CSE-312

OPERATING SYSTEMS

SPRING 2024

ONUR ATASEVER

210104004087

Project Design:

First of all, I tried to understand the main outline of the project by watching Engelman. Since it will be a project on system calls, I learned how this will go. After saving the Interrup Handler functions with proper data and introducing them to the CPU, I learned how to create system calls with certain parameters and functions. After the system call occurs, the processor calls the correct interrupt handler according to the incoming interrupt value. If you do not interrupt before doing this, it will push the data of the previous process to the stack. Then the handler function is called. The function here checks whether the incoming interrupt value is in the previously set interrupt handler function. If not, it gives an Unhandled exception error. When the timer interrupt (0X20) comes, it calls the scheduler function. With the result of the algorithm, the scheduler returns the CPUState of the new process to run. The return value from the handler is returned to the CPU by the handler and the program is changed with the new esp value.

Then, since I did not want to use the memory part of the draft I had, I changed the array in which the process pointers were kept to directly hold the process itself. Since I changed this, I changed the addTask function to copy the newly created task instead of throwing it into the task array.

I added two variables to hold the process id of the task and the parent in the task object.

My first goal here was to successfully create tasks and run them with the existing schedule. Fork each Task object creation, I used s static variable to assign process id. Then I increased it. So for each process I have a unique id.

Then I started the fork implementation. I researched how the values in the fork are stored on the stack, in what order the registers are not push and pop, which values will be copied when copying, which values will be recalculated.

In the fork system call, first of all I create an interrupt. I sent 0X80 interrupt number to create an interrupt and I put SYSCALLS::FORK (I used enum for system calls) number to the a register to understand which syscall should I handle.

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

And then with Cpu's help it goes to handler function and according to a register it decides which function call supposed to made.

```
    case SYSCALLS::FORK:
        InterruptHandler::sys_fork(cpu);
        break;
```

In the fork function it copies all the stack from parent to child process. It sets necessary values like parent id, process id. Then the most important thing is cpuState adjustment. According to parents stack positions, child's relative cpuState is calculated. And the other important part is setting ebp (which is base pointer). According to difference between parent's stack base adress and ebp, child's ebp is calculated. Fork supposed to return two value. One of the return value for the parent and other return value is fort he child. To handle this my approach is using ecx registers of the processes. Parent's ecx register has the process id of child and child's ecx register has zero.

```
    tasks[numTasks].cpustate->ecx = 0;
    cpustate->ecx = tasks[numTasks].pId;
```

Then this it returns from cpu to fork function which creates an interrupt and it assigns ecx value to a variable then it returns this value (which is zero for child and child's pid for parent).

After fork system call, I started to implement exit system call. Then I realised that when a process exit it will continue to work because there were not any adjustments for process states. Then I added four states which are ready, running, blocked and finished. And for each Task creation I assigned them READY state. Like fork, I created a function to make an interrupt for exit system call. And I assgined to a register SYSCALLS::EXEC to understand which handler supposed to call. When it exits, it makes it's state finished.

```
void TaskManager::ExitProcess()
{
    if(numTasks <= 0 || currentTask < 0)
        return;
    tasks[currentTask].taskState = FINISHED;
    int parentIndex = getIndex(tasks[currentTask].pPid);
    tasks[parentIndex].waitPid[tasks[currentTask].childIndex]=-1;
    tasks[parentIndex].numChildren--;
}
```

(I implement last three row later. I will explain it).

After exit system call and new process states implementations, I started to implement waitpid. There were to choices to wait children, I could have put a while loop, or I could have change parent's state as blocked. Since using while is busy waiting, I chose second option. But also I had to know which process or processes should parent wait. To do this I created a children array and counter to know how many children should I wait. When fork system call is made, the process id of child is added to children array of the parent. And the index where the child's id is kept in the array is stored in the childIndex data in the child.

```
tasks[currentTask].waitPid[tasks[currentTask].numChildren++]=tasks[numTasks].pId;
tasks[numTasks].childIndex=tasks[currentTask].numChildren-1;
```

So parent knows how many children it has and what are their pid. And children knows the index of it's pid stored. So when the child exits, without linear search by using childIndex variable, it directly access parent's waitPid array and it assigns -1. Then it decreases children counter. So parents know which child is alive and which is passed away.

In the waitpid system call it takes an parameter which is an integer. If it is -1, it means parent waits for all the children. But if the paremeter is more than 0 it waits a spesific child according to pid.

```
void TaskManager::WaitProcess(common::uint32_t pid)
{
    if(numTasks <= 0 || currentTask < 0)
        return;
    tasks[currentTask].taskState = BLOCKED;
    tasks[currentTask].waitPidFor=pid;
}
```

Then I started to implement scheduler which cares processes states. This algorithm increases currentTask value to find next first READY process. To achieve this, I used do while loop. It scans all the array until find a READY state process. The important part is handling BLOCKED processes. In the while loop it searchs READY state but also it checks if the process has BLOCKED state. If the process has BLOCKED state it means process is waiting for a child or children. To understand this, it checks waitPidFor variable. If it is -1 it means it is waiting for all children. Then it checks the counter of children. If it is zero it means, all the children have exited and parent's state has to be READY. If waitPidFor is more than -1 it means the blocked parent is waiting for a spesific child and to check child it needs the state of child. If the state of spesific child FINISHED it means parent's state has to be READY. And then it finds READY process.

And since it is a round robin algorithm it always checks if the value of the current task is more than the value of the numTaks. If it is it takes modula like a circular array.

```
do
{
    if(tasks[currentTask].taskState == RUNNING)
        tasks[currentTask].taskState = READY;
    currentTask++;
    if(currentTask >= numTasks)
        currentTask %= numTasks;
    if(tasks[currentTask].taskState == BLOCKED)
    {
        //WaitpidFor -1 ise processin child'ı yoktur ve processin state'i ready yapılır.
        if(tasks[currentTask].waitPidFor == -1)
        {
            if(tasks[currentTask].numChildren == 0)
                tasks[currentTask].taskState = READY;
        }
        else if(tasks[currentTask].waitPidFor > -1) // WaitPidFor'un -1'den büyük olması durumunda processin
        {
            int childIndex = getIndex(tasks[currentTask].waitPidFor);
            if(tasks[childIndex].taskState == FINISHED)
            {
                tasks[currentTask].taskState = READY;
                tasks[currentTask].waitPidFor = -2;
            }
        }
    }
        // tasks[currentTask].taskState = READY;

} while (tasks[currentTask].taskState != READY);
```

Then I implemented execve system call. To create an interrupt, I used a function again.

```
int myos::execve(void entrypoint())
{
    int result;
    asm("int $0x80" : "=c" (result) : "a" (SYSCALLS::EXEC), "b" ((uint32_t)entrypoint));
    return result;
}
```

It takes as parameter to registers the number for system call and the entrypoint for eip (instruction pointer). It goes handler and it sets all the registers 0 and the most important thing for the execve operation is setting eip. And then it returns to the handler the cpuState of process. This esp value goes to the Cpu and it switch the process and it takes the registers value from the stack back. Then new program starts to be executed.

```
common::uint32_t TaskManager::ExecTask(void entrypoint())
{
    if(numTasks >= 256 || currentTask < 0 || numTasks < 0)
        return 0;

    tasks[numTasks].cpustate = (CPUState*)(tasks[currentTask].stack + 4096 - sizeof(CPUState));

    tasks[numTasks].cpustate -> eax = 0;
    tasks[numTasks].cpustate -> ebx = 0;
    tasks[numTasks].cpustate -> ecx = 0;
    tasks[numTasks].cpustate -> edx = 0;

    tasks[numTasks].cpustate -> esi = 0;
    tasks[numTasks].cpustate -> edi = 0;
    tasks[numTasks].cpustate -> ebp = 0;
    tasks[numTasks].cpustate -> eip = (uint32_t)entrypoint;
    tasks[numTasks].cpustate -> cs = gdt->CodeSegmentSelector();
    tasks[numTasks].cpustate -> eflags = 0x202;
    return (uint32_t)tasks[numTasks].cpustate;
}
```

This is the output of the lifecycle:

It creates 3 process for each task:

```
Task task6(&gdt,TaskForkCollatz);
Task task7(&gdt,TaskForkCollatz);
Task task8(&gdt,TaskForkCollatz);
Task task9(&gdt,TaskForkLongRunningProgram);
Task task10(&gdt,TaskForkLongRunningProgram);
Task task11(&gdt,TaskForkLongRunningProgram);


// taskManager.AddTask(&task1);
// taskManager.AddTask(&task2);
// taskManager.AddTask(&task3);
// taskManager.AddTask(&task4);
// taskManager.AddTask(&task5);
 taskManager.AddTask(&task6);
 taskManager.AddTask(&task7);
 taskManager.AddTask(&task8);
taskManager.AddTask(&task9);
taskManager.AddTask(&task10);
taskManager.AddTask(&task11);
```

For collatz it uses fork functions:

```
void TaskForkCollatz()
{
    int child = -1;
    int i=0;
    child =  myos::fork();
    // printf("Fork value: ");
    // printDigit(child);
    // printf("\n");

    if (child == 0) {
        collatz_sequence(3);
        exit_process();
    } else {
        wait_process(child);
        collatz_sequence(7);
        exit_process();
    }
}
```

Child calls collatz function and parent waits for child and then it calls collatz in parent. So in total it prints 6 times.

```
void TaskForkLongRunningProgram()
{
    int child = -1;
    int i=0;
    child =  myos::fork();
    // printf("Fork value: ");
    // printDigit(child);
    // printf("\n");

    if (child == 0)
    {
        int exec1=execve(long_running_program);
        exit_process();
    }
     else
    {
        wait_process(child);
        exit_process();
    }
}
```

Fork long running program it forks again and in the child it uses execve system call and parents waits child.

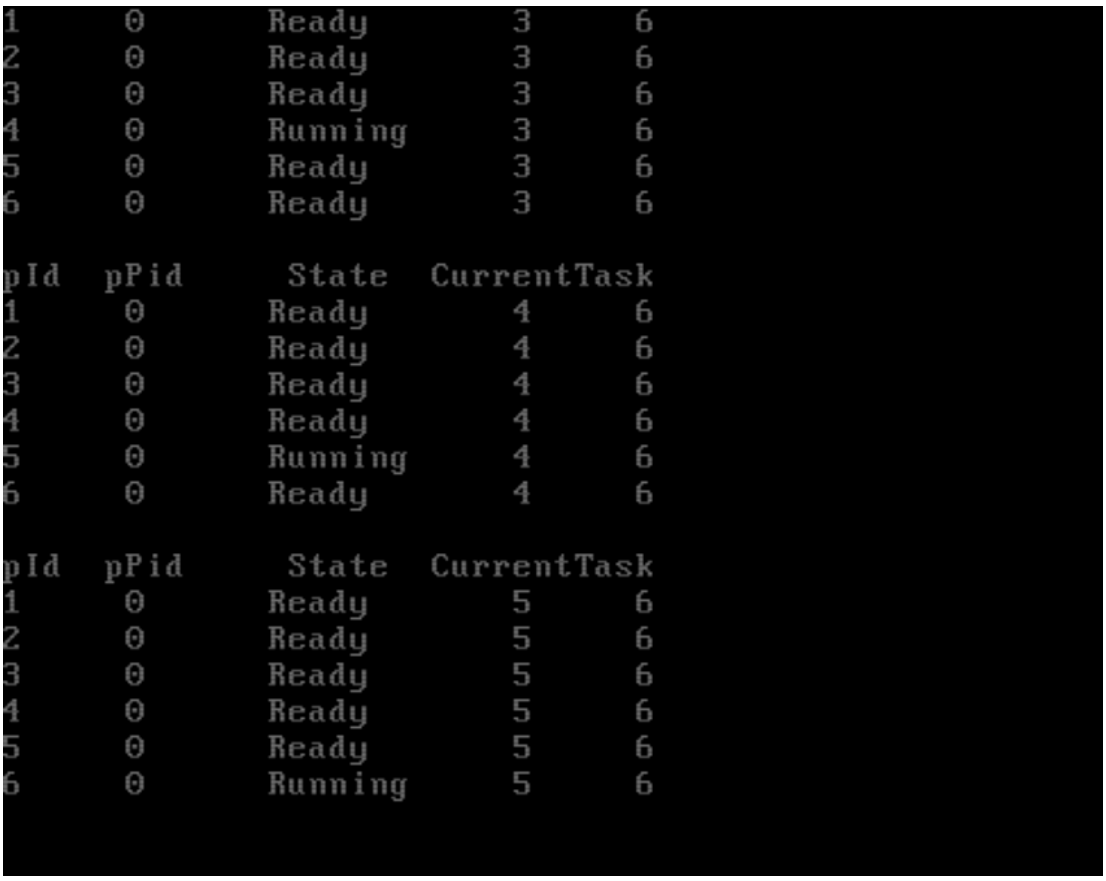
This is the output:

```
Hello World! --- http://www.AlgorithMan.de
3 : 10 5 16 8 4 2 1
3 : 10 5 16 8 4 2 1
3 : 10 5 16 8 4 2 1
Result: 392146832
Result: 392146832
Result: 392146832
7 : 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
7 : 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
7 : 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

This is with process table:

```
pId  pPid     State  CurrentTask
1     0      Finished     2     12
2     0      Finished     2     12
3     0      Running     2     12
4     0      Finished     2     12
5     0      Finished     2     12
6     0      Finished     2     12
7     1      Finished     2     12
8     5      Finished     2     12
9     2      Finished     2     12
10     4      Finished     2     12
11     6      Finished     2     12
12     3      Finished     2     12
7 : 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

This is before fork:

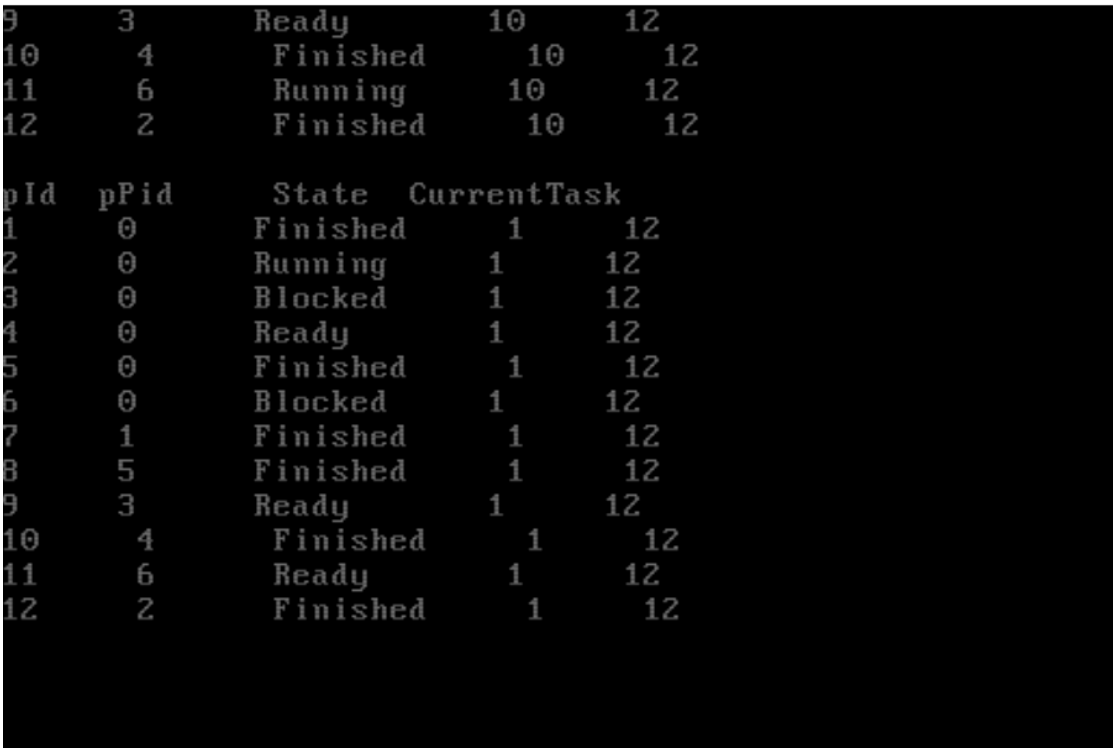


```
1      0       Ready        3      6
2      0       Ready        3      6
3      0       Ready        3      6
4      0       Running      3      6
5      0       Ready        3      6
6      0       Ready        3      6

pId   pPid      State   CurrentTask
1      0       Ready        4      6
2      0       Ready        4      6
3      0       Ready        4      6
4      0       Ready        4      6
5      0       Running      4      6
6      0       Ready        4      6

pId   pPid      State   CurrentTask
1      0       Ready        5      6
2      0       Ready        5      6
3      0       Ready        5      6
4      0       Ready        5      6
5      0       Ready        5      6
6      0       Running      5      6
```

This is process table after fork :



```
9      3       Ready       10      12
10      4       Finished      10      12
11      6       Running      10      12
12      2       Finished      10      12

pId   pPid      State   CurrentTask
1      0       Finished      1      12
2      0       Running      1      12
3      0       Blocked      1      12
4      0       Ready        1      12
5      0       Finished      1      12
6      0       Blocked      1      12
7      1       Finished      1      12
8      5       Finished      1      12
9      3       Ready        1      12
10      4       Finished      1      12
11      6       Ready        1      12
12      2       Finished      1      12
```