CPS 310
Unix I/O and
Inter-Process Communication (IPC)

Jeff Chase
Duke University

http://www.cs.duke.edu/~chase/cps310
The kernel

### System Call Layer
- Files, processes, IPC, thread syscalls

### Fault Entry
- VM page faults, signals, etc.

### Thread/CPU/Core Management
- Sleep and ready queues

### Memory Management
- Block/page cache

### I/O Completions

### Interrupt/Return

### Timer Ticks
thread states

- running
- ready
- blocked
- sleep
- yield preempt
- dispatch
- wakeup
- stop
- wait
Process, kernel, and syscalls

```
read() {
...
}
write() {
...
}
```

I/O descriptor table

syscall dispatch table

syscall stub

user buffers

process user space

Read to user mode
Unix syscalls we care about now

- `fork`, `exec*`, `exit`, `wait*`
  - Use `execvp` and `waitpid`
- `open`, `read`, `write` (+ `lseek`)
- `pipe`
- `dup*`
  - Use `dup2`
- `close`
- `chdir`

- (getpid, getpgid, setpgid, tcsetpgrp, kill, killpg)
Platform abstractions

- Platforms provide “building blocks”…
- ...and APIs to use them.
  - Instantiate/create/allocate
  - Manipulate/configure
  - Attach/detach
  - Combine in uniform ways
  - Release/destroy

The choice of abstractions reflects a philosophy of how to build and organize software systems.
Second there have always been fairly severe size constraints on the system and its software. Given the partiality antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy but a certain elegance of design. This may be a thinly disguised version of the “salvation through suffering” philosophy, but in our case it worked.
Unix programming environment

Standard Unix programs read a byte stream from standard input (fd==0).

They write their output to standard output (fd==1).

Stdin or stdout might be bound to a file, pipe, device, or network socket.

That style makes it easy to combine simple programs using pipes or files.

If the parent sets it up, the program doesn’t even have to know.

The processes may run concurrently and are synchronized automatically.
A process has multiple **channels** for data movement in and out of the process (I/O).

The channels are typed.

Each channel is named by a file **descriptor**.

The parent process and parent program set up and control the channels for a child (until **exec**).

I/O channels (“file descriptors”): stdin, stdout, stderr, tty, pipe, socket.
Standard I/O descriptors

Open files or other I/O channels are named within the process by an integer **file descriptor** value.

I/O channels ("file descriptors")

- `stdin`
- `stdout`
- `stderr`

Open files or other I/O channels are named within the process by an integer **file descriptor** value.

```
count = read(0, buf, count);
if (count == -1) {
    perror("read failed"); /* writes to stderr */
    exit(1);
}

count = write(1, buf, count);
if (count == -1) {
    perror("write failed"); /* writes to stderr */
    exit(1);
}
```

Standard descriptors for primary input (stdin=0), primary output (stdout=1), error/status (stderr=2).

Their bindings are **inherited** from the parent process and/or set by the parent program.

By default they are bound to the **controlling tty**.
Files: open syscall

Possible `<option>` flags for `open`:

- **O_RDONLY**: open for reading only
- **O_WRONLY**: open for writing only
- **O_RDWR**: open for reading and writing
- **O_APPEND**: append data to end of file on each write
- **O_CREAT**: create file if it does not already exist
- **O_TRUNC**: truncate size to 0 if it exists
- **O_EXCL**: give an error if O_CREAT and the file exists

**Example:**
```
#include <fcntl.h>
fd = open("somefile", O_CREAT | O_TRUNC | O_WRONLY);
```
Syscalls like `open` (or `exec*`) interpret a file `pathname` relative to the current directory of the calling process.

If the pathname starts with "/" then it is relative to the root directory.

A process inherits its current directory from parent process across `fork`.

A process can change its current directory with the `chdir` syscall.

“.” in a pathname means current directory (curdir).
“..” in a pathname means parent directory of curdir.
Unix process: parents rule

Created with **fork** by parent program running in parent process.

Parent program running in child process, or **exec’d** program chosen by parent program.

**Inherited** from parent process, or modified by parent program in child (e.g., using **dup* to remap descriptors**).

Parent chooses environment (**argv[]** and **envp[]**) on **exec**.

**Inherited** from parent on **fork**, or reset for child program on **exec**.

Initial VAS contents **inherited** from parent on **fork**, or reset for child program on **exec**.
Unix shell(s)  
An early interactive programming environment

- Classical Unix programs are packaged software components: specific functions with narrow, text-oriented interfaces.
- Shell is a powerful (but character-based) user interface.
  - Shell is a user program that uses kernel syscalls to execute utility programs as child processes.
  - A typical shell command is just the name of a program to run.
- Shell is also a programming environment for composing and coordinating programs interactively.
- So: it’s both an interactive programming environment and a programmable user interface.
  - Both ideas were “new” at the time (late 1960s).
- Its powerful scripting capabilities are widely used in large-scale system administration and services, for 40+ years!
A simple program: cat

- /bin/cat is a standard Unix utility program.
- By default it copies stdin to stdout in a loop of read/write syscalls.
- It sleeps as needed and exits only if an error occurs or it reads an EOF (end-of-file) from its input.
- We can give cat arguments with a list of files to read from, instead of stdin. It copies all of these files to stdout. (“concatenate”)

```c
while ((nr = read(rfd, buf, bsize)) > 0) {
    for (off = 0; nr; nr -= nw, off += nw) {
        if ((nw = write(wfd, buf + off, (size_t)nr)) < 0)
            err(1, "stdout")
    }
}
```

From http://svnweb.freebsd.org/base/head/bin/cat/cat.c
A shell running a command

**dsh** runs in child to initialize kernel state and launch the program. The child inherits stdin and stdout bound to the controlling terminal.

**dsh** calls **execvp** to overlay itself with a child program in a named file (/bin/cat). It passes the empty argv array (argv[0] == "cat") and the env array inherited from the parent.

**cat** executes, starting with main(). It issues a **read** syscall on stdin and **sleeps** until an input line is entered on the terminal....
A shell running a command

Child process running **dsh**: it is a clone of the parent, including a snapshot of parent’s internal data (from parser, etc.), and any open descriptors, current directory, etc.

Parent program (**dsh**) runs in child to initialize kernel state: e.g.: **open** files, **close** descriptors, remap descriptors with **dup2**, modify **argv/env** arrays.

**dsh** calls **execvp** to overlay itself with a child program in a named file (**cat**), passing **argv/env** arrays.

**cat** executes, starting with **main**(argc, argv, envp). It copies data with **read**(stdin) and **write**(stdout) syscalls until it is done, then exits the child.

Parent calls **waitpid** to sleep until child exits.

Parent wakes up, checks exit status…
A shell running a command

**dsh** runs in child to initialize kernel state. Open file “in” in the current directory and use **dup2** to map it to stdin. Open/truncate/create file “out” and use **dup2** to map it to stdout. Close any unneeded descriptors.

**dsh** calls **execvp** to overlay itself with a child program in a named file (**/bin/cat**). It passes the empty argv array (argv[0] == “cat”) and the env array inherited from the parent.

**cat** executes, starting with main(). It copies data from stdin to stdout with a loop of **read/write** syscalls until a syscall fails or **read** gets an EOF. If EOF, it exits with status 0.

Parent calls **waitpid** to sleep until child exits.

Parent wakes up, receives 0 status, prints next prompt…
If child is to run in the **foreground**: 
Child receives/takes control of the terminal (tty) input (tcsetpgrp). 
The foreground process receives all tty input until it stops or exits. 
The parent **waits** for a foreground child to stop or exit.
Unix defines uniform, modular ways to combine programs to build up more complex functionality.
A key idea: Unix pipes

Creating a programming philosophy from pipes and a tool box

As technically neat as the accomplishment was, when Thompson created pipes, he also put something else into UNIX -- a philosophy.

As McIlroy described it, "the philosophy that everyone started to put forth was 'Write programs that do one thing and do it well. Write programs to work together. Write programs that handle text streams, because that is a universal interface.'"

"All of these ideas, which add up to the tool approach, might have been there in unformed way prior to pipes, but they really came in afterwards," he said.

Kernighan agreed. He noted that while input/output direction predates pipes, the development of pipes led to the concept of tools -- software programs that would be in a "tool box," available when you need them.

He noted that pipes made software programs somewhat analogous to working with Roman numerals instead of Arabic numerals. "It's not that you can't do arithmetic," he said, "but it is a bear."

"I remember the preposterous syntax, the ">,>" or whatever the syntax that everyone came up with, and then all of a sudden there was the vertical bar ( | ) and just everything clicked at that point," he said. The bar was the syntax that made pipes work: who | cat | grep.

"And that's, I think, when we started to think consciously about tools, because then you could compose things together....compose them at the keyboard and get 'em right every time."

Next: 'What you think is going on, is going on'

[http://www.bell-labs.com/history/unix/philosophy.html]
Shell pipeline example

chase$ who | grep chase
chase  console  Jan 13 21:08
chase  ttys000  Jan 16 11:37
chase  ttys001  Jan 16 15:00
chase$
A pipe is a bounded kernel buffer for passing bytes.

- **The pipe()** system call creates a **pipe** object.
- A pipe has one read end and one write end: **unidirectional**.
- Bytes placed in the pipe with **write** are returned by **read** in order.
- The **read** syscall blocks if the pipe is empty.
- The **write** syscall blocks if the pipe is full.
- **Write** fails (SIGPIPE) if no process has the read end open.
- **Read** returns EOF if the pipe is empty and all writers close the pipe.

```c
int pfd[2] = {0, 0};
pipe(pfd);
/* pfd[0] is read, pfd[1] is write */

b = write(pfd[1], "12345\n", 6);
b = read(pfd[0], buf, b);
b = write(1, buf, b);
12345
```
How to plumb the pipe?

1. P creates pipe.

2. P forks C1 and C2. Both children inherit both ends of the pipe, and stdin/stdout/ stderr. Parent closes both ends of pipe after fork.

3A. C1 closes the read end of the pipe, closes its stdout, "dups" the write end onto stdout, and execs.

3B. C2 closes the write end of the pipe, closes its stdin, "dups" the read end onto stdin, and execs.
Simpler example
Feeding a child through a pipe

Parent
```c
int ifd[2] = {0, 0};
pipe(ifd);
cid = fork();
```

Child
```c
close(0);
close(ifd[1]);
dup2(ifd[0],0);
close(ifd[0]);
execve(...);
```

Parent
```c
close(ifd[0]);
count = read(0, buf, 5);
count = write(ifd[1], buf, 5);
waitpid(cid, &status, 0);
printf("child %d exited…");
```

Chase
```c
man dup2
cc -o childin childin.c
./childin cat5
12345
12345
5 bytes moved
child 23185 exited with status 0
```
Pipe reader and writer run concurrently

Context switch occurs when one process is forced to sleep because the pipe is full or empty, and maybe at other times.
Notes on pipes

• All the processes in a pipeline run **concurrently**. The order in which they run is not defined. They may execute at the same time on multiple cores. They do **NOT** execute in sequence (necessarily).

• Their execution is “synchronized by producer/consumer bounded buffer”. (More about this next month.) It means that a writer blocks if there is no space to **write**, and a reader blocks if there are no bytes to **read**. Otherwise they may both run: they might not, but they could.

• They do **NOT** read all the input before passing it downstream.

• The **pipe** itself must be created by a common parent. The children inherit the pipe’s file descriptors from the parent on **fork**. Unix has no other way to pass a file descriptor to another process.

• How does a reader “know” that it has read all the data coming to it through a pipe? **Answer**: there are no bytes in the pipe, and no process has the write side open. Then the **read** syscall returns EOF.
C1/C2 user pseudocode
while(\texttt{until EOF}) {
    \texttt{read}(0, \texttt{buf}, \texttt{count});
    \texttt{compute/transform data in buf};
    \texttt{write}(1, \texttt{buf}, \texttt{count});
}

\texttt{cat} \mid \texttt{cat}
(Note: try this scenario to debug pipes!)

Kernel pseudocode for pipes: Producer/consumer bounded buffer

**Pipe write:** copy in bytes from user buffer to in-kernel pipe buffer, blocking if k-buffer is full.

**Pipe read:** copy bytes from pipe's k-buffer out to u-buffer. Block while k-buffer is empty, or return EOF if empty and pipe has no writer.
Kernel-space pseudocode
System call internals to read/write N bytes for buffer size B.

read(buf, N)
{
    for (i = 0; i++ < N) {
        move one byte into buf[i];
    }
}

Pipes
Pipes

```c
read(buf, N) {
    pipeMx.lock();
    for (i = 0; i++ < N) {
        while (no bytes in pipe)
            dataCv.wait();
        move one byte from pipe into buf[i];
        spaceCV.signal();
    }
    pipeMx.unlock();
}
```

Read N bytes from the pipe into the user buffer named by `buf`. Think of this code as deep inside the implementation of the `read` system call on a pipe. The `write` implementation is similar.
Pipes

read(buf, N)
{
    readerMx.lock();
    pipeMx.lock();
    for (i = 0; i++; i<N) {
        while (no bytes in pipe)
            dataCv.wait();
        move one byte from pipe into buf[i];
        spaceCV.signal();
    }
    pipeMx.unlock();
    readerMx.unlock();
}

In Unix, the read/write system calls are “atomic” in the following sense: no read sees interleaved data from multiple writes. The extra lock here ensures that all read operations occur in a serial order, even if any given operation blocks/waits while in progress.
Why exactly does Pipe (bounded buffer) require a nested lock?

First: remember that this is the exception that proves the rule. Nested locks are generally not necessary, although they may be useful for performance. Correctness first: always start with a single lock.

Second: the nested lock is not necessary even for Pipe if there is at most one reader and at most one writer, as would be the case for your typical garden-variety Unix pipe.

The issue is what happens if there are multiple readers and/or multiple writers. The nested lock is needed to meet a requirement that read/write calls are atomic. Understanding this requirement is half the battle.

Consider an example. Suppose three different writers \{A, B, C\} write 10 bytes each, each with a single write operation, and a reader reads 30 bytes with a single read operation. The read returns the 30 bytes, so the read will "see" data from multiple writes. That’s OK. The atomicity requirement is that the reader does not observe bytes from different writes that are interleaved (mixed together).

A necessary condition for atomicity is that the writes are serialized: the system chooses some order for the writes by A, B, and C, even if they request their writes "at the same time". The data returned by the read reflects this ordering. Under no circumstances does a read see an interleaving, e.g.: 5 bytes from A, then 5 bytes from B, then 5 more bytes from A,… (Note: if you think about it, you can see that a correct implementation must also serialize the reads.)

This atomicity requirement exists because applications may depend on it: e.g., if the writers are writing records to the pipe, then a violation of atomicity would cause a record to be “split”.

This is particularly important when the size of a read or write (N) exceeds the size of the bounded buffer (B), i.e., N>B. A read or write with N>B is legal. But such an operation can’t be satisfied with a single buffer’s worth of data, so it can’t be satisfied without alternating execution of a reader and a writer ("ping-pong style"). On a single core, the reader or writer is always forced to block at least once to wait for its counterparty to place more bytes in the buffer (if the operation is a read) or to drain more bytes out of the buffer (if the operation is a write). In this case, it is crucial to block any other readers or writers from starting a competing operation. Otherwise, atomicity is violated and at least one of the readers will observe an interleaving of data.

The nested lock ensures that at most one reader and at most one writer are moving data in the “inner loop” at any given time.
Processes reference objects

The kernel objects referenced by a process have **reference counts**. They may be destroyed after the last ref is released, but not before.

What operations release ref counts?
Fork clones all references

Child **inherits** open descriptors from parent.

Cloned descriptors share a read/write offset.

**Fork** increments reference counts on shared objects.

Cloned references to VM segments are likely to be **copy-on-write** to create a lazy, virtual copy of the shared object.

**What operations release ref counts?**
Unix dup* syscall

dup2(oldfd1, newfd2). What does it mean?

Yes, fd1 and fd2 are integer variables. But let’s use “fd” as shorthand for “the file descriptor whose number is the value of the variable fd”.

Then fd1 and fd2 denote entries in the file descriptor table of the calling process. The dup2 syscall is asking the kernel to operate on those entries in a kernel data structure. It doesn’t affect the values in the variables fd1 and fd2 at all! And it doesn’t affect the I/O objects themselves (e.g., the pipe) at all.
Unix dup* syscall

Then \texttt{dup2(oldfd1, newfd2)} means: “\texttt{close(fd2) (if it was open) then set fd2 to refer to the same underlying I/O object as fd1}.”

It results in two file descriptors referencing the same underlying I/O object. You can use either of them to \texttt{read/write}: they are equivalent.

But you should probably just \texttt{close(fd1)}. 

---

**Diagram:**

- \texttt{int fd2 = 0}
- \texttt{int fd1 = 5}
- Per-process descriptor table
- Final state after \texttt{dup2(fd1, fd2)} syscall.
Unix dup* syscall

Why should you close(fd1) after dup2(fd1, fd2)? Because the kernel uses reference counting to track open file descriptors. Resources associated with an I/O object are not released until all of the open descriptors on the object are closed. They are closed automatically on exit, but you should always close any descriptors that you don’t need.
Unix dup* syscall

Then \texttt{dup2(fd1,fd2); close(fd1)} means: “remap the object referenced by file descriptor fd1 to fd2 instead”.

It is convenient for remapping descriptors onto \texttt{stdin, stdout, stderr}, so that some program will use them “by default” after \texttt{exec*}.

Note that we still have not changed the values in \texttt{fd1} or \texttt{fd2}. Also, changing the values in \texttt{fd1} and \texttt{fd2} can never affect the state of the entries in the file descriptor table. Only the kernel can do that.
MapReduce is a filter-and-pipe model for **data-intensive cluster computing**. Its programming model is “like” Unix pipes. It adds “parallel pipes” (my term) that split streams among multiple downstream children.
Unix: simple abstractions?

- users
- files
- processes
- pipes
  - which “look like” files

These persist across reboots. They have **symbolic names** (you choose it) and **internal IDs** (the system chooses).

These exist within a running system, and they are transient: they disappear on a crash or reboot. They have internal IDs.

Unix supports dynamic create/destroy of these objects. It manages the various name spaces. It has system calls to access these objects. It checks permissions.
Let’s pause a moment to reflect...


Note log scale

Today Unix runs embedded in devices costing < $100.
Sockets

A **socket** is a buffered channel for passing data over a network.

- The **socket()** system call creates a **socket** object.
- Other socket syscalls establish a connection (e.g., **connect**).
- A file descriptor for a connected socket is **bidirectional**.
- Bytes placed in the socket with **write** are returned by **read** in order.
- The **read** syscall blocks if the socket is empty.
- The **write** syscall blocks if the socket is full.
- Both **read** and **write** fail if there is no valid connection.

```c
client
int sd = socket(<internet stream>);
gethostbyname(“www.cs.duke.edu”);
<make a sockaddr_in struct>
<install host IP address and port>
connect(sd, <sockaddr_in>);
write(sd, “abcdefg”, 7);
read(sd, ….);
```
A simple, familiar example

request
“GET /images/fish.gif HTTP/1.1”

client (initiator)

s = socket(...);
connect(sd, name);
write(sd, request...);
read(sd, reply...);
close(sd);

server

s = socket(...);
bind(s, name);
sd = accept(s);
write(sd, request...);
read(sd, reply...);
close(sd);
struct sockaddr_in socket_addr;
sock = socket(PF_INET, SOCK_STREAM, 0);

memset(&socket_addr, 0, sizeof(socket_addr);
socket_addr.sin_family = PF_INET;
socket_addr.sin_port = htons(port);
socket_addr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *) &socket_addr, sizeof(socket_addr) < 0) {
    perror("bind failed");
    exit(1);
}
listen(sock, 10);

while (1) {
    int acceptsock = accept(sock, NULL, NULL);
    forkme(acceptsock, prog, argv);
    close(acceptsock);
}