CS3210: Shell & OS organization

Changwoo Min

Summary of last lecture

- System power on
- Load BIOS
- Find a bootable device
- Execute a boot-loader in the MBR (master boot record)
- Handover the control to the operating system kernel

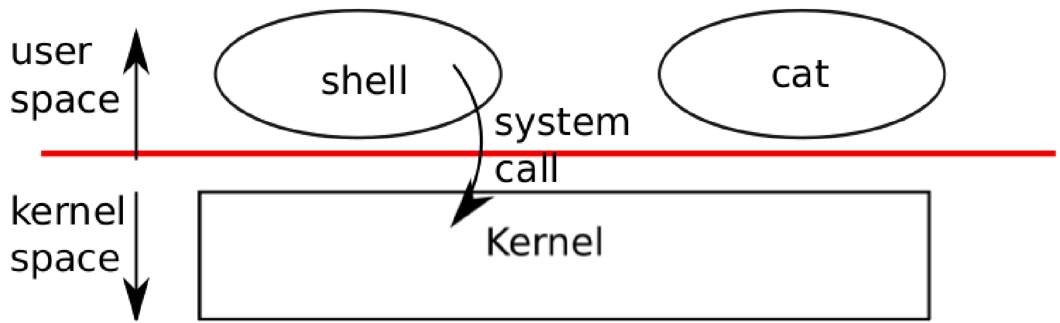
Operating system interfaces

- How programs interact with OS
  - `echo hello`
- How multiple programs interact each other
  - `echo hello | wc --chars`
  - `cat < y | sort | uniq | wc > y1`
- How OS supports such interactions
  - Process
  - System call & files

Kernel space vs. User space

- Kernel
  - a special program that provides services to running programs
- Process
  - has memory containing instructions, data, and a stack
- System call
  - interface between kernel space and user space
  - e.g., `open()`, `close()`, `read()`, `fork()`, …

A kernel and two user processes



- Protection between user and kernel spaces
  - CPU's mechanism: privileged mode vs. unprivileged mode
  - each process in user space can access only its own memory
- `strace`
  - a tool to trace system calls

Example: echo hello

```
$ strace echo hello
```

```
execve("/usr/bin/echo", ["echo", "hello"], [/* 60 vars */]) = 0
...
write(1, "hello\n", 6)                    = 6
...
exit_group(0)                             = ?
```

- system calls: `execve`, `write`, `exit_group`

Example: echo hello > output

```
$ strace -f sh -c "echo hello > output"
```

```
execve("/usr/bin/sh", ["sh", "-c", "echo hello > output"], [/* 60vars*/
...
open("output", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
...
dup2(3, 1)                                       = 1
close(3)                                         = 0
...
write(1, "hello\n", 6)                           = 6
...
exit_group(0)                                    = ?
```

More examples

```
$ echo hello | wc --chars
$ uptime
```

- Pipe between `echo` and `wc`
- Get uptime from `/proc/loadavg`

Shell

- A program that reads commands from the user and executes them
- One of user interface to UNIX-like systems
- A user program, not part of the kernel
    - easily replaceable
    - e.g., bash, zsh, csh, etc.
- Shows power of system call interface

Processes and memory

```
int pid = fork();
if(pid > 0) {          /* parent */
    pid = wait();
} else if(pid == 0){ /* child */
    exec("/bin/echo", "hello");
    exit();            /* never be here */
}
```

- `fork` system call create a new process
  - a child process has the same memory contents with its parent
- `exec` system call loads new memory image from a file
- `wait` system call waits until child `exit` s

File descriptors

- A small integer representing a kernel-managed object
  - file, directory, device, pipe, etc.
- Abstract away the differences between files, pipes, etc.
  - making them all look like byte stream
  - a process may read from or write to file descriptors
- Maintains an offset associated with it
  - `read(fd, buf, n)`
  - `write(fd, buf, n)`

File descriptor table

- Each process has a file descriptor table
  - 0, 1, 2: standard input, output, error
  - 3, ...: `open("output", ...)`
- File descriptor in xv6 and linux kernel
  - an index of the per-process FD table
- System calls which allocate new file descriptor
  - `open()`, `dup()`, `pipe()`, ...
- A newly allocated file descriptor
  - the lowest-numbered unused descriptor in per-process table

Example: cat

- `cat input.txt`, `cat < input.txt`, `ls | cat`

```c
for(;;){
  n = read(0, buf, sizeof(buf));  /* stdin */
  if(n == 0)
    break;
  if(n < 0){
    fprintf(2, "read error\n");  /* stderr */
    exit();
  }
  if(write(1, buf, n) != n){      /* stdout */
    fprintf(2, "write error\n"); /* stder */
    exit();
  }
}
```

Example: a shell for "cat < input.txt"

```
argv[0] = "cat";
argv[1] = 0;
if(fork() == 0) {
  close(0);
  open("input.txt", O_RDONLY); /* what is fd of open? why? */
  exec("cat", argv);
}
```

- `fork` also copies the file descriptor table
  - a parent and its child process shares the file descriptor
- `exec` does not override the file descriptor

Duplicating a file descriptor

```
fd = dup(1);
write(1, "hello ", 6);
write(fd, "world\n", 6);
```

- `dup` system call duplicates an existing file descriptor
  - a returning new FD refers the same file
  - dup2(newfd, oldfd)
- `ls existing-file non-existing-file > tmp1 2>&1`
  - `2>&1`: redirecting stderr to stdout
  - `close(2); dup(1);`

Pipes

- A unidirectional data channel that can be used for interprocess communication
- Exposes a pair a file descriptors
  - `int pipe(int pipefd[2])`
  - pipefd[0] is for reading
  - pipefd[1] is for writing

Example: a shell for "echo hello | wc --char"

```c
int p[2];
char *argv[2];
argv[0] = "wc";
argv[1] = 0;
pipe(p);                /* create a pipe */
if(fork() == 0) {       /* child process */
  close(0);
  dup(p[0]);            /* stdin = p[0] */
  close(p[0]);
  close(p[1]);
  exec("/bin/wc", argv);
} else {                /* parent process */
  write(p[1], "hello\n", 6);
  close(p[0]);
  close(p[1]);
}
```

Code review: xv6 shell (xv6-public/sh.c)

- An ordinary user-space program
  - `main()`: entry function
  - `parsecmd()`: parse command line
  - `rundcmd()`: execute programs
- Can you spot followings?
  - executing a simple command: `echo hello`
  - redirection: `echo hello > output`
  - pipes: `echo hello | wc --char`
- Why `cd` is implemented at the shell?

Summary & Questions

- Now we have a feel for what Unix system call interface provides
- How to implement the interface?
- Why have an OS at all? why not just a library?
    - then apps are free to use it, or not -- flexible apps can directly interact with hardware
    - some tiny OSes for embedded processors work this way

Operating system organization

- Goal: process isolation & sharing
    - a process should *not* corrupt the memory of the kernel or another process
    - *nor* consume all the CPU time/memory
    - *nor* run arbitrary privileged instructions, etc.
- Applications must use OS interface, cannot directly interact with hardware so that apps cannot harm operating system

Key design factors

- What to put below/above the system call interface
- How to isolate user space and kernel space
  - for applications not to harm kernel space

Hardware support for isolation

- Processors support user mode and kernel mode
  - some instructions can only be executed in kernel mode
  - e.g., change the address translation map, talk to I/O devices
- If an application executes a privileged instruction, hardware doesn't allow it
  - instead switches to kernel mode then kernel can clean up

Hardware isolation in x86

- x86 support: kernel/user mode flag
- CPL (current privilege level): lower 2 bits of `%cs`
  - 0: kernel, privileged
  - 3: user, unprivileged
- system calls: controlled transfer
  - `int` or `sysenter` instruction set CPL to 0
  - set CPL to 3 before going back to user space

Monolithic kernel: Linux, xv6, etc.

- A traditional design: all of the OS runs in kernel mode
- Kernel interface ~= system call interface
- Good: easy for subsystems to cooperate
  - one cache shared by file system and virtual memory
- Bad: interactions are complex
  - leads to bugs, no isolation within kernel

Alternative: microkernel design

- Many OS services run as ordinary user programs
  - e.g., file system in a file server
- Kernel implements minimal mechanism to run services in user space
  - IPC, virtual memory, threads
- Kernel interface != system call interface
  - applications talk to servers via IPCs
- Good: more isolation
- Bad: IPCs may be slow

Debate

- Tanenbaum-Torvalds debate
- Most real-world kernels are mixed: Linux, OS X, Windows
    - e.g., X Window System