David Rohweder

CS473

10/10/2022

473 CPU Scheduler

When approaching the design of the scheduler there were three main components that needed to be considered. The first component was analyzing how synchronization between the two devices would occur. Secondly, I needed to consider how threads would interact and maintain a consistent data state as data was changing, and threads were stagnant or moving in and out of the interface functions. Lastly, I needed to design a clean, robust scheduler that would be separate entirely from the interface, such that no changes needed to be made in the interface upon a change of schedule type.

Synchronization between the two devices has proven to be the most challenging aspect of this project. Several layers of redundancy were needed to ensure that race conditions, deadlocks, and random scheduling time would not interfere with the data coherence of the program. To ensure I maintained this consistency such precautions I implemented included: multiple barriers, condition variables, and a single source of truth with my mutex. There were also several states that the program could be in at a given instant based on the native execution of the threads. Such states include, but are not limited to, IO target time not met and CPU leaving the function, CPU, and IO both leaving the interface, and threads in CPU and IO but CPU threads not "arrived" yet, so CPU needs to artificially increment global time and check IO again, nothing in CPU so we need to use the IO as the global time setter, and several others. Also, to help maintain consistency I used more so of a linked list approach rather than a pure queue. This is because I wanted my data structure to keep track of important information such as if the thread is in IO or frozen when I am considering which thread to pick in my scheduler. Before any of the threads were allowed to execute, I had a specific condition variable that kept each thread waiting until a counter incremented until all threads were counted for and that execution could commence via a broadcast. Also, to help keep track of the states of my threads in relation to the scheduler and the CPU I had an Enum marking if a thread was either bursting, waiting, or had expired (no more bursts left). This allowed me to communicate to the IO device and the scheduler if there were any threads in the CPU (in burst or in wait) or to the scheduler to not consider expired threads. When a new Cx command is initiated, the thread would be updated with the new duration and arrival time.

The communication between the devices proved to be tricky at times. Although I had multiple barriers in place, the interleaving of scheduled threads would often interfere with which thread was where at a given instant. Therefore, additionally, I needed the usage of several Boolean variables to ensure that when I was at a synchronization point that it was warranted, and the other thread was also ready to synchronize. If one thread was ready to synchronize, but the other was stuck, or interfered with, then a deadlock would be incurred.

Tying into the communication between the devices encompasses threads within the functions that are waiting for the executing threads to leave to decide whether they can run or not. Moreover, I needed to use two condition variables to have threads wait inside of these functions if they were not the appropriate threads according to the scheduling policy. If the threads were not the chosen best for the scheduler, they were put in a wait state and tried to see if they were the best upon mutex unlock and broadcast of that condition variable.

Another difficult obstacle that I encountered was maintaining the state of the program when threads were leaving functions. This made making sure that the native scheduler did not push them on and on without letting the other ones go according to how they should be executed challenging. To fix this issue I set up certain synchronization barriers such that when two threads both left a function at the same time such as a P and V or IO and CPU both exiting that they had to synchronize in whatever next function first before any other execution could occur. Furthermore, I had an array of behavior states such that any code executing when the others leave would not be allowed to since I marked states dirty upon function exit. Moreover, I had a function that would go through all thread states and if any were false or undetermined then waiting for points such as CPU lowest or IO lowest or IO time sync would not continue until all states were determined.

The last part of the project was the easiest to implement and was the scheduling aspect. Once I had built my interface, I would easily have a single return point from the scheduler that would return the best thread according to the specified policy. Upon initialization, I keep a schedule type variable and when the CPU hits the lowest CPU arrival time function, I do a switch on the schedule type and then go into whichever policy function and return the thread that best fits that policy. This ensures that there is a single source of truth for where the interface gets its best thread from to help reduce complexity and fail points in the interface. The FCFS was a relatively easy function to implement. Moreover, the MLFQ was simply the FCFS with an additional loop on the outside and if no threads were found at priority 1 then it would increment to the next and the next until priority level 5. Additionally, I maintained a quantum value in the threads data structure and whenever I decremented the duration of the thread, I incremented the quantum of the thread. If the quantum of the thread was greater than the value of the quantum at the MLFQ_TIME_QUANTUM[current priority] then I would increase the priority of the thread in the data structure. Next, the SRTF function was slightly trickier to implement; however, once I figured out that I was also considering threads that did not arrive, my code worked perfectly. Likewise for the scheduling of the CPU I had a dedicated shortest IO function that served as the FCFS for the IO queue.

As there were two devices, I split up the inflow of execution on these devices into two queues the CPU and IO queues respectively. The main heartbeat of this scheduler was the CPU interface. That is why I maintained threads in circulation in the queue even after an expiration on the final tick and only removed a thread upon an end request. Otherwise, the CPU threads would maintain information such as if they were frozen from a P function and what sem_id froze them, and what value of sem_id they were frozen at. To keep the program of scale I decided to reuse the PCB data structure which has held all these values mentioned above (priority, frozen, frozen

David Rohweder

CS473

10/10/2022

by, frozen at, in IO, thread state, duration, arrival time, a thread id) for the IO queue as it still had the necessary components needed for usage, but some extra unneeded values.

In conclusion, I learned a lot about synchronization and deadlocks from this project. Moreover, I learned about some new features in C such as mutexes, semaphores, condition variables, and barriers. I faced numerous obstacles, but with some investigation, I was able to fix and remediate them for the final product.