Operating Systems Report

2019 S1 COMP2006 Assignment

Kevin Le - 19472960

Sections:

1. Mutual Exclusion

Mutual exclusion was achieved using the Posix Pthreads library with the use of 3 pthread condition variables (`qElement, qServiced, qEmpty`) and one pthread mutex variable (`mutex`). Critical sections involved accessing the queue to dequeue and enqueue, and using the global variables to increment task count, waiting time and turnaround time. With 3 CPU (reader) threads and 1 task (writer) thread, if any one of the four threads accessed the queue or the variables (the shared resources), it must first obtain a lock, which was done using `pthread_mutex_lock(&mutex)` using the `mutex` variable.

Upon obtaining this lock, the CPU that successfully obtains the lock will be able to dequeue a task from the queue and execute it, it would also be able to modify the global variables that all the CPUs share. Thus, mutual exclusion is achieved, since other CPUs that do not have the lock, will block until the mutex variable becomes available before entering their critical sections. The mutex is then released using `pthread_mutex_unlock(&mutex)`. A situation that must be considered is if the queue (buffer) does not contain any elements to dequeue, to prepare for this, a variable `pthread_cond_t qEmpty` is provided, for the CPU to signal the condition that the CPU is empty (which the writer receives), it then waits for the condition variable `qElement` with `pthread_cond_wait(&qElement, &mutex)` and releases the mutex for the writer. The writer then picks up and enqueues 2 more tasks, signalling the `qElement` variable for the CPU to resume work as elements now exist in the queue.

The task function obtains the lock on the queue in a similar fashion. It gets the lock using `pthread_mutex_lock(&mutex)` and then enqueues tasks. Mutual exclusion is achieved here as when the writer holds the lock, the CPU threads will have to block and cannot access the queue and global variables. Once elements have been added to the queue, it broadcasts to all 3 CPUs the signal `qElement` to notify of a new task. Then waiting for the CPUs to service the task with the condition `qServiced`. Then finally unlocking the mutex.

2. Shared Variables

- **`num_tasks`**
  - **Integer** that keeps track of the number of tasks. Increments for every task serviced by the CPU/s. Used to calculate final total averages.
- **`total_waiting_time`**
  - **Real** value that keeps track of the total waiting time across all tasks (how long it takes from arrival to when the task gets serviced). Incremented by adding the difference between service time and arrival time for each task serviced.
- **`total_turnaround_time`**
  - **Real** value that keeps track of the total turnaround time across all tasks (from arrival to completion)
- **`tasksInFile`**
  - **Integer** to hold count of number of tasks counted in file. Calculated by getTaskNum() in scheduler.c
- **`val`**
  - **Integer** to determine when CPU threads should exit. Depends on file line length due to task() in scheduler.c adding 2 tasks at a time. Exit conditions vary for odd length files. Set by getTaskNum(). Used by task() to signal when it has added the last task.
- **`lastTask`**
  - Exit condition for cpu(), required to keep CPU threads going until the final task has been serviced. Then allows graceful exit with changing of while condition.
- **`c1, c2, c3`**
  - **Integer** values containing thread ID of the spawned CPU threads. Used by getCPUID in scheduler.c to obtain the right CPU number by comparing thread IDs of current thread with the list of thread IDs. E.g. CPU 1.
- **`fileName`**
  - **String** (Char array) containing fileName, specified in run command. Stored globally to allow access by getNumTasks() and task() in scheduler.c

3. Testing

Known Issues:

1. *Doesn't finish execution with files containing an even number of lines or a file. Skips the final task in file as reader threads (CPU) exit prematurely.
2. Reader threads do not finish if file contains 1 line.

Test Cases:

Invalid Queue Size – PASS

./scheduler task_file 20 **or** -3

```
Incorrect Queue Size!
```

Invalid File – PASS

./scheduler does_not_exist 5

```
Error Opening File!
```

File containing 1 task – FAIL

```
T: 1 C: 1
Task Thread Finished
```

CPU threads do not exit.

File containing 25 (Cropped) tasks – PASS

```
T: 1 C: 3
T: 2 C: 1
T: 3 C: 2
T: 4 C: 1
T: 5 C: 1
T: 6 C: 2
T: 7 C: 1
T: 8 C: 2
T: 9 C: 1
T: 10 C: 3
T: 11 C: 2
T: 12 C: 1
T: 13 C: 2
T: 14 C: 3
T: 15 C: 3
T: 16 C: 2
T: 17 C: 3
T: 18 C: 3
T: 19 C: 1
T: 20 C: 2
T: 21 C: 23
T: 22 C: 38
T: 23 C: 11
T: 24 C: 29
T: 25 C: 17
Task Thread Finished
CPU Thread 1 Finished
CPU Thread 2 Finished
CPU Thread 3 Finished
```

File containing 10 tasks – FAIL*

./scheduler task_file X (X = any size queue)

```
T: 1 C: 1
T: 2 C: 1
T: 3 C: 2
T: 4 C: 3
T: 5 C: 2
T: 6 C: 3
T: 7 C: 1
T: 8 C: 1
T: 9 C: 2
Task Thread Finished
CPU Thread 1 Finished
CPU Thread 2 Finished
CPU Thread 3 Finished
```

Exits prematurely. Skipping final task. (Issue known to occur with even amount of lines)

Log File (Cropped) – PASS
Successfully writes expected log entries.

```
Task 7
Arrival Time: 19:06:10
Completion Time: 19:06:11
--
CPU-2 terminates after servicing 1 tasks
--
9:2
Arrival Time: 19:06:11
--
10:3
Arrival Time: 19:06:11
--
Statistics for CPU-3:
Task 8
Arrival time: 19:06:10
Service time: 19:06:11
--
Statistics for CPU-3:
Task 8
Arrival Time: 19:06:10
Completion Time: 19:06:12
--
CPU-3 terminates after servicing 5 tasks
--
Number of tasks put into Ready-Queue: 10
Terminated at time: 19:06:12
--
Statistics for CPU-1:
Task 9
Arrival time: 19:06:11
Service time: 19:06:12
--
Statistics for CPU-1:
Task 9
Arrival Time: 19:06:11
Completion Time: 19:06:14
--
CPU-1 terminates after servicing 3 tasks
--
Number of tasks: 9
Average waiting time: 0.78 seconds
Average turnaround time: 2.56 seconds
```

Using queue of every size (On file of size 20) – PASS

./scheduler task_file 3

```
T: 1 C: 3
T: 2 C: 1
T: 3 C: 2
T: 4 C: 1
T: 5 C: 1
T: 6 C: 2
T: 7 C: 1
T: 8 C: 2
T: 9 C: 1
T: 10 C: 3
T: 11 C: 2
T: 12 C: 1
T: 13 C: 2
T: 14 C: 3
T: 15 C: 3
T: 16 C: 2
T: 17 C: 3
T: 18 C: 3
T: 19 C: 1
Task Thread Finished
CPU Thread 1 Finished
CPU Thread 2 Finished
CPU Thread 3 Finished
```

Input File Sample:

```
1 14        11 13       21 23       31 2        41 41
2 44        12 31       22 38       32 6        42 10
3 43        13 34       23 11       33 20       43 36
4 10        14 18       24 29       34 50       44 20
5 15        15 45       25 17       35 18       45 5
6 13        16 10       26 32       36 40       46 20
7 21        17 17       27 4        37 27       47 33
8 35        18 18       28 3        38 19       48 38
9 2         19 20       29 11       39 43       49 19
10 31       20 18       30 20       40 6        50 42
```

Output from Input File – PASS*:

```
T: 1 C: 14   T: 11 C: 13  T: 21 C: 23  T: 31 C: 2
T: 2 C: 44   T: 12 C: 31  T: 22 C: 38  T: 32 C: 6   T: 41 C: 41
T: 3 C: 43   T: 13 C: 34  T: 23 C: 11  T: 33 C: 20  T: 42 C: 10
T: 4 C: 10   T: 14 C: 18  T: 24 C: 29  T: 34 C: 50  T: 43 C: 36
T: 5 C: 15   T: 15 C: 45  T: 25 C: 17  T: 35 C: 18  T: 44 C: 20
T: 6 C: 13   T: 16 C: 10  T: 26 C: 32  T: 36 C: 40  T: 45 C: 5
T: 7 C: 21   T: 17 C: 17  T: 27 C: 4   T: 37 C: 27  T: 46 C: 20
T: 8 C: 35   T: 18 C: 18  T: 28 C: 3   T: 38 C: 19  T: 47 C: 33
T: 9 C: 2    T: 19 C: 20  T: 29 C: 11  T: 39 C: 43  T: 48 C: 38
T: 10 C: 31  T: 20 C: 18  T: 30 C: 20  T: 40 C: 6   T: 49 C: 19
Task Thread Finished
CPU Thread 1 Finished
CPU Thread 2 Finished
CPU Thread 3 Finished
```

Valgrind – PASS:

```
==17457==
==17457== HEAP SUMMARY:
==17457==     in use at exit: 0 bytes in 0 blocks
==17457==   total heap usage: 154 allocs, 154 frees, 41,423 bytes allocated
==17457==
==17457== All heap blocks were freed -- no leaks are possible
==17457==
==17457== For counts of detected and suppressed errors, rerun with: -v
==17457== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 8 from 6)
```

4. Source Code

4.1 scheduler.c

```
#include "scheduler.h"
#include "queue.h"

queue* q;

/**
 * IMPORTS: Command line argument array and count
 * ASSERTION: Spawn 3 CPU threads and 1 Task thread, then joins the threads to complete execution.
 *            Appends final log data, containing average wait time and turnaround time.
 */
int main(int argc, char* argv[])
{
    if(argc != 3) //Check sufficient arguments
    {
        printf("Error: Insufficient Arguments\n");
        return -1; //Exit
    }
    else if(atoi(argv[2]) > 10 || atoi(argv[2]) < 1) //Check allowed queue size
    {
        printf("Incorrect Queue Size!\n");
        return -1; //Exit
    }

    int i, j, k; //For loops
    double avgWait, avgTAround; //Store real values of average calculations
    char logLine[256]; //Line to write to simulation_log

    pthread_t cpuTID[3]; //Thread IDs of CPU threads
    pthread_t taskTID; //Thread ID of Task thread

    strcpy(fileName, argv[1]);

    q = newQueue(atoi(argv[2])); //Initialize queue
    if(!getNumTasks())
    {
        printf("Error Opening File!\n");
        return -1;
    }

    for(i = 0; i < 3; i++) //Create 3 CPU threads
    {
        pthread_create(&cpuTID[i], NULL, cpu, NULL);
    }

    pthread_create(&taskTID, NULL, task, NULL); //Create 1 Task thread

    c1 = cpuTID[0]; //Store CPU IDs in global ints for getCPUID()
    c2 = cpuTID[1];
    c3 = cpuTID[2];

    pthread_join(taskTID, NULL); //Join task thread. (Wait for it to complete exec)
    printf("Task Thread Finished\n");

    pthread_cond_broadcast(&qElement);

    for(j = 0; j < 3; j++) //Join CPU threads. (Wait for it to complete exec)
    {
        pthread_join(cpuTID[j], NULL);
        printf("CPU Thread %d Finished\n", j+1);
    }

    avgWait = total_waiting_time/(double)num_tasks; //Calculate average waiting time
    avgTAround = total_turnaround_time/(double)num_tasks; //Calculate average turnaround time

    sprintf(logLine, "Number of tasks: %d\nAverage waiting time: %.2f seconds\nAverage turn around
time: %.2f seconds\n", num_tasks, avgWait, avgTAround);
    writeLog(logLine); //Write final statistics to simulation_log

    free(q->array);
    free(q);
}
```

```
/**
 * IMPORTS: N/A
 * EXPORTS: Void pointer.
 * ASSERTION: Producer/Writer component.
 *            Opens file, obtains lock on queue, then checks if queue is full - if full waits for
Consumer/Reader.
 *            to empty. Then enqueues 2 tasks at a time, writing arrival time to log. Then broadcasts
the condition
 *            qElement to all readers. Waits for reader to service task before continuing. Finishes
once all tasks
 *            from file added.
 */
void* task()
{
    char line[10];

    int taskN, cpuB, j;
    int totalQueued = 0; //Total elements enqueued (so far).

    time_t curTime; //Current time

    char timeStr[256]; //Current time represented as a string
    char termTime[256]; //Termination time represented as a string

    char logLine[256]; //Line 1 to write to simulation_log
    char logLine2[256]; //Line 2 to write to simulation log

    int finished = FALSE; //Variable modified by the writer to signal if writer has finished

    FILE* taskFile = fopen(fileName, "r");
    if(taskFile == NULL) //Check open file success
    {
        printf("Error opening file!\n");
        finished = TRUE;
        return (void*)-1;
    }

    while(!finished)
    {
        pthread_mutex_lock(&mutex); //Get lock on queue

        while(isFull(q)) //While the queue is full
        {
            pthread_cond_wait(&qEmpty, &mutex); //Wait for the readers to empty the queue (signal
empty)
        }

        if(totalQueued < tasksInFile) //If there are still lines to be read from the file
        {
            if(totalQueued == tasksInFile-val) //Condition to stop CPU threads once last task has
been added
            {
                lastTask = TRUE;
            }

            for(j = 0; j < 2; j++)
            {
                if(!(isFull(q)))
                {
                    if(fgets(line, 10, taskFile) != NULL) //Read from the file and add to queue
                    {
                        if(sscanf(line, "%d %d", &taskN, &cpuB) > 1)
                        {
                            time(&curTime); //Current time
                            strftime(timeStr, 256, "%H:%M:%S", localtime(&curTime)); //Format current
time

                            enqueue(q, taskN, cpuB); //Enqueue task

                            sprintf(logLine, "%d:%d\nArrival Time: %s", taskN, cpuB, timeStr);
//format line for log file
                            writeLog(logLine); //Write to logfile

                            totalQueued++;
                        }
                    }
                }
```

```
            }

            pthread_cond_broadcast(&qElement); //Signal to all readers that elements now exist in
queue
            pthread_cond_wait(&qServiced, &mutex); //Wait for readers to service the element/s -
Required to enqueue 2 task at a time
        }
        else
        {
            finished = TRUE; //Stop writing as entire file has been read.
        }
        pthread_mutex_unlock(&mutex); //Release lock on queue
    }

    time(&curTime); //Current time
    strftime(termTime, 256, "%H:%M:%S", localtime(&curTime)); //Format current time

    sprintf(logLine2, "Number of tasks put into Ready-Queue: %d\nTerminated at time: %s",
totalQueued, termTime);
    writeLog(logLine2); //Write total tasks added and termination time to log

    fclose(taskFile);
}

/**
 * IMPORTS: N/A
 * EXPORTS: Void pointer.
 * ASSERTION: Consumer/Reader component. Obtains lock on queue, checks if queue is empty, if it is
then waits
 *            for task() to signal condition qElement. Dequeues task and sleeps for its CPU burst.
Writing the
 *            arrival, service and completed times to log. Then signals condition qServiced.
 */
void* cpu()
{
    time_t curTime;
    int serviced = 0;

    char arrivTime[256]; //Arrival time as string
    char servTime[256]; //Served time as string
    char compTime[256]; //Completed time as string

    char logLine[256]; //Line 1 to write to simulation_log
    char logLine2[256]; //Line 2 to write to simulation_log
    char logLine3[256]; //Line 3 to write to simulation_log

    int cID; //CPU ID

    while(!lastTask) //Continue if last task has not been serviced
    {
        pthread_mutex_lock(&mutex); //Obtain lock on queue

        cID = getCPUID((int)pthread_self()); //Retrieve CPU ID based on thread ID

        while(isEmpty(q)) //If the queue is empty
        {
            pthread_cond_signal(&qEmpty); //Signal that it is empty
            pthread_cond_wait(&qElement, &mutex); //Wait for task() to signal that element exists.
        }

        tsk* t = dequeue(q); //Dequeue element

        num_tasks++; //Global task counter
        serviced++; //Individual thread task counter

        strftime(arrivTime, 256, "%H:%M:%S", localtime(&t->arrivalTime)); //Retrieve and convert
arrival Time from task struct

        time(&curTime); //Get current time
        strftime(servTime, 256, "%H:%M:%S", localtime(&curTime)); //Format current time

        sprintf(logLine, "Statistics for CPU-%d:\nTask %d\nArrival time: %s\nService time: %s", cID,
t->taskNum, arrivTime, servTime);
        writeLog(logLine); //Write service time stats

        total waiting time = total waiting time + difftime(curTime, t->arrivalTime); //Increment
total_waiting_time by wait time for task
```

```
        printf("T: %d C: %d\n", t->taskNum, t->cpuBurst);
        //sleep(t->cpuBurst); //Simulate cpuBurst with sleep

        time(&curTime); //Get current time again
        strftime(compTime, 256, "%H:%M:%S", localtime(&curTime)); //Format new current time
        sprintf(logLine2, "Statistics for CPU-%d:\nTask %d\nArrival Time: %s\nCompletion Time: %s",
cID, t->taskNum, arrivTime, compTime);
        writeLog(logLine2); //Write completion time stats

        total_turnaround_time = total_turnaround_time + difftime(curTime, t->arrivalTime);
//Increment total_turnaround_time by turnaround time for task

        pthread_cond_signal(&qServiced); //Signal that task has been serviced
        pthread_mutex_unlock(&mutex); //Release lock on queue
    }

    sprintf(logLine3, "CPU-%d terminates after servicing %d tasks", cID, serviced);
    writeLog(logLine3); //Write CPU stats on thread finish.
}

/**
 * IMPORTS: N/A
 * ASSERTION: Retrieves number of tasks from file.
 */
int getNumTasks()
{
    FILE* taskFile = fopen(fileName, "r");
    char line[10];
    //Count num tasks
    if(taskFile != NULL)
    {
        while(fgets(line, 10, taskFile) != NULL)
        {
            tasksInFile++; //Increase task count.
        }

        fclose(taskFile);
    }
    else
    {
        return 0;
    }

    if(tasksInFile % 2 == 0)
    {
        val = 4; //!Skips final value from even numbered file.
    }
    else
    {
        val = 3;
    }

    return 1;
    //Let writer exit on error.
}

/**
 * IMPORTS: Integer representing thread ID
 * EXPORTS: Integer representing CPU ID (1/2/3)
 * ASSERTION: Compares thread ID with c1,c2,c3 values in global to return CPU number that is
executing.
 */
int getCPUID(int TID)
{
    if(TID == c1)
    {
        return 1;
    }
    else if(TID == c2)
    {
        return 2;
    }
    else if(TID == c3)
    {
        return 3;
    }
```

```
}


/**
 * IMPORTS: Formatted string (sprintf) to write to simulation_log
 * ASSERTION: Opens simulation_log for appending, writes formatted log string and seperates each log
write with --
 */
void writeLog(char* logLine)
{
    FILE* logFile = NULL;
    logFile = fopen("simulation_log", "a");
    if(logFile != NULL)
    {
        fprintf(logFile, strcat(logLine, "\n--\n"));
        fclose(logFile);
    }
    else
    {
        printf("Error Opening Log File!\n");
    }
}
```

4.2 queue.c

```
#include "queue.h"

/*Creates queue that can hold ints. Initially empty*/

/**
 * IMPORTS: Integer representing queue capacity retrieved from "m" cmdline in scheduler.
 * ASSERTION: Creates a new queue of inCapacity size, mallocing an array of task structs to represent
queue.
 */
queue* newQueue(int inCapacity)
{
    queue* q = (queue*)malloc(sizeof(queue));

    q->capacity = inCapacity;
    q->size = 0;
    q->front = q->size;
    q->rear = inCapacity - 1;

    q->array = (tsk*)malloc(q->capacity * sizeof(tsk));
}

/**
 * IMPORTS: Queue
 * ASSERTION: Determines if queue is full
 */
int isFull(queue* q)
{
    return(q->size == q->capacity);
}

/**
 * IMPORTS: Queue
 * ASSERTION: Determines if queue is empty
 */
int isEmpty(queue* q)
{
    return(q->size == 0);
}

/**
 * IMPORTS: Queue, Integer task number from file, Integer cpu burst from file
 * ASSERTION: Handles creation of the task struct on the stack, calculating its arrival time
 *            (time it is inserted) storing the time into the task struct. Then inserting into queue.
 */
void enqueue(queue* q, int inTaskNum, int inCpuBurst)
{
    if(isFull(q))
    {
```

```
        printf("Queue Full!\n");
    }
    else
    {
        tsk t; //Create on stack

        t.taskNum = inTaskNum;
        t.cpuBurst = inCpuBurst;
        t.arrivalTime = time(NULL); //Assign time_t value to arrivalTime field in tsk struct

        q->rear = (q->rear + 1) % q->capacity;
        q->array[q->rear] = t;
        q->size = q->size + 1;
    }
}

/**
 * IMPORTS: Queue
 * ASSERTION: Removes item from queue (FIFO Order)
 */
tsk* dequeue(queue* q)
{
    if(isEmpty(q))
    {
        printf("Queue Empty!\n");
        return NULL;
    }
    else
    {

        tsk* t = &q->array[q->front];
        q->front = (q->front + 1) % q->capacity;
        q->size = q->size - 1;

        return t;
    }
}

/**
 * IMPORTS: Queue
 * ASSERTION: Returns value at front of queue
 */
tsk* front(queue* q)
{
    if(isEmpty(q))
    {
        printf("Empty!");
    }
    else
    {
        return &q->array[q->front];
    }

}

/**
 * IMPORTS: Queue
 * ASSERTION: Returns value at rear of queue
 */
tsk* rear(queue* q)
{
    if(isEmpty(q))
    {
        printf("Empty!");
    }
    else
    {
        return &q->array[q->rear];
    }

}
```

4.3 scheduler.h

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

```
#include <pthread.h>

#define TRUE 1
#define FALSE 0

//Values required per assignment spec. Used to calculate averages.
int num_tasks = 0;
double total_waiting_time = 0;
double total_turnaround_time = 0;

char fileName[15]; //Stores filename specified in commandline

int tasksInFile = 0; //Number of tasks in the file, modified by the getNumTasks()
int val = 0; //When to finish CPU thread.
int c1,c2,c3; //Stores CPU IDs

int lastTask = FALSE; //Variable modified by the writer to signal if writer has enqueued the last
task in the file

pthread_cond_t qServiced = PTHREAD_COND_INITIALIZER; //Pthread condition to signal/ wait on if the
task has been serviced
pthread_cond_t qEmpty = PTHREAD_COND_INITIALIZER; //Pthread condition to signal/wait if queue is
empty
pthread_cond_t qElement = PTHREAD_COND_INITIALIZER; //Pthread condition to signal if

pthread mutex t mutex = PTHREAD MUTEX INITIALIZER; //Pthread condition to obtain a lock on the queue

void* task();
void* cpu();

int getNumTasks();
int getCPUID(int TID);
void writeLog(char* line);
```

4.4 queue.h

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

typedef struct tsk
{
    int taskNum;
    int cpuBurst;

    time_t arrivalTime;
} tsk;

typedef struct queue
{
    int front, rear, size;
    unsigned capacity;
    tsk* array;
} queue;

queue* newQueue(int inCapacity);

void enqueue(queue* q, int inTaskNum, int inCpuBurst);
tsk* dequeue(queue* q);

int isFull(queue* q);
int isEmpty(queue* q);

tsk* front(queue* q);
tsk* rear(queue* q);
```

5. References

Queue implementation obtained from: GeeksforGeeks at "Queue | Set 1 (Introduction and Array Implementation)". Retrieved from: https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/ Accessed 21/04/19.

Time implementation based on: Tutorialspoint at “C library function - difftime()” Retrieved from: https://www.tutorialspoint.com/c_standard_library/c_function_difftime.htm Accessed 20/04/19.

Error fix for readers synchronization based on “user3629249”s answer to: “Readers-Writers problem with pthreads resulting in only 1 reader”. https://stackoverflow.com/questions/55759068/readers-writers-problem-with-pthreads-resulting-in-only-1-reader Accessed 20/04/19.

Broadcast usage in task() based on: Brown University “Programming with POSIX Threads I” – CS168 Fall 2018. Retrieved from http://cs.brown.edu/courses/cs168/f18/content/threads1.pdf Accessed 19/04/19.