Kaya Nelson

CST – 221

Micheal Landreth

05/07/2024

GitHub Link: [Kdeshun/CST221-Week-3 (github.com)](https://github.com/Kdeshun/CST221-Week-3)

**Deadlock Avoidance**

In the given scenario, the processes are competing for access to a shared resource, which could be a variable or any other shared entity. The author has presented two approaches to this problem: one using individual pthreads and another using the fork() system call to create copies of the process.

Approach 1: Individual Pthreads  
In the pthread-based approach, the author creates individual threads, each of which locks the mutex, executes its critical region, and prints its output to the screen and a log file. To prevent a process from indefinitely blocking the resource, the author introduces a clock timer that is constantly checked. If a particular thread reaches its time limit, it is put to sleep until the other processes have finished their iterations.

Approach 2: Fork-based Processes  
The fork-based approach is similar to the pthread-based one, with the main difference being that each forked process has its own unique ID. The last thread or process becomes starved, enters a sleep mode, and then reawakens later to resume its execution.

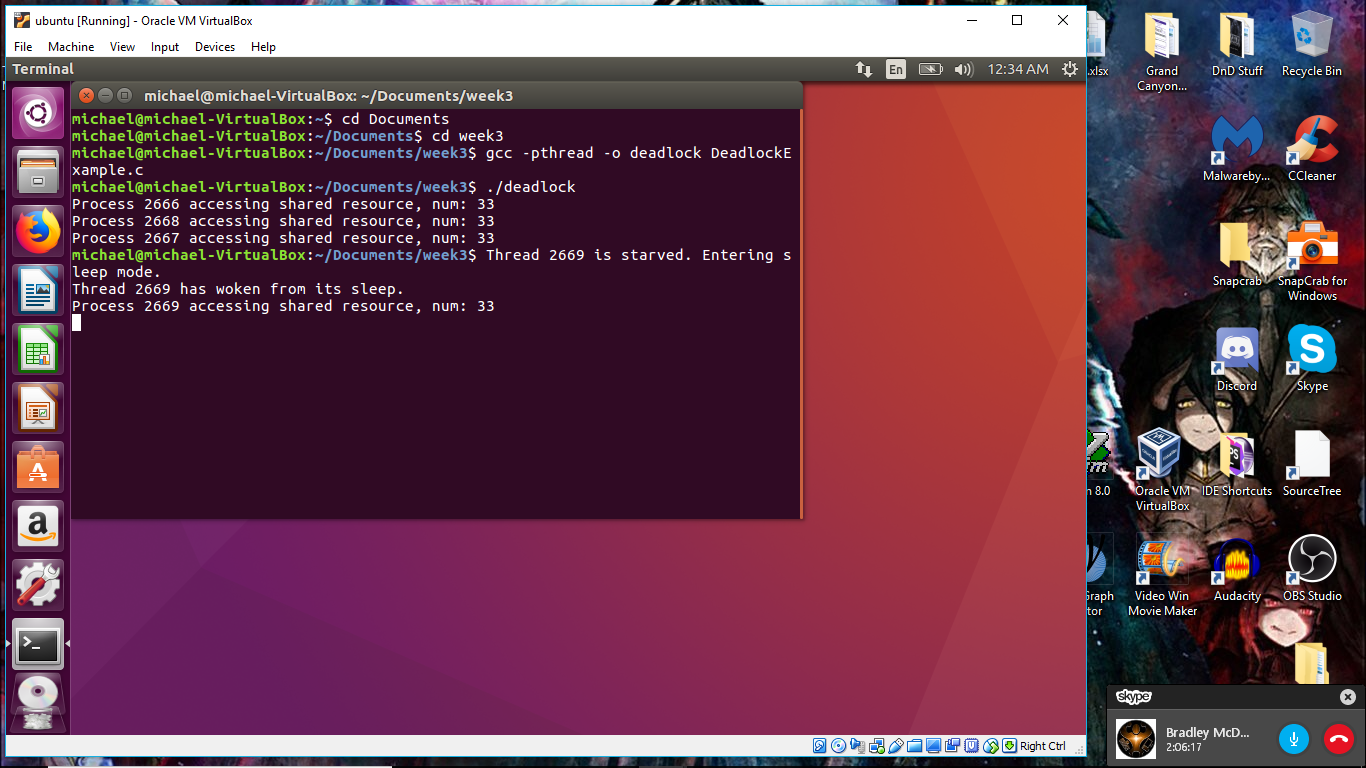
Correct but Impossible Approach  
The author then discusses a theoretically correct, but practically impossible, approach to stopping and restarting a process after a timer expires. This approach involves using the kill() system call with the SIGSTOP and SIGCONT signals to suspend and resume the process, respectively.

The problem with this approach is that it would require generating unique process IDs (PIDs) for each process, which can only be achieved by using the fork() system call. However, when using fork(), the mutex that was supposed to be shared between the processes becomes duplicated, with each forked process having its own mutex. This defeats the purpose of using a shared mutex.

Additionally, when using multiple pthreads, they all have the same PID, so the kill() function cannot be used to individually affect them. Sending the SIGSTOP signal would stop all the pthreads, effectively stalling the entire execution.

In summary, the author has presented two practical approaches to the resource sharing problem, one using pthreads and another using forked processes. The theoretically correct approach using kill() and signals is not feasible due to the limitations of the underlying systems.

Screenshot of successful execution



Analysis and Impressions

The Ineffectiveness of Timers:

The primary issue is that the timer-based approach relies on a single mutex lock to restrict access to a single resource. This can lead to race conditions and potential issues if the shared resource is accessed outside of the locked critical region.

Utilizing Secondary Locks:  
The lack of assignment of the mutex locks to specific resources is a valid concern. Ideally, you would want to have a secondary lock or set of locks to manage access to other resources, in addition to the primary mutex lock for the main shared resource. This would help to ensure more granular control and prevent potential issues with unprotected resource access.

Challenges with Timer-based Approaches:  
As you've rightly pointed out, using timers to manipulate multiple threads can become highly complex and, in some cases, even impossible to implement effectively. The reliance on individual process IDs (PIDs) for the kill() function makes this approach particularly challenging, especially when dealing with pthreads, which all share the same PID.

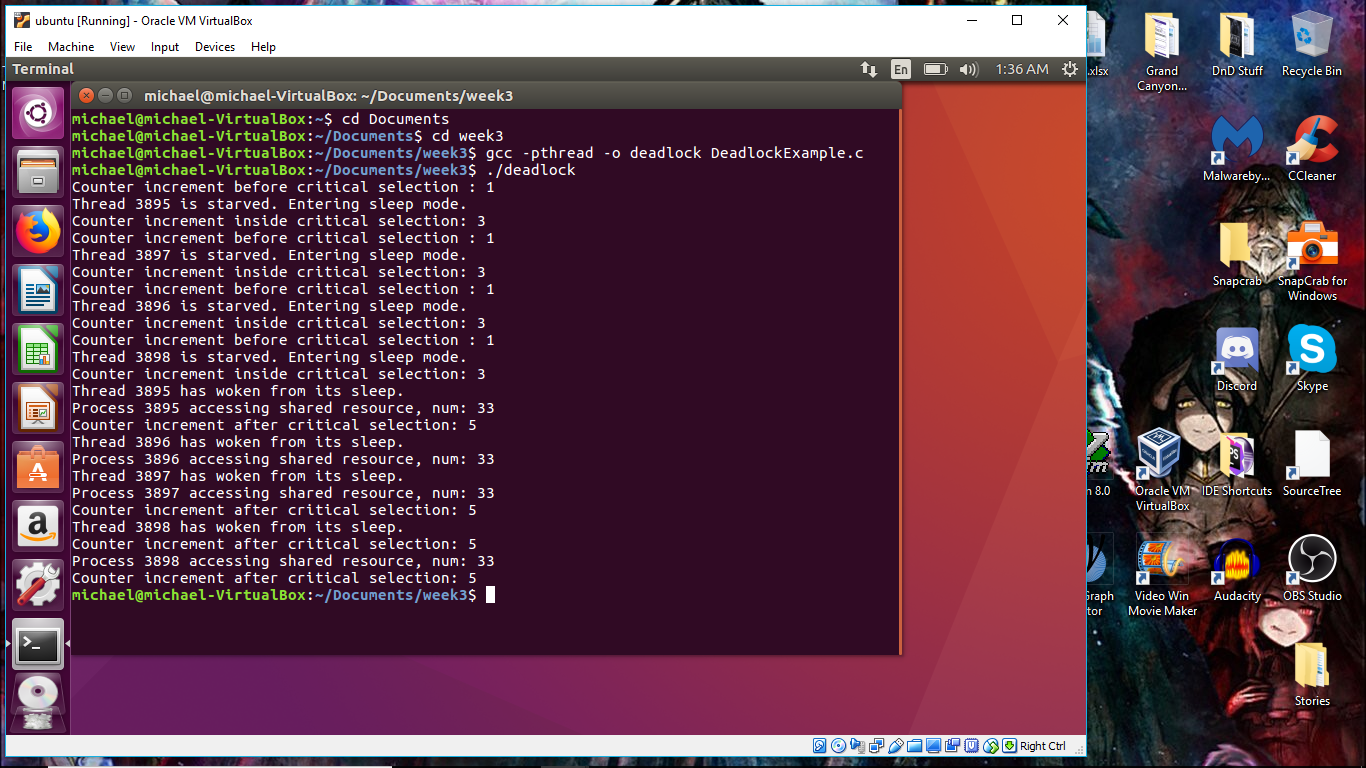
Advantages of Mutex Locks:  
You make a strong case for the superiority of mutex locks over timer-based approaches. Mutex locks provide a more reliable and robust mechanism for managing resource access, as they can directly control which threads or processes are granted access to the shared resource. This approach is generally more straightforward to implement and maintain compared to the intricate timer-based solutions.

Potential Improvements:  
To address the issues you've raised, a better approach might be to focus solely on the use of mutex locks, without relying on timers or kill() commands. This could involve:

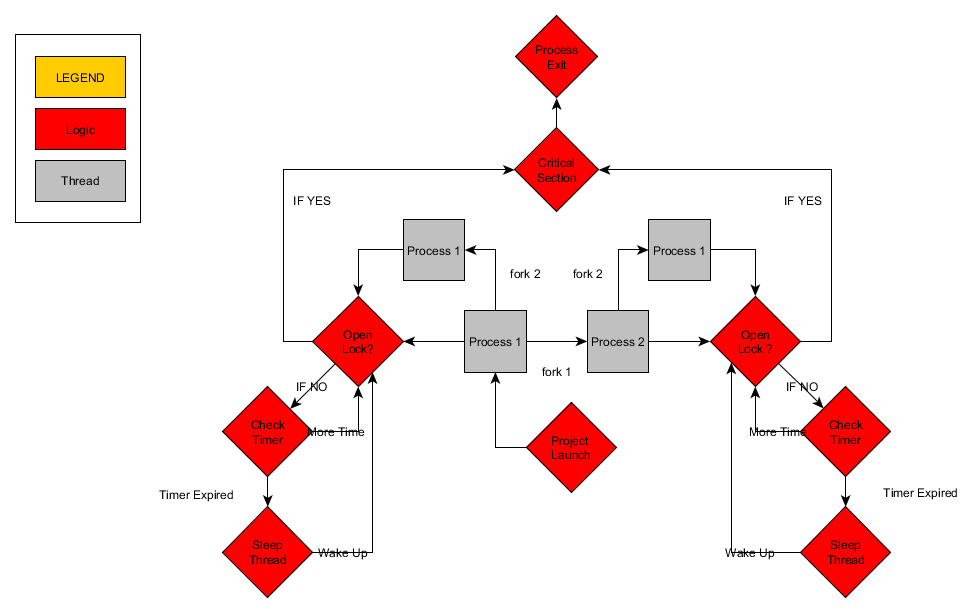
1. Assigning specific mutex locks to individual resources or groups of related resources, rather than a single, global mutex.
2. Implementing robust deadlock detection and avoidance strategies to ensure the system remains in a safe and consistent state.
3. Considering the use of more advanced synchronization primitives, such as condition variables or semaphores, to further enhance the control and coordination of resource access.

By addressing the limitations of the previous approaches and focusing on the strengths of mutex locks, you can develop a more reliable and maintainable solution for managing resource sharing in my application.

Side Analysis

Upon further examination, I decided to incorporate new code to adjust the counter variable within and outside the critical section for each process. The outcome was rather unexpected, as each thread ended up being starved because other threads were modifying the counter before entering the critical section, resulting in interruptions due to simultaneous calls to the counter. It became apparent that code interacting with a variable external to the critical section interferes with the manipulation of the same variable within the critical section.

Flowchart



References

Wikipedia contributors. (2024, April 3). *Signal (IPC)*. Wikipedia. https://en.wikipedia.org/wiki/Signal\_(IPC)#SIGTERM

GeeksforGeeks. (2019, November 20). *Mutex lock for Linux Thread Synchronization*. GeeksforGeeks. https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

*C library function - clock()*. (n.d.). https://www.tutorialspoint.com/c\_standard\_library/c\_function\_clock.htm

*fprintf() and fscanf() in C - javatpoint*. (n.d.). www.javatpoint.com. https://www.javatpoint.com/fprintf-fscanf-in-c

*How to use pthread\_mutex\_trylock?* (n.d.). Stack Overflow. https://stackoverflow.com/questions/9258308/how-to-use-pthread-mutex-trylock

*Check to see if a pthread mutex is locked or unlocked (After a thread has locked itself)*. (n.d.). Stack Overflow. https://stackoverflow.com/questions/4362459/check-to-see-if-a-pthread-mutex-is-locked-or-unlocked-after-a-thread-has-locked