Operating Systems 200 Assignment

Peskett, Timothy Darryl
Student No.:16243969

Submission Date 19/5/2014

Contents

1. Assumptions Made

The assumptions made to ensure correct operation of the program are as follows:

- It is assumed that the file name entered by the user will be 10 characters or less. A file name longer than this is NOT handled gracefully because it would make the code more tightly coupled. Adding a constraint to scanf would require hard-coding the value of 10 in two places, making it harder to change the limit later on. Using a different method that does not require hard-coding twice is possible but would complicate the code too much for such a small benefit. It stands to reason that the user is well aware of this limit on the number of characters.
- It is assumed that there is a time quantum as the first line of the file in all situations, even if running the shortest job first algorithm. This time quantum is then ignored if not applicable. The specification was somewhat unclear on whether the time quantum would be present or not if it were not required.
- It is assumed that the arrival times and burst times are positive integers. Furthermore it is assumed that these integers fall between 0 and MAX_INT. This condition is required for the arrival and burst times to fit into C's int variable.
- It is assumed that newly arrived processes are to be added to the ready queue before a process whose time quantum has expired if these occur in the same tick.
- It is assumed that the turnaround and waiting times for a file with no processes should be 0. This could have been handled in any way so I decided to handle it in a way that would keep the interfaces to functions more flexible and the code slightly cleaner.

2. Mutual Exclusion

Proper use of mutual exclusion to ensure a good solution to the critical section problem was instrumental in the correct operation of part three of the assignment. There two critical sections in each thread, these are:

- When reading/writing the name of the file to buffer1.
- When reading/writing the average waiting time and the average turnaround time to buffer2.

In both cases it is important that only one thread is in its critical section at a time (hence why it is a critical section). If there were no critical sections then the code would be filled with race conditions.

2.1 Solution to the File Name Reading/Writing Problem

Solution to the file name reading/writing problem was achieved using one mutex and two condition variables. The mutex controls access to buffer1 (the buffer for the file name) and to state variables fNameRead[0..1]. The conditions signal whenever a write to fNameRead[0..1] occurs. Given this, this strategy for **writing** the file name to the buffer is as follows:

1. Lock the mutex so that we can read fNameRead[0..1] can be safely read without race conditions.
2. If fNameRead[0] and fNameRead[1] are **both** true then the file name already buffered has already been read by the worker threads. We can immediately proceed and write the file name in to buffer1 and proceed to step 5.

3. If fNameRead[0] and fNameRead[1] are **both not** true then at least one of our worker threads still has to read the file name. We now wait on one of our two conditions depending on which thread(s) we are waiting on. When the thread that we are waiting on finally reads the buffer it will signal the condition and we can wake up and continue.
4. When our thread finally wakes up we know that both threads have read the buffer and so we are free to write the new file name in to buffer1.
5. Finally we must set the state variables for fNameRead[0...1] to false so that the threads know they have not yet read this file name. We then signal the two condition variables so that our working threads who are waiting to read the file name can wake up and begin processing.

It is important to note here that when a condition variable is being waited on, the mutex is temporarily relinquished. This is what allows us to achieve synchronisation.

The strategy for **reading** the file name, employed by both worker threads, is as follows. Note that we take the viewpoint of a single worker thread, the other worker thread works symmetrically:

1. Lock the mutex so that we can read fNameRead[0] (as applicable) without race conditions.
2. If fNameRead[0] is **false** then we have no yet read the new file name. We can proceed directly to step 5.
3. If fNameRead[0] is **true** then we have already read this file name. We must wait on the condition variable for the thread. This will relinquish the mutex so that the main thread can write the file name and then signal us to wake back up.
4. When we wake up from waiting on our condition we know that the buffer has been written so we are free to read the buffer.
5. We now read from buffer1 and set fNameRead[0] and signal the thread's condition to tell the main thread that we have finished reading the file name from buffer1.

This use of state variables, conditions and mutexes ensures that the critical section is solved. A quick analysis of the criteria for a solution to the critical section problem follows:

Mutual Exclusion: A mutex is required before any reading or writing to fNameRead[0..1] or buffer1 occurs. This ensures that only one thread can be in its critical section at a time.

Progress: The only way a thread can loop inside its critical section is if fNameRead[0...1] is not true/false (however is required). Fortunately, this loop invokes pthread_cond_wait which relinquishes the lock on the mutex and allows other threads to progress.

Bounded waiting: The use of our two condition variables and our state variable here ensure that the bounded waiting criteria is met. Our state variable switching from true to false means that our main and worker threads have to take turns. The main thread can never enter its critical section twice before both worker threads have entered their critical sections once. Similarly in the other direction, each worker thread must wait for the main thread to set fNameRead[x] to false before they can re-enter their critical section.

2.2 Solution to the Results Reading/Writing Problem

A solution to the results writing/reading problem is achieved using one mutex, one condition variable, and one state variable. The state variable is simply a number used to indicate the worker thread that has currently written their results to buffer2. The strategy for **reading** from buffer2 is as follows:

1. A bounded loop (here of size 2) is entered to ensure that we read the right number of sets ofr results from the buffer.
2. We acquire a lock on our writing mutex so that we can safely access our state and buffer variables.
3. If our state variable is **not NONE** then a worker thread has written its results to buffer2 and we are able to read them. We can proceed directly to step 6.
4. If our state variable is **NONE** then no worker thread has yet written its results. We now wait on our condition variable, relinquishing the lock we have on the mutex and allowing a worker thread to run.
5. Once a worker thread has run far enough it will signal us to wake us up.
6. We now have data in our buffer2 and a lock on the mutex, it is time to safely read the data in buffer2.
7. We now set our state variable back to **NONE,** release the mutex and signal on our condition variable so that whichever worker thread has **not** yet written its data is now able to.
8. We repeat the loop to read the other set of data if necessary.

The strategy for **writing** to buffer 2 follows. Note that this strategy is the exact same in both worker threads:

1. A lock on the writing mutex is requested so that we can read and write our state and buffer variables.
2. If the state variable is **NONE** then we are able to write our data. We can proceed straight to step 5.
3. If the state variable is **not NONE** then we must wait on the condition variable. The mutex will be relinquished and the main thread will have a chance to read the data currently in the buffer. The main thread will then signal us to wake back up.
4. A signal is received on our condition variable to wake the thread back up.
5. The state variable is now **NONE** and we can write the results in to buffer2 and set the state variable appropriately.
6. We now must release the mutex and **broadcast** on the condition variable. The reason that a broadcast is necessary is because we are only using one condition variable. We must make sure to wake up both threads, otherwise the wrong thread could get woken up.

This satisfies a solution to the critical section problem. An analysis of the criteria for a solution to the critical section problem is as follows:

Mutual Exclusion: Every read/write to a shared variable is preceded by a lock on the writing mutex. This ensures that no two threads can read or write a shared variable at the same time.

Progress: Similar to the file name reading/writing problem, the only way a thread can stay in its critical section is if it is looping. Fortunately again, the only time we loop inside a critical section is to wait on a condition variable. Waiting on this condition variable will wake up any other waiting threads and allow them to progress into and out of their critical sections.

Bounded Waiting: Similar to the file name reading/writing problem, this is controlled through the state variable and condition variables. The state variable changing to identify the required thread means that the threads will definitely take turns to execute. Each thread changes the state variable to a value that only one other different thread can wake up on.

This solution seems to satisfy the three criteria needed for a solution to the critical section problem.

3. Program Testing

3.1 Synchronisation Testing

Synchronisation testing consisted of ensuring that there were no race conditions or deadlocks in my synchronisation code. When performing this sort of testing, I was not concerned with actually getting the correct answers for the given inputs, I was more concerned with making sure that my program would always run to completion. Unfortunately threading bugs are very often timing based. That is to say that they may only occur when a specific order of execution happens.

To try and test my program for these threading bugs I simply ran it over and over again until it would run for thousands of iterations without any problems. I created a simple file filled with 10,000 file names (some valid and some invalid) and supplied that as input to my program. This approach assisted me in catching a bug related to the use of pthread_cond_broadcast(...) that would only occur once in every few thousand simulations.

This approach to testing is by no means foolproof, but coupled with the analysis I have provided above I believe it provides strong evidence for the correctness of the synchronisation aspect of my program.

3.2 Simulation Testing

Simulation testing consisted of ensuring that the results that I received from my individual simulations were correct.

This was achieved by unit testing important modules of the program and by testing/debugging on different inputs. Specifically, the linked list implementation, the gantt chart implementation and the readProc function were extensively unit tested to ensure that they were more or less foolproof. Not doing this unit testing would have made it much harder to catch these bugs later on.

To check using input files I created several input files and calculated the average waiting times and average turnaround times by hand. By comparing these I could find out whether my program was operating correctly.

Finally, running my finished program through valgrind ensured that there were no memory leaks present.

4. Sample Inputs/Outputs

For a sample input I found it prudent to use that which was in the assignment specification. A listing of inputs and outputs for the separate parts of the assignment are given in this section.

4.1 Part 1

Input File: tdata2

```
5
1    24
1    23
3    1
4    20
4    100
12   30
20   20
```

Program Output:

```
>>RR simulation:badname
>>Can not open file: badname
>>RR simulation:tdata2
>>Average Waiting Time: 83.285714
>>Average Turnaround Time: 114.428571
>>RR simulation:QUIT
```

4.2 Part 2

Input File: tdata2

5	
1	24
1	23
3	1
4	20
4	100
12	30
20	20

Program Output:

```
>>SJF simulation:badname
>>Can not open file: badname
>>SJF simulation:tdata2
>>Average Turnaround: 77.285714
>>Average Waiting: 46.142857
>>SJF simulation:QUIT
```

4.3 Part 3

Input File: tdata2

5	
1	24
1	23
3	1
4	20
4	100
12	30
20	20

Program Output:

```
>>Scheduling simulation:badname
>>RR: Error occurred reading file!
>>SJF: Error occurred reading file!
>>Scheduling simulation:tdata2
>>SJF: Average Turnaround time = 77.285714, Average Waiting Time =
46.142857
>>RR: Average Turnaround time = 114.428571, Average Waiting Time =
83.285714
>>Scheduling simulation:QUIT
```