CS3500 : Operating Systems
Lab 3 : System Calls

21$^{\text{th}}$ August 2020

In this lab you will add some new system calls to xv6, which will help you understand how they work and will expose you to some of the internals of the xv6 kernel. You will add more system calls in later labs.
The goals of this assignment are:

- Understand the system call interface
- Understand how user programs send parameters to the kernel, and receive values back

Resources:

Before you start coding, read Chapter 2 of the xv6 book, and Sections 4.3 and 4.4 of Chapter 4, and related source files:

1. user-space code for systems calls is in *user/user.h* and *user/usys.pl*
2. kernel-space code is *kernel/syscall.h*, *kernel/syscall.c*
3. The process-related code is *kernel/proc.h* and *kernel/proc.c*

Problem 1: 10 points

In this problem you will create a new system call: *echo_simple*, which should receive one argument, a string, and print it to stdout. As part of this problem, you should write a user program *test_problem_1* that invokes the *echo_simple* system call.

```
$ test_problem_1 Hello
Hello
```

Your solution is correct if your program behaves as shown above (though the inputs may be different).

Some hints:

- Add *$U/_test_problem_1* to *UPROGS* in *Makefile*
- Run *make qemu* and you will see that the compiler cannot compile *user/test_problem_1.c*, because the user-space stubs for the system call don't exist yet: add a prototype for the system call to *user/user.h*, a stub to *user/usys.pl*, and a syscall number to *kernel/syscall.h*. The *Makefile* invokes the perl script *user/usys.pl*, which produces *user/usys.S*, the actual system call stubs, which use the RISC-V *ecall* instruction to transition to the kernel. Once you fix the compilation issues, run *test_problem_1*; it will fail because you haven't implemented the system call in the kernel yet.

- Add a *sys_echo_simple()* function in *kernel/sysproc.c* that implements the new system call. The functions to retrieve system call arguments from user space are in *kernel/syscall.c*, and you can see examples of their use in *kernel/sysproc.c*.

Problem 2 : 20 points

echo is a built-in user command that writes its arguments to standard output. In this problem you will create a new system call: *echo_kernel*, that is similar to *echo* user command in its functionality, but executes in Kernel space rather than user space. As part of this problem, you should write a user program *test_problem_2* that invokes the *echo_kernel* system call.

```
$ test_problem_2 Hello World is passed
Hello World is passed
```

Your solution is correct if your program behaves as shown above (though the inputs may be different).

Some hints:

- Add *$U/_test_problem_2* to *UPROGS* in *Makefile*
- Run *make qemu* and you will see that the compiler cannot compile *user/test_problem_2.c*. Add the system call *echo_kernel* following the same steps as in previous problem. Once you fix the compilation issues, run *test_problem_2*; it will fail because you haven't implemented the system call in the kernel yet.
- Add a *sys_echo_kernel()* function in *kernel/sysproc.c* that implements the new system call. It should have the same functionality as echo() function. (see user/echo.c)

Problem 3: 20 points

In this problem you will create a new *get_process_info* system call that returns the Process ID, Process Name and the Size of process memory. As part of this problem, you should write a user program *test_problem_3* that invokes the *get_process_info* system call. The system call takes one argument: a pointer to a *struct processinfo* (see *processinfo.h*). The kernel should fill out the fields of this struct.

```
$ test_problem_3
Process ID -> 23
Process Name -> test_problem_3
Memory Size -> 2405 Bytes
```

Your solution is correct if your program behaves as shown above (though the output may be different).

Some hints:

- Add *$U/_test_problem_3* to *UPROGS* in *Makefile*
- Run *make qemu*; *user/test_problem_3.c* will fail to compile. Add the system call *get_process_info*, following the same steps as in the previous problem. To declare the prototype for *get_process_info()* in *user/user.h* you need predeclare the existence of *struct processinfo*:

  ```
  struct processinfo;
  int get_process_info(struct processinfo *);
  ```

Once you fix the compilation issues, run *test_problem_3*; it will fail because you haven't implemented the system call in the kernel yet.

- *get_process_info* needs to copy a *struct processinfo* back to user space; see *sys_fstat()* (*kernel/sysfile.c*) and *filestat()* (*kernel/file.c*) for examples of how to do that using *copyout()*.
- To fill the *processinfo* structure, make use of the *proc* structure (see *kernel/proc.h*).

Problem 4: 20 points

In this problem you will add a system call tracing feature that may help you when debugging later labs. You'll create a new system call: *trace*, that will control tracing. It should take one argument, an integer "mask", whose bits specify which system calls to trace. For example, to trace the *read* system call, a program calls *trace (1 << SYS_read)*, where *SYS_read* is a syscall number from *kernel/syscall.h*. You have to modify the xv6 kernel to print out a line when each system call is about to return, if the system call's number is set in the mask. The line should contain the process id, the name of the system call and the return value; you don't need to print the system call arguments. The *trace* system call should enable tracing for the process that calls it, but should not affect other processes.
We provide a *trace* user-level program that runs another program with tracing enabled (see *user/trace.c*). When you're done, you should see output like this:

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 959
3: syscall read -> 0
$
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 959
4: syscall read -> 0
4: syscall close -> 0
$
$ grep hello README
$
```

In the first example above, *trace* invokes *grep* tracing just the *read* system call. The 32 is 1<<*SYS_read*. In the second example, *trace* runs *grep* while tracing all system calls; the 2147583647 has all 31 low bits set. In the third example, the program isn't traced, so no trace output is printed. Your solution is correct if your program behaves as shown above (though the process IDs may be different).

Some hints:

- Add *$U/_trace* to *UPROGS* in *Makefile*
- Run *make qemu* and you will see that the compiler cannot compile *user/trace.c*. Add the system call *trace* following the same steps as in the previous problem. Once you fix the compilation issues, run *trace 32 grep hello README*; it will fail because you haven't implemented the system call in the kernel yet.

- Add a *sys_trace()* function in *kernel/sysproc.c* that implements the new system call by remembering its argument in a new variable in the *proc* structure (see *kernel/proc.h*).
- Modify the *syscall()* function in *kernel/syscall.c* to print the trace output.

Challenge Problem 1 : 30 points (Optional)

Extend the *trace* system call by printing the system call arguments for the traced system calls.

Lab Workflow:

1. Copy the *trace.c* file (provided as part of the question) to *user/* directory and *processinfo.h* file to *kernel/* directory
2. Solve the problems
3. Answer the *Lab_3_Questionnaire* (provided as part of the question).
4. Once your ready to submit, run ***make clean*** and zip the answered Lab_3_Questionnaire along with the entire *xv6-riscv* repo, containing your solutions and upload it on Moodle. Follow the naming convention as *<ROLLNO>_<LABNO>.<EXTENSION>*.

```
$ tar -cvzf XXX_XXX.tar.gz xv6-riscv/* Lab_3_Questionnaire.docx
```

Note: You need not submit any Writeup. Ensure that you have zipped the filled Questionnaire along with the solution repo.