Project 0 has been released, Due 11/09. This is an individual assignment, but future projects will be in assigned teams.

Homework 1 has been released and is due Monday, 18/9.
Project 0 has been released, Due 11/09. This is an individual assignment, but future projects will be in assigned teams.

Homework 1 has been released and is due Monday, 18/9.
We (still) don’t bite!

Lectures go fast.

When reviewing the material, ask questions on EdStem now!

You can do so anonymously.
Recall: Hardware must support

1) Privileged Instructions
   Unsafe instructions cannot be executed in user mode

2) Memory Isolation
   Memory accesses outside a process’s address space prohibited

3) Interrupts
   Ensure kernel can regain control from running process

4) Safe Transfers
   Correctly transfer control from user-mode to kernel-mode and back
Recall: Really Really Really Really Big Idea

The state of a program’s execution is succinctly and completely represented by CPU register state

EIP, ESP, EBP, Eflags/PSW
Recall: Interrupt Summary

1) Device sends signal to APIC

2) Processor detects interrupt

3) CPU saves Recovery State and switch to Kernel Stack

4) CPU jumps to interrupt handler table at appropriate vector.

5) Kernel runs interrupt handler

6) Restore user program
Recall: System Call/Exceptions

1) Processor traps

2) CPU saves Recovery State and switch to Kernel Stack

3) CPU jumps to interrupt handler table at appropriate vector.

4) Kernel runs interrupt handler

5) Restore user program
Recall: User Stack/Kernel Stack

User-level Process

```plaintext
code:

foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}

stack:
```

Registers

- SS: ESP
- CS: EIP
- EFLAGS
- other registers: EAX, EBX, ...

Kernel

```plaintext
code:

handler() {
    pusha
    ...
}

Exception Stack
```

Crooks CS162 © UCB Fall 2023
Recall: User Stack/Kernel Stack

User-level Process

code:

foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}

stack:

Registers

SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...

Kernel

code:

handler() {
    pusha
    ...
}

Exception Stack

SS
ESP
EFLAGS
CS
EIP
eerror
Three “Prongs” for the Class

Understanding OS principles

System Programming

Map Concepts to Real Code
What APIs should the OS present for process creation and control?

“Everything is a file”: says Unix. What does IO look like in Unix?
Goal 1: The Process API
Simple is Beautiful

Can describe majority of process management (and input/output) using only a small number of system calls

System calls (mostly) unchanged since 1973
Keeping it in the family

Processes in Linux form a family tree

Each process in Linux has exactly one parent

Each process in Linux can have many children

All processes start from the main init process
Examples

A parent process spawns a child process

crooks@laptop> ./parent
Parent: Hello, World!
Parent: Waiting for Child to complete.
Child: Hello, World! 1053

crooks@laptop > ps -x --forest
PID TTY STAT TIME COMMAND
1 ? Ss 0:00 /init
7 ? Ss 0:00 /init
8 ? S 0:01 init
9 pts/0 Ss 0:00 __/init
563 pts/0 Sp 0:00 \_ tmux
565 ? Ss 0:04 \_ tmux
566 pts/1 Ss 0:00 \_ bash
1054 pts/1 Sp 0:00 \_ ps -x --forest
883 pts/2 Ss 0:00 \_ bash
1052 pts/2 Sp 0:00 \_ ./parent
1053 pts/2 Sp 0:01 \_ \_
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
struct task_struct *task;
for (task = current; task != &init_task; task = task->parent) {
    printk("%s[%d]\n", task->comm, task->pid);
}
printk("%s[%d]\n", task->comm, task->pid);

What does the final print statement print?
Process Management API

- **exit** – terminate a process
- **fork** – copy the current process
- **exec** – change the *program* being run by the current process
- **wait** – wait for a process to finish
- **kill** – send a *signal* (interrupt-like notification) to another process
- **sigaction** – set handlers for signals
Process Management API

- **exit** – terminate a process
- **fork** – copy the current process
- **exec** – change the *program* being run by the current process
- **wait** – wait for a process to finish
- **kill** – send a *signal* (interrupt-like notification) to another process
- **sigaction** – set handlers for signals
Process Management API

exit – terminate a process

fork – copy the current process

exec – change the program being run by the current process

wait – wait for a process to finish

kill – send a signal (interrupt-like notification) to another process

sigaction – set handlers for signals
A fork in the road

Creates a new child process from an original parent process

Child is copy of parent process

Fork call makes complete copy of process state
- Address Space
- Code/Data Segments
- Registers (including PC and SP)
- Stack
- Pointers to Files/IO (File descriptors – see later)
Forking under the hood

1. Allocate a new PCB.

2. Duplicates:
   - Register Values
   - Address Space
   - Flags
   - Register State
   - Open Files

3. Allocates new PID

4. Mark process as in the READY state

In Linux `do_fork()` defined in `<kernel/fork.c>`
Using Fork

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

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid(); /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
Forked Processes & Identical Twins

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

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid(); /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid();            /* get current
processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid();                /* get current
     processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) {                      /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) {             /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
Fork Ordering

What does this print?

Would adding the calls to `sleep()` matter?

Remember! Full copy of address space. Once forked, modify different memory locations.

Arbitrary interleaving of processes
Review: The Life of a Fork() Syscall

1. Fork() is a System Call! Invoke int 0x80 instruction

2. CPU switches to kernel stack and copies recovery state

3. CPU Jumps to interrupt vector table (index 128). Invokes system_call_handler()

4. Handler identifies fork() using system call dispatch table (syscall number stored in %eax register)

5. do_fork() creates a new child PCB with duplicated memory context and *same* EIP

6. Schedule either child or parent process
The Battle Continues

A fork() in the road

Andrew Baumann
Microsoft Research

Jonathan Appavoo
Boston University

Orran Krieger
Boston University

Timothy Roscoe
ETH Zurich

ABSTRACT

The received wisdom suggests that Unix’s unusual combination of fork() and exec() for process creation was an inspired design. In this paper, we argue that fork was a clever hack for machines and programs of the 1970s that has long outlived its usefulness and is now a liability. We catalog the ways in which fork is a terrible abstraction for the modern programmer to use, describe how it compromises OS implementations, and propose alternatives.

As the designers and implementers of operating systems, we should acknowledge that fork’s continued existence as a first-class OS primitive holds back systems research, and deprecate it. As educators, we should teach fork as a historical artifact, and not the first process creation mechanism students encounter.

ACM Reference Format:

1 INTRODUCTION

When the designers of Unix needed a mechanism to create processes, they added a peculiar new system call: fork(). As every undergraduate now learns, fork creates a new process identical to its parent (the caller of fork), with the exception of the system call’s return value. The Unix idiom of fork() followed by exec() to execute a different program in the child is now well understood, but still stands in stark contrast 50 years later, fork remains the default process creation API on POSIX. Attilakis et al. [8] found 1304 Ubuntu packages (7.2% of the total) calling fork, compared to only 41 uses of the more modern posix_spawn(). Fork is used by almost every Unix shell, major web and database servers (e.g., Apache, PostgreSQL, and Oracle), Google Chrome, the Redis key-value store, and even Node.js. The received wisdom appears to hold that fork is a good design. Every OS textbook we reviewed [4, 7, 9, 35, 75, 78] covered fork in uncritical or positive terms, often noting its “simplicity” compared to alternatives. Students today are taught that “the fork system call is one of Unix’s great ideas” [46] and “there are lots of ways to design APIs for process creation; however, the combination of fork() and exec() are simple and immensely powerful … the Unix designers simply got it right” [7]. Our goal is to set the record straight. Fork is an anachronism: a relic from another era that is out of place in modern systems where it has a pernicious and detrimental impact. As a community, our familiarity with fork can blind us to its faults (§4). Generally acknowledged problems with fork include that it is not thread-safe, it is inefficient and unscalable, and it introduces security concerns. Beyond these limitations, fork has lost its classic simplicity; it today impacts all the other operating system abstractions with which it was once orthogonal. Moreover, a fundamental challenge with fork is that, since it conflates the process and the address space in which it runs, fork is hostile to user-mode implementation of OS functionality, breaking everything from buffered IO to kernel-bypass networking. Perhaps most problematically, fork doesn’t compose—every layer of a system from the kernel to the smallest user-mode library must support it.

We illustrate the harm fork wreaks on OS implementa...
Process Management API

- **exit** – terminate a process
- **fork** – copy the current process
- **exec** – change the *program* being run by the current process
- **wait** – wait for a process to finish
- **kill** – send a *signal* (interrupt-like notification) to another process
- **sigaction** – set handlers for signals
Exec()

Call to Exec replaces running program!

... cpid = fork();
if (cpid > 0) {
    /* Parent Process */
    ....
} else if (cpid == 0) {
    /* Child Process */
    char *args[] = {"ls", "-l", NULL};
    execv("/bin/ls", args);
    /* execv doesn’t return when it works. So, if we got here, it failed! */
    perror("execv");
    exit(1);
}
...

Exec System Call handler will:

1. Replace the code and data segment

2. Set EIP to point to start of new program/reinitialize SP and FP

3. Push arguments to program onto stack.
Isn’t this wasteful?

OS copies entire memory of process, only to overwrite it with new process

Can actually be made quite fast using intelligent copy-on-write mechanisms

(Only physically copy memory when content is different)
Fork/Exec Pattern
Process Management API

- **exit** – terminate a process
- **fork** – copy the current process
- **exec** – change the *program* being run by the current process
- **wait** – wait for a process to finish
- **kill** – send a *signal* (interrupt-like notification) to another process
- **sigaction** – set handlers for signals
Wait blocks parent process until one of its children processes exits

In what order will the (parent/child) says bye sentences be outputted?

Question: how would parent wait for all children to finish?
Process Management API

- **exit**: terminate a process
- **fork**: copy the current process
- **exec**: change the *program* being run by the current process
- **wait**: wait for a process to finish
- **kill**: send a *signal* (interrupt-like notification) to another process
- **sigaction**: set handlers for signals
**What is a Signal?**

A *signal* is a very **short** message that may be sent to a process or a group of processes.

1) **Make process aware that specific event has occurred**

2) **Allow process to execute a signal handler function when event has occurred**

**Example of a kernel- user mode transition**
What is a Signal?

Each signal has a default action. Can override default action with signal handler.

Program jumps to signal handler function.

Control of the program resumes at the previously interrupted instructions.
Signals in the Wild (in the Linux Kernel)

Ctrl + C . Sends SIGINT signal. Default actual is to kill the program

Timer signal. Check every T seconds that a condition still holds

```
crooks@laptop> man 7 signal
```
Process Management API

exit – terminate a process
fork – copy the current process
exec – change the program being run by the current process
wait – wait for a process to finish
kill – send a signal (interrupt-like notification) to another process
sigaction – set handlers for signals
Goal 2: Input/Output in Linux
UNIX offers the same IO interface for:

- All device Input/Output
- Reading/Writing Files
- Interprocess communication

Everything is a file!
The UNIX Time-Share System

Dennis M. Ritchie and Ken Thompson
Bell Laboratories

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems, including a dozen languages. This paper discusses the nature and implementation of the file system and of the user command interface.

Key Words and Phrases: time-sharing, operating system, file system, command language, PDP-11

CR Categories: 4.30, 4.32

1. Introduction

There have been three versions of UNIX. The earliest version (circa 1969–70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. This paper describes only the PDP-11/40 and -45 [1] system since it is more modern and many of the differences between it and older UNIX systems result from redesign of features found to be deficient or lacking.

Since PDP-11 UNIX became operational in February 1971, about 40 installations have been put into service; they are generally smaller than the system described here. Most of them are engaged in applications such as the preparation and formatting of patent applications and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as $40,000, and less than two man years were spent on the main system software. Yet UNIX contains a number of features seldom offered even in much larger systems. It is hoped, however, that the users of UNIX will find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the system proper, the major programs available under UNIX are: assembler, text editor based on QED [2], linking loader, symbolic debugger, compiler for a language resembling BCPL [3] with types and structures (C), interpreter for a dialect of BASIC, text formatting program, Faster, graphics editor, and several others.
Core tenants of UNIX/IO interface

**Uniformity**
Same set of system calls
Open, read, write, close

**Open Before Use**
Must explicitly open
file/device/channel

**Byte-Oriented**
All devices, even block
devices, are access
through byte arrays

**Kernel Buffered**
Reads/Writes
Data is buffered at
kernel to decouple
internals from application

**Explicit Close**
Must explicitly close
resource
Introducing the File Descriptor

Number that *uniquely* identifies an open IO resource in the OS

It’s another index!

File descriptors index into

a per-process file descriptor table
FDs in the Wild (well, in the Kernel)

In Linux, the `fdtable` structure is defined in `include/kernel/fdtable.h`.

```c
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE }

// Per-process state
struct proc {
    uint sz;                     // Size of process memory (bytes)
    pde_t* pgdir;                // Page table
    char *kstack;                // Bottom of kernel stack for this process
    enum procstate state;       // Process state
    int pid;                     // Process ID
    struct proc *parent;        // Parent process
    struct trapframe *tf;        // Trap frame for current syscall
    struct context *context;     // swtch() here to run process
    void *chan;                  // If non-zero, sleeping on chan
    int killed;                  // If non-zero, have been killed
    struct file *ofile[NOFILE];  // Open files
    struct inode *cwd;           // Current directory
    char name[16];               // Process name (debugging)
};
```
Table of Open File Description

Each FD points to an open file description in a system-wide table of open files

File offset
File access mode (from open())
File status flags (from open())
Reference to physical location (inode – more later)
Number of times opened

In Linux struct file defined in
<include/linux/fs.h>
Manipulating FDs

### Open/Create

All files explicitly opened via open or create. Return the lowest-numbered file descriptor not currently open for the process. Creates new open file description.

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags, [mode_t mode]);

int creat (const char *filename, mode_t mode);

int close (int filedes);
```
Manipulating FDs (2)

Read data from open file using file descriptor:

\[
\text{ssize_t read (int filedes, void *buffer, size_t maxsize)}
\]

Write data to open file using file descriptor:

\[
\text{ssize_t write (int filedes, const void *buffer, size_t size)}
\]

Reposition file offset within kernel:

\[
\text{off_t lseek (int filedes, off_t offset, int whence)}
\]
Example

char buffer1[100];
char buffer2[100];
Example

```c
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
```
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
write(fd2, buffer2, 100);
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
write(fd2, buffer2, 100);
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
write(fd2, buffer2, 100);
close(fd)
Example

```c
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
write(fd2, buffer2, 100);

close(fd); close(fd2)
```
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
write(fd2, buffer2, 100);
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
write(fd2, buffer2, 100);

int fd3 = dup(fd2);

**Duplicating FDs!**

Creates copy fd3 of file descriptor fd2
Duplicating FDs!

char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
write(fd2, buffer2, 100);

int fd3 = dup(fd2);
read(fd2, buffer1, 100);
```c
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
write(fd2, buffer2, 100);

int fd3 = dup(fd2);
read(fd2, buffer1, 100);
read(fd3, buffer1, 100);
```
Duplicating FDs!

```c
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
write(fd2, buffer2, 100);

int fd3 = dup(fd2);
read(fd2, buffer1, 100);
read(fd3, buffer1, 100);
close(fd2);
```
Duplicating FDs!

char buffer1[100];
char buffer2[100];
int fd = open("foo.txt", O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

int fd2 = open("bar.txt", O_RDWR);
read(fd2, buffer1, 100);
write(fd2, buffer2, 100);

int fd3 = dup(fd2);
read(fd2, buffer1, 100);
read(fd3, buffer1, 100);
close(fd2); close(fd3)

Open file description remains alive until no file descriptors refer to it
Forking FDs

Per-Process File Descriptor Table

O: STDIN
1: STDOUT
2: STDERR
3
4

Global Open File Description Table

<table>
<thead>
<tr>
<th>Mode</th>
<th>Flags</th>
<th>Offset</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>R</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>RW</td>
<td>200</td>
<td></td>
</tr>
</tbody>
</table>
Forking FDs

Per-Process File Descriptor Table

O: STDIN
1: STDOUT
2: STDERR
3
4

Global Open File Description Table

Mode | Flags | Offset | Phys
--- | --- | --- | ---
U | R | 200 | 
U | RW | 200 | 

Per-Process File Descriptor Table

O: STDIN
1: STDOUT
2: STDERR
3
4

Forked process inherits copies of file descriptors
Interprocess Communication: Pipes

Pipe implements a queue abstraction. Implemented as a kernel buffer with two file descriptors, one for writing to pipe and one for reading.

Block if pipe full. Block if pipe empty.

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

Allocates two new file descriptors in the process
Writes to `fileds[1]` read from `fileds[0]`
Implemented as a fixed-size queue
### Single-Process Pipe Example

```c
#include <unistd.h>
int main(int argc, char *argv[])
{
    char *msg = "Message in a pipe.\n";
    char buf[BUFSIZE];
    int pipe_fd[2];
    if (pipe(pipe_fd) == -1) {
        fprintf(stderr, "Pipe failed.\n"); return EXIT_FAILURE;
    }
    ssize_t writelen = write(pipe_fd[1], msg, strlen(msg)+1);
    printf("Sent: %s [%ld, %ld]\n", msg, strlen(msg)+1, writelen);

    ssize_t readlen  = read(pipe_fd[0], buf, BUFSIZE);
    printf("Rcvd: %s [%ld]\n", msg, readlen);

    close(pipe_fd[0]);
    close(pipe_fd[1]);
}
```
Pipes Between Processes

int pipe_fd[2];
pipe(pipe_fd);
Pipes Between Processes

```
int pipe_fd[2];
pipe(pipe_fd);
```
After last "write" descriptor is closed, pipe is effectively closed:

Reads return only "EOF"

After last "read" descriptor is closed, writes generate SIGPIPE signals:

If process ignores, then the write fails with an "EPIPE" error
**IPC across machines: Sockets**

Sockets are an **abstraction of two queues**, one in each direction

Can read or write to either end

Used for communication between multiple processes on different machines

File descriptors obtained via 
socket/bind/connect/listen/accept

Still a file! Same API/datastructures as files and pipes
Namespaces for Network Communication

Hostname
www.eecs.berkeley.edu

IP address
128.32.244.172 (IPv4, 32-bit Integer)
2607:f140:0:81::f (IPv6, 128-bit Integer)

Port Number
0-1023 are system ports
1024-49151 are registered ports
49152–65535 are free
Sockets in concept

**Client**
- Create Client Socket
- Connect to address (Host:Port)
- Read/Write Request
- Close Client Socket

**Server**
- Create Server Socket
- Bind to address (host:port)
- Listen for Connection
- Accept Connection
- Read/Write Request
- Close Client Socket
char *host_name, *port_name;

// Create a socket
struct addrinfo *server = lookup_host(host_name, port_name);
int sock_fd = socket(server->ai_family, server->ai_socktype,
                      server->ai_protocol);

// Connect to specified host and port
connect(sock_fd, server->ai_addr, server->ai_addrlen);

// Carry out Client-Server protocol
run_client(sock_fd);

/* Clean up on termination */
close(sock_fd);
// Socket setup code elided...
while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);
    pid_t pid = fork();
    if (pid == 0) { // I am the child
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);
        exit(0);
    } else { // I am the parent
        close(conn_socket);
    }
}
close(server_socket);
Goal 2: Input/Output Unix

Everything is a file!
Files, sockets, pipes all look the same!

Per-process file descriptor table points to a global table of open file descriptions

Use open/create/read/write/close to manipulate FDs.

Forked processes inherit FDs of parents