Cameron Bourque UIN: 826004886

Options:

I added in option one where I manage interrupts.

Design:

Thread:

For the thread class I started by adding a public pointer to the next thread in the ready queue. I left the rest of the header file as is. Then, in the C file, I added an extern to the SYSTEM_SCHEDULER defined in kernel.C.

I left the push, CurrentThread, ThreadID, and dispatch_to functions as they were.

For the constructor, I made sure to make the next variable I added set to null. The rest I left as is.

For the setup context function, I changed the push for the EFLAGS to 0x200 instead of 0 to enable interrupts. The rest I left as is.

For the thread start function, I check if interrupts are enabled. If not then I enable them. Otherwise nothing happens.

For the thread shutdown function, I use the externed scheduler to call terminate on the current thread. Then I assert false.

Scheduler:

For the scheduler class I added 2 private pointers for the beginning and end of the ready queue. I also added a private thread pointer for the idle thread and a private char pointer for its stack. I left the rest of the header as is. In the C file, I added an extern to the SYSTEM_SCHEDULER defined in kernel.C and made a function called idle.

The idle function I made in the C file is for the idle thread and loops forever and calls the yield function using the SYSTEM_SCHEDULER.

For the scheduler constructor I start by allocating the idle thread's stack and the thread so that it uses the stack and runs the idle function. Then I set up the ready queue by making it empty.

For the yield function, I check if interrupts are enabled. If so then I disable them. Then I check what the current thread is. If it is the idle thread and the ready queue is empty then interrupts are reenabled and nothing further happens. Otherwise, if the queue is empty then I fill it with the current thread and dispatch to the idle thread and reenable the interrupts. Otherwise I put the current thread to the back of the ready queue and pop the front one off and dispatch it. Then I reenable interrupts.

For the resume function, I check if interrupts are enabled. If so then I disable them. Then I check if my queue is empty. If so then I fill the queue with the provided thread. Otherwise I place the thread at the back of the queue. Then I set the thread's next to null. Finally I reenable interrupts.

For the add function, I check if interrupts are enabled. If so then I disable them. Then I call the resume function.

For the terminate function, I check if interrupts are enabled. If so then I disable them. Then, if

the thread we are given is the current thread, we get the next thread on the ready queue and pop it off. Then we make sure the old thread is not placed back on the queue and dispatch the popped off thread. Then we reenable interrupts and return. Otherwise, if the thread is at the beginning of the queue, we just shift the beginning of the queue to the next thread. Otherwise we search the list for the thread whose next is the given thread. Then we move its next to the given thread's next. At the end we just set the given thread's next to null and then reenable interrupts.

FIFO:

The fifo scheduler is implemented based on a queue which pops off the beginning thread when trying to dispatch a new thread and moves incoming threads onto the back of the queue.