, f = 6;
    int *p1, *p2, *p3, *top, *end;

    printf("\n====enqueue. values %d, %d, %d\n", d, e, f);
    enqueue(q, &d);
    enqueue(q, &e);
    enqueue(q, &f);

    printf("===check first and last values and length with qs peekFirst(), peekLast() and length field===\n");
    top = peekFirst(q);
    end = peekLast(q);


    printf("peekFirst %d\n", *top);
    printf("peekLast %d\n", *end);
    printf("insert length %d\n", q->length);

    printf("====dequeue. expected order of delete 4, 5, 6\n");
    p1 = dequeue(q);
    printf("delete %d\n", *p1);
    p2 = dequeue(q);
    printf("delete %d\n", *p2);
    p3 = dequeue(q);
    printf("delete %d\n", *p3);
    printf("%d was last deletion next delete will indicate nothing to delete\n", *p3);
    dequeue(q);

    printf("delete length %d\n", q->length);
}


/*print function for linkedq displayList method*/
void printFormat(void* data)
{
    int *num;

    num = (int*)data;

    printf("%d\n", *num);
}
```

How to run

1. type "make" to compile

2. run ./gentask.sh #tasks to generate task_file
   - #tasks - number of task u want to generate

3. ./scheduler file_name m
   - m is size of buffer between 1 to 10

List of headers included in scheduler, fileIO and Queue source code

Scheduler.c

- #include <stdio.h>
- #include <stdlib.h>
- #include <time.h>
- #include <pthread.h>
- #include <unistd.h>
- #include "scheduler.h"
- #include "fileIO.h"
- #include "Queue.h"

fileIO.c

- #include <stdio.h>
- #include <string.h>
- #include <stdlib.h>
- #include "fileIO.h"
- #include "Queue.h"
- #include "scheduler.h"

Queue.c

- #include <stdio.h>
- #include <stdlib.h>
- #include "Queue.h"

Achieving Mutual Exclusion

Observations

- Based on producer and consumer problem
- Three Processor threads, a Task thread and a Ready-Queue that is shared among the threads.
- In general if there are no tasks in the ready queue, the processors must wait until at least one task is inserted into ready queue by task thread.
- If the number of tasks in ready queue reaches its limit, the task thread must wait until a processor takes a task out of ready queue to insert another again.
- Inserting and removing from ready queue is a critical section since race conditions are most likely to occur with these operations among the threads

Use of pthread library

pthread library are used to achieve mutual exclusion between the threads and the shared buffer.

- The shared buffer is implemented as a linked list based queue.
- Processors threads are created using pthread_create function and each of these runs the cpu() function which simulates cpu(consumer) operations.
- Task thread is created using pthread_create and runs the function task() which simulates producer operations.
- Pthread mutex is a mutex variable shared among the threads, this variable prevents other threads from entering critical sections if a thread currently is in critical section and holds lock and can be unlocked after a thread leaves critical section.
- Pthread condition variables full and empty are used to block a thread running. “full” is utilised by the cpu threads when the ready queue is empty but the file queue still contains tasks thus wait for task thread to fill the ready queue. Task thread uses the “empty” when the ready queue is full and waits until processors takes at least one task out.

Mutual exclusion is implemented in the task() and cpu() functions since critical sections are present in these functions.

Task thread – if there are tasks in the file queue the loop begins, at this point task thread locks the mutex in order to insert into the ready queue. CPU threads cannot access the ready queue during task threads insert and must wait until **task thread** signals and unlocks the mutex or has to waits if the ready is queue full. As a result the insertion of tasks by task thread is done atomically without interruption by the CPU threads.
Outside of critical section of inserting task, task() does not modify lock thus satisfying progress condition where no thread outside critical section can stop other threads of entering the critical section.
When the task thread has no more tasks to insert into ready queue, task thread signals all cpu threads that are currently waiting to prevent waiting forever. This case occurs when number of tasks are less than the number of cpu threads, task thread will not signal all threads since it will iterate in respect to number of tasks. This satisfies the bounded waiting condition.

CPU threads – when the CPU threads begin cpu() function, the mutual exclusion lock and conditional wait begins when the while loop calls criticalNotEmpty() for the condition. The lock and wait begins in critcalNotEmpty() since we want to ensure that all threads check if the ready queue is empty or not atomically. If not atomically checked, race conditions could occur, for example, cpu threads may enter loop to get a task but there is only one task left in ready queue, one cpu thread will get last task thus ready queue is now empty, the other cpu threads will try to get a task but there are no tasks left. This issue occurred because all threads entered at the same time and checked that the ready queue was not empty, however multiple threads were in the critical section thus race condition occurred. Therefore checking if ready queue is empty and removing from ready queue are both critical section for this cpu() function thus each cpu thread must check and access ready queue on at a time.
In relation to task thread, if a CPU is in critical section getting a task then task thread should also not access the ready queue at this point either to satisfy mutual exclusion.

Cpu threads also need to wait for task thread to insert tasks before entering the critical section since the cpu threads won't have anything to consume if they enter when ready queue is empty.

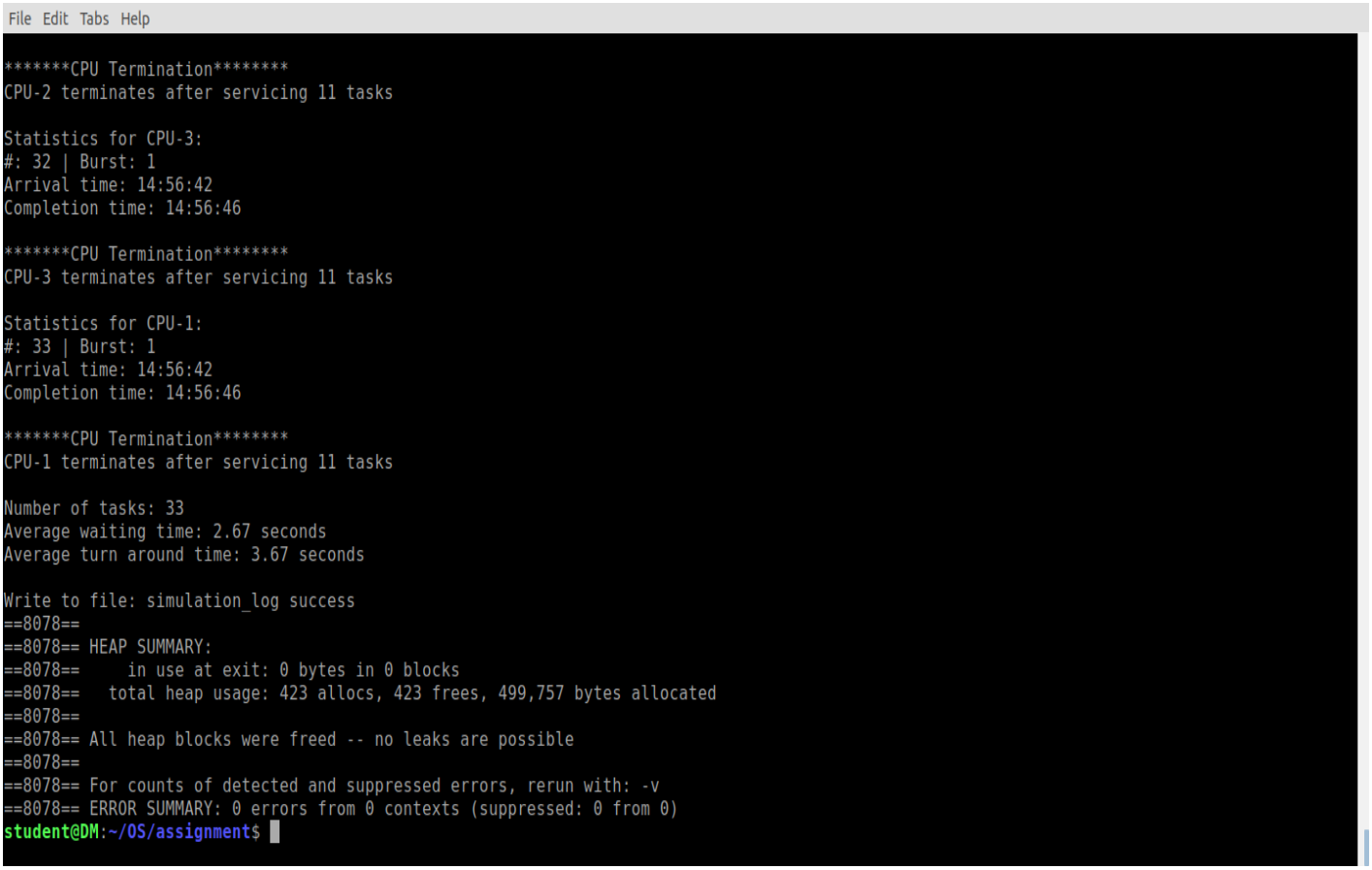
At the remainder section where cpu is consuming the lock is not touched or modified, this is to satisfy the progress condition.

After getting a task, cpu thread will signal the task thread to wake up and insert this prevent task thread from waiting forever thus satisfying bounded waiting condition.

Cpu() also unlocks mutex at the end of function since each iteration of loop sets the lock including when ready queue is empty checks. This also prevent incorrect ordering of termination statistics when comparing each cpus last task completion.

Are there any issues with the program?

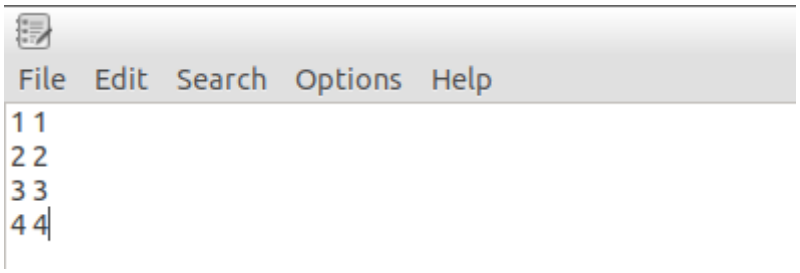
There are no known issues with the program such as leaks or other errors.



Valgrind output

Sample input and output

Sample task_file



running - ./scheduler order_in 4
in simulation log

#: 1 | Burst: 1
Arrival time: 15:6:48

#: 2 | Burst: 2
Arrival time: 15:6:48

Statistics for CPU-2:
#: 1 | Burst: 1
Arrival time: 15:6:48
Service time: 15:6:48

Statistics for CPU-1:
#: 2 | Burst: 2
Arrival time: 15:6:48
Service time: 15:6:48

#: 3 | Burst: 3
Arrival time: 15:6:48

#: 4 | Burst: 4
Arrival time: 15:6:48

Statistics for CPU-3:
#: 3 | Burst: 3
Arrival time: 15:6:48
Service time: 15:6:48

Number of tasks put into Ready-Queue: 4
Terminate at time: 15:6:48

Statistics for CPU-2:
#: 1 | Burst: 1
Arrival time: 15:6:48
Completion time: 15:6:49

Statistics for CPU-2:
#: 4 | Burst: 4
Arrival time: 15:6:48
Service time: 15:6:49

Statistics for CPU-1:
#: 2 | Burst: 2
Arrival time: 15:6:48

Completion time: 15:6:50

CPU-1 terminates after servicing 1 tasks

Statistics for CPU-3:
#: 3 | Burst: 3
Arrival time: 15:6:48
Completion time: 15:6:51

CPU-3 terminates after servicing 1 tasks

Statistics for CPU-2:
#: 4 | Burst: 4
Arrival time: 15:6:48
Completion time: 15:6:53

CPU-2 terminates after servicing 2 tasks

Number of tasks: 4
Average waiting time: 0.25 seconds
Average turn around time: 2.75 seconds