Operating System CS 202

Lab Assignment Report 1

Madhusudhan Tungamitta(862325125)
Akhilesh Reddy Gali(862258042)

Summary:

1. List Of Files Modified
2. Screenshots of the changes made
3. Output screenshots
4. Working flow
5. Running Commands
6. Demo Video Link

List of Files Modified:

We have created a new file "testinfo.c" in the user folder and below are the files we made changes

Folder Name	**Files**
Kernel	syscall.h
Kernel	syscall.c
Kernel	sysproc.c
Kernel	proc.c
Kernel	proc.h
Kernel	defs.h
user	usys.pl
user	user.h
user	testinfo.c
	MakeFile

Kernel\Syscall.h:

By adding sys_info in line 22 we are reserving a place for our own system call.

```
15  #define SYS_uptime 14
16  #define SYS_open   15
17  #define SYS_write  16
18  #define SYS_mknod  17
19  #define SYS_unlink 18
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  #define SYS_info   22  // Defining system call
```

Kernel\Syscall.c

We create an entry for the system call "SYS_info" and its function "sys_info" in the system call table. Here we are adding a pointer to the system call where this file contains an array of function pointers to the system calls which are defined in different locations, So the sys_info acts as a pointer for our system call

```
extern uint64 sys_write(void);
extern uint64 sys_uptime(void);
extern uint64 sys_info(void); // info is declared here

static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_info]    sys_info, //Info:entry,Added that acts as function pointer to our system call

};
```

Kernel\Sysproc.c

As we created a new function sys_info in the system call table we defined previously in the sysproc.c file, we define the sys_info function. The clear need for writing sysproc is that when the user wants to pass arguments to kernel-level functions from the user level function it is not directly possible in xv6.

So Xv6 developers came up with their inbuilt function "argint"() that helps in passing arguments from user to kernel level. Then that integer is passed into the print_info function using argint().

```
// Process related system call functions are created here and
// when n = 1 then it counts the proccesses in the system and returns the count
//when n=2 it return total no of system calls the current process made
//when n=3 return no of memory pages current process is using

uint64
sys_info(void)
{
  int n;
  argint(0,&n);
  print_info(n);
  return 0;
}
```

Kernel\Proc.c:

In our proc. c file we have added a print_info function if the user enters values other than 1,2,3 it is going to return -1.

If the value passed is 1: It returns the count of the processes in the system.

If the value passed is 2: It returns the Number of system calls the process has made.

If the value passed is 3: It returns the number of memory pages the current system is using.

Code Explanation:

1. The sequence of instructions between the acquire and release is called the critical section, Here we use a lock for protection purposes. P iterates through " " and checks if the process state is used or not, the if condition is used to check if the process is in an unused condition. Else it will increment the counter, we will return a counter that contains the process count.
2. We have already initialized ncount in the process header file and the count is increased it goes through the system call loop.

3. PGsize standard value is 4096 and its used to return the number of memory pages current process is running.

```
void print_info(int n)
{
  struct proc *p;
  if(n==1)
  {
    int pcount =0;
      p = proc;
    while(p<&proc[NPROC])
     {
     acquire(&p->lock);
     if(p->state == UNUSED)
    {}
    else{ pcount++;}
    release(&p->lock);
    p++;
  }
  printf("Total No of process running: %d\n",pcount);
  }
  else if(n==2){
    printf("No of system calls that current process Made: %s %d\n", myproc()->name, myproc()->ncount);
  }
  else if (n==3)
  {
  printf("No of memory pages current process is running: %s %d\n", myproc()->name, ((myproc()->sz)/PGSIZE));
  }
  else
  {
  printf("-1 \n");
  }
}
```

Kernel\proc.h:

This is the header files for the process and here we are declaring the counter variable of type integer which is used to track the number of system calls that a current process has made so far.

```
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int ncount;                  // counts no of system calls that current process has made
};
```

Kernel\defs.h

```
int             either_copyin(void *dst, int user_src, uint64 src, uint64 len);
void            procdump(void);
void            print_info(int); // Forward declaration for our new system call
```

User\user.h:

```
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int info(int);
```

User\usys.pl

This is the entry point for the info

```
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("info"); #info sys call for user
```

Make File:

```
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_testinfo\
```

Testinfo.c:

This is the user program function that is added to test the functionality and it will be used to call the system call, when we run the user program n qemu on the terminal it will display the results

```c

int main(int argc, char *argv[])
{
    int n=0;
    if(argc == 2)
    {
        n = atoi(argv[1]);
        // printf("Hello, Lab 1 is running succesfully %d\n",n);
        info(n);
    }
    exit(0);
}
```

Output

```
 kernel/spinlock.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline
ernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kerne
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 3 -nographic -drive file=fs
bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ testinfo 1
Total No of process running: 3
$ testinfo 2
No of system calls that current process Made: testinfo 5
$ testinfo 3
No of memory pages current process is running : testinfo 3
$ testinfo 4
-1
$
```

Running Commands:

Running on Windows subsystem:

1. Install ubuntu from Microsoft store
2. Run the below commands

$ sudo apt-get update && sudo apt-get upgrade

$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu

3.Copy our project and got to the folder “cd xv6-riscv”

4. Run the command “make qemu”
5. Once it starts running
    a. Command: “testinfo 1” to check the first test case(Count of processes in system)
    b. Command: “testinfo 2” will help you to retrieve the results that show the total number of system calls that the current process has made so far.
    c. Command: “testinfo 3” will return the number of memory pages that the current process is using.

Demo Link:

https://drive.google.com/drive/folders/152AYCubWDxqgqdP0xvgyuK_erinaOlUK?usp=sharing