Operating Systems Assignment

Zrinski, Kyle (19324773)

Document version: 1.1 (2015-11-15)

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

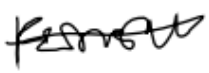
Last name:	Zrinski	Student ID:	19324773
Other name(s	Kyle Llewellyn		
Unit name:	Operating Systems	Unit ID:	COMP2006
Lecturer / unit coordinator:	Dr Sie Teng Soh	Tutor:	Arlen
Date of submission:	4th of May 2019	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessibl* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin)
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have don myself, specifically for this assessment. I cannot re-use the work of others, or my own previousl submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature  Date of signature: 4/5/2019

(By submitting this form, you indicate that you agree with all the above text.)

main.c

```
//system made header files
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <string.h>
#include <unistd.h> //for sleep() function

//user made header files
#include "fileIO.h"
#include "macros.h"
#include "log.h"

//static functions the rest of the program doesnt need to know about
static void* task(void* inArgs);
static void* cpu(void* args);
static int notInvalid(Task t);
static int validateCMD(char** arr, int count);

//global variables to be shared between threads
//ONLY MODIFIED BY THREADS or ACCESSED BY REFERENCE
int num_tasks;
int total_waiting_time;
int total_turnaround_time;

int main(int argc, char const *argv[])
{
    pthread_t CPU0,CPU1,CPU2,TASK; //decleration of 3 CPU threads
    pthread_mutex_t m; //decleration of mutex lock
    pthread_cond_t full, empty; //decleration of thread condition

    int ii; //integer iterator & error flag for task/main thread
    cpuArgs** arg = (cpuArgs**)malloc(3 * sizeof(cpuArgs*)); //struct for
variables to passed to CPU threads
    CPU** processors = (CPU**)malloc(3 * sizeof(CPU*)); //struct for holding CPU
id and number of tasks each CPU services
    taskArgs* tArg = (taskArgs*)malloc(sizeof(taskArgs)); //struct holding
variables passed to task()
    Queue* q = NULL; //Queue decleration
    FILE* f = NULL; //file decleration
    int* taskSize = (int*)malloc(sizeof(int));
    num_tasks = 0; //num_tasks initialisation

    char taskFile[20]; //declaring stack memory for taskFile.
    char logFile[] = "./simulation_log"; //name of log file in this case always
called simulation_log and to be stored in the root directory
    remove(logFile); //delete any old simulation_logs

    if(validateCMD((char**)argv,argc)) //ensure cmdline arguments are valid
before attempting to open filename from cmdline
    {
        strcpy(taskFile,argv[1]);
        f = openFile(taskFile);
    }

    //initialising mutex's and conditions;
    pthread_mutex_init(&m,NULL);
    pthread_cond_init(&full,NULL);
    pthread_cond_init(&empty,NULL);
```

```
    if(f != NULL && validateFile(taskFile) && validateCMD((char**)argv,argc)) //
check all FileIO and cmdline is valid before starting
    {
        Queue* q = createQueue(atoi(argv[2])); //creates a Queue the size of 3rd
cmdline arg
        *taskSize = fileLines(taskFile); //getting number of tasks in file.

        //initialising thread args and CPU info for each thread
        for(ii = 0; ii < 3; ii++)
        {
            processors[ii] = (CPU*)malloc(sizeof(CPU)); //malloc CPU info to be
passed around
            processors[ii]->id = ii;
            processors[ii]->tasksServed = 0;

            arg[ii] = (cpuArgs*)malloc(sizeof(cpuArgs)); //malloc thread args
for CPU-ii
            arg[ii]->c = processors[ii]; //thread arg-ii gets processor-ii for
(thread) CPU-ii
            arg[ii]->queue = q; //pointer to queue
            arg[ii]->tFile = f; //pointer to task file
            arg[ii]->tSize = taskSize; //informing cpu's how many tasks to
complete
            arg[ii]->logF = logFile; //pointer to log file

            //addresses of mutex and conditions which are shared between threads
            arg[ii]->mutex = &m;
            arg[ii]->full = &full;
            arg[ii]->empty = &empty;
        }

        //initialising struct to be passed to task thread
        tArg->q = q;
        tArg->tFile = f;
        tArg->simLog = logFile;
        tArg->mutex = &m;
        tArg->full = &full;
        tArg->empty = &empty;

        //If successful in the creating of thread x print to stderr.
        if(pthread_create(&CPU0,NULL,&cpu,(void*)arg[0]) == 0)
        {
            DEBUG_FPRINTF(stderr,"%s###LOG### [CPU-0] Successfully started\
n","");
        }

        if(pthread_create(&CPU1,NULL,&cpu,(void*)arg[1]) == 0)
        {
            DEBUG_FPRINTF(stderr,"%s###LOG### [CPU-1] Successfully started\
n","");
        }

        if(pthread_create(&CPU2,NULL,&cpu,(void*)arg[2]) == 0)
        {
            DEBUG_FPRINTF(stderr,"%s###LOG### [CPU-2] Successfully started\
n","");
        }

        if(pthread_create(&TASK,NULL,&task,(void*)tArg) == 0)
        {
            DEBUG_FPRINTF(stderr,"%s###LOG### [TASK] Successfully started\
n","");
        }
```

```
        //wait for CPU threads to complete before continuing
        pthread_join(CPU0,NULL);
        pthread_join(CPU1,NULL);
        pthread_join(CPU2,NULL);
        pthread_join(TASK,NULL);

        //log completion of task and CPU threads
        logDone(logFile, num_tasks, total_waiting_time, total_turnaround_time);

        //free memory in the succesful validation case
        for(ii = 0; ii < 3; ii++)
        {
            //free(arg[ii]->tSize);
            free(processors[ii]); //free cpu info
            free(arg[ii]); //free thread data
        }

        free(taskSize);
        free(tArg);
        free(processors); //free outer array for cpu info
        free(arg); //free outer array for thread data

        freeQ(q); //free queue
        closeFile(f); //close file

        //destroy mutex and conditions
        pthread_mutex_destroy(&m);
        pthread_cond_destroy(&empty);
        pthread_cond_destroy(&full);
    }
    else if (argc != 3) //if incorrect # of cmd args
    {
        if(f != NULL)
        {
            fclose(f); //close only if opened
        }
        fprintf(stderr,"###ERROR### Invalid # command line args\n"); //print
error
        free(taskSize);
        free(tArg);
        free(processors); //same as previous free in successful case
        free(arg); // ^^

        if(q != NULL)
        {
            freeQ(q); //free queue if created
        }
    }
    else //same as else if but different error message
    {
        fprintf(stderr,"###ERROR### Error in cmdline arguments\n");
        if(f != NULL)
        {
            fclose(f);
            fprintf(stderr,"###ERROR### Invalid/Missing file: %s\n",taskFile);
        }
        free(taskSize);
        free(tArg);
        free(processors);
        free(arg);

        if(q != NULL)
        {
            freeQ(q);
```

```
        }
    }

    return 0;
}

//retrieve 1 or 2 tasks from task_file based on free space
static void* task(void* inArgs)
{
    int error = FALSE; //error flag

    Task task1,task2;

    taskArgs* args = (taskArgs*)inArgs; //convert void pointer to taskArgs

    while(error == FALSE) //while not at EOF or invalid line
    {
        pthread_mutex_lock(args->mutex);
        while(isFull(args->q) == TRUE)
        {
            pthread_cond_wait(args->full,args->mutex); //wait for consumer to
consume an item
        }
        pthread_mutex_unlock(args->mutex);

        pthread_mutex_lock(args->mutex);
        if(isFull(args->q) == FALSE) //if ANY room
        {
            task1 = getTask(args->tFile); //retrieve from file
            if(notInvalid(task1)) //check validity
            {
                logArrival(args->simLog,task1); //log insertion to queue
                ins(task1,args->q); // insert to queue
                num_tasks++; //increment read elements
            }
            else
            {
                error = -1; //return error code
                ins(task1,args->q); //insert poison pill
            }
        }
        pthread_mutex_unlock(args->mutex);
        pthread_cond_signal(args->empty); //signal consumer of new items to
consume


        //same as above but for a second task
        pthread_mutex_unlock(args->mutex);
        if(isFull(args->q) == FALSE)
        {
            task2 = getTask(args->tFile);
            if(notInvalid(task2))
            {
                logArrival(args->simLog,task2); //log and insert into queue
                ins(task2,args->q);
                num_tasks++;
            }
            else
            {
                error = -1;
                ins(task2,args->q);
            }
        }
```

```
        DEBUG_FPRINTF(stderr, "###LOG### tasks read from file: %d\n",num_tasks);
//debug print to stderr
        pthread_mutex_unlock(args->mutex);
        pthread_cond_signal(args->empty); //signal consumer of new items to
consume
    }

    logTask(args->simLog, num_tasks); //log completion of task/main thread
    DEBUG_FPRINTF(stderr, "%s###LOG### [TASK] Stopping\n","");

    return NULL;
}

//function which runs on pthread_create for CPU's
static void* cpu(void* args)
{
    static int count = 0; //count of total tasks serviced
    int done = FALSE; //flag to indicate all tasks have been serviced
    Task temp;

    cpuArgs* details = (cpuArgs*)args; //conversion from void* back to cpuArgs*

    while(!done)
    {
        pthread_mutex_lock(details->mutex);
        while(hasElement(details->queue) == FALSE && count < *(details-
>tSize)) //while queue empty
        {
            pthread_cond_wait(details->empty,details->mutex); //wait for
producer to produce
        }
        temp = rem(details->queue); //take element from queue
        pthread_cond_signal(details->full); //signal producer to produce
        count++;
        pthread_mutex_unlock(details->mutex);

        if(temp.id > 0 && count <= *(details->tSize)) //if poison pill not
reached and under max size
        {
            pthread_mutex_lock(details->mutex);
            total_waiting_time += logService(details->logF,temp,*details->c); //
add to total wait time and log service
            pthread_mutex_unlock(details->mutex);

            sleep(temp.burst); //"process" task
            details->c->tasksServed++; //add to individual CPU count

            pthread_mutex_lock(details->mutex);
            total_turnaround_time += logComplete(details->logF,temp,*details-
>c); //add to total turn around and log completion
            free(temp.aTime); //free arrival time for task as its no longer
used.
            pthread_mutex_unlock(details->mutex);
        }
        else
        {
            done = TRUE; //if poison pill reached exit loop to kill thread
        }
    }

    logCPU(details->logF,*details->c); //log cpu thread finishing
    DEBUG_FPRINTF(stderr, "###LOG### [CPU-%d] Stopping\n", details->c->id);
//debug print to stderr
    return NULL; //return a void* no return so NULL is used.
```

```
}

//basic check that task if not a poison pill (negative number)
static int notInvalid(Task t)
{
    int result = TRUE;

    if(t.id <= 0 || t.burst <= 0)
    {
        result = FALSE; //INVALID
    }

    return result;
}

//validate the command line arguments both for content and amount
static int validateCMD(char** arr, int count)
{
    int valid = FALSE; //default cmdline args are invalid until proven valid

    if(count == 3) //make sure there is a program name, file and queue size
    {
        if(atoi(arr[2]) <= 10 && atoi(arr[2]) > 0) //check valid queue size
        {
            valid = TRUE; //valid cmdline args
        }
    }

    return valid;
}
```

Queue.c

```
#include <stdio.h>
#include <stdlib.h>

#include "Queue.h"

Queue* createQueue(int size)
{
    int ii;
    Queue* q = (Queue*)malloc(sizeof(Queue) + (sizeof(Task*) * size));
//assigning memory to struct and flexible array member
    q->count = 0;
    q->max = size;

    for(ii = 0; ii < q->max; ii++)
    {
        q->arr[ii] = (Task*)malloc(sizeof(Task)); //initialising each task in
queue
        q->arr[ii]->id = -1; //-1 for purposely empty values
        q->arr[ii]->burst = -1;
    }

    return q;
}

void ins(Task x, Queue* q)
{
    if(q->count < q->max) //if free spot in queue
    {
        *(q->arr[q->count]) = x; //add item x at end of queue
```

```
        (q->count)++; //increment last item index counter
    }
    else
    {
        perror("Queue full");
    }
}

Task rem(Queue* q)
{
    Task ret;
    int ii;
    ret = *(q->arr[0]); //storing value to be returned

    for(ii = 1; ii < q->count; ii++)
    {
        *(q->arr[ii-1]) = *(q->arr[ii]); //shufflinf the queue down after
removal
    }

    if(q->count > 0)
    {
        q->arr[q->count-1]->id = -1; //adding blank item to end of queue after
shuffle
        q->arr[q->count-1]->burst = -1;
        (q->count)--;
    }

    return ret;
}

void freeQ(Queue* q)
{
    int ii;

    for(ii = 0; ii < q->max; ii++)
    {
        free(q->arr[ii]); //freeing elements from queue
    }

    free(q); //freeing queue itself
}

int hasElement(Queue* q)
{
    int ret = FALSE;
    if(q->count > 0)
    {
        ret = TRUE; //has more than one element
    }

    return ret;
}

int isFull(Queue* q)
{
    int ret = FALSE;
    if(q->count >= q->max)
    {
        ret = TRUE; //has no free spaces
    }

    return ret;
}
```

fileIO.c

```
#include "fileIO.h"

static Task split(char* line);

//function which checks a file contains at least SOMETHING and leaves it open.
FILE* openFile(char* fName)
{
    FILE* file;
    int ch;

    file = fopen(fName,"r"); //opening file in read mode

    if(file != NULL)
    {
        ch = fgetc(file);
        if(ch != EOF) // if file isnt immediately empty rewind it.
        {
            rewind(file);
        }
        else
        {
            perror("###Error### file empty");
        }
    }
    else
    {
        perror("###Error### Error opening file");
    }

    return file; //return file pointer

}

//ensures that a task file follows the appropriate format and contains no blank
or incorrect lines.
int validateFile(char* file)
{
    char temp[LEN]; // temp string to check each line
    int tempID, tempBurst; //temp string to check values of id and burst
    int ret = FALSE; //returns whether the file is valid (TRUE/FALSE) default is
False
    int error = FALSE; //returns IF a line is invalid at ANY point, default is
False.
    FILE* f = fopen(file,"r");

    if(f != NULL) //if file exists
    {
        while(fgets(temp,LEN,f) != NULL) //if line not empty or EOF
        {
            if(fscanf(f,"%d %d\n",&tempID,&tempBurst) == 2) //if only contains 2
integers
            {
                if(tempBurst > 50 && tempBurst <= 0) //if burst not within
limits of 1-50 inclusive
                {
                    error = TRUE; //error occured
                }
            }
```

```
            else
            {
                error = TRUE; //error occured
            }
        }

        //if no errors occured the file is valid
        if(!error)
        {
            ret = TRUE;
        }

        fclose(f); //close file so it may be opened again at the top/
    }

    return ret;
}

//writes a string to specified file.
int writeFile(char* fName, char* entry)
{
    FILE* file;
    file = fopen(fName,"a"); //open file in append mode.
    int error = FALSE; //error flag for return.

    if(file != NULL) //if file opens successfully.
    {
        fputs("---\n",file); //To allow for easy differentiation between entries

        if(entry != NULL)
        {
            fputs(entry,file); //add line to file
            fflush(stdout); //making sure line is printed to file
        }
        else //entry is null
        {
            perror("Invalid string");
            error = TRUE;
        }

        if(ferror(file))//checking for file errors
        {
            perror("###Error### Error writing to file: ");
            fclose(file); //close file after error
            error = TRUE;
        }//else "do nothing"

        fclose(file); //close file after done writing
    }
    else
    {
        perror("###Error### Error Opening log file: ");
        error = TRUE;
    }

    return error; //returns error code to calling function
}

//retrieves a single task from an already open file.
Task getTask(FILE* file)
{
    Task t = {-1,-1}; //initialise Task with "poison pill" values so anything
that reads a NULL task knows it to be purposefully invalid
    char* line = malloc(LEN * sizeof(char)); //holds line read in from file
```

```
    static int count = 1; //maintains a count of which line the program is up to
for Warnings.

    if(feof(file) == 0) //if not at EOF
    {
        if(fgets(line,LEN,file) != NULL) //if next line read is not NULL or EOF
        {
            t = split(line); //split line
            t.aTime = (char*)malloc(9 * sizeof(char)); //allocate memory for
valid task
            count++;
        }
        else
        {
            fprintf(stderr,"###Warning### Invalid/EOF line at %d\n",count);
        }
    }
    else
    {
        t.id = -2; //poison pill values for EOF
        t.burst = -2;
        //t.aTime = (char*)malloc(9 * sizeof(char));
    }

    free(line); //line has been copied elsewhere and may now be free'd

    return t; //return read in task valid or not.
}

//splits a line on spaces and returns a Task.
static Task split(char* line)
{
    char str1[LEN], str2[LEN], str3[LEN];
    int numRead;

    Task t;

    numRead = sscanf(line, "%s %s %s\n",str1,str2,str3); //checks for correct
number and format of line
    if(numRead == 2)
    {
        t.id = atoi(str1); //convert id string to integer
        t.burst = atoi(str2); //converts burst string to integer
    }
    else
    {
        perror("###Error### Invalid parameter");
    }

    return t;
}

//function that returns the number of lines in a file
int fileLines(char* fName)
{
    int count = 0;
    char temp[LEN]; //temp string to hold line and "validate"
    FILE* f = fopen(fName,"r");

    if(f != NULL)//if file exists
    {
        while(fgets(temp,LEN,f) != NULL) //if line not empty or EOF
        {
            count++;
```

```
        }
    }

    fclose(f);

    return count;
}

//closes an already open file and checks for error.
void closeFile(FILE* file)
{
    if(fclose(file) != 0)
    {
        perror("###Error### Could not close file");
    }
}
```

log.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>

//#include "Queue.h"
#include "fileIO.h"
#include "macros.h"
#include "log.h"

static int timeDiff(char* before, char* after); // static function for getting
difference in seconds
static int convertSeconds(char* time); //converts time string to integer of
seconds

static char* getTime() //gets current system time via ctime() function
{
    time_t curTime;
    char* strTime = (char*)malloc(9 * sizeof(char)); //HH:MM:ss and room for
null terminator

    time(&curTime);
    sscanf(ctime(&curTime),"%*s %*s %*d %s:%s:%s %*s", strTime, &strTime[2],
&strTime[4]); //sending relevent info to strTime

    return strTime;
}

//Logs the arrival of a task to the queue and gives a value to the task structs
aTime value.
void logArrival(char* fName, Task t)
{
    char entry[LEN];
    char* currTime = getTime();
    strcpy(t.aTime,currTime);

    sprintf(entry, "task#%d: %d\nArrival time: %s\n", t.id, t.burst,t.aTime);
    writeFile(fName,entry); //write to simulation_log

    free(currTime); // as currTime is copied the original can be free'd
}
```

```
//Logs when a CPU begins to service a task as well as returning the wait time of
the task
int logService(char* fName, Task t, CPU c)
{
    int waitTime;
    char entry1[LEN];
    char entry2[LEN];

    char* currTime = getTime();
    waitTime = timeDiff(t.aTime,currTime); //gets wait time to be returned

    sprintf(entry1,"Statistics for CPU-%d\ntask#%d: %d\n", c.id, t.id, t.burst);
    sprintf(entry2, "Arrival time: %s\nService time: %s\n", t.aTime, currTime);
    writeFile(fName,strcat(entry1,entry2)); //concats the two lines then sends
to simulation_log

    free(currTime); // as currTime is copied the original can be free'd

    return(waitTime);
}

//Logs when a CPU is finished servincing a task as well as returning the turn
around time of the task
int logComplete(char* fName, Task t, CPU c)
{
    int turnTime;
    char entry1[LEN];
    char entry2[LEN];

    char* currTime = getTime();
    turnTime = timeDiff(t.aTime,currTime); //gets turn around time to be
returned

    sprintf(entry1,"Statistics for CPU-%d\ntask#%d: %d\n", c.id, t.id, t.burst);
    sprintf(entry2, "Arrival time: %s\nCompletion time: %s\n", t.aTime,
currTime);
    writeFile(fName,strcat(entry1,entry2)); //concats the two lines then sends
to simulation_log

    free(currTime); //as currTime is no longer used it may be free'd
    return(turnTime);
}

//to be called when all tasks have been read from file.
void logTask(char* fName, int numT)
{
    char entry1[LEN];
    char entry2[LEN];
    char* currTime = getTime();

    sprintf(entry1,"Number of tasks put into Ready-Queue: %d\n", numT);
    sprintf(entry2, "Terminated at time: %s\n", currTime);
    writeFile(fName,strcat(entry1,entry2)); //concats the two lines then sends
to simulation_log

    free(currTime); //as currTime is no longer used it may be free'd
}

//to be called when a CPU has no more tasks to be serviced
void logCPU(char* fName, CPU c)
{
    char entry1[LEN];
```

```
    sprintf(entry1,"CPU-%d terminates after servicing %d tasks\n", c.id,
c.tasksServed);
    writeFile(fName,entry1);

}

//to be called when all threads have completed, also calculating the average
wait and turn around times.
void logDone(char* fName, int numT, int wTime, int tTime)
{
    char entry1[LEN];
    char entry2[LEN];
    char entry3[LEN];

    sprintf(entry1, "Number of tasks: %d\n", numT);
    sprintf(entry2, "Average waiting time: %ds\n", (wTime/numT));
    sprintf(entry3, "Average turn around time: %ds\n", (tTime/numT));
    strcat(entry2,entry3); //concats 2 string to make one
    writeFile(fName,strcat(entry1,entry2)); //concats the two lines then sends
to simulation_log.

}

//returns an integer (in seconds) of the difference between two times
static int timeDiff(char* before, char* after)
{
    int diff = 0;

    diff = (convertSeconds(after) - convertSeconds(before));

    return diff;
}

//returns an integer of how many seconds a HH:mm:ss formated string contains
static int convertSeconds(char* t)
{
    int totalSecs = 0;
    int hours, mins, secs;

    sscanf(t,"%2d:%2d:%2d",&hours,&mins,&secs);

    totalSecs += secs;
    totalSecs += (mins * 60); // 60 seconds per minute
    totalSecs += (hours * 3600); // 3600 seconds per hour

    return totalSecs;
}
```

Discussion:

Mutual Exclusion:

Mutual exclusion is achieved through the use of POSIX thread mutexs and two POSIX thread conditions. The former is used protect the critical section, when the program is attempting to modify the Read_Queue or when the program is attempting to edit the

simulation_log file. The two condition statements ensure both synchronisation and mutual exclusion between the task() thread (producer) and the cpu() threads (consumers). The “full” condition is used to signal the cpu() threads to begin consuming from the Queue, the “empty” condition is used to signal the task() thread that there free element/s in the queue. These two conditions ensure that the producer threads and consumer threads wait for each other to do their respective jobs and not over-produce or over-consume.

Testing:

The program works according to assignment specification and works in all cases. The program has been tested in a variety of test cases with no memory leaks or segmentation faults. The command line arguments were all tested, file name/location and queue size for validity (correct value and data types). The program was also tested with a variety of false file formats for the task_file. Random blank lines within a valid file, values in the right format but wrong data types, burst times exceeding 50 seconds, blank files, negative id’s and burst times, non-existant files.

Sample IO:

Compiling via make:

```
kz@kz-xps:~/OS2019$ make
[COMPILING] main.c
[COMPILING] log.c
[COMPILING] fileIO.c
[COMPILING] Queue.c
[LINKING] ./build/main.o ./build/log.o ./build/fileIO.o ./build/Queue.o
kz@kz-xps:~/OS2019$
```

Compiling with DEBUG via make:

```
kz@kz-xps:~/OS2019$ make debug
[COMPILING] main.c
[COMPILING] log.c
[COMPILING] fileIO.c
[COMPILING] Queue.c
[COMPILING] Source files
[LINKING] Object files
[DONE] Compiled with DEBUG
```

Bad Queue Size:

```
kz@kz-xps:~/OS2019$ ./scheduler resource/task file20 0
###ERROR### Error in cmdline arguments
kz@kz-xps:~/OS2019$
```

Bad File Name:

```
kz@kz-xps:~/OS2019$ ./scheduler resource/doesnt_exist 5
###Error### Error opening file: No such file or directory
###ERROR### Error in cmdline arguments
kz@kz-xps:~/OS2019$
```

Discussion Cont. :

Bad File:

```
kz@kz-xps:~/OS2019$ ./scheduler resource/task_fileBAD 5
###ERROR### Error in cmdline arguments
###ERROR### Invalid/Missing file: resource/task_fileBAD
kz@kz-xps:~/OS2019$
```

```
kz@kz-xps:~/OS2019$ make
[COMPILING] main.c
[COMPILING] log.c
[COMPILING] fileIO.c
[COMPILING] Queue.c
[LINKING] ./build/main.o ./build/log.o ./build/fileIO.o ./build/Queue.o
kz@kz-xps:~/OS2019$ ./scheduler resource/task_file20 5
```

Good File without Logging:

Sample IO Cont. :

Good File with Logging:

```
kz@kz-xps:~/OS2019$ make debug
[COMPILING] main.c
[COMPILING] log.c
[COMPILING] fileIO.c
[COMPILING] Queue.c
[COMPILING] Source files
[LINKING] Object files
[DONE] Compiled with DEBUG
kz@kz-xps:~/OS2019$ ./scheduler resource/task_file20 5
###LOG### [CPU-0] Successfully started
###LOG### [CPU-1] Successfully started
###LOG### [CPU-2] Successfully started
###LOG### [TASK] Successfully started
###LOG### tasks read from file: 2
###LOG### tasks read from file: 4
###LOG### tasks read from file: 6
###LOG### tasks read from file: 8
###LOG### tasks read from file: 9
###LOG### tasks read from file: 10
###LOG### tasks read from file: 11
###LOG### tasks read from file: 12
###LOG### tasks read from file: 13
###LOG### tasks read from file: 14
###LOG### tasks read from file: 15
###LOG### tasks read from file: 16
###LOG### tasks read from file: 17
###LOG### tasks read from file: 18
###LOG### tasks read from file: 19
###LOG### tasks read from file: 20
###LOG### tasks read from file: 20
###LOG### [TASK] Stopping
###LOG### [CPU-1] Stopping
###LOG### [CPU-2] Stopping
###LOG### [CPU-0] Stopping
kz@kz-xps:~/OS2019$
```

Discussion Cont. :

Simulation Log from Good File:

```
---
task#1: 14
Arrival time: 17:13:46
---
task#2: 21
Arrival time: 17:13:46
---
task#3: 27
Arrival time: 17:13:46
---
Statistics for CPU-0
task#1: 14
Arrival time: 17:13:46
Service time: 17:13:46
---
Statistics for CPU-1
task#2: 21
Arrival time: 17:13:46
Service time: 17:13:46
---
task#4: 40
Arrival time: 17:13:46
---
Statistics for CPU-2
task#3: 27
Arrival time: 17:13:46
Service time: 17:13:46
---
task#5: 31
Arrival time: 17:13:46
---
task#6: 47
Arrival time: 17:13:46
---
task#7: 7
Arrival time: 17:13:46
---
task#8: 30
Arrival time: 17:13:46
---
Statistics for CPU-0
task#1: 14
Arrival time: 17:13:46
Completion time: 17:14:00
---
Statistics for CPU-0
task#4: 40
Arrival time: 17:13:46
Service time: 17:14:00
---
task#9: 49
Arrival time: 17:14:00
---
Statistics for CPU-1
task#2: 21
Arrival time: 17:13:46
Completion time: 17:14:07
---
Statistics for CPU-1
```

```
task#5: 31
Arrival time: 17:13:46
Service time: 17:14:07
---
task#10: 35
Arrival time: 17:14:07
---
Statistics for CPU-2
task#3: 27
Arrival time: 17:13:46
Completion time: 17:14:13
---
Statistics for CPU-2
task#6: 47
Arrival time: 17:13:46
Service time: 17:14:13
---
task#11: 14
Arrival time: 17:14:13
---
Statistics for CPU-1
task#5: 31
Arrival time: 17:13:46
Completion time: 17:14:38
---
Statistics for CPU-1
task#7: 7
Arrival time: 17:13:46
Service time: 17:14:38
---
task#12: 21
Arrival time: 17:14:38
---
Statistics for CPU-0
task#4: 40
Arrival time: 17:13:46
Completion time: 17:14:40
---
Statistics for CPU-0
task#8: 30
Arrival time: 17:13:46
Service time: 17:14:40
---
task#13: 27
Arrival time: 17:14:40
---
Statistics for CPU-1
task#7: 7
Arrival time: 17:13:46
Completion time: 17:14:45
---
Statistics for CPU-1
task#9: 49
Arrival time: 17:14:00
Service time: 17:14:45
---
task#14: 40
Arrival time: 17:14:45
---
Statistics for CPU-2
task#6: 47
Arrival time: 17:13:46
Completion time: 17:15:00
---
```

```
Statistics for CPU-2
task#10: 35
Arrival time: 17:14:07
Service time: 17:15:00
---
task#15: 31
Arrival time: 17:15:00
---
Statistics for CPU-0
task#8: 30
Arrival time: 17:13:46
Completion time: 17:15:10
---
Statistics for CPU-0
task#11: 14
Arrival time: 17:14:13
Service time: 17:15:10
---
task#16: 47
Arrival time: 17:15:10
---
Statistics for CPU-0
task#11: 14
Arrival time: 17:14:13
Completion time: 17:15:24
---
Statistics for CPU-0
task#12: 21
Arrival time: 17:14:38
Service time: 17:15:24
---
task#17: 7
Arrival time: 17:15:24
---
Statistics for CPU-1
task#9: 49
Arrival time: 17:14:00
Completion time: 17:15:34
---
Statistics for CPU-1
task#13: 27
Arrival time: 17:14:40
Service time: 17:15:34
---
task#18: 30
Arrival time: 17:15:34
---
Statistics for CPU-2
task#10: 35
Arrival time: 17:14:07
Completion time: 17:15:35
---
Statistics for CPU-2
task#14: 40
Arrival time: 17:14:45
Service time: 17:15:35
---
task#19: 49
Arrival time: 17:15:35
---
Statistics for CPU-0
task#12: 21
Arrival time: 17:14:38
Completion time: 17:15:45
```

```
---
Statistics for CPU-0
task#15: 31
Arrival time: 17:15:00
Service time: 17:15:45
---
task#20: 35
Arrival time: 17:15:45
---
Statistics for CPU-1
task#13: 27
Arrival time: 17:14:40
Completion time: 17:16:01
---
Statistics for CPU-1
task#16: 47
Arrival time: 17:15:10
Service time: 17:16:01
---
Number of tasks put into Ready-Queue: 20
Terminated at time: 17:16:01
---
Statistics for CPU-2
task#14: 40
Arrival time: 17:14:45
Completion time: 17:16:15
---
Statistics for CPU-2
task#17: 7
Arrival time: 17:15:24
Service time: 17:16:15
---
Statistics for CPU-0
task#15: 31
Arrival time: 17:15:00
Completion time: 17:16:16
---
Statistics for CPU-0
task#18: 30
Arrival time: 17:15:34
Service time: 17:16:16
---
Statistics for CPU-2
task#17: 7
Arrival time: 17:15:24
Completion time: 17:16:22
---
Statistics for CPU-2
task#19: 49
Arrival time: 17:15:35
Service time: 17:16:22
---
Statistics for CPU-0
task#18: 30
Arrival time: 17:15:34
Completion time: 17:16:46
---
Statistics for CPU-0
task#20: 35
Arrival time: 17:15:45
Service time: 17:16:46
---
Statistics for CPU-1
task#16: 47
```

```
Arrival time: 17:15:10
Completion time: 17:16:48
---
CPU-1 terminates after servicing 6 tasks
---
Statistics for CPU-2
task#19: 49
Arrival time: 17:15:35
Completion time: 17:17:11
---
CPU-2 terminates after servicing 6 tasks
---
Statistics for CPU-0
task#20: 35
Arrival time: 17:15:45
Completion time: 17:17:21
---
CPU-0 terminates after servicing 8 tasks
---
Number of tasks: 20
Average waiting time: 38s
Average turn around time: 68s
```