Hayden Kendall

Prof. Jean-Yves Hervé

CSC 412: Operating Systems and Networks

3 September 2020

Programming Assignment Five

Processes that are working together are often sharing a common storage that one can read or write. It is imperative that processes do not attempt to access a common storage location at the same time. This is what can produce a race condition and cause systems to operate incorrectly. In this assignment the objective was to use pthreads and mutex calls to manage shared resources and prevent race conditions in a simulation. The simulation focusing mainly on a visual user interface.

The simulation was composed of a 400 square grid occupied by some traveler nodes. These traveler nodes are simply threads derived from the C library that run simultaneously with one another. Each traveler may move around the grid at their desired distance within the constraints of their position. This is so they do not travel off the grid. Before each move however, the traveler must gather ink resources from their respective color. Naturally, these ink tanks run out and a traveler may only move until the tank is replenished. A resource manager and ink producing threads are included to solve the problem. To prevent a race condition, the resource manager determines whether a traveler may move based on the number of units per ink tank. The producer threads are working to create more ink and allow the simulation to continue.

When it came to deciding on the displacement code, I felt it best to allow as little functionality on the resource manager as possible. Computationally, I felt it more efficient to have the hub of all traveler threads be as quick and painless as possible. A thread should lock the resource manager and open it back up as quickly as it can. For code organization as well, I felt it made the most sense to allow travelers to handle their own displacement. The traveler displacement is solely determined by the traveler itself based on pseudo randomness. Within the constraints of the grid, the traveler can decide how far they want to move and in which direction. Travelers may choose their orientation once before moving to prevent sporadic orientation changes of the travelers. Distance is constantly changing.

Ink acquisition was where the resource manager would primarily function. Much like a brief interrupt from an older mouse, a signal is needed to allocate and claim resources from the CPU. When the signal is reached, no other process may attempt to use the CPU until opened. I implemented that same concept in this simulation. A thread must first claim the resource manager using the mutex call to prevent other threads from attempting to access it. The resource manager may then view the ink levels of each tank and either approve or reject the traveler's displacement inquiry. Once the answer is given to the traveler, the mutex opens back up and another traveler may attempt access. If the traveler is rejected, it simply generates a new displacement length and tries for approval once again. If approved, the traveler may move the amount it asked for in a single shot. This was done to prevent race conditions of travelers gathering resources and to keep travelers moving.

I found the third-party graphics library to be the most difficult of the learning objectives. At the start of the project, I was struggling with understanding how the graphical user interface is refreshed which resulted in problems with creating and joining threads. It needed to be understood that the interface would only refresh relative to the main thread which handled the graphics. Meaning that if the traveler threads were joined prior to the main thread finishing then the simulation could not redraw itself.

Once that was understood, I did have some trouble determining how to create travelers. My first instinct out of habit was to use object-oriented programming which is a concept that is relatively absent in C. Using the provided structs to store basic traveler info, I realized that these "objects" could be handle as function. A while loop is used in said function that repeats the same code so long as the traveler is alive given what the structs provide. From there, a traveler that has moved would have an old and new coordinate. The problem simply became a matter of calculating displacement and making sure each traveled grid was painted. Calling each function in a new thread allowed for individual travelers to be created.

As with many programs, limitations are present. In this simulation, it would be computationally beneficial to have travelers decide their length of displacement one time. If a traveler is rejected, it attempts to calculate a new distance and try the resource manager again for approval. An alternative solution would be to use the travelers in a priority queue of some sort like in many operating systems. I would likely do something along the lines of a priority level based on how long a traveler may be waiting for approval. Round-robin could also be used here for travelers of each distinctive color so that travelers move consistently.

As an optional assignment I take great pain in not taking enough time to complete it during my studies. With many learning objectives, they are often easiest when an individual is thoroughly invested. This was certainly a project which I could thrive from. I have sharpened my C programming skills and have gained a solid understanding of operating systems concepts at the level of this course. I think I am ready for the final project to demonstrate my new knowledge.