

OPERATING SYSTEM

PROJECT 2 – System Call

1. General rule:

- The project is done in groups: each group has a maximum of **3 students**
- **The same exercises will all be scored 0 for the entire practice (even though there are scores for other exercises and practice projects).**
- Operating System: **Linux**

2. Submission:

Submit assignments directly on the course website (MOODLE), not accepting submissions via email or other forms.

Filename: **StudentID1_StudentID2_StudentID3.zip** (with StudentID1 < StudentID2 < StudentID3)

Ex: Your group has 2 students: 2312001, 2312002 and 2312003, the filename is:
2312001_2312002_2312003.zip

Include:

- **StudentID1_StudentID2_StudentID3_Report.pdf:** Writeups should be short and sweet. Do not spend too much effort or include your source code on your writeups. The purpose of the report is to give you an opportunity to clarify your solution, any problems with your work, and to add information that may be useful in grading. If you had specific problems or issues, approaches you tried that didn't work, or concepts that were not fully implemented, then an explanation in your report may help us to assign partial credit
- **Release:** File diff (diff patch, Ex: \$ git diff > **StudentID1_StudentID2_StudentID3.patch**)
- **Source:** Zip file of xv6 (the version is made clean)

3. Demo Interviews

Your implementation is graded on completeness, correctness, programming style, thoroughness of testing, your solution, and code understanding.

When administering this course, we do our best to give a fair assessment to each individual based on each person's contribution to the project

4. Requirements

In the last lab you used system calls to write a few utilities. 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.

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:

- The user-space "stubs" that route system calls into the kernel are in `user/usys.S`, which is generated by `user/usys.pl` when you run `make`. Declarations are in `user/user.h`
- The kernel-space code that routes a system call to the kernel function that implements it is in `kernel/syscall.c` and `kernel/syscall.h`.
- Process-related code is `kernel/proc.h` and `kernel/proc.c`.

To start the lab, switch to the `syscall` branch:

```
$ git fetch
$ git checkout syscall
$ make clean
```

If you run `make grade` you will see that the grading script cannot `exec trace` and `sysinfotest`. Your job is to add the necessary system calls and stubs to make them work.

4.1. System call tracing

In this assignment you will add a system call tracing feature that may help you when debugging later labs. You'll create a new `trace` system call 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 `fork` system call, a program calls `trace(1 << SYS_fork)`, where `SYS_fork` 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 and any children that it subsequently forks, 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 -> 966
3: syscall read -> 70
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 -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
$
$ grep hello README
$
$ trace 2 usertests forkforkfork
usertests starting
test forkforkfork: 407: syscall fork -> 408
408: syscall fork -> 409
409: syscall fork -> 410
410: syscall fork -> 411
409: syscall fork -> 412
410: syscall fork -> 413
409: syscall fork -> 414
411: syscall fork -> 415
...
$

```

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 2147483647 has all 31 low bits set.

In the third example, the program isn't traced, so no trace output is printed.

In the fourth example, the fork system calls of all the descendants of the forkforkfork test in usertests are being traced. 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`, 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 `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**). 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`.
- Modify `fork()` (see `kernel/proc.c`) to copy the trace mask from the parent to the child process.
- Modify the `syscall()` function in `kernel/syscall.c` to print the trace output. You will need to add an array of syscall names to index into.
- If a test case passes when you run it inside `qemu` directly but you get a timeout when running the tests using `make grade`, try testing your implementation on Athena. Some of tests in this lab can be a bit too computationally intensive for your local machine (especially if you use WSL).

Challenge: Print the system call arguments for traced system calls.

4.2. Sysinfo

In this assignment you will add a system call, `sysinfo`, that collects information about the running system. The system call takes one argument: a pointer to a struct `sysinfo` (see `kernel/sysinfo.h`). The kernel should fill out the fields of this struct: the `freemem` field should be set to the number of bytes of free memory, and the `nproc` field should be set to the number of processes whose state is not `UNUSED`. We provide a test program `sysinfotest`; you pass this assignment if it prints `"sysinfotest: OK"`.

Some hints:

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

```
struct sysinfo;
int sysinfo(struct sysinfo *);
```

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

- `sysinfo` needs to copy a struct `sysinfo` 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 collect the amount of free memory, add a function to `kernel/kalloc.c`
- To collect the number of processes, add a function to `kernel/proc.c`

Challenge: Compute the load average and export it through `sysinfo`.

5. Grade

No.	Exercise	Grade
1	tracing	5
2	sysinfo	5

6. Reference

- <https://pdos.csail.mit.edu/6.1810/2023/labs/syscall.html>