Michael Weaver

CST – 221

John Zupan

2/4/2018

**Deadlock Avoidance**

Scenario

In this scenario each of the processes are fighting for access to a shared variable, which could be substituted for anything. I have created both a method which uses individual pthreads and one that create copies of the process through fork. In each case threads are created, lock the mutex, execute their critical region, all the while printing their output to the screen and file.

Approach

In the case of the individual pthread, each are joined and then executed. While they are waiting for access to the resource a clock timer is constantly checked. If a particular thread reaches its time limit it is put to sleep until the other processes have finished their iterations. Each generated output to the screen and to a log file.

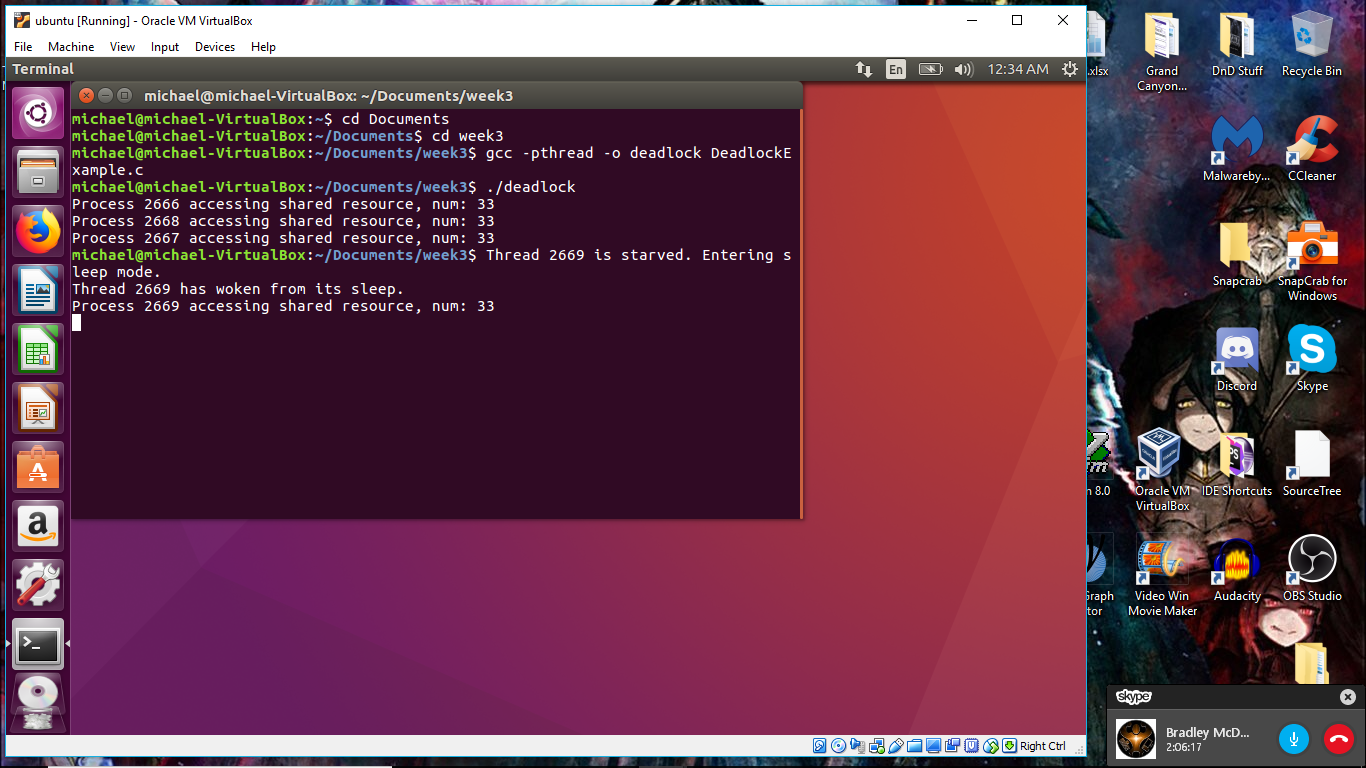
The fork version is extremely similar to the individual thread one, only the difference is that each thread has its own unique ID. The last thread becomes starved, enters into sleep mode, then reawakens later to resume its execution.

Correct but impossible approach

The correct way to implement a stop a process after a timer expires and then restart it is to use the kill command to execute kill(pid, SIGSTOP) and then kill(pid, SIGCONT). The problem with this is that the only way to generate different process ID’s is to fork them. This causes the mutex that is supposed to be shared between processes to be duplicated with each fork call, ultimately making four mutexes that none of the fork children share.

When using multiple processes created through pthread, they all have the same PID and therefor cannot be individually affected by the kill function. Sending SIGSTOP would in fact stop all the pthreads and stall the execution completely.

Screenshot of successful execution



Analysis and Impressions

The timer seemed to be ineffective and mostly useless, this may be the result of only having a single lock to restrict access to a single resource though. I am uncertain of how to utilize a secondary lock specifically for other resources especially when the mutex locks never are even assigned to anything. It makes me wonder what would happen if I wrote a line of code changing the shared resource that is not inside a locked mutex portion.

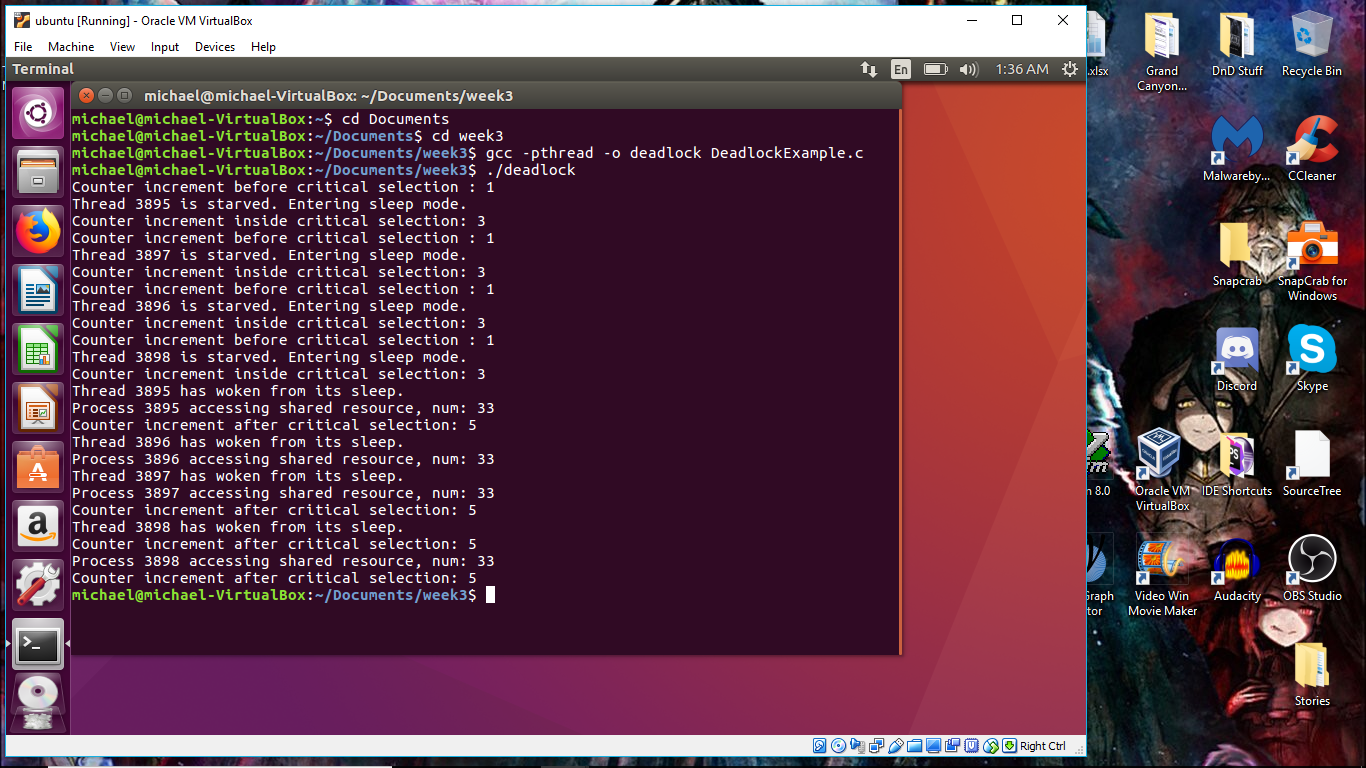
Using a timer to manipulate multiple threads becomes highly intricate if not impossible, therefor I do not recommend it whatsoever. The use of mutex locks is far superior in every facet.

Timer’s can only interact with forked processes because in order to use KILL(PID, SIGSTOP) you need the individual process ID. With fork that changes, enabling you to use the command, though then it creates copies of all the variables created beforehand, making the mutex lock far less efficient. When you use pthread to create multiple threads, they all contain the same PID and therefor are not compatable with the KILL commands.

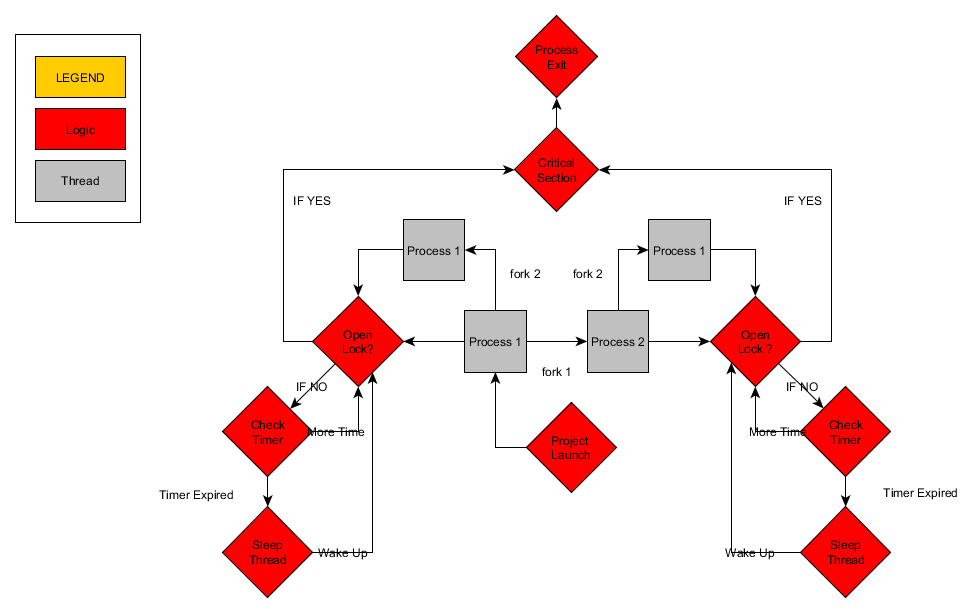
It may very well be possible to use KILL commands if the fork interacts correctly with the processes. Otherwise it is a better option to rely on Mutexes to cause threads to remain waiting for the lock to open.

Side Analysis

I actually went ahead and added some code to manipulate the counter variable inside and outside of the critical selection for each of the processes and found the result to be rather strange. Each of the threads ends up becoming starved as the others ones manipulate the counter increment before critical selection, and the threads are interrupted by the other thread calls to the counter. As I thought, code that interacts with a variable outside of critical selection interferes with the variable being interacted with inside the critical selection.



Flowchart



References

Wikipedia. (2018). *Signal (IPC).* Retrieved from <https://en.wikipedia.org/wiki/Signal_(IPC)#SIGTERM>

GeeksforGeeks. (2018). *Mutex lock for Linux Thread Synchronization.* Retrieved from <https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/>

Tutorialspoint. (2018). *C library function – clock().* Retrieved from <https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm>

Icegood. (2010). *How correctly continue paused process.* Retrieved from <https://cboard.cprogramming.com/linux-programming/132657-how-correctly-continue-paused-process.html>

Javatpoint. (2018). *C fprintf() and fscanf().* Retrieved from <https://www.javatpoint.com/fprintf-fscanf-in-c>

Aquarius\_Girl. (2012). *How to use pthread\_mutex\_trylock*? Retrieved from <https://stackoverflow.com/questions/9258308/how-to-use-pthread-mutex-trylock>

Ubiquabacon. (2010). *Check to see if a pthread mutex is locked or unlocked (After a thread has locked itself)*. Retrieved from <https://stackoverflow.com/questions/4362459/check-to-see-if-a-pthread-mutex-is-locked-or-unlocked-after-a-thread-has-locked>