OS200 CPU Scheduler Assignment

Jason Giancono
16065985

Contents

1 Introduction

Concurrent programs with correct synchronization are very conceptually difficult to create and hard to follow the logic because of their non-deterministic nature. I have made this report in the so that my thought process in creating the CPU scheduler can be understood.

2 Program Structure

My program is split into two files. The main file which has the cpu, io and main functions in it is called scheduler.c and the other file is a library I created for manipulating linked lists called linked_list.c. The main functions reads all the input files, loads them into the data structures, creates the two threads for io and cpu then waits for them to join. Because main is never running concurrently with any other thread, we don't need to worry about race conditions when dealing with the global variables.

2.1 Important Variables

The global variables in my program can be split into four catagories and this affects how I deal with mutual exclusion with them. There are the mutex and condition variables for synchronization, variables which can be edited and viewed by both thread, variables which can only be changed by one thread but read by both and variables which are only read by one thread and the main function after the threads execute.

2.1.1 Mutex and Condition Variables

There are two Mutex variables and two Condition Variables. One pair is for controlling synchronization when editing the queue count variable and the other is used when synchronizing the time value between threads, I will talk more about their uses in the other sections.

2.1.2 Globally Editable Variables

There are two variables which can be edited by both io and cpu threads. These are the cpuCount and ioCount variables which display the amount of processes in their respective queues. They need to be edited by both threads because both threads will add processes to the others queue (which means they need to increment plus one) and they both need to be able to read the others count to prevent a deadlock when all processes are finished. Because they are globally editable, the threads need to aquire the count mutex lock before editng or reading the count values. This is to avoid deadlocks. Whenever the count is incremented, a signal is sent with the count condition to 'wake up' the other thread incase their queue was empty.

There is also an 'processes' array. The threads don't need a lock to grab/remove things in the lists in this array because a process will only ever be in one of the threads queues at once, which means there will never be a point where both threads are accessing the same element of the array.

2.1.3 Globally Readable Variables

There are two variables that are globally readable, the ioTime and cpuTime variables. With these variables their 'owner' threads can edit them but all other threads will only read them. This means that the owner thread needs to lock the time mutex when changing the time variable but not when just reading it. This means that both threads can read the time without mutual exclusiveness being a problem because only one thread can change the value. The other thread (who is not the owner) must get the time lock before it can read the variable. Whenever the owner thread updates the time variable, it also sends a signal using the time condition. This signal will tell the other thread to check it's time against the other threads time to see who is 'in front'. Threads will not write to the log until the other thread is 'in front' of them (with respects to time). This needs to happen because there is no real clock in this simulation, so all the 'processing' is done almost instantly regardless of what time it is supposed to take. Without this waiting, the logfile would not be written in time-cronological order.

I originally wanted to have a third thread which was supposed to be a clock which kept the two threads synchronized but that proved very difficult to do while keeping to the assignment specification.

2.1.4 Exclusive Variables

There are also some variables only used exclusively by each thread, namely the cpuBusy, ioBusy, cpuWait, ioWait variables. The Busy values keep a running total of all the time the thread has been 'executing' processes and the Wait variables keep a running total of how long each process waited in the queue. These values are used by the main function at the end to calculate the average waiting time and the utilization of each thread.

In regards to the IO utilization, the assignment specification example showed that the total running time of the IO did not include the last CPU burst, so that is how I programmed it in my program.

2.2 Flow of Execution

The program starts with just a single thread (main) while it reads the input files and sets up all the data structures, it then creates the cpu thread and the io thread. The IO thread will go into its wait section because none of the programs start off requiring IO. The CPU will execute the first item in the queue, write it to a file then grab the next job and then put the PA in the IO queue, then increase the ioCount and signal using the count condition. This will wake up the IO Queue. The IO thread and CPU thread will continue doing the PA jobs. When a queue is empty, the thread will wait for the signal from the other thread. When a PA has no more jobs, it is removed from the queue and freed. Keeping the threads synchronized was quite difficult and will be gone into in the next section.

3 Race Conditions

In my program, a thread enters a critical section whenever

- Any thread reads or changes a count variable
- A thread reads the other threads time
- A thread changes it's own time

The reason these are critical sections is because when you share data between threads, you don't know if the other thread will be accessing/changing the variables at the same time unless you specifically make sure they don't. I will go through each race condition and why I meet them in my code.

3.1 Mutual Exclusion

I achieve Mutual Exclusion by using the pthreads_mutex_lock and pthread_mutex_unlock around any code which meets the conditions outlined in section 3. The pthreads mutex function ensure that only one thread can have the lock at one time. Note that the mutex lock for the time and count are different, so one thread could be in a time critical section and the other in a count critical section and that is OK.

3.2 Bounded Waiting

Bounded waiting is ensured by pthreads. When a process is waiting for a lock, it will obtain it and enter it's critical section as soon as the thread with the lock exits it's one.

3.3 Progress

Progress is assured through the pthreads mutex functions. If a thread is blocked waiting for a lock, as soon as that lock is released (after the threads critical section is over) the other thread will get the lock atomically. There is one part of my code that requires locks on both time and count, but this will not result in a deadlock because there is no circular waiting becuase locks are always obtained in the same order in both threads. There are two times when threads are waiting for a signal from the other thread in order to progress. One is when there are no processes left in the queue. It is assured that it will not wait for the signal forever because

1. If it is the CPU thread it will only be waiting for a signal when the IO Queue has processes still in it (it it doesn't it will have exited). The IO Queue will certainly return processes to the CPU Queue and signal it because the cpu has no locks so the io can move freely in/out of critical sections

2. If it is the IO thread and the all the CPU processes don't have any IO left, the CPU will send a signal once both counts are at zero, which will mean the io will exit execution. If there is io left in any of the CPU

processes then the io will get a signal when they are done and in the IO queue

The other time the threads will wait for a signal is when they are checking to see if the time of the other thread is 'in the past' (has small time value) compared to their time. It is assured that it will not wait forever here because:

1. The other thread's time will continue increasing while the thread who is waiting is blocked, because the blocked thread has none of the locks so the other thread has nothing stopping it.

2. If the other thread's queue becomes empty and the time value is still empty, it sends a signal and the orginal thread is now synchronized to the time and will be allowed to continue.

4 Testing

In testing, I a known test case from the assignment sheet to make sure the end stats were computing correctly. I also tested on the 10 PID example posted on the website. With this file I was mostly looking to make sure the log file was written in order (by looking at arrival times) to make sure the synchronization was working. I made sure to run the test many times to pick up on any race condition errors that could happen (the bad thing about errors in multi-threaded programs is they can only happen some of the time). I have included my test files in my upload to blackboard.

5 Sample Output

5.1 log-A

New Process:
PID=1
AC=1
State=CPU
Arrive=0
Time=2

New Process:
PID=2
AC=1
State=CPU
Arrive=0
Time=1

New Process:
PID=3
AC=1
State=CPU
Arrive=0
Time=30

New Process:
PID=5
AC=1
State=CPU
Arrive=0
Time=3

New Process:
PID=7
AC=1
State=CPU
Arrive=0
Time=1

New Process:
PID=4
AC=1
State=CPU
Arrive=0
Time=2

New Process:
PID=6
AC=1
State=CPU
Arrive=0
Time=2

New Process:
PID=10
AC=1
State=CPU
Arrive=0
Time=2

New Process:
PID=8
AC=1
State=CPU
Arrive=0
Time=30

New Process:
PID=9
AC=1
State=CPU
Arrive=0
Time=2

Finishing CPU Activity.
PID=1
AC=2
State=I/O
Arrive=2
Time=10

Finishing CPU Activity.
PID=2
AC=2
State=I/O
Arrive=3
Time=7

Finishing I/O Activity.
PID=1
AC=3
State=CPU
Arrive=12
Time=3

Finishing I/O Activity.
PID=2
AC=3
State=CPU
Arrive=19
Time=2

Finishing CPU Activity.
PID=3
AC=2
State=I/O
Arrive=33
Time=5

Finishing CPU Activity.
PID=5
AC=2
State=I/O
Arrive=36
Time=8

Finishing CPU Activity.
PID=7
AC=2
State=I/O
Arrive=37
Time=20

Finishing I/O Activity.
PID=3
AC=3
State=CPU
Arrive=38
Time=1

Finishing CPU Activity.
PID=4
AC=2
State=I/O
Arrive=39
Time=18

Finishing CPU Activity.
PID=6
AC=2
State=I/O
Arrive=41
Time=7

Finishing CPU Activity.
PID=10
AC=2
State=I/O
Arrive=43
Time=7

Finishing I/O Activity.
PID=5
AC=3
State=CPU
Arrive=46
Time=1

Finishing I/O Activity.
PID=7
AC=3
State=CPU
Arrive=66
Time=1

Finishing CPU Activity.
PID=8
AC=2
State=I/O
Arrive=73
Time=5

Finishing CPU Activity.
PID=9
AC=2
State=I/O
Arrive=75
Time=15

Finishing CPU Activity.
PID=1
AC=4
State=I/O
Arrive=78
Time=12

Finishing CPU Activity.
PID=2
AC=4
State=I/O
Arrive=80
Time=4

Finishing CPU Activity.
PID=5
AC=4
State=I/O
Arrive=82
Time=8

Finishing I/O Activity.
PID=4
AC=3
State=CPU
Arrive=84
Time=5

Finishing CPU Activity.
PID=4
AC=4
State=I/O
Arrive=89
Time=12

Finishing I/O Activity.
PID=6
AC=3
State=CPU
Arrive=91
Time=1

Finishing CPU Activity.
PID=6
AC=4
State=I/O
Arrive=92
Time=8

Finishing I/O Activity.
PID=10
AC=3
State=CPU
Arrive=98
Time=1

Finishing CPU Activity.
PID=10
AC=4
State=I/O
Arrive=99
Time=8

Finishing I/O Activity.
PID=8
AC=3
State=CPU
Arrive=103
Time=20

Finishing I/O Activity.
PID=9
AC=3
State=CPU
Arrive=118
Time=2

Finishing CPU Activity.
PID=8
AC=4
State=I/O
Arrive=123
Time=5

Finishing CPU Activity.
PID=9
AC=4
State=I/O
Arrive=125
Time=12

Finishing I/O Activity.
PID=1
AC=5
State=CPU
Arrive=130
Time=1

Finishing I/O Activity.
PID=2
AC=5
State=CPU
Arrive=134
Time=1

Finishing CPU Activity.
PID=2
AC=6
State=I/O
Arrive=135
Time=15

Finishing I/O Activity.
PID=5
AC=5
State=CPU
Arrive=142
Time=2

Finishing I/O Activity.
PID=4
AC=5
State=CPU
Arrive=154
Time=1

Finishing CPU Activity.
PID=4
AC=6
State=I/O
Arrive=155
Time=14

Finishing I/O Activity.
PID=6
AC=5
State=CPU
Arrive=162
Time=1

Finishing I/O Activity.
PID=10
AC=5
State=CPU
Arrive=170
Time=1

Finishing I/O Activity.
PID=8
AC=5
State=CPU
Arrive=175
Time=1

Finishing I/O Activity.
PID=9
AC=5
State=CPU
Arrive=187
Time=1

Finishing I/O Activity.
PID=2
AC=7
State=CPU
Arrive=202
Time=1

Finishing I/O Activity.
PID=4
AC=7
State=CPU
Arrive=216
Time=1

5.2 log-B

Process PID-3 is terminated.
PID=3
AC=3
State=CPU
Arrive=38
Time=1

Process PID-7 is terminated.
PID=7
AC=3
State=CPU
Arrive=66
Time=1

Process PID-1 is terminated.
PID=1
AC=5
State=CPU
Arrive=130
Time=1

Process PID-5 is terminated.
PID=5
AC=5
State=CPU
Arrive=142
Time=2

Process PID-6 is terminated.
PID=6
AC=5
State=CPU
Arrive=162
Time=1

Process PID-10 is terminated.
PID=10
AC=5
State=CPU
Arrive=170
Time=1

Process PID-8 is terminated.
PID=8
AC=5
State=CPU
Arrive=175
Time=1

Process PID-9 is terminated.
PID=9
AC=5
State=CPU
Arrive=187
Time=1

Process PID-2 is terminated.
PID=2
AC=7
State=CPU
Arrive=202
Time=1

Process PID-4 is terminated.
PID=4
AC=7
State=CPU
Arrive=216
Time=1

Average waiting time in CPU queue: 52.700001
Average waiting time in I/O queue: 70.699997
CPU utilization: 0.566820%
I/O utilization: 0.925926%