CS3210: Shells and Interfaces

Lecture 3

Instructor: Dr. Tim Andersen

Shells

- OS needs a default mode of interaction with the user
- The shell allowed complex programs to be built up from small programs
- The first scripting language (required no compilation)
- Pipes and redirects allowed standalone programs to communicate

Operating System Interfaces

- Need the interface to be simple but allow for complexity
- Trick is to create a general system based on a few mechanisms
- xv6 mimics Ken Thompson and Dennis Richie's Unix interface and internals
- Provides a narrow interface with a surprising degree of generality
- BSD, Linux, Mac OS X, Solaris, and even Windows rely on a Unix-like interface

Operating system interfaces

- Multiple programs interact each other with pipes and redirects
    - `echo hello | wc --chars`
    - `cat < y | sort | uniq | wc > y1`
- The OS supports interactions with
    - Processes and scheduling
    - System call & files and file descriptors
- Everything the OS does, all its requirements, begin with the shell
- Even devices have shell interfaces with `/dev`, e.g., `ls > /dev/tcp/127.0.0.1/8000` dumps the directory contents to localhost port 8000

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()`, …

Open

```
NAME
       open, creat - open and possibly create a file or device

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);

DESCRIPTION
    Given  a pathname for a file, open() returns a file descriptor, a small,
    nonnegative integer for use in subsequent system calls (read(2), write(2),
    lseek(2), fcntl(2), etc.). The file descriptor returned by a successful call
    will be the lowest-numbered file descriptor not currently open for the process.
```

Dup

```
NAME
       dup, dup2, dup3 - duplicate a file descriptor

SYNOPSIS
       #include <unistd.h>

       int dup(int oldfd);
       int dup2(int oldfd, int newfd);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions */
       #include <unistd.h>

       int dup3(int oldfd, int newfd, int flags);

DESCRIPTION
       These system calls create a copy of the file descriptor oldfd.

       dup() uses the lowest-numbered unused descriptor for the new descriptor.

       After a successful return from one of these system calls, the old and new file
       descriptors may be used interchangeably. They  refer to the same open file
       description (see open(2)) and thus share file offset and file status flags; for
       example, if the file offset is modified by using lseek(2) on one of the
       descriptors, the offset is  also  changed  for  the other.
```

Fork

```
NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a  new  process  by duplicating the calling process.
       The new process, referred to as the child, is an exact duplicate
       of the calling process, referred to as the parent, except for the following points:

       *  The child has its own unique process ID.
       *  The child's parent process ID is the same as the parent's process ID.

RETURN VALUE

       On success, the PID of the child process is returned in the parent, and 0 is
       returned in the child.  On failure, -1  is returned in the parent, no child
       process is created, and errno is set appropriately.
```

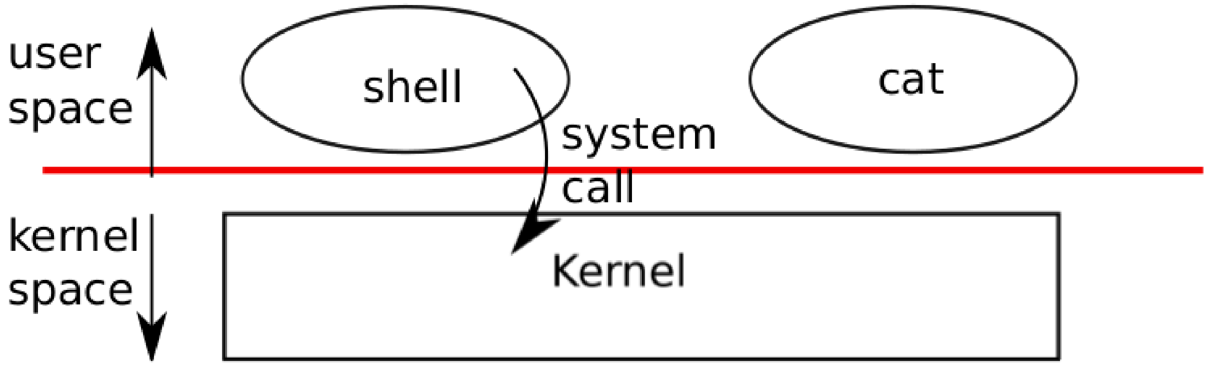
Pipe

```
NAME
       pipe, pipe2 - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);

DESCRIPTION
       pipe() creates a pipe, a unidirectional data channel that can be used for
       interprocess communication.  The array pipefd is used to return two file
       descriptors referring to the ends of the pipe.  pipefd[0] refers to
       the  read  end  of  the pipe.   pipefd[1]  refers  to  the write end

       of the pipe.  Data written to the write end of the pipe is buffered by the
       kernel until it is read from the read end of the pipe.
```

Close

```
NAME
       close - close a file descriptor

SYNOPSIS
       #include <unistd.h>

       int close(int fd);

DESCRIPTION
       close()  closes  a  file  descriptor, so that it no longer refers to any file and
       may be reused.  Any record locks (see fcntl(2)) held on the file it was
       associated with, and owned by the  process,  are  removed (regardless
       of the file descriptor that was used to obtain the lock).

       If  fd is the last file descriptor referring to the underlying open file
       description (see open(2)), the resources associated with the open file
       description are freed; if the descriptor was the last reference to a file
       which has been removed using unlink(2) the file is deleted.


RETURN VALUE
       close() returns zero on success.  On error, -1 is returned, and errno
       is set appropriately.
```

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*/]) = 0
...
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
- 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
- Essentially a user program designed to allow users to interact with the kernel via system calls

Processes and memory

```
int pid = fork();
if(pid > 0) {          /* parent */
    pid = wait();
} else if(pid == 0){ /* child */
    execl("/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 but does not share memory (unless there is a shared memory space allocated)
- `execl` system call loads new memory image from a file
- `wait` system call waits until child `exits`

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)`
- Each process starts with 3 file descriptors: 0, 1, and 2.
    - 0 = standard input (stdin)
    - 1 = standard output (stdout)
    - 2 = standard error (stderr)
- These can be closed and reassociated

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
- Fork causes a child to inherit the parent descriptor table

Example: cat

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

```
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"

```c
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

```c
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]);

}
```

DEMO

```
$ echo "hello there" | sed "s/hello/hi/" >& hi.txt
```

```c
int fd[2], filefd;
pipe(fd);
if (fork() == 0) // child process
{
    close(fd[1]);
    close(0);
    dup(fd[0]);
    filefd = open("hi.txt", O_CREAT | O_WRONLY | O_TRUNC);
    close(1);
    dup(filefd);
    close(2);
    dup(filefd);
    close(fd[0]);
    close(filefd);
    execlp("sed", "sed", "s/hello/hi/", NULL);
}
else // parent process

{
    close(fd[0]);
    close(1);
    dup(fd[1]);
    close(fd[1]);
    execlp("echo", "echo", "hello there", NULL);
}
```

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