CSCE 611 Operating Systems Summer 2017

825008857 Chaiwei Chang

MP5

Design for Scheduler

Originally, professor provided the source code that with threading and all the context switching for dispatching. However, thread knows nothing about scheduling. So, we are asked to write a scheduler starting with the easiest FIFO scheduler. Firstly, I created a global static scheduler object with data structure of linked thread list to represent our ready. Additionally, I implemented this list structure with its thread and its next ready thread. Here comes how do I design all the required method, for the virtual void yield(), I took the first element in the linked list that stands for the next thread that we will execute so as the ask the kernel to dispatch it. And then I pop that element from the list and asked the CPU to run it. For the resume and add, I implemented them exactly the same which is append it to the end of the list. So, resume doesn't resume execution just back into the end of ready queue. I also record the size of the list to leverage the terminate function. Referring to the terminate function, I have come up with two algorithms: one is to use two pointer prev and curr, as soon as we find curr->ThreaId equal to the Id that we want to terminate we will reorganize the list pointer that prev-> next= cur->next; Second one is that we pop every thread out if it is not match we will put it back, on the other hands we will delete it. I choose to use the second one since I don't have to rewrite any method just calling previous API will be done. By the delete I just mentioned above, I meant that knowing that it is ready to kill but only take it out of queue and finish every critical part first, then clean up every memory related to it a little bit latter. It is easy when thread terminate be called by other thread, just traverse through ready queue and delete it. However, Thread suicide need to give CPU resource out before it dies. Secondly, for the yield() function I have something more to say, if there is no more thread inside ready queue but one thread still want to give up the CPU, I would create an idle looping thread to go run when nobody running on CPU. Finally, what happened inside the kernel is that thread 1234 will do the relay before thread 1 and thread 2 finish and terminate themselves and only thread 3 4 will play the pingpung by switching back and force the CPU. Last but not least, I have also done bonus option1 and option2. I finish the option1 by enable the interrupt in the start thread method. As for option2, I modified the handler to inform the control of the interrupt is being handled before the call to the handler so that we are allow context switch during handler process. Furthermore, in the timer file we use the ticks to calculate the time, we will encounter interrupt ever 50msec. And SYSTEM_SCHEDULER which is a global scheduler will preempt the ongoing thread so that inside of the thread function such as func0 func1 we no longer need to call pass_on_CPU(thread2);. Lastly, some code need to be commented out or uncomment in but not limit to kernel.C, interrupt.C and simple_timer.C. All of my output file of console print are recorded in out.out. I wrote some comment on my code, but if grader of TA have any question about my logic feel free to contact me at hunkywei@tamu.edu