CS162
Operating Systems and Systems Programming
Lecture 5

File Descriptors (Continued)
OS Library
Threads and the Thread API

Professor Natacha Crooks
https://cs162.org/

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

Recall: 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!

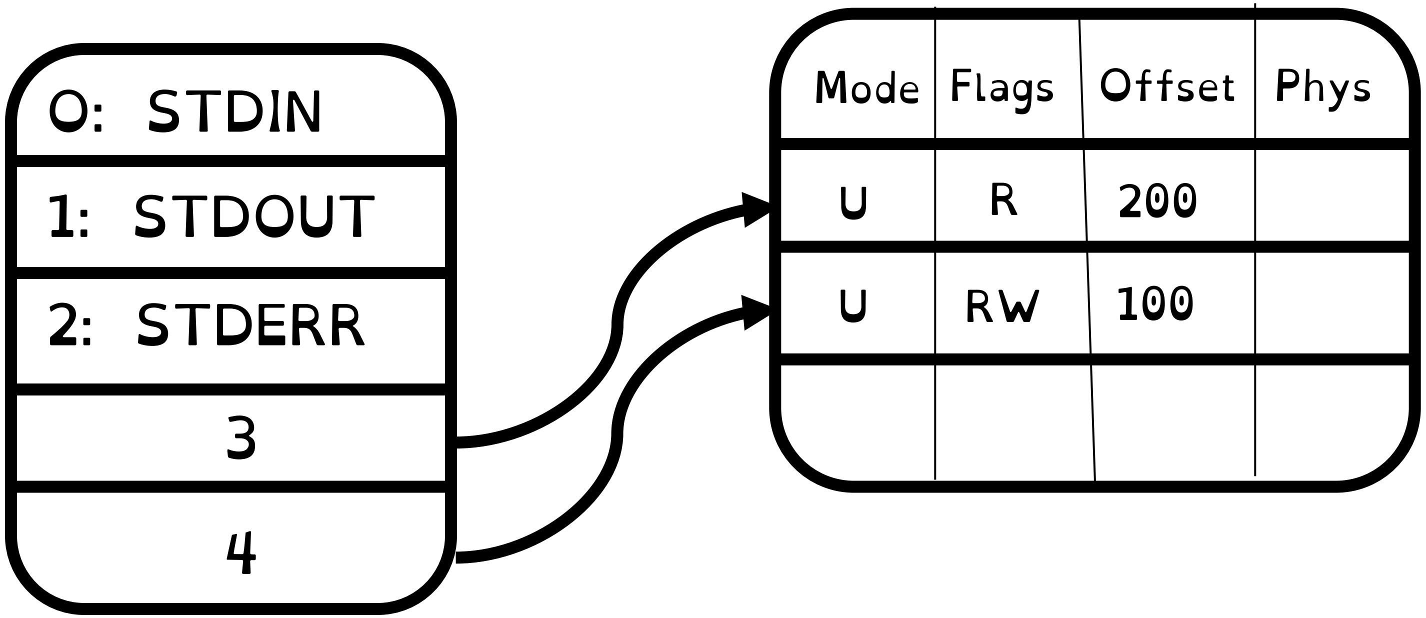
Goals For Today

- File descriptors (Continued)
- How does the OS library make it easier to program?
- What are threads and why are they useful?
- How are they implemented?
- How to write a program using threads?

Recall: File Descriptors

File descriptors index into
a per-process file descriptor table

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



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

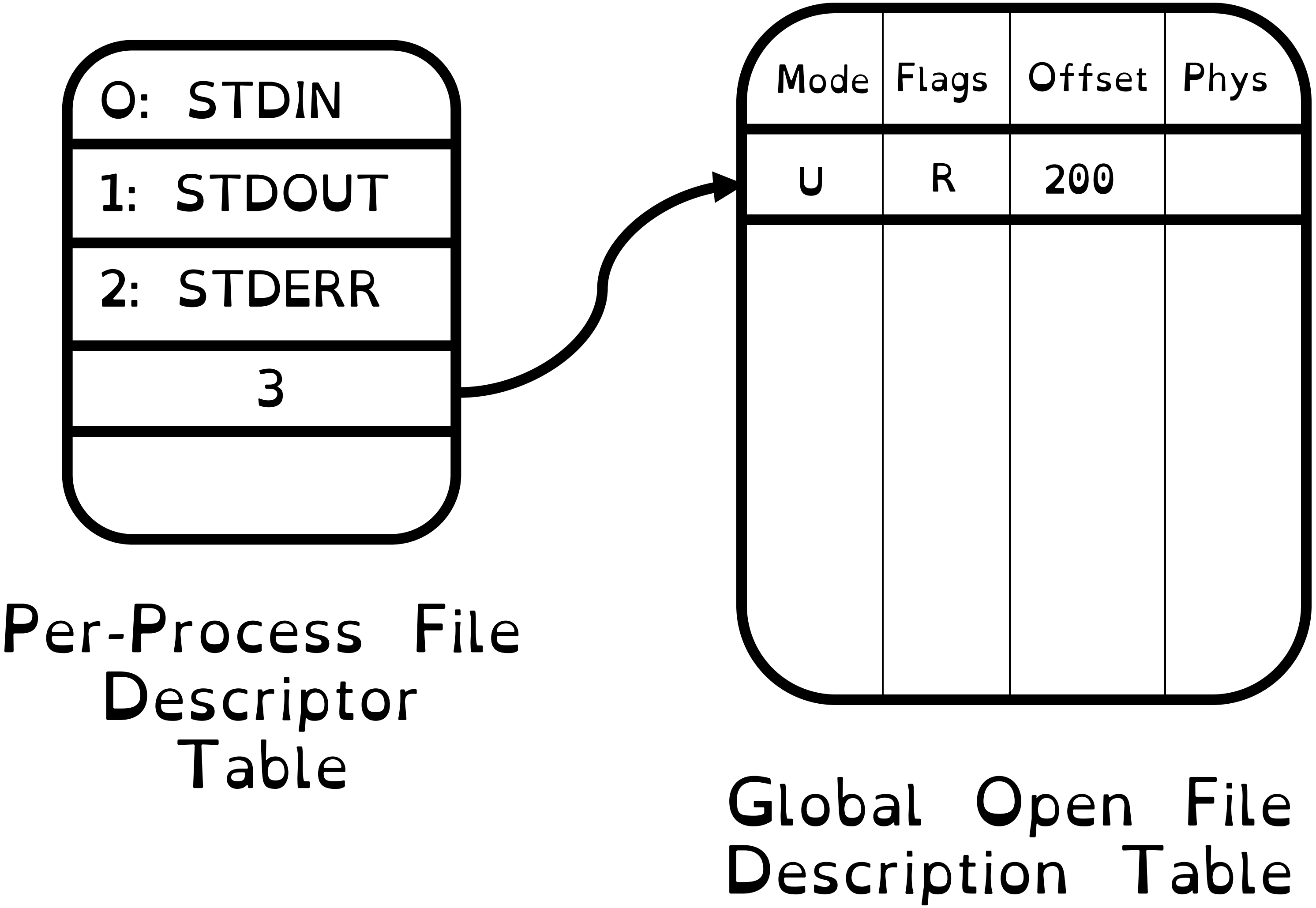
```
ssize_t write (int filedes, const void *buffer, size_t size)
```

Reposition file offset within kernel

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

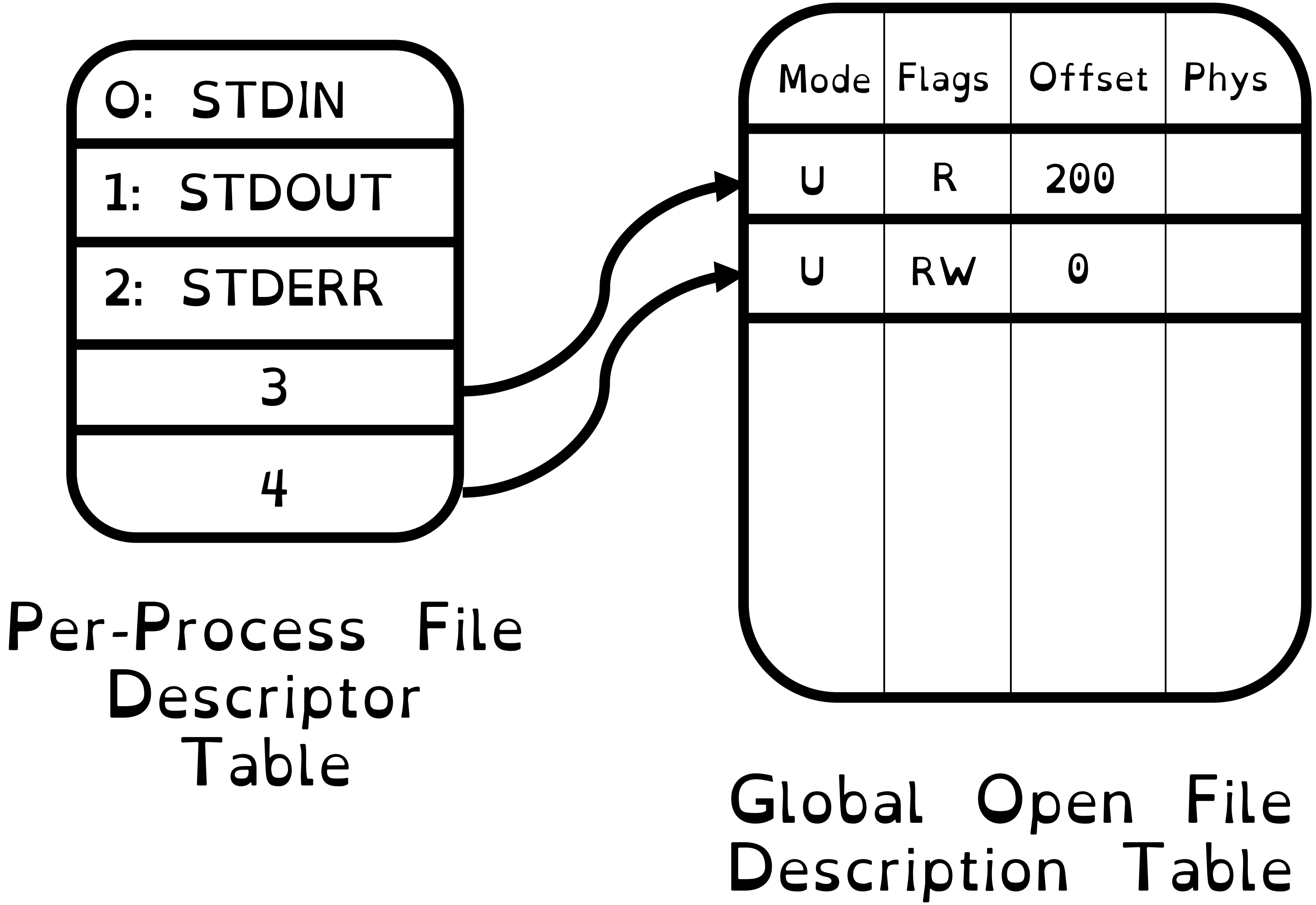
Example

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



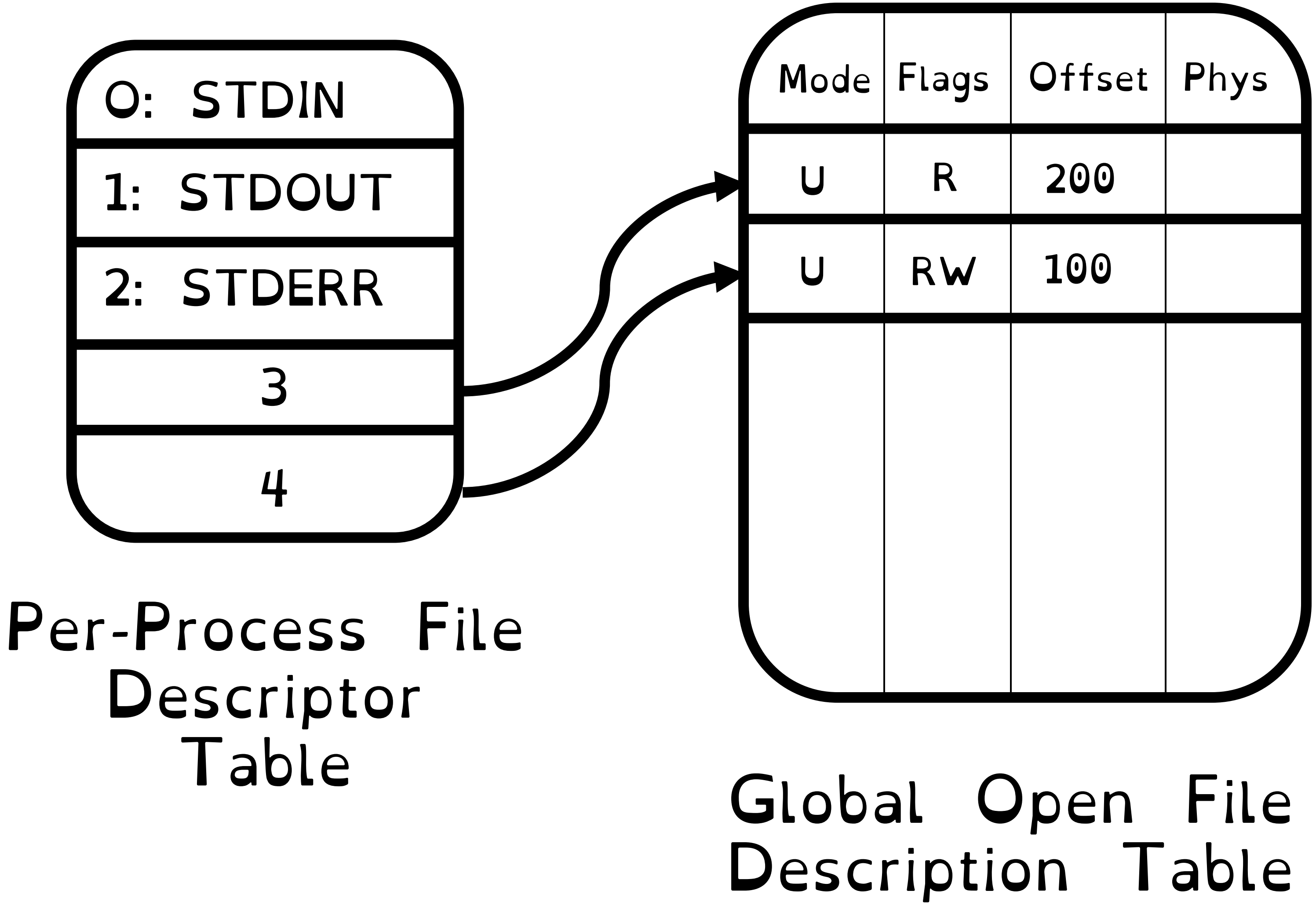
Example

```
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);
```



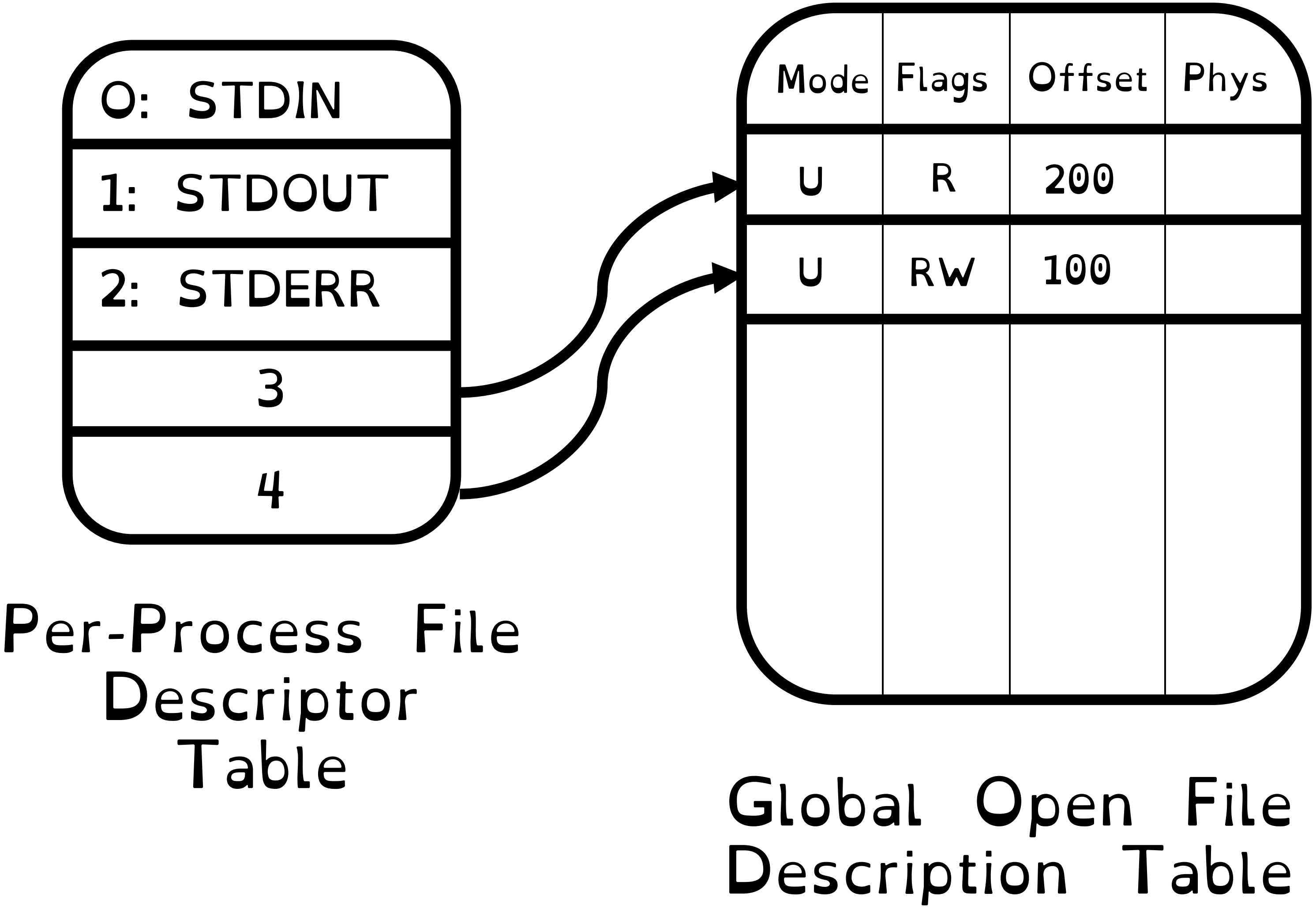
Example

```
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

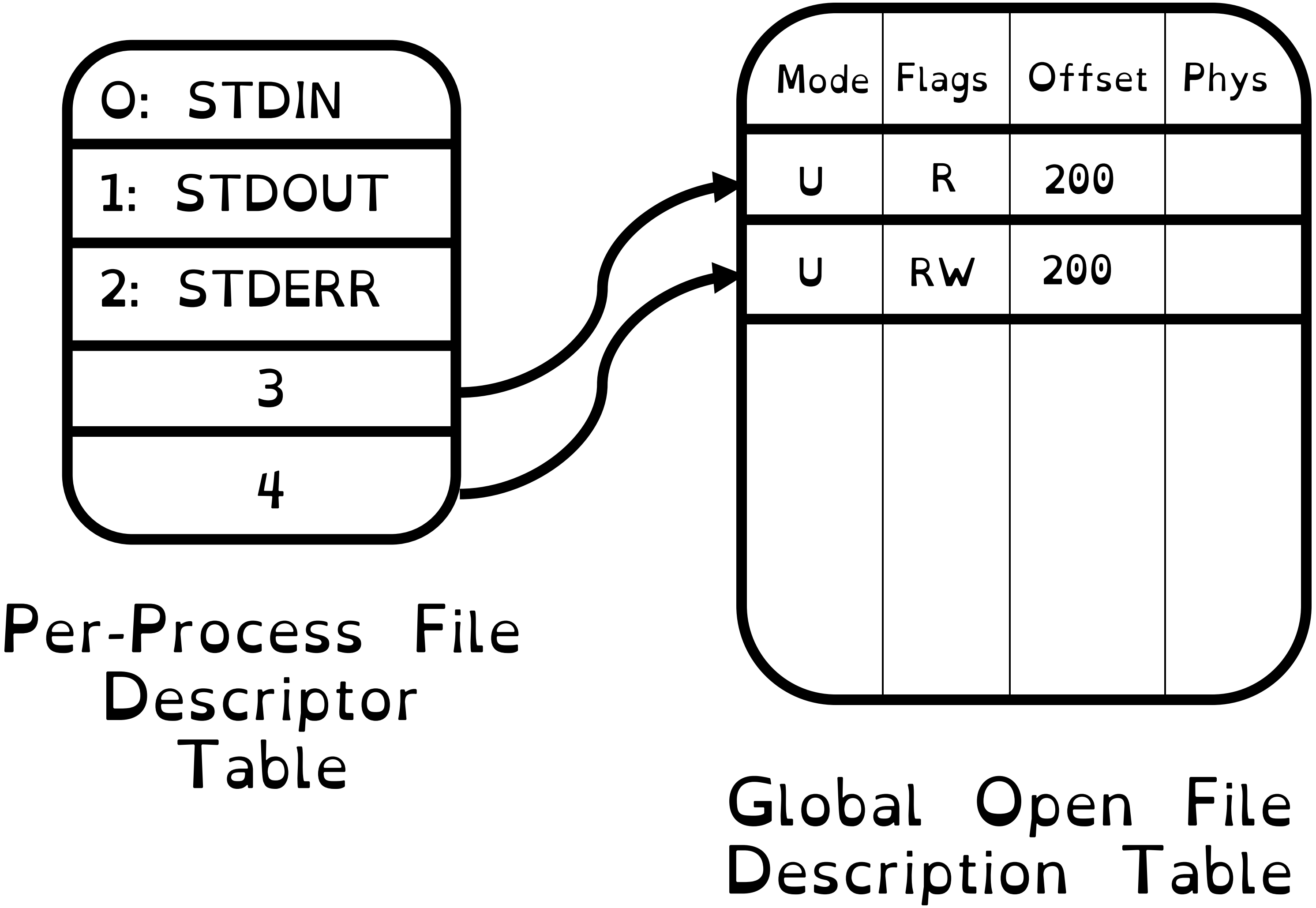
```
char buffer1[100];
char buffer2[100];
int fd = open("foo.txt",
O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

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



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

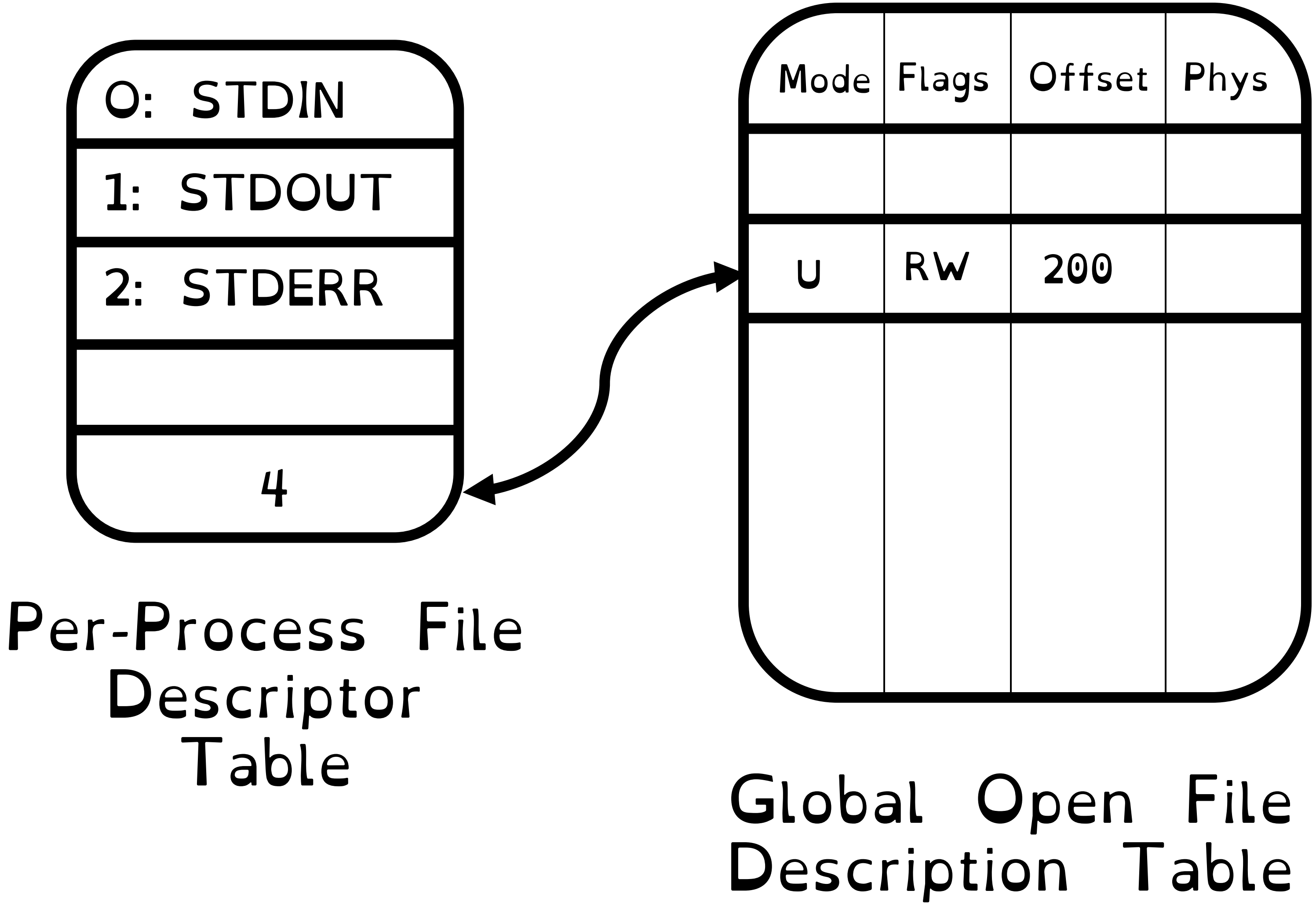
Example

```
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);
```



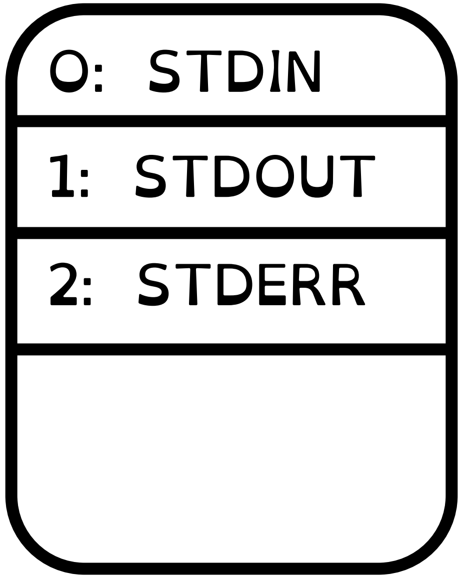
Example

```
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)
```

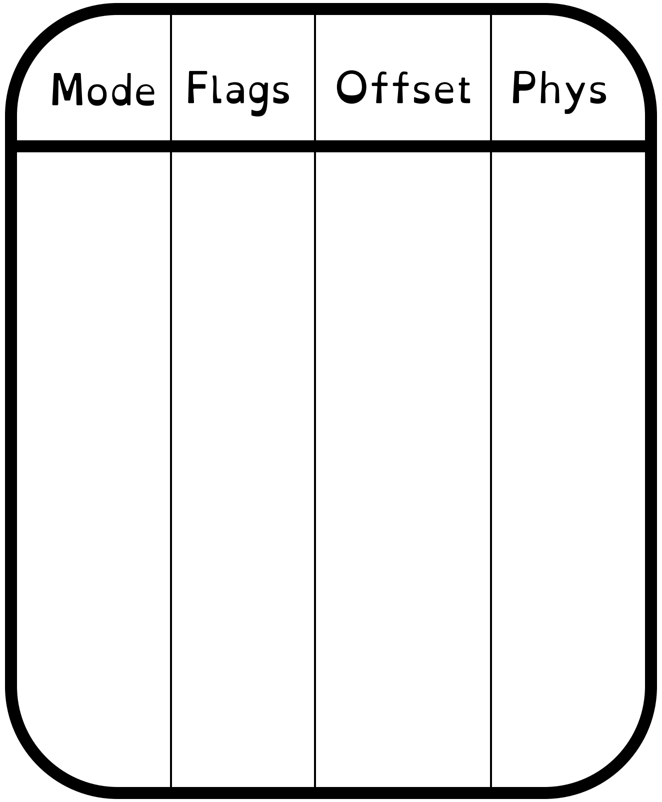


Example

```
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)
```



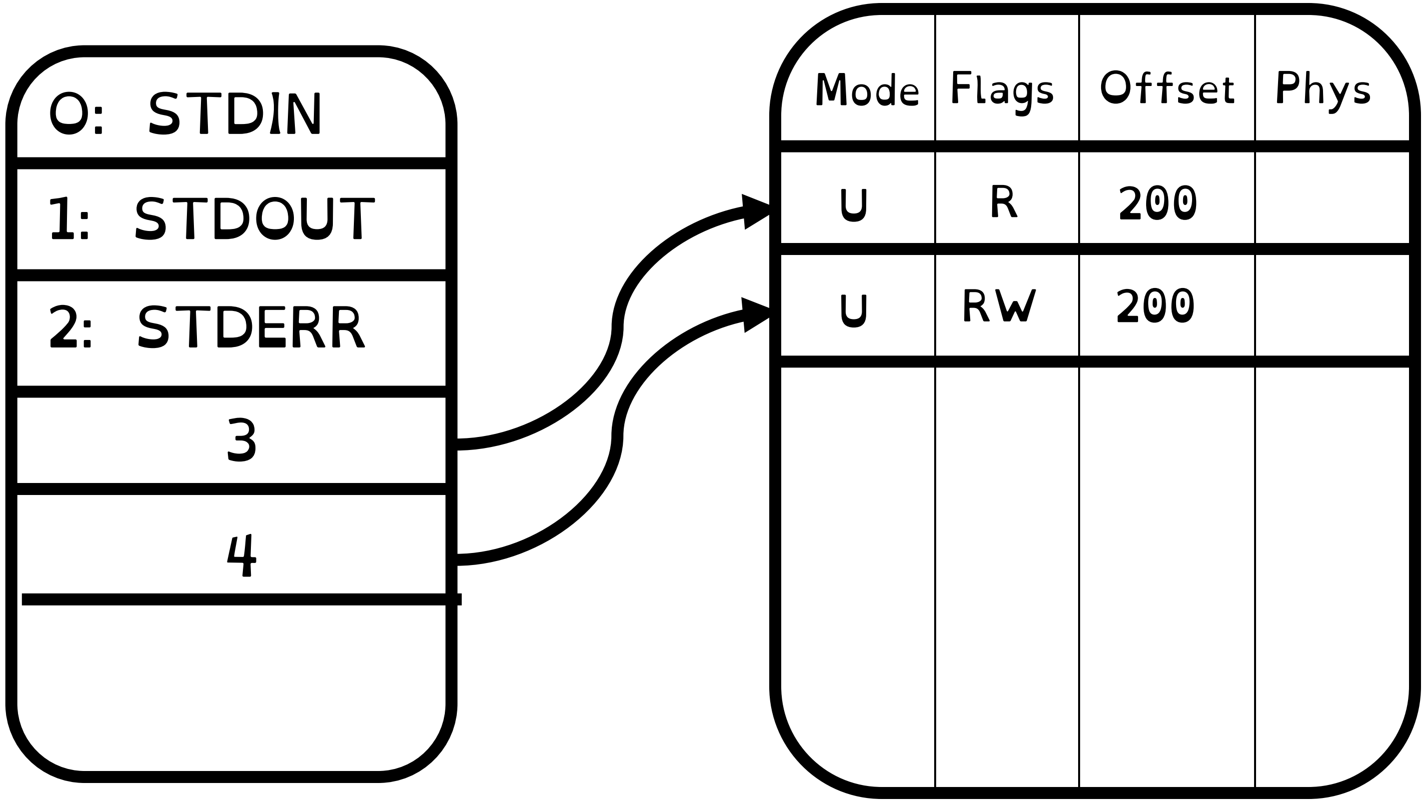
Per-Process File Descriptor Table



Global Open File Description Table

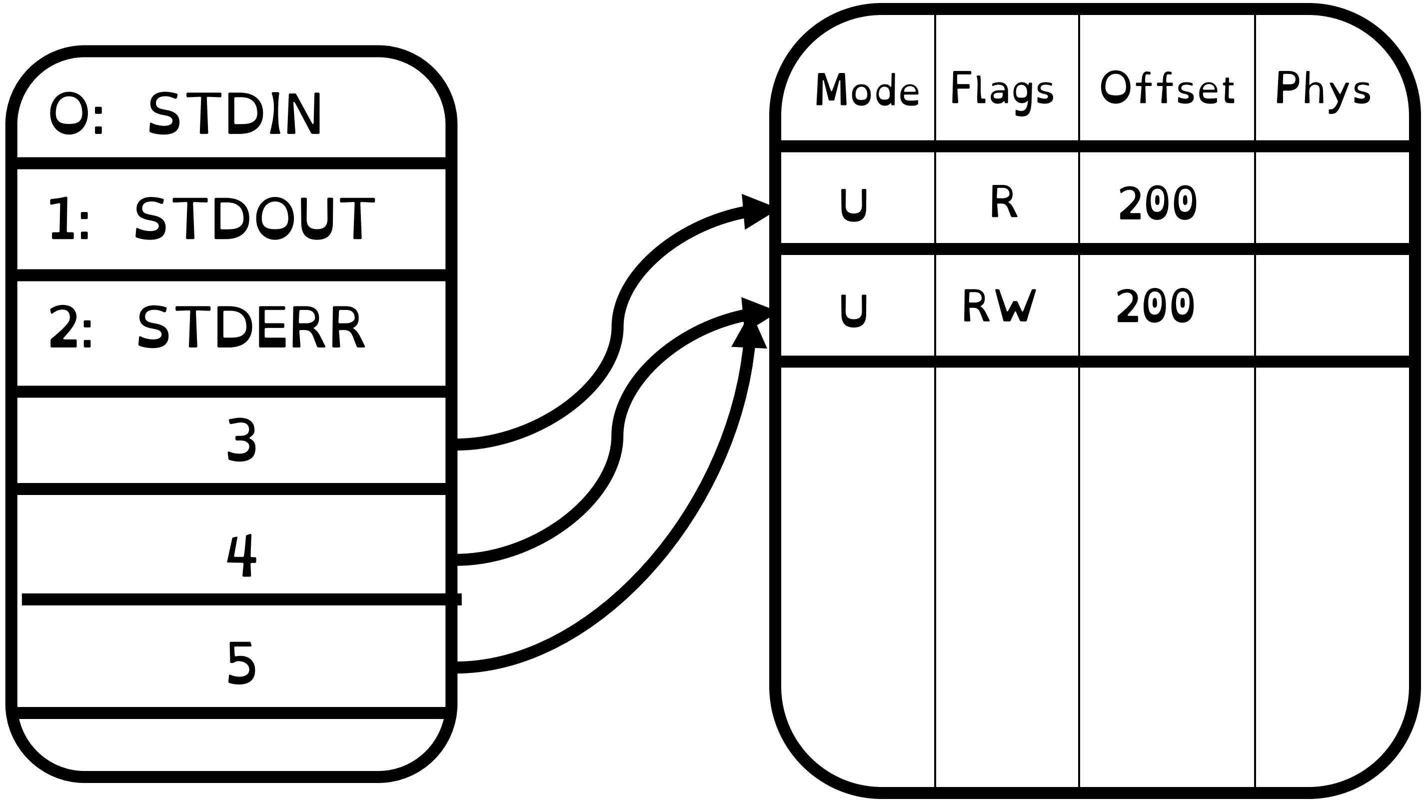
Duplicating FDs!

```
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);
```



Duplicating FDs!

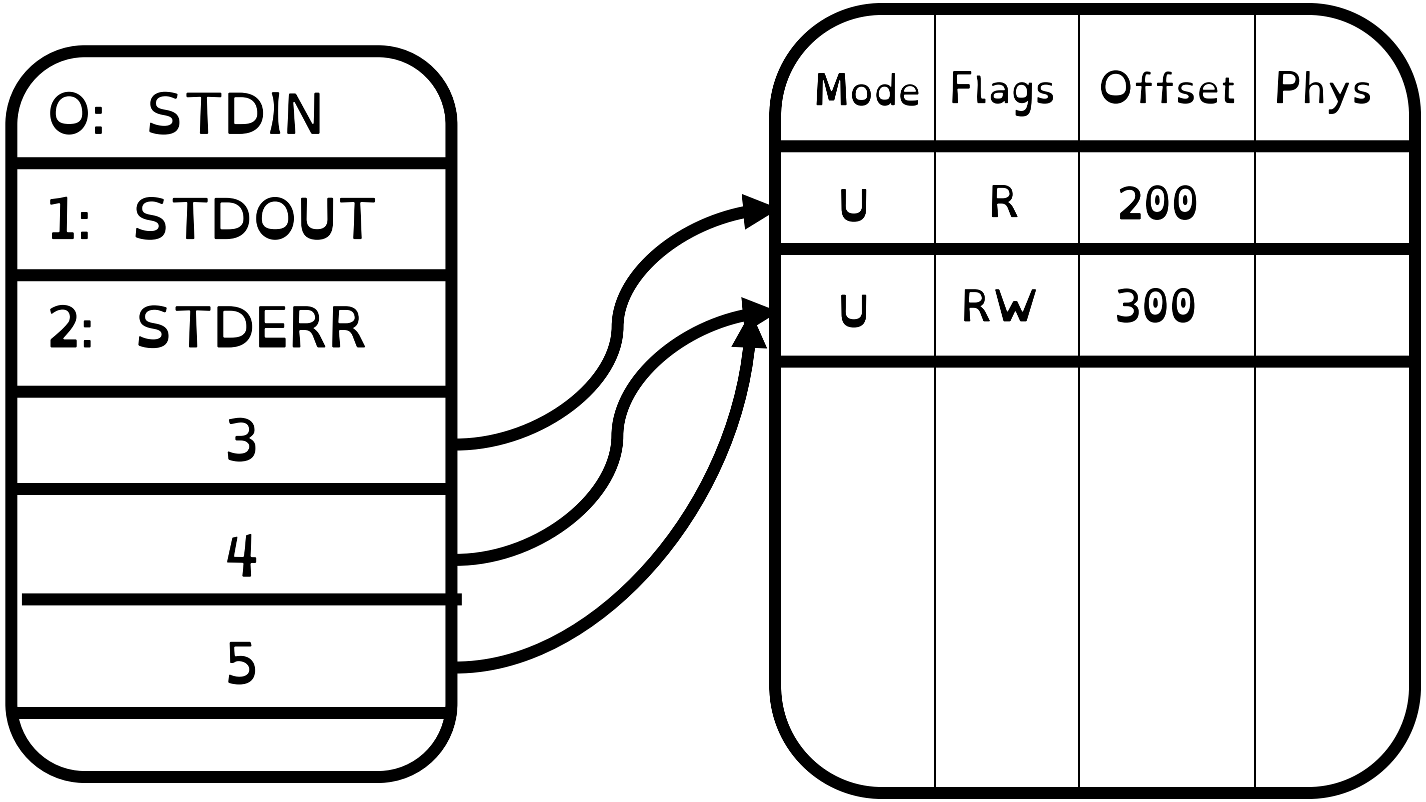
```
char buffer1[100];
char buffer2[100];
int fd = open(“foo.txt”,
O_RDONLY);
read(fd, buffer1, 100);
read(fd, buffer2, 100);

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

int fd3 = dup(fd2);
```



Creates copy fd3 of file descriptor fd2

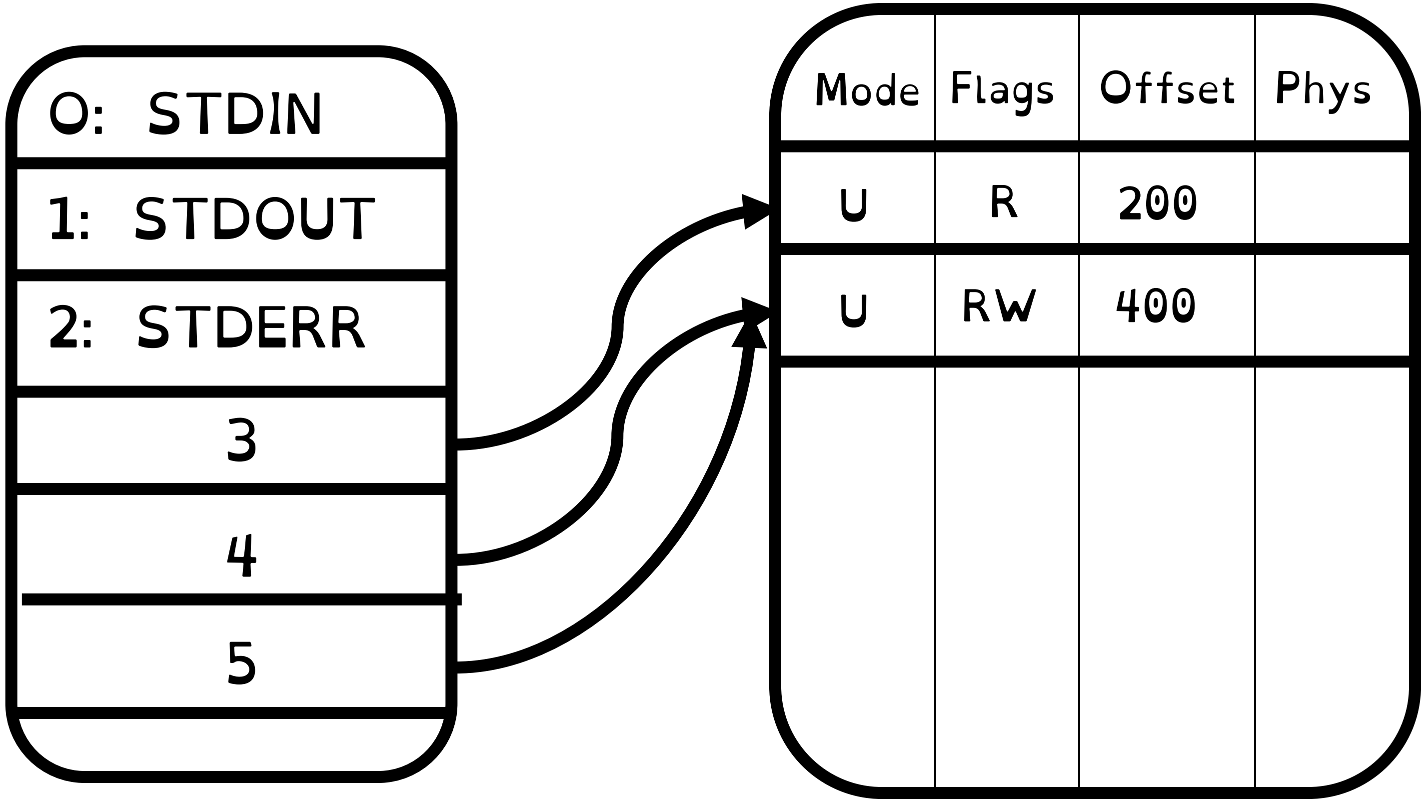
Duplicating FDs!

```
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);
```



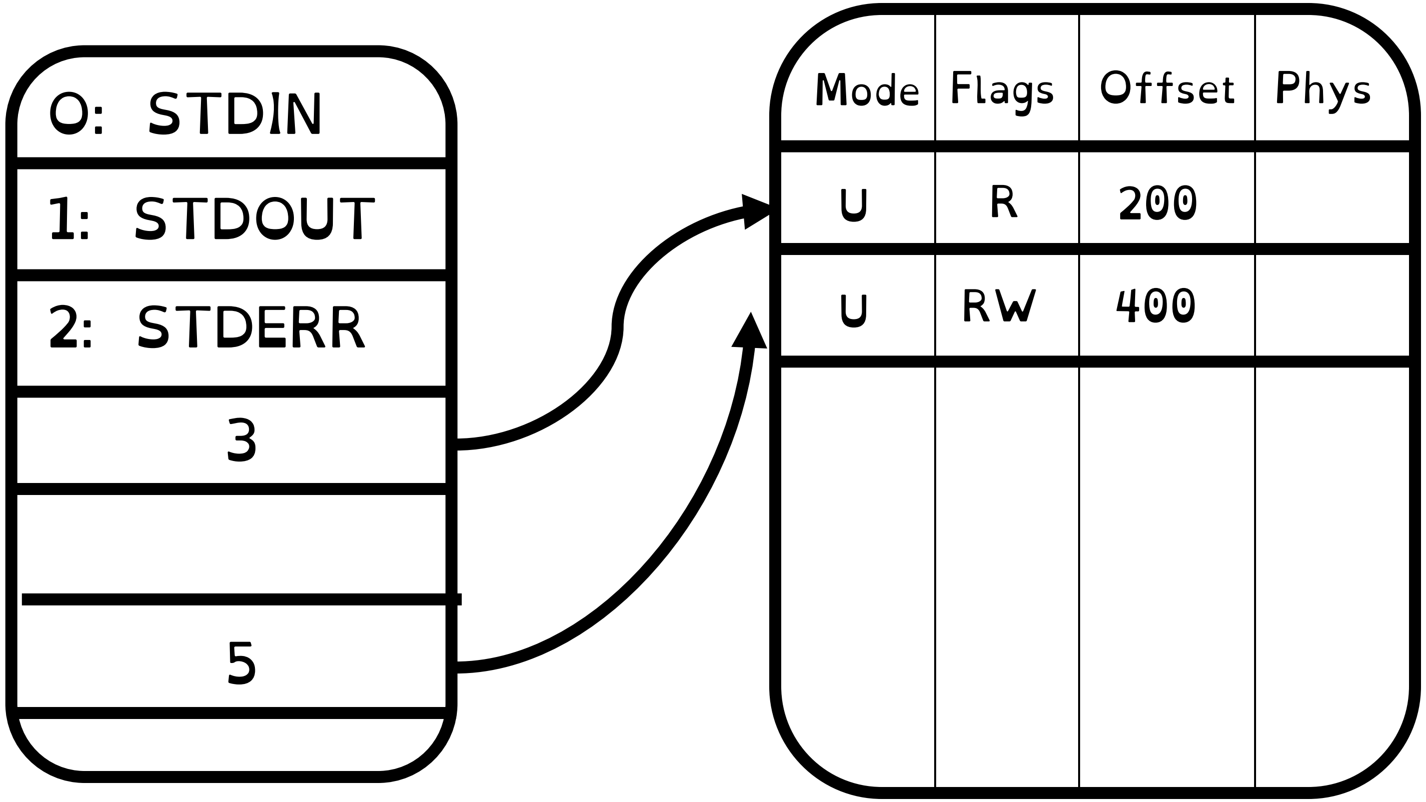
Duplicating FDs!

```
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);
```



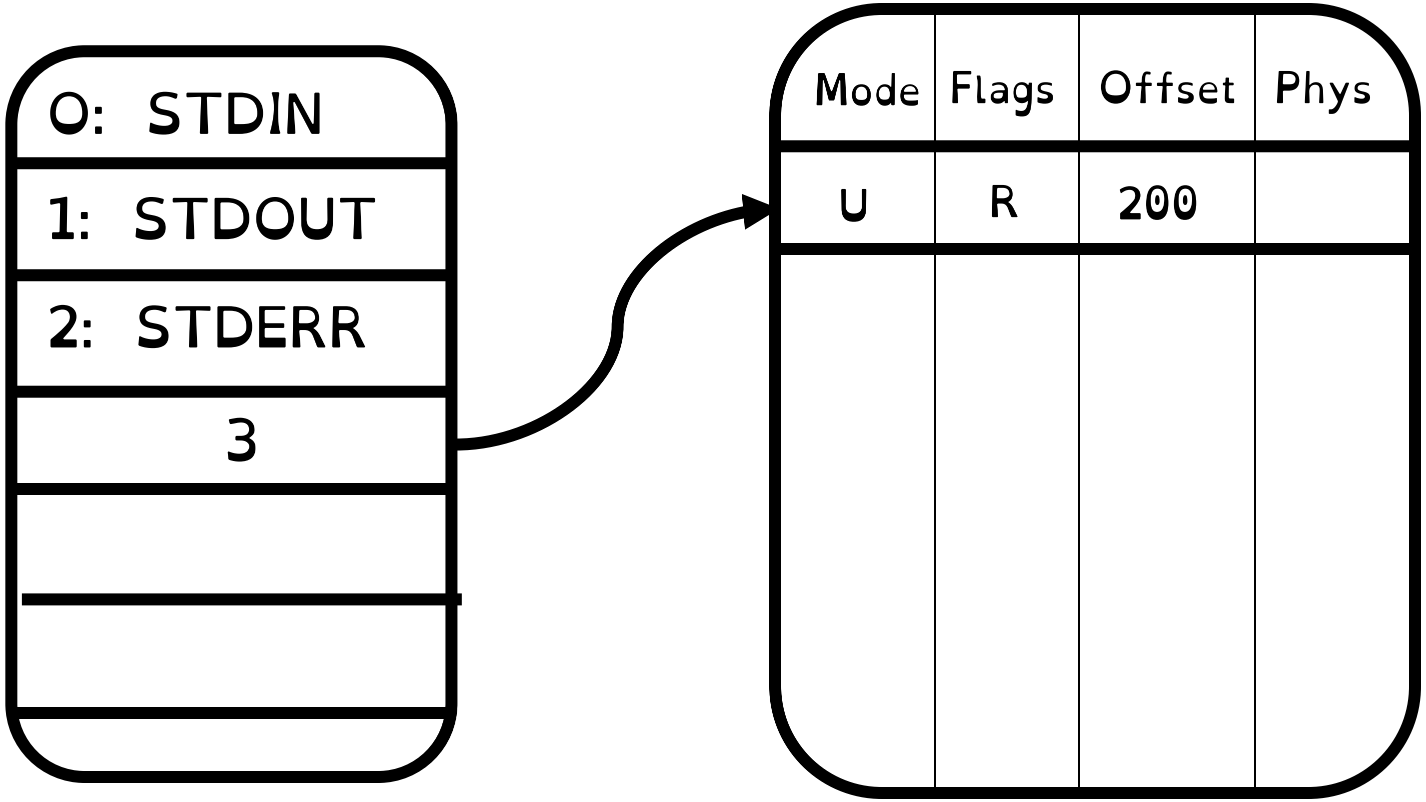
Duplicating FDs!

```
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);
```



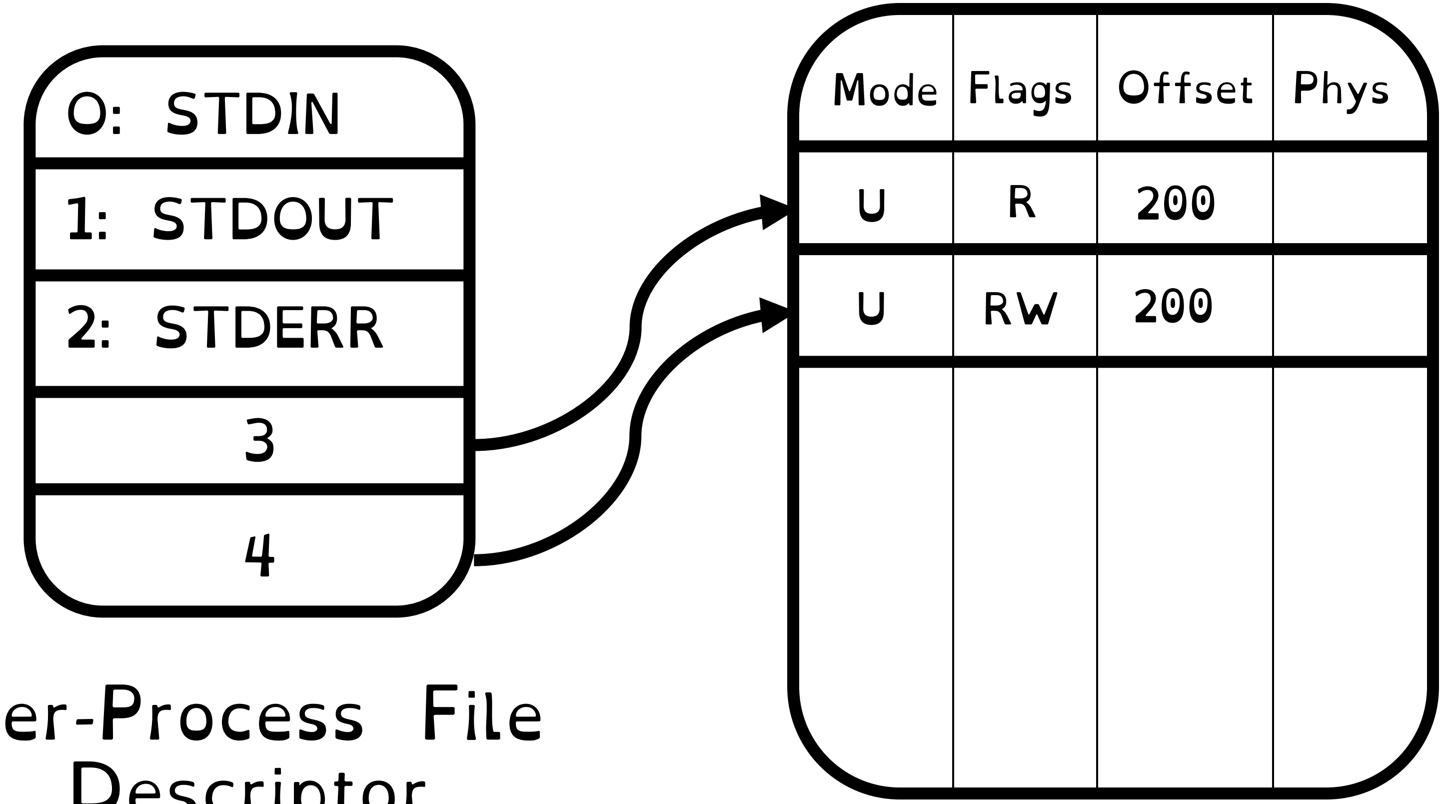
Duplicating FDs!

```
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

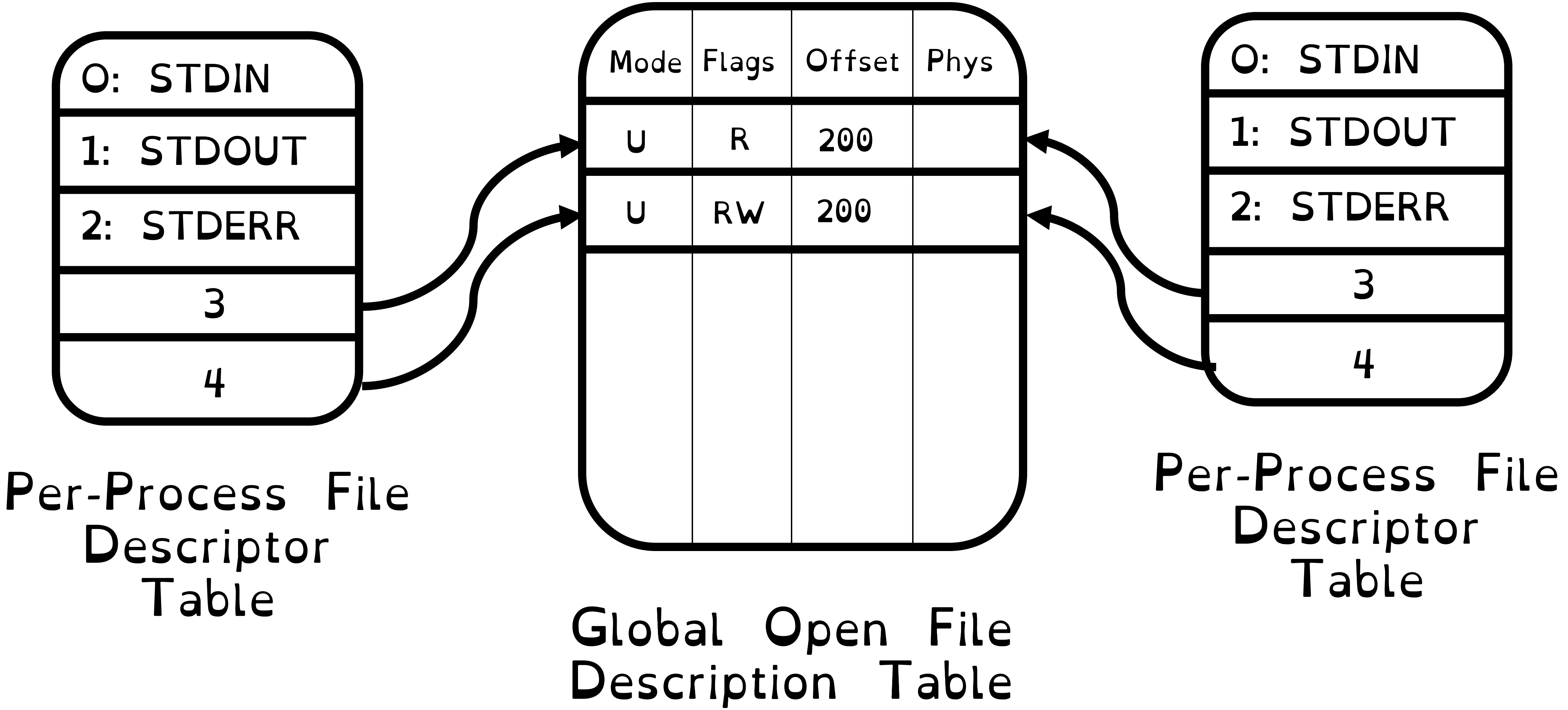
Forking FDs



Per-Process File Descriptor Table

Global Open File Description Table

Forking FDs



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

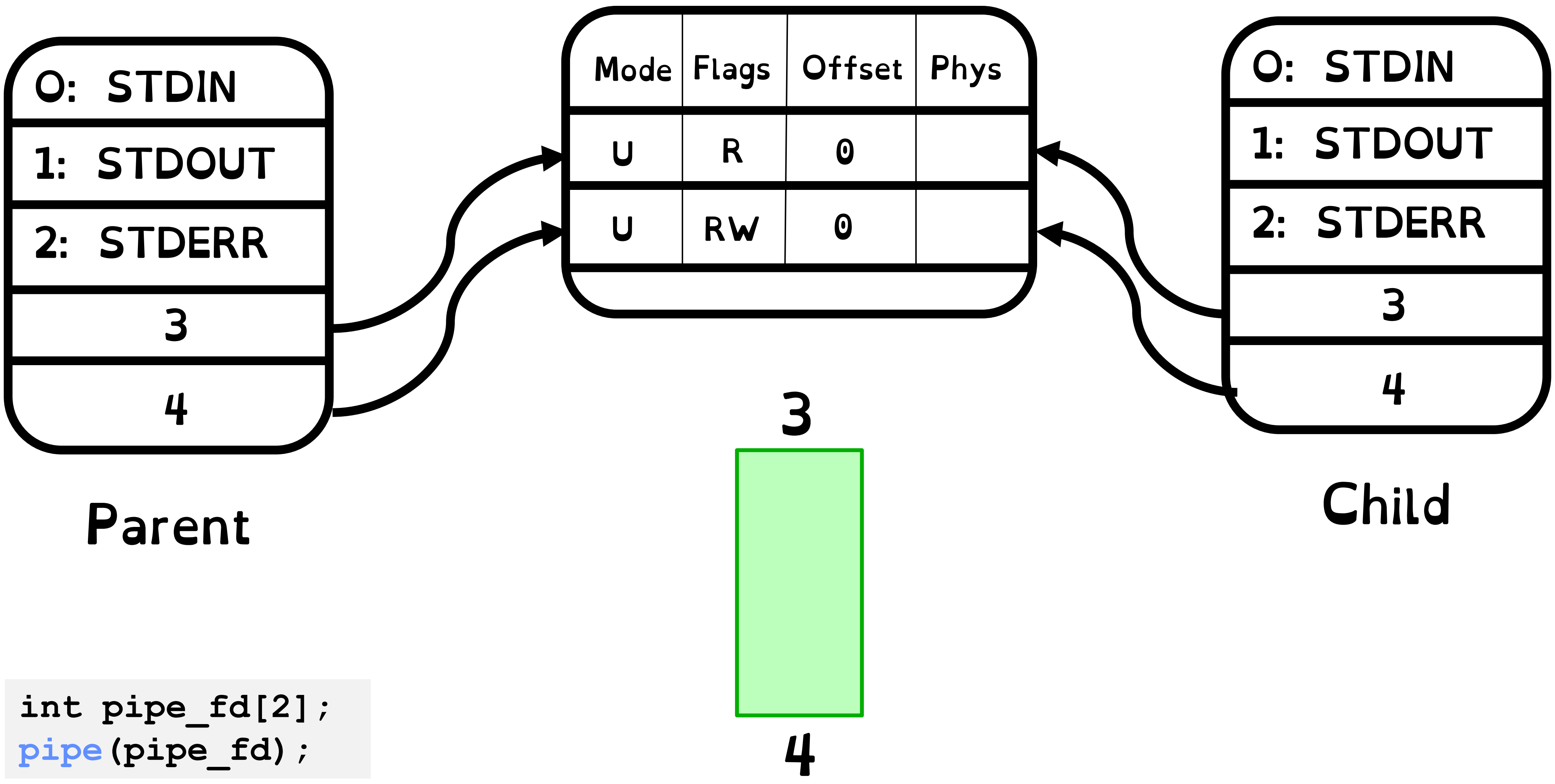
Block if pipe full. Block if pipe empty.

```
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

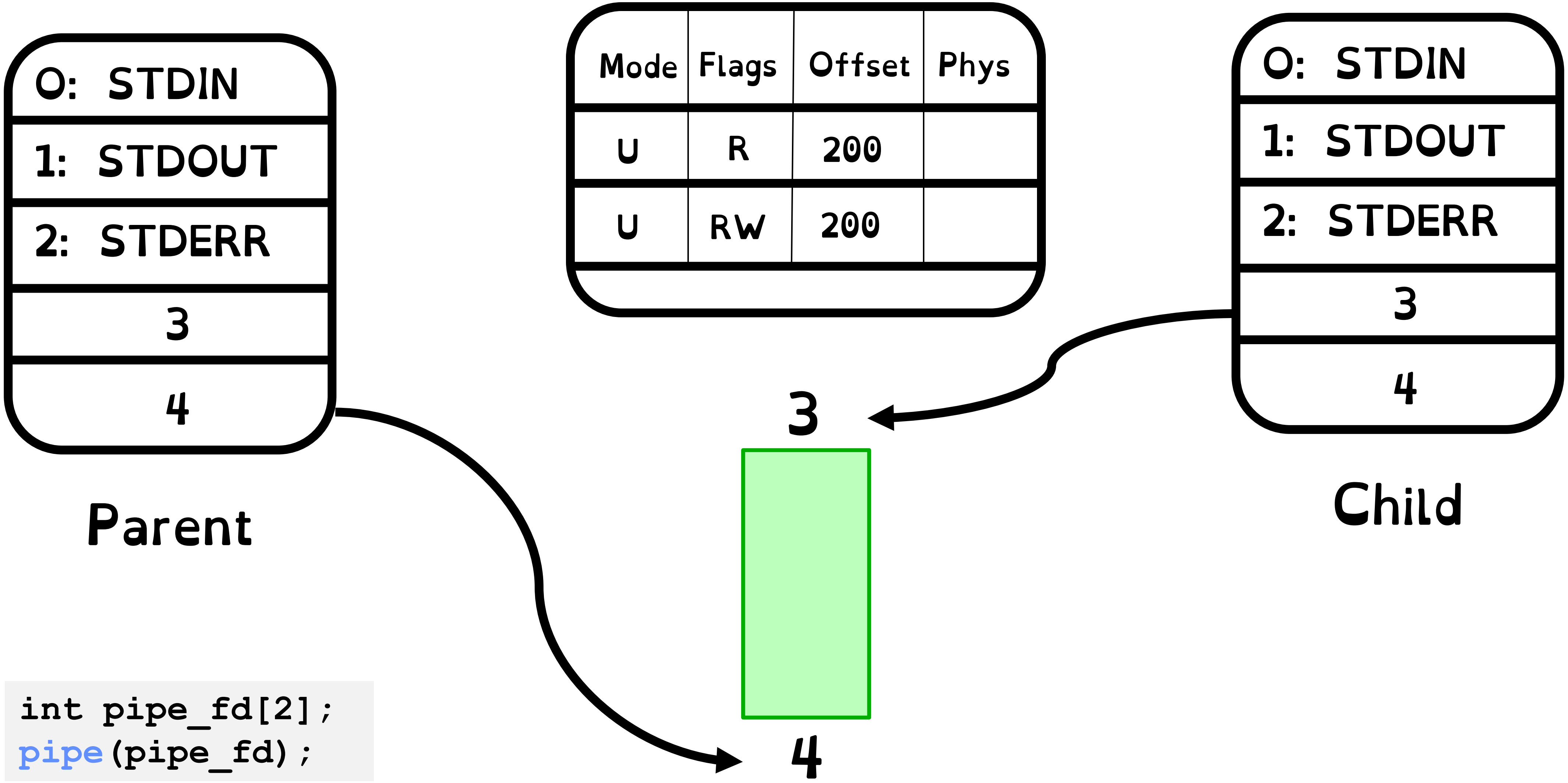
```
#include <unistd.h>
int main(int argc, char *argv[])
{
  char *msg = "Message in a pipe.\n";
  char buf[BUFSIZE];
  int pipe_fd[2];
  if (pipe(pipe_fd) == -1) {
    fprintf (stderr, "Pipe failed.\n"); return EXIT_FAILURE;
  }
  ssize_t writelen = write(pipe_fd[1], msg, strlen(msg)+1);
  printf("Sent: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

Pipes Between Processes

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)

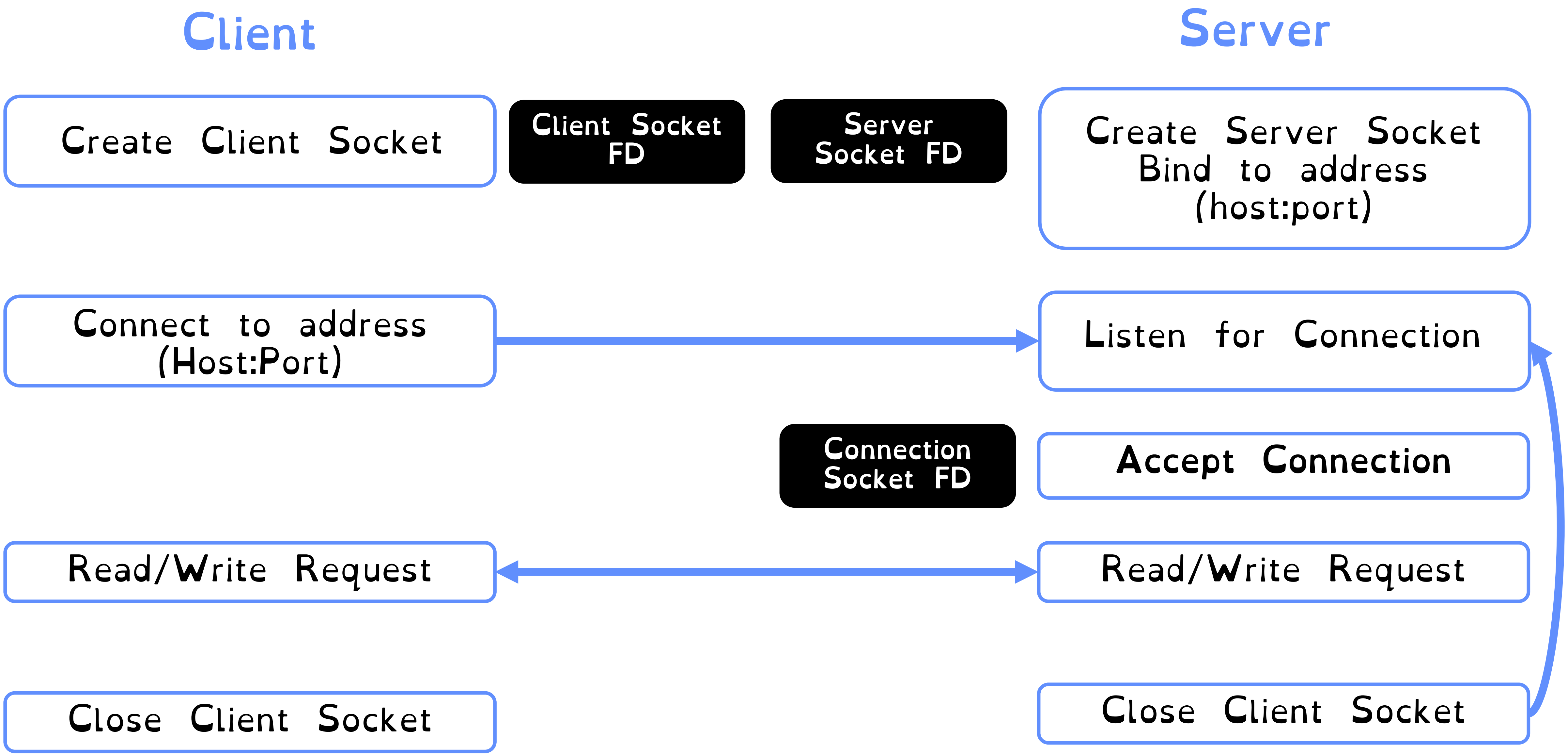
Port Number

0-1023 are system ports

1024-49151 are registered ports

49152–65535 are free

Sockets in concept



Client Protocol

```
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);
```

Server Protocol

```
// 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);
```

Summary: Input/Output Unix

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

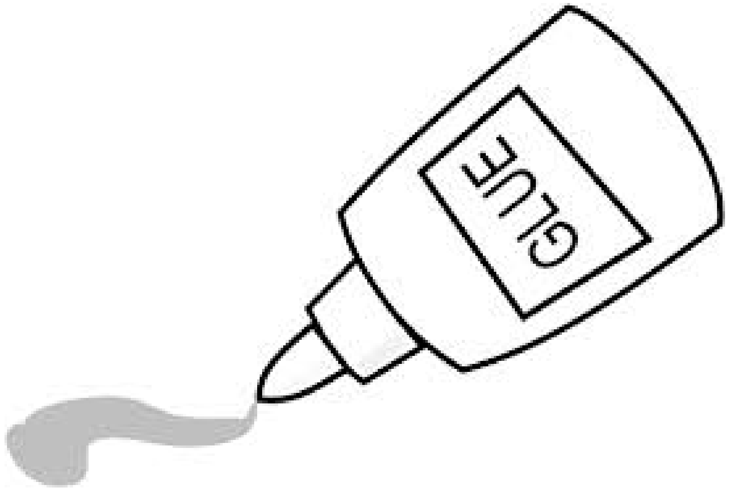
Per-process file descriptor table points to a global table of open file descriptions

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

Forked processes inherit FDs of parents

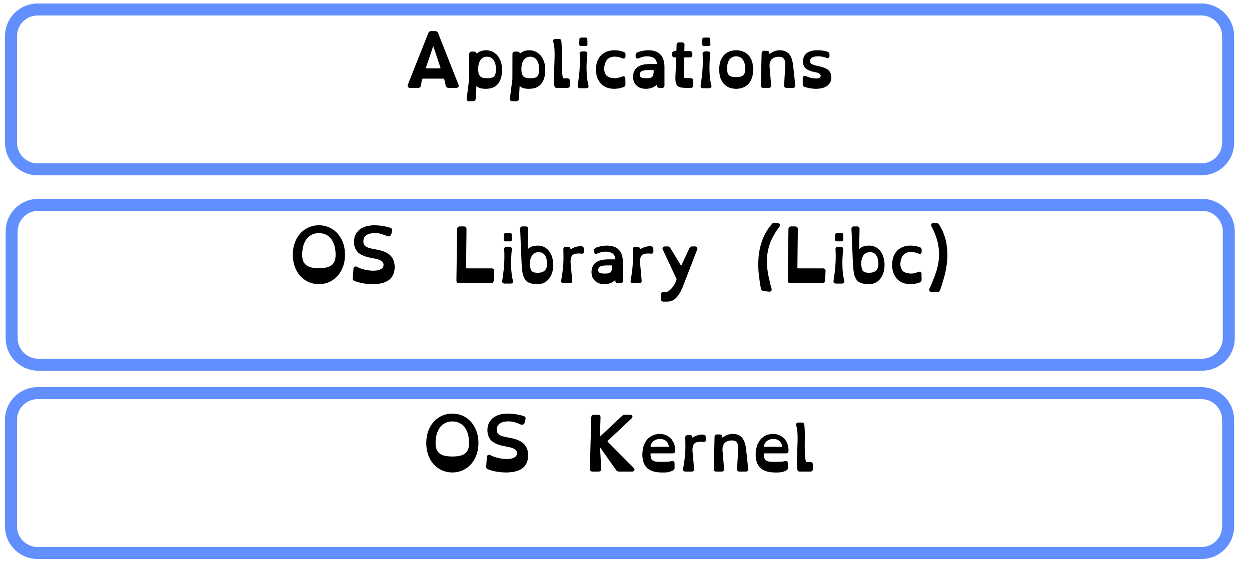
Goal 2: High-Level Systems API

OS Library

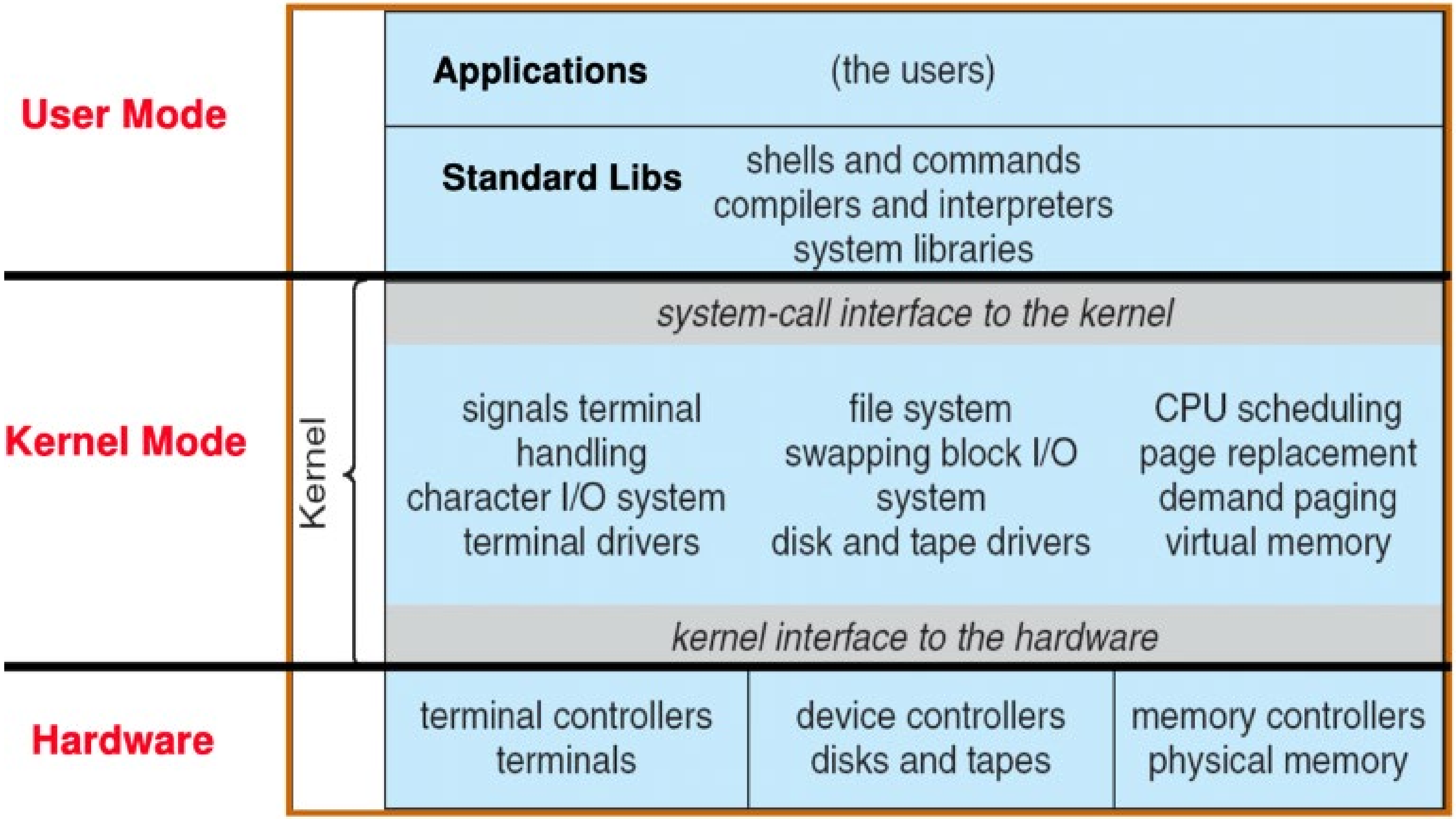


Glue

Provides a set of common services



OS Library (Standard Libraries)



OS Library (Standard Libraries)

1) Improve Programming API

Minimises glue clode

Simulates additional functionality

2) Performance

Minimises cost of syscalls

"High Level C API"

From FDs to Files

FILE* is OS Library wrapper for manipulating explicit files

Internally contains:
- File descriptor (from call to open)
- Buffer (array)
- Lock (in case multiple threads use the FILE concurrently)

FILE* API operates on streams – unformatted sequences of bytes (text or binary data), with a position

```
#include <stdio.h>

FILE *fopen( const char *filename,
             const char *mode );
int fclose( FILE *fp );
```

C High-Level File API

```
// character oriented
int fputc( int c, FILE *fp );              // rtn c or EOF on err
int fputs( const char *s, FILE *fp );      // rtn > 0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );
```

```
// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```

```
// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ... );
```

C Streams: Char-by-Char I/O

```
int main(void) {
  FILE* input = fopen("input.txt", "r");
  FILE* output = fopen("output.txt", "w");
  int c;

  c = fgetc(input);
  while (c != EOF) {
    fputc(output, c);
    c = fgetc(input);
  }
  fclose(input);
  fclose(output);
}
```

From Syscall to Library Call

read()	fread(), fgetc(), fscan()
	User-level logic
Trap into Kernel	Trap into Kernel
Execute read syscall handler()	Execute read syscall handler()
Switch to User Mode	Switch to User Mode
	User-level logic

FILE* is Buffered IO

Maintains a per-file user-level buffer.

Write Calls write to buffer. System flushes buffer to disk when full (or on special character)

Read Calls read from buffer. System reads from disk when buffer empty

Operations on file descriptors are unbuffered & visible immediately

API Benefit

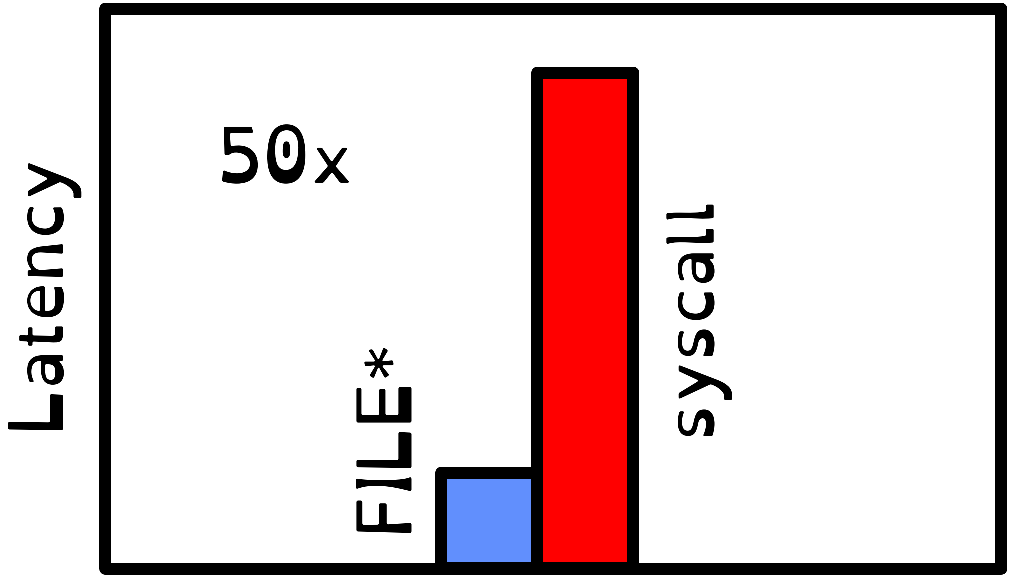
Buffering key to support different FILE IO APIs. Simulate additional functionality!

Kernel always read fixed size block from disk. Buffer into user-space.

OS Library parse buffer to read/write character/blocks/lines

User thinks they are writing individual characters or lines!

Performance Benefit



Syscalls are 25x more expensive than function calls (~100 ns)

Minimise amount copied

Great Power => Great Responsibility

If not careful, buffering can cause inconsistencies

```
char x = 'c';
FILE* f1 = fopen("file.txt", "w");
fwrite("b", sizeof(char), 1, f1);
FILE* f2 = fopen("file.txt", "r");
fread(&x, sizeof(char), 1, f2);
print("%c", x);
```

```
fflush(f1);
```

What will be printed?

1) The call to fread might see the latest write 'b'. Print b
2) Or it might miss it and see end of file. Print c

Avoid Mixing FILE* and File Descriptors

```
char x[10];
char y[10];
FILE* f = fopen("foo.txt", "rb");
int fd = fileno(f);
fread(x, 10, 1, f);
read(fd, y, 10);
```

Which bytes from the file are read into y?

A. Bytes 0 to 9
B. Bytes 10 to 19
C. None of these?

Answer: C! None of the above.

- The fread() reads a big chunk of file into user-level buffer
- Might be all of the file!

Goal 2: Introducing the Thread

Real-World Concurrency

Millions of drivers on motorway at once.

Student does homework while watching TV

Faculty has lunch while grading papers and watching the Rugby World Cup

* The character portrayed in this slide are fictitious. No identification with actual persons should be inferred.

OS Concurrency

Efficiently manage many different processes

Efficiency manage concurrent interrupts

Efficiently manage network interfaces

Must provide programmers with abstractions for expressing and managing concurrency

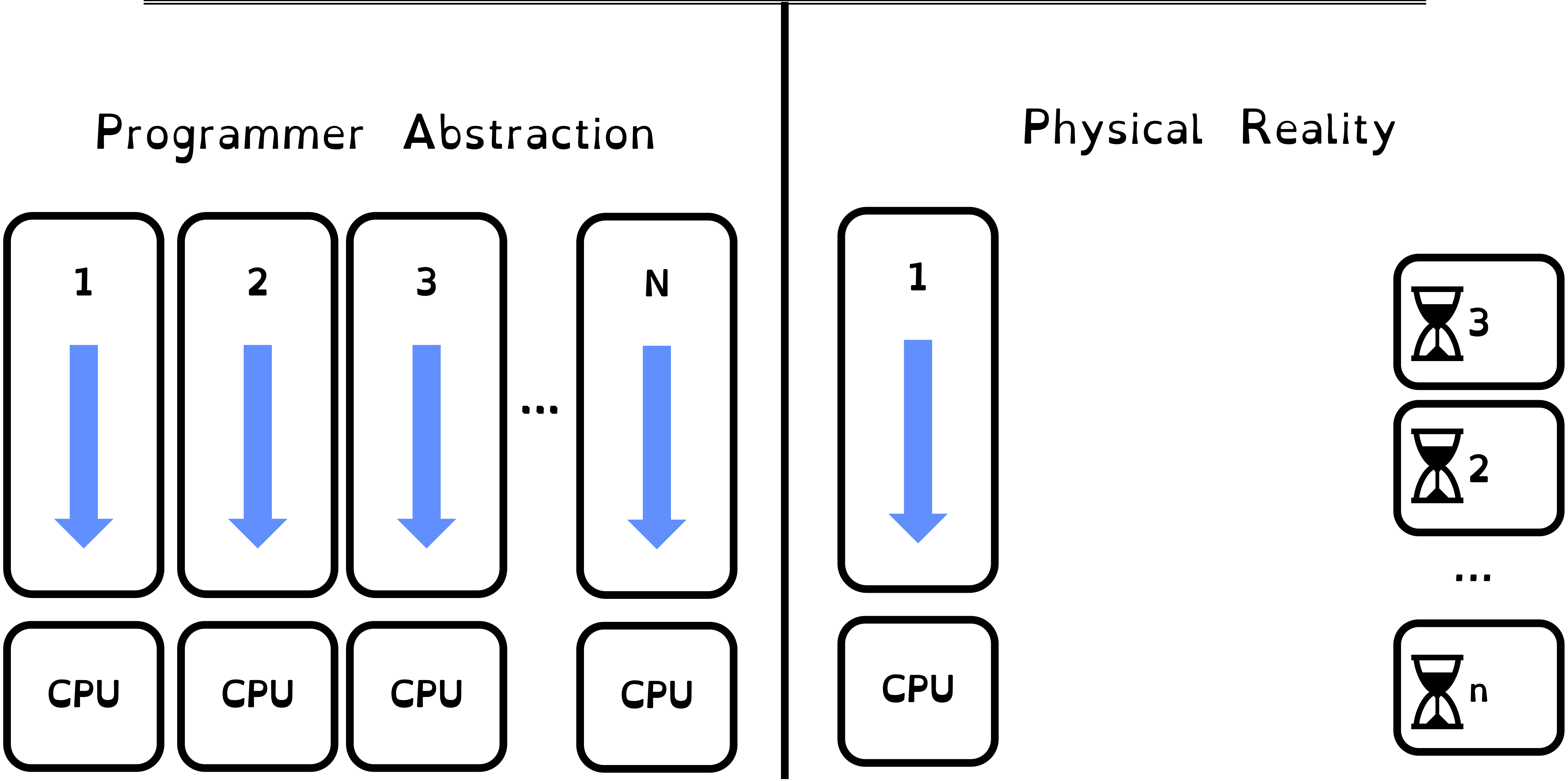
What is a thread?

A single execution sequence that represents a separately schedulable task

Virtualizes the processor. Each thread runs on a dedicated virtual processor (with variable speed). Infinitely many such processors.

Threads enable users to define each task with sequential code. But run each task concurrently

What is a thread?



Why do we need threads?

Natural Program Structure

Simultaneously update screen, fetch new data from network, receive keyboard input

Exploiting parallelism

Split unit of work into n tasks and process tasks in parallel on multiple cores.

Responsiveness

High priority work should not be delayed by low priority work. Schedule as separate threads for independence

Masking IO latency

Continue to do useful work on separate thread while blocked on IO

Thread $\neq$ Process

Processes defines the granularity at which the OS offers isolation and protection

Threads capture concurrent sequences of computation

Processes consist of one or more threads!

Process Protection

Thread Concurrency

All you need is love (and a stack)

No protection

Threads inside the same process and are not isolated from each other

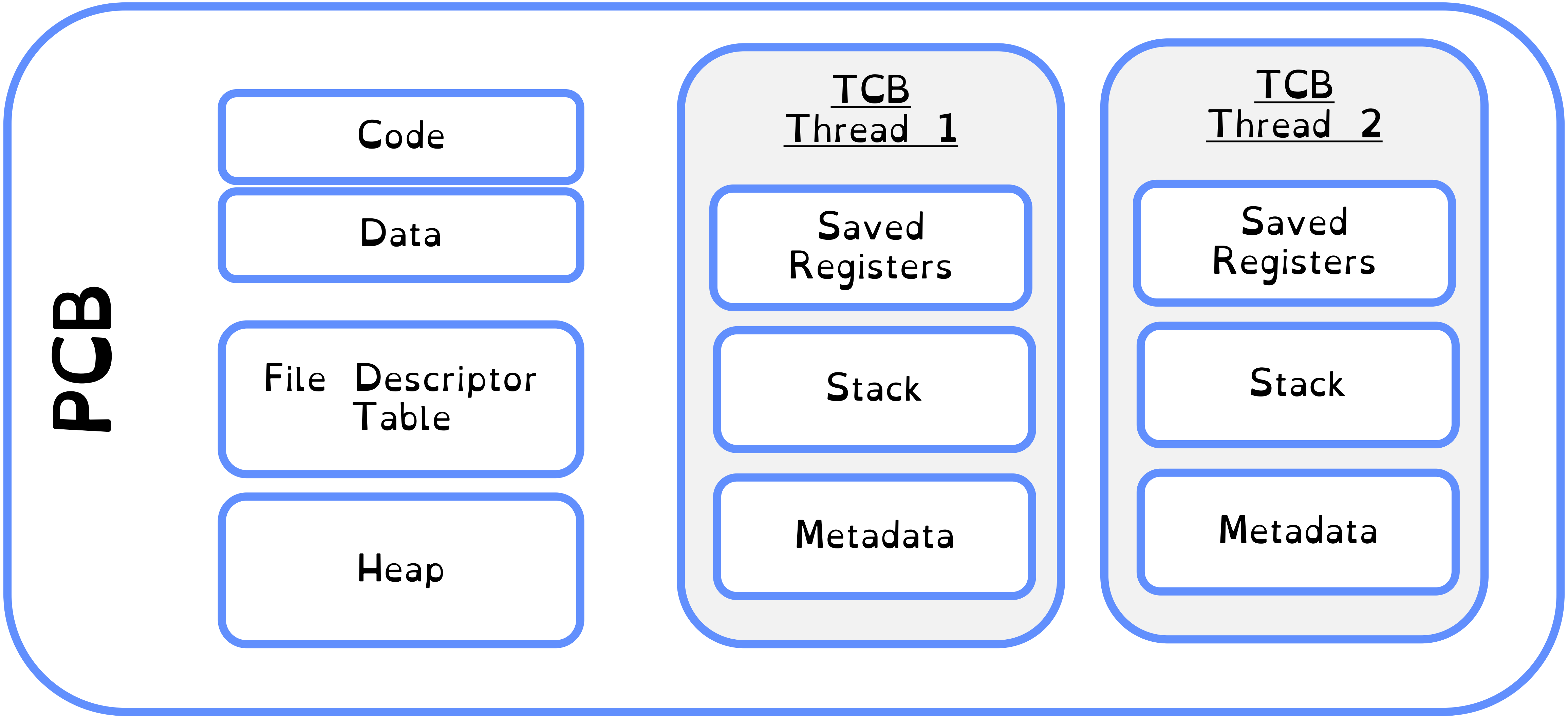
Share an address space & share IO state (FDs)

Individual execution

Threads execute disjoint instruction streams. Need own execution context

Individual stack, register state (including EIP, ESP, EBP)

All you need is love (and a stack)



One Thread, Two Abstractions

User Threads	Kernel Threads
One PCB for the process	Each thread has own TCB
Each thread has own TCB stored in heap of process.	Each thread individually schedulable.
Threads in user-space only. Invisible to kernel	Requires mode switch to switch threads

User Threads

Run mini-OS/scheduler in user space

Real OS is unaware of threads. Stores a single PCB for all user threads within the same process

Each thread has associated Thread Control Block (TCB) kept by process in heap

User-level threads incur lower overhead than kernel-level thread

Kernel Threads

Kernel knows about threads.

Schedules each thread individually

Each thread has a separate PCB.

PCBs of threads mapped in the same process share address space, files, code/data.
Different stack and registers.

Context-switching requires a mode switch

User Threads vs Kernel Threads

	Kernel-Level Threads	User-Level Threads
Ease of Implementation	Easy to implement: just like process, but with shared address space	Requires implementing user-level schedule and context switches
Handling System Calls	Thread can run blocking systems call concurrently	Blocking system call blocks all threads: needs OS support for non-blocking system calls (scheduler activations)
Cost of Context Switching	Thread switch requires three context switches	Thread switch efficiently implemented in user space

(Kernel) Threads in Linux

To create a process

Call (internally)
`Clone` system call
(`do_fork()` in `kernel/fork.c`)

Duplicate `task_struct`.

Mark new process as runnable.

?

To create a thread

Call (internally)
`Clone` system call
`do_fork()` in `kernel/fork.c`)

Duplicate `task_struct`.

Mark new process as runnable.

(Kernel) Threads in Linux

Everything is a thread (`task_struct`)

Scheduler only schedules `task_struct`

To fork a process:

Invoke `clone(…)`

To create a thread:

Invoke `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0)`

CLONE_VM: Share address space. CLONE_FS: share file system. CLONE_FILES: share open files. CLONE_SIGHAND: share handlers with parents

Processes are better viewed as the containers in which threads execute

OS Library API for Threads (pThreads)

```
int pthread_create(pthread_t *thread, …
          void *(*start_routine)(void*),  void *arg);
```

Thread created and runs `start_routine`

```
void pthread_exit(void *value_ptr);
```

Terminates thread and makes value_ptr available to any successful join

```
int pthread_yield();
```

Causes thread to yield the CPU to other threads

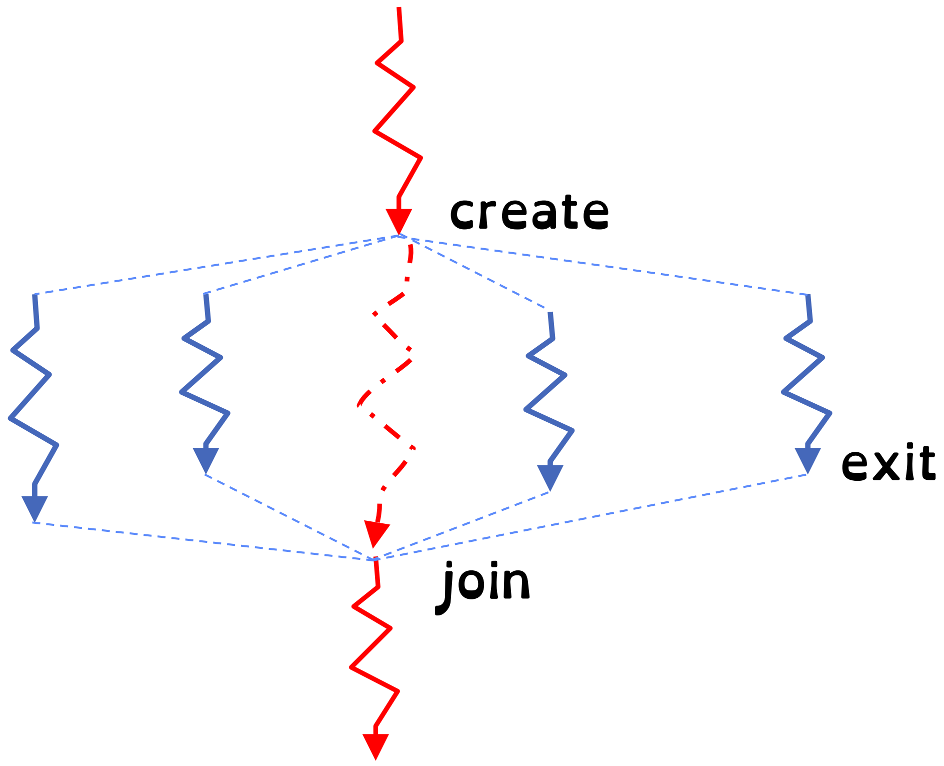
```
int pthread_join(pthread_t thread, void **value_ptr);
```

Suspends execution of calling thread until target thread terminates.

Pthread Example

```
 void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
 }

 int main(int argc, char *argv[]) {
   pthread_t p1, p2;
   printf("main: begin\n");
   pthread_create(&p1, NULL, mythread, "A");
   pthread_create(&p2, NULL, mythread, "B");
   // join waits for the threads to finish
   pthread_join(p1, NULL);
   pthread_join(p2, NULL);
   printf("main: end\n");
}
```

Fork-Join Pattern



Main thread *creates* (forks) collection of sub-threads passing them args to work on…

… and then *joins* with them, collecting results.

Revisit the Server Protocol

```
// Socket setup code elided…
while (1) {
  // Accept a new client connection, obtaining a new sc
  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);
```

How would you rewrite the concurrent server example using threads rather than processes?

~~Multiprocess~~ Multithreaded server!

```
// Socket setup code elided…
Int
while (1) {
  // Accept a new client connection, obtaining a new socket
  pthread_t tid;
  int conn_socket = accept(server_socket, NULL, NULL);
  int* arg = (int*) malloc(sizeof(int));
  *arg = conn_socket;
  pthread_create(&tid, NULL &serve_client, &arg);
}
close(server_socket);
```

Reviewing the pthread_create(…)

Do some work like a normal fn… place syscall # into %eax put args into registers %ebx, … special trap instruction	OS Library
Mode switches & switches to kernel stack. Saves recovery state Jump to interrupt vector table at location 128. Hands control to syscall_handler	CPU
Use %eax register to index into system call dispatch table. Invoke do_fork() method. Initialise new TCB. Mark thread READY. Push errcode into %eax	Kernel
Restore recovery state and mode switch	CPU
get return values from regs Do some more work like a normal fn…	OS Library

With great power comes great concurrency

```
pthread_t tid[2];
int counter;

void* doSomeThing(void *arg) {
  unsigned long i = 0;
  for (int i = 0 ; i < 1000 ; i++) {
    counter += 1;
  }
  return NULL;
}

int main(void) {
  int i = 0;
  while(i++ < 2) {
    pthread_create(&(tid[i]), NULL, &doSomeThing,
  }
  pthread_join(tid[0], NULL);
  pthread_join(tid[1], NULL);
  printf("Counter %d \n", counter);
  return 0;
}
```

What will be the final answer?

```
crooks@laptop> gcc concurrency.c -o concurrency -pthread
```

```
crooks@laptop> ./concurrency
```

```
Counter 2000
```

```
crooks@laptop> ./concurrency
```

```
Counter 1937
```

```
crooks@laptop> ./concurrency
```

```
Counter 1899
```

With great power comes great concurrency

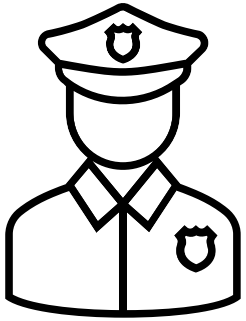
Protection is at process level.

Threads not isolated.
Share an address space.

Non-deterministic interleaving of threads

T1	T1	T1	T2	T2	T2
T1	T2	T2	T1	T1	T2
T2	T2	T2	T1	T1	T1

With great power comes great concurrency



Public Enemy #1:

THE RACE CONDITION

Next four lectures: how can we regulate access to shared data across threads?