Introduction to UNIX System Calls

Why OS?

Computer systems are layered as abstractions: transistors, gates, functional units (adders / multipliers), ISAs, high-level languages, higher-level languages. Transistors allow large designs, because of their composable nature without loss of signal quality. For example, a typical computer has billions of transistors. However, programming these large designs require abstractions. Abstraction design is guided by tradeoffs in efficiency, ease-of-use, security, reliability, etc.

OS is one such abstraction; actually, it provides several abstractions for different parts of the system. Consider a multi-processing system running an editor, a browser, a window manager, and so on. This system can be written as one large event-driven loop. However, that would be extremely complex to manage, as it would need to implement diverse functionalities, yet interoperate cleanly. Instead, an OS provides abstractions, whereby these different functionalities can be implemented as different programs, and run as different *processes*. The OS implements services as *system calls*, which a process can invoke. The OS also ensures safety among these processes. This course is about the design of these abstractions/services that OS provides, their resulting tradeoffs, and the implementation of these abstractions on modern ISAs.

One of the must successful set of abstractions, were the ones used in the UNIX operating system in the 1970s. Unlike its predecessors (e.g., DOS), UNIX was a multi-processing operating system, i.e., multiple processes could co-exist at the same time. Most production operating systems today are based on UNIX abstractions. We will begin by studying the UNIX abstractions.

Outline

- System Calls: fork, exec, exit, wait, open, read, write, close
- Case Study: Unix/xv6 shell (simplified)

System Calls

- "Kernel functions" that perform privileged operations on behalf of the process. As an OS designer, one of the goals is to minimize the system call interface.

Case study: Unix/xv6 shell (simplified)

- provides an interactive command execution and programming language
- typically handles login session, runs other processes
- look at some simple examples of shell operations, how they use different OS abstractions, and how those abstractions fit together. See Unix paper if you are unfamiliar with the shell.
- Basic structure:

```
while (1) {
    write (1, "$ ", 2);   // 1 = STDOUT_FILENO
    readcommand (0, command, args); // parse user input, 0 = STDIN_FILENO
    if ((pid = fork ()) == 0) {  // child?
 exec (command, args, 0);
    } else if (pid > 0) {  // parent?
 wait (0);   // wait for child to terminate
    } else {
 perror ("Failed to fork\n");
    }
}
```

- system calls: `read`, `write`, `fork`, `exec`, `wait`. conventions: -1 return value signals error, error code stored in `errno`, `perror` prints out a descriptive error message based on `errno`.
- What's the shell doing? fork, exec, wait: process diagram (PID, address space -- memory of the process, parent links). fork returns twice, in some sense!
- why call "wait"? to wait for the child to terminate and collect its exit status. (if child finishes, child becomes a zombie until parent calls wait.)
- Example:

```
$ ls
```

- how does ls know which directory to look at? (cwd variable is copied during fork)
- how does it know what to do with its output? (fd[1] is initialized by the shell)
- I/O: process has file descriptors, numbered starting from 0.
- system calls: open, read, write, close
- numbering conventions:
  - file descriptor 0 for input (e.g., keyboard). read_command:

```
read (0, buf, bufsize)
```

    - file descriptor 1 for output (e.g., terminal)

      ```
      write (1, "hello\n", strlen("hello\n"))
      ```

    - file descriptor 2 for error (e.g., terminal)
- on fork, child inherits open file descriptors from parent (show in process diagram).
- on exec, process retains file descriptors, except those specifically marked as close-on-exec: `fcntl(fd, F_SETFD, FD_CLOEXEC)`
- How does the shell implement:

  ```
  $ ls > tmp1
  ```

  just before exec insert:

  ```
  close(1);
  creat("tmp1", 0666);   // fd will be 1
  ```

  The kernel always uses the first free file descriptor, 1 in this case. Could use `dup2()` to clone a file descriptor to a new number.

- Good illustration for why fork + exec vs. CreateProcess on Windows. (CreateProcess takes 10 arguments.)