CSCI4730

Name: Igor Rodrigues Goncalves
UGA#: 811 363 160

1) List of Implementations

a) **Shared Bounded Buffer:** *Firstly I declared a fixed global-size circular buffer to act as a queue for incoming client socket file descriptors. I also added two index variables, buffer_in (for producer) and buffer_out (for consumers) that manage this queue.*

b) Synchronization Primitives: *I used a combination of one mutex and two counting semaphores to manage the buffer's access.*

- ***pthread_mutex_t_lock:*** *this is a mutex lock to provide mutual exclusion for the critical sections where the buffer array are changed.*
- ***sem_t_sem_empty:*** *this is a counting semaphore initialized to 'MAX_REQUEST' and it represents the count of empty slots (aka available space) in the buffer).*
- ***sem_t_sem_full:*** *this is the other counting semaphore, but this time initialized to 0, It represents the number of filled slots (aka pending requests) in the buffer.*

c) main() Function: *The main function now…*

- *passes and validates the port and numThread arguments.*
- *initializes the mutex lock, sem_empty, and sem_full to its beginning values.*
- *creates one listener thread (producer).*
- *creates a thread pool of numThread worker threads (consumers).*
- *passes a unique ID to each worker thread that is dynamically allocated.*
- *joins all threads to wait for them to complete before cleaning up the sync primitives.*

d) listener() (Producer) Function Synchronization Protocol:

- *Program accepts a new socket 's'.*
- *sem_wait(&sem_empty) takes an empty slot. If the buffer is full (sem_empty == 0), this will block the producer until a consumer frees a slot.*
- *pthread_mutex_lock(&lock) acquires the mutex to safely access the buffer.*
- *request_buffer[buffer_in] = s places the new request in the buffer.*

- *buffer_in = (buffer_in + 1) % MAX_REQUEST updates the 'in' index for the circular buffer.*
- *pthread_mutex_unlock(&lock) releases the mutex so another can take it.*
- *sem_post(&sem_full) signals that a new item is available which then increments the 'sem_full' count and wakes up any sleeping worker thread that may be waiting.*

e) **worker() (Consumer) Function Synchronization Protocol:**

- *sem_wait(&sem_full) claims an item. If the buffer is empty (aka sem_full = 0) this call blocks the worker until the producer adds another item.*
- *pthread_mutex_lock[buffer_out] retrieves the request from the buffer.*
- *buffer_out = (buffer_out + 1) % MAX_REQUEST updates the out index.*
- *pthread_mutex_unlock(&lock) then finally releases the mutex.*
- *sem_post(&sem_empty) signals that an empty slot is now available, which then increments the 'sem_empty' count and also wakes up the producer if it was blocked.*
- *process(fd) is called after mutex is released.*

2) Testing

a) **Functional Testing:** *Looking at the output of the client program, I could infer that the test was successful since the final line reported '(0 failed)'.*

b) **Load / Stress Testing:** *To test for stability and potential deadlocks, I ran the client program in a loop with very high thread counts (aka 100+) to simulate a "worse case" scenario where many threads would fight for the server's limited buffer slots.*

c) **Logging:** *While the client program was running, I watched the server output in the first Terminal window and since it outputs all of the Worker Threads one by one, this proved the work was being distributed across the entire thread pool evenly.*

3) Explanation of Shared Data and Synchronization:

a) **Shared Data:**

- *int request_buffer[MAX_REQUEST] is the FIFO circular array queue, it is shared because the listener thread writes to it and all the worker threads can then read from it.*
- *int buffer_in is the index which the producer writes the next item.*
- *int buffer_out is the index which the consumers read the next item.*

b) **Synchronization:**

- *sem_t sem_empty (counting semaphore) makes it so if the buffer is full the producer thread is put to sleep by the OS. The consumer then uses 'sem_post' after removing an item which adds an empty slot and wakes up the producers if it was asleep.*
- *sem_t sem_full (counting semaphore) makes it so if the buffer is empty the consumer thread is put to sleep by OS. The producer then uses 'sem_post' which adds an item signaling that an item is ready and waking up a sleeping consumer.*
- *pthread_mutex-t lock (mutex) ensures that only one thread at a time can execute the code in the critical section which modifies 'request_buffer', 'buffer_in', or 'buffer_out' preventing a race condition.*

4) Race Condition and Deadlock Prevention

a) Race Conditions:

- Verified by sending 100 requests on multiple occasions making a race condition likely, however the 'Failed requests 0" result in the output assures a race condition did not occur.

b) Deadlock Prevention: *For a deadlock to happen, it requires certain conditions including HOLD, WAIT, and CIRCULAR WAIT.*

- ***Circular Wait Prevention:*** *Since there is only one mutex, a circular wait is impossible since threads will only ever wait for this one mutex lock meaning there is no way for a circle of dependency to form.*
- ***Hold and Wait Prevention:*** *The 'sem_wait' comes before the 'pthread_mutex_lock', meaning a thread must wait for the semaphore first and can only acquire the lock to modify the buffer after it is guaranteed a resource. This makes it so the buffer never sleeps while holding the mutex which prevents deadlock.*