Lab Syscall

Get familiar with xv6's system calls, use them, and create new one!

Environment Preparation

Follow this guide to install the required tools for labs.

It's highly recommended to finish the labs under Linux environment. You can use **Windows Subsystem for Linux** (on Windows) or **Virtual Machine** (on macOS). It's not recommended to install the RISC-V toolchain on macOS, because it will take a lot of time.

Now download codes for the labs using `git`:

```
$ git clone https://github.com/chengjiagan/SJTU-CS2953-Spring.git
$ git checkout syscall
$ cd SJTU-CS2953-Spring && make clean
```

Git

Our labs use `git`. Any modification you make should be committed to `git` so that it would be uploaded by our scripts. You can use the following commands to commit the modified files:

```
$ git add <files being modified>
$ git commit -m "finish task X"
$ # check if every modification is commited
$ git status
```

(you can learn more about `git` from this link)

Lab Tasks

sleep

> Implement the UNIX program `sleep` for xv6; your `sleep` should pause for a user-specified number of ticks. A tick is a notion of time defined by the xv6 kernel, namely the time between two interrupts from the timer chip. Your solution should be in the file `user/sleep.c`.

Run the program from the xv6 shell:

```
$ make qemu
...
init: starting sh
$ sleep 10
(nothing happens for a little while)
$
```

Some hints:

- Look at some of the other programs in `user/` (e.g., `user/echo.c`, `user/grep.c`, and `user/rm.c`) to see how you can obtain the command-line arguments passed to a program.
- If the user forgets to pass an argument, sleep should print an error message.
- The command-line argument is passed as a string; you can convert it to an integer using `atoi` (see user/ulib.c).
- Use the system call `sleep`.
- See `kernel/sysproc.c` for the xv6 kernel code that implements the `sleep` system call (look for `sys_sleep`), `user/user.h` for the C definition of `sleep` callable from a user program, and `user/usys.S` for the assembler code that jumps from user code into the kernel for `sleep`.
- `main` should call `exit(0)` when it is done.
- Add your `sleep` program to `UPROGS` in Makefile; once you've done that, `make qemu` will compile your program and you'll be able to run it from the xv6 shell.

pingpong

> Write a program that uses UNIX system calls to "ping-pong" a byte between two processes over a pair of pipes, one for each direction. The parent should send a byte to the child; the child should print ": received ping", where is its process ID, write the byte on the pipe to the parent, and exit; the parent should read the byte from the child, print ": received pong", and exit. Your solution should be in the file `user/pingpong.c`.

Run the program from the xv6 shell and it should produce the following output:

```
$ make qemu
...
init: starting sh
$ pingpong
4: received ping
3: received pong
$
```

Some hints:

- Use `pipe` to create a pipe.
- Use `fork` to create a child.
- Use `read` to read from a pipe, and `write` to write to a pipe.
- Use `getpid` to find the process ID of the calling process.
- Add the program to `UPROGS` in Makefile.
- User programs on xv6 have a limited set of library functions available to them. You can see the list in `user/user.h`; the source (other than for system calls) is in `user/ulib.c`, `user/printf.c`, and `user/umalloc.c`.

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.

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`

Submit the Lab

Report

Put your report about this lab in a new file `report.txt`. You can write down the errors you met when doing the lab, or some techniques you found that helps you to finish the lab better. You can use both **English** and **Chinese**.

Don't forget to `git add` and `git commit` the file.

Time spent

Create a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.

Check

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

Submit

Run `make tarball`, you should see a new file in the root directory called `lab-syscall-handin.tar.gz`. Upload this file to the corresponding section in course canvas page.