Design for MP4

1. Implementation of scheduler

1.1. FIFO queue

First of all, we implement a simple FIFO algorithm for thread scheduling.

Here, we noticed that a linked list approach should be difficult and resource consuming. The reason is that to achieve that we not only need to overload the new operator but also every thread pointer will occupy one frame of memory. So we do not support FIFO mechanism by any dynamic management.

We use array to store pointers to the thread object for simplicity. We also noticed that one frame of memory can contain exactly 1024 pointers — therefore 1024 the maximum number of thread we can support here. (We could set any number we want, as long as the number is fixed)

To achieve FIFO, we simply use two index to record the head and the tail of the queue.

1.2. Relative operations

Here we apply several operations.

In resume function, we put the given thread on the tail of the ready queue.

In add function, we add the given thread on the tail of the ready queue.

In yield function, we first select the thread that is on the head of the ready queue. Then we do the context switch between this thread and the current thread. By doing this, yield function calls dispatch_to function, which further calls threads_low_switch_to function. The last function is an assembly function and in this function we do the following: we modify the function stack into an interrupt stack and then return from the interrupt. The reason to do this is that we can simply switch context — register values, CS, EFLAGS, thread functions, and so on. Note that the new thread will not return from the same function, it starts from where it ought to — calling start_thread function if this is the first time it occupies CPU, or calling thread function or shut_down function if that is where it gives up CPU some time earlier.

2. Implementation of thread termination mechanism

When uncomment _TERMINATING_FUNCTIONS_ in kernel.C, thread 1 and thread 2 will no longer be eternally-executed threads. After they finish the thread function, say j reaches 10, the thread function pointer will pop out from the thread stack and then the shut_down function will be executed.

In shut_down function, the thread that is to die let the scheduler know the fact and let the scheduler handle this properly by calling terminate function of the scheduler. In the terminate function, what the scheduler does is simple — it calls yield function to do the context switch without resume the dying thread.

However, the problem is that the memory occupied by the terminated thread has not been released yet. The problem is that the thread that is to terminate cannot release its resources by itself. Our approach is to release these resources by the next thread.

The resources need to be released is the memory used for (1) thread stack and (2) the thread object. We use two global variable to record the address of the two region. Also, we set another global flag to indicate whether there are resources that need to be cleaned. When a thread is about to terminate, it set the flag and then pass its stack pointer and the pointer to itself to the global variables, and then let the scheduler to the context switch.

Now we consider how to clean them. Theoretically, the next thread can only return from two places: (1) if this is a new thread, the very top of its stack will be executed; (2) if this is a running thread, it returns from yield function from where it was preempted earlier. Since we cannot modify thread function itself, we put the cleaning code at both thread_start function and yield function. The job of the cleaning code is to release those frames recoded by global variables and set the flag to be false.

3. Bonus Options

3.1. Bonus Option 1 — Correct handling of interrupts

Now we want to let interrupts work.

Interrupts must be disabled during some operations of the scheduler like resume, yield and add, since we do not want these critical sections to be interrupted. However, during any thread is running, we do want to let the interrupt to be activated.

The solution is that whenever a scheduler operation is about to execute, we disable interrupt.

The moment a thread begins running is the moment interrupt is enabled. As is illustrated above, a thread can only begin from thread_start function or yield function; therefore we add corresponding code in these places.

3.2. Bonus Option 2 — Round-Robin scheduling

From experiment, the required time interval — 50ms — is too long for a thread. In thread function, it ticks 10 times and then gives up CPU. This process can be done in less than 50ms. Therefore, if we insist on setting time interval to 50ms, the thread will give up CPU voluntarily before the round-robin mechanism performs. So here we set the time interval to 20ms.

We also need to modify the timer. Since every time a new thread occupies CPU the timer needs to be reset, therefore a function reset is added in the timer. In this function, we simply clear the tick number. Also, when tick number satisfies the given threshold, say one time interval passed, an interrupt takes place. In interrupt handler, it looks up its register table, find the corresponding item, and calls the handler function of that item. Therefore, we modify the handler function of the timer — simply resume followed by yield. After that, the interrupt handler returns, EOI signal will send to PIC.

The problem is that the new thread may not return from an interrupt handler, meaning that the EOI signal may not be sent in time. What is worse, when the preempted thread regains CPU much later, it returns from interrupt and send an EOI signal that confuses PIC. To avoid that, we move the EOI sending part prior to the interrupt handler. By doing this, we claimed that whenever an interrupt is “ready” to call interrupt handler, the interrupt reaches its end. This somehow makes sense: the interrupt only belongs to the current thread, after we finish context switch, the interrupt already has nothing to do with the new thread.