CPS 310
Unix Process Model

Jeff Chase
Duke University

http://www.cs.duke.edu/~chase/cps310
Operating Systems: The Classical View

Programs run as independent processes.

Protected OS kernel mediates access to shared resources.

Protected system calls

...and upcalls (e.g., signals)

Each process has a private virtual address space and one or more threads.

Threads enter the kernel for OS services.

The kernel code and data are protected from untrusted processes.
Today

- We understand threads and concurrency now.
- Add a kernel and address spaces → “vanilla Unix”
- Flesh out Unix process model and syscall API
- Kernel interactions
- Using the process model
- **Imagine:**
  - Your thread library is “really” a kernel.
  - Each thread is a classical process.
  - i.e., Each thread has its own VAS.
  - i.e., Threads/processes share data only within/via the kernel.
Thread states and transitions

We will presume that these transitions occur only in kernel mode. This is true in classical Unix and in systems with pure kernel-based threads.

Before a thread can **sleep**, it must first enter the kernel via trap (syscall) or fault.

Before a thread can **yield**, it must enter the kernel, or the core must take an interrupt to return control to the kernel.

On entry to the **running** state, kernel code decides if/when/how to enter user mode, and sets up a suitable context.

E.g., for initial start, return from fault or syscall, or to deliver a signal.
Threads execute user code on a **user stack** in user space (the process virtual address space).

Each thread has a second **kernel stack** in **kernel space** (VM accessible only in kernel mode).

System calls and faults run in kernel mode on a kernel stack for the current thread.

Kernel code running in P’s process context has access to P’s virtual memory.

The syscall (trap) handler makes an indirect call through the system call dispatch table to the handler registered for the specific system call.
Upcall example: Unix signals

- Signals are asynchronous notifications to a process that some event of interest to it has occurred.
- A process may register **signal handlers** for various events relating to the process. The signal handlers are procedures in user space.
- To deliver a signal, the kernel redirects a user thread to execute a selected registered signal handler in user mode.
- Unix signals take a default action if no handler is registered.
  - E.g., segmentation fault $\Rightarrow$ die. Other actions: ignore, stop
The kernel must be bulletproof

Secure kernels handle system calls verrry carefully.

Syscalls indirect through syscall dispatch table by syscall number. No direct calls to kernel routines from user space!

What about references to kernel data objects passed as syscall arguments (e.g., file to read or write)?

Use an integer index into a kernel table that points at the data object. The value is called a handle or descriptor. No direct pointers to kernel data from user space!

Kernel copies all arguments into kernel space and validates them.

Kernel interprets pointer arguments in context of the user VAS, and copies the data in/out of kernel space (e.g., for read and write syscalls).
Unix: A lasting achievement?

“Perhaps the most important achievement of Unix is to demonstrate that a powerful operating system for interactive use need not be expensive…it can run on hardware costing as little as $40,000.”

The UNIX Time-Sharing System*
D. M. Ritchie and K. Thompson
1974

DEC PDP-11/24

http://histoire.info.online.fr/pdp11.html
Process management

- OS offers system call APIs for managing processes.
  - Create processes (children)
  - Control processes
  - Monitor process execution
  - “Join”: wait for a process to exit and return a result
  - “Kill”: send a signal to a process
  - Establish interprocess communication (IPC: later)
  - Launch a program within a process

- We study the Unix process abstraction as an example.
  - Illustrative and widely used for 40+ years!
  - Optional: Use it to build your own shell.
Example: Unix fork

The Unix `fork()` system call creates/launches a new thread, in its own fresh virtual address space: it creates a new process. (Thread + VAS == Process.)

Strangely, the new (“child”) process is an exact clone of the calling (“parent”) process.

Oh Ghost of Walt, please don’t sue me.
Unix fork/exit syscalls

int pid = fork();
Create a new process that is a clone of its parent. Return child process ID (pid) to parent, return 0 to child.

exit(status);
Exit with status, destroying the process. Status is returned to the parent.
Note: this is not the only way for a process to exit!

parent
fork
child
exit

pid: 5587
pid: 5588

Note: this is not the only way for a process to exit!
```c
int pid;
int status = 0;

if (pid = fork()) {
    /* parent */
    // ...
} else {
    /* child */
    // ...
    exit(status);
}
```

The **fork** syscall returns **twice**:

It returns a zero in the context of the new child process.

It returns the new child process ID (pid) in the context of the parent.
A simple program: sixforks

```c
... int main(int argc, char* argv) {
    fork();
    fork();
    fork();
    fork();
    fork();
    fork();
    printf("Process %d exiting.\n", getpid());
} getpid syscall: Get processID of current process.
```
A simple program: sixforks

... int main(int argc, char* argv)
{
    fork();
    fork();
    fork();
    fork();
    printf("Process %d exiting.\n", getpid());
}
sixforks: some questions

• What if I want to create six children, but I don’t want my children to have children of their own?
• What if I want the program to print the total number of processes created? How? (Other than by having the program do the math.)
• How much memory does this program use? How many pages?
• How does this test system assign process IDs?
• Why do the process IDs print out of order?
fork (original concept)

5.1 Processes

Except while UNIX is bootstrapping itself into operation, a new process can come into existence only by use of the fork system call:

processid = fork (label)

When fork is executed by a process, it splits into two independently executing processes. The two processes have independent copies of the original core image, and share any open files. The new processes differ only in that one is considered the parent process: in the parent, control returns directly from the fork, while in the child, control is passed to location label. The processid returned by the fork call is the identification of the other process.

Because the return points in the parent and child process are not the same, each image existing after a fork may determine whether it is the parent or child process.
fork in action today

void
dofork()
{
    int cpid = fork();
    if (cpid < 0) {
        perror("fork failed: ");
        exit(1);
    } else if (cpid == 0) {
        child();
    } else {
        parent(cpid);
    }
}

Fork is conceptually difficult but syntactically clean and simple.

I don’t have to say anything about what the new child process “looks like”: it is an exact clone of the parent!

The child has a new thread executing at the same point in the same program. The child is a new instance of the running program: it has a “copy” of the entire address space. The “only” change is the process ID and return value cpid!

The parent thread continues on its way. The child thread continues on its way.
int pid;  
int status = 0;  

if (pid = fork()) {  
    /* parent */  
        .....  
        pid = wait(&status);  
} else {  
    /* child */  
        .....  
        exit(status);  
}  

**Parent uses wait to sleep until the child exits; wait returns child pid and status.**

**Wait** variants allow wait on a specific child, or notification of stops and other signals.  
**Recommended**: use **waitpid()**.

**Warning**: the uses of the terms **wait** and **signal** should not be confused with the monitor/CV primitives of the same names.
Note: in modern Unix systems the wait syscall has many variants and options.

5.4 Process Synchronization
Another process control system call

```
processid = wait()
```
causes its caller to suspend execution until one of its children has completed execution. Then `wait` returns the `processid` of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available. `Wait` may also present status from a grandchild or more distant ancestor; see §5.5.

5.5 Termination
Lastly,

```
exit(status)
```
terminates a process, destroys its image, closes its open files, and generallyobliterates it. When the parent is notified through the `wait` primitive, the indicated `status` is available to the parent; if the parent has already terminated, the status is available to the grandparent, and so on. Processes
A simple program: forkdeep

```c
int count = 0;
int level = 0;

void child() {
    level++;
    output pids
    if (level < count)
        dofork();
    if (level == count)
        sleep(3); /* pause 3 secs */
}

void parent(int childpid) {
    output pids
    wait for child to finish
}

main(int argc, char *argv[]) {
    count = atoi(argv[1]);
    dofork();
    output pid
}
```

We’ll see later where arguments come from.
Example: Chrome browser

When we started this project, the gears guys were saying that one of the problems with browsers is that they're inherently single-threaded.

For example, once you have JavaScript executing, it's going to keep going, and the browser can't do anything else until JavaScript returns control to the browser.
Processes in the browser

Chrome makes an interesting choice here. But why use processes?
Problem: heap memory and fragmentation

- But as time goes on, fragmentation results — little bits of memory still get used even when a tab gets closed.
- Either we have memory that nothing can refer to again, or there's a piece of de-allocated memory we still have pointers to.
- So when the browser wants to open a new tab, it can't fit it in the existing space --
Solution: whack the whole process

When a process exits, all of its virtual memory is reclaimed as one big slab.

[Google Chrome Comics]
Processes for fault isolation

[Google Chrome Comics]
With sandboxing, our goal is to prevent malware from installing itself on your computer or using what happens in one tab to affect what happens in another.

So, for each of these processes we’ve stripped away all of their rights.

They can compute but they can’t write files to your hard drive or read files from sensitive areas like your documents or desktop.
Processes: A Closer Look

The address space is a private name space for a set of memory segments used by the process.

The kernel must initialize the process virtual memory for the program to run.

Each process has at least one thread (the “main thread”) bound to the VAS.

Each thread has a stack addressable in the VAS.

The kernel can suspend/restart a thread wherever and whenever it wants.

The OS maintains some kernel state for each process in the kernel’s internal data structures: e.g., a file descriptor table, links to maintain the process tree, current directory, and a place to store the exit status.
Process, kernel, and syscalls

syscall stub

syscall dispatch table

I/O descriptor table

I/O objects

user buffers

process user space

trap

copyout

copyin

Return to user mode

read() {…}

write() {…}

syscall stub

syscall dispatch table

I/O descriptor table

I/O objects

Return to user mode
Unix fork/exec/exit/wait syscalls

- **fork**
  - **parent**
  - **child**
  - **program initializes child context**
  - **time**

- **exec**
  - **program**
  - **argvp, envp**

- **exit**
  - **status**

- **wait**
  - **&status**

**int pid = fork();**
Create a new process that is a clone of its parent, running the same program.

**exec*(“program” [argvp, envp]);**
Overlay the calling process with a new program, and transfer control to it, passing arguments and environment.

**exit(status);**
Exit with status, destroying the process.

**int pid = wait*(&status);**
Wait for exit (or other status change) of a child, and “reap” its exit status. **Recommended**: use **waitpid()**.
But how do I run a new program in my child process?

- The child, or any process really, can replace its program in midstream.

- `exec*` system call: “forget everything in my address space and reinitialize my entire address space with stuff from a named program file.”

- The exec system call never returns: the new program executes in the calling process until it dies (exits).
  - The code from the parent program runs in the child process and controls its future. The parent program selects the child program that the child process will run (via `exec`)
  - But don’t forget to check error status from `exec*`! It returns an error to parent program if it fails.
Running a program

When a program launches, the OS creates a **process** to run it, with a main **thread** to execute the code, and a **virtual memory** to store the running program’s code and data.
**exec (original concept)**

5.3 Execution of Programs

Another major system primitive is invoked by

\[
\text{execute}(\text{file}, \arg_1, \arg_2, ..., \arg_n)
\]

which requests the system to read in and execute the program named by *file*, passing it string arguments \(\arg_1, \arg_2, ..., \arg_n\). Ordinarily, \(\arg_1\) should be the same string as *file*, so that the program may determine the name by which it was invoked. All the code and data in the process using *execute* is replaced from the file, but open files, current directory, and interprocess relationships are unaltered. Only if the call fails, for example because *file* could not be found or because its execute-permission bit was not set, does a return take place from the *execute* primitive; it resembles a “jump” machine instruction rather than a subroutine call.
### A simple program: forkexec

```c
main(int argc, char *argv[]) {
    int status;
    int rc = fork();
    if (rc < 0) {
        perror("fork failed: ");
        exit(1);
    } else if (rc == 0) {
        printf("I am a child: %d\n", getpid());
        argv++;
        execve(argv[0], argv, 0);
        /* NOTREACHED */
    } else {
        waitpid(rc, &status, 0);
        printf("Child %d exited with status %d\n", rc,
               WEXITSTATUS(status));
    }
}
```

- Always check return from syscalls and show any errors!
- Parent program running in child process
- A successful exec* never returns to calling program.
- Reap `exit status` return value from child via exit/wait.
A simple program: prog0

```c
... int main() {
    printf("Hi from %d!\n", getpid());
    exit(72);
}
```

**getpid syscall**: Get processID of current process.

**exit syscall**: Pass exit status return value to parent via exit/wait.

```
chase$ cc –o forkexec forkexec.c
chase$ cc –o prog0 prog0.c
chase$ ./forkexec prog0
I am a child: 11384.
Hi from 11384!
Child 11384 exited with status 72.
chase$
```
Kernel/user transitions for fork/exec/exit

The kernel may start and kill threads, and/or arbitrarily change the user virtual memory and/or thread context.

It does it all the time.
A trap is a system call, e.g., fork/exec/exit/wait/ or open/close/read/write or pipe/dup2.

A program is an executable file that may be launched in a process, e.g., with an exec* system call. When a program is running in a process that program controls the process. More precisely, the program controls the behavior of a thread in the process while that thread is running in user mode.

When I say that "a program invokes a system call" or "a process does a trap" I mean that a thread is running in a user program in a process, and that thread executes a trap instruction in the program, for the purpose of entering the kernel to perform a system call.

In the example:

- Exec* system call is invoked by the parent program running in the child process.
- Exec* system call "returns" into the program whose name was the first argument to exec*. That is the program I call the child program: it is now running in the child process, having replaced the parent program in the child process. After exec*, the child program begins executing in its main(). (Be sure you understand how that happened.)
What does this code do?

```c
int main(int argc, char *argv[]) {
    printf("about to run program %s\n", argv[0]);
    execve(argv[0], argv, 0);
    perror("exec failed");
}
```
The details aren’t important. The point is:

The exec system call sets up the VAS of the calling process to execute a named program.

Exec passes two arrays of strings to the new program’s main(): an array of arguments and an array of named environment variables.

It stages the argv/env arrays in the VAS before returning to user mode to start execution at main().
Unix process view: data

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

channels

- tty
- pipe
- socket

Files

Process

Thread

Program
Standard I/O descriptors

I/O channels ("file descriptors")

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.
Unix defines uniform, modular ways to combine programs to build up more complex functionality.
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.
A shell running a command

```
“cat <in >out”
```

```
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.
```
**Files: open syscall**

```c
fd = open(pathname, <options>);
write(fd, “abcdefg”, 7);
read(fd, buf, 7);
lseek(fd, offset, SEEK_SET);
close(fd);
```

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);
```
Files: open and chdir

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).

Virtual address space (Virtual Memory, VM)

Initial VAS contents **inherited** from parent on **fork**, or reset for child program on **exec**. Parent chooses environment (argv[] and envp[]) on **exec**.
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]
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.

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

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

The processes may run concurrently and are synchronized automatically.
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$
Pipes

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.
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 (until EOF) {
    read(0, buf, count);
    compute/transform data in buf;
    write(1, buf, count);
}

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.

cat | cat
(Note: try this scenario to debug pipes!)
Pipes

**Kernel-space pseudocode**
System call internals to read/write N bytes for buffer size B.

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

**read(buf, N)**

```cpp
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

```c
read(buf, N)
{
    readerMx.lock();
    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();
    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.
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.
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?
Environment variables and property lists

- The environment variable array is a **property list**.
  - The **property list** construct is very common and useful!
  - Also commonly used for configuration files.
  - It goes by various names: Java plist, Windows Registry, INI files.
- Each element of the list is a string: “NAME=VALUE”.
- The standard library has primitives to look up the VALUE corresponding to a NAME.
- In Unix systems: standard environment variables are handed down through the shell: they give programs lots of information about the environment.
- The parent specifies them to the `exec*` syscall.
Simple I/O: args and printf

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i;

    printf("arguments: %d\n", argc);
    for (i=0; i<argc; i++) {
        printf("%d: %s\n", i, argv[i]);
    }
}
```

chase$ cc –o prog1 prog1.c
chase$ ./forkexec prog1
arguments: 1
0: prog1
child 19178 exited with status 0
chase$ ./forkexec prog1 one 2 3
arguments: 4
0: prog1
1: one
2: 2
3: 3
Child 19181 exited with status 0.
Environment variables (rough)

```c
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char* argv[], char* envp[])
{
    int i;
    int count = atoi(argv[1]);

    for (i=0; i < count; i++) {
        printf("env %d: %s\n", i, envp[i]);
    }
}
```
Environment variables (rough)

chase$ cc –o env0 env0.c
chase$ ./env0

**Segmentation fault: 11**

chase$ ./env0 12

env 0: TERM_PROGRAM=Apple_Terminal
env 1: TERM=xterm-256color
env 2: SHELL=/bin/bash
env 3: TMPDIR=/var/folders/ti/ng76cpqn4zl1wrs57hldf1vm0000gn/T/
env 4: Apple_PubSub_Socket_Render=/tmp/launch-OtU5Bb/Render
env 5: TERM_PROGRAM_VERSION=309
env 6: OLDPWD=/Users/chase/c210-stuff
env 7: TERM_SESSION_ID=FFCE3A14-1D4B-4B08...
env 8: USER=chase
env 9: COMMAND_MODE=unix2003
env 10: SSH_AUTH_SOCK=/tmp/launch-W03wn2/Listeners
env 11: __CF_USER_TEXT_ENCODING=0x1F5:0:0
chase$
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[], char* envp[]) {
    int i;
    int count;

    if (argc < 2) {
        printf(stderr, "Usage: %s <count>\n", argv[0]);
        exit(1);
    }
    count = atoi(argv[1]);

    for (i=0; i < count; i++) {
        if (envp == 0) {
            printf("env %d: nothing!\n", i);
            exit(1);
        } else if (envp[i] == 0) {
            printf("env %d: null!\n", i);
            exit(1);
        } else
            printf("env %d: %s\n", i, envp[i]);
    }
}

Environment variables (safer)
Where do environment variables come from?

chase$ cc –o env env.c
chase$ ./env
chase$ ./forkexec env
Usage: env <count>
child 19195 exited with status 1
chase$ ./forkexec env 1
env 0: null!
child 19263 exited with status 1
chase$
forkexec revisited

```c
char *lala = "lalala\n";
char *nothing = 0;
...
main(int argc, char *argv[]) {
    int status;
    int rc = fork();
    if (rc < 0) {
        ...
    } else if (rc == 0) {
        argv++;
        execve(argv[0], argv, &lala);
    } else {
        ...
    }
}
```

chase$ cc -o fel forkexec-lala.c
chase$ ./fel env 1
env 0: lalala

child 19276 exited with status 0
chase$
forkexec revisited again

... main(int argc, char *argv[],
       char *envp[]) {
  int status;
  int rc = fork();
  if (rc < 0) {
    ...
  } else if (rc == 0) {
    argv++;
    execve(argv[0], argv, envp);
  } else {
    ...
  }
}
How about this?

chase$ ./fe fe fe fe fe fe fe fe fe fe fe fe fe env 3

<???>
How about this?

chase$ .fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe env 3
env 0: TERM_PROGRAM=Apple_Terminal
env 1: TERM=xterm-256color
env 2: SHELL=/bin/bash
child 19303 exited with status 0
child 19302 exited with status 0
child 19301 exited with status 0
child 19300 exited with status 0
child 19299 exited with status 0
child 19298 exited with status 0
child 19297 exited with status 0
child 19296 exited with status 0
child 19295 exited with status 0
child 19294 exited with status 0
child 19293 exited with status 0
child 19292 exited with status 0
chase$

It is easy for children to inherit environment variables from their parents.

Exec* enables the parent to control the environment variables and arguments passed to the children.

The child process passes the environment variables “to itself” but the parent program controls it.
**Isolation**

We need protection domains and protected contexts (“sandboxes”), even on single-user systems like your smartphone. There are various dimensions of isolation for protected contexts (e.g., processes):

- **Fault isolation.** One app or app instance (context or process) can fail independently of others. If it fails, the OS can kill the process and reclaim all of its memory, etc.

- **Performance isolation.** The OS manages resources (“metal and glass”: computing power, memory, disk space, I/O throughput capacity, network capacity, etc.). Each instance needs the “right amount” of resources to run properly. The OS prevents apps from impacting the performance of other apps. E.g., the OS can prevent a program with an endless loop from monopolizing the processor or “taking over the machine”. (How?)

- **Security.** An app may contain malware that tries to corrupt the system, steal data, or otherwise compromise the integrity of the system. The OS uses protected contexts and a reference monitor to check and authorize all accesses to data or objects outside of the context.
VM and files: the story so far

**Files on “disk”**

- Program
- File tree
  - /usr
  - /home
    - jeremy
    - chris
  - local
  - bin

**Process** (running program)
- Thread
- Register context for main thread of process
- And its stack

**Segments (regions) in Virtual Address Space**
- globals
- text
- heap
- stack
- Anonymous segments (zero-fill)
- Memory-mapped sections of program file

**File system calls** (e.g., open/read/write)
slowcat

- Ctrl-z, fg, bg
- Sleep(3)
- Speed matching for pipes
- File refcounts
Unix, looking backward: UI+IPC

- Conceived around keystrokes and byte streams
  - User-visible environment is centered on a text-based command shell.

- Limited view of how programs interact
  - files: byte streams in a shared name space
  - pipes: byte streams between pairs of sibling processes
X Windows (1985)

Big change: GUI.
1. Windows
2. Window server
3. App events
4. Widget toolkit
Files: hierarchical name space

- root directory
- mount point
- applications etc.
- user home directory
- external media volume or network storage
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);
}
void forkme(int s, char* prog, char* argv[])
{
    int status;
    int rc;
    rc = fork();
    if (rc < 0) {
        perror("fork failed: ");
        exit(1);
    } else if (rc == 0) {
        printf("I am a child: %d\n", getpid());
        dup2(s, 0);
        dup2(s, 1);
        dup2(s, 2);
        execve(prog, argv, 0);
        /* NOTREACHED */
        perror("execve failed");
        exit(1);
    } else {
        /* Comment out this waitpid to let the server handle many concurrent requests. */
        if (waitpid(rc, &status, 0) == -1)
            perror("waitpid failed");
        else
            printf("Child %d exited with status %d\n", rc, WEXITSTATUS(status));
    }
}