Ayman Shehzad Awal
Tejas Chadha Marimuthu

Project 1 Report

FCFS

cpu_me()

In the scheduler.c we implemented a thread queue in the form of a linked list, with each element being a struct with data such as current time, tid, remaining time. We also implemented an array thread conditions associated with specific tids.

In the cpu_me function when the threads arrive we will mutex lock, and check if the arrived thread is present in the queue or not using the checkExistenceCPU(tid); we pass the tid as an argument to uniquely identify the threads. If the function returns 0, we know that the thread does not exist in the queue, therefore the thread is able to enter the if statement.

In the beginning of the code we created a global variable called arrivedCPU_threads, initially assigned to 0. This value will now be incremented by 1 since we have a new thread. We will add this thread to the queue using the enqueueCPU() function, which accepts the current_time, tid and remaining_time as arguments. In the enqueueCPU() function a new instance of the node struct is created and memory is dynamically allocated for that using malloc. Also another thing to point out, since FCFS is all about the threads getting priority based on arrival time, the queue automatically gets sorted based on current_time with the help of the sortCPU() function after we enqueue.

In the init_scheduler function we will call createCondList() function and pass thread_count as an argument to allocate enough memory for two conditional arrays. Going back to cpu_me, we have a while loop statement to check if all the threads have arrived or not by comparing arrivedCPU_threads and total_threads. If all the threads have not arrived yet, we will make the thread wait conditionally by calling pthread_cond_wait(), by passing the arguments of &cpuLock and getCPUThreadCond(). This function takes the argument of tid, and returns the thread condition of that specific thread from the condList. So this process, i.e the while loop, will keep running till all the threads have arrived.

Once all the threads have arrived, it will skip the while loop and signal the head of the queue and mutex unlock it. When the thread exits the critical section, we check if the remaining time of the current tid is 0, if so we will dequeue it using either dequeue() or specificDequeueCPU(). Otherwise if the remaining_time is not 0 we will compare the current_time to the global_time. If current_time is less than global_time then we will perform the ceiling operation on global_time and increment it by 1. Otherwise, we will do ceiling operation on the current_time + 1. Every iteration we will decrement the remaining_time and eventually return the global_time.

io_me()

For io_me function, we will begin by mutex locking the thread and increment arrivedIO_threads, which keeps track of all the threads that have arrived in IO function. After that we will enqueue the thread into the IOqueue designed for IO by passing the arguments current_time and tid in enqueueIO() function. When we call enqueueIO() function, it will also create a condition for that specific thread in an array of conditions initialized in the scheduler class called condListIO. Once the thread is enqueued, we check if the first thread in the queue is the current thread. If it is not the current thread, then we wait for our turn. If it is, then the thread exits the mutex lock and performs a similar operation to cpu me, where it compares the current time and global time and properly adds the duration to it. After this, the thread then dequeues itself and signals the next thread in the IO queue if there is another thread. Finally, the thread returns the time it finished in IO.

p() & v()

For the P() function we will begin by mutex locking the thread that arrives using the pthread_mutex_lock(&pLock). After that we will use the function enqueuePV() using the arguments tid and sem_id. After that we will instantly use the decrementSem() function to decrement the semaphore by 1. Initially semaphores are set to 0 and will become -1. We then proceed into the while loop with the condition which checks whether the semaphore of the given sem_id is less than 0 or not. It uses the getSem(sem_id) function which returns the semaphore. The thread will be conditionally waited using pthread_cond_wait(getPVThreadCond(tid), &pLock). Once it breaks out of the while loop the thread will be mutex unlocked and dequeue it from the queue. We will end up returning the global_time.

For the V() function we begin by mutex locking the thread that arrives using the pthread_mutex_lock(&vLock). After that we will instantly increment the semaphore by incrementSem(sem_id) function. If the semaphore value, retrieved by the getSem(sem_id) function, is less than or equal to 0, we will use pthread_cond_signal(getPVThreadCond(tid)) to conditionally signal the waiting thread. After the thread breaks out of the while loop, the thread will get mutex unlocked using pthread_mutex_unlock(&vLock) and return global_time.

Incompleted Implementations:

- srtf and mlfq policies incompleted
- Incompleted E() function
- Incompleted p() and v() functions
    - The proper idea we have created but we were not able to complete it in time.

Distribution of Workload:

- Majority of workload was split in half. We both equally contributed on working on creating each of the functions and data structures. Both parties put in a many hours to get to this point of the project, despite starting very early.