Operating System Lab Project 2 Report

Reza Abdoli
Mohammad Reza Alavi
Farbod Azimmohseni

November 12, 2023

Questions

1. *ULIB* contains implementation of functions like *printf*, *strcpy*, *strlen*, etc... to make a interface for system calls. Also **usys.S** file in *ULIB* sets the entry point for all system calls of xv6. It first set all neccesary registers and then triggers software interupt of the system call with its specific number. It also contains *usys.S* code which is the entry points for system calls.

2. *O*ne way to access kernel from user mode is **pseudo-file**./proc, */dev*, and */sys* are called pseudo-file systems because they are not backed by disk, but rather export the contents of kernel data structures to an application or administrator as if they were stored in a file. These files give useful information about kernel. for example */dev* contains all devices connected to system as files. Another way user mode interacts with kernel mode is using Exceptions and signals. signal is a message sent by kernel to user and exception happens when user process run into a problem.

3. No. It is not possible. If a user try to activate a trap with kernel privilege kernel will return an error. If it was possible a harmful process would cause damage.

4. Both user and kernel have their own stack. **ss** points to the segment where our stack is and **esp** is the address where the stack starts. When switching from user mode to kernel mode data of user stack data should be saved and these two registers need to be used for kernel stack data and address. Obviously there is no need to do this operation if we do not want to switch.

5. *argint()* calculates address of the n'th system call argument in the stack.Using value of the *esp* register, this function only do a simple calculation.

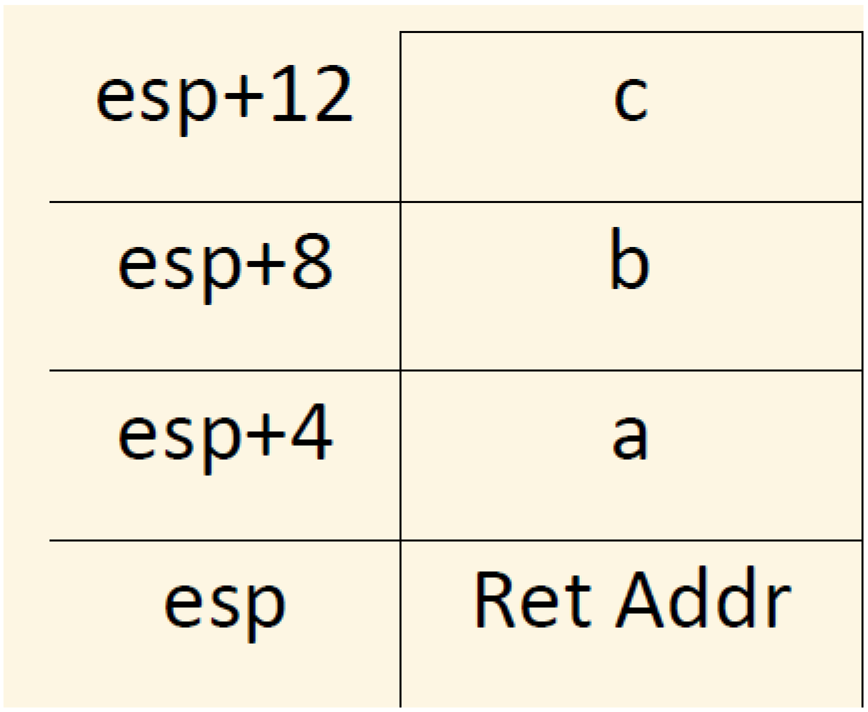


Figure 1: stack structure

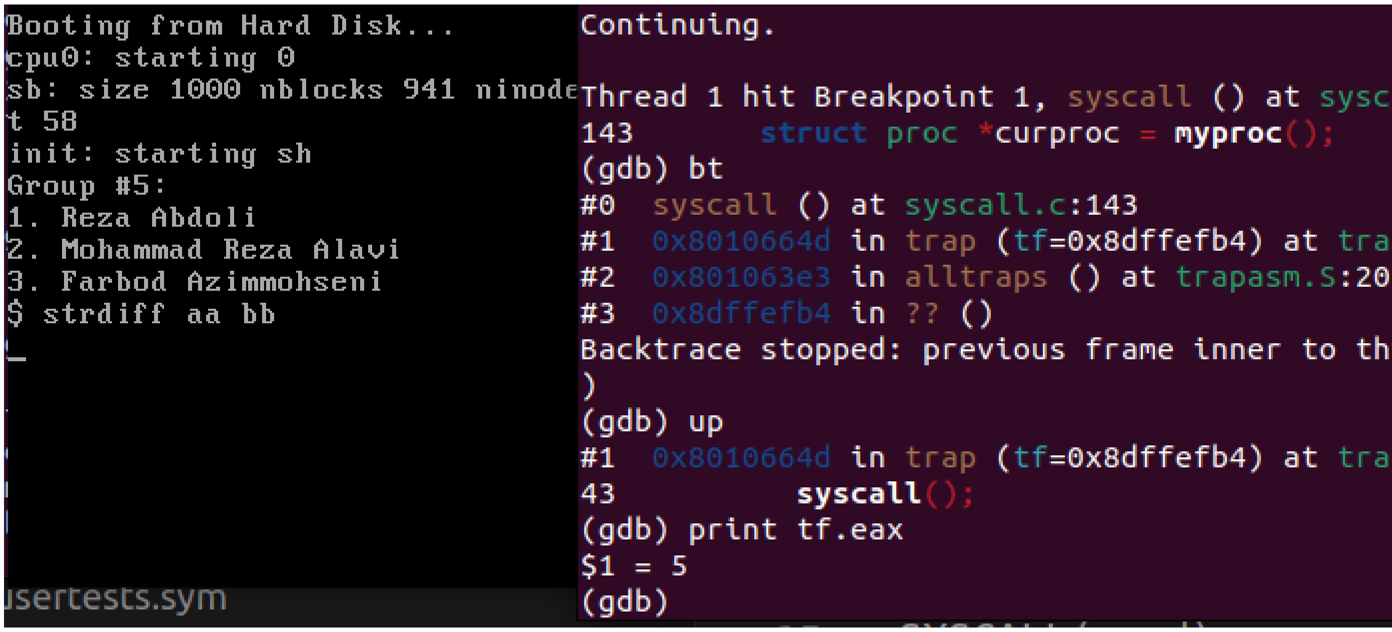
The function *argptr()* is used to validate the memory specified for a system call argument. This function takes three arguments. The first argument, **n**, represents the index of the argument. The second argument, **p**, is a pointer to the user address where the argument resides. The third argument is the address of the argument itself.

The purpose of *argptr()* is to verify that **p** is within the user space. By checking the range of the address, we can ensure that we are not manipulating any memory that we should not have access to. Additionally, the size of the memory is checked to ensure that it has enough space to hold the argument. This helps prevent any potential buffer overflows or memory corruption issues.

Another related function, *argstr()*, is used to validate string arguments. It starts from the string pointer and reads the memory until it encounters a null character, indicating the end of the string. This ensures that the string argument is valid and prevents any unintended memory access or manipulation.

For example, when using the system call sys-read(), which is used for reading data from a file or input stream, it takes a **buffer** argument. By checking that all of the buffer memory is within the process's memory space, we can ensure that the read() operation only uses the intended process's memory and avoids any potential damage to other processes.

System call monitoring with GDB



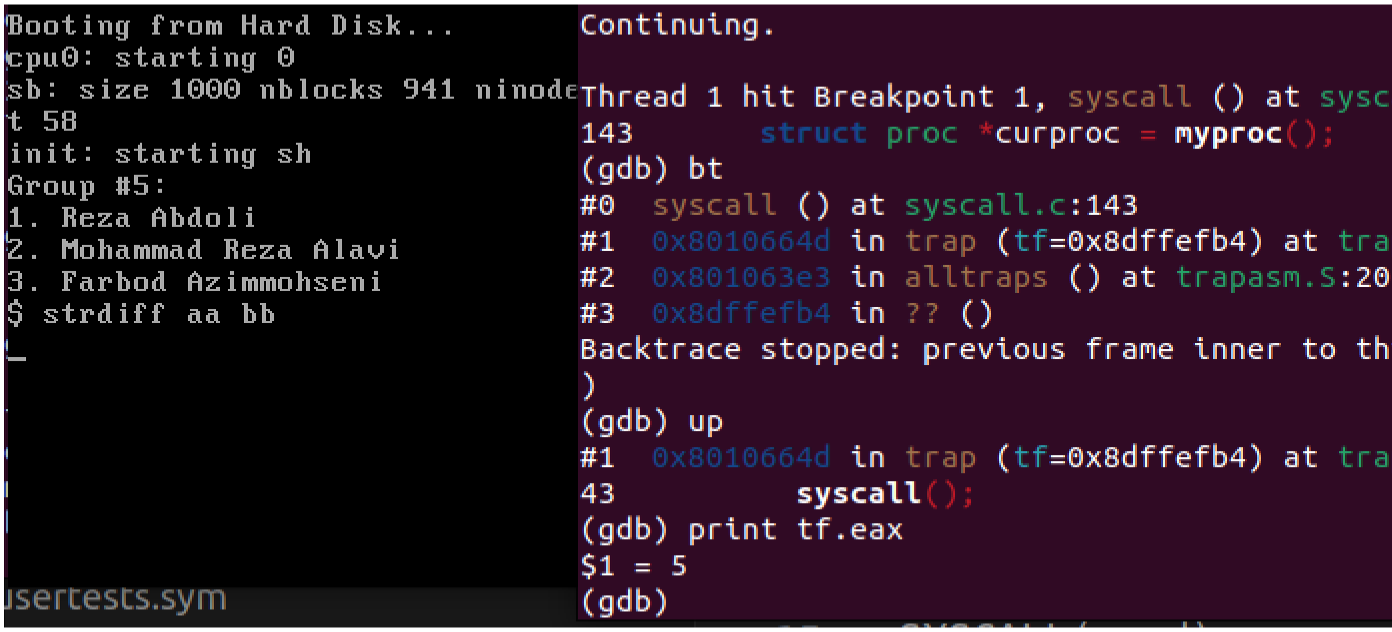
```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninode
t 58
init: starting sh
Group #5:
1. Reza Abdoli
2. Mohammad Reza Alavi
3. Farbod Azimmohseni
$ strdiff aa bb
_
usertests.sym
```

```
Continuing.

Thread 1 hit Breakpoint 1, syscall () at sysc
143         struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:143
#1  0x8010664d in trap (tf=0x8dffefb4) at tra
#2  0x801063e3 in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to th
)
(gdb) up
#1  0x8010664d in trap (tf=0x8dffefb4) at tra
43              syscall();
(gdb) print tf.eax
$1 = 5
(gdb)
```

Figure 2: system call trap

bt shows the function call stack. You can go into nested calls with **up** and **down** commands.



```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninode
t 58
init: starting sh
Group #5:
1. Reza Abdoli
2. Mohammad Reza Alavi
3. Farbod Azimmohseni
$ strdiff aa bb
_
```

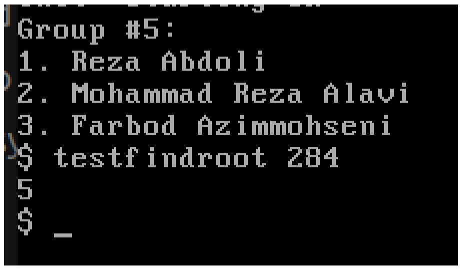
```
Continuing.

Thread 1 hit Breakpoint 1, syscall () at sysc
143         struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:143
#1  0x8010664d in trap (tf=0x8dffefb4) at tra
#2  0x801063e3 in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
Backtrace stopped: previous frame inner to th
)
(gdb) up
#1  0x8010664d in trap (tf=0x8dffefb4) at tra
43            syscall();
(gdb) print tf.eax
$1 = 5
(gdb)
```

Figure 3: process getting trapped to kernel

System call blocks the process and scheduler runs other processes. That is the reason we need to continue several times.

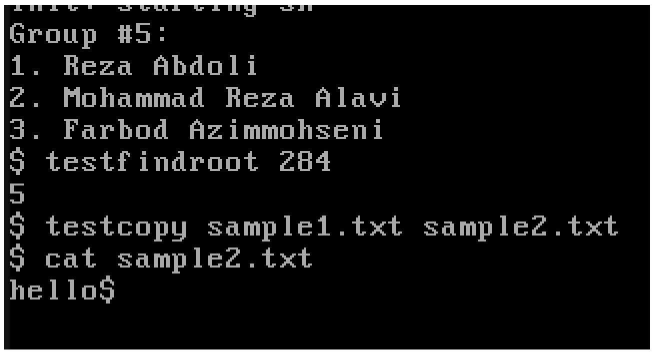
Digital Root



```
Group #5:
1. Reza Abdoli
2. Mohammad Reza Alavi
3. Farbod Azimmohseni
$ testfindroot 284
5
$ _
```

Figure 4: digital root

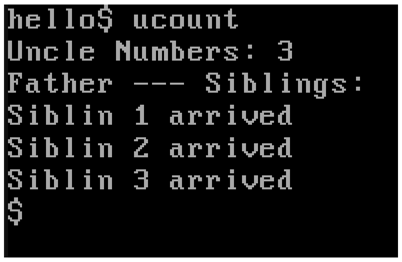
Copy File



```
Group #5:
1. Reza Abdoli
2. Mohammad Reza Alavi
3. Farbod Azimmohseni
$ testfindroot 284
5
$ testcopy sample1.txt sample2.txt
$ cat sample2.txt
hello$
```

Figure 5: copy

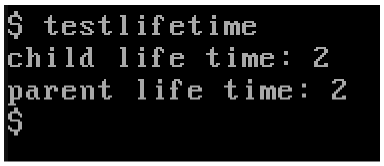
Uncle Count



```
hello$ ucount
Uncle Numbers: 3
Father --- Siblings:
Siblin 1 arrived
Siblin 2 arrived
Siblin 3 arrived
$
```

Figure 6: uncle count

Process Lifetime



```
$ testlifetime
child life time: 2
parent life time: 2
$
```

Figure 7: process lifetime