Operating Systems: Sync

1 Introduction

In this lab, we will explore the synchronization mechanism using POSIX Threads library. The POSIX Threads (Pthreads) library is a standard for multithreading in C and C++, providing a set of APIs to create and manage threads efficiently. It enables concurrent execution within a process, allowing tasks to run in parallel and improving performance on multi-core systems. Pthreads support synchronization mechanisms like mutexes and condition variables to prevent race conditions and ensure thread safety. Commonly used in system programming and high-performance applications, Pthreads offer fine-grained control over thread execution, making them a powerful tool for parallel processing.

2 Background

Before proceeding further, it is worthwhile to take a look at some of the important APIs.

Important APIs

- pthread_mutex_lock & pthread_mutex_unlock
  These functions lock and unlock a mutex (mutual exclusion) to ensure only one thread accesses a critical section at a time, preventing race conditions.
- pthread_cond_wait & pthread_cond_signal
  Conditional variables allow threads to wait for a condition to be met. pthread_cond_wait makes a thread wait while releasing the associated mutex, and pthread_cond_signal wakes up one waiting thread when the condition changes.
- pthread_spin_lock & pthread_spin_unlock
  Spinlocks are lightweight locks where a thread continuously checks (spins) until it acquires the lock. These are useful in low-contention scenarios for faster synchronization.
- sem_wait & sem_post
  These semaphore functions manage access to shared resources. sem_wait decreases (locks) the semaphore, blocking if the value is zero, while sem_post increases (unlocks) it, allowing other threads to proceed.

3 Programs

1. cmpxchg.c
   Summary: In this program, the compare_and_swap function implements an atomic compare-and-swap (CAS) operation using inline assembly with the *CMPXCHG* instruction. It takes a pointer to a volatile integer (*lock_var*), an expected value, and a new value. If the current value of *lock_var* matches expected, it atomically replaces it with *new_value*; otherwise, it leaves *lock_var* unchanged. The function returns the original value of *lock_var*, allowing the caller to determine whether the swap was successful.

2. cond_var.c
   Summary: In this program, the classic producer-consumer problem is implement using conditional variables. If the consumer acquires the lock first, then it checks the conditional variable in the wait() portion. While waiting on the conditinal variable, it unlocks the mutex lock and lets the producer to add data. Once the producer produces some data, it signals the conditional variable and signals the consumer to use the data.

3. deadlock.c
   Summary: In this program, deadlock is mostly likely to occur as two threads try to acquire 2 locks in different order. This creates a hold and wait dependency (Thread 1 acquires mutex1 and waits for mutex2, while thread 2 acquires mutex 2 and waits for mutex 1) which is one of the conditions for deadlock.

4. deadlock2.c
   Summary: In this program, deadlock occurs as the thread tries to acquire the lock(mtx) which it has already acquired.

5. fork.c
   Summary: This is a simple program, where a process creates a child process using the fork() method and prints the child PID from both processes.

6. mcs.c
   Summary: In this program, we see the implementation of the Mellor-Crummey and Scott (MCS) lock. The MCS lock is a scalable, fair, and efficient spinlock designed for multiprocessor systems. It minimizes contention and other overhead by maintaining a queue of waiting threads. Each thread waiting for the lock spins on a private variable rather than a shared lock variable, reducing traffic and improving performance in high-contention scenarios. When a thread releases the lock, it hands over ownership directly to the next thread in the queue, ensuring fairness and avoiding starvation. This makes MCS locks well-suited for parallel systems with a large number of threads.
   In this program, the *mcs_lock_acquire()* and *mcs_lock_release()* explains

how these locks are acquired and released based on inidividual threads private variables (node-¿locked) instead of checking a global variable.

7. milli.c, milli_trick.c, milli_fixed.c and milli_fast.c
   Summary: In these 3 programs, the advantages of using threads but the importance of maintaining mutual exclusion can be seen.
   In **milli.c**, the *goal* variable gets incremented until it reaches the *target* value (which is 1000000) by a single thread.
   In **milli_trick.c**, the *goal* variable is incremented by multiple threads. Based on the number of threads given by the user (in command line arguments) the work gets divided among the threads. However, as the updating of variable *goal* is not protected for mutual exclusion, the final value of *goal* is never the *target* value (= 1000000) except when the number of threads is 1 or the system is single core machine.
   In **milli_fixed.c**, this issue is fixed using a spin lock to ensure mutual exclusion. This enables the variable *goal* to get updated in a thread safe manner. However, it is not recommended to use spin locks in user programs as they cause deadlock in a system with single core and FCFS scheduling policy scenario (Solaris had this issue).
   In **milli_fast.c**, the threads completely avoid the critical section by incrementing the local variable and then finally adding it to the global variable after the join.

8. mutex.c
   Summary: In this program, a mutex lock is used to ensure mutual exclusion (allowing only one thread to update the counter at a time).

9. mutexvssem.c
   Summary: In this program, a scenario where a thread tries to unlock a mutex that has been locked by another thread happens. In case of mutex locks, the lock should be unlocked by the same thread which tried to lock it. But in case of semaphores wait() and signal() can be done by different threads (more like conditional variables).

10. pipe.c
    Summary: In this program, using fork() method, a child process is created. Using the pipe() API, 2 pipes (one for writing message from parent to child and other for reading message sent by the parent) are created. Then the parent send *Hello from parent!* message from parent to the child. The child process prints the message once it receives it. This method can be used for interprocess communication.

11. priority.c
    Summary: In this program, two threads with different priority are created. They are made to acquire a lock and enter the critical section. Due to *sleep(1)* in *line 21*, always the low priority thread acquires the lock and makes the high priority thread to wait until it releases the lock.

12. process_race.c
    Summary: In this program, similar to *pipe.c*, multiple pipes are created for inter process communication. Unlike *pipe.c*, child processes use the same file descriptor(same queue) to communicate with the parent process. This can cause the race condition and may lead to undesirable behavior (like receiving message out of order) as write() does not do explicit synchronization.

13. recursive_mutex.c
    Summary: In this program, recursive aspect of mutex lock is explored. If the mutex lock has a recursive behavior (using PTHREAD_MUTEX_RECURSIVE) mutex shall maintain the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count shall be set to one. Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire.
    If you comment *line 21* (not setting the recursive nature), then the thread fails to acquire the mutex lock in second recursive call and leads to deadlock.

14. rwlock.c
    Summary: In this program, solution to reader-writers' problem is implemented using mutex lock and rwlock (particular library lock available for this purpose). While mutex lock treats both readers and writers same (allows only one user (either reader or writer) at a time), *pthread_rwlock_t* allows multiple readers to read a shared resource concurrently while ensuring exclusive access for one writer. It improves performance over mutexes when frequent reads and infrequent writes occur.
    It also provides comparison in terms of execution time between two approaches.

15. semaphores.c
    Summary: In this program, 3 resources are shared among 10 threads with one thread to use one resource at a time and no resources are shared simultaneously among mulitple threads. Using wait() and signal() APIs of the semaphore, this gets achieved.

16. sem.c
    Summary: In this program, by initializing a semaphore as a binary semaphore, it is made to behave like a mutex lock. It allows only one thread to access the critical section at a time.

17. spinlock.c
    Summary: Similar to milli_fixed.c, a spin lock is used to ensure mutual exclusion while updating a shared variable. Spin lock achieves mutual exclusion by doing busy wait (while loop based continuous checking). This

is recommended only when the critical section is very small and using mutex lock becomes expensive in terms of performance.

18. spinvsmutex.c
    Summary: In this program, main program holds both locks both spin lock and mutex lock and releases it after 30 seconds. Once it is released, the threads try to acquire the lock and release them.

19. thunderherd.c and thunderherd_fixed.c
    Summary: In these 2 programs, the main task is that producers produce an item and fill it in a slot (number of slot (*slots_available* = 1)). The mutual exclusion is ensured by using a mutex lock. However in the first program (thunderherd.c), the threads check for free slot by continously polling on the lock which is similar to spin lock approach. This leads to busy wait and leads to wastage of CPU cycles. This issue is fixed in thunder_fixed.c program by using conditional variable. When the slot is available it wakes up one of the waiting threads instead of making all threads to poll. This prevents the wastage of CPU cycles.

20. tsan.c
    Summary: This program explores the idea of race condition (similar to milli_trick.c program). In this program 2 threads try to update the shared variable (*shared_data* initialized to 0) 1000000 times each. However due to race condition, the final value is most likely not 2000000. In order to reach the value of 2000000, critical section (updating shared variable) should be protected via locks to ensure mutual exclusion.

21. unfair_spin.c
    Summary: In this program, a spin lock is used to ensure mutual exclusion. However, upon execution one can see that the lock is acquired not in the First Come First Serve (FCFS) fair way, rather it is given randomly.

22. xchg.c
    Summary: This program demonstrates an atomic exchange operation using inline assembly in C. The *atomic_xchg* function swaps a given value with the memory location atomically using the *xchg* instruction, ensuring thread safety. In the main function, an integer variable is initialized and printed before calling *atomic_xchg* to replace its value with a new one. The program then prints both the updated value and the previous value stored in the variable. This atomic swap operation is particularly useful in concurrent programming and low-level system tasks where synchronization is required.

23. reversing_ptmalloc/arena.c and reversing_ptmalloc/tcache.c
    In these programs, ptmalloc's strategy to use different heap arena to avoid contention. In *arena.c* file, multiple threads malloc() and free() 16 bytes of data successively. In *tcache.c*, 2 threads malloc() and free() in a particular order to show how caching happens at a per thread level. This nature can

be further explored by using the gdb file given in the same folder through the following command: **$ gdb -X ./[name_of_the_gdb_file] ./a.out**

4 Takeaway

In this lab, we explored different synchronization mechanisms offered by the Pthread library and their behavior under various circumstances. Students are recommended to take a look into the bonus programs for further exploration.