CSE 321 HW BONUS REPORT

Abdullah Çelik

Table of Contents

1. Introduction

I completed all requirements. Only execve couldn't be implemented because we have no knowledge of file system and our OS doesn't have file system. Instead, another version of execve was implemented. Details are in chapter 2.

Requirements:

- Fork system call (exactly POSIX fork syscall)
- Execve system call (different version of POSIX execve)
- Read system call (exactly POSIX read syscall)
- Handling keyboard interrupt
- Handling mouse interrupt

2. Updates

a. fork systemcall

Fork syscall in HW1 is not exactly like the POSIX system call. Since I couldn't manage the memory, I give an entry point to the child process. This is because when I give a new block to the child process, the child process don't need the parent's stack. In the bonus assignment, the child process copies the parent process's stack.

```c
void binarySearch()
{
    // child continues from here
    exit(exit_success);
}

void init()
{
    pid_t child_pid = fork(binarySearch);

    // parent continues from here

    waitpid(child_pid);
    exit(exit_success);
}
```

(a) HW1 fork sycall

```c
void init()
{
    pid_t child_pid = fork();

    if (child_pid == 0)
    {
        // child continues from here
        exit(exit_success);
    }
    else
    {
        // parent continues from here
        waitpid(child_pid);
    }
}
```

(b) HW Bonus fork syscall

b. execve systemcall

Our OS doesn't have a file system. So i imlement different version of execve syscall. It accepts an function pointer an changes eip pointer to new point.

```c
void binarySearch()
{
    // ...
}

void init()
{
    pid_t child_pid = fork();

    if (child_pid == 0) // child continues from here
    {
        execve(binarySearch);
        exit(exit_failure); // execve() returns only on error
    }
    else    // parent continues from here
    {
        waitpid(child_pid);
        exit(exit_success);
    }
}
```

c. random function

rand() function takes system clock and uses it as random seed. Then generates random number accordingly.

```c
uint32_t rand()
{
    uint32_t randomSeed = 0;

    __asm__ volatile("rdtsc" : "=a" (randomSeed));
    randomSeed = randomSeed * 1103515245 + 12345;

    return randomSeed;
}
```

3. Keyboard Interrupt Handler

The keyboard interrupt handler registers the interrupt manager with the keyboard interrupt number (0x21). The interrupt handler catches the keyboard interrupt when the keyborad is pressed. It saves the registers of the process in the CPU and calls the KeyboardDriver's HandleInterrupt function. This function reads the key from the

relevant port and takes action. Keyboard keys handled are keys 0 to 9, enter key and a key (Key a for scheduling. Details are chapter scheduling). Keys other than this are not accepted. The reason for this is that the homework is as simple as desired. If you want to add a new key, all you have to do is add a case to KeyboardDriver::HandleInterrupt.

```
switch(key)
{
    case 0x02:
        interruptManager->tableManager->AddCurrentProcessBuffer('1');
        handler->OnKeyDown('1');
        break;
```

Here is the case created for the 1 key. As here, a case should be created for the desired key number. Then the desired action should be taken.

4. Mouse Interrupt Handler

Mouse interrupt handler registers to interrupt manager with mouse interrupt number (0x2C). The interrupt handler catches the mouse interrupt when mouse movement occurs. It saves the registers of the process in the CPU and calls the MouseDriver's HandleInterrupt function. This function reads mouse action from the relevant port and takes action. Mouse event handled is left click only. Actions other than this are not accepted. The reason for this is that the homework is as simple as desired.

5. Reading from User

I build a structure to read values from the user. First of all, each process has its own buffer. And manipulates their buffer (reading, writing buffer). So processes will be able to read from their buffers. After the keyboard interrupt handler handles the keyboard interrupt, it saves the character they read in the buffer of the process in the CPU.

The process uses read syscall when they want to read. The read syscall makes software interrupt and reads from its own buffer until the desired count bytes.

```
ssize_t read(int32_t fd, void* buf, size_t count)
```

fd = file descriptor (STDIN)
buf = buffer
count = count bytes

Thus, making read syscall, processes read up to the count bytes they want from their buffers.

6. Scheduling (Context Switching)

In HW1, scheduling is done with timer interrupt. In bonus assignment, scheduling with timer interrupt is canceled. Instead, scheduling is done with left click from mouse or 'a' key from the keyboard . Thus, it is much easier to control output.

```
case 0x1E:
    esp = (uint32_t)interruptManager->tableManager->Schedule((CPUState*)esp);
    break;
```

Pressing key a for scheduling

```
if(offset == 0)
{
    for(uint8_t i = 0; i < 3; i++)
        if((buffer[0] & (0x1<<i)) != (buttons & (0x1<<i)))
            if(buttons & (0x1<<i))
                esp = (uint32_t)interruptManager->tableManager->Schedule((CPUState*)esp);

    buttons = buffer[0];
}
```

Mouse left click