CS202 Lab1 Report

Md Abdullah Al Mamun *mmamu003@ucr.edu*
Yicheng Zhang *yzhan846@ucr.edu*

1 THE LIST OF ALL FILES MODIFIED

In this project, the files that we modified are listed below.

- kernel/syscall.h: Define new system call number.
- kernel/syscall.c: Update system call table.
- kernel/sysproc.c: Define system call function.
- kernel/proc.c: Create a new kernel function for first lab.
- kernel/defs.h: Create a new system call.
- user/usys.pl and user/user.h: Update user-space syscall interface.
- user/test.c: Create " test.c" file in the user directory to test functionality.
- Makefile: Append UPROGS in Makefile.

2 EXPLANATION ON WHAT CHANGES WE HAVE MADE IN FILES AND THE SCREENSHOTS

In the first step, we define define new system call number 23 to our system call — SYS_lab1 in "`kernel/syscall.h`" and update system call table in "`kernel/syscall.c`", which is shown in figure 1 and 2.

```
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_hello  22 // hello
#define SYS_lab1   23 // First lab
```

Figure 1: Syscall.h

```
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_hello]   sys_hello,   // hello: system call entry
[SYS_lab1]    sys_lab1, // lab1: system call entry
```

Figure 2: Syscall.c

In the next step, we firstly declare the lab 1 system call function — int info(int param) in the "`kernel/sysproc.c`", which is shown in figure 3. In "`kernel/proc.c`", we program the int info(int param), shown in figure 4., which takes as input one integer parameter of value 1, 2 or 3. Depending on the input value, it returns: (1) A count of the processes in the system; (2) A count of the total number of system calls that the current process has made so far; (3) The number of memory pages the current process is using. Also, we update "`kernel/defs.h`" with int info(int param), which take integer as input, which is shown in figure 5.

```
uint64 sys_lab1(void) // lab 1 syscall definition
{
  int n;
  argint(0, &n);
  info(n);
  return 0;
}
```

Figure 3: sysproc.c

```
//lab1_function:
//if param == 1, return # of process in the system;
//if param == 2, return # of syscalls that current process has made;
//if param == 3, return # of mem pages of current process.
int info(int param)
{
  struct proc *p;
  int count = 0;
  if(param == 1)
  {
    for(p = proc; p < &proc[NPROC]; p++)
    {
      if(p->state != UNUSED)
          count++;
    }
    printf("The number of  process in the system is: %d\n", count);
  }

  else if (param == 2)
  {
    struct proc *p;
    p = myproc();
    printf("Total %d System Calls that current process (Process ID %d) has made so far\n",systemCallCount[p->pid],p->pid);
  }

  else if (param == 3)
  {
      struct proc *p;
      p = myproc();
      int numPages;
      if(p->sz%PGSIZE == 0) numPages =  p->sz/PGSIZE;
      else
          numPages = 1+(p->sz/PGSIZE);
      printf("Current Process (Process ID %d) is using %d memory pages\n",p->pid,numPages);
  }
  return 1;

}
```

Figure 4: proc.c

```
int             either_copyin(void *dst, int us
void            procdump(void);
void            print_hello(int); //hello
int             info(int); // lab1_function
```

Figure 5: defs.h

In the third step, we update user-space syscall interface with new lab1 system call, which is shown in figure 6 and 7.

```
entry("sleep");
entry("uptime");
entry("hello"); # hello syscall for user
entry("lab1"); # lab 1 syscall for user
```

Figure 6: usys.pl

The last step is to create a "`user/test.c`" file in the user directory to show functionality of lab1 system call. Also, Makefile is needed to be modified to gurantee it can be compiled. The modification is shown in figure 8 and 9.

3 EXPLANATION ON WHAT CHANGES WE HAVE MADE IN CODE IMPLEMENTATION

We had to define a new system call function named **sys_lab1 in Kernel/sysproc.c** which calls the function named **info in Kernel/proc.c** and passes the parameter collected from the user (i.e.,

```
24    int sleep(int);
25    int uptime(void);
26    int hello(int); // hello
27    int lab1(int); // lab1
```

Figure 7: user.h

```
1   #include "kernel/types.h"
2   #include "kernel/stat.h"
3   #include "user/user.h"
4
5   int main(int argc,  char *argv[])
6   {
7       int n = 0;
8       if (argc >= 2) n = atoi(argv[1]);
9
10      printf("Tell me your input for lab1: %d\n", n);
11      lab1(n);
12      exit(0);
13  }
14
```

Figure 8: test.c

```
118   UPROGS=\
119       $U/_cat\
120       $U/_echo\
121       $U/_forktest\
122       $U/_grep\
123       $U/_init\
124       $U/_kill\
125       $U/_ln\
126       $U/_ls\
127       $U/_mkdir\
128       $U/_rm\
129       $U/_sh\
130       $U/_stressfs\
131       $U/_usertests\
132       $U/_grind\
133       $U/_wc\
134       $U/_zombie\
135       $U/_test\
```

Figure 9: Makefile

1, 2 or 3) to perform the responsibilities of the system call that takes as input one integer parameter of value 1, 2 or 3. Depending on the input value, it returns: (a) A count of the processes in the system;(b) A count of the total number of system calls that the current process has made so far; (c) The number of memory pages the current process is using.

3.1 Process of computing the count of the processes in the system

To implement this functionalities, we have used the function named " `int info(int param)`" in " `Kernel/proc.c`". If the function receives 1 as its parameter then it computes the count of the process in the system, which is shown in figure 10. Initially we have declared a structure of process and set the total process Count to zero. Xv6 [1] uses the following names for process states: UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE. Then we have run a loop upto NPROC (maximum number of processes in the system) times to check how many processes are holding any other state except UNUSED in the system. We have increased the value of count by 1 for each process having the state except UNUSED in the system. Finally, It prints the value of count which represents the count of the processes in the system.

```
struct proc *p;
int count = 0;
if(n == 1)
{
  for(p = proc; p < &proc[NPROC]; p++)
  {
    if(p->state != UNUSED)
        count++;
  }
  printf("The number of  process in the system is: %d\n", count);
}
```

Figure 10: implementation of first task

3.2 Process of counting the total number of system calls that the current process has made so far

To implement this functionalities, we have declared an array named " `systemCallCount`" of size NPROC (maximum number of processes in the system) in " `Kernel/proc.h`". Then we use the reference of the array in " `Kernel/proc.c`" again. When a process is initialized within allocproc function in " `Kernel/proc.h`" then systemCallCount of that process is set to zero. Again, When a process is destroyed within freeproc function in " `Kernel/proc.h`" then systemCallCount of that process is also set to zero cause later a new process may be initialized in that place, and for that new process, we have to start counting from zero. Finally, inside " `Kernel/syscall.c`", we increased the value of count by 1 for each corresponding process ID so that later on we can get the total number of system call initiated by a process just giving its process ID. Finally, we used the function named " `int info(int param)`" in " `Kernel/proc.c`". If the function receives 2 as its parameter then it prints the total number of system calls that the current process has made so far just using the " `systemCallCount`" array directly, which is shown in figure 11. we have declared a structure of process, and then it holds the reference of current process. From that process, we fetch the process ID. Using the ID, we get the total number of system calls that the current process has made so far which is stored in the array named " `systemCallCount`".

```
else if (n == 2)
{
    struct proc *p;
    p = myproc();
    printf("Total %d System Calls that current process (Process ID %d)
            has made so far\n",systemCallCount[p->pid],p->pid);
}
```

Figure 11: implementation of second task

3.3 Process of computing the number of memory pages the current process is using.

To implement this functionalities, we have declared a structure of process, and then it holds the reference of current process. From that process, we fetch the the Size of process memory (bytes). There is also a built in variable named "PGSIZE" in "kernel/riscv.h" of the Xv6 OS which holds the sie (bytes) per page. We check if the process size is divisible by "PGSIZE" then we print the integer division value, otherwise print the rounding up value. As the process may not use always a full size Page. The code implementaion is is shown in figure 12.

```
else if (n == 3)
{
    struct proc *p;
    p = myproc();
    int numPages;
    if(p->sz%PGSIZE == 0) numPages =  p->sz/PGSIZE;
    else
        numPages = 1+(p->sz/PGSIZE);
    printf("Current Process (Process ID %d) is using %d memory pages\n",p->pid,numPages);
}
```

Figure 12: implementation of third task

4 RESULTS

After implementing all of the functionalities that we were asked to implemented (task1,2 and 3), we simulate the OS by providing the argument 1, 2 and 3 to the test.c function correspondingly to pass it to the kernel system call from user process. The output is given below in figure 13, figure 14, figure 15 and figure 16.

```
mamun@LAPTOP-9O07R6MT:/mnt/c/Users/Md Abdullah Al Mamun/Desktop/xv6-riscv$ make qemu
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -MD -mcmodel=medany -ffreestanding -fno-common -nos
tdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie   -c -o kernel/proc.o kernel/proc.c
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -MD -mcmodel=medany -ffreestanding -fno-common -nos
tdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie   -c -o kernel/syscall.o kernel/syscall.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/start.o kernel/con
sole.o kernel/printf.o kernel/uart.o kernel/kalloc.o kernel/spinlock.o kernel/string.o kernel/main.o kernel/vm.o kernel/
proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kerne
l/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o k
ernel/virtio_disk.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographic -drive file=fs.img,if=none,
format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting
```

Figure 13: After starting the qemu simulator

```
xv6 kernel is booting

init: starting sh
$ test 1
Tell me your input for lab1: 1
The number of  process in the system is: 3
$
```

Figure 14: Output of task1

```
$ test 2
Tell me your input for lab1: 2
Total 33 System Calls that current process (Process ID 4) has made so far
$
```

Figure 15: Output of task2

```
Tell me your input for lab1: 3
Current Process (Process ID 5) is using 3 memory pages
$
```

Figure 16: Output of task3

5 DESCRIPTION OF XV6 SOURCE CODE (INCLUDING OUR MODIFICATIONS) ABOUT HOW THE INFO SYSTEM CALL IS PROCESSED, FROM USER TO KERNEL TO AGAIN USER PROGRAM

In this section, we discuss how the info system call is processed, from the user-level program into the kernel code, and then back into the user-level program.

Firstly, in user-level program "user/test.c" , we call lab1(n) function in line 11 of figure 6, where n refers to the input to this program. The value of n is taken as an argument while executing the user-level program "user/test.c". Then, lab1(n) will refer to "user/usys.pl" in line 40 of figure 7, which is the user interface provided by a kernel to let user programs see.

With assistance of user interface, we invoke the system call sys_lab1() in "kernel/sysproc.c.c" in line 11, which is shown in figure 3. By invoking such system call, the user program can have access to the kernel and run info() defined in "kernel/proc.c" in line 50 of figure 4.

In last step, the system call will return a desired count as we expected to user-level program. As a result, the specific desired count will be printed out based on the different input value provided while executing the user-level program "user/test.c".

REFERENCES

[1] R. Cox, M. F. Kaashoek, and R. Morris. Xv6, a simple unix-like teaching operating system, 2011.