Project 1 report

Name: Christian Magnus Engebretsen Heimvik Kingo ID: 2025318314

Behind the scenes: At initialization

At init of the XV6, we can see from main() that tvinit() is run. This initializes the IDT (Interrupt Descriptor Table). The IDT is x86's implementation of the interrupt vector table, and has 256 entries it maps to different ISRs. In tvinit() in trap.c the elements are filled with its corresponding addresses to ISRs, which is the vectors.S instructions generated by perl script vectors.pl. This is done with the macro SETGATE() in mmu.h. When having set all entries in the IDT to the address of the code spesified in vectors.S, a SW interrupt occurs (with int xxx) or a HW interrupt will case an interrupt and the corresponding vectors.S code will be executed. The important contents of tvinit() is shown below.

```
for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
```

The entry we care about in the IDT is number T_SYSCALL (0x40). This entry is overwritten with a nonzero DPL (Descriptor Priviledge Level) after setting all entries. This is becasue a user must be able to invoke a system call, which we will see later is invoked by int T_SYSCALL. This instruction performs the SW interrupt and jumps to the entry T_SYSCALL in the IDT and start executing at the spesified address.

Behind the scenes: At systemcall

Lets go to usys.S. Here we have the very first thing that happens whenever a user code system call happens. The macro SYSCALL() shown below

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

makes assembly instructions with label name, and uses the name with the SYS_ prefix put the value of the corresponding systemcall number in the eax register as an argument for the interrupt called with the int instruction as previously mentioned. This is to ensure the execution downstreams can know what kind of syscall the user is invoking. The execution will the be interrupted with

```
int T_SYSCALL
```

which will cause an interrupt in the CPU, making the execution to follow the path outlined below.

1. Execution jumps to IDT entry given by T_SYSCALL (entry 0x40).
2. Here lies the address to the lies the code set up by vectors.S, which is executed. The code in vectors.S whos adress was loaded into the IDT duing the init, will in all cases do

```
jmp .alltraps
```

3. This instruction moves the execution to the alltraps label in trapasm.S. Here the obective is to build a trapframe. This is a structure pushed to the current process' kernel stack, which is needed in order to restore the context of the user process invoking the system call. The trapframe also contains information about the interrupt and trap itself. After having pushed the trapframe, trap() is trap.c is then called.
4. In the file trap.c, we find the function traps(struct trapframe* tf) which has the purpose of determining what to do with the current interrupt. Note that it's capable of doing so only because we built up the trapframe earlier, which contains the entry number of the current entry in the IDT (trapno).
5. At this point, the mechanism utilize that we saved the system call number to the eax register before doing int T_SYSCALL back in usys.S. The function syscall() uses this system call number, now in the built up trapframe to find the respective entry in the system call table. The argument to trap() is set up by

```
push %esp
```

in trapasm.S, which is possible as the current trapframe lies on top of the kernel stack. The syscall table is a table mapping the system call numbers (as macros) as indecies, and wrapper functions for the actual implementations of the syscalls as function pointers. The wrapper functions are implemented to make separation and reusability between the fetching of arguments with helper funcitons argxxx() and the actual functionality of the syscall. The function syscall() in syscall.c executes the correct wrapper function (defined in sysproc.c or sysfile.c) in the system call table to retreive the arguments from the calling process' user stack.

7. In sysproc.c, the wrapper functions will do the error handling and argument retrival, for then to call the actual system call functiuonality with the correct arguments defined in proc.c.
8. ***System call executes***
9. However, the system call still needs to return value returned to the calling user process. This is done by letting the syscall functionality return to syscall wrapper, which again returns to syscall().
10. As the function syscall() in syscall.c calls the wrapper funciton by doing curproc->tf->eax = syscallsnum, will the eax register now hold the return value of the wrapper syscall, and then proceed to trap.c where syscall() was called.
11. In trap() in trap.c returns again unless the process we came from is killed.
12. We return to alltraps(), where the trapframe adress at the kernel stack is deallocated, for the return to continue execution down to the next label, being trapret. Here we pop the trapframe off the stack.

We then perform iret, which is the interrupt return instruction. This restores the context of the user space process, and the return value of the syscall is to be found in register eax.

Implemenation of the 3 new systemcalls

As described above, there are several mechanisms that makes the system call work from user space all the way into kernel space and back into user space. Considering the way of working as described above, there are mainly 6 places where changes were applied in order to get implement the 3 system calls.

1. Add the system call number of the 3 system calls to the system call numbers list in syscall.h. This is done in order to be able to pass the system call number to the eax register and to be picked up by syscall() function in syscall.c to execute the correct system call wrapper function in the system all table.

```
#define SYS_setnice 22
#define SYS_getnice 23
#define SYS_ps      24
```

2. Add the user-end assembly system call label in usys.S. This makes it possible that when a user performs the system call, an interrupt will be generated with the new system call numbers in the eax registers.

```
SYSCALL(setnice)
SYSCALL(getnice)
SYSCALL(ps)
```

The same macro used for the other ones are used for these 3 as well. 2. Make the wrapper funciton. It's to be called from the system call table, based of the system call number. Arguments are fetched from the user stacks accessible trough the helper functions argxxx().

```
int sys_setnice(void){
    int pid, nice;
    if(argint(0,&pid) < 0 || argint(1,&nice)<0){
        return -1;
    }
    return setnice(pid,nice);
}
int sys_getnice(void){
    int pid;
    if(argint(0,&pid) < 0){
        return -1;
    }
    return getnice(pid);
}
void sys_ps(void){
    int pid;
    if(argint(0,&pid) < 0){
```

```
        return;
    }
    return ps(pid);
}
```

3. Add the system call wrapper fucntions to the system call table. To enable access to the wrappers via their respective system call number in syscall.c.

```
[SYS_setnice] sys_setnice,
[SYS_getnice] sys_getnice,
[SYS_ps]      sys_ps,
```

4. Declare the wrapper functions as extern in syscall.c (to tell the linker that their definition is in another file, in this case sysproc.c or sysfile.c)
5. Make the actual implementation of the system calls. Due to space constraints, their full implementation will not be listed here. Common for all system call implementations is that they operate on the process table which has a mutex. This is to ensure that the current process (now in the kernel) gets exclusive access to this resource. Also it ensures no other process will access the structure in case of another interrupt.

- setnice() system call is implemented by locking the system call table, for then to iterate through it, checking the PID against the argument. A match releases the mutex and reutns 0, while no matches releases the mutex and returns -1.

```
int setnice(int pid, int nice){
    if(nice<0 || nice > 39){
        return -1;
    }
    acquire(&ptable.lock);
    for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(pid == p->pid && pid != 0){
            p->nice = nice;
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

- getnice() system call is implemented through the same approach as described above. One thing to note here is the copy of the nice value, as no deferal of the mutex release is possible after return.

```
int getnice(int pid){
    acquire(&ptable.lock);
    for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++){
```

```
        if(pid == p->pid && pid != 0){
            int nice = p->nice;
            release(&ptable.lock);
            return nice;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

- ps() system call utilizes 4 helper functions for formatting the output: one for formatting the state of the processes (strprocstate()), and one for formatting required spaces to present the data in a neat table format (cprintpad()), one that prints the header (printheader()) and lastly one that prints the content of a process struct(printcontent()). In the system call ps itself, only printheader() and printcontent() is used to print the different processes' properties based on the pid provided in the argument for better code quality.

```
void ps(int pid){
    acquire(&ptable.lock);
    if(pid==0){
        int hasHeader = 0;
        for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(!hasHeader){
                printheader();
                hasHeader = 1;
            }
            printcontent(p);
        }
    } else if(pid>0){
        for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(pid == p->pid){
                printheader();
                printcontent(p);
                break;
            }
        }
    }
    release(&ptable.lock);
}
```

6. Add the declarations of the new system calls to user.h adn defs.h. This is to make the calls avalibale for any user.