# Lab: system calls

In the last lab you used systems 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 code for systems calls is in `user/user.h` and `user/usys.pl`.
- The kernel-space code is `kernel/syscall.h`, kernel/syscall.c.
- The 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.

## 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 2147583647 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

**This completes the lab.** Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.Time spentCreate 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.SubmitYou will turn in your

assignments using the **submission website**. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type make handin to submit your lab.

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

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting
https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time
  Time  Current
                                 Dload  Upload   Total   Spent
   Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:--
--:--:--  276k
$
```

make handin will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let make handin generate it again (*myapi.key* must not include newline characters).

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

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

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 ??. You can cause `git` to track a new file that you create using `git add filename`.

If make handin does not work properly, try fixing the problem with the curl or Git commands. Or you can run make tarball. This will make a tar file for you, which you can then upload via our **web interface**.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at **https://6828.scripts.mit.edu/2020/handin.py/**