CSCI 4730 Operating Systems | Project 2

Ruben Cristea
811427744

b) What I have done

The multithreading and crash handling is working. I first started by making sure the multi-threading was working. To test, I ran $ ./webserver_multi [PORT_NUMBER] 10 to test with 10 threads first. Then I ran $ ./client localhost [PORT_NUMBER] 10 to send 10 requests. I received files with waiting time around 0.100 seconds, so I determined the multi-threading was working.

Next the crash handling tries to join the threads with pthread_tryjoin_np. If the thread does not crash, it successfully joins and moves on to the next thread. If the thread does crash it will recreate the thread to try again. To test this I ran webserver_multi with crash rate of 50 to get the highest possible chance of crashing, then with client I sent 10 requests each time. The requests from the client that crashed were restarted and the wait time would increase with the number of crashes that happened. If it was just one crash it took about 0.3 seconds if there were more, like 8 crashes, wait time could end up reaching 1 full second.

c) Design of shared data structure and synchronization

The shared data structure is an array of integers that hold the request number from the client that would await processing in the consumer / worker threads. The synchronization design uses the Bounded Buffer Problem. The mutex is initialized to 1, the full semaphore to 0, and the empty semaphore to MAX_REQUEST, which is a const set to 100 max requests allowed to be handled. Testing the synchronization went the same way as testing to make sure the crash handling worked. If there was an error, the server would get stuck and not process any more requests. The race conditions were fixed with the mutual exclusion from the mutex lock. The mutex lock was locked before the buffer was being written to or being read from. Then unlocked after those operations are completed, so the buffer is not changed during a reading or a writing from another thread.

In the producer, the empty semaphore would be waited on in case the buffer was full. If it was full, then the producer would wait before writing to the buffer. Then it would post to the full semaphore to let the consumer start consuming.

Continued …

```
void producer(int s) {
        sem_wait(&sem_empty);
        pthread_mutex_lock(&mutex);


        int semfull;
        sem_getvalue(&sem_full, &semfull);
        request[semfull] = s;

        pthread_mutex_unlock(&mutex);
        sem_post(&sem_full);
}
```

The consumer waits for the producer to produce an item first before consuming it by waiting for sem_full. Then it would post the empty semaphore to let the producer know that there is a new available slot in the buffer. The semaphore value of sem_full was used to manage which place in the buffer we are. It starts at 0 and will get updated for each new place in the buffer that was written to, thanks to posting it in the producer.

```
void* consumer() {
        while (1) {
                int req;
                sem_wait(&sem_full);
                pthread_mutex_lock(&mutex);

                int semfull;
                sem_getvalue(&sem_full, &semfull);
                req = request[semfull];

                pthread_mutex_unlock(&mutex);
                sem_post(&sem_empty);

                process(req);
        }
}
```

Using the semaphores and the mutex lock prevents race conditions and deadlocks in the program.