Lecture 2: OS Interfaces

Anton Burtsev
September, 2021
Recap: 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
  - Controlled ways to communicate (in a secure manner)
Typical UNIX OS

User

Application (Shell)
System Libraries

Kernel

System Call Interface
Scheduler
Network Stack
File System
Virtual Memory

Application (Apache Web Server)
System Libraries
System calls

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

- System calls implement the interface of the OS
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

...
What system calls do we need?
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 (standing) are working together on a PDP-11 (around 1970). They are using Teletype Model 33 terminals.
DEC LA36 DECwriter II Terminal
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
But what is shell?
But what is shell?

- Normal process
  - Kernel starts it for each user that logs 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
What happens underneath?

\> wc -l ls.c

85  ls.c
\>

• Shell starts *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 (*wait()*)
  • Prints (to the same terminal) its command prompt
  • Ready to execute the next command
Console and file I/O
File open

- \texttt{fd = open("ls.c", O_RDONLY)} – open a file
  - Operating system returns a file descriptor
File descriptors

```
fd = open("ls.c", ...);
```

Process (e.g., "cat 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 file – 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 I/O

- `fd = open("foobar.txt", O_RDONLY)` – open a file
  - Operating system returns a file descriptor

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

- `write(fd, buf, n)` – write `n` bytes from `buf` into `fd`
File descriptors: two processes

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

read(3, buf, size);

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

read(4, buf, size);
Console I/O
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
Console read (read of standard input)

```
read(0, buf, size);
```
Console write (write of standard output)

```c
read(0, buf, size);
write(1, buf, size);
```
Example: cat

1. char buf[512];
2. int n;
3. for(;;) {
   4.     n = read(0, buf, sizeof buf);
   5.     if(n == 0)
   6.         break;
   7.     if(n < 0) {
        8.         fprintf(2, "read error\n");
        9.         exit(); }
   10.   if(write(1, buf, n) != n) {
          11.     fprintf(2, "write error\n");
          12.     exit();
   13.         }
   14. }

Creating processes
fork()

Shell

pid = fork()

Kernel
fork()

Shell (parent)

32 = fork()

Shell (child)

θ = fork()

Kernel
fork() -- creates a 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);
} else if(pid == 0){
   8. printf("child: exiting\n");
   9. exit();
} else {
   11. printf("fork error\n");
}
This is weird... `fork()` creates copies of the same process, why?
fork() is used together with exec()

- exec() -- replaces 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");
```
fork() and exec()

Parent (Shell)

32 = fork()

Child (Shell)

0 = fork();
exec("/bin/wc", argv);
fork() and exec()
• Still weird... why first `fork()` and then `exec()`?
• Why not `exec()` directly?
I/O Redirection
Motivating example #1

• Normally `wc` sends its output to the console (screen)
  • Count the number of lines in `ls.c`

\>` wc -l ls.c

85  `ls.c`

• What if we want to save the number of lines into a file?
Motivating example #1

• Normally `wc` sends its output to the console (screen)
  • Count the number of lines in `ls.c`

```bash
\> wc -l ls.c
85 ls.c
```

• What if we want to save the number of lines into a file?
  • We can add an argument

```bash
\> wc -l ls.c -o foobar.txt
```
Motivating example #1

\> wc -l ls.c -o foobar.txt

• But there is a better way

\> wc -l ls.c > foobar.txt
I/O redirection

• > redirect output
  • Redirect output of a command into a file
  \> wc -l ls.c > foobar.txt
  \> cat ls.c > ls-new.c

• < redirect input
  • Redirect input to read from a file
  \> wc -l < ls.c
  \> cat < ls.c

• Redirect both
  \> wc -l < ls.c > foobar.txt
Standard output is now a file

```c
read(0, buf, size);
write(1, buf, size);
```
Powerful design choice

- File descriptors don't have to point to files only
  - Any object with the same read/write interface is ok
  - Files
  - Devices
    - Console
  - 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. }
```
Why do we need I/O redirection?
Motivating example #2

- We want to see how many strings in ls.c contain “main”
Motivating example #2

- We want to see how many strings in ls.c contain “main”
  - Imagine we have `grep`
    - `grep` filters strings matching a pattern
  
  `grep "main" ls.c`

  ```
  main(int argc, char *argv[])
  ```

- Or the same written differently
  
  `grep "main" < ls.c`

  ```
  main(int argc, char *argv[])
  ```
Motivating example #2

- Now we have
  - `grep`
    - Filters strings matching a pattern
  - `wc -l`
    - Counts lines

- Can we combine them?
Pipes

• Imagine we have a way to redirect output of one process into input of another

\> cat ls.c | grep main

• | (a “pipe”) does redirection
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”
Pipe - inter-process 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(4, buf, size);

Green Process File Descriptor Table
Kernel
Pipe
Pipes allow us to connect programs, i.e., the output of one program to the input of another.
Composability

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

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


  1

• .. but if we want to count the once that contain “a”:

  cat ls.c | grep a | wc -l


  33

• We change only input to grep!
  • Small set of tools (ls, grep, wc) compose into complex workflows
Better than this...
Building I/O redirection
How can we build this?

\$ cat ls.c | grep main | wc -l

- `wc` has to operate on the output of `grep`
- `grep` operates on the output of `cat`
Back to \texttt{fork()}

Shell

\texttt{pid = fork()}

Kernel
fork()

Shell (parent)
32 = fork()

Shell (child)
0 = fork()

Kernel
File descriptors after `fork()`

Shell (parent)

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

Shell (child)

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

Parent's File Descriptor Table

Child's File Descriptor Table

Kernel

File
Two system calls for I/O redirection

- `close(fd)` – closes file descriptor
  - The next opened file descriptor will have the lowest number
File descriptors after `close()`/`open()`

Example: `> cat < ls.c`

```
32 = fork()
read(0, buf, size);
```

```
0 = fork()
close(0)
0 = open("ls.c");
```
Two system calls for 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)
  - A process 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()`
File descriptors after `exec()`

Example: `> cat < ls.c`

Shell (parent):

32 = fork()
read(0, buf, size);

Shell (child):

0 = fork()
close(0)
0 = open("ls.c");
Example: `\> cat < ls.c`

1. `char *argv[2];`
2. `argv[0] = "cat";`
3. `argv[1] = 0;`
4. `if(fork() == 0) {
   close(0);
   open("ls.c", 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 Motivating example #2
(\> cat ls.c | grep main | wc -l)
Pipes

• We now understand how to use a pipe to connect two programs
  • Create a pipe
  • Fork
  • Attach one end to standard output
    - of the left side of “|”
  • Another to the standard input
    - of the right side of “|”
1. int p[2];
2. char *argv[2]; argv[0] = "wc"; argv[1] = 0;
3. pipe(p);
4. if(fork() == 0) {
  5.    close(0);
  6.    dup(p[0]);
  7.    close(p[0]);
  8.    close(p[1]);
  9.    exec("/bin/wc", argv);
10. } else {
11.    write(p[1], "hello world\n", 12);
12.    close(p[0]);
13.    close(p[1]);
14. }
Parent

\texttt{write(p[1], \textquotedblleft hello\ world\n\textbackslash n\textquotedblright, 12);} \\

\texttt{wc -l}

\texttt{exec("/bin/wc", argv)}

\texttt{read(0, buf, size);}
cat ls.c | grep main | wc -l
Powerful conclusion

- `fork()`, standard file descriptors, `pipes` and `exec()` allow complex programs out of simple tools
- They form the core of the UNIX interface
More system calls
Process management

- `exit()` -- terminate current process
- `wait()` -- wait for the child to exit
Creating files

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

BSD family
- BSD (Berkeley Software Distribution) 4.4
  - SunOS 4.1.4
  - NextStep 3.3
- GNU
  - Richard Stallman
- GNU/Hurd
- Linux
  - Linus Torvalds
- Minix
  - Andrew S. Tanenbaum
- Research UNIX
  - Bell Labs: Ken Thompson, Dennis Ritchie, et al.
- Commercial UNIX
  - AT&T
  - Solaris
    - Sun/Oracle
  - UnixWare
    - Univel/SCO
- System III & V family
  - HP-UX
  - AIX
    - IBM
  - IRIX
    - SGI

FreeBSD
- 11.0
- Matthew Dillon
- NetBSD
- 7.1
- OpenBSD
- 6.1
- Darwin
- 16.4
- macOS
- Apple
- 10.12
- FreeBSD
- 4.8

Evolution of Unix and Unix-like systems
Backup slides
Pipes

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

- Create a pipe and connect ends
System call
User address space
Kernel address space

Process (e.g., Apache, shell)

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

Code, data, heap

int 0×80

Interrupt Vector #80

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.

Last stack frame

EBP →
Kernel and user address spaces