Lab: system calls

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.

Using gdb (easy)

In many cases, print statements will be sufficient to debug your kernel, but sometimes being able to single step through some assembly code or inspecting the variables on the stack is helpful.

To learn more about how to run GDB and the common issues that can arise when using GDB, check out this page.

To help you become familiar with gdb, run **`make qemu-gdb`** and then fire up gdb in another window (see the gdb bullet on the guidance page). Once you have two windows open, type in the gdb window:

```
(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. The `backtrace` prints out the stack backtrace. See Using the GNU Debugger for helpful GDB commands.

Answer the following questions in `answers-syscall.txt`.

> 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.

> 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 kernel mode, and we can print privileged registers such as sstatus (see RISC-V privileged instructions for a description):

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

> What was the previous mode that the CPU was in?

In the subsequent part of this lab (or in following labs), it may happen that you make a programming error that causes the xv6 kernel to panic. 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 0x000000000000000d
sepc=0x000000008000215a stval=0x0000000000000000
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.

> 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 *0x000000008000215a
Breakpoint 1 at 0x8000215a: file kernel/syscall.c, line 247.
(gdb) layout asm
(gdb) c
Continuing.
[Switching to Thread 1.3]

Thread 3 hit Breakpoint 1, syscall () at kernel/syscall.c:247
```

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

> 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 out the process's name:

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

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

This concludes a brief introduction to tracking down bugs with gdb; it is worth your time to revisit Using the GNU Debugger when tracking down kernel bugs. The guidance page also has some other other useful debugging tips.

System call tracing (moderate)

> 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).

Sysinfo (moderate)

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`

Submit the lab

Time spent

Create a new file, `time.txt`, and put in a single integer, the number of hours you spent on the lab. `git add` and `git commit` the file.

Answers

If this lab had questions, write up your answers in `answers-*.txt`. `git add` and `git commit` these files.

Submit

You will turn in your assignments using the google classroom.

After committing your final changes to the lab, type `make tarball` to create archive which you'll submit to classroom.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make tarball
$
```

If you run `make tarball` and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
M hello.c
M bar.c

You have uncomitted changes.  Please commit or stash them.
make: *** [Makefile:348: handin-check] Error 1
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with `M`. You can cause `git` to track a new file that you create using `git add filename`.

If `make tarball` does not work properly, try fixing the problem with the Git commands.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make tarball`

Optional challenge exercises

- Print the system call arguments for traced system calls (easy).
- Compute the load average and export it through sysinfo(moderate).