ELEC 377 Operating Systems F22

Lab 2

Due October 18th, 2022

By: Kevin Yu, Raatik Sharma

Student Numbers: 20203451, 20120770

Problem Statement

The objective for this lab is to simulate a distributed system monitor with multiple threads running in parallel using semaphores, which are synchronized, to access to shared data structures in a central monitor.

Main

In the main function, we first error-check the input arguments passed into the program and initialize the number of monitor threads and printer delay values. Next, we delete any semaphores if they exist with sem_unlink and loop through the machine_stats for each monitor thread and set its read value to 1 to indicate the data to be read. Afterwards, the default attributes for thread initialization are defined using pthread_attr_init(&thread_attr). A for-loop is then used to populate the machine_number and the shmeptr struct properties for each monitor in the monitor_threads array. Afterwards, the for-loop pthread_create function is called with its respective arguments to generate the threads for each monitor. The last section in the main loop repeats the same process for generating threads, but instead, we are generating threads for both the reader and the printer thread.

init_shared

A parameter called “shmemptr” is a pointer to the shared memory data and, once dereferenced, is used for accessing the struct data from the shared memory segment. The init_shared function, which includes a pointer to shared memory as an argument, first initializes the semaphores for access_stats, access_summary and mutex to the value of 1 for the critical section to only have 1 thread in at a time and then assigns the monitorCount equal to zero in the shared memory segment. If any of those semaphores fail to be initialized, the console will display an error message with the specified failed semaphore. The variables access stats, mutex, and access_summary are semaphores. The access_stats and mutex are used to synchronize access between the monitor threads and the summarizer thread. The access_summary semaphore is to synchronize access between the summarizer and the printer threads.

monitor_update_status_entry

Inside the monitor_update_status_entry, the machine_id, status_id, a pointer to cur_read_stat, and a pointer to shared memory are passed in as arguments into the function. Firstly, the function prints the struct data from the simulated machine to the console. Next up, we lock the mutex semaphore to prevent other monitor threads from accessing the monitorCount prevent race conditions. Then, we increment the monitorCount, and if monitorCount is equal to 1, we lock the access_stats semaphore to prevent the summarizer thread from accessing the shared memory but still allow multiple monitor threads. We then release the mutex lock to allow other monitor threads to access the monitorCount and read the shared memory segment. Afterwards, if the read struct property is equal to 0, a message to the console indicates the monitor data was overwritten. At the end of the function, the mutex semaphore is locked again, so other monitor threads can’t access monitorCount to cause a race condition. The monitorCount is then decremented and checked to see if the value is equal to zero; if so, the

access_stats semaphore is released, meaning the summarizer thread can access the shared memory segment. Lastly, the mutex is unlocked to allow other monitor threads to access monitorCount.

reader_thread

The reader_thread has a parameter called "parms," which is used to access the shared memory and the number of machines in the central monitor. In this function, a while loop is continually checked to see if there are any more monitors still running. Inside -the while loop, we do a threadLog and then we lock the access_stats semaphore to prevent monitors threads from accessing the shared memory segment. We then store local read_update_times and read_machines_state for each machine in the shared memory. An if statement then checks to see if a machine -is down by checking the machine's state property for 0. Afterwards, statistical data for each machine is accumulated and stored in their respective variables, and the machine's "read" value is set to 1, meaning the data has been read. Next, each update requires a generation of a checksum; therefore, the gen_checksum_seed and the gen_summary_checksum functions accomplish that task. In addition, we also update each machine's state and last updated properties with a for-loop with the local read_machines_state and the local read_update_times arrays. We also update the machines_online_since property by checking to see if the machine has first turned on for the first time by checking the current state of the machine with its previous state. The statistical data is then logged to the console, and new averages are calculated for each statistical property. We then release the access_summary semaphore to allow monitor threads access to the shared memory, and we use a for-loop to set the "last_machines_state" to the current "read_machine_state" for the next iteration. Lastly, we check if any monitors are still running, and if not, we set "more_updates" equal to zero to exit the while loop and log a message to the console saying the reader thread loop has ended.

printer_thread

The printer_thread function has a parameter called “parms,” which is used to access the shared memory, print_period, and num_machines. The while loop checks whether the monitors are still running by checking the local variable more_updates. Inside the while loop, the code sleeps for a specified number of milliseconds that was passed as an argument. Then locks the access_summary semaphore to prevent the reader thread from accessing the shared memory. Afterwards, a for loop prints a summary of each machine’s properties such as state, up time and the last updated. Afterwards, the access_summary semaphore is released so the reader thread can access the shared memory again. Lastly, an if statement checks to see if the number of monitors in the shared memory is zero and if so, more_updates is set to zero and the while loop exits, and the function is terminated successfully.