Homework 1

Student Name	Muhammed
Student Surname	Oğuz
Student Number	1801042634

Content

Understanding The Basics

Since the main goal is implementing a multi threading functionality to an OS. It should be learned first to how to execute and run the given OS.

Compiling and Running

Since I use WSL (Windows Subsystem for Linux) as my main development enviroment, I had to figure out how to do this with it.

After some research, with downloading those given apps, Enviroment was ready to go.

```
sudo apt-get install grub-common
sudo apt-get install libisoburn-dev
sudo apt-get install xorriso
sudo apt-get install grub-pc-bin
```

But unfortunately, using `make run` command to execute and running the OS is not possible with this setup.

For avoiding this, I changed my makefile a little bit to work on.

I have to compile and convert to ISO file and manually select this file from VM.

Modified makefile part

```
muo: mykernel.iso
  mv mykernel.iso /mnt/c/Users/Muhammed/Desktop/testIso
```

This command, first creates ISO file and moves to my desktop to select easily from my VM.

I commented this part when delivered my homework.

What does this OS do?

I watched most of the videos of given YouTube series.

This OS mainly have a very basic driver, GUI and very basic multitasking functionality.

Also has some keyboard and mouse interrupts functionality,.

Those things could be helpful while implementing this multi threading functionality.

How Multithread - Multitasking works?

Without multithread, program counter and OS assign desired program to initial step. After that, stack counter increments itself to handle.

With multithread, program counter and OS loads the program to different place of RAM. After that, stack pointer executes both program simultanesly with changing

state all the time.

In this homework, It is expected that only two thread communicate each other.

Implementing The Homework

There are 6 main parts of this homework.

Creating thread, removing thread, yield, joining thread, Petarson algorithm and showing race conditions.

General Implementation

WYOOS (Write your own operating system) lecturer gives the fundamental code to start to the homework.

`multitasking.h` file

It contains necessary classes to work with.

Task: This class initializes the tasks with their corresponding functions. In another word programs.

TaskManager: This class keeps task array and uses it to execute the tasks. This class should has all expected functions for working with threads.

CpuState: This struct type holds different CPU states to handle accordingly.

`multitasking.cpp` file

This file contains implementation of the header with expected implementations.

`kernel.cpp` file

This file contains main function when OS initialized. Instructor creates task functions and their task things in this file.

Schedule function

This functions works as is name. It changes active tasks acordingly to execute simultanesly.

I changed it a bit. It uses modulation to get current task to different state. But I know it is expensive to calculate modulation for CPU. So I change it to this

```
if (++currentTask >= numTasks)
    currentTask = 0; // old -> currentTask %= numTasks;
```

Converting to Thread Library

After understanding the parts of the multitasking, It seems, working way is the same as Threads. So, I create a `multithread.h` and `multithread.cpp` files to achieve the same goal.

I reimplement and change names according to thread.

Since we are allowed to use same functions if we need in this homework, I copied necessary functions from multitasking part.

Creating Thread

Creating thread is actually adding new task with to our TaskManager. Since there is a addTask function implemented by instructor, I only change it name to `createThread`.

It works as expected. When new task is ready to add to tasks, It add corresponding task to task manager task array.

If there is no task, It will not add anything and immediately return the sended cpuState

Yield Thread

When user calls thread manager and sends an thread id to yield, desired thread will wait for some seconds (easily chaneable) and reruns again.

Does a busy waiting while waiting other thread.

This one is pretty hard for me. I almost tried everything to work with. I change shceduler, some other functions, added tons of variables and after I had to change my tasks defination. Afther changing those, It is solved.

A tasks waits in a while loop if correspendin thread is yielded. After unyield, it releases the while loop.

Join Thread

My join strategy was founded when I was trying to implement Yield.

When user calls Join function from ThreadManager with sending thread id, Correspending thread will be wait till other threads complete their works.

In this homework, Since I use my task with a while loop, Other thread will wait in a infinite loop.

Terminating Thread

Terminate thread was very easy when compering others. It is just like works regular array remove function. Remove corresponding thread with comparing send id. And remove it from thread array.

Producer Consumer Part

For producing and consuming, as mentioned in homework pdf, I used peterson algorithm.

Why Using Peterson Algorithm?

Without peterson algorithm, there is a situation that callled race condition. For avoiding to happen this. We use some technics and one of them is peterson algorithm.

This algorithm basically does the following

- Determine which task is should be performed.
- Wait other task in busy waiting till other task is ready to use main tasks result

Other Possible Race Condition Solve Paths

Since, two thread or process uses same data, there will be race condition. Peterson algorithm was one of the solutions.

Other solution might be, when sensitive data is in the process, other thread can perform another job and it will be more performance instead of busy waiting.

Another solution might be implementing process handler to determine which data used from which thread. It is very same solution to above.

My Consumer Producer Problem

I use a variable called fish. 🐠

Producer produces 1000000 fish, and after reaching this count. Consumer consumes till fish count is 0.

Thanks to the peterson algorithm, Fish will be never be big than 1000000 and never will be negative.

Problems That I Encountered

Implementing yield was very hard. I tried almost everything to handle this.

Also, understanding the main code is a bit tough. Since, there is a very much abstraction while you start to homework.