Milo Bauman
CSCI 6730 Operating Systems
Programming Assignment 2 README
10/23/25

Code Breakdown

```
void thread_control() {

    // Set the semaphore values
    pthread_mutex_init(&mutex, NULL);    // init the mutex
    sem_init(&full, 0, 0);               // number of full spaces in buffer (0)
    sem_init(&empty, 0, MAX_REQUEST);    // number of empty spaces in buffer (MAX_REQUEST)

    // Set buffer pointer
    in = 0;
    out = 0;

    // Create 1 listener thread
    pthread_t listenerThread;
    if (pthread_create(&listenerThread, NULL, listener, NULL)) {
        printf("Error: Failed to create listener thread.");
        exit(1);
    } // if
    // printf("Listener thread created and ready.\n");

    // Create numThreads worker threads
    pthread_t workerThread[numThread];
    for (int i = 0; i < numThread; i++) {
        if (pthread_create(&workerThread[i], NULL, worker, NULL)) {
            printf("Error: Failed to create worker thread %i.\n", i);
            exit(1);
        } // if
        // printf("Worker thread %i created and ready to process requests.\n", i+1);
    } // for

    // Monitor the threads
    threadMonitor(workerThread, &listenerThread);

} // thread_control
```

The first thing my thread control function does is initialize the semaphore logic. I used the examples from the class slides as a guide for adding my semaphores. Just before creating the threads, I initialize the semaphores to their proper values. Full is set to 0 since there are 0 full slots to begin with. Empty is set to MAX_REQUEST, since we want to allow that many requests at a time. The mutex is initialized using the pthread function pthread_mutex_init, but this essentially is setting is to 1, or “unlocked.”

I also set my in and out buffer “pointers” to 0. They aren’t really pointers, they just tell the consumer and producer what part of the buffer should be added to and what part should be removed from. They don’t necessarily want to be adding and removing from the same place, so this helps ensure that requests are processed in the order that they are received.

The first thing I added was the listener thread which performs the listener function. I then added a loop to create the requested number of worker threads. If any of these thread creations fails, the program prints an error message and exits.

After all of this, the main thread then executes the function threadMonitor. I will explain this later on.

```
    /* PLACE FD IN BUFFER */
    sem_wait(&empty);               // Wait for there to be an empty spot in the buffer
    pthread_mutex_lock(&mutex);     // Lock the mutex

    buffer[in] = s;                 // Place fd into buffer
    in = (in + 1) % MAX_REQUEST;    // Move the buffer index forward 1

    pthread_mutex_unlock(&mutex);   // Unlock the mutex
    sem_post(&full);                // Signal that there is a new full space in the buffer
} // while
```

In the listener function, I added the semaphore and mutex logic as well as some logic to make the listener safely add to the buffer. Once it has received a request, it first waits for there to be an empty slot in the buffer. It locks the mutex, which prevents any other threads from accessing the buffer. The “in” pointer tells the listener the location of the next spot to be filled in the buffer.

The listener then adds the new fd to the buffer and increments its “in” pointer. This means that next time, it will add to the next slot of the buffer. It then unlocks the mutex and signals the full semaphore to indicate that there is a new full slot in the buffer. The listener the restarts its listening loop.

```
void *worker() {
    // Worker loop
    while (1) {
        sem_wait(&full);                // Wait for a spot in the buffer to fill
        pthread_mutex_lock(&mutex);     // Lock the mutex

        /* DO SOMETHING */
        int fd = buffer[out];           // Get the fd from the buffer
        out = (out + 1) % MAX_REQUEST;  // Move the out buffer pointer forward 1

        pthread_mutex_unlock(&mutex);   // Unlock the mutex
        sem_post(&empty);               // Signal there is a new empty space in buffer

        process(fd);                    // Process the fd's request
    } // while

    return;
} // worker
```

I then added the worker function. This is the function that each worker thread is assigned to execute. The first thing each thread does is wait for a spot in the buffer to fill. As soon as a spot fills up, one of the worker threads will claim it. It then waits for the mutex to be free so that it can have exclusive access to the buffer. The thread then increments the “out” buffer pointer so that the next thread will process the next filled buffer slot. It unlocks the mutex so that another thread can have access to the buffer. It also signals empty to indicate that there is a new empty slot in the buffer now.

After it is done fetching the file descriptor from the buffer, the worker can now process it. This happens outside of the semaphores since there is no race condition here.

The reason we are allowed to use 2 separate “in” and “out” buffer pointers without having to worry about the “out” moving past the “in” is because of the semaphores. The semaphores make sure that “out” is incremented no more times than “in.” Therefore, the buffer logic will not result in a worker reading a spot in the buffer which has not been filled.

```c
void threadMonitor(pthread_t workers[], pthread_t *listenerThread) {
    while (1) {
        // Check each of the worker threads
        for (int i = 0; i < numThread; i++) {
            if (pthread_kill(workers[i], 0) != 0) {
                printf("Error: Worker thread %i died! Restarting thread %i...\n", i+1, i+1);
                while (pthread_create(&workers[i], NULL, worker, NULL) != 0) {
                    printf("Error restarting thread %i, trying again...\n", i+1);
                } // while
            } // if
        } // for

        // Check the listener thread
        if (pthread_kill(*listenerThread, 0) != 0) {
            printf("Error: Listener thread died! Restarting thread...\n");
            while (pthread_create(listenerThread, NULL, listener, NULL) != 0) {
                printf("Error restarting listener thread, trying again...\n");
            } // while
        } // if

        // Sleep
        sleep(1);
    } // while
} // threadMonitor
```

The threadMonitor function is run by the main thread as soon as the other threads have been dispatched. It enters a continuous while loop. Each time this loop is executed, it checks on the status of each thread using pthread_kill. Since 0 is supplied as a parameter, all this does is check on the status of the thread. If pthread_kill returns a non-zero value, indicating thread failure, it attempts to restart that thread. After it has checked each worker thread, it then checks the status of the listener thread. If the listener thread has crashed, it attempts to restart it also.

Once it has checked the status of every other thread, the main thread then sleeps for 1 second. This is to prevent performance degradation from unnecessarily checking the status of all of the threads too often.

Performance Comparison:

For the performance comparison, I ran each test on Odin. The client sends 100 requests to the web server each time. The time indicated is according to the time displayed by the client.c program.

Number of web server worker threads	Time (seconds) to process 100 requests
1	100.043588
25	4.004438
50	2.009073

75	2.003382
90	2.003007
100	1.009321

As shown by the performance table, as the number of threads increases, performance is indeed increased as well. The most surprising part to me was that the speed seemed to cap out at 2 seconds until 100 threads were allocated.