143A: Principles of Operating Systems

Lecture 2: OS Interfaces

Anton Burtsev
September, 2017
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, namesapces
  - Controlled ways to communicate (in a secure manner)
Typical UNIX OS

User

Kernel

Application
(Shell)

System Libraries

System Call Interface

Scheduler

Network Stack

Virtual Memory

File System

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

- drwxrwxr-x  3 aburtsev aburtsev 12288 Oct  1 08:27 ./
- drwxrwxr-x 43 aburtsev aburtsev 4096 Oct  1 08:25 ../
- -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
Shell

- Normal process
- Interacts with the kernel through system calls
  - Creates new 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)
  - 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
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. }
fork()

Shell

pid = fork()

Kernel
fork()

Shell (parent)
32 = fork()

Shell (child)
θ = fork()

Kernel
This is weird... fork() creates copies of the same process, why?

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

  \(\texttt{\color{red}{\textbackslash > \text{cat ls.c | grep main | wc -l}}}\)

  1

- .. or contain “a”

  \(\texttt{\color{red}{\text{cat ls.c | grep a | wc -l}}}\)

  33

- Composability is great
  
  - Small set of tools (ls, grep, wc) compose into more complex programs
Better than this...
How to assemble this pipeline?

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

1

- `wc` has to operate on the output of `grep`
- `grep` operates on the output of `cat`
Lets 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., Apache, shell)

\texttt{read(3, buf, size);}

---

Kernel

File

Process' File Descriptor Table

0 \rightarrow 3
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

read(3, buf, size);

read(5, buf, size);
File descriptors don't have to point only to files

- 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

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

```
read(5, buf, size);
```
Now we're ready to build the pipelines
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. }

File I/O redirection

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

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

```
0 = fork()
read(3, buf, size);
```
File I/O redirection

- `close(fd)` – closes file descriptor
  - The next opened file descriptor will have the lowest number
- `fork` 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()`
• `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");
```
Example: `\> cat < input.txt`

1. `char *argv[2];`
2. `argv[0] = "cat";`
3. `argv[1] = 0;`
4. `if(fork() == 0) {
   close(0);
   open("input.txt", O_RDONLY);
   exec("cat", argv);
}
   `
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]);
}
More process management

- `exit()` -- terminate current process
- `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_CREATE)` – 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
  ```
  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

Xv6 system calls
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)

Code, data, heap

int 0x80

Interrupt Vector #80

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)

Code, data, heap

Interrupt Vector #80

int 0x80

EBP

Last stack frame

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

Kernel code

Kernel Address Space

vector80

IDT

CS : HANDLER ADDR

...
Kernel address space

Process (e.g., Apache, shell)

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

Code, data, heap

int 0x80

Interrupt Vector #80

Kernel code

vector80

Process 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

Kernel Address Space
- Kernel code
- vector80

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

Interrupt Vector #80

EBP

IDT
- CS : HANDLER_ADDR
- ...