Network System Programming Quiz 4
2025/11/18 Tuesday

To synchronize multiple Linux processes accessing a shared memory region for reading and writing, you must use inter-process synchronization mechanisms to avoid race conditions and data corruption. The most common and portable approach is to use process-shared POSIX mutexes or POSIX semaphores placed inside or alongside the shared memory segment, allowing safe and efficient synchronization between multiple processes accessing the shared data concurrently. You are advised to choose the approach in which you have the greatest confidence.

1. POSIX Mutex with Process-Shared Attribute:
   - Use a pthread_mutex_t placed inside the shared memory and initialize it with the attribute PTHREAD_PROCESS_SHARED.
   - Processes lock the mutex (pthread_mutex_lock()) before accessing the shared data and unlock (pthread_mutex_unlock()) afterward.
   - Guarantees mutual exclusion ensuring only one process accesses the data at a time.
2. POSIX Semaphores
   - Named semaphores created with sem_open() or unnamed semaphores in shared memory with sem_init().
   - Use sem_wait() before entering critical section and sem_post() after finishing.
   - Allows coordination between processes to manage access rights.

Design and implement an efficient *M*-Producer 1-Consumer architecture using processes, where Shared Memory is employed for communication, along with suitable synchronization to ensure safe concurrent access. The program must accept two command-line arguments: The number of Producer processes (e.g., *M*=10). The number of data items each Producer will generate (*N*=10). Shared Memory Structure (Stack): Shared memory must contain a *C* struct named ***SharedStack***. This struct must implement a stack data structure with a capacity of *M* elements, enforced by an array. Each element must store a *PID/Value* pair. The structure must include a *top* index to manage the stack.

In synchronization schemes utilizing semaphores, a typical implementation requires the creation of a semaphore set containing three individual semaphores to effectively manage access control: *Mutex* (Index 0): Initial value 1. Used to protect the SharedStack.top index. *Empty Slots* (Index 1): Initial value *M*. Counts available slots in the stack. *Full Slots* (Index 2): Initial value 0. Counts data items available for the Consumer. Producer Logic is: Each of the *M* Producers must iterate N times: push(): Wait on Full, acquire Mutex, write a random value (1-10) along with its own PID to the stack, release Mutex, signal Full. Consumer Logic is: The single Consumer must read until it has processed a total of *M X N* data items. pop(): Wait on Empty, acquire Mutex, read a *PID/Value* pair from the stack, update a cumulative sum, release Mutex, signal Empty.

$ ./data_aggregator 2 3 # M=2 Producers, N=3 writes each (Total 6 points) # Producer P1 (PID 500) starts writing... # Producer P2 (PID 501) starts writing... # Consumer (PID 400) waits on Semaphore 1 (Full Count) # P1 writes (500, 7), increments Sem 1. # P2 writes (501, 3), increments Sem 1.	# Consumer reads (500, 7). Sum = 7. # Consumer reads (501, 3). Sum = 10. # ... continues until 6 total points processed. # Output: Total Data Points Processed: 6 Final Cumulative Sum: [Calculated Sum]