|  |  |
| --- | --- |
| OPERATING SYSTEMS ASSIGNMENTAssignment  Semester 1 2019 | Declaration of Originality  I Adam Camer-Pesci certify that all content in this assignment is the result of my own research and work. Any resources or previously used code has been acknowledged in the source file header/relevant locations.  Adam Camer-Pesci  SID: 19313425  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Table of Contents

[Overview 2](#_Toc7858295)

[1.0 Variables 3](#_Toc7858296)

[2.0 Threads 4](#_Toc7858297)

[3.0 Testing and Functionality 5](#_Toc7858298)

[Software Solution 6](#_Toc7858299)

[References 6](#_Toc7858300)

[Figures 6](#_Toc7858301)

[FIG A 6](#_Toc7858302)

[FIG B 7](#_Toc7858303)

[FIG C 7](#_Toc7858304)

# Overview

This report will discuss in detail each of the global variables and their data structures (where applicable). The threads that access them and Lastly how mutual exclusion was achieved. A breakdown of how the program works will also be included. Sample test data will be examined to prove the functionality of the scheduler and that the correct outputs are achieved. (All figures will be listed at the end of the report).

All relevant code and sample output used and produced for this assignment is outlined in the software solution section of this report.

# Variables

Excluding the pthread\_mutex\_t and pthread\_cond\_t variables for now, there are five global variables shared by the threads. We will discuss the mutex and condition variables shortly. The variables that are used by all threads are listed below:

* Queue\_t \***queue**: This is the queue struct which serves as a ready queue. All code can be found in the ready\_queue.c source file. The queue is implemented using a linked list, it also contains a (int)maxSize and a (int)numElem variable which identify the maximum size of the queue and the current number of elements in the queue respectively.   
    
  ready\_queue.c mostly contains renamed methods of those found in linked\_list.c however it does do some other actions these methods are called as certain functions need to alter maxSize (i.e. makeQueue). While others need to alter numElem (i.e. enqueue() and dequeue()).

(Please see documentation found in ready\_queue.c/ready\_queue.h for more details).

* int **num\_tasks**: This is the collective total of completed tasks. Each time a CPU completes a task this data is updated.
* int **total\_waiting\_time**: This is the total waiting time of all tasks; this data is used to calculate the average waiting time of all processes.
* int **total\_turnaround\_time**: This is the total turn around time of all tasks, this data is used to calculate the average total turnaround time of all processes.
* int **is\_done**: This is an integer control flag which is used when there is no longer anymore tasks to be read into the queue from the task\_file.

These variables excluding ‘is\_done’ are protected by the mutex lock as they are all changed by each CPU at some point. The reason I have omitted ‘is\_done’ from the list of variables that require synchronization is because only the task thread alters the value of ‘is\_done’; the CPU threads need only read the data and as race condition will never occur.

As mentioned earlier, the pthread\_mutex\_t and pthread\_cond\_t variables are also global variables used by the task and CPU threads. They are described as follows:

* pthread\_mutex\_t **m\_lock**: This is the mutex lock used by all threads whenever a race condition is encountered. It is used to synchronize data that would otherwise be incorrect if multiple threads accessed it at once.
* pthread\_cond\_t **more**: This is a conditional variable used as a synchronization mechanism that allows threads to temporarily suspend execution until some predicate on shared data is satisfied. ‘more’ is used to signal the CPU threads to let them know that there are tasks in the ready queue waiting to be serviced.
* pthread\_cond\_t **less**: This is the same as above however it is used to signal the task thread to let it know that there is room available for more tasks in the queue.

# Threads

There are two implemented thread functions:

* void \*task(void \*arg) the function used to call the task() thread function.
* void \*cpu(void \*arg): The function used to call the cpu() thread function.

Task() and CPU() which adhere to the requirements of the assignment specification are discussed in more detail below. Mutual exclusion is achieved through the use of calls to pthread\_mutex\_lock(), pthread\_cond\_signal() as well as pthread\_cond\_broadcast().

* task(): This is the first thread to be created. Task() requires one parameter, (void \*arg) which is the name of the task\_file. Once the thread been created, its local variables initialized and the first line of the task\_file has been read, it is important a few checks are conducted.

Firstly, task() will check if the queue is not empty, this is because if the queue is not empty, all cpu threads need to be notified so they can start processing tasks in the queue.

Secondly, once task() has made sure there are tasks in the queue, it will check to see if the queue is full. As this is the first condition task() must wait, the mutex lock is acquired and the function blacks for the duration of the condition. Once one of the cpu threads remove tasks from the queue, task() will be notified to resume. (TBD shortly).

If task() is not required to block it will proceed to add tasks from the task\_file to the queue and write to the simulation log. All of this is done while task() has the lock. After each addition to the queue the lock is released so that another thread can access the queue while the race condition is over. Mutual exclusion is achieved as whenever we are accessing the shared resources (in this case the queue and even the access to the simulation\_log) we are stopping any other thread from accessing it.

* cpu(): This is the function used to create the consequent threads. cpu() also requires one parameter, (void \*arg) which in this instance, is an integer used to identify the cpu thread, (1,2 or 3). Once cpu()’s local variables have been initialized checks are conducted.

Firstly, the mutex lock is acquired and cpu() checks if the queue is empty, if so, it will immediately block for the duration that the queue is empty. This ensures that current cpu() will never access the queue whilst there are no tasks available to be processed. Allowing task() the opportunity to fill the queue without interruption. Once this check has been performed the lock is then released as due to the structure of the code the lock must be acquired within the while loop responsible for dequeuing tasks.

Once cpu() can proceed it will re-acquire the lock and get the first task from the queue, the memory for the task is copied to a new location and the task original task is dequeued. The simulation\_log is then appended, and the lock released. This copying is done so the lock may be released as early as possible and the threads that require the lock can continue operation while the current cpu thread is sleeping for the burst time of the task. It should be noted that before the cpu proceeds to sleep, a quick check to see if the queue has room available for more tasks. If there is room available and task() has not changed the control flag **is\_done** from 0 (false), the cpu thread will signal the task thread to resume its operation. If **is\_done** is set to 1 (true) then cpu() will not need to signal task() as the thread would have been terminated.

Finally, once a cpu thread has finished its sleep period, the lock is then acquired to exclude other threads from accessing the shared resources; “**num\_tasks, total\_waiting\_time** and **total\_turnaround\_time**”. The values of these variables require the lock as previously mentioned as they are shared among all threads and if not synchronized will produce incorrect results. Once the shared variables have been updated the lock is then released and the cpu will repeat this until there are no more tasks to be processed. Cpu() then terminates when there are no more tasks in the queue.

To summarize, we can see that every time a shared variable is accessed a lock is acquired before any change is made to the variable/data. We can also confirm that every time the simulation\_log is altered, it is only altered by the thread who has the lock. Mutual exclusion is achieved as no shared resource is ever accessed without the possession of the mutex lock.

# Testing and Functionality

For testing purposes, the version **scheduler\_debug** has been used. This version of the program is nearly identical to **scheduler**, the only difference is that it produces some output whenever a key action is performed. This debugging version will output when the following actions occur:

* Threads are created.
* Threads are joined back to main/calling method.
* Tasks are being enqueued by task(). (it will also note how many are being added).
* task() has possession of the mutex lock.
* task() signals CPU threads.
* task() blocks/waits.
* task() resumes/wakes up.
* task() completes and terminates.
* cpu()# processes a task.
* cpu()# signals tasks when there is more room in the queue.
* cpu()# blocks/waits.
* cpu()# completes all actions and terminates.

To ensure the output data was correct, a manual check of the Dtask\_file output was made (**See FIG A**). I manually calculated the waiting time and turnaround time based off the arrival, service and completion times that were written to the Dsim\_log (simulation\_log for scheduler\_debug) and was able to conclude that the output data was correct.

The following figure (**FIG C**) is the result of running “**scheduler\_debug Dtask\_file 4**” which allows us to run a brief simulation with a max queue size of four and set of ten tasks which all have burst times of no more than fifteen seconds. We can see quite clearly from the output what actions are happening. After examining the output, notice that if a cpu has no tasks to process, it will wait until task() has signalled before attempting to continue, Similarly task() will wait until it has been signalled before attempting to add any more tasks to the queue once the queue has been filled. Additionally, no action is ever taken when accessing shared data without the possession of a mutex lock.

(**See FIG C**) for all actions taken by each thread.

Finally, Valgrind is used to check for any memory errors or leaks. (**See FIG B**). After executing the command “**time valgrind ./scheduler task\_file 10**” This will use the main scheduler program with a task\_file with one hundred tasks allowing us to make sure that under the required load no memory leaks occur.

Using the time command, we can also ensure that the expected run time of scheduler is achieved. (we can confirm this by adding all burst times and dividing by number of processors).

# Software Solution

I have included a print out of all the code used for my assignment. All relevant .h files have been attached with their corresponding .c files. All files have been ordered in order of execution. A copy of the task\_file and simulation\_log that was used for (**FIG A**) and (**FIG C**) has also been included as sample input/output for scheduler.

# References

Linked\_list.c/h:   
 Submitted for Curtin College UCP assignment Sem 2 2018  
 Author: Adam Camer-Pesci.

# Figures

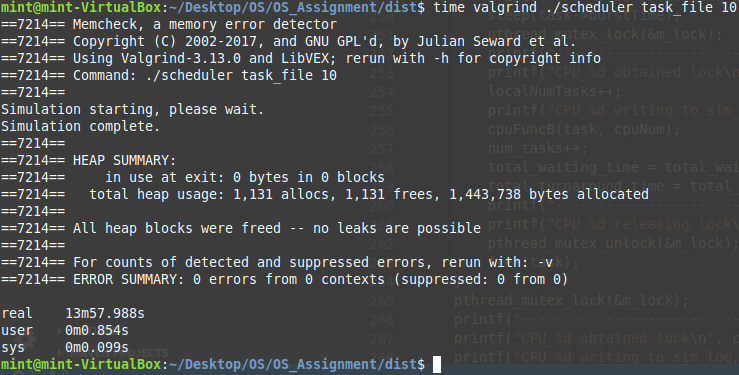
Below are the figures referenced during Testing and functionality.

## FIG A

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **task** | **burst time** | **arrival time** | **service time** | **wait time** | **comp time** | **turnaround time** | **average wait time** | **average turnaround time** |
| 1 | 7 | 0 | 0 | 0 | 7 | 7 | 5.7 | 13.2 |
| 2 | 5 | 0 | 0 | 0 | 5 | 5 |  |  |
| 3 | 9 | 0 | 0 | 0 | 9 | 9 |  |  |
| 4 | 5 | 0 | 5 | 5 | 10 | 10 |  |  |
| 5 | 2 | 0 | 7 | 7 | 9 | 9 |  |  |
| 6 | 7 | 0 | 9 | 9 | 16 | 16 |  |  |
| 7 | 15 | 0 | 9 | 9 | 24 | 24 |  |  |
| 8 | 12 | 5 | 10 | 5 | 22 | 17 |  |  |
| 9 | 7 | 7 | 16 | 9 | 23 | 16 |  |  |
| 10 | 6 | 9 | 22 | 13 | 28 | 19 |  |  |

**Figure A**: Table of process times for Dtask\_file and Dsim\_log

## FIG B

**Figure B**: Valgrind check of scheduler program

## FIG C

mint@mint-VirtualBox:~/Desktop/OS/OS\_Assignment/dist$ time ./scheduler\_debug Dtask\_file 4

Simulation starting, please wait.

------------------------------------------------------------------

creating task thread

creating CPU\_1 thread

creating CPU\_2 thread

creating CPU\_3 thread

------------------------------------------------------------------

CPU\_1 obtained lock : checking if needing to block

------------------------------------------------------------------

CPU\_1 waiting, lock released

------------------------------------------------------------------

CPU\_2 obtained lock : checking if needing to block

------------------------------------------------------------------

CPU\_2 waiting, lock released

------------------------------------------------------------------

task() obtained lock

task() enqueuing 1/2 tasks

task() writing to sim\_log arrival of task 1/2

task() enqueuing 2/2 tasks

task() writing to sim\_log arrival of task 2/2

task() releasing lock

Signalling CPU's

------------------------------------------------------------------

task() obtained lock

task() enqueuing 1/2 tasks

task() writing to sim\_log arrival of task 1/2

task() enqueuing 2/2 tasks

task() writing to sim\_log arrival of task 2/2

task() releasing lock

Signalling CPU's

------------------------------------------------------------------

task() obtained lock

------------------------------------------------------------------

queue is full, task() waiting, locked released

------------------------------------------------------------------

CPU\_3 obtained lock : checking if needing to block

------------------------------------------------------------------

CPU\_3 released lock : check complete

------------------------------------------------------------------

CPU\_1 received signal from task() : re-acquired lock, resuming

------------------------------------------------------------------

CPU\_1 released lock : check complete

------------------------------------------------------------------

CPU\_2 received signal from task() : re-acquired lock, resuming

------------------------------------------------------------------

CPU\_2 released lock : check complete

------------------------------------------------------------------

CPU\_2 obtained lock

CPU\_2 writing to sim\_log service of task

CPU\_2 releasing lock

there is more room in the queue, signalling task()

CPU\_2 is processing a task (sleeping)

------------------------------------------------------------------

task() received signal from cpu, resuming

task() enqueuing 1 task

task() writing to sim\_log arrival of task

task() releasing lock

Signalling CPU's

------------------------------------------------------------------

task() obtained lock

------------------------------------------------------------------

queue is full, task() waiting, locked released

------------------------------------------------------------------

CPU\_3 obtained lock

CPU\_3 writing to sim\_log service of task

CPU\_3 releasing lock

there is more room in the queue, signalling task()

------------------------------------------------------------------

CPU\_1 obtained lock

CPU\_1 writing to sim\_log service of task

CPU\_3 is processing a task (sleeping)

CPU\_1 releasing lock

there is more room in the queue, signalling task()

CPU\_1 is processing a task (sleeping)

------------------------------------------------------------------

task() received signal from cpu, resuming

task() enqueuing 1/2 tasks

task() writing to sim\_log arrival of task 1/2

task() enqueuing 2/2 tasks

task() writing to sim\_log arrival of task 2/2

task() releasing lock

Signalling CPU's

------------------------------------------------------------------

task() obtained lock

------------------------------------------------------------------

queue is full, task() waiting, locked released

------------------------------------------------------------------

CPU\_3 obtained lock

CPU\_3 writing to sim\_log completion of task

------------------------------------------------------------------

CPU\_3 releasing lock

------------------------------------------------------------------

CPU\_3 obtained lock

CPU\_3 writing to sim\_log service of task

CPU\_3 releasing lock

there is more room in the queue, signalling task()

------------------------------------------------------------------

task() received signal from cpu, resuming

task() enqueuing 1 task

task() writing to sim\_log arrival of task

task() releasing lock

Signalling CPU's

------------------------------------------------------------------

task() obtained lock

------------------------------------------------------------------

queue is full, task() waiting, locked released

CPU\_3 is processing a task (sleeping)

------------------------------------------------------------------

CPU\_2 obtained lock

CPU\_2 writing to sim\_log completion of task

------------------------------------------------------------------

CPU\_2 releasing lock

------------------------------------------------------------------

CPU\_2 obtained lock

CPU\_2 writing to sim\_log service of task

CPU\_2 releasing lock

there is more room in the queue, signalling task()

CPU\_2 is processing a task (sleeping)

------------------------------------------------------------------

task() received signal from cpu, resuming

task() enqueuing 1 task

task() writing to sim\_log arrival of task

task() releasing lock

Signalling CPU's

------------------------------------------------------------------

task() obtained lock

------------------------------------------------------------------

queue is full, task() waiting, locked released

------------------------------------------------------------------

CPU\_1 obtained lock

CPU\_1 writing to sim\_log completion of task

------------------------------------------------------------------

CPU\_1 releasing lock

------------------------------------------------------------------

CPU\_1 obtained lock

CPU\_1 writing to sim\_log service of task

CPU\_1 releasing lock

there is more room in the queue, signalling task()

CPU\_1 is processing a task (sleeping)

------------------------------------------------------------------

task() received signal from cpu, resuming

task() enqueuing 1 task

task() writing to sim\_log arrival of task

task() releasing lock

------------------------------------------------------------------

task() COMPLETE, NO MORE TASKS IN FILE

task() local counter = 10

task() obtained lock, writing task complete to sim\_log

task() released lock

------------------------------------------------------------------

task exiting

------------------------------------------------------------------

CPU\_2 obtained lock

CPU\_2 writing to sim\_log completion of task

------------------------------------------------------------------

CPU\_2 releasing lock

------------------------------------------------------------------

CPU\_2 obtained lock

CPU\_2 writing to sim\_log service of task

CPU\_2 releasing lock

CPU\_2 is processing a task (sleeping)

------------------------------------------------------------------

CPU\_3 obtained lock

CPU\_3 writing to sim\_log completion of task

------------------------------------------------------------------

CPU\_3 releasing lock

------------------------------------------------------------------

CPU\_3 obtained lock

CPU\_3 writing to sim\_log service of task

CPU\_3 releasing lock

CPU\_3 is processing a task (sleeping)

------------------------------------------------------------------

CPU\_1 obtained lock

CPU\_1 writing to sim\_log completion of task

------------------------------------------------------------------

CPU\_1 releasing lock

------------------------------------------------------------------

CPU\_1 obtained lock

CPU\_1 writing to sim\_log service of task

CPU\_1 releasing lock

CPU\_1 is processing a task (sleeping)

------------------------------------------------------------------

CPU\_3 obtained lock

CPU\_3 writing to sim\_log completion of task

------------------------------------------------------------------

CPU\_3 releasing lock

------------------------------------------------------------------

CPU\_3 obtained lock

CPU\_3 writing to sim\_log service of task

CPU\_3 releasing lock

CPU\_3 is processing a task (sleeping)

------------------------------------------------------------------

CPU\_1 obtained lock

CPU\_1 writing to sim\_log completion of task

------------------------------------------------------------------

CPU\_1 releasing lock

------------------------------------------------------------------

CPU\_1 obtained lock

CPU\_1 writing to sim\_log,complete all tasks

CPU 1 exiting

CPU\_1 releasing lock

------------------------------------------------------------------

CPU\_2 obtained lock

CPU\_2 writing to sim\_log completion of task

------------------------------------------------------------------

CPU\_2 releasing lock

------------------------------------------------------------------

CPU\_2 obtained lock

CPU\_2 writing to sim\_log,complete all tasks

CPU 2 exiting

CPU\_2 releasing lock

------------------------------------------------------------------

CPU\_3 obtained lock

CPU\_3 writing to sim\_log completion of task

------------------------------------------------------------------

CPU\_3 releasing lock

------------------------------------------------------------------

CPU\_3 obtained lock

CPU\_3 writing to sim\_log,complete all tasks

CPU 3 exiting

CPU\_3 releasing lock

------------------------------------------------------------------

All threads joined to main

------------------------------------------------------------------

Scheduler complete

Simulation complete.

real 0m28.009s

user 0m0.004s

sys 0m0.000s  
  
**Figure C**: A complete log of scheduler\_debug