Computer Science 432
Spring 2022
Lab 3: System Calls, due next Monday before lab.

Objective. To implement one or two new system calls.

Discussion. This week we will be extending the kernel to include some new system calls. We add new system calls to the kernel when we're interested in improving *the kernel's interface*. There are lots of reasons why this might be necessary. Adding new system calls is a challenge and, in the end, a huge responsibility.

Access to the kernel is difficult. It is a long and winding road through various gates that make sure that the user's rogue behavior is not able to harm or *wreck* the kernel. Our interest, here, is to understand how to improve services for users while not compromising the kernel by violating its constraints.

Let's imagine that you're calling `open(path,access)`. This is a typical service that the kernel provides its users. Here's what happens when we call `open(2)`:

1. In user space, the `open(2)` entry-point is defined in `user/user.h` as follows:

   ```
   int open(const char*, int);  // open a file, returning a descriptor
   ```

   The actual definition of this call is an assembly routine, found in `user/usys.S`:[1]

   ```
   # generated by usys.pl - do not edit
      ...
           .global open
   open:
           li a7, SYS_open     # SYS_open is defined in kernel/syscall.h
           ecall               # "System entry call"; jumps to supervisor mode
           ret
   ```

   The value of `SYS_open` (defined as `15` in the kernel), is saved in register `a7` and then the `ecall` instruction causes the machine to enter supervisor mode.

2. The machine trap mechanism (eventually) calls `syscall` (see `kernel/syscall.c`). This "dispatch" routine carefully routes the call to the appropriate routine found in the `syscalls` vector. In this case, it looks in `syscalls[SYS_open]` and finds the pointer to the routine `sys_open`. These system routines are located throughout the kernel, typically in the area that is most appropriate to manage the particular service.

   While we're here, notice that *every* system call is routed through this routine. If we're interested in observing or controlling access to the kernel, this is good place to do it.

3. The `sys_open` routine is found in `kernel/sysfile.c`. This routine is responsible for accepting the parameters from user space and returning an appropriate value after the service has been completed. Because the user cannot be trusted, the parameters (a string and an integer) are manually copied from the user's address space into the kernel. This is accomplished by "copy-in/copy-out" data portage routines. Here's what the entry and exit of `sys_open` routine looks like:

   ```
   uint64
   sys_open(void)
   {
     char path[MAXPATH];
     int fd, omode;
     int n;

     if((n = argstr(0, path, MAXPATH)) < 0 || argint(1, &omode) < 0)
       return -1;

     ... code for opening a file at a path ...

     return fd;
   }
   ```

[1] The uppercase `.S` extension is a signal to the `gcc` compiler that it should run the C pre-processor on the file before assembly, allowing you to define values and include header files. Indeed, `usys.S` includes `kernel/syscall.h`.

The `argstr` and `argint` routines are responsible for the copy-in portage of data from user to kernel space. It's vitally important that we use these routines because (1) the incoming arguments are stored in *user* registers and (2) because they will verify the structure of the arguments. For example, `argstr` will make certain that the pointer is valid in user space and does fool the kernel into copying data from a part of the system the user cannot access. Every system call, of course, makes use of different calling signatures, so each will make use of different portage commands. Here's a short list to choose from:

Routine	**Purpose**
`argraw(n)`	Fetch the n-th argument as a 64-bit integer.
`argint(n,ip)`	Fetch the n-th argument as a 32-bit integer through reference `ip`.
`argaddr(n,ap)`	Fetch the n-th argument as a 64-bit user pointer through pointer `ap`.
`argstr(n,buf,max)`	Fetch the n-th argument, a C string, `max` character kernel `buf`.
`argfd(n,fdp,filep)`	Fetch the n-th argument, a descriptor, return it and its `struct file`.
`copyout(upgtbl,uptr,kptr,s)`	Copy s bytes from kernel address kptr to user address uptr.

The source for these routines is found in `kernel/syscall.c`. It is worth looking at the source code.

4. Notice that `sys_open` returns a `uint64`. Typically this is a non-negative file descriptor, but if there are problems with the system call, it returns a negative number. When `sys_open` returns, its return value is captured by `syscall`, and returned in the *user's* `a0` register. This linkage—through the *trapframe*—will be discussed in Chapter 4. We'll get there soon enough.

5. In any case, the trap mechanisms eventually perform a downgrade of privilege and a return of the system call result in `a0`, with an `sret`. We're now in the user-space `open`, which immediately returns `a0` to the user. Notice that the user interprets `a0` as a signed 32-bit integer.

We will want to make sure the linkage at each step works correctly. Failure, here, is not an option.

The Assignment. We'll implement two system calls in `xv6`. The basis for this lab is found in the `lab3` repository:

```
$ git clone ssh://22xyz@lohani.cs.williams.edu/~cs432/22xyz/lab3.git
```

The repository contains a fairly stripped-down version of `xv6` with a few modifications to support this lab. We're starting fresh; there's not much utility in carrying-forward utilities from one lab to the next.

Assuming you're up to speed on the readings, here is a workflow that will get you through this week's tasks:

1. Our first task is to write, `trace`, a utility for tracing system calls made by user processes.

   Here is a typical user process: we look for `hello` in the file `README`:

   ```
   $ grep hello README
   $
   ```

   It's not there. Still there was lots going on.

   Our `trace` command will accept a hexadecimal bit mask that identifies which system calls you wish to follow during an execution. Here, the `read(2)` system call is traced. As they happen, each system call is logged, along with its return value. Internally, `read` has system call number 5 (see `kernel/syscall.h`), so we provide a hex bit mask with bit 5 set:

   ```
   $ trace 20 grep hello README
   3: read -> 1023
   3: read -> 968
   3: read -> 235
   3: read -> 0
   $
   ```

   The value before the colon is the process id of the process reporting the kernel call. The identified system calls originate at line 17 of `user/grep.c`. The return values, remember, are the number of bytes actually read. Clearly, the `grep` process is finished when there is no more data to be read from `README`.

Here is a more significant trace. It traces any system call:

```
$ trace ffffffe grep hello README
8: trace -> 0
8: exec -> 3
8: open -> 3
8: read -> 1023
8: read -> 968
8: read -> 235
8: read -> 0
8: close -> 0
$
```

Here, we see the `trace` system call is returning. The `trace` program then `exec`s the `grep` executable. The `grep` utility must, of course, `open` (and later `close`) the `README` file. We're seeing that that file is accessed through descriptor 3.

While there are only system call numbers between 1 and 21 or 22, this hex mask (covering calls 1 through 31) is a good general purpose use case for the `trace` utility.

Here is an approach to implementing `trace`:

(a) I've written the `trace` utility in `user/trace.c`. You should update the `UPROGS` variable in `Makefile` to include this in `xv6`.

(b) Try to make `xv6`. Notice it cannot compile `trace.c` because there is no `trace` system call.

(c) Pick an appropriate non-zero system call number for `trace`. These numbers are found in `kernel/syscall.h`. (Do not, by the way, put anything other than `define` statements in that file; it is included in both C and assembly code.)

(d) Add a prototype for the `trace` call to `user/user.h`. You will see an area dedicated to system calls. The `trace` call takes a `uint32` and returns an `int`.

(e) Add a stub to `user/usys.pl`. This is a `perl` script that automatically generates the file `user/usys.S`. From the top directory, you should now be able to:

```
$ make user/usys.S
```

This will create the `ecall` linkage to the system. Inspect `user/usys.S` to ensure this is the case.

(f) Now, we'll allow every process to keep track of the dedicated trace mask for that process. This information should be stored in the `struct proc` definition, found in `kernel/proc.h`. I suggest adding the declaration to the end of the definition so that we don't disturb the relative offsets of other fields. There should be no problem moving the fields around, but let's minimize the number of things that might go wrong.

(g) Write the system call, `uint64 sys_trace(void)`, and place it in `kernel/sysproc.c`. This routine should retrieve its (user) argument and place it in the field you've reserved in the `struct proc` pointed to by the result of calling `myproc()`. You can see how other system calls do this in the same file you're placing `sys_trace`.

(h) We need to make sure that each process maintains a reasonable value for your trace mask. In the file `kernel/proc.c`, we find the routines that maintain processes. Two are important to us: `allocproc`, which allocates a new process from the process table, and `fork`, which effectively copies process state to a new process. In `allocproc`, we need to initialize the process state mask appropriately. In `fork`, we need to `copy` the trace mask from the parent to the new child process.

    Be aware that we are playing with a part of the operating system that may be concurrently accessed by several cores at once. Make sure that you respect the local locking rules. My advice is to keep your manipulations of process code near similar manipulations of other fields in the process structure.

(i) Now, we need to provide the final connection—from `syscall` to `sys_trace`. In `kernel/syscall.c`, find the `syscall` system call dispatcher. Notice, again, how it *vectors through* a table of pointers to routines. Update the table to include the trace system call. Notice the syntax of the array initialization: each entry specifies where in the statically allocated array the pointer is stored. Edit this carefully.

(j) At this point you should be able to build `xv6`. Running the `trace` program will simply run the subordinate command. That's because we haven't actually *interpreted* the trace mask anywhere. We'll do that in the next step.

(k) In the `syscall` dispatch method, capture the return values from system calls and, if when necessary, print the process id, the name of the system call, and the return value from the system call. You may use `printf`, here, but you will have to build a table that translates system call numbers to system call names. Declare that table using the `syscalls` vector declaration as a model.

(l) Build the system and test your implementation of the `trace` program.

2. Now, write a new system call, `sysinfo`. This call collects and reports information about the running system. It takes one argument, a pointer to a `struct sysinfo`, (see `kernel/sysinfo.h`). The kernel should fill in the fields of this structure: 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`.

   Here is the rough outline of an approach:

   (a) I've provided a program, `sysinfo.c`, that you can use to exercise your system call. You should modify the build system to include this program.

   (b) You will need to declare the userland version of the system call, `sysinfo`:

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

   Notice that you will *not* need to add an include the `struct sysinfo` definition here; the compiler only needs to know that the pointer should be to a `struct sysinfo`. In the `sysinfo.c` program, however, I *did* need to include the definition since I'm interpreting the results of your system call.

   With a little more careful work, on the user side, you should be able to build the system. Using the `sysinfo` program, however, will fail since there is no system call in kernel space.

   (c) To collect the amount of free memory, you should add a helper function to `kernel/kalloc.c`. This function should compute the number of bytes available in the `kmem` free list. You should make sure you acquire the spinlock `kmem.lock` before you interpret the free list structure and release it immediately after. Failing to do this may cause a race condition in your routine that will be hard to survive.

   (d) To collect the number of processes whose state is *not* `UNUSED`, you should add a helper function to `kernel/proc.c`. You've been here before. It may be helpful to see how `allocproc` traverses the process list as a model for your code.

   (e) Now, write the kernel side of the `sysinfo` system call, including the routine that does all the real work, `sys_sysinfo`. Notice that the results of this routine must be shipped back to a structure in user space. You will have to carefully use the `copyout` routine (see `kernel/defs.h`) to move a chunk of memory from the kernel to user space. A good example of how this is done is in the implementation of the `fstat` system call. The routine `sys_fstat` is in `kernel/sysfile.c` and it calls `filestat` in `kernel/file.c`.

   (f) You should now be able to build everything. I've also included the `sysinfotest.c` program from MIT. You may find this useful in detecting problems with your code.

3. Test, add, commit, and push your changes for collection and review in small group meetings.

Thought Questions. Please think about the following questions before we meet in small groups.

1. When we perform

   ```
   $ trace fffffffe grep hello README
   ```

   We see a call to `exec`, but no call to `fork`. Looking at `user/trace.c` we see that's correct. Why doesn't `fork` get called?

2. There is no system call with id zero. Why do you think this is?

3. Looking at the `trace.c` assembly in `user/trace.asm`, we note that `ishex` saves the return address to the call frame, but `hextoi` does not. What is happening, here?

4. How does the MIT `sysinfotest.c` program work?

⋆