CSE 3025 Principles of Operating Systems
Fall 2025
Lab 2 xv6 System Calls

This lab is to be completed individually. Due by the end of Tuesday, Sept. 16, 2025.

In the last lab, you used system calls to write a few user-level programs. 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 in `kernel/proc.h` and `kernel/proc.c`.

First, clone a new copy of xv6 for lab 2:

```
$mkdir xv6lab2
$cd xv6lab2
$git clone https://github.com/mit-pdos/xv6-riscv.git
```

Exercise 1: Using gdb (20 pts)

In many cases, print statements will be sufficient to debug your kernel, but sometimes it is useful to single step through code or get a stack back-trace. The GDB debugger can help.

To help you become familiar with gdb, run `make qemu-gdb` and then fire up gdb (`gdb-multiarch`) in another window (see the gdb material on the guidance page). Once you have two windows open, type in the gdb window (you may need to type `source .gdbinit` before the following steps):

```
(gdb) b syscall
Breakpoint 1 at 0x80002142: file kernel/syscall.c, line 243.
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, syscall () at kernel/syscall.c:243
243     {
(gdb) layout src
(gdb) backtrace
```

The layout command splits the window in two, showing where gdb is in the source code. backtrace prints a stack backtrace.

Answer the following questions in ***answers-Lab2-firstname-lastname.txt***.

Q1: Looking at the backtrace output, which function called syscall?

Type `n` a few times to step past `struct proc *p = myproc();` Once past this statement, type `p /x *p`, which prints the current process's `proc struct` (see `kernel/proc.h`) in hex.

Q2: What is the value of `p->trapframe->a7` and what does that value represent? (Hint: look `user/initcode.S`, the first user program xv6 starts.)

The processor is running in supervisor mode, and we can print privileged registers such as `sstatus` (see RISC-V privileged instructions for a description):

```
(gdb) p /x $sstatus
```

Q3: What was the previous mode that the CPU was in?

The xv6 kernel code contains consistency checks whose failure causes the kernel to panic; you may find that your kernel modifications cause panics. For example, replace the statement `num = p->trapframe->a7;` with `num = * (int *) 0;` at the beginning of `syscall`, run `make qemu`, and you will see something similar to:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
scause=0xd sepc=0x80001bfe stval=0x0
panic: kerneltrap

Quit out of qemu.
```

To track down the source of a kernel page-fault panic, search for the `sepc` value printed for the panic you just saw in the file `kernel/kernel.asm`, which contains the assembly for the compiled kernel.

Q4: Write down the assembly instruction the kernel is panicing at. Which register corresponds to the variable `num`?

To inspect the state of the processor and the kernel at the faulting instruction, fire up gdb, and set a breakpoint at the faulting `epc`, like this:

```
(gdb) b *0x80001bfe
Breakpoint 1 at 0x80001bfe: file kernel/syscall.c, line 138.
(gdb) layout asm
(gdb) c
```

```
Continuing.
[Switching to Thread 1.3]

Thread 3 hit Breakpoint 1, syscall () at kernel/syscall.c:138
```

Confirm that the faulting assembly instruction is the same as the one you found above.

Q5: Why does the kernel crash? Hint: look at figure 3-3 in the text; is address 0 mapped in the kernel address space? Is that confirmed by the value in `scause` above? (See description of scause in RISC-V privileged instructions)

Note that `scause` was printed by the kernel panic above, but often you need to look at additional info to track down the problem that caused the panic. For example, to find out which user process was running when the kernel paniced, you can print the process's name:

```
(gdb) p p->name
```

Q6: What is the name of the process that was running when the kernel paniced? What is its process id (pid)?

You may want to visit Using the GNU Debugger as needed. The debugging tips can be found on Canvs.

Exercise 2: System call tracing (35 pts)

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 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.

A user-level program, `trace.c`, that runs another program with tracing enabled is provided. You can download it from Canvas and place it in the `user` directory. 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:
(1) Add `$U/_trace` to `UPROGS` in Makefile.
(2) Run `make qemu` and you will see that the compiler cannot compile `user/trace.c`, because the user-space stubs for the trace system call don't exist yet: add a prototype for trace 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.
(3) 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`. Add your new `sys_trace` to the syscalls array in `kernel/syscall.c`.
(4) Modify `kfork()` (see `kernel/proc.c`) to copy the trace mask from the parent to the child process.

(5) 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.

Exercise 3: Nice Values of Processes (35 pts)

In this assignment, you need to add two new system calls to xv6, which set and get the nice value of a process, indicating the priority level of the process. A high nice value means low priority. In Linux, the nice value is between +19 (lowest priority) and -20 (highest priority). You will need to store the nice value in the process structure. Note that xv6 stores the process control block (PCB, or PEB in Windows parlance, or task struct in Linux parlance) in `struct proc` in the file `kernel/proc.h`.

Simplification: You do not need to enforce permissions on nice. On a typical Linux system, only root can increase priorities (lower nice values); to keep life simple for this assignment, you can skip this part of the specification.

Create a system call `int set_priority(int pid, int priority)`, which sets the nice value of a process with process ID pid to `priority`. `priority` is a value in the range 0 to 39. Attempts to set a nice value out of the range are silently clamped to the range. If the call is successful, return 0. Otherwise, return -1. Create a system call `int get_priority(int pid)`, which gets the nice value of the process with process ID `pid`. If the call is successful, return the nice value of the process. Otherwise, return -1.

You will also write a user-level application called `lab2e3test.c` to test the two system calls. The test program should be able to test the following things: (1) set the nice value of a process; (2) get the nice value of a process and print it out.

Some hints:
(1) In Linux, the `getpriority()` call returns -1 on error for the nice value in the range of -20 to 10. You can check http://man7.org/linux/man-pages/man2/getpriority.2.html on how Linux solves the issue that -1 can be a legitimate return value.
(2) When a new process is created, its nice value should be the default value 20. You need to modify the `kfork()` call in `kernel/proc.c` to do this.

What to submit

In `time-logging-lab2-firstname-lastname.txt`, please include your time logging for this lab and a few words of explanation for each of the files you had to modify for exercises 2 and 3 (10 points).

AFTER committing all changes, you run `make clean`, and `tar zcf ../lab2-firstname-lastname.tar.gz ../xv6lab2` assuming your xv6 source directory is `xv6lab2`. Then upload the complete tarred and compressed xv6 directory (`lab2-firstname-lastname.tar.gz`) through the submission link on Canvas.