

CSE 312

Operating Systems

Homework - 1

Muharrem Ozan Yeşiller
171044033

Problem Definition

Our problem is to design a library for multi-threading support for a base operating system kernel written specifically for the x86 processor architecture.

While these threads must be scheduled, they must also provide `create()`, `yield()`, `terminate()` and `join()` methods.

Another requirement is that 2 threads, a producer and a consumer, synchronously block each other (communicate) in critical areas with the Peterson's Solution.

Design Decisions

In terms of design, the thread class that I have implemented contains some data and some functionality. In the kernel version we have developed, there is a class developed for schedule called Task Manager. Thread class is friends with this class. In C++, friendship is not received, it is given. That's why Thread class gives friendship to Task Manager class because Task Manager needs to update some things about Thread class. If the setter were made with getter, everyone using Thread would have access to these variables, so it would be an insecure design.

I also made changes in the Interrupt and Task Manager class. Unhandled exceptions within the Interrupt class are hardware related exceptions. If a thread terminates its job and still continues in scheduling, the program counter will show a completely different place in memory and we will receive hardware exceptions. Since our responsibility in this assignment is only threads, only threads can create hardware exceptions in terms of memory. I handled this, if a thread gets an unhandled exception when it is scheduled, I understand that it is finished and I terminate it.

```

else if(interrupt != hardwareInterruptOffset)
{
    if (taskManager -> RemoveCurrentThread() == false)
    {
        printf("UNHANDLED INTERRUPT 0x");
        printfHex(interrupt);
        printf("\n");
    }
    else {
        esp = (uint32_t)taskManager->Schedule((CPUState*)esp,
true);
    }
}

```

On the task manager side, I kept a CPUState variable, which represents the empty CPU. I create a sample object from an empty CPU before scheduling starts and keep it so that when there are no threads in the middle, I need to empty the CPU. If this is not provided, when the threads are finished, the remnants of the last thread will be in the CPU and we do not want this. I implemented methods in which threads are added and removed. These methods are used to delete elements from the array that Task Manager keeps and add elements.

```

class TaskManager
{
private:
    CPUState* empty_cpustate;
    Thread* threads[256];
    int thread_size;
    int current_thread_idx;
    int testVar;
    int threadIDCounter;
    bool firstInterrupt;

```

```

public:
    TaskManager();
    ~TaskManager();
    bool AddThread(Thread* thread);
    bool RemoveThread(Thread* thread);
    bool RemoveCurrentThread();
    int getTestVar();
    CPUState* Schedule(CPUState* cpustate, bool unhandled);
}

```

I made changes to the Schedule method in the Task Manager. I will explain this in the area where I explained the schedule algorithm.

Main Thread Structure

```

class Thread {
    friend class TaskManager;
private:
    THREAD_STATE state;
    TaskManager* taskManager;
    common::uint8_t stack[4096]; // 4 KiB
    CPUState* cpustate;
    int ID;
public:
    Thread();
    ~Thread();
    bool CreateThread(GlobalDescriptorTable *gdt, TaskManager*
taskManager, void entrypoint());

    bool TerminateThread();
    bool YieldThread();
    bool JoinThread();
    int getID();
    THREAD_STATE getState(); };

```

- 1) `THREAD_STATE` is an enum and holds the current state of the thread.

```
enum THREAD_STATE { NEW, RUNNABLE, RUNNING, DEAD };
```

- 2) Task Manager is a class that schedules and manages threads. The reason why I use it in threads here is to benefit from the Observer Design Pattern principle. Started threads subscribe themselves to Task Manager. If the thread wants to terminate, it can unsubscribe from Task Manager. At the same time, since Task Manager will have Threads, when there is an update, it updates Threads directly by reference. (These changes are the instant stack and cpu state of the thread.)
- 3) Stack represents the address of the thread's own stack. Cpu State, on the other hand, refers to the current state of the registers while the thread is running on the CPU, in case each thread yields, these registers should be backed up. This part will be explained in the "scheduling" heading.
- 4) ID refers to the unique identity of each thread.
- 5) `CreateThread()`: The Task Manager registers the information that Thread has, and to some extent, the subject subscribes to the observer. So now Task Manager will be able to run this thread while scheduling.
- 6) `TerminateThread()`: DEAD the state of the thread and this thread is deleted when scheduling runs. (Scheduling is also mentioned.
- 7) `YieldThread()`: Thread is momentarily switched from `RUNNING` to `RUNNABLE`. It has been used a lot in `Schedule`.

8) JoinThread(): It makes the calling method wait until the related Thread finishes its job. This wait is done with a while loop. There are ways to do it outside of Busy Waiting, but it is necessary to access the physical address of the program counter.

Scheduling Algorithm

I used the "Round Robin Schedule" algorithm as the schedule. In this algorithm, each thread in a linear data structure runs in turn. This linear data structure is combined from the tail and head and behaves like a circular array.

Threads continue to run until each interrupt hits the hardware. If the hit from the hardware is not a hardware-related exception, there is no exception and the scheduled method is executed.

```
if(interrupt == hardwareInterruptOffset) {  
    esp = (uint32_t)taskManager->Schedule((CPUState*)esp, false);  
}
```

For each interrupt, the current thread is backed up and the next thread is loaded into memory. (Registers, stack, program counters etc.) And so, our threads will run sequentially.

If I had to explain what I did in the Schedule method:

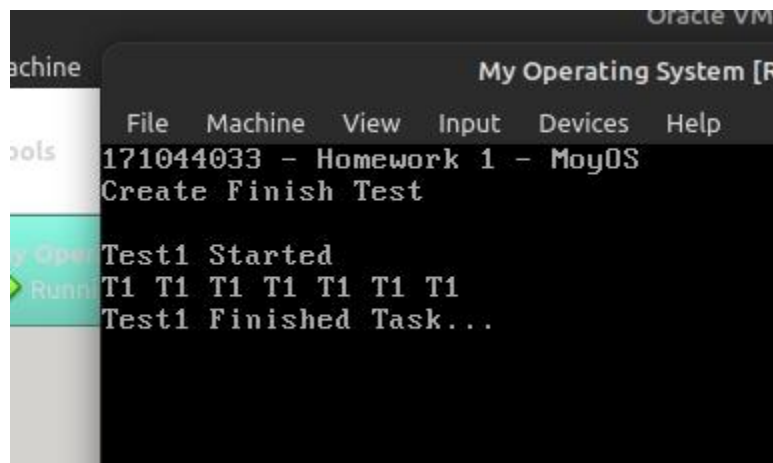
First of all, if the kernel is up for the first time, the available free CPU is backed up (I mentioned the reason in threads). Then, if the thread size is less than 0, this free CPU data is loaded to the CPU (when the threads are finished, you will be 0). If there are still threads in our linear data structure, the CPU's registers, program counter etc. It is backed up for the relevant

thread and the `yield()` method of the thread is called. This method will take the thread from `RUNNING` to `RUNNABLE`. Afterwards, all threads with `DEAD` state in the data structure are removed (for efficiency, if a thread terminates, it is more efficient not to delete the data structure every time, but to delete all of them completely in each scheduling by not running the `DEAD` ones anyway). If Task Manager does not have any threads as a result of these operations, then all of them are in `DEAD` mode and an empty CPU is returned. However, if there are still living threads, the relevant thread that needs to run is loaded into the CPU and its state becomes `RUNNING`.

Test Scenarios and Outputs

1) Create Thread and Finish It Work

In this part, a test thread starts, does a few prints and ends.

A screenshot of a terminal window titled "My Operating System [R" with a menu bar containing "File", "Machine", "View", "Input", "Devices", and "Help". The terminal shows the following output:

```
171044033 - Homework 1 - MoyOS
Create Finish Test
Test1 Started
T1 T1 T1 T1 T1 T1 T1
Test1 Finished Task...
```

2) Create Threads, Terminate and Join These Threads

In this part, producer and consumer threads wait for a while in an endless loop and print the initials of their names. Meanwhile, the `Terminate()` method is called by another thread and the `Join()` is called.

4) Protect Critical Section With Peterson's Solution

Let's imagine that in this field, the producer returns 9999999 times 5 times and creates a product. Since this will be a long time, it is inevitable that there will be an interruption. But the producer will protect the critical area (creating the product) with Peterson's solution. In this test, the consumer shows that if the product has been produced, it has received the product, and if it has not been produced, it is in the critical area. Producer increases an int value while creating the product, and the consumer understands the message to be printed here.

```
My Operating System [Running] - C
File Machine View Input Devices Help
171044033 - Homework 1 - MoyOS
With Peterson Test

Producer Start To Produce Item
Assume that 5 P are one item
P
P
P
P
P
Producer Produced Item
Consumer Start To Consume Item
Consumer Consumed Item
```

5) Showing Race Condition Without Peterson's Solution

```
File Machine View Input Devices Help
171044033 - Homework 1 - MoyOS
Without Peterson Test

Producer Start To Produce Item
Assume that 5 P are one item
P
P
Consumer Start To Consume Item
Consumer Can not Consumed Item (Critical Section)
P
P
P
Producer Produced Item
```