OS Assignment – CPU Scheduler
Jhi Morris (19173632) – 05/05/19

Part 1 Explanation

The only resources which the *task* and *cpu* threads share are the Ready-Queue and access to write to the log file. When *task* needs to load data from the task_file, it can do so safely as no other thread accesses that file stream. When inserting the loaded data into the queue, it must get exclusive access first, and release it once finished; this is implemented using the *pthread_mutex_lock* and *pthread_mutex_unlock* functions. Naturally this is required to avoid corruption of the queue or the nodes contained within it. The mutex for this management is stored within the struct for the queue itself; so that it can also be accessed easily by the *cpu* threads when they pop tasks off of the queue to execute.

Access to write to the log file is managed with a similar implementation; the mutex is stored in a struct (*logAccess*) which both *task* and the *cpu*s share access to. This may not be required as all log messages are smaller than the minimum buffer size for the stdout pipe. However the case may occur where multiple log entries are created before stdout can process the data and the buffer reaches capacity; at which point the thread attempting to add data to the buffer will block, and some other thread may then attempt to write data to the buffer. It could occur that between the first thread blocking due to a full buffer and the second thread attempting to create a log that the buffer empties; in which case the log entry of the second thread will begin part way through the log of the first thread. For this reason I have implemented mutual exclusion on writing to the log.

Note that the times logged in the simulation_log are printed in the current system time. If your BIOS time is set to UTC time, the timestamps logged will be in UTC, not your local time. The only odd other implementation decision I have made with this assignment is that invalid lines of the task_file are simply ignored, and the total amount skipped is logged by *task* after it has reached the end of the file. It would be trivial to add a check in *main* to run through the task_file and validate every line, then fail to start if an invalid line is found. As there is no requirement for this, I have made the choice not to implement this early check. Still, invalid lines should not be processed or executed.

One of the requirements of *task* was that it loaded and inserted two tasks at a time into the Ready-Queue. I have implemented this implicitly, though not explicitly; as I will explain. If the queue length is equal to one, then this implementation is impossible and any solution will operate by adding a single task at a time. If the queue length is greater than two, then this is not a problem; however the issue may occur where there is space in the queue though not enough space for two tasks. There are two prototypical solutions to this problem: Firstly, an implementation could decide to add only one task and save the other for later Secondly it could alternatively cache both tasks and wait for the space required for both to enter the queue at once. Both of these are acceptable, although the first one may not seem so at first. My implementation can prove that both can be correct.

It simply does neither; and each cycle of the loop loads and then adds a single task at a time to the Ready-Queue. If the queue length is 1, then this operates as acceptably as any other solution. If the queue length is greater, it will insert two or more tasks during the execution of the program; though not during a single loop cycle. I argue that this technically meets the requirements of getting two functions at a time and putting them in the Ready-Queue, and that I have simply optimised the implementation to include less checks during the loop. The only difference is that my implementation loads the tasks from the file in two separate IO operations; not a single one. This has little

effect on performance and the assignment specification was not explicit in its requirement of how atomic the read operations of the two tasks involved must be. Furthermore, if the program were to support task_files with an odd number of tasks in them, it would likely need to load the tasks in two read operations regardless.

As my implementation operates similarly to the first of the two prototypical implementations I explained earlier, in that it loads two tasks and inserts both if possible. It differs in that it does not cache the second task when there is no room for it; rather it has not loaded it in the first place. The assignment specification states that the queue size is bounded with a configurable parameter, and it is fair to presume from this that this is so due to a limit in available memory for the theoretical system this scheduler would run on. Thus caching the remaining second task would require additional memory roughly equal to if it were just added to the queue in the first place; ergo defeating the purpose of the queue length limit entirely.

Part 2 Source Code

```
/* scheduler.c
#AUTHOR: Jhi Morris (19173632)
#MODIFIED: 27-04-19
#PURPOSE: Main class for the scheduler simulator program.
*/

#include "scheduler.h"

/* main
#RETURN: int (return code)
#ARGUMENTS: int (argc), char* array (arguments)
#PURPOSE: Prepares to run scheduler routine by loading required files and
      handling associated errors.*/
int main(int argc, char *argv[])
{
  FILE *input, *output;
  ErrorFlag error = NONE;
  TaskQueue *queue = malloc(sizeof(TaskQueue));

  if(argc == 3) //no args are optional
  { //args: 1=task_file, 2=queue capacity
    queue->head = NULL;
    queue->spaceLeft = atoi(argv[2]); //done here because of args
    queue->maxCap = queue->spaceLeft;
    pthread_mutex_init(&queue->mutex, NULL);

    if(queue->spaceLeft > 0)
    {
      input = fopen(argv[1], "r");

      if(input != NULL)
      {
        output = fopen("simulation_log", "w");

        if(output != NULL)
        {
          scheduler(queue, input, output); //begin actual operation
        }
        else
        {
          error = BAD_LOG;
        }
      }
      else
      {
        error = BAD_INPUT;
      }
    }
    else
    {
      error = INVALID_CAP;
    }
  }
  else
  {
    error = WRONG_ARGS;
  }

  pthread_mutex_destroy(&queue->mutex);
  free(queue);

  switch(error)
  {
    case NONE:
      fclose(input);
      fclose(output);
      break;
    case BAD_INPUT:
      perror("Unable to load input file:");
      break;
    case BAD_LOG:
      perror("Unable to load log file:");
```

```
      break;
    case INVALID_CAP:
      fprintf(stderr, "Second argument not recognized. Expecting for queue size an integer greater than zero.\n");
      break;
    case WRONG_ARGS:
      fprintf(stderr, "Invalid arguments.\nExpecting form: scheduler \"/path/to the/task_file\" queueSize. Where
queueSize is integer greater than zero.\n");
      break;
  }
}

/* scheduler
#RETURN: void
#ARGUMENTS: TaskQueue (Ready-Queue), FILE* (task_file), FILE* (log file)
#PURPOSE: Begins task() and cpu() threads, then handles wrap-up and logging.*/
void scheduler(TaskQueue *queue, FILE *input, FILE *output)
{
  int i;
  pthread_t reader;
  pthread_t cpus[NUM_CPUS];
  cpuData *cpuDatums[NUM_CPUS];
  int numTasks = 0;
  int totalWaitingTime = 0;
  int totalTurnaroundTime = 0;
  int *readFinish = malloc(sizeof(int)); //signal that task() is done
  taskData *read = malloc(sizeof(taskData));

  *readFinish = 0;

  read->log = malloc(sizeof(logAccess));
  read->log->output = output;
  pthread_mutex_init(&read->log->mutex, NULL);
  read->input = input;
  read->queue = queue;

  pthread_create(&reader, NULL, task, read);

  for(i = 0; i < NUM_CPUS; i++)
  {
    cpuDatums[i] = malloc(sizeof(cpuData));

    cpuDatums[i]->cpuNum = i;
    cpuDatums[i]->numTasks = 0;
    cpuDatums[i]->totalWaitingTime = 0;
    cpuDatums[i]->totalTurnaroundTime = 0;
    cpuDatums[i]->queue = queue;
    cpuDatums[i]->readFinish = readFinish;
    cpuDatums[i]->log = read->log;

    pthread_create(&cpus[i], NULL, cpu, cpuDatums[i]);
  }

  pthread_join(reader, NULL);
  *readFinish = 1; //now CPUs can end if queue is empty

  for(i = 0; i < NUM_CPUS; i++)
  { //slightly inefficient as it won't end a CPU thread until earlier CPU threads are done
    pthread_join(cpus[i], NULL);

    numTasks += cpuDatums[i]->numTasks;
    totalWaitingTime += cpuDatums[i]->totalWaitingTime;
    totalTurnaroundTime += cpuDatums[i]->totalTurnaroundTime;

    free(cpuDatums[i]);
  }

  /*log entry:
  Number of tasks: numTasks
  Average waiting time: totalWaitingTime/numTasks
  Average turn around time: totalTurnaroundTime/numTasks*/
  pthread_mutex_lock(&read->log->mutex);
  fprintf(read->log->output, "Number of tasks: %d\nAverage waiting time: %ld seconds\nAverage turn around time: %ld
seconds\n", numTasks, (totalWaitingTime/(long int)numTasks), (totalTurnaroundTime/(long int)numTasks));
  pthread_mutex_unlock(&read->log->mutex);

  pthread_mutex_destroy(&read->log->mutex);
```

```
  free(readFinish);
  free(read->log);
  free(read);
}

/* task
#RETURN: void* (unused)
#ARGUMENTS: void* (expecting taskData struct pointer)
#PURPOSE: Uses configuration data contained within taskData struct to load
      process tasks from task_file, inserting them into the queue.*/
void *task(void *in)
{
  TaskQueueNode *node;
  taskData *data = (taskData*)in; //cast now so as not to repeat
  int numTasks = 0; //running total for logging purposes
  int unrecognizedLines = 0;
  int inTask, inBurst, ret, done;

  while((ret = fscanf(data->input, "%i %i", &inTask, &inBurst)) != EOF)
  {
    if(ret == 2 && inTask > 0 && inBurst > 0) //validate
    {
      done = 0;
      numTasks++;

      //create new process node
      node = malloc(sizeof(TaskQueueNode));
      node->next = NULL;
      node->taskNum = inTask;
      node->burstDuration = inBurst;
      node->arrivalTime = time(0);

      while(!done)
      { //TODO convert this to use cond
        //add to queue
        pthread_mutex_lock(&data->queue->mutex);

        if(data->queue->spaceLeft > 0)
        {
          insertQueue(data->queue, node);
          done = 1;
        } //if there's no space, unlock and try later

        pthread_mutex_unlock(&data->queue->mutex);
      }

      /*log entry:
      taskNum: burstDuration
      Arrival time: arrivalTime*/
      pthread_mutex_lock(&data->log->mutex);
      fprintf(data->log->output, "Task %d: %d\nArrival time: %d:%d:%d\n", node->taskNum, node->burstDuration,
TIME(node->arrivalTime));
      pthread_mutex_unlock(&data->log->mutex);
    }
    else
    {
      unrecognizedLines++; //inc and skip line
    }
  }

  /*log entry:
  Number of tasks put into Ready-Queue: numTasks
  Terminate at time: time(0)*/
  pthread_mutex_lock(&data->log->mutex);
  fprintf(data->log->output, "Number of tasks put into Ready-Queue: %d\nInvalid lines skipped: %d\nTerminate at
time: %d:%d:%d\n", numTasks, unrecognizedLines, TIME(time(0)));
  pthread_mutex_unlock(&data->log->mutex);

  return NULL;
}

/* cpu
#RETURN: void* (unused)
#ARGUMENTS: void* (expecting cpuData struct pointer)
#PURPOSE: Processes tasks from queue (sleeping for burstDuration seconds) and
      logs appropriate information.*/
```

```
void *cpu(void *in)
{
  cpuData *data = (cpuData*)in;
  TaskQueueNode *task;
  time_t serviceTime, completionTime;

  while(!(*(data->readFinish)) || (data->queue->maxCap != data->queue->spaceLeft))
  { //keep looping until task() says it is done and queue is empty
    pthread_mutex_lock(&data->queue->mutex); //need to lock before checking if empty, incase pre-empted between
checking and locking

    if(data->queue->maxCap > data->queue->spaceLeft) //TODO change this to use cond
    { //ie if not empty
      task = popQueue(data->queue);

      pthread_mutex_unlock(&data->queue->mutex);

      serviceTime = time(0);
      data->totalWaitingTime += (serviceTime - task->arrivalTime);

      /*log entry:
      Statistics for CPU-data->cpuNum:
      Task task->taskNum
      Arrival time: task->arrivalTime
      Service time: serviceTime*/
      pthread_mutex_lock(&data->log->mutex);
      fprintf(data->log->output, "Statistics for CPU-%d:\nTask %d\nArrival time: %d:%d:%d\nService time: %d:%d:%d\
n", data->cpuNum, task->taskNum, TIME(task->arrivalTime), TIME(serviceTime));
      pthread_mutex_unlock(&data->log->mutex);

      sleep(task->burstDuration); //"process" task

      completionTime = time(0);
      data->totalTurnaroundTime += (completionTime - task->arrivalTime);

      /*log entry:
      Statistics for CPU-data->cpuNum:
      Task task->taskNum
      Arrival time: task->arrivalTime
      Completion time: completionTime*/
      pthread_mutex_lock(&data->log->mutex);
      fprintf(data->log->output, "Statistics for CPU-%d:\nTask %d\nArrival time: %d:%d:%d\nService time: %d:%d:%d\
n", data->cpuNum, task->taskNum, TIME(task->arrivalTime), TIME(completionTime));
      pthread_mutex_unlock(&data->log->mutex);

      free(task);
      data->numTasks++;
    } //if empty, skip activity and try again later
    else
    {
      pthread_mutex_unlock(&data->queue->mutex);
    }
  }

  /*log entry:
  CPU-data->cpuNum terminates after servicing data->numTasks tasks.*/
  pthread_mutex_lock(&data->log->mutex);
  fprintf(data->log->output, "CPU-%d terminates after servicing %d tasks.\n", data->cpuNum, data->numTasks);
  pthread_mutex_unlock(&data->log->mutex);

  return NULL;
}

/* insertQueue
#RETURN: void
#ARGUMENTS: TaskQueue* (queue), TaskQueueNode (node to be added)
#PURPOSE: Inserts node into tail of queue and decrements space left.*/
void insertQueue(TaskQueue *queue, TaskQueueNode *node)
{
  TaskQueueNode *next;

  if(queue->head == NULL)
  { //if the queue is empty, attach right away
    queue->head = node;
  }
  else
```

```
  {
    next = queue->head;

    while(next->next != NULL)
    { //traverse to find end of queue
      next = next->next;
    }

    next->next = node;
  }

  queue->spaceLeft--;
}

/* popQueue
#RETURN: TaskQueueNode* (node removed from queue)
#ARGUMENTS: TaskQueue* (queue)
#PURPOSE: Pops node from head of queue and increments space left.*/
TaskQueueNode *popQueue(TaskQueue *queue)
{
  TaskQueueNode *node;

  node = queue->head;
  queue->head = queue->head->next; //set head to second item in queue; removing first item

  queue->spaceLeft++;

  return node;
}
```

```
/* scheduler.h
#AUTHOR: Jhi Morris (19173632)
#MODIFIED: 27-04-19
#PURPOSE: Header file for scheduler.c.
*/

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <stdint.h>
#include <pthread.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <unistd.h>

//number of CPUs. Default is 3
#define NUM_CPUS 3

//conversion from unix time to hh:mm:ss
#define HOUR(n) (int)(((n)%86400)/3600)
#define MIN(n) (int)(((n)%3600)/60)
#define SEC(n) (int)(((n)%3600)%60)

//janky shortcut used for printf-like arguments. pairs with "%d:%d:%d" mask
#define TIME(n) HOUR(n), MIN(n), SEC(n)

/* ErrorFlag
#PURPOSE: Used by main() to handle error flags more cleanly.*/
typedef enum ErrorFlag {NONE, BAD_INPUT, BAD_LOG, INVALID_CAP, WRONG_ARGS} ErrorFlag;

/* logAccess
#PURPOSE: Stores data required to make log entries.*/
typedef struct logAccess
{
  FILE *output;
  pthread_mutex_t mutex;
} logAccess;

/* TaskQueueNode
#PURPOSE: Stores information about task as a queue node.*/
typedef struct TaskQueueNode
{
  struct TaskQueueNode *next;
  int taskNum;
  int burstDuration;
  time_t arrivalTime;
```

```
} TaskQueueNode;

/* TaskQueue
#PURPOSE: Stores data required to access task queue safely.*/
typedef struct TaskQueue
{
  TaskQueueNode *head;
  int spaceLeft;
  int maxCap;
  pthread_mutex_t mutex;
} TaskQueue;

/* taskData
#PURPOSE: Stores data required by task() thread to load from task_file into
      Ready-Queue and to log data.*/
typedef struct taskData
{
  FILE *input;
  TaskQueue *queue;
  int *readFinish;
  logAccess *log;
} taskData;

/* cpuData
#PURPOSE: Stores data required by cpu() thread to process task and to log data.*/
typedef struct cpuData
{
  int cpuNum;
  int numTasks;
  time_t totalWaitingTime;
  time_t totalTurnaroundTime;
  int *readFinish;
  TaskQueue *queue;
  logAccess *log;
} cpuData;

void scheduler(TaskQueue *queue, FILE *input, FILE *output);

void *task(void *in);

void *cpu(void *in);

void insertQueue(TaskQueue *queue, TaskQueueNode *node);

TaskQueueNode *popQueue(TaskQueue *queue);
```

Part 3 Sample Input & Output

The program can be run from the command line with the following argument structure:

./scheduler “task_file” n

Where n is a positive integer; the length of the Ready-Queue.
Below is a short sample task_file and the resultant output when run with a queue length of five.

```
1 2
2 5
3 2
5 5
4 6
6 15
1922 2
10 4
015 5
6 15
50 50
0 0
```

```
Task 1: 2
Arrival time: 8:56:0
Task 2: 5
Arrival time: 8:56:0
Task 3: 2
Arrival time: 8:56:0
Task 5: 5
Arrival time: 8:56:0
Task 4: 6
Arrival time: 8:56:0
Statistics for CPU-1:
Task 1
Arrival time: 8:56:0
Service time: 8:56:0
Statistics for CPU-2:
Task 2
Arrival time: 8:56:0
Service time: 8:56:0
Task 6: 15
Arrival time: 8:56:0
Task 1922: 2
Arrival time: 8:56:0
Statistics for CPU-0:
Task 3
Arrival time: 8:56:0
Service time: 8:56:0
Task 10: 4
Arrival time: 8:56:0
Statistics for CPU-1:
Task 1
```

```
Arrival time: 8:56:0
Service time: 8:56:2
Statistics for CPU-0:
Task 3
Arrival time: 8:56:0
Service time: 8:56:2
Statistics for CPU-1:
Task 5
Arrival time: 8:56:0
Service time: 8:56:2
Task 13: 5
Arrival time: 8:56:0
Statistics for CPU-0:
Task 4
Arrival time: 8:56:0
Service time: 8:56:2
Task 6: 15
Arrival time: 8:56:2
Statistics for CPU-2:
Task 2
Arrival time: 8:56:0
Service time: 8:56:5
Statistics for CPU-2:
Task 6
Arrival time: 8:56:0
Service time: 8:56:5
Task 50: 50
Arrival time: 8:56:2
Number of tasks put into Ready-Queue: 11
Invalid lines skipped: 1
Terminate at time: 8:56:5
Statistics for CPU-1:
Task 5
Arrival time: 8:56:0
Service time: 8:56:7
Statistics for CPU-1:
Task 1922
Arrival time: 8:56:0
Service time: 8:56:7
Statistics for CPU-0:
Task 4
Arrival time: 8:56:0
Service time: 8:56:8
Statistics for CPU-0:
Task 10
Arrival time: 8:56:0
Service time: 8:56:8
Statistics for CPU-1:
Task 1922
Arrival time: 8:56:0
```

```
Service time: 8:56:9
Statistics for CPU-1:
Task 13
Arrival time: 8:56:0
Service time: 8:56:9
Statistics for CPU-0:
Task 10
Arrival time: 8:56:0
Service time: 8:56:12
Statistics for CPU-0:
Task 6
Arrival time: 8:56:2
Service time: 8:56:12
Statistics for CPU-1:
Task 13
Arrival time: 8:56:0
Service time: 8:56:14
Statistics for CPU-1:
Task 50
Arrival time: 8:56:2
Service time: 8:56:14
Statistics for CPU-2:
Task 6
Arrival time: 8:56:0
Service time: 8:56:20
CPU-2 terminates after servicing 2 tasks.
Statistics for CPU-0:
Task 6
Arrival time: 8:56:2
Service time: 8:56:27
CPU-0 terminates after servicing 4 tasks.
Statistics for CPU-1:
Task 50
Arrival time: 8:56:2
Service time: 8:57:4
CPU-1 terminates after servicing 5 tasks.
Number of tasks: 11
Average waiting time: 5 seconds
Average turn around time: 15 seconds
```