Design notes

Mian Qin UIN:725006574

1 Overview

In this machine problem, we mainly implement a simple **FIFO scheduling algorithm** for kernel threads. I also finished the **bonus option1 that fix the interrupt management so that interrupts remain enabled outside of critical** sections**.**

In this FIFO scheduling algorithm, the thread function need to explicitly call the resume() and yield() function of the scheduler to give up the CPU and let the scheduler schedule another thread to run. The core design of the FIFO scheduling is a double linked-list data structure that works as a FIFO queue. Every time a new thread is added to a scheduler or a current thread wants to give up CPU, it will add a thread node to the **front** ready queue (double linked-list). Then, we the scheduler calls yield(), it will grab a thread node from the **back** of the ready queue, and switch to that thread context.

For the thread termination implementation, it's a little bit tricky since we have to clean up the thread resources (the stack and the Thread class instance). When a thread returns from the thread function, it will call the thread_shutdown() since it is pushed into the stack when the thread is created. In the thread_shutdown() it will call the scheduler's terminate() function to do the thread clean up. In order to interact with scheduler, the thread instance must register with the scheduler. I did this by adding a **static Scheduler pointer** in the Thread class, the first time we add a thread into the scheduler, it will call register_scheduler() function in the Thread class (which is also static) to keep record of scheduler in the Thread class. In the Scheduler terminate() function, we cannot directly clean up the thread since we are still in the current thread context. If we delete the current thread stack we lose the current context and the system will hang. So, in the Scheduler class, I keep another list (single linked-list) to keep record of the thread that are going to terminate, i.e. need to be cleaned up. In the terminate() function, I will put the current thread in to this clean up list and call yield() which will do a thread switch and switch to another thread's context. The actual thread clean up happens in the yield() function, but only when the need to be cleaned thread is not the current thread. Then, we can safely terminate and clean up threads.

For fixing the interrupt management, it happens in the yield() function. Since the interrupted must be disabled when we do the thread context switch in the **threads_low_switch_to().** What I did is, in the Scheduler class yield() function, before calling dispatch_to(), I will check whether interrupt is enabled and if so disabled interrupt. And when the new thread context return from the yield() after the dispatch_to() call, I will re-enable interrupt. After doing so, the timer "One second has passed" prints come back.

2 Implementation

2.1 Scheduler

The Scheduler class maintain two list, one double linked-list for FIFO queue for scheduling and one single linked-list for the terminated threads. It will keep the head and tail of the

FIFO queue and the head of the cleanup queue.

2.1.1 Scheduler::Scheduler ()

In Scheduler constructor, it will initialize the head/ tail of the two linked-list.

2.1.2 Scheduler::yield()

For the yield() function, it need to do two jobs. First it will scan the clean up queue that stores the terminated threads and as long as is not in the terminated threads' context, it will perform the clean up (delete the Thread class instance, release the thread stack memory).

Then, it will do the scheduling job, by grab the tail node of the FIFO queue and switch to the thread in the tail of the FIFO queue. Also, it need to delete the thread node in the FIFO queue to prevent memory leakage.

2.1.3 Scheduler::add(Thread * _thread)

For the add function, it will first check whether the Thread class has registered the scheduler, if not, it will pass the scheduler pointer to the static member of the Thread class for termination use.

Then it will put the _thread information to the front of the FIFO queue, basically perform a insert to head of the double linked-list.

2.1.4 Scheduler::resume(Thread * _thread)

For resume() basically same will add(), here directly call add().

2.1.5 Scheduler::terminate(Thread * _thread)

In the terminate() function, since it's called by the thread_shutdown() function in the terminated thread's context, it cannot directly release the thread memory resources. So, it will put the thread information into the clean up list. And the call the yield() to do a thread switch to release the terminated thread in other threads' context. (Done in the yield() call)

2.2 Thread

For the Thread class, it need some small change to make it work for thread termination.

1) In order to interact with the Scheduler and call terminate() function in the thread_shutdown(). In the Thread class I add a static member of Scheduler pointer and registered the scheduler we first add a thread.
2) Add a destructor function for the Thread class to delete the stack memory segment. And when we delete the Thread instance it will automatically release the stack memory resource.