MP5: Simple Kernel Threads

Jackson Stewart
329009382
CSCE 410

Assigned Tasks
Main: done
Bonus Options: not attempted

MP5: Simple Kernel Threads

Jackson Stewart
329009382
CSCE 410

Scheduler Definition (scheduler.H): defines helper utilities (ThreadNode and queue) for use in Scheduler implementation

```
/*--------------------------------------------------------------------------*/
/* SCHEDULER */
/*--------------------------------------------------------------------------*/

You, 2 hours ago | 1 author (You)
struct ThreadNode {
    Thread* thread;
    ThreadNode* next;

    ThreadNode(Thread* _thread) {
        thread = _thread;
        next = NULL;
    }
};

You, 2 hours ago | 1 author (You)
class queue {
    ThreadNode* head;
    ThreadNode* tail;

   public:
    queue();
    void add(Thread* _thread);
    Thread* pop();
    void remove(Thread* _thread);
    bool isEmpty();
};

You, 2 hours ago | 1 author (You)
class Scheduler {
    /* The scheduler may need private members... */
    queue readyQueue;
    Thread* currentThread;

   public:
    Scheduler();
    /* Setup the scheduler. This sets up the ready queue, for example.
       If the scheduler implements some sort of round-robin scheme, then the
       end_of_quantum handler is installed in the constructor as well. */

    /* NOTE: We are making all functions virtual. This may come in handy when
             you want to derive RRScheduler from this class. */

    virtual void yield();
    /* Called by the currently running thread in order to give up the CPU.
       The scheduler selects the next thread from the ready queue to load onto
       the CPU, and calls the dispatcher function defined in 'Thread.H' to
       do the context switch. */

    virtual void resume(Thread* _thread);
    /* Add the given thread to the ready queue of the scheduler. This is called
       for threads that were waiting for an event to happen, or that have
       to give up the CPU in response to a preemption. */

    virtual void add(Thread* _thread);
    /* Make the given thread runnable by the scheduler. This function is called
       after thread creation. Depending on implementation, this function may
       just add the thread to the ready queue, using 'resume'. */

    virtual void terminate(Thread* _thread);
    /* Remove the given thread from the scheduler in preparation for destruction
       of the thread.
       Graciously handle the case where the thread wants to terminate itself.*/
};
```

MP5: Simple Kernel Threads

Jackson Stewart
329009382
CSCE 410

Scheduler Implementation (scheduler.C): The queue helper class is implemented first…

```
queue::queue() {
    head = nullptr;
    tail = nullptr;
}

void queue::add(Thread* _thread) {
    ThreadNode* newNode = new ThreadNode(_thread);

    if (head == nullptr) {
        head = newNode;
        tail = newNode;
    } else {
        tail->next = newNode;
        tail = newNode;
    }
}

Thread* queue::pop() {
    if (head == nullptr) {
        return nullptr;
    }

    Thread* _thread = head->thread;
    ThreadNode* temp = head;

    head = head->next;
    if (head == nullptr) {
        tail = nullptr;
    }

    delete temp;
    return _thread;
}

void queue::remove(Thread* _thread) {
    ThreadNode* current = head;
    ThreadNode* previous = nullptr;

    while (current != nullptr) {
        if (current->thread == _thread) {
            if (previous == nullptr) {
                head = current->next;
            } else {
                previous->next = current->next;
            }

            if (current == tail) {
                tail = previous;
            }

            delete current;
            return;
        }

        previous = current;
        current = current->next;
    }
}

bool queue::isEmpty() {
    return head == nullptr;
}
```

Jackson Stewart
329009382
CSCE 410

… followed by the Scheduler class itself using the queue class:

```
Scheduler::Scheduler() {
    currentThread = nullptr;
}

void Scheduler::yield() {
    if (!readyQueue.isEmpty()) {
        currentThread = readyQueue.pop();
        Thread::dispatch_to(currentThread);
    }
}

void Scheduler::resume(Thread* _thread) {
    readyQueue.add(_thread);
}

void Scheduler::add(Thread* _thread) {
    readyQueue.add(_thread);
}

void Scheduler::terminate(Thread* _thread) {
    readyQueue.remove(_thread);
}
```

MP5: Simple Kernel Threads

Jackson Stewart
329009382
CSCE 410

Thread changes (thread.C): Implemented the thread_shutdown function

- (Uses a static scheduler variable (set in kernel.C) to access it)
- Releases the current thread
- Terminates current thread in the scheduler (removes it from the queue)
- Yields to the scheduler (to resume the next thread in the queue and continue as normal)

```
static void thread_shutdown() {
    /* This function should be called when the thread returns from the thread function.
       It terminates the thread by releasing memory and any other resources held by the thread.
       This is a bit complicated because the thread termination interacts with the scheduler.
     */
    assert(current_thread != 0);

    current_thread->Release();
    Console::puts("Thread ");
    Console::puti(current_thread->ThreadId());
    Console::puts(" is terminating.\n");

    Scheduler *scheduler = current_thread->get_scheduler();
    assert(scheduler != nullptr);
    scheduler->terminate(current_thread);

    Console::puts("Thread ");
    Console::puti(current_thread->ThreadId());
    Console::puts(" is terminated.\n");

    scheduler->yield();

    Console::puts("Thread ");
    Console::puti(current_thread->ThreadId());
    Console::puts(" is terminated and yielded.\n");
}
```

TESTING

I relied on the test framework provided in kernel.C, which revealed a bug along the way that took a while to figure out (I wasn't quite yielding correctly). I added some extra print statements along the way as I was debugging.