FATİH DOĞAÇ
1901042654

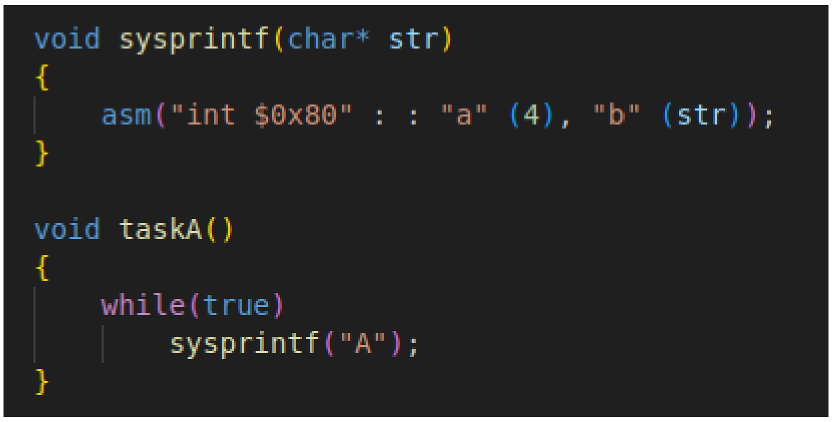
CSE312 - OS
HOMEWORK #1 REPORT

1. Introduction

In this homework, I implemented 4 system calls, round robin and printing collatz number all the numbers less than 100.

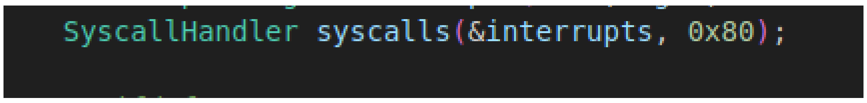
2. System Calls

In the original code, Mr. Engelmann did something like that to create a system call about printf:



```
void sysprintf(char* str)
{
    asm("int $0x80" : : "a" (4), "b" (str));
}

void taskA()
{
    while(true)
        sysprintf("A");
}
```

in the asm() function, “int $0x80” is where we put our data. It is defined as System Call interrupter.



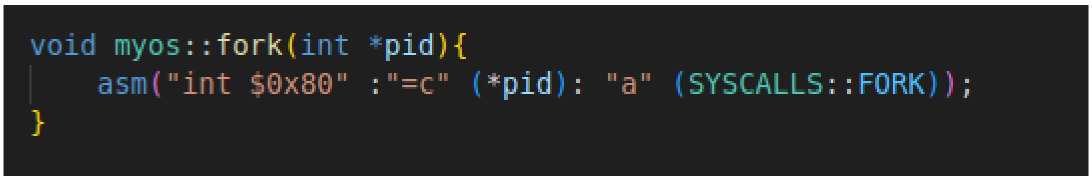
```
    SyscallHandler syscalls(&interrupts, 0x80);
```

So the OS checks 0x80 to handle system calls.

And after the first “:” , this place works as a return register. If we put a variable in there, the result will be written to that variable.

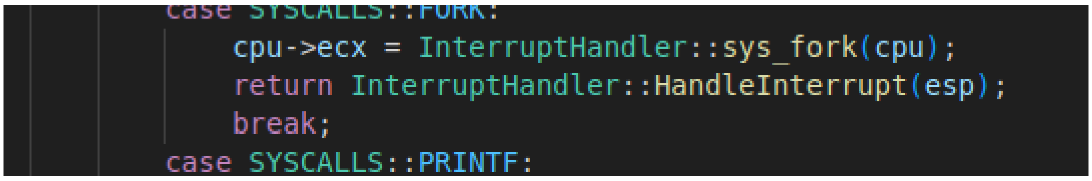
After the second “:” , this place works as parameter registers. “a” goes to cpu→eax, “b” goes to cpu → ebx. And there is also ecx and edx for c and d.

Fork:



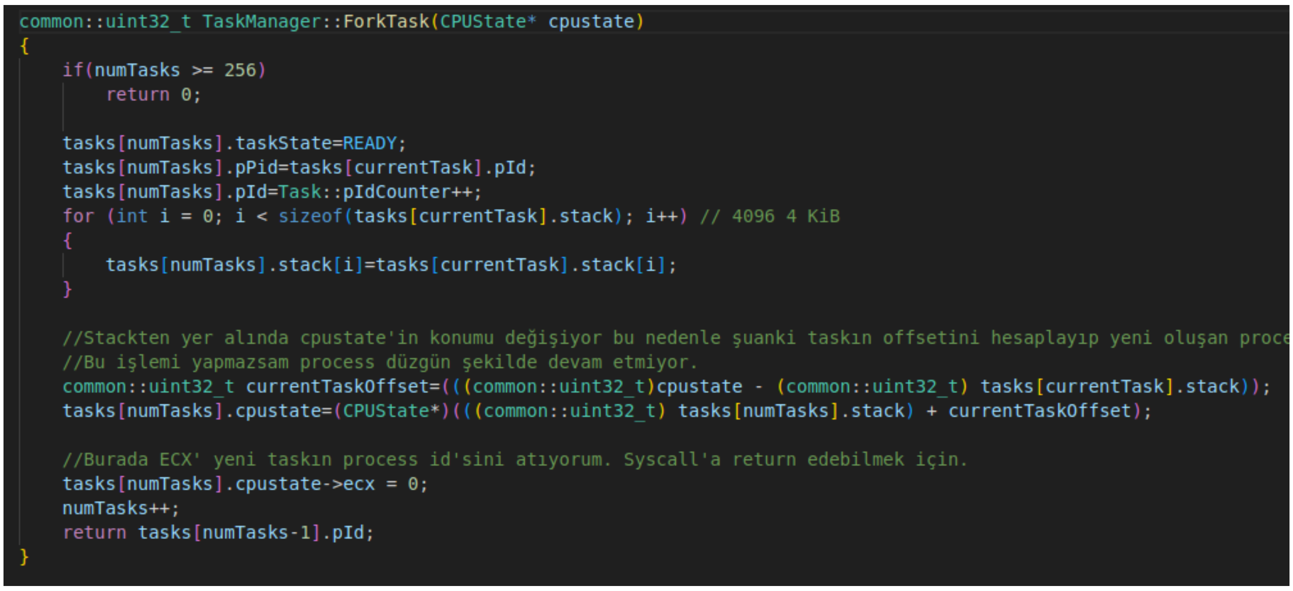
```
void myos::fork(int *pid){
    asm("int $0x80" :"=c" (*pid): "a" (SYSCALLS::FORK));
}
```

It will change the pid by reference with the new child’s pid with c register.



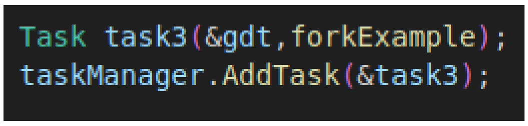
```
        case SYSCALLS::FORK:
            cpu->ecx = InterruptHandler::sys_fork(cpu);
            return InterruptHandler::HandleInterrupt(esp);
            break;
        case SYSCALLS::PRINTF:
```

Then SyscallHandler::HandleInterrupt will check the system call type which is fork, and will pass it to InterruptHandler. Because we have to go to the taskManager somehow.



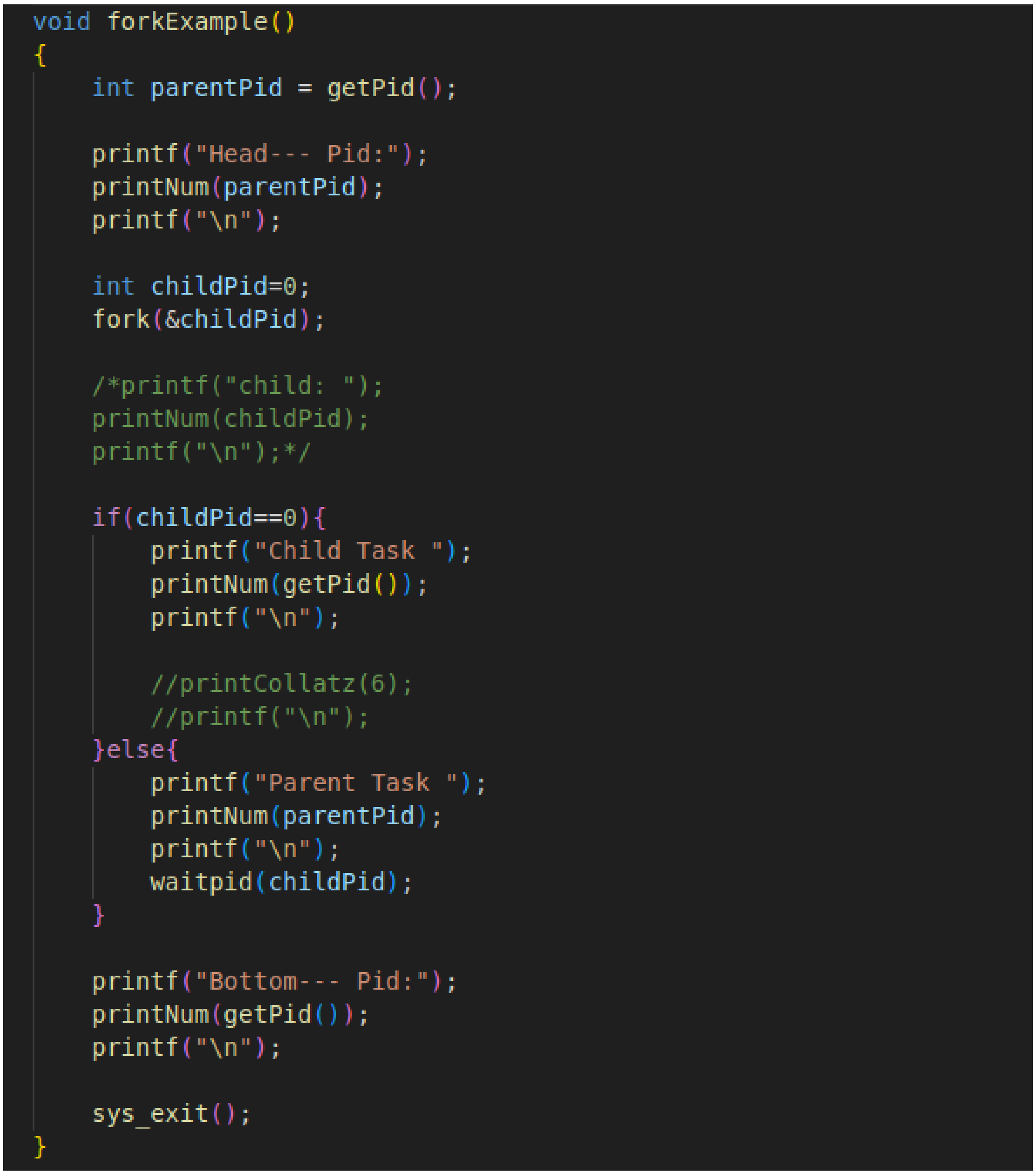
```
common::uint32_t TaskManager::ForkTask(CPUState* cpustate)
{
    if(numTasks >= 256)
        return 0;

    tasks[numTasks].taskState=READY;
    tasks[numTasks].pPid=tasks[currentTask].pId;
    tasks[numTasks].pId=Task::pIdCounter++;
    for (int i = 0; i < sizeof(tasks[currentTask].stack); i++) // 4096 4 KiB
    {
        tasks[numTasks].stack[i]=tasks[currentTask].stack[i];
    }

    //Stackten yer alında cpustate'in konumu değişiyor bu nedenle şuanki taskın offsetini hesaplayıp yeni oluşan proce
    //Bu işlemi yapmazsam process düzgün şekilde devam etmiyor.
    common::uint32_t currentTaskOffset=(((common::uint32_t)cpustate - (common::uint32_t) tasks[currentTask].stack));
    tasks[numTasks].cpustate=(CPUState*)(((common::uint32_t) tasks[numTasks].stack) + currentTaskOffset);

    //Burada ECX' yeni taskın process id'sini atıyorum. Syscall'a return edebilmek için.
    tasks[numTasks].cpustate->ecx = 0;
    numTasks++;
    return tasks[numTasks-1].pId;
}
```

Here we are making the child’s state ready and giving it a copy of the parent’s stack. But stack pointer’s location will be changed so we rewind it. And giving the new child a proper pid and increasing the total task count.



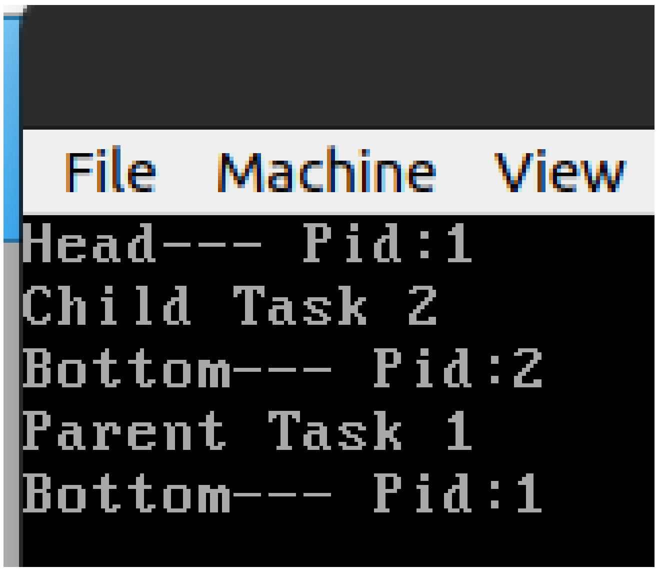
```
Task task3(&gdt,forkExample);
taskManager.AddTask(&task3);
```

First we call the fork example from kernel.



```
void forkExample()
{
    int parentPid = getPid();

    printf("Head--- Pid:");
    printNum(parentPid);
    printf("\n");

    int childPid=0;
    fork(&childPid);

    /*printf("child: ");
    printNum(childPid);
    printf("\n");*/

    if(childPid==0){
        printf("Child Task ");
        printNum(getPid());
        printf("\n");

        //printCollatz(6);
        //printf("\n");
    }else{
        printf("Parent Task ");
        printNum(parentPid);
        printf("\n");
        waitpid(childPid);
    }

    printf("Bottom--- Pid:");
    printNum(getPid());
    printf("\n");

    sys_exit();
}
```

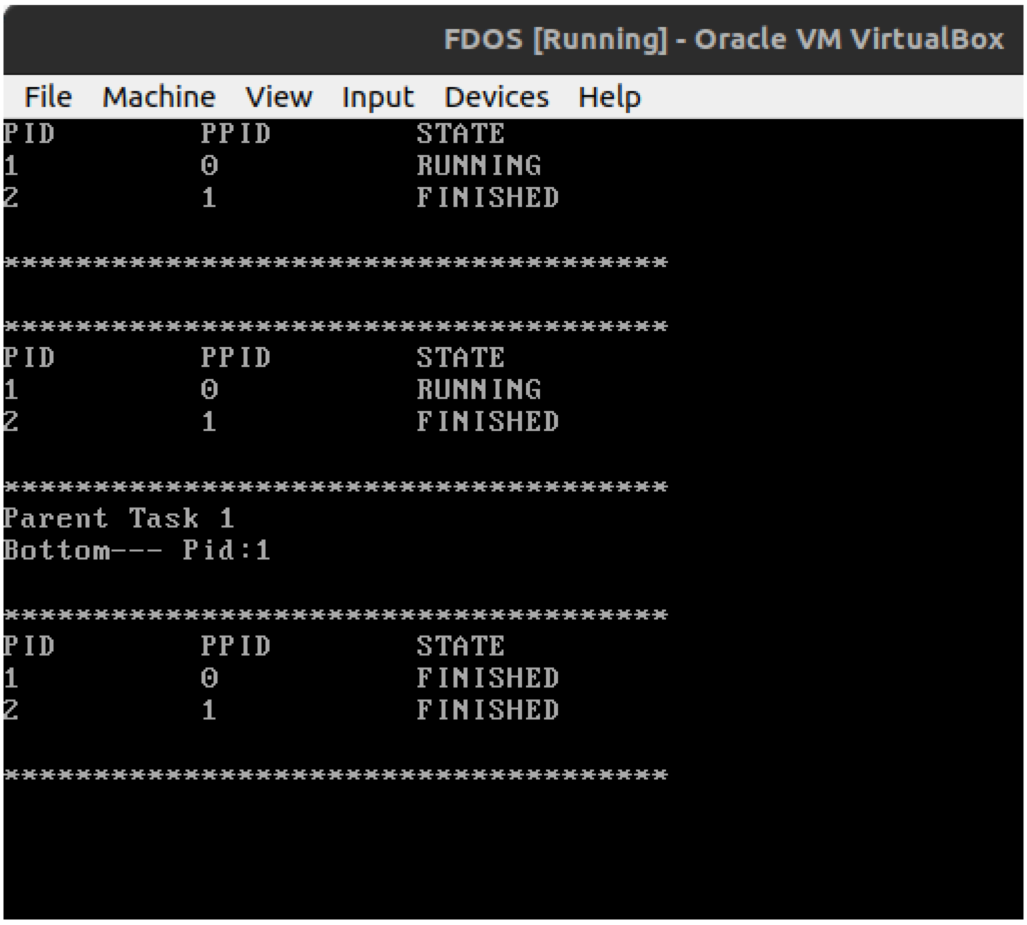
This function prints the parent pid at the head of the function. Then forks a child and make it print its pid. Parent is also printing its own pid. Then at the bottom of the function there is a print too. We expect both child and parent should print that bottom text and to indicate which one is which we append their pids next to it. So the output is this:



```
Head--- Pid:1
Child Task 2
Bottom--- Pid:2
Parent Task 1
Bottom--- Pid:1
```

1 is parent’s pid and 2 is child’s pid. I also have a process table to make it easier to see process states after each tick but it overwrites the screen so I can’t take screen shots of it. I will make you see with a screen recorder in demo.

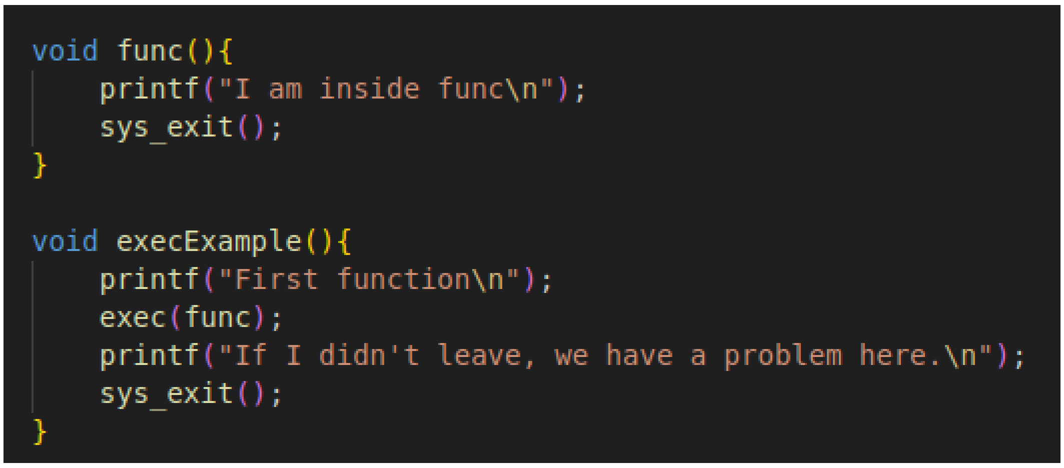
WaitPid:


FDOS [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
PID         PPID        STATE
1           0           RUNNING
2           1           FINISHED

*************************************

*************************************
PID         PPID        STATE
1           0           RUNNING
2           1           FINISHED

*************************************
Parent Task 1
Bottom--- Pid:1

*************************************
PID         PPID        STATE
1           0           FINISHED
2           1           FINISHED

*************************************
```

I couldn’t take a screenshot of the state == WAITING. I will try to make you see in demo with a screen recorder.

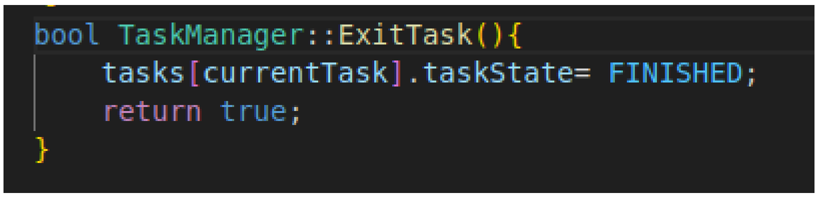
Exec:



```
void func(){
    printf("I am inside func\n");
    sys_exit();
}

void execExample(){
    printf("First function\n");
    exec(func);
    printf("If I didn't leave, we have a problem here.\n");
    sys_exit();
}
```

I expect the program to start in execExample, print “First function” then go the func() and print “I am inside func” and then exit but it prints neither of them and never exits. I couldn’t figure out why in time.

Exit:



```
bool TaskManager::ExitTask(){
    tasks[currentTask].taskState= FINISHED;
    return true;
}
```

Exit makes the currentTask’s state FINISHED. That’s it. It indicates that the task is finished.

3. Round-robin

Actually the Engelmann’s version is already a round-robin implementation.