CS162
Operating Systems and Systems Programming
Lecture 4

Systems Programming
Processes and Communication

Professor Natacha Crooks & Matei Zaharia

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Admistratrivia

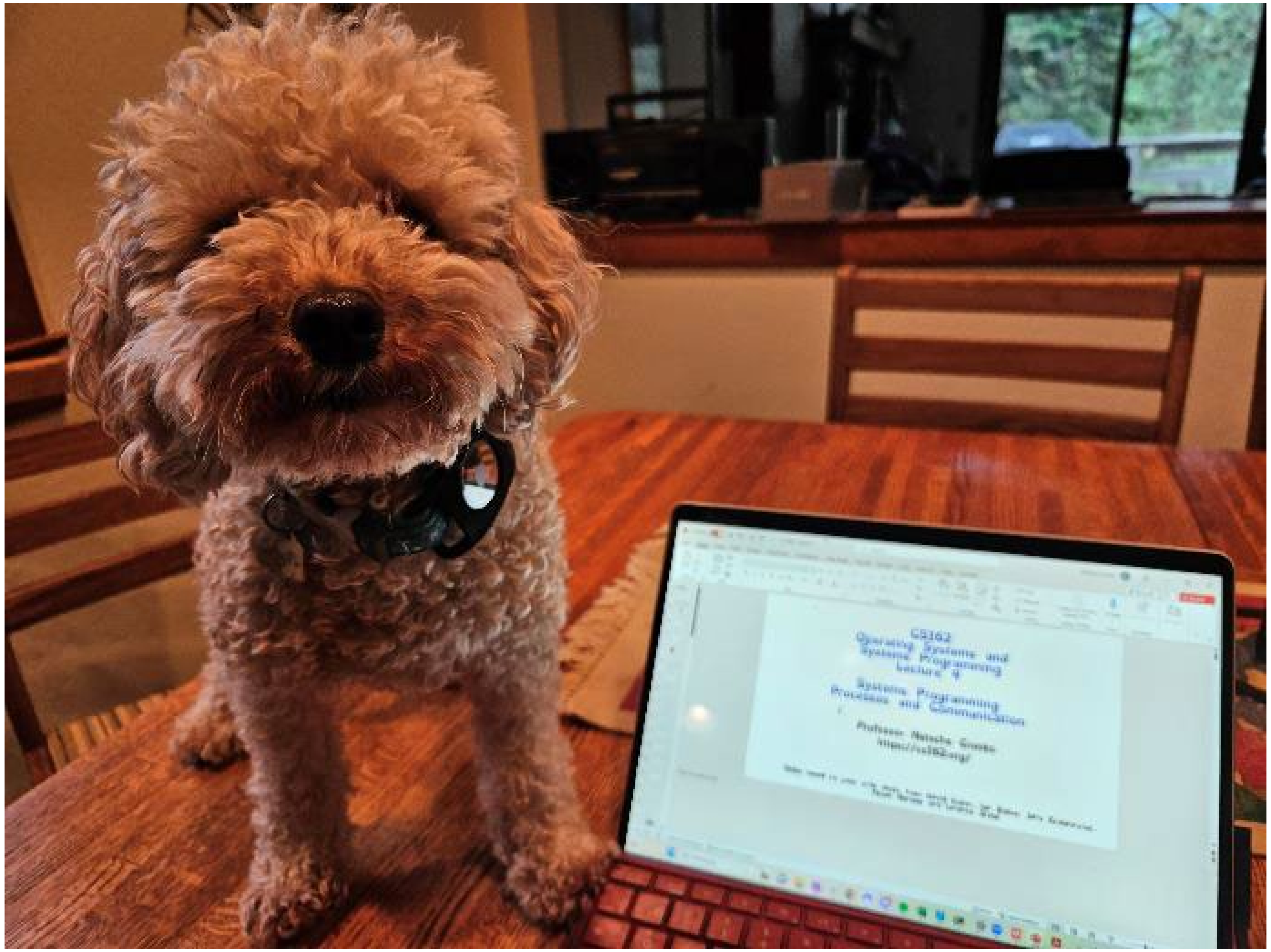
Admistratrivia

Early Drop Deadline is tomorrow!

Homework 0 is due today!

Project 0 has been released**, Due 09/02.**
This is an **individual assignment**, but future projects will be in assigned teams.

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 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
(setting kernel mode)

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

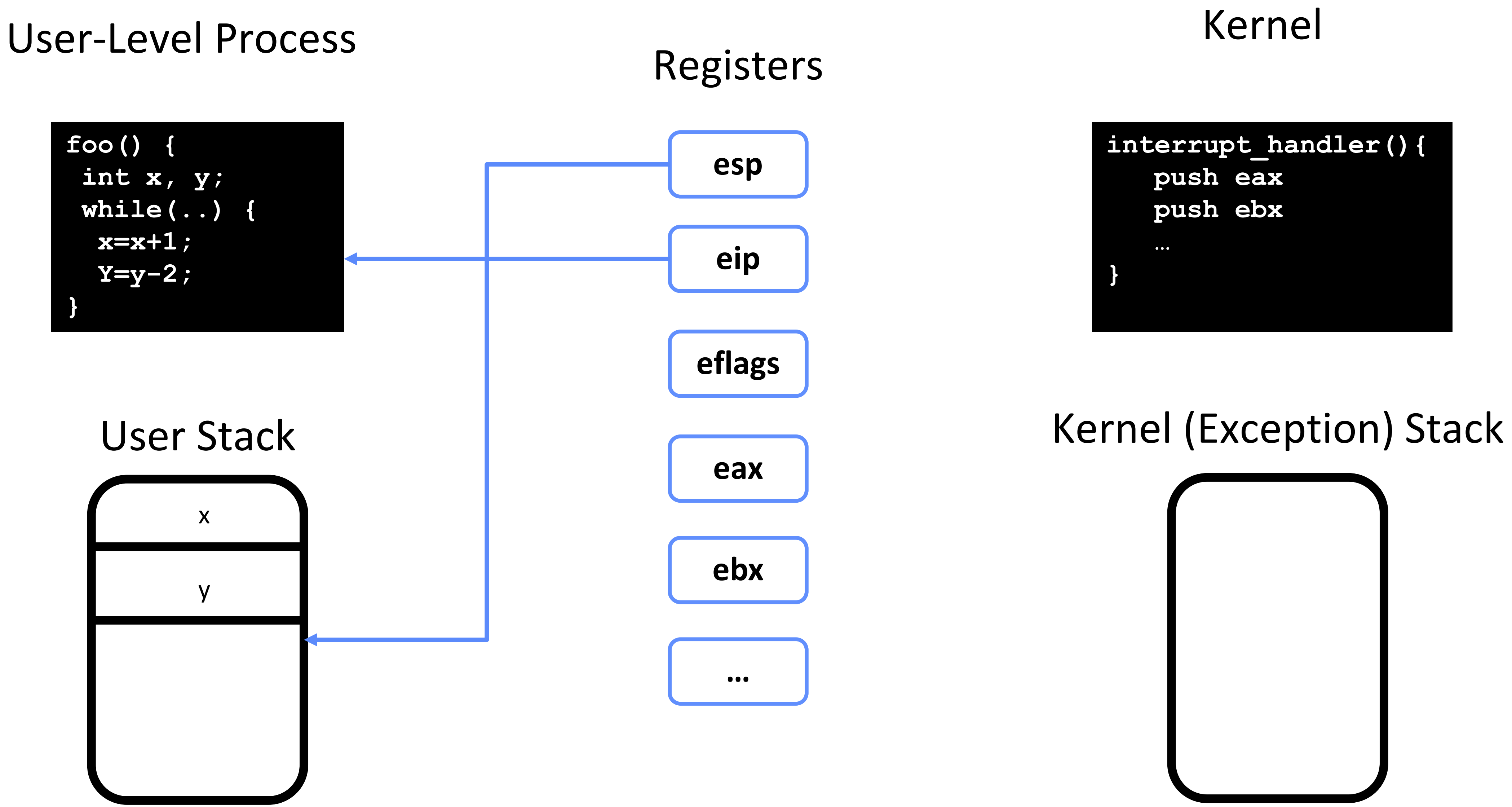
2) CPU saves Recovery State, sets kernel mode and switch to Kernel Stack

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

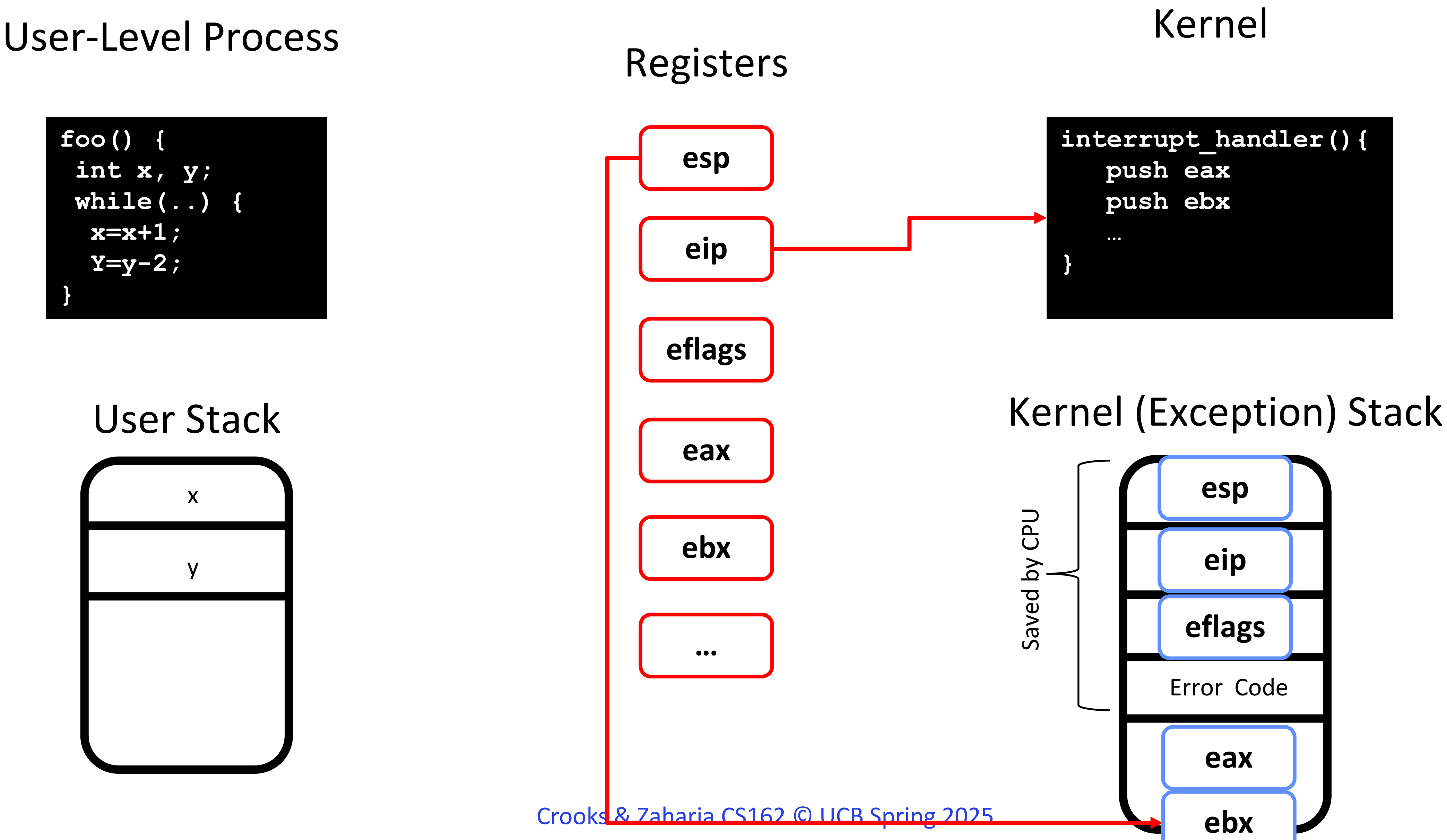
4) Kernel runs interrupt handler

5) Restore user program

Recall: User Stack/Kernel Stack (User Mode)



Recall: User Stack/Kernel Stack (Kernel Mode)



Three "Prongs" for the Class

Understanding OS principles

System Programming

Map Concepts to Real Code

Goals for Today

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

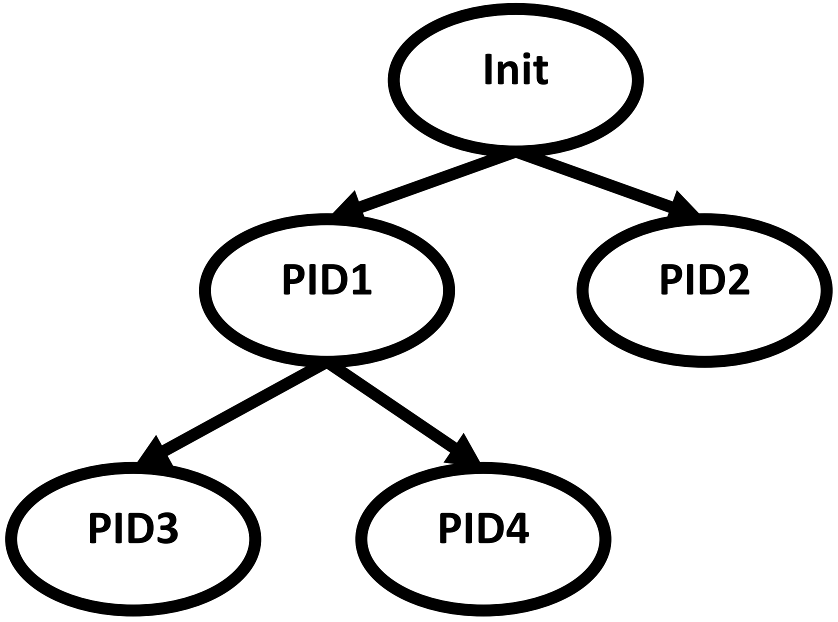
Keeping it in the family

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 ?        Sl     0:00 /init
    7 ?        Ss     0:00 /init
    8 ?        S      0:01  \_ /init
    9 pts/0    Ss     0:00      \_ -bash
  563 pts/0    S+     0:00      |   \_ tmux
  565 ?        Ss     0:04      \_ tmux
  566 pts/1    Ss     0:00          \_ -bash
 1054 pts/1    R+     0:00          |   \_ ps -x --forest
  883 pts/2    Ss     0:00          \_ -bash
 1052 pts/2    S+     0:00              \_ ./parent
 1053 pts/2    R+     0:01                  \_
```

Children in the Wild (well, in the Kernel)

```
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)
};
```

Xv6 Kernel (proc.h)

Children in the Wild (well, in the Kernel)

```
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?

In Linux `task_struct` defined in `<linux/sched.h>`

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

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

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

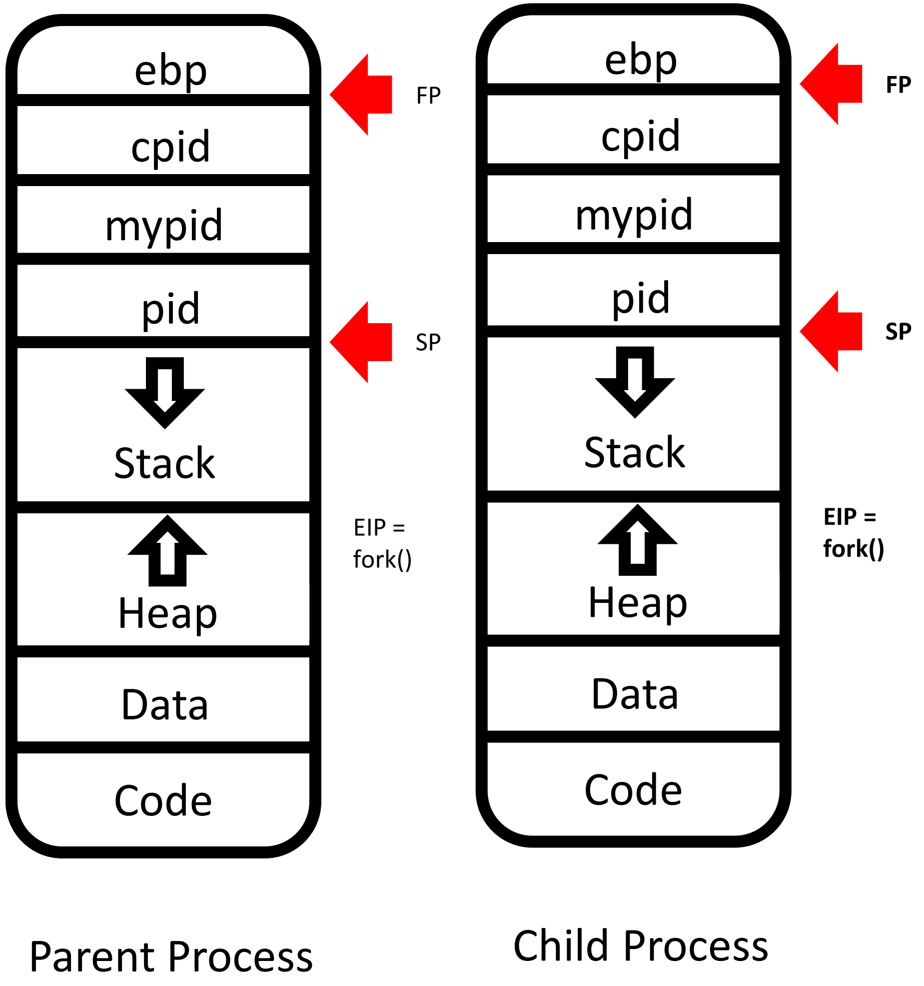
Using Fork

```c
#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");
  }
}
```

What do you think this code does?

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");
  }
}
```



Forking: Where to restart?

```
#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");
  }
}
```

Where does the child process start executing?

1) **From int main?**
2) **If (cpid > 0)?**

Remember! Instruction pointer is pointing to the same fork() instruction

Forked Process: Who am I?

```
#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");
  }
}
```

How do you determine whether you are the parent or the child?

Fork() returns PID of child to parent

If returns 0 then are child

Fork Ordering

```
int i;
pid_t cpid = fork();
if (cpid > 0) {
  for (i = 0; i < 10; i++) {
    printf("Parent: %d\n", i);
    // sleep(1);
  }
} else if (cpid == 0) {
  for (i = 0; i > -10; i--) {
    printf("Child: %d\n", i);
    // sleep(1);
  }
}
```

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 idenfies 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:

Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. 2019. A `fork()` in the road. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3317550.3321435

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: Atlidakis 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 havoc 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.

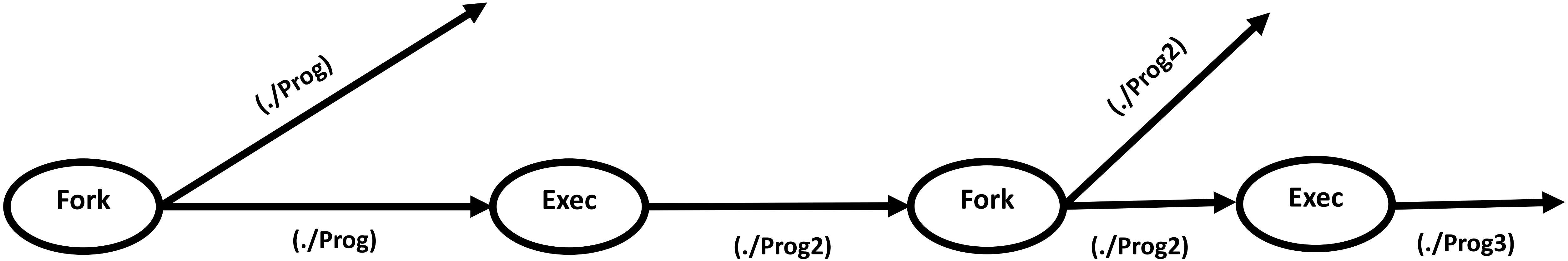
Isn't this wasteful?

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()

```c
int status;
pid_t tcpid;
…
cpid = fork();
if (cpid > 0) {  /* Parent Process */
  mypid = getpid();
  printf("[%d] parent of [%d]\n", mypid, cpid);
  tcpid = wait(&status);
  printf("[%d] Parent says bye %d(%d)\n",
          mypid, tcpid, status);
} else if (cpid == 0) { /* Child Process */
  mypid = getpid();
  printf("[%d] child\n", mypid);
  printf("[%d] Child says bye %d \n",
          mypid);
  exit(42);
}
…
```

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?

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

void signal_callback_handler(int signum) {
  printf("Caught signal!\n");
  exit(1);
}
int main() {
  struct sigaction sa;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sa.sa_handler = signal_callback_handler;
  sigaction(SIGINT, &sa, NULL);
  while (1) {}
}
```

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)

Ctr + 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
```

```
NAME
       signal - overview of signals

DESCRIPTION
       Linux supports both POSIX reliable signals (hereinafter "standard signals") and POSIX real-time signals.

   Signal dispositions
       Each signal has a current disposition, which determines how the process behaves when it is delivered the signal.

       The entries in the "Action" column of the table below specify the default disposition for each signal, as follows:

       Term   Default action is to terminate the process.

       Ign    Default action is to ignore the signal.

       Core   Default action is to terminate the process and dump core (see core(5)).

       Stop   Default action is to stop the process.

       Cont   Default action is to continue the process if it is currently stopped.

       A  process  can change the disposition of a signal using sigaction(2) or signal(2).  (The latter is less portable when establishing a signal handler; see signal(2) for details.)
       Using these system calls, a process can elect one of the following behaviors to occur on delivery of the signal: perform the default action; ignore the signal; or catch the sig-
       nal with a signal handler, a programmer-defined function that is automatically invoked when the signal is delivered.
```

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

Goal 2: Input/Output in Linux

UNIX offers the same IO interface for:

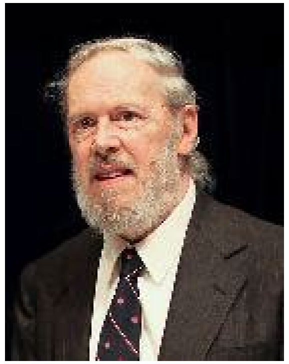
All device Input/Output — **Printers**, **Mouse**

Reading/Writing Files — **Disk**

Interprocess communication — **Pipes**, **Socket**

Everything is a file!

Radical in 1974!



The UNIX Time-Sharing 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

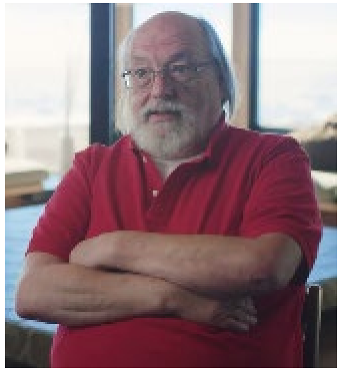
1. Introduction

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 [l] 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, 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,



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)

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNI

// 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)
};
```

In Linux struct fdtable defined in <include/kernel/fdtable.h>

Xv6 Kernel (proc.h)

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

```
#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);
```

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

Close

Closes a file descriptor, so that it no longer refers to any file and may be reused

Manipulating FDs (2)

Read data from open file using file descriptor:

```
ssize_t read (int filedes, void *buffer, size_t maxsize)
```

Write data to open file using file descriptor

```
ssize_t write (int filedes, const void *buffer, size_t size)
```

Reposition file offset within kernel

```
off_t lseek (int filedes, off_t offset, int whence)
```

Example

```
char buffer1[100];
char buffer2[100];
```

O: STDIN
1: STDOUT
2: STDERR

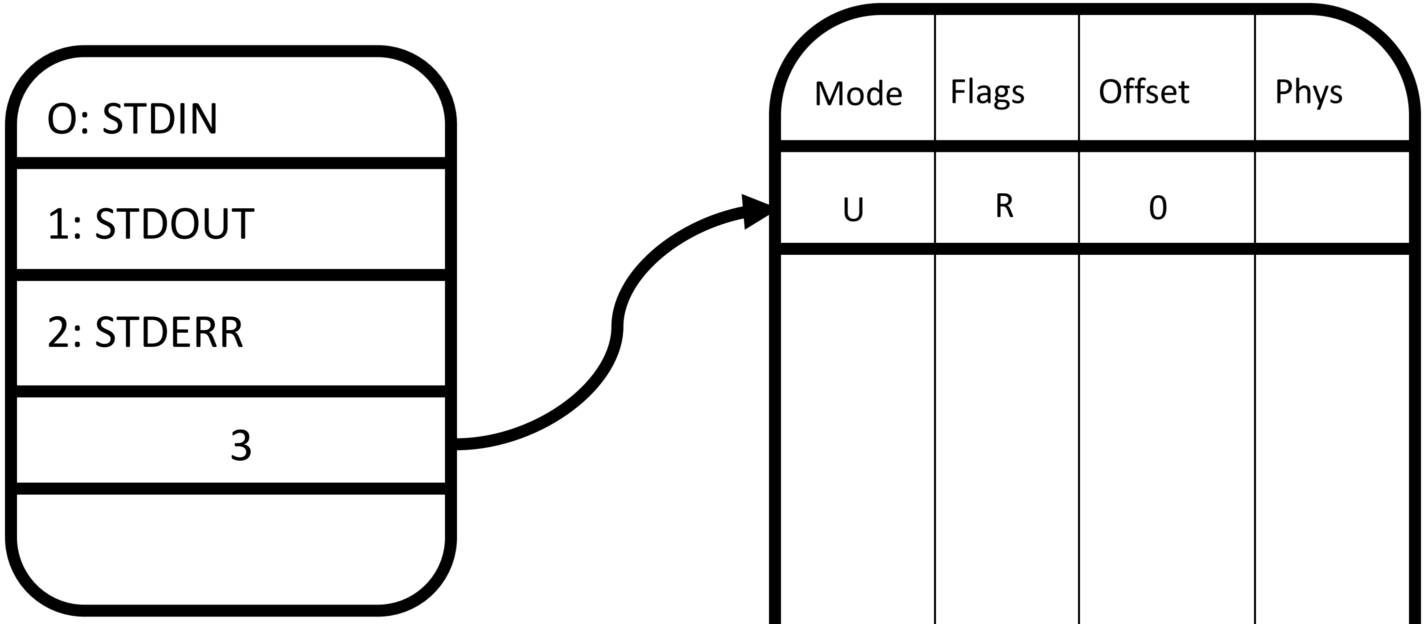
Per-Process File Descriptor Table

Mode	Flags	Offset	Phys

Global Open File Description Table

Example

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

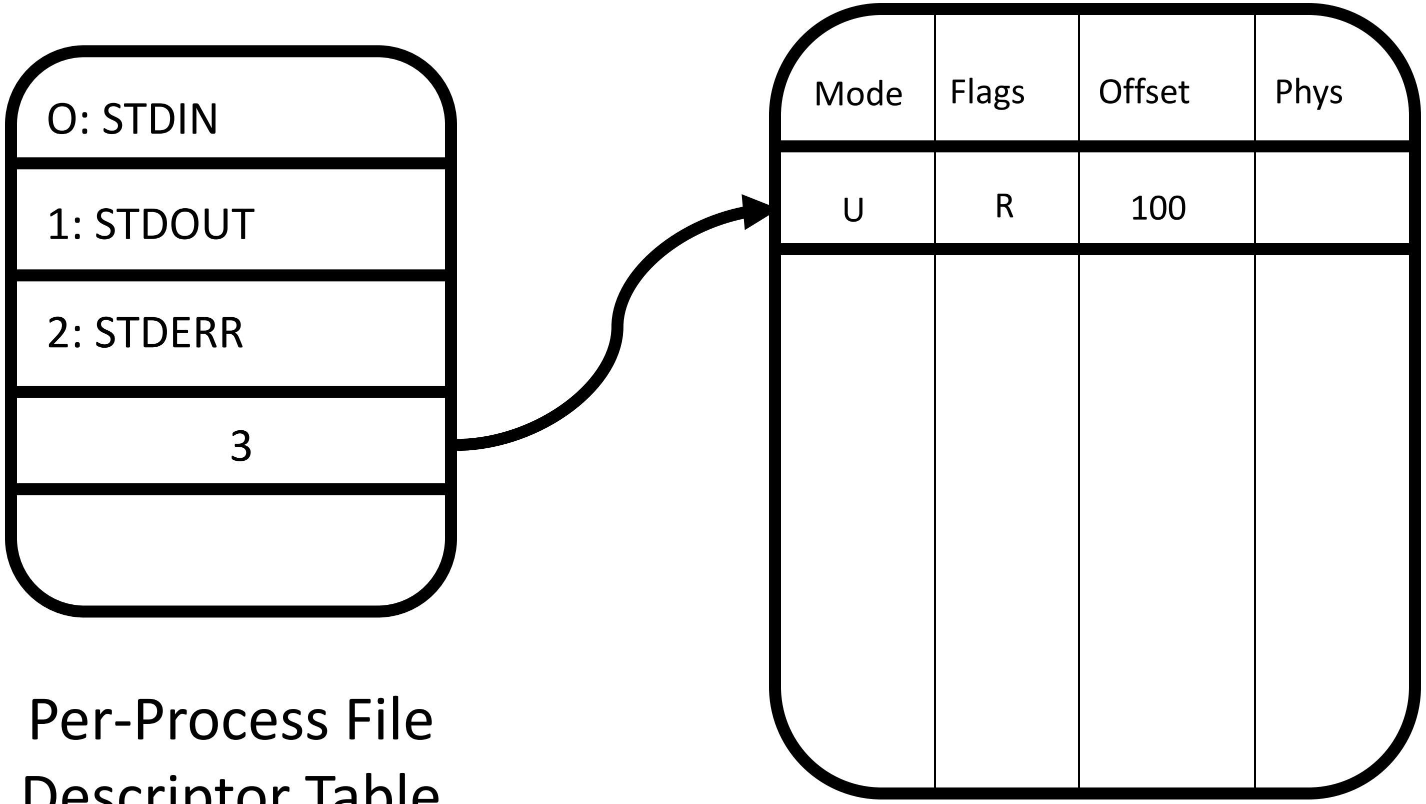


Per-Process File Descriptor Table

Global Open File Description Table

Example

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

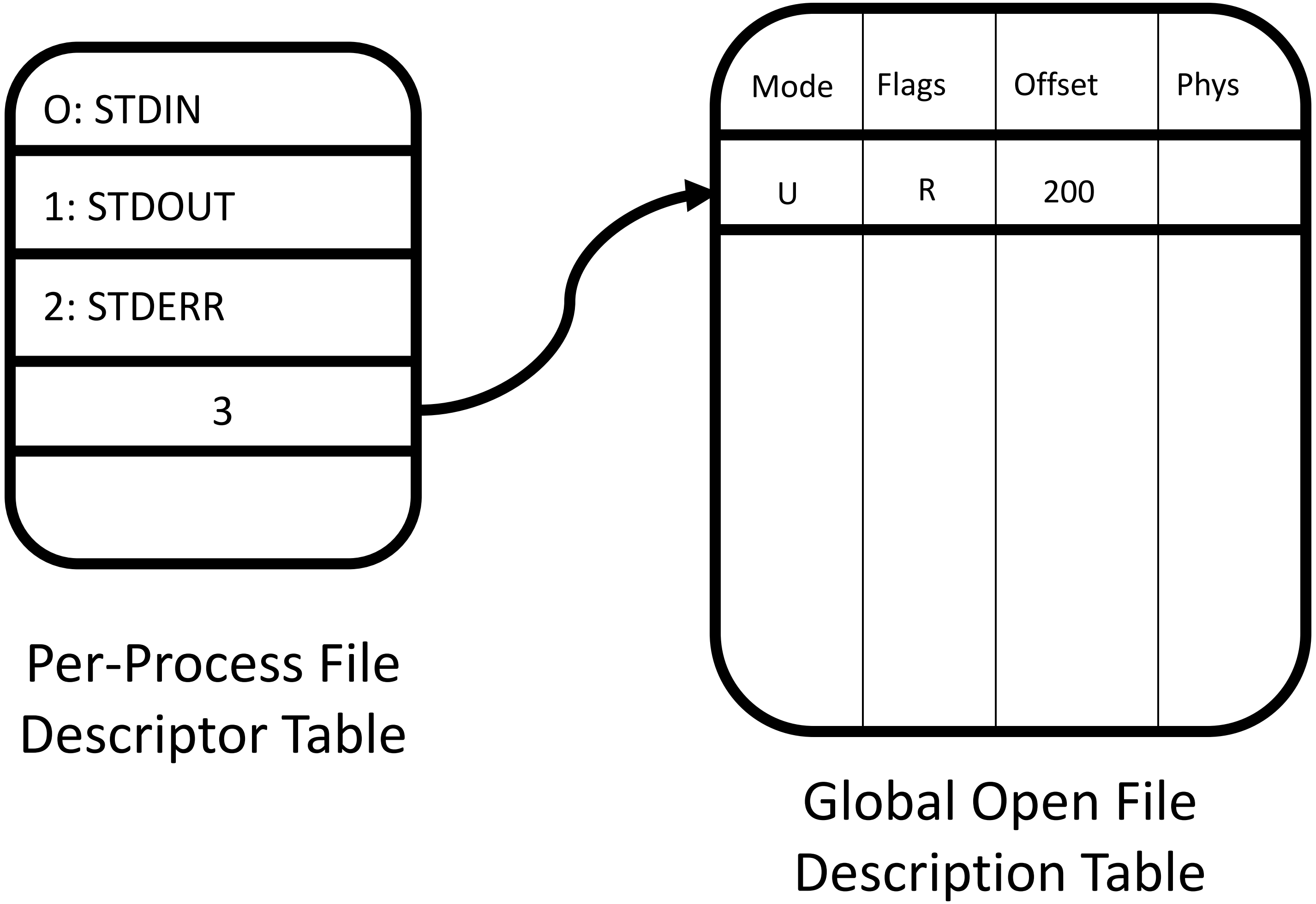


Per-Process File Descriptor Table

Global Open File Description Table

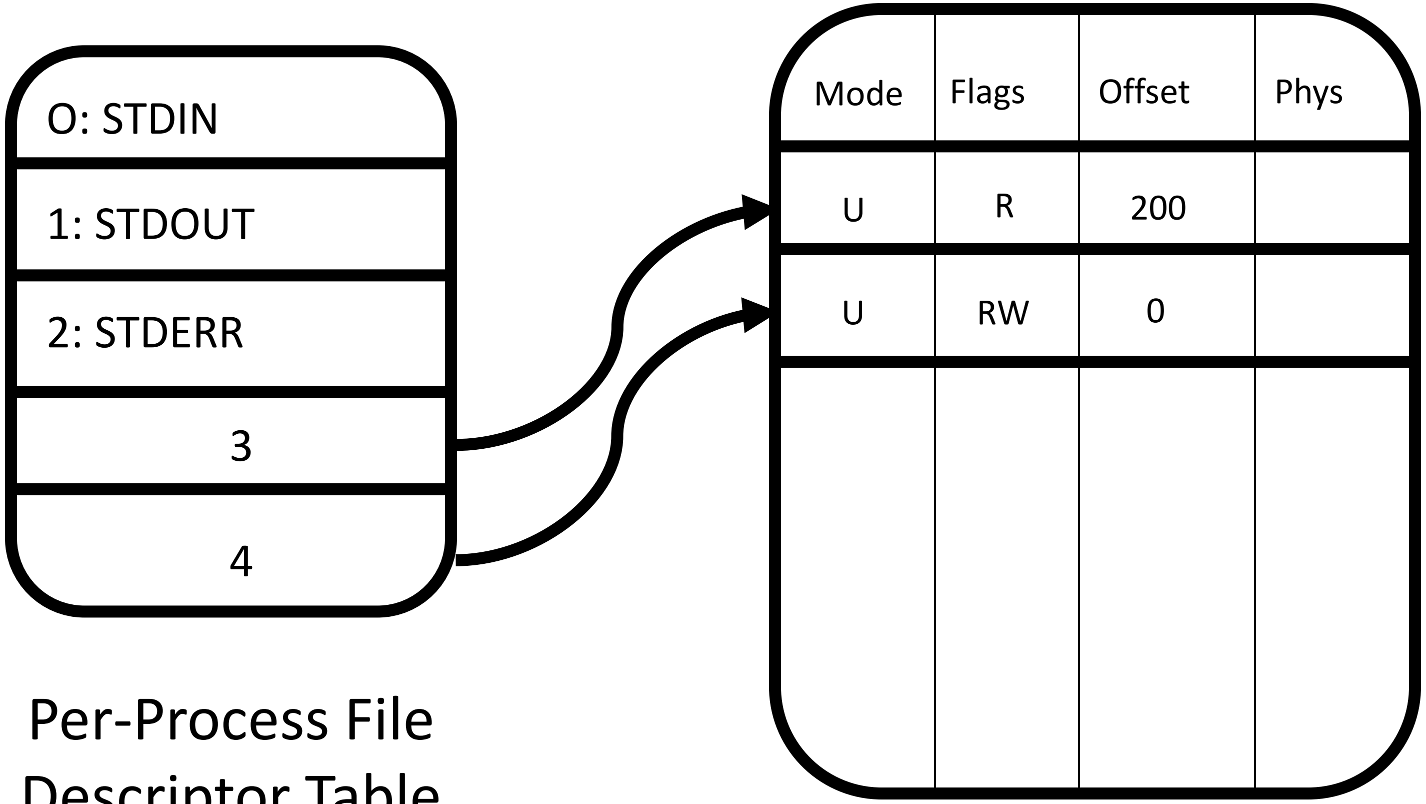
Example

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



Example

```
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);
```

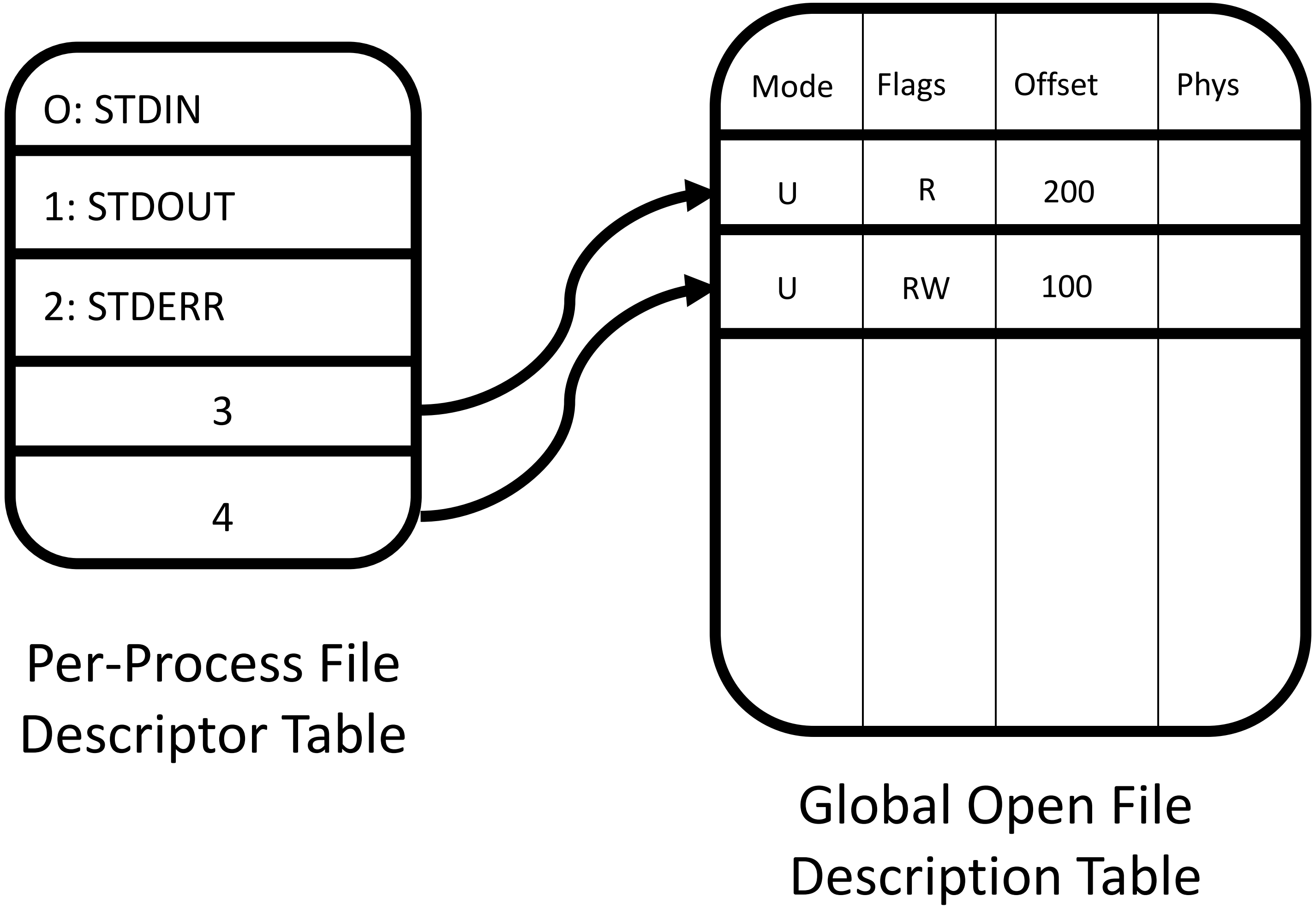


Per-Process File Descriptor Table

Global Open File Description Table

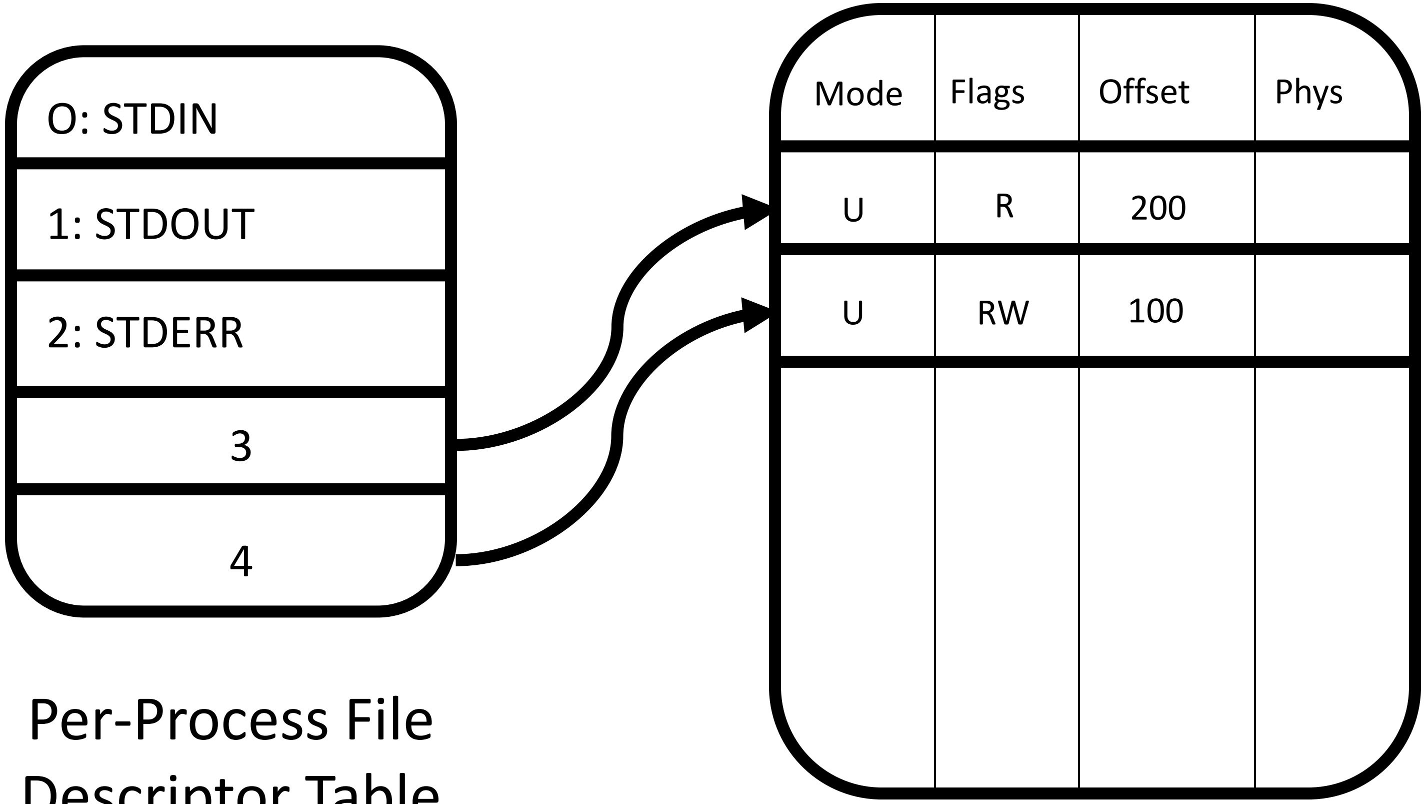
Example

```
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);
```



Example

```
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);
```



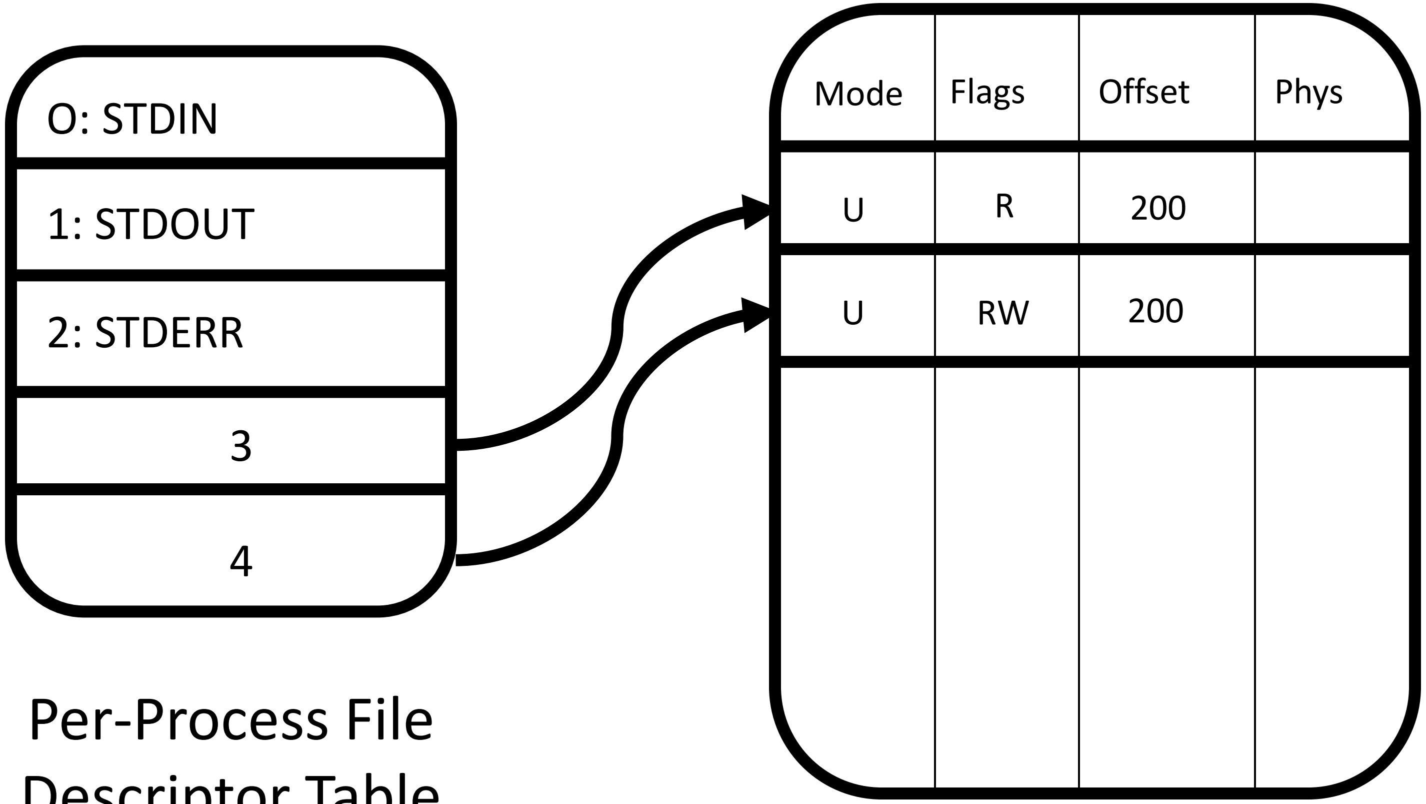
Per-Process File Descriptor Table

Global Open File Description Table

Type man 2 write in terminal. What do you think?

Example

```
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);
```

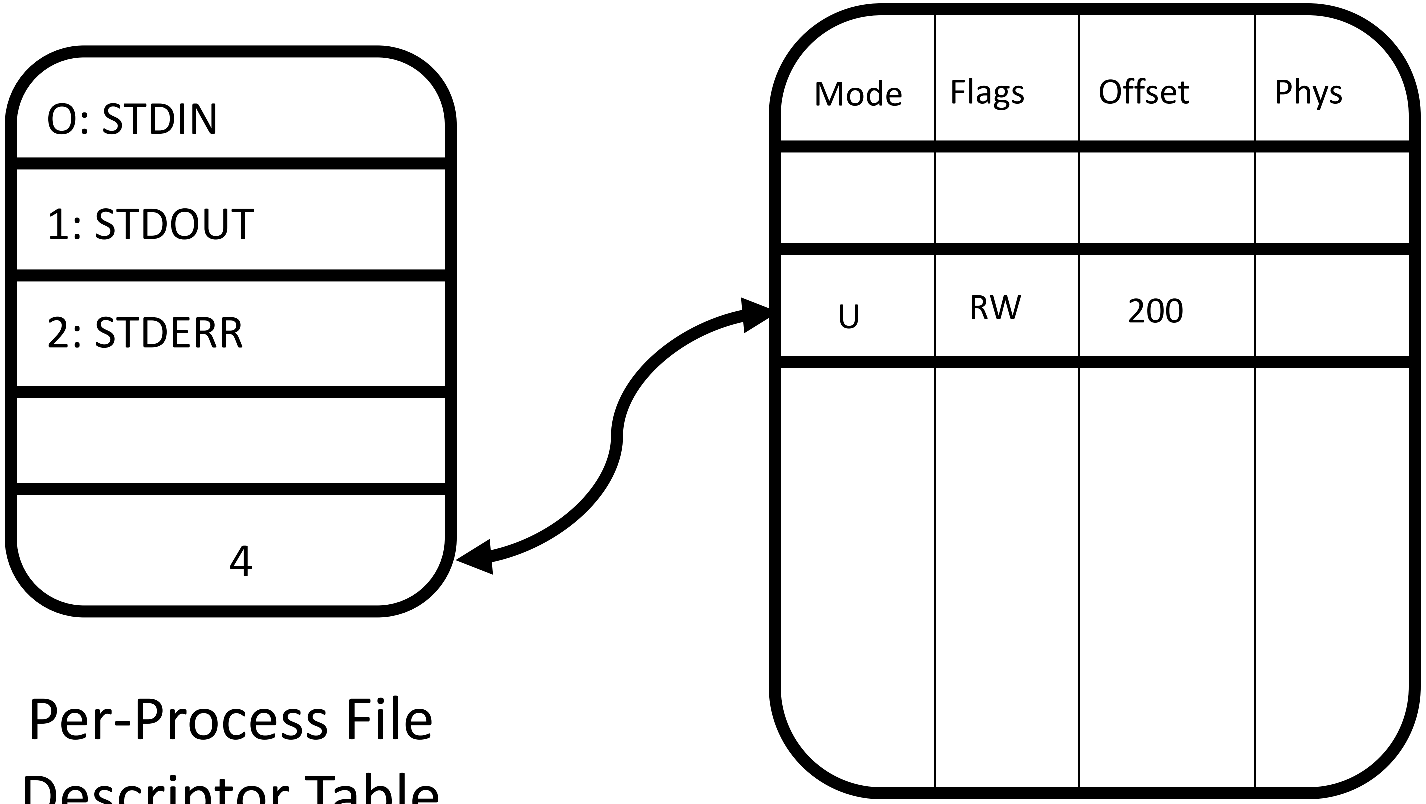


Per-Process File Descriptor Table

Global Open File Description Table

Example

```
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)
```

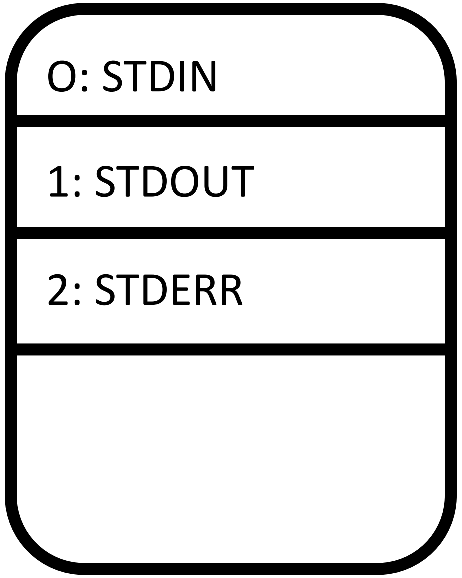


Per-Process File Descriptor Table

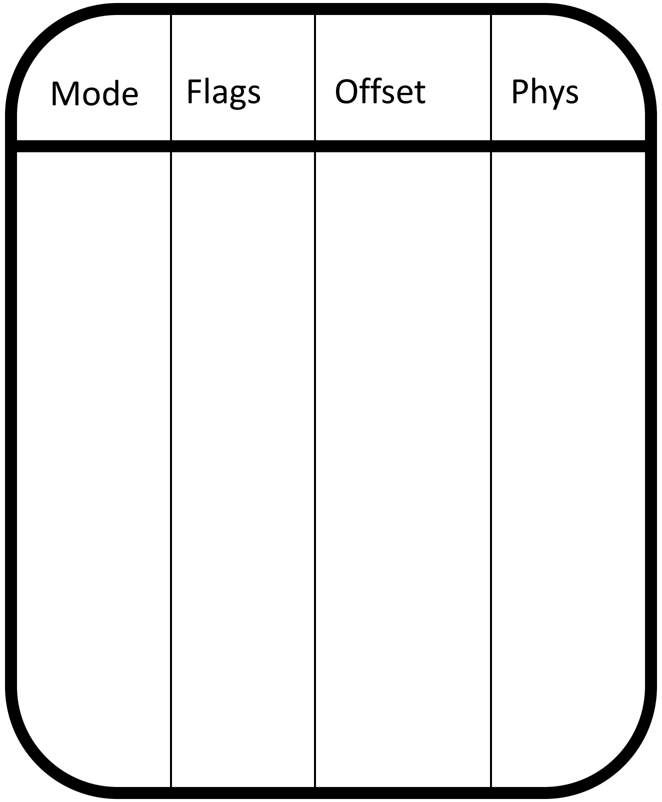
Global Open File Description Table

Example

```
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)
```



Per-Process File Descriptor Table



Global Open File Description Table