Lab 1 Design Doc: System Calls

Overview

The goal of this lab is to implement an interface for users to interact with persistent media or with other I/O devices, without having to distinguish between them.

Major Parts

File Interface: Provide an abstraction for the user that doesn't depend on the type of "file". In user space, this will allow for seamless switching of "file" without large changes in the code (ex: Reading input from a file vs reading from stdin, the method for attaining bytes will be the same).

System Calls: The system call interface provides a barrier for the kernel to validate user program input. This way, we can keep the I/O device state consistent. No user program can directly affect the state of the kernel's structure. Furthermore, when we are in the kernel code, we don't have to go through the syscall interface, which cuts down superfluous error checking for trusted code.

In-depth Analysis and Implementation

File Interface

Bookkeeping

`include/kernel/fs.h` provides a `struct file` that we can use to back each file descriptor.
`include/kernel/console.h` provides console file structs for stdin and stdout.

Process View

Each process will have an array of open files (Bounded by `PROC_MAX_FILE`) in the process struct.
The file descriptor will be the respective index into the file table.
Ex: stdin is typically file descriptor 0, so the corresponding file struct will be the first element.
A system call can use the `proc_current()`

System Calls

We need to parse arguments from the user and validate them (we never trust the user).
There are a few useful functions provided by osv:

- `bool fetch_arg(void *arg, int n, sysarg_t *ret)`:
  Given args, fetch nth argument and store it at `*ret`.
  Return true if fetched successfully, false if nth argument is unavailable
- `static bool validate_str(char *s)`:
  Given a string `s`, validates if the whole string is within a valid memory region of the process.
- `static bool validate_bufptr(void* buf, size_t size)`:
  Given a buffer `buf` of `size`, validate if the buffer is within a valid memory region of the process.

Since all our system calls will be dealing with files, we think it will be useful to
add a function that allocates a file descriptor, and another that validates a file descriptor:

- `static int alloc_fd(struct file *f)`:
  Will get a pointer to file, look through process's open file table to find an available fd, and store the pointer there. Returns the chosen fd.
- `static bool validate_fd(int fd)`:
  Will get the file descriptor, making sure it's a valid file descriptor (in the open file table for the process).

`sys_open`, `sys_read`, `sys_close`, `sys_readdir`, `sys_write`, `sys_dup`, `sys_fstat`

The main goals of the `sys_*` functions is to do argument parsing and validation, and then calling the
associated `fs_*_file` functions.

- `sys_write`, `sys_read`:
  - Writing or reading of a file.
  - Will change the `f_pos` of the respective file struct.
  - Return values specified by [lab1.html] (also in `include/lib/usyscall.h`)
- `sys_open`:
  - Finds an open spot in the process's open file table and stores the opened file by the file system.
  - Return values specified by [lab1.html] (also in `include/lib/usyscall.h`)
- `sys_close`:
  - Close the file.
  - Release the file from the process, clear out its entry in the process's open file table.
  - Return values specified by [lab1.html] (also in `include/lib/usyscall.h`)
- `sys_readdir`:
  - Reading a directory

  - Return values specified by [lab1.html] (also in `include/lib/usyscall.h`)
- `sys_dup`:
  - Will find an open spot in the process open file table and have it point to the same file struct of the first duped file.
  - Will need to update the reference count of the file through `fs_reopen_file()`.
  - Return values specified by [lab1.html] (also in `include/lib/usyscall.h`)
- `sys_fstat`:
  - Retrieve statistics of a file from `struct file` and its `struct inode`.
  - Return values specified by [lab1.html] (also in `include/lib/usyscall.h`)

fs.h

We will need to use a number of functions in fs.h. These include:
- `fs_open_file`: Used to open a file.
- `fs_reopen_file`: Used to up a file's reference count.
- `fs_read_file`: Performs a read operation on the given file.
- `fs_write_file`: Performs a write operation on the given file.

Risk Analysis

Unanswered Questions

- What happens when two different process try to update data on the file?
- What happens when the user or the kernel has the maximum number of file's open?

Staging of Work

First, I will implement the per process open file table. Then I will retrieve and validate syscall inputs, and call the respective file functions. I will also update fd 0 and 1 to be backed by console files.

Time Estimation

- File interface (__ hours)
  - Process portion (__ hours)
- System calls (__ hours)
  - sys_open (__ hours)
  - sys_read (__ hours)
  - sys_write (__ hours)

  - sys_close (__ hours)
  - sys_dup (__ hours)
  - sys_fstat (__ hours)
- Edge cases and Error handling (__ hours)