# CSE 321 HW1 REPORT

**Abdullah Çelik**

## Table of Contents



## 1. Introduction

Operating System has requirements. I completed all requirements except execve system call.

Requirements:

- Implementing POSIX systemcalls (fork, execve, waitpid, exit)
- Loading multiple program into memory
- Handling multi-programming
- Handling interrupts
- Performing Round Robin scheduling
- Implementing 3 different flavors of MicroKernel

## 2. Implementing POSIX systemcalls

POSIX systemcalls ask the OS to do things for them. For this, proceses write what they want to some special registers and make software interrupts. The software interrupt is handled by the OS and the desired thing is fulfilled. Just like this, I set the registers like eax ebx for systemcalls and made a software interrupt. Then the OS handled the software interrupt (described in detail in Chapter 5).

### a. fork systemcall

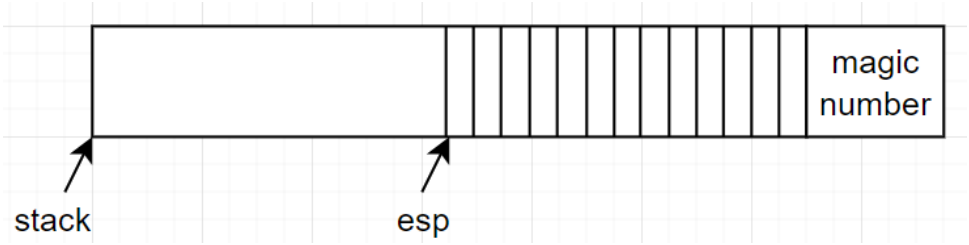
$eax = 2 (interrupt number of fork)
$ebx = entry point (function address)

```
pid_t fork(void (*entry_point)(void))
{
    uint32_t ret = -1;

    __asm__ volatile("movl %0, %%eax; movl %1, %%ebx; int $0x80;" :
                     : "r" (sys_fork), "r" (entry_point) : "eax", "ebx");
    __asm__ volatile("" : "=a"(ret));

    return ret;
}
```

Makes a fork system call. OS creates child process and adds it to the process table with copying all parent registers (makes a copy of parent).

The reason for getting an entrypoint is that the child process cannot fully copy the parent process. For example, the ebp register is a register used to manage the stack frame and creates a new stack frame when the function is called. It puts local variables and temporary variables in this frame. When the function terminates, it will pop and restore these values. I was getting the 0x0D interrupt because I couldn't copy such fields. This interrupt was telling me that I was accessing areas I shouldn't have accessed. In other words, when it couldn't make a full copy, the child process was trying to pop the incorrect addresses. Since I've given the entry point (a new block) it doesn't need to pop up previously used local variables.

Since I do not know how much data is in the stack of a process, I could not fully copy it. I tried different ways for this. One of them was to put a special number on its stack first when I created a process. So the child process could copy this particular number until it sees it. But it doesn’t work.




## b. exit systemcall

$eax = 1 (interrupt number of exit)
$ebx = status of program exit (failure or success)

```
void exit(Exit exit)
{
    __asm__ volatile("movl %0, %%eax; movl %1, %%ebx; int $0x80;" :
                     : "r" (sys_exit), "r" (exit) : "eax", "ebx");

    while(true);
}
```

Makes a exit system call. The status of the process on the CPU is set as terminated. And then this process waits in an infinite loop until the waitpid systemcall is done from parent and the process is cleaned.

### c. waitpid systemcall

$eax = 7 (interrupt number of waitpid)
$ebx = child process id to wait

```
void waitpid(uint32_t pid)
{
    Process::Status ret = Process::Running;

    while(ret != Process::Terminated)
    {
        __asm__ volatile("movl %0, %%eax; movl %1, %%ebx; int $0x80;" :
                         : "r" (sys_waitpid), "r" (pid) : "eax", "ebx");
        __asm__ volatile("" : "=a"(ret));
    }
}
```

Makes a waitpid system call. If child terminates, it is cleared from the table. If not, waitpid returns fail and makes busy waiting here until the child terminates and repeatedly makes waitpid system call. Exits this loop when child process terminates.

## 3. Loading multiple program into memory

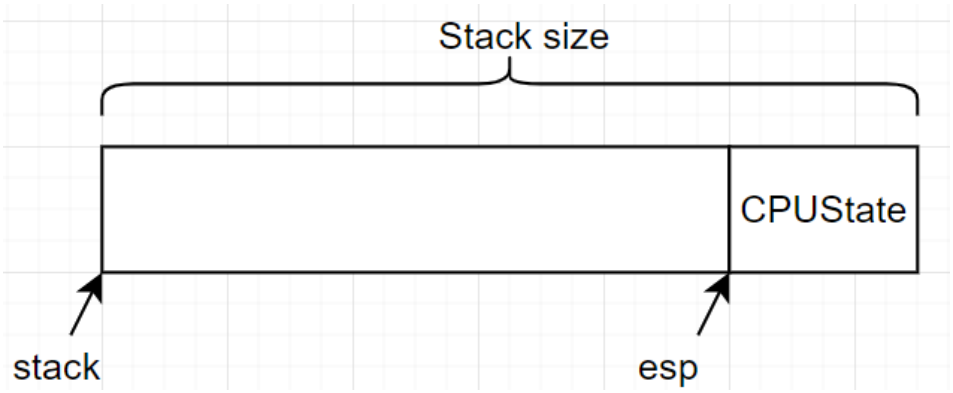
I implemented the fork system call correctly to add multiple programs into memory. Thus, this software interrupt will be handled when fork systemcall is done (chapter 5). Then the child process is created. This child process copies all the values of its parent and is saved to the process table. Thus, another program is taken into memory. This program will run when there is a scheduling.

## 4. Handling multi-programming

I created a process table to do multi-programming. A process is kept in each entry of this table. Each process keeps all the necessary information for itself. It holds its own stack, register values, process id, and state. The interrupt mechanism (chapter 5) takes the next process from the table to the CPU by context switching every time the timer interrupt comes. There is an important detail here. Processes keep their registers values in their own stacks.

```
cpustate = (CPUState*)(stack + STACK_SIZE - sizeof(CPUState));

cpustate->eax = 0;
cpustate->ebx = 0;
cpustate->ecx = 0;
cpustate->edx = 0;

cpustate->esi = 0;
cpustate->edi = 0;
cpustate->ebp = 0;

cpustate->error = 0;

cpustate->esp = (uint32_t)cpustate;
cpustate->cs = cs;

cpustate->eflags = 0x202;
```

It goes forward from the stack address to the stack_size and comes back by the CPUState struct size. This puts the CPUState structure (the structure including the registers) on its stack. After assigning the $esp register, a structure like in the below. Thus, process will be able to both manage his own stack and save the register values here.




While pushing, the stack will grow to the left, and while poping, it will shrink to the right.

## 5. Handling interrupts

Viktor Engelmann builds the structure that communicates with the hardware and handles interrupts (pci, ports, interruptubs). This structure handles hardware interrupts. What about software interrupts (systemcall)? I don't need a very complicated structure for this. Things to do are:

- Define a function to Interrupt Manager for 0x80 (software interrupt number for Intel x86). The name format of this function should be: HandleInterruptRequest0x$$. This is because a macro has been created in interruptubs.s. The macro creates a tag with this name. When the interrupt comes, the handle function will call this assembly label.

```
static void HandleInterruptRequest0x80();
```

- We need to save this software interrupt with the necessary information in the Interrupt descriptor table.

```
SetInterruptDescriptorTableEntry(0x80, CodeSegment,
    &HandleInterruptRequest0x80, 0, IDT_INTERRUPT_GATE);
```

- Creating a label for this interrupt in the Interruptstubs.s file. This label must be in the format specified above

```
HandleInterruptRequest 0x80
```

- All that remains is to override the HandleInterrupt function. For this, I have set up a mechanism here. I created a new class for SystemCalls. This class is inherited from InterruptHandler. SyscallHandler says it will take the interrupt manager as a parameter and handle the software interrupts itself.

```
SyscallHandler::SyscallHandler(InterruptManager* interruptManager, uint8_t InterruptNumber)
    : InterruptHandler(interruptManager, InterruptNumber  + interruptManager->HardwareInterruptOffset()) {}
```

- And then it override the HandleInterrupt function. So when the software interrupt comes, the interrupt manager will handle it. Later, this software will call the handle function for the interrupt and SyscallHandler::HandleInterrupt will run.

## 6. Performing Round Robin Scheduling

When the timer interrupt comes, interruptsubs.s pushes all registers to the stack. Apart from that, it also pushes the interupt number and stack pointer. It then calls the HandleInterrupt function. By looking at the HandleInterrupt interrupt number, it understands that it is a timer interrupt and calls the ProcessTable::Schedule function by giving the current cpu registers. Schedule function does round robin scheduling.

```
CPUState* ProcessTable::Schedule(CPUState* cpustate)
{
    if(size <= 0)
        return cpustate;

    if(current >= 0)
    {
        table[current].cpustate = cpustate;    // Store the old CPU state

        if (table[current].status == Process::Running)
            table[current].status = Process::Ready;
    }

    do
    {
        if (++current >= size)
            current = 0;

    } while (table[current].status == Process::Terminated);

    table[current].status = Process::Running;
    return table[current].cpustate;
}
```

The current index shows which process is currently in the CPU. After saving the CPU using the current index, it moves to the next non-terminated process. It then returns the CPUstate (structure containing all registers) of the new process. Interruptsubs.s then assigns the returned address to the stack pointer and restores all registers. Thus, context switching is done and the new process is taken to the CPU.

## 7. Implementing 3 different flavors of MicroKernel

Three different microkernels implemented. How to run these kernels is specified in readme.txt. These microkernels are not very different from each other. They just load programs into memory differently. You can refer to the homework pdf for this difference.