CS238P: Principles of Operating Systems

Lecture 2: OS Interfaces

Anton Burtsev
January, 2018
Recap from last time: role of the operating system

- Share hardware across multiple processes
  - Illusion of private CPU, private memory
- Abstract hardware
  - Hide details of specific hardware devices
- Provide services
  - Serve as a library for applications
- Security
  - Isolation of processes, users, namespaces
  - Controlled ways to communicate (in a secure manner)
Typical UNIX OS

User

Application (Shell)
System Libraries

Kernel

System Call Interface

Scheduler
File System

Network Stack

Virtual Memory

Application (Apache Web Server)
System Libraries
System calls

- Provide user to kernel communication
  - Effectively an invocation of a kernel function

- System calls are the interface of the OS
System calls, interface for...

- Processes
  - Creating, exiting, waiting, terminating
- Memory
  - Allocation, deallocation
- Files and folders
  - Opening, reading, writing, closing
- Inter-process communication
  - Pipes
UNIX (xv6) system calls are designed around the shell
Why shell?
Ken Thompson (sitting) and Dennis Ritchie working together at a PDP-11
DEC VT100 terminal, 1980
Suddenly this makes sense

• List all files

\> ls

```
total 9212
```

```
-rw-rw-r--  1 aburtsev aburtsev   936 Oct  1 08:26 asm.h
-rw-rw-r--  1 aburtsev aburtsev  3397 Oct  1 08:26 bio.c
-rw-rw-r--  1 aburtsev aburtsev  100 Oct  1 08:26 bio.d
-rw-rw-r--  1 aburtsev aburtsev  6416 Oct  1 08:26 bio.o
```

• Count number of lines in a file (ls.c implements ls)

\> wc -l ls.c

```
85  ls.c
```
But what is shell?
But what is shell?

• Normal process
  • Kernel starts it for each user that logs in into the system
  • In xv6 shell is created after the kernel boots
• Shell interacts with the kernel through system calls
  • E.g., starts other processes
But what happens underneath?

\> wc -l ls.c

85 ls.c

\>

- Shell invokes `wc`
  - Creates a new process to run `wc`
  - Passes the arguments (`-l` and `ls.c`)
  - `wc` sends its output to the terminal (console)
  - `wc` exits when done with `exit()`
- Shell detects that `wc` is done
  - Prints (to the same terminal) its command prompt
  - Ready to execute the next command
How do we create a process?
fork()

Shell

pid = fork()

Kernel
System call

Process (e.g., Apache, shell)

User stack of a process (can grow up to 2GBs)

Code, data, heap

Interrupt Vector #80

int 0x80

Kernel code

vector80

IDT

CS : HANDLER ADDR

...
fork()

Shell (parent)

32 = fork()

Shell (child)

0 = fork()

Kernel
fork() -- create new process

1. int pid;
2. pid = fork();
3. if(pid > 0){
4.   printf("parent: child=%d\n", pid);
5.   pid = wait();
6.   printf("child %d is done\n", pid);
7. } else if(pid == 0){
8.   printf("child: exiting\n");
9.   exit();
10. } else {
11.   printf("fork error\n");
12. }
This is weird... fork() creates copies of the same process, why?
Pipes

• What if we want to see how many strings in ls.c contain “main”
Pipes

• Imagine we have a way to redirect output of one process into input of another
  
  \> cat ls.c | grep main

• | (or a “pipe”) does redirection
Pipes

• In our example:

\>` cat ls.c | grep main

• cat outputs ls.c to its output
  • cat's output is connected to grep's input with the pipe
  • grep filters lines that match a specific criteria, i.e., once that have “main”
Composability

- Now if we want to see how many strings in ls.c contain “main” we do:

  ```shell
  \> cat ls.c | grep main | wc -l
  1
  ```

- But if we want to count the ones that contain “a”:

  ```shell
cat ls.c | grep a | wc -l
33
  ```

- We change only input to grep!
  - Small set of tools (ls, grep, wc) compose into more complex programs
Better than this...
How to assemble this pipeline?

\[ \text{cat ls.c | grep main | wc -l} \]

1

- `wc` has to operate on the output of `grep`
- `grep` operates on the output of `cat`
Let's look at file I/O

- `read(fd, buf, n)` – read n bytes from fd into buf
- `write(fd, buf, n)` – write n bytes from buf into fd
File descriptors

Process (e.g., "cat ls.c")

read(3, buf, size);

Process' File Descriptor Table

Kernel

File (ls.c)
File descriptors

- An index into a table, i.e., just an integer
- The table maintains pointers to “file” objects
  - Abstracts files, devices, pipes
  - In UNIX everything is a pipe – all objects provide file interface
- Process may obtain file descriptors through
  - Opening a file, directory, device
  - By creating a pipe
  - Duplicating an existing descriptor
File descriptors: two processes

Process (e.g., "cat ls.c")

```c
read(3, buf, size);
```

Process (e.g., "wc -l wc.c")

```c
read(5, buf, size);
```
• File descriptors don't have to point to files *only*
  • Any object with the same read/write interface is ok
  • Network channel
  • Pipe
pipe - interprocess communication

- Pipe is a kernel buffer exposed as a pair of file descriptors
  - One for reading, one for writing
- Pipes allow processes to communicate
  - Send messages to each other
Two file descriptors pointing to a pipe

Process (e.g., "cat ls.c")

write(3, buf, size);

Process (e.g., "grep main")

read(5, buf, size);
Pipes allow us to connect programs, i.e., output of one program to input of another
Each process has standard file descriptors

- Numbers are just a convention
  - 0 – standard input
  - 1 – standard output
  - 2 – standard error
- This convention is used by the shell to implement I/O redirection and pipes
Example: cat

1. char buf[512]; int n;
2. for(;;) {
3.     n = read(0, buf, sizeof buf);
4.     if(n == 0)
5.         break;
6.     if(n < 0) {
7.         fprintf(2, "read error\n");
8.         exit(); }
9.     if(write(1, buf, n) != n) {
10.        fprintf(2, "write error\n");
11.        exit();
12.    }
13. }

Now we can redirect standard input and output
fork() is used together with exec()

- exec() -- replace memory of a current process with a memory image (of a program) loaded from a file

```c
char *argv[3];
argv[0] = "echo";
argv[1] = "hello";
argv[2] = 0;
exec("/bin/echo", argv);
printf("exec error\n");
```
Two system calls for I/O redirection

- `close(fd)` – closes file descriptor
  - The next opened file descriptor will have the lowest number
- `exec()` preplaces process memory, but
  - leaves its file table (table of the file descriptors untouched)
File descriptors after fork()

Shell (parent)

32 = fork()
read(3, buf, size);

Shell (child)

0 = fork()
read(3, buf, size);
File descriptors after `exec()`

Shell (parent)

```c
32 = fork()
read(3, buf, size);
```

wc -l

```c
exec("/bin/wc", argv)
read(3, buf, size);
```

Parent's File Descriptor Table

```
0 3
```

Child's File Descriptor Table

```
0 3
```
File I/O redirection

- `close(fd)` – closes file descriptor
  - The next opened file descriptor will have the lowest number
- `exec()` replaces process memory, but
  - leaves its file table (table of the file descriptors untouched)
  - Shell can create a copy of itself with `fork()`
  - Change the file descriptors for the next program it is about to run
  - And then execute the program with `exec()`
Example: `\> cat < input.txt`

1. `char *argv[2];`
2. `argv[0] = "cat";`
3. `argv[1] = 0;`
4. `if(fork() == 0) {
   5.     close(0);
   6.     open("input.txt", O_RDONLY);
   7.     exec("cat", argv);
   8. }
`
Why `fork()` not just `exec()`

- The reason for the pair of `fork() / exec()`
  - Shell can manipulate the new process (the copy created by `fork()`)
  - Before running it with `exec()`
Back to pipes

• It's possible to use a pipe to connect two programs
  • Create a pipe
  • Attach one end to standard output
    - of the left side of “|”
  • Another to the standard input
    - of the right side of “|”
int p[2];
char *argv[2]; argv[0] = "wc"; argv[1] = 0;
pipe(p);
if(fork() == 0) {
    close(0);
dup(p[0]);
    close(p[0]);
    close(p[1]);
    exec("/bin/wc", argv);
} else {
    write(p[1], "hello world\n", 12);
close(p[0]);
close(p[1]);
}

wc on the read end of the pipe
More process management

- `exit()` -- terminate current processes
- `wait()` -- wait for the child to exit
Powerful conclusion

- `fork()`, standard file descriptors, `pipes` and `exec()` allow complex programs out of simple tools.
- They form the core of UNIX interface.
Of course there is more
You need to deal with files

- Files
  - Uninterpreted arrays of bytes
- Directories
  - Named references to other files and directories
Creating files

- `mkdir()` – creates a directory
- `open(O_CREAT)` – creates a file
- `mknod()` – creates an empty files marked as device
  - Major and minor numbers uniquely identify the device in the kernel
- `fstat()` – retrieve information about a file

- Named references to other files and directories
Fstat

• `fstat()` – retrieve information about a file

```c
#define T_DIR 1 // Directory
#define T_FILE 2 // File
#define T_DEV 3 // Device

struct stat {
    short type; // Type of file
    int dev; // File system’s disk device
    uint ino; // Inode number
    short nlink; // Number of links to file
    uint size; // Size of file in bytes
};
```
Links, inodes

- Same file can have multiple names – links
  - But unique inode number
- link() – create a link
- unlink() – delete file
- Example, create a temporary file

```c
fd = open("/tmp/xyz", O_CREAT|O_RDWR);
unlink("/tmp/xyz");
```
fork() Create a process
exit() Terminate the current process
wait() Wait for a child process to exit
kill(pid) Terminate process pid
getpid() Return the current process’s pid
sleep(n) Sleep for n clock ticks
exec(filename, *argv) Load a file and execute it
sbrk(n) Grow process’s memory by n bytes
open(filename, flags) Open a file; the flags indicate read/write
read(fd, buf, n) Read n bytes from an open file into buf
write(fd, buf, n) Write n bytes to an open file
close(fd) Release open file fd
dup(fd) Duplicate fd
pipe(p) Create a pipe and return fd’s in p
chdir(dirname) Change the current directory
mkdir(dirname) Create a new directory
mknod(name, major, minor) Create a device file
fstat(fd) Return info about an open file
link(f1, f2) Create another name (f2) for the file f1
unlink(filename) Remove a file
In many ways xv6 is an OS you run today
Speakers from the 1984 Summer Usenix Conference (Salt Lake City, UT)
Backup slides
Pipes

- Shell composes simple utilities into more complex actions with pipes, e.g.
  
grep FORK sh.c | wc -l

- Create a pipe and connect ends
System call

Process (e.g., Apache, shell)

User stack of a process (can grow up to 2GBs)

int 0x80

Interrupt Vector #80

Code, data, heap

Last stack frame

EBP

Kernel code

vector80

IDT

... CS : HANDLER ADDR

...
User address space

Process (e.g., Apache, shell)

User stack of a process (can grow up to 2GBs)

int 0x80

Interrupt Vector #80

Kernel code

Kernel Address Space

EBP

Last stack frame

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc
Kernel and user address spaces

Process (e.g., Apache, shell)

User stack of a process (can grow up to 2GBs)

Code, data, heap

Interrupt Vector #80

int 0x80

EBP

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc

Last stack frame

Kernel Address Space

Kernel code

vector80

IDT

CS : HANDLER ADDR

...