Operating Systems
COMS W4118
Lecture 5

Alexander Roth

2015 – 02 – 05

1 System Calls

1.1 `lseek`

- Deals with file descriptors, give offset, *whence* controls how to interpret the offset.
- Three offset markers `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`.

1.2 `read`

- Reads in a file from a buffer using the file descriptor.
- There is no end of file byte within a file, the file is just straight content.
- End Of File is a condition, which points to one past the last byte in the file.
- When you call `read`, the number of bytes read may be less than the requested number of bytes.
- This is mainly due to the size of the file.
- There is a notion of "slow system call." That is, a system call where it is possible for the system to block forever.
- Reading from a disk is not a "slow system call", it will take a long time though.
- Reading from a keyboard device is an example of a slow system call. It is waiting for input from a user.
- `read` call is waiting for input from anything, whether it be another driver or a keyboard.

- `read` listening on file descriptor 0 is listening for input from the keyboard.
- `pipes` also behave like files. Thus, the incoming input from one program may never happen, which would cause the second program to block forever.
- `FIFO` are just named pipes, so they behave like `pipe`s.
- For `sockets`, we use `recv` as it is socket-specific; however, we can use an equivalent version of `read` for it as well.
- In UNIX, everything is a file. There is a uniform interface of reading and writing.

1.3 `write`

- Again returns the number of bytes written to a file, it can be less than the specified size.
- Returns -1 on error, otherwise it returns the number of bytes written to the file.
- `write()` can be slow as well.
- `pipe` involves coordination between the two processes to write into the `pipe` and read from the `pipe`.
- `write` in socket is equivalent to `send` for sockets.
- With the `O_APPEND` flag declared, all `write`s will automatically go to the end of the file.
- *Atomic* operations are operations that cannot be broken apart; they must happen together or they won't happen at all.
- Setting the offset and writing to the file happen in an atomic operation together.
- It's very difficult to ensure consistency across a Network-File-System. We say it may be atomic, but there are scenarios where the operation is not atomic.

2 Signals and System Calls Revisted

- When you block on a slow system call, a signal will interrupt the system call and return the system call with an error.
- Once you catch a signal, it gets reset to the default value. The signal disposition gets reset to its initial value.
- This only occurs on slow system calls.

- When we have a signal that needs to be reset by a signal handler, we have an issue where there in a brief time where the signal can happen without the desired functionality.
- Signal handlers cannot call reentrant functions.
- When a signal handler gets called, you must use reentrant functions.
- Non-reentrant functions are functions where in the middle of the function you cannot stop and have another thread of your program call that function.
- Because signal is asynchronous, execution is stopped when signal interrupts.
- If a function cannot be re-entered in the after an interrupt has occured, then you cannot use it in a signal interrupt.
- From a signal handler, call only *async-signal-safe* functions.

3 File Sharing

- In order to understand interprocess code, you need to understand `pipe`, in order to understand `pipe`, you need to understand file descriptors.
- The kernel maintains a process table entry for every process within the system.
- There is a file descriptor table, and pointers to files.
- The pointers point to file table entries.
- These table entries contain file status flags, current file offset, and v-node pointer.
- You can duplicate a file descriptor from another file descriptor using `dup(1)`.
- `v-node` and `i-node` are essentially the same thing.
- When you `fork`, each file descriptor entry gets duplicated into the child process.
- Child processes share the same offset for a file, so they will not overwrite each other. They can interleave each other, but they will not overwrite their data.
- The process table entries remain the same and are controlled by the kernel, not the user-space.
- Signal handlers are lost when you `exec` as the program is not the same memory space.

4 `pipe`

- Creates two file descriptors, one that reads from the pipe and the other that writes to the pipe.
- By linking the `pipe` to the same process, we can then fork and have a connection between the parent and the child process.
- We can then close the read end of the parent and the write end of the child to create a half-duplex channel from the parent to the child.