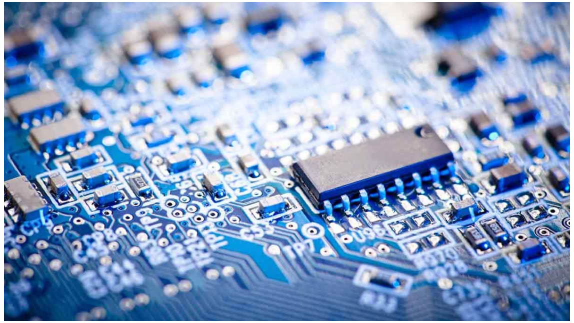
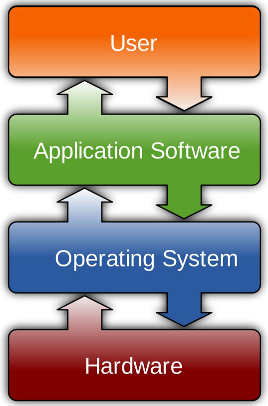


CS 3650 Computer Systems

Ferdinand Vesely / Alden Jackson



Intro	Virtualization		Concurrency	Persistence	Appendices
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	
		21 Swapping: Mechanisms	34 *Summary*	44 Flash-based SSDs	
		22 Swapping: Policies		45 Data Integrity and Protection	
		23 Case Study: VAX/VMS		46 *Summary*	
		24 *Summary*		47 *Dialogue*	
				48 Distributed Systems	
				49 Network File System (NFS)	
				50 Andrew File System (AFS)	
				51 *Summary*	

POSIX File I/O

Everything is a file, until it isn;t.

POSIX File System Basics

We've been introduced to two types of virtualization:

- The **process**, which virtualizes the CPU
- The **address space**, which virtualizes memory (more details on this later)
- Together, they allow a program to run as if it had its own private processor and its own memory

Persistent storage, i.e., disk drives, which keep data intact when power is lost, is one more element in the virtualization model

Two major abstractions: files and directories

Files and Directories

File

- Linear array of bytes that can be written or read
- Name
    - Low-level: inode, an non-zero integer, used by the OS
    - User-readable

Directory

- File containing list of (low-level name, user-readable name) pairs
- Can contain other directories, as a directory is a file
- Root directory: **/** Current directory: **.** Parent directory: **..**

open / close

Opening an existing or creating a new file, is done with the `open()` system call

2	open	sys_open	fs/open.c

%rdi	%rsi	%rdx
const char __user * filename	**int** flags	**umode_t** mode

```
// Create file "foo" and return a file descriptor
int fd = open("foo",
 O_CREAT|O_WRONLY|O_TRUNC, // create write-only
 S_IRUSR|S_IWUSR);         // set permissions
```

File descriptor, `fd`: an integer, private per process, used by OS to access files

Use `fd` to read or write the file.

open / close

To close the file:

```c
// Close an open file descriptor
close(fd);  // returns 0 on success
```

3	close	sys_close	fs/open.c

%rdi

unsigned int fd

Example: using `strace`

```
$ echo "hello cs3650" > foo
$ strace cat foo
…
openat(AT_FDCWD, "foo", O_RDONLY)       = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=13,  ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 1056768, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8f66844000
read(3, "hello cs3650\n", 1048576)      = 13
write(1, "hello cs3650\n", 13hello cs3650
)         = 13
read(3, "", 1048576)                    = 0
munmap(0x7f8f66844000, 1056768)         = 0
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
…
$
```

stdin = 0, stdout = 1, stderr = 2

openat() returns file descriptor = 3
fstat() returns status information on 3, in particular length of file (13 bytes)

read(13 bytes from 3)
write(13 bytes to 1)

read(0 bytes from 3)

close() all open fds

read / write

```
ssize_t read(int fd, void *buf, size_t count);
```

read() attempts to read up to **`count`** bytes from file descriptor **`fd`** into the buffer starting at **`buf`**.

```
    read(3, "hello cs3650\n", 1048576) = 13
```

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.

0	read	sys_read	fs/read_write.c

%rdi	%rsi	%rdx
unsigned int fd	**char __user *** buf	**size_t** count

read / write

```
ssize_t write(int fd, const void *buf, size_t count);
```

`write()` writes up to **`count`** bytes from the buffer starting at **`buf`** to the file referred to by the file descriptor **`fd`**.

```
write(1, "hello cs3650\n", 13hello cs3650) = 13
```

On success, the number of bytes written is returned. On error, -1 is returned and errno is set to indicate the cause of the error.

1	write	sys_write	fs/read_write.c

%rdi	%rsi	%rdx
unsigned int fd	**const char __user *** buf	**size_t** count

Redirecting I/O

All running programs have 3 default I/O strear

- Standard Input: `stdin` (0)
- Standard Output: `stdout` (1)
- Standard Error: `stderr` (2)

By default,

- `stdin` is the keyboard
- `stdout` and `stderr` are the terminal

But these can be redirected…

```
#  redirect a.out's stdin to read from file
infile.txt:
$ ./a.out < infile.txt

#  redirect a.out's stdout to print to file
outfile.txt:
$ ./a.out > outfile.txt

# redirect a.out's stdout and stderr to a file
out.txt
$ ./a.out &> outfile.txt

# redirect all three to different files:
#   (< redirects stdin, 1> stdout, and 2> stderr):
 $ ./a.out < infile.txt 1> outfile.txt 2>
 errorfile.txt
```

https://diveintosystems.org/singlepage/#_io_in_c

Pipes

At its simplest, a pipe is a unidirectional data channel

- Typical use is to connect the ‘output’ of a process to the ‘input’ of another process
- In the shell (see right) or in a program

```
# find the number of processes
# option 1
$ ps axu > output.txt
$ wc -l output.txt
  120  output.txt
# option 2 using a pipe ‘|’
$ ps axu | wc -l
    121
# why are the numbers different?
```

Creating pipes in C

```c
int pipe(int pipefd[2]);
```

Creates a unidirectional data channel.

`int pipefd[2]`: contains the newly created file descriptors created.

- `pipefd[0]` is the 'read' end
- `pipefd[1]` is the 'write' end

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.

Illustrated example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char **argv) {
    // Lets confirm the values of the default file
descriptors of our
    // input, output, and error.
    printf("STDIN_FILENO = %d\n", STDIN_FILENO);
    printf("STDOUT_FILENO = %d\n", STDOUT_FILENO);
    printf("STDERR_FILENO = %d\n\n",
STDERR_FILENO);

    // First, lets have some storage for file
descriptors for which
    // our pipes 'read' end and 'write' end will
be.
    // Thus, we need an array of two integers to
hold our file descriptors.
    int fd[2];

    // fd[0] is the 'read' end
    // fd[1] is the 'write' end.
    pipe(fd);

    //  two new file descriptors were created
using the next available
    //  integers, giving handles to the read and
write end of the
    //  pipe.

    printf("pipe fd[0] (for reading) = %d\n",
fd[0]);
    printf("pipe fd[1] (for writing) = %d\n\n",
fd[1]);

    // Let's store the child process id and status
    pid_t childProcessID;
    int child_status;
```

Illustrated example, continued

```
    // Execute our fork() and duplicate our
parent.
    childProcessID = fork();
    // Check that a child was successfully
created.
    if(-1 == childProcessID) {
        printf("fork failed for some reason!");
        exit(EXIT_FAILURE);
    }

    // Now we want to execute the child code
first.
    // Whatever happens in the child, we will
output that into our
    // pipe and then our parent will print out the
resulting output.
```

```
    if (childProcessID == 0) {
        // Now remember, our child inherits
(almost) everything from
        // the parent.  This includes the file
descriptors.  lets
        // print the child file descriptors just
to see.
        printf("child copy of pipe fd[0] = %d\n",
fd[0]);
        printf("child copy of pipe fd[1] = %d\n\n",
fd[1]);
        // Let's do something with our child
process
        char* myargv[3];
            myargv[0] = "echo";
            myargv[1] = "hello from child from
exec\n";
            myargv[2] = NULL;
```

Illustrated example, continued

```
        //we want our child to execute, and then
          // whatever the output is, we are going to
          // pipe that to our parent process.
          // Our parent process will then exec using
the
          // childs output as its input data,
reading in from the read end
          // of our pipe.
          //
          // Let's setup the communication through
our pipe.
          // (1) First thing is-- we don't want our
child to output
          //     as soon as it executes to the
terminal.
          close(STDOUT_FILENO);
```

```
          // (2) Okay, now we do want to capture the
output somewhere however!
          //     The 'dup2' command duplicates the
file descriptor
          //     fd[1] into STDOUT_FILENO.
          //     Note: Printing out their values
would still be unique, but
          //           they are both writing to the
same locations.
          dup2(fd[1], STDOUT_FILENO);
          //   ^
          //   So this means we can now 'write' to
our pipe either explicitly
          //   through fd[1] or STDOUT.
          //
          //   Let's go ahead and write some data
into our pipe now.
          //   It won't be printed until later on
however.
```

Illustrated example, continued

```
        dprintf(fd[1], "hello msg from child sent
and buffered in pipe\n");

        // So when we are done with a file
descriptor (just like a file)
        // we always close it (and now you know
when we open a file up, it
        // is just opening up a handle to read
and/or write to some file using
        // a file handle or a file descriptor)
        close(fd[1]);   // We are done with fd[1].
        close(fd[0]);   // We also do not need
stdin.

        // Now that everything is setup, we can
execute our child.
        // We will then use the output from this
command, as the input
        // into our parent.
        execvp(myargv[0], myargv);
    }
```

```
  else {
        // The 'waitpid' command allows us to wait
on a specific child process
        // id. And we have this childProcessID
stored, so we use that.
        waitpid(childProcessID, &child_status, 0);
        // Okay, now lets finish off process
communication.
        close(STDIN_FILENO); // close stdin,
because that is going to come
                             // from our child
process.
        dup2(fd[0], 0);      // Our 'new' stdin is
going to come from the
                             // read end of the
pipe.
        close(fd[1]);        // We can also close
the 'write' file desc.
                             // because from our
parent we can simply
                             // write out to
STDOUT_FILENO by default.
```

Illustrated example, continued

```
        // Now we can write out the data that is
in our pipe.
        // The data has been sitting in a buffer
in our pipe, and is
        // ready to be 'flushed' out and written
through STDOUT.
        // We are going to do this one character
at a time.
        printf("======= In parent process
=====\n");
        char c;
        while(read(STDIN_FILENO, &c, 1) > 0) {
          write(STDOUT_FILENO, &c, 1);
        }
        // And at this point, we are done!
    }


    return 0;
}
```

End of Lecture