

文件、进程间的通信

Files

IPC, Pipes and Sockets

参考教材：操作系统概念 第3章 进程

扩展阅读：深入理解计算机系统（CSAPP, 3rd E）第8、11章

上节回顾

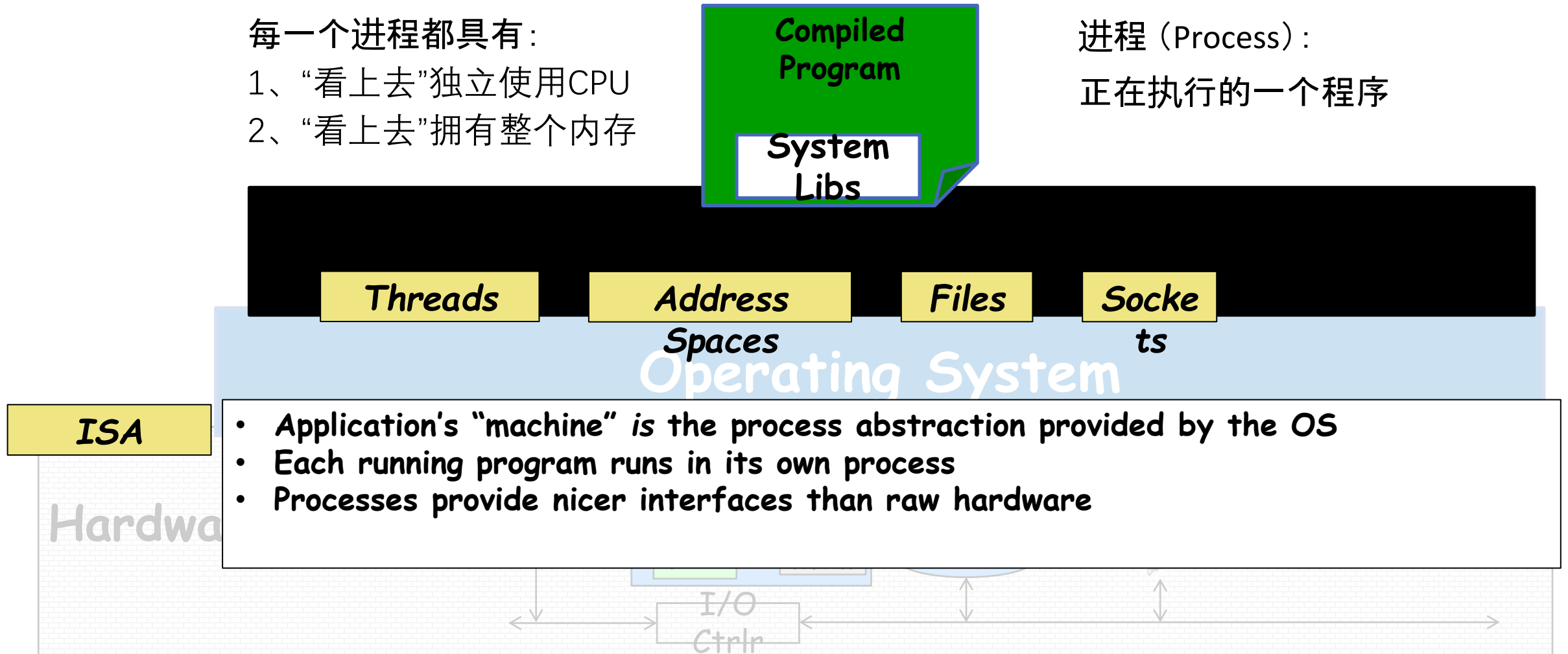
- 进程的概念
- 进程在操作系统中的表示
- 进程管理
 - exit : 进程撤销 terminate a process
 - fork : 进程创建 copy the current process
 - exec : 装入和运行其它程序 change the program being run
 - wait : 等待子集进程结束 wait for a process to finish
 - kill : 发信号 send a signal (interrupt-like notification) to another process
 - signal: set handlers for signals

进程的概念

每一个进程都具有：

- 1、“看上去”独立使用CPU
- 2、“看上去”拥有整个内存

进程 (Process) :
正在执行的一个程序



Linux 中对进程的表示

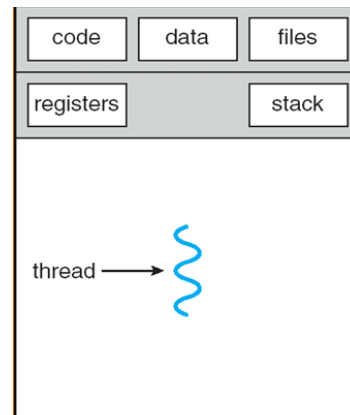
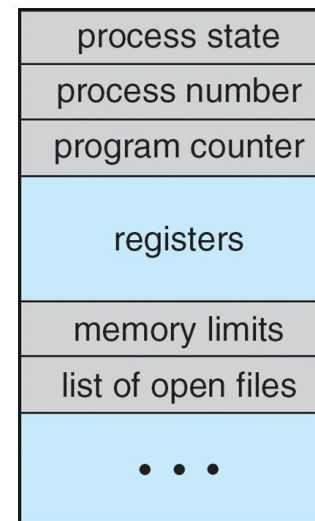
- linux中每一个进程都由task_struct 数据结构来定义. 是对进程控制的唯一手段.

- C structure **task_struct**

```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this pro */
```

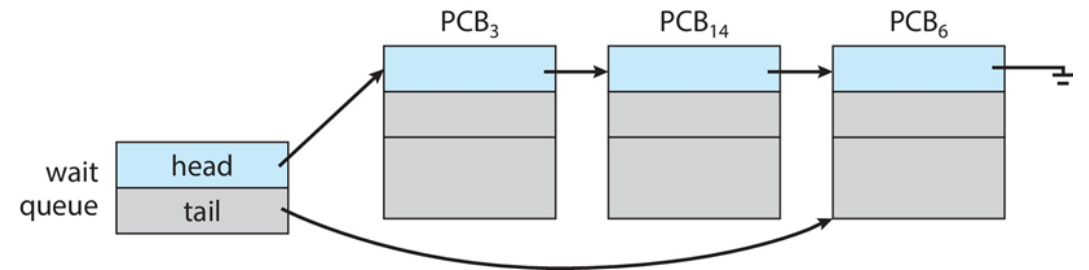
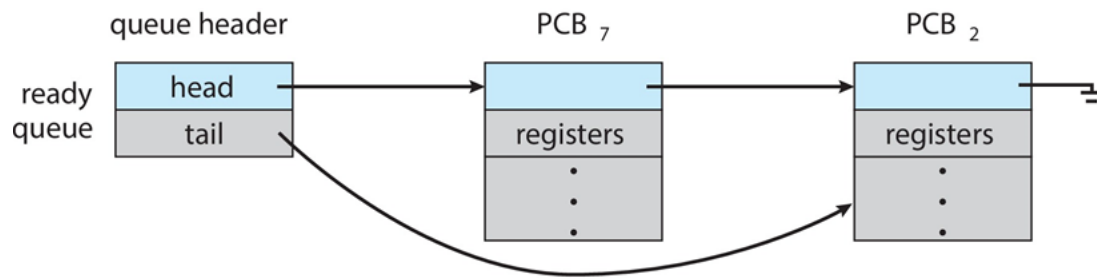
```
union task_union {
    struct task_struct task;
    unsigned long stack[2408];
}
```

给内核栈分配8K的内存，并把其中的一部分给task_struct使用。



进程调度

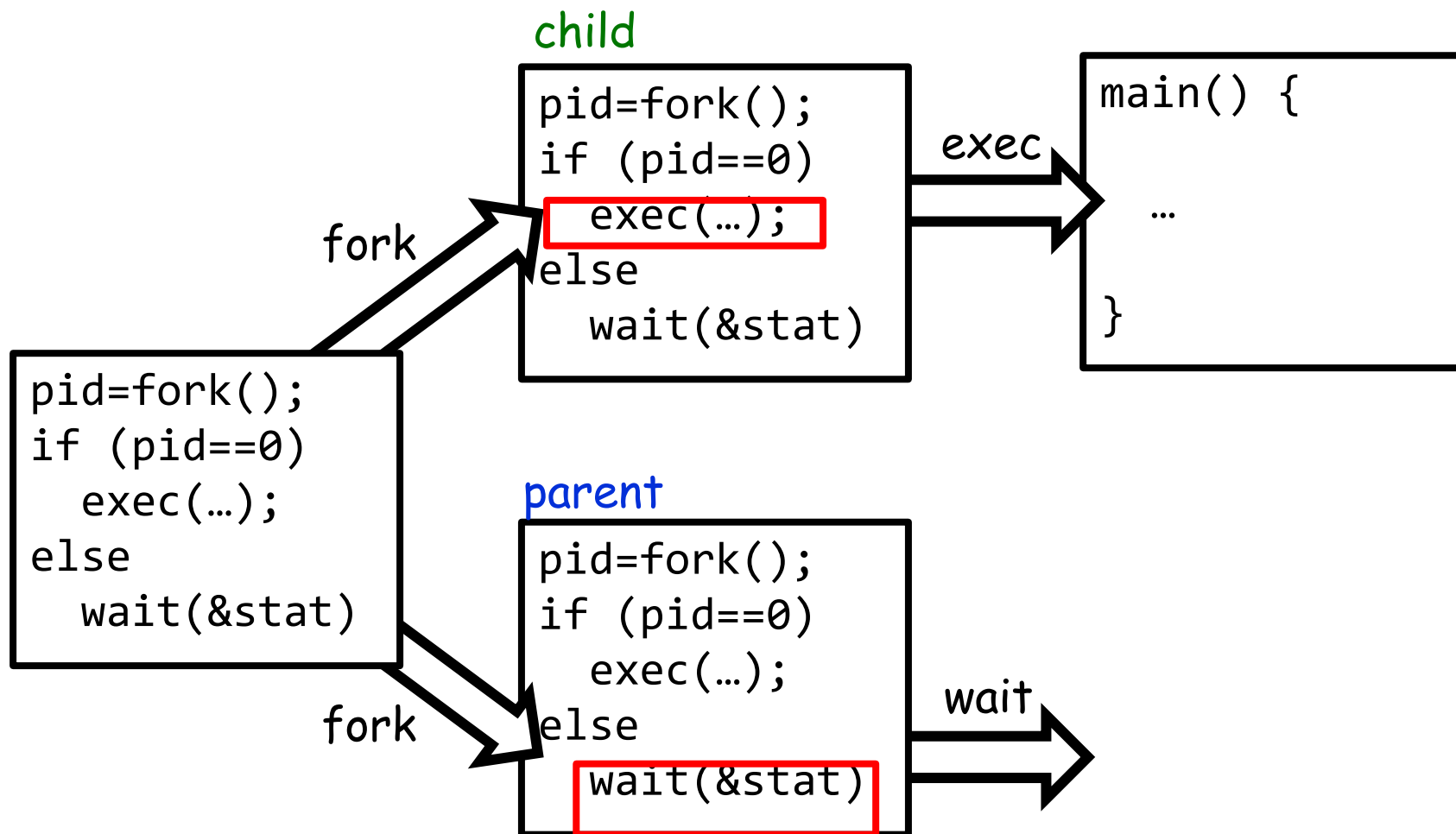
```
if ( readyProcesses(PCBs) ) {  
    nextPCB = selectProcess(PCBs);  
    run( nextPCB );  
} else {  
    run_idle_process();  
}
```



进程管理 API

- 进程管理
 - `exit` : 进程撤销, 释放大大部分空间, 但不包括当前进程的PCB
 - `fork` : 进程创建, 创建新的PCB, 复制父进程PCB中部分内容 (file, sighand, mm...)
 - `exec` : 装入和运行其它程序, 覆盖原PCB中 `code`, `data`, `stack` 等信息
 - `wait` : 挂起当前进程, 直到某个子进程终止, 回收该子进程
 - `kill` : 发信号给一个进程 send a signal (interrupt-like notification) to another process
 - `signal`: 设置信号处理程序 set handlers for signals
 - 扩展阅读: 深入理解计算机系统 (CSAPP, 3rd E) 第章

启动新程序 (举例: Shell 程序)



Linux 启动的过程

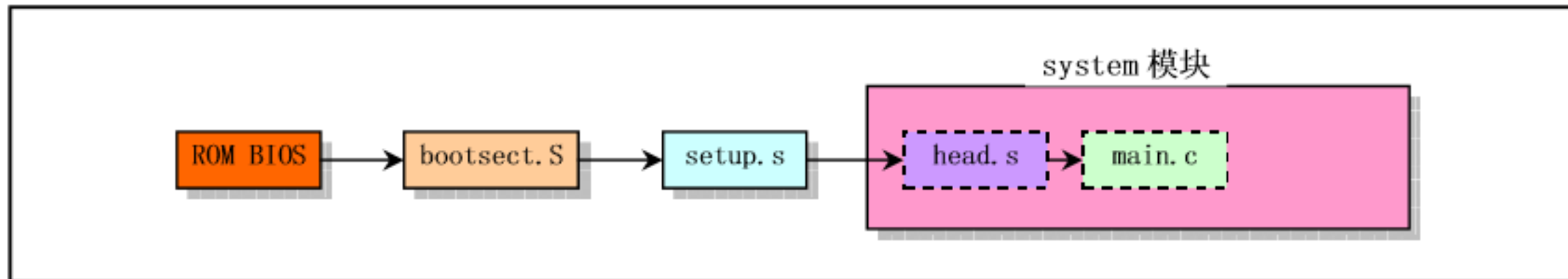
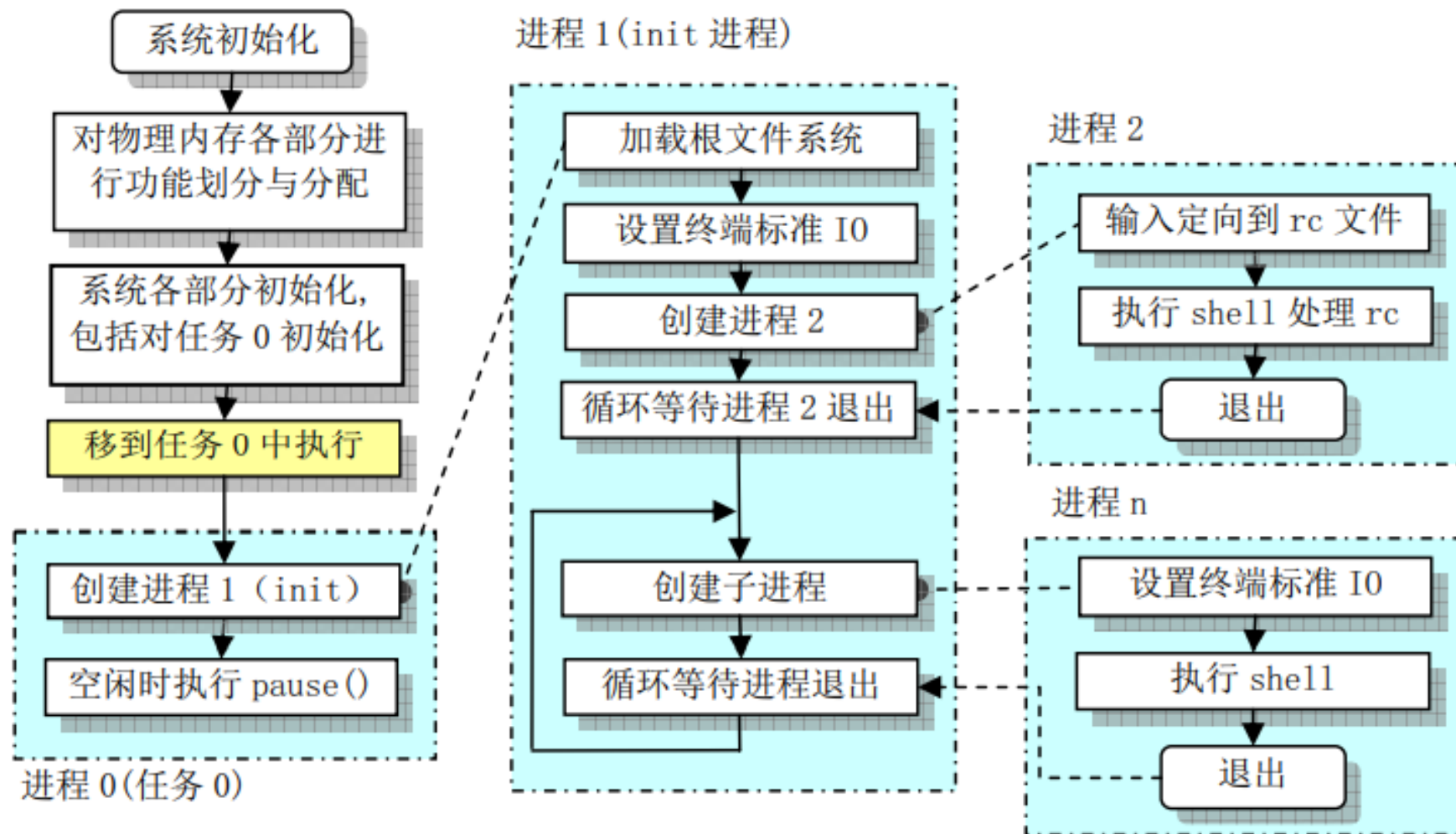


图 6-1 从系统加电起所执行程序的顺序

Head.s 代码的主要作用是初步初始化中断描述符表中的 256 项门描述符，检查 A20 地址线是否已经打开，测试系统是否含有数学协处理器。然后初始化内存页目录表，为内存的分页管理作好准备工作。最后跳转到 system 模块中的初始化程序 init/main.c 中继续执行。

Linux 0.11 内核初始化流程图



Shell

- **main**中的**fork()**创建了第1个进程

- **init**执行了**shell**(Windows桌面)

- **shell**再启动其他进程

```
int main(int argc, char * argv[])  
{ while(1) { scanf("%s", cmd);  
    if(!fork()) {exec(cmd);} wait(); } }
```

一命令启动一个进程，返回**shell**再启动其他进程...

- A shell is a job control system
 - Allows programmer to create and manage a set of programs to do some task
- You will build your own shell in Project 1...
 - ... using fork and exec system calls to create new processes...

信号机制

- 功能
 - 信号是送到进程的“软件中断”，通知进程出现了非正常事件
 - » 例如：浮点错、非法内存访问、执行无效指令,某些按键（如ctrl-c、del等）
- 信号
 - 内核产生信号
 - 进程自己或者其他进程发出的
- 例如：

SIGINT	中断进程。用户按Del键或Ctrl-C键时产生
SIGTERM	软件终止信号。用kill命令时产生
SIGCLD	进程的一个子进程终止。
SIGFPE	浮点溢出
SIGILL	非法指令
SIGSEGV	段违例

概念: 发送信号

- 操作系统内核通过修改目标进程的上下文状态来发送信号
- 例如: task struct 中和信号相关的字段

```
1  /* signal handlers */
2      struct signal_struct *signal;           // signal指向进程的信号描述符
3      struct sighand_struct *sighand;
4
5      sigset_t blocked, real_blocked;
6      sigset_t saved_sigmask; /* restored if set_restore_sigmask() was us
7      struct sigpending pending;
8
9      unsigned long sas_ss_sp;
10     size_t sas_ss_size;
11     int (*notifier)(void *priv);
12     void *notifier_data;
13     sigset_t *notifier_mask;
```

发信号? (1) 通过命令kill 发信号

- 用法与功能

ctrl+c 终止当前在终端窗口中运行的命令或脚本

`kill -signal PID-list`

kill命令用于向进程发送一个信号

- 举例

- kill 1275

- » 向进程1275的进程发送信号，默认信号为15(SIGTERM)，一般会导致进程死亡

- kill -9 1326

- » 向进程1326发送信号9(SIGKILL)，导致进程死亡

- kill -9 pid，是不顾后果的强制终止(如果速度够快，和ctrl + c是一样的)
- kill -15 pid，先关闭和其有关的程序，再将其关闭

kill -l (查看Linux/Unix的信号变量)

```
root@localhost:/# kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS     8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM     15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD    18) SIGCONT    19) SIGSTOP     20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF    28) SIGWINCH   29) SIGIO       30) SIGPWR
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
root@localhost:/#
```

发信号? (2) 通过系统调用发送信号

- 系统调用kill 函数

`int kill(int pid, int sig)`

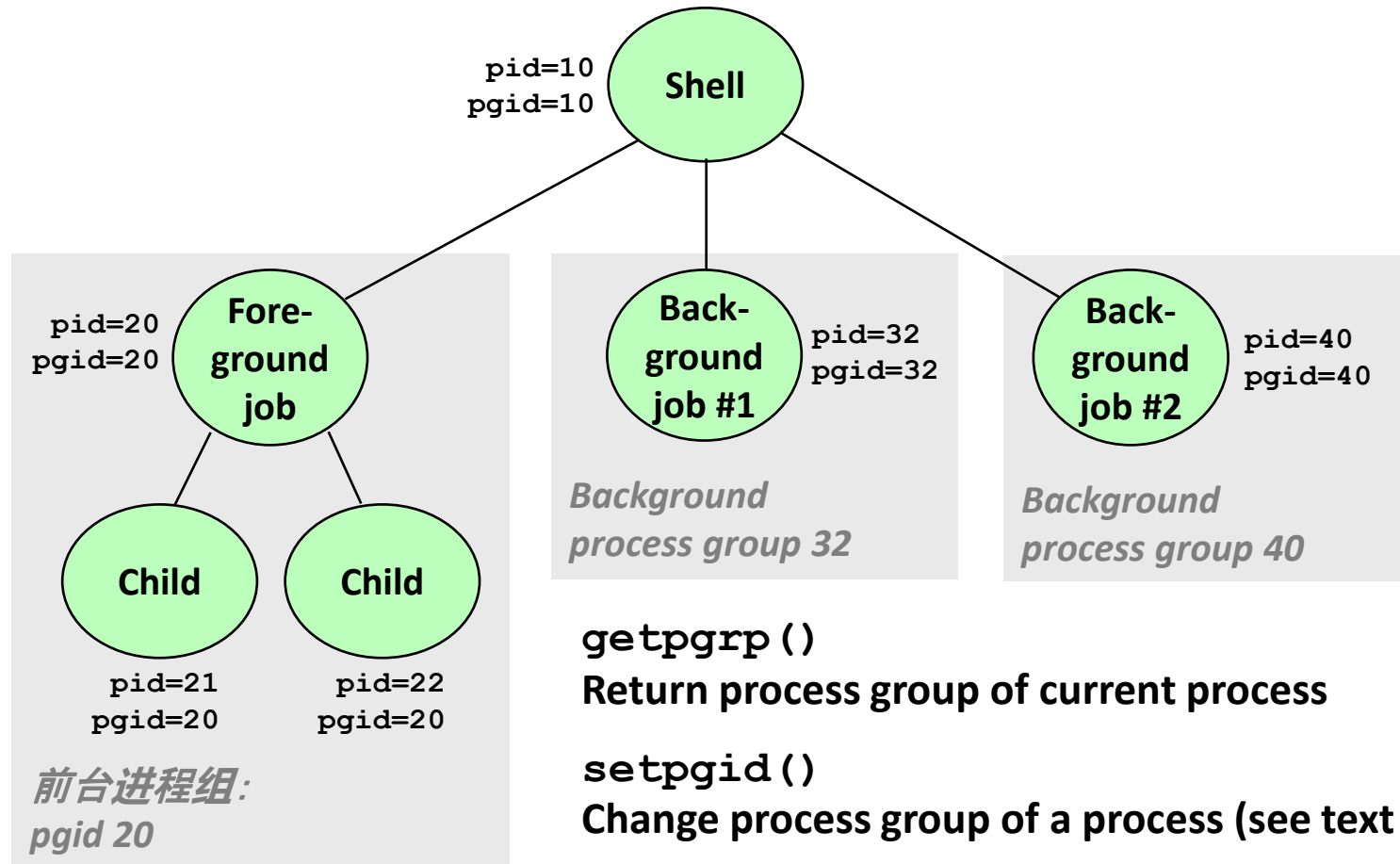
返回值: 0--成功 -1--失败

- kill调用分几种情况

- 当pid>0时, 向指定的进程发信号
- 当pid=0时, 向与本进程同组的所有进程发信号
- 当pid<0时, 向以-pid为组长的所有进程发信号
- 当sig=0时, 则信号根本就没有发送, 但可据此判断一个已知PID的进程是否仍然运行
 - » kill(pid, 0); 如果函数返回值为-1就可根据errno判断: errno=ESRCH说明不存在pid进程

概念: Process Groups

- 每个进程属于一个进程组



进程可以设置对信号的处理方式

- 进程对到达的信号可以在下列处理中选取一种:
 - 设置为缺省处理方式
 - `signal(SIGINT,SIG_DFL);`
 - 忽略该信号
 - » `signal(SIGINT,SIG_IGN);`
 - 在执行了这个调用后，进程就不再收到SIGINT信号
 - 注意：被忽略了的信号作为进程的一种属性会被它的子进程所继承
 - 捕捉该信号，并定义相关的handler
 - » 用户在自己的程序中事先定义好一个函数，当信号发生后就去执行这一函数

定义一个信号handler

- 信号被捕捉并由一个用户函数来处理
- 信号到达时，这个函数将被调用来处理那个信号

```
#include <sys/signal.h>
```

```
sig_handler(int sig)
```

```
{  
    printf("HELLO! Signal %d caught.\n",sig);  
}
```

```
main()
```

```
{  
    signal(SIGINT, sig_handler);  
    signal(SIGQUIT, sig_handler);  
    for(;;) sleep(500);  
}
```

- 注意：信号处理有缺省方式，可以重新为它定义handler，也可以定义为忽略该信号，不处理。

SIGTERM 软件终止信号。用kill命令时产生

SIGHUP 挂断。当从注册shell中logout时，

同一进程组的所有进程都收到SIGHUP

SIGINT 中断。用户按Del键或Ctrl-C键时产生

SIGQUIT 退出。按Ctrl-\时产生，产生core文件

SIGALRM 闹钟信号。计时器时间到，与alarm()有关

SIGCLD 进程的一个子进程终止。

SIGKILL 无条件终止，该信号不能被捕获或忽略。

SIGUSR1, SIGUSR2 用户定义的信号

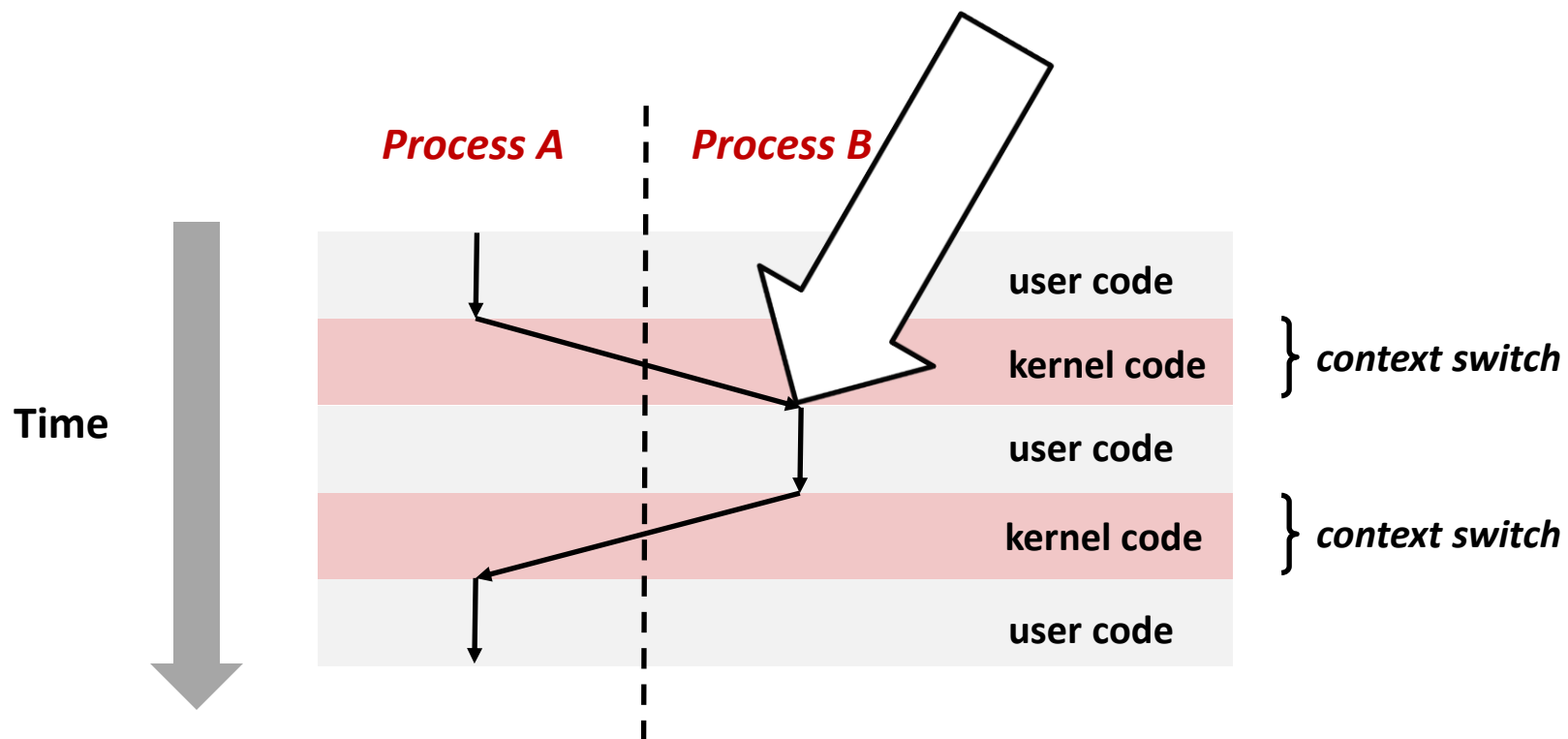
SIGFPE 浮点溢出

SIGILL 非法指令

SIGSEGV 段违例

什么时候处理Signals ?

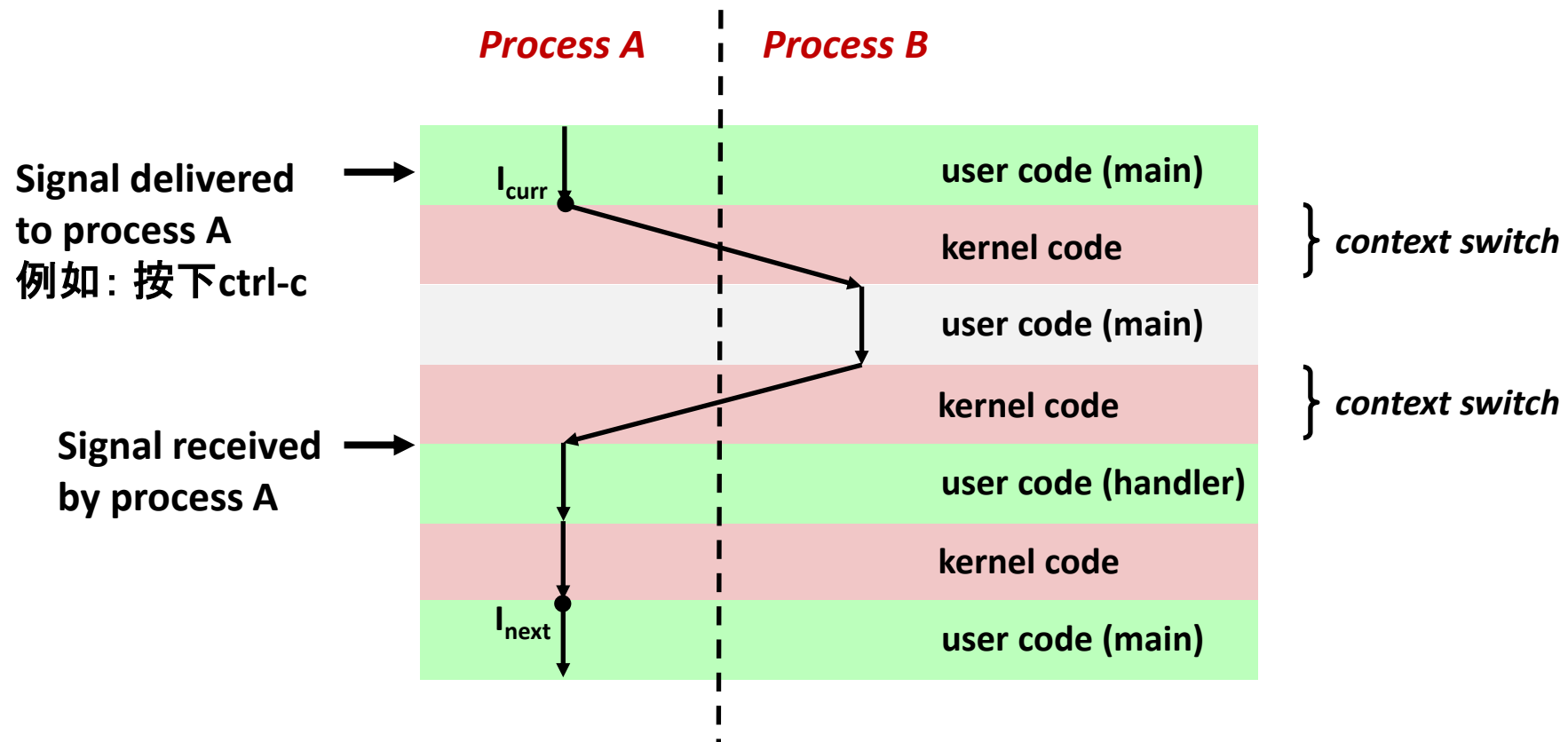
- 内核从中断返回后，准备好将处理器的控制权转换给进程 p



接收信号

- 内核从中断返回后，准备好将处理器的控制权转换给进程 p
- 内核为该进程计算 $\text{pnb} = \text{pending} \& \sim\text{blocked}$
 - The set of pending nonblocked signals for process p
- **If** ($\text{pnb} == 0$)
 - Pass control to next instruction in the logical flow for p
- **Else**
 - Choose least **nonzero bit** k in pnb and force process p to **receive** signal k
 - The receipt of the signal triggers some **action** by p
 - Repeat for all nonzero k in pnb
 - Pass control to next instruction in logical flow for p

Another View of Signal Handlers as Concurrent Flows



本节内容

- **文件抽象** The File Abstraction
 - High-Level File I/O: Streams
 - Low-Level File I/O: File Descriptors
- **进程间的通信** Interprocess Communication” (IPC)
 - 管道 pipe
 - 套接字 socket

Unix/POSIX Idea: Everything is a “File”

- **Identical interface** for:
 - 外部存储中的文件 Files on disk
 - 外部设备 Devices (terminals, printers, etc.)
 - 网络 Networking (sockets)
 - 进程间的通信 Local interprocess communication (pipes, sockets)
- 标准、统一的系统调用接口：
- **open()**, **read()**, **write()**, and **close()**
- Additional: **ioctl()** : 一些没办法归类的函数（例如弹出光驱）

文件系统抽象

- File **文件**

- Named collection of data in a file system
- **数据** File data: sequence of bytes
 - » Could be text, binary, serialized objects, ...
- **元数据** File Metadata: information about the file
 - » Size, Modification Time, Owner, Security info, Access control

- Directory **目录**

- “Folder” containing files & directories
- Hierarchical (graphical) naming
 - » Path through the directory graph
 - » Uniquely identifies a file or directory
 - /home/ff/public_html/index.html
- Links and Volumes (later)

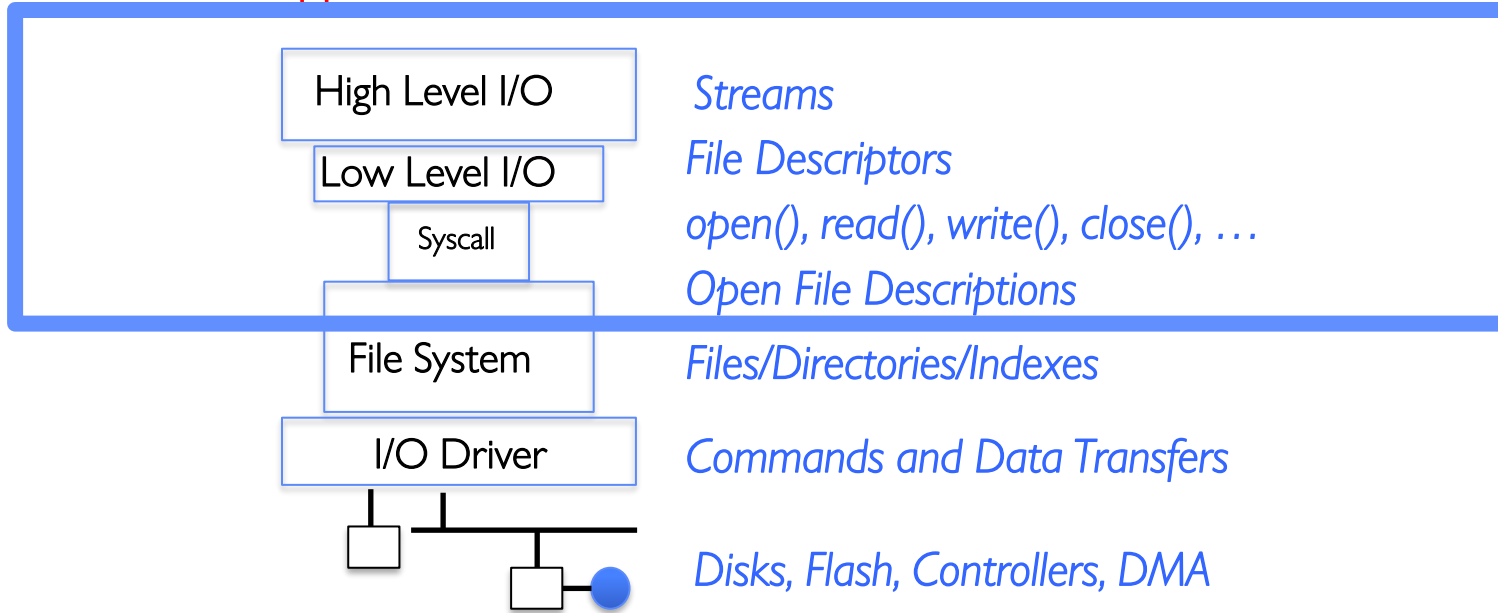
Connecting Processes, File Systems, and Users

- **每一个进程都具有的属性:** *current working directory (CWD)* **当前工作目录**
 - Can be set with system call:
`int chdir(const char *path); //change CWD`
- **绝对路径忽略CWD**
 - `/home/john/cs2302`
- **相对路径与 CWD 有关**
 - `index.html`, `./index.html`
 - » Refers to `index.html` in current working directory
 - `../index.html`
 - » Refers to `index.html` in parent of current working directory
 - `~/index.html`, `~cs2302/index.html`
 - » Refers to `index.html` in the home directory

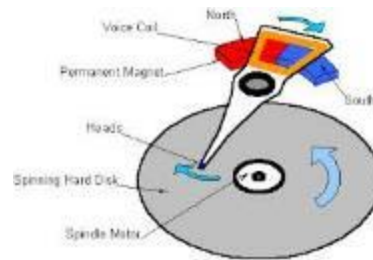
process state
process number
program counter
registers
memory limits
list of open files
...

I/O and Storage Layers

Application / Service



Focus of today's lecture




C High-Level File API – Streams

- Operates on “streams” – unformatted sequences of bytes (with text or binary data), with a position:



```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```



Mode Text	Binary	Descriptions
r	rb	Open existing file for reading
w	wb	Open for writing; created if does not exist
a	ab	Open for appending; created if does not exist
r+	rb+	Open existing file for reading & writing.
w+	wb+	Open for reading & writing; truncated to zero if exists, create otherwise
a+	ab+	Open for reading & writing. Created if does not exist. Read from beginning, write as append

- Open stream represented by **pointer** to a **FILE** data structure
 - Error reported by returning a NULL pointer

What's in a FILE?

- What's in the **FILE*** returned by **fopen**?
 - File descriptor (from call to **open**) <= Need this to interface with the kernel!
 - Buffer (array)
 - Lock (in case multiple threads use the **FILE** concurrently)
- Of course, there's other stuff in a **FILE** too...
- ... but this is useful model to have

C API Standard Streams – `stdio.h`

- Three predefined streams are opened implicitly when the program is executed.
 - `FILE *stdin` – normal source of input, can be redirected
 - `FILE *stdout` – normal source of output, can too
 - `FILE *stderr` – diagnostics and errors
- `STDIN / STDOUT` enable composition in Unix
- All can be redirected
 - `cat hello.txt | grep "World!"`
 - **`cat`'s `stdout`** goes to **`grep`'s `stdin`**

C High-Level File API

// character oriented 面向字符

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 面向字符块

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 格式化函数

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);
}
```

C Streams: Block-by-Block I/O

```
#define BUFFER_SIZE 1024
int main(void) {
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    char buffer[BUFFER_SIZE];
    size_t length;
    length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
    while (length > 0) {
        fwrite(buffer, length, sizeof(char), output);
        length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
    }
    fclose(input);
    fclose(output);
}
```


Low-Level File I/O: The RAW system-call interface

```
#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)
```

Bit vector of:

- Access modes (Rd, Wr, ...)
- Open Flags (Create, ...)
- Operating modes (Appends, ...)

Bit vector of Permission Bits:

- User|Group|Other X R|W|X

- Integer return from **open()** is a *file descriptor*
 - Error indicated by return < 0: the global **errno** variable set with error (see man pages)
- Operations on *file descriptors*:
 - Open system call created an *open file description* entry in system-wide table of open files
 - *Open file description* object in the kernel represents an instance of an open file
 - Why give user an integer instead of a pointer to the file description in kernel?

Low-Level File API

- Read data from open file using file descriptor:

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

- Reads up to **maxsize** bytes – **might actually read less!**
- returns bytes read, 0 => EOF, -1 => error

- Write data to open file using file descriptor

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

- returns number of bytes written

- Reposition file offset within kernel (this is independent of any position held by high-level FILE descriptor for this file!

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

High-Level vs. Low-Level File API

- Streams are buffered in user memory:
`printf("Beginning of line ");`
`sleep(10); // sleep for 10 seconds`
`printf("and end of line\n");`

Prints out everything at once

- Operations on file descriptors are visible immediately
`write(STDOUT_FILENO, "Beginning of line ", 18);`
`sleep(10);`
`write("and end of line \n", 16);`

Outputs "Beginning of line" 10 seconds earlier than "and end of line"

Example

```
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);
```

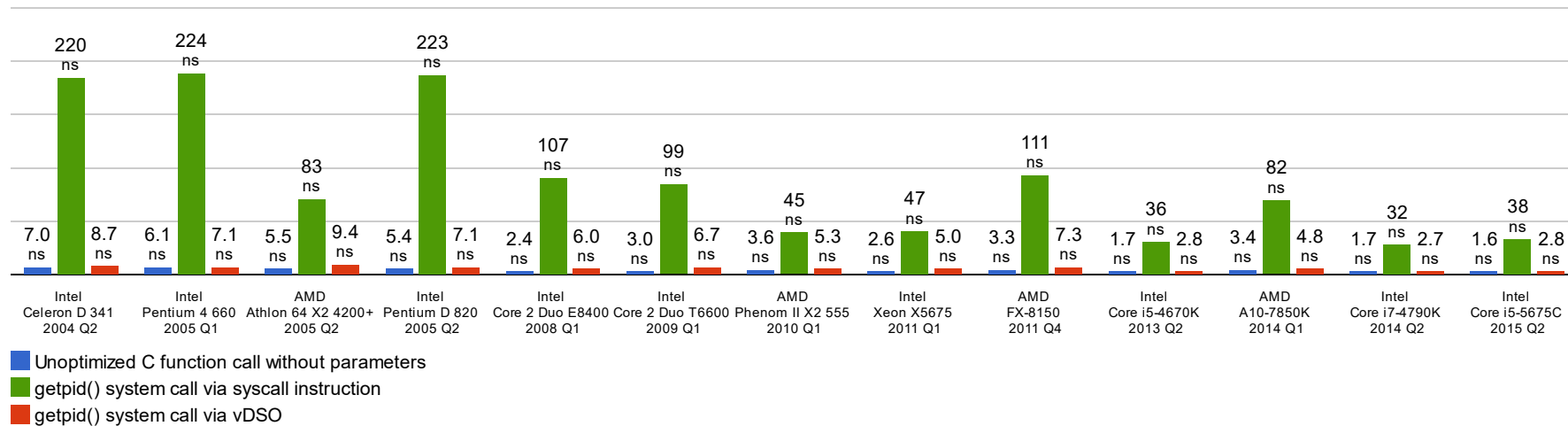
- The call to fread might see the latest write 'b'
- Or it might miss it and see end of file (in which case **x** will remain 'c')

Example

```
char x = 'c';  
FILE* f1 = fopen("file.txt", "wb");  
fwrite("b", sizeof(char), 1, f1);  
fflush(f1);  
FILE* f2 = fopen("file.txt", "rb");  
fread(&x, sizeof(char), 1, f2);
```

- Now, the call to fread will definitely see the latest write 'b'

Why Buffer in Userspace? Overhead!



- Syscalls are 25x more expensive than function calls (~100 ns)
 - This example about special shared-memory interface to the **getpid()** functionality, but point is the same!
- **read/write** a file byte by byte? Max throughput of ~10MB/second
- With **fgetc**? Keeps up with your SSD

Avoid Mixing FILE* and File Descriptors

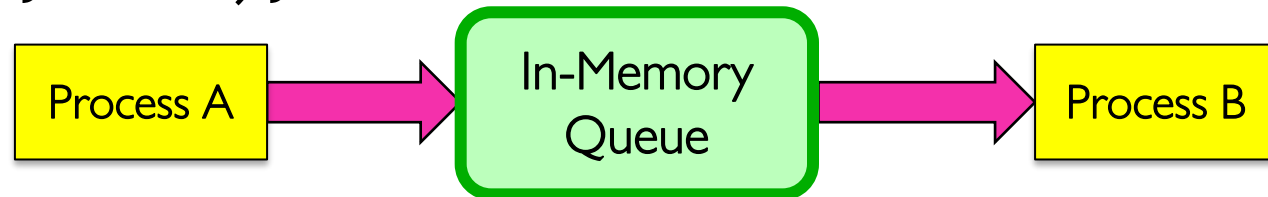
```
char x[10];  
char y[10];  
FILE* f = fopen("foo.txt", "rb");  
int fd = fileno(f);  
fread(x, 10, 1, f); // read 10 bytes from f  
read(fd, y, 10); // assumes that this returns data starting at offset 10
```

- 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!

本节内容

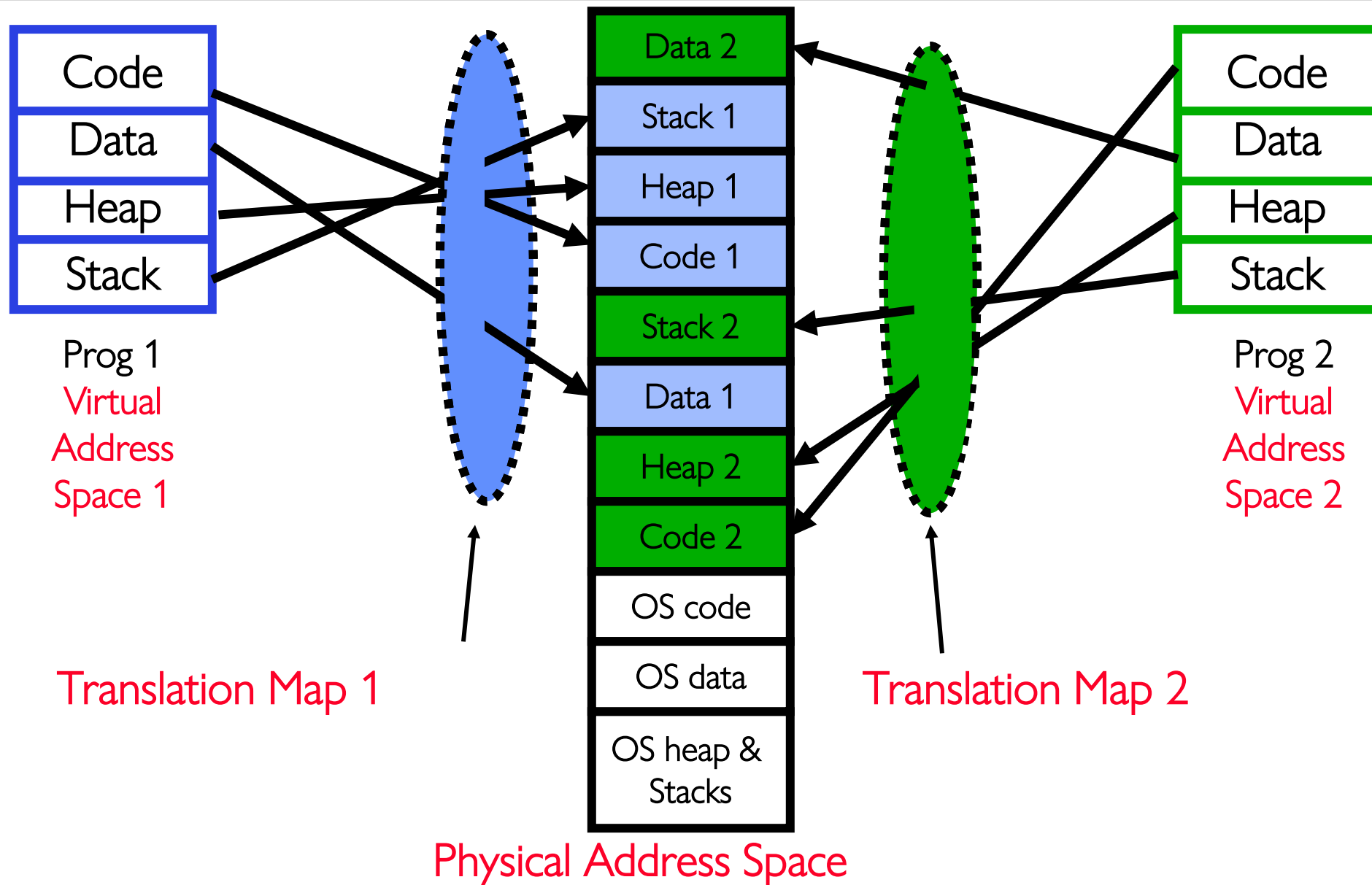
- 文件抽象 The File Abstraction
 - High-Level File I/O: Streams
 - Low-Level File I/O: File Descriptors
- 进程间的通信 Interprocess Communication” (IPC)
 - 管道 pipe
 - 套接字 socket

```
write(wfd, wbuf, wlen);
```



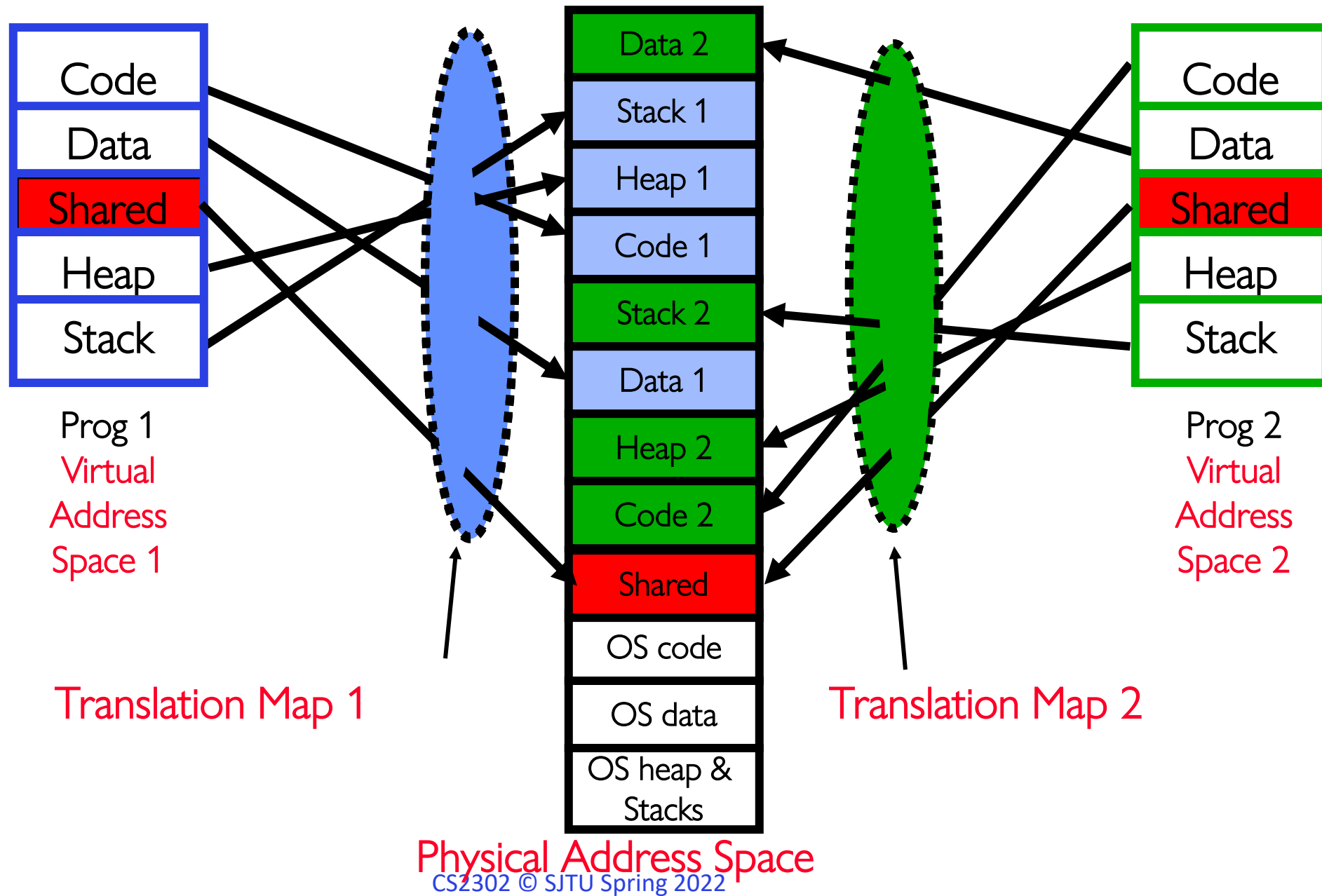
```
n = read(rfd, rbuf, rmax);
```


Recall: Processes Protected from each other



Shared Memory: Better Option?

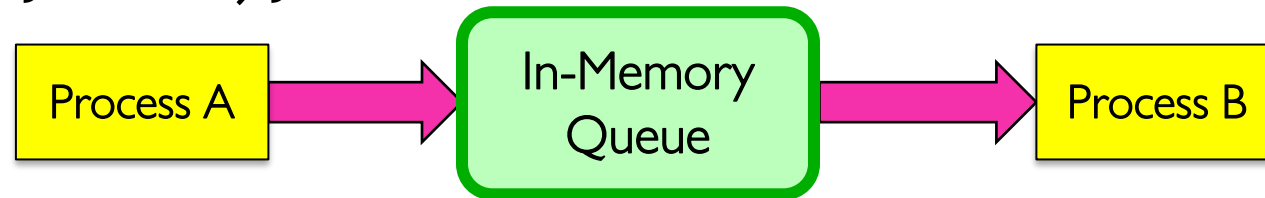
Topic for another day!



Communication Between Processes (Another Option)

- Suppose we ask Kernel to help?
 - Consider an in-memory queue
 - Accessed via system calls (for security reasons):

```
write(wfd, wbuf, wlen);
```

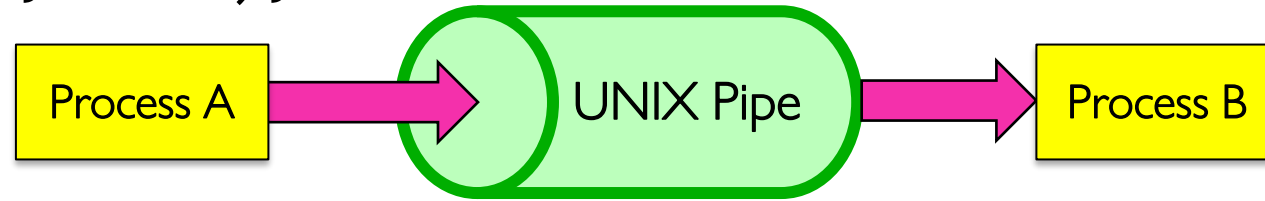


```
n = read(rfd, rbuf, rmax);
```

- Data written by A is held in memory until B reads it
 - 使用和文件相同的接口!
 - Internally more efficient, since nothing goes to disk

One example of this pattern: POSIX/Unix PIPE

```
write(wfd, wbuf, wlen);
```



```
n = read(rfd, rbuf, rmax);
```

- Memory Buffer is finite:
 - If producer (A) tries to write when buffer full, it *blocks* (Put sleep until space)
 - If consumer (B) tries to read when buffer empty, it *blocks* (Put to sleep until data)

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

- Allocates two new file descriptors in the process
- 相当于分配了两个文件描述符
- Writes to `fileds[1]` read from `fileds[0]`
- Implemented as a fixed-size queue

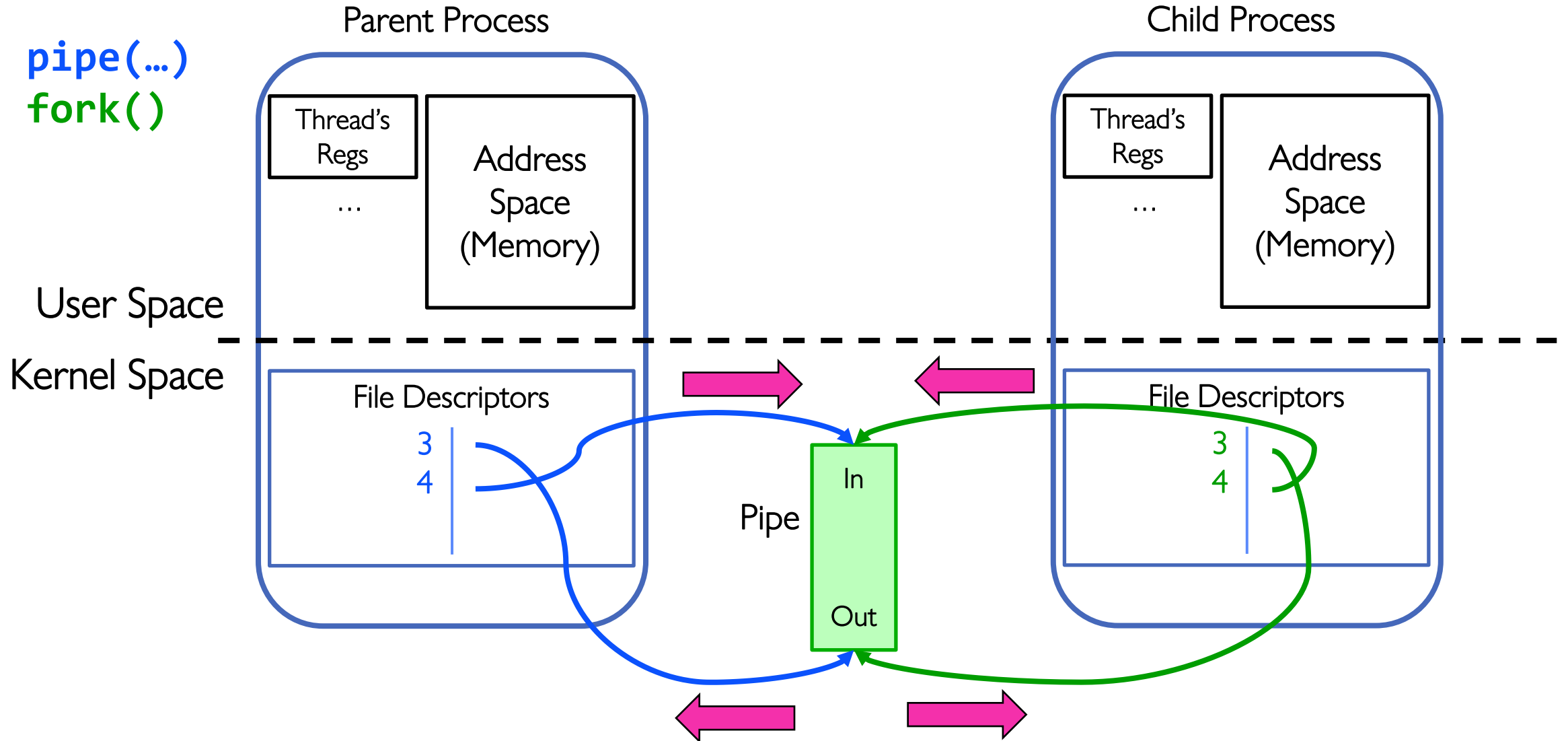
Single-Process Pipe Example

```
#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]);
}
```

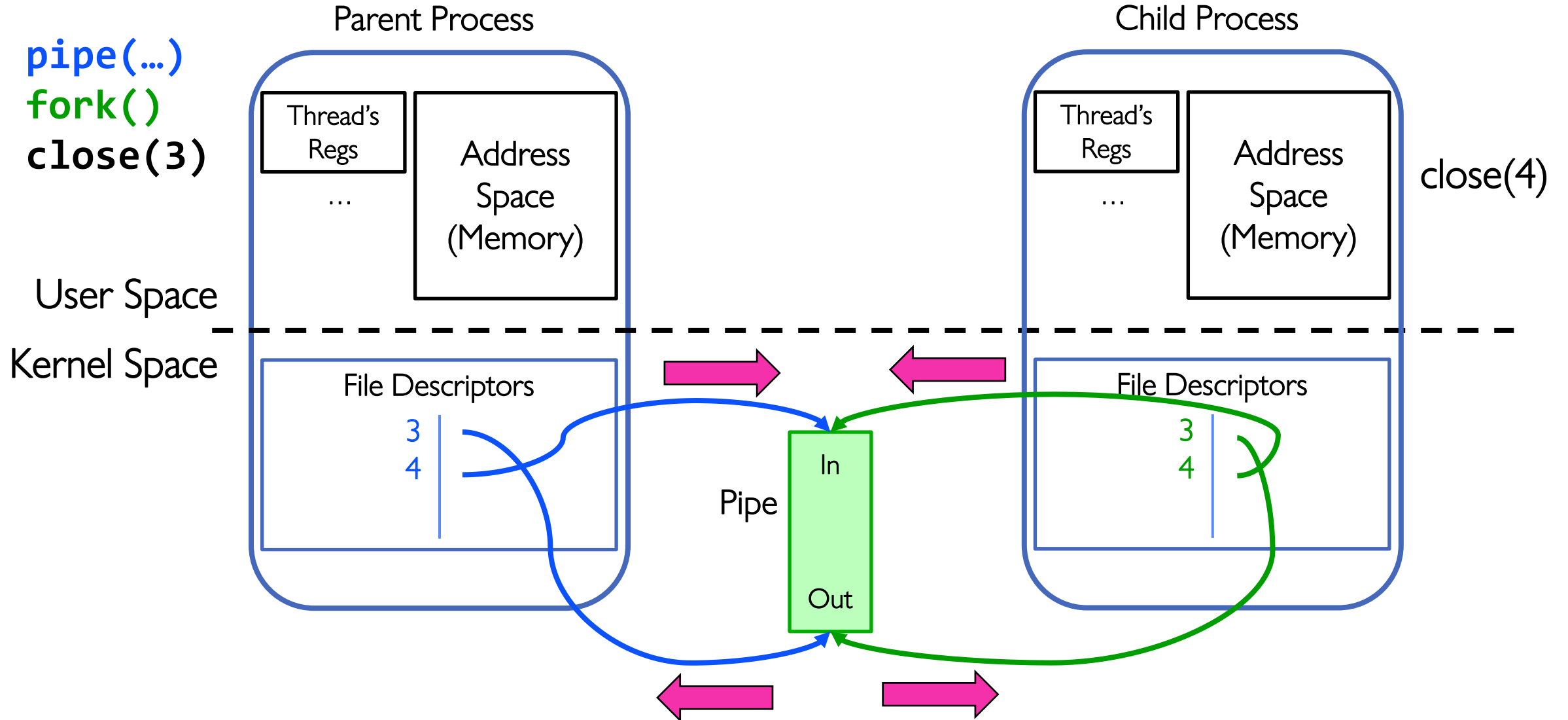
Pipes Between Processes



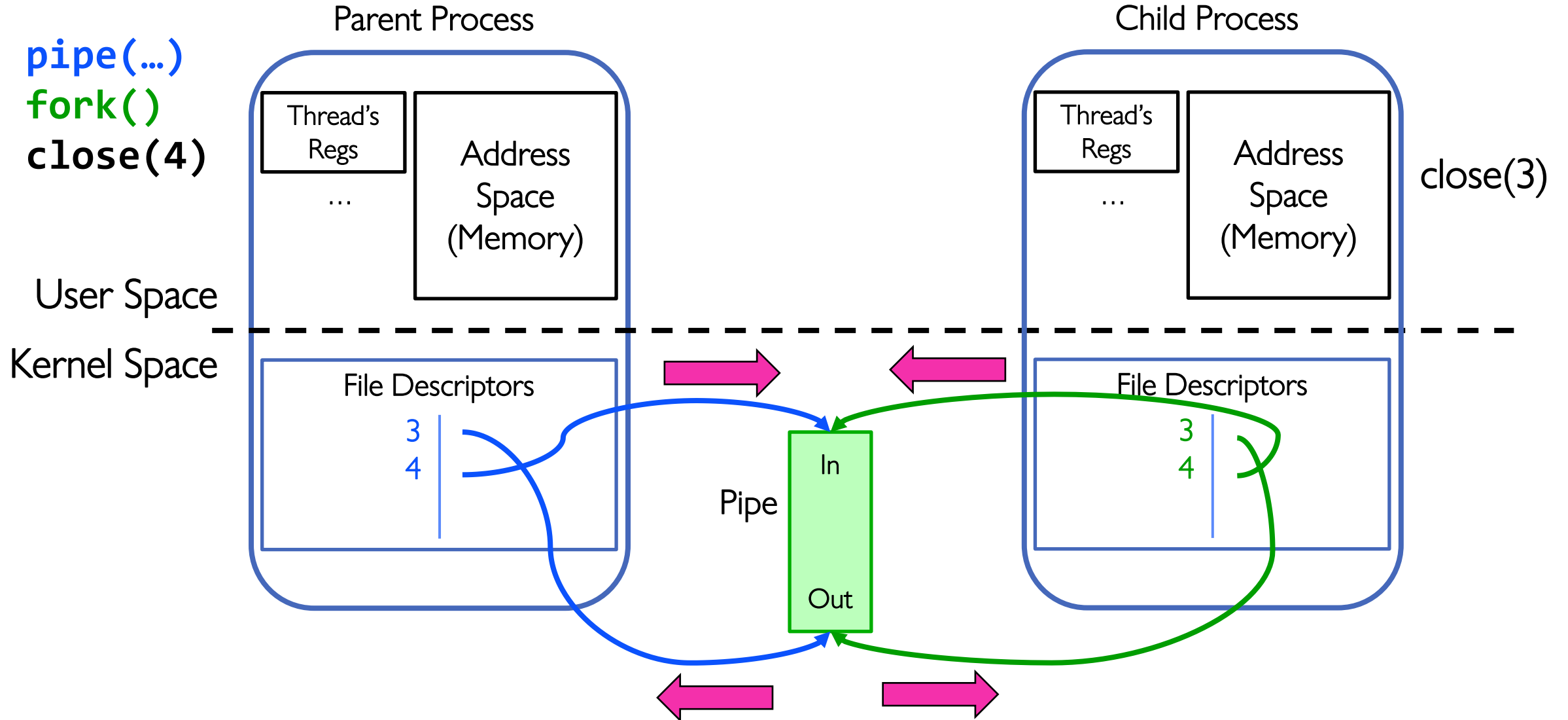
Inter-Process Communication (IPC): Parent \Rightarrow Child

```
// continuing from earlier
pid_t pid = fork();
if (pid < 0) {
    fprintf (stderr, "Fork failed.\n");
    return EXIT_FAILURE;
}
if (pid != 0) {
    ssize_t writelen = write(pipe_fd[1], msg, msglen);
    printf("Parent: %s [%ld, %ld]\n", msg, msglen, writelen);
    close(pipe_fd[0]);
} else {
    ssize_t readlen = read(pipe_fd[0], buf, BUFSIZE);
    printf("Child Rcvd: %s [%ld]\n", msg, readlen);
    close(pipe_fd[1]);
}
```

Channel from Parent \Rightarrow Child



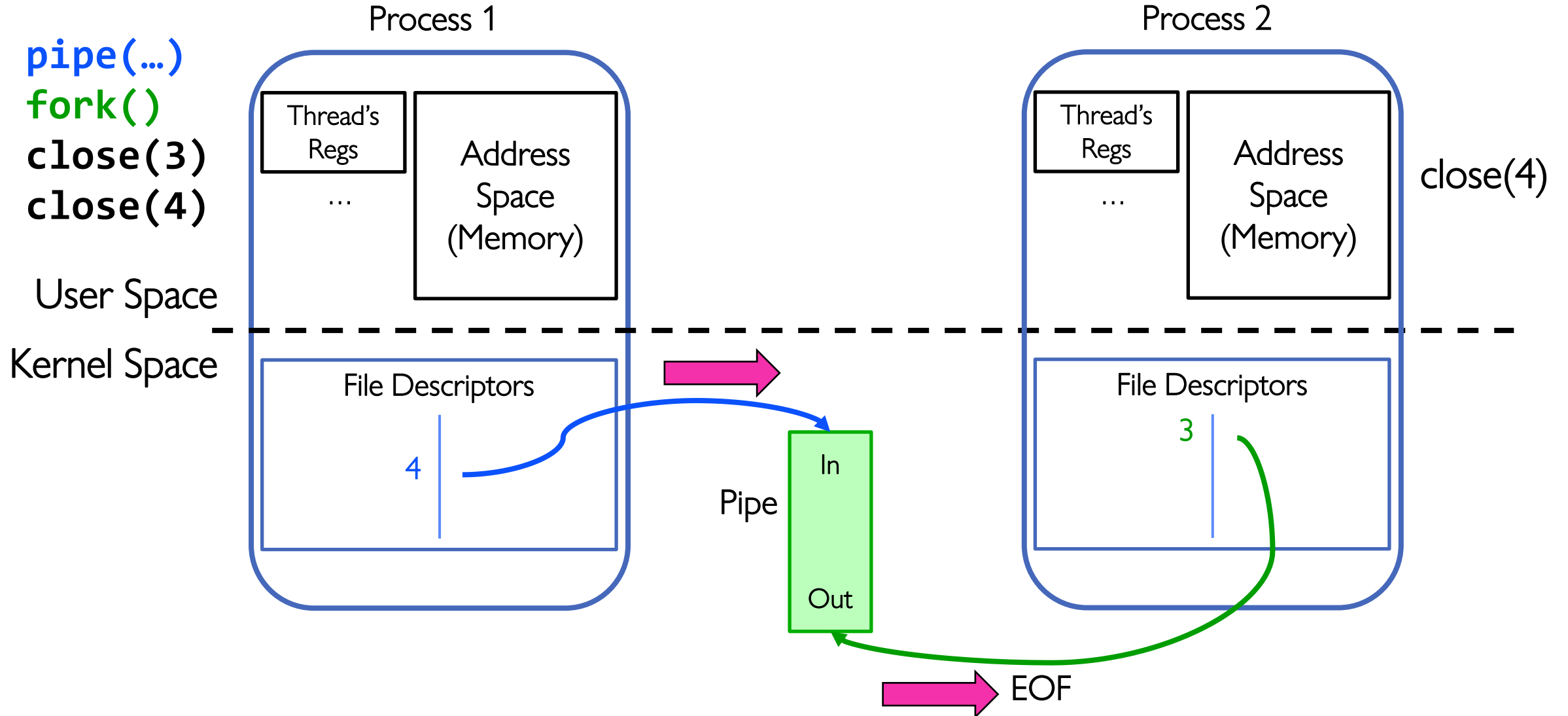
Instead: Channel from Child \Rightarrow Parent



When do we get EOF on a pipe?

- 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

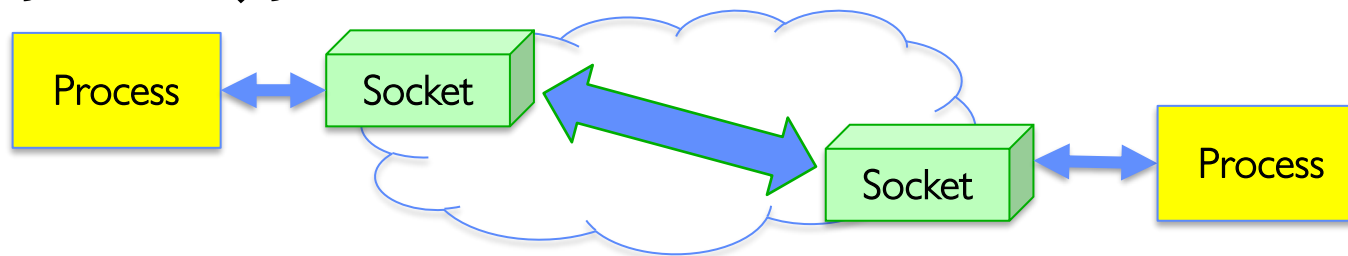
EOF on a Pipe



本节内容

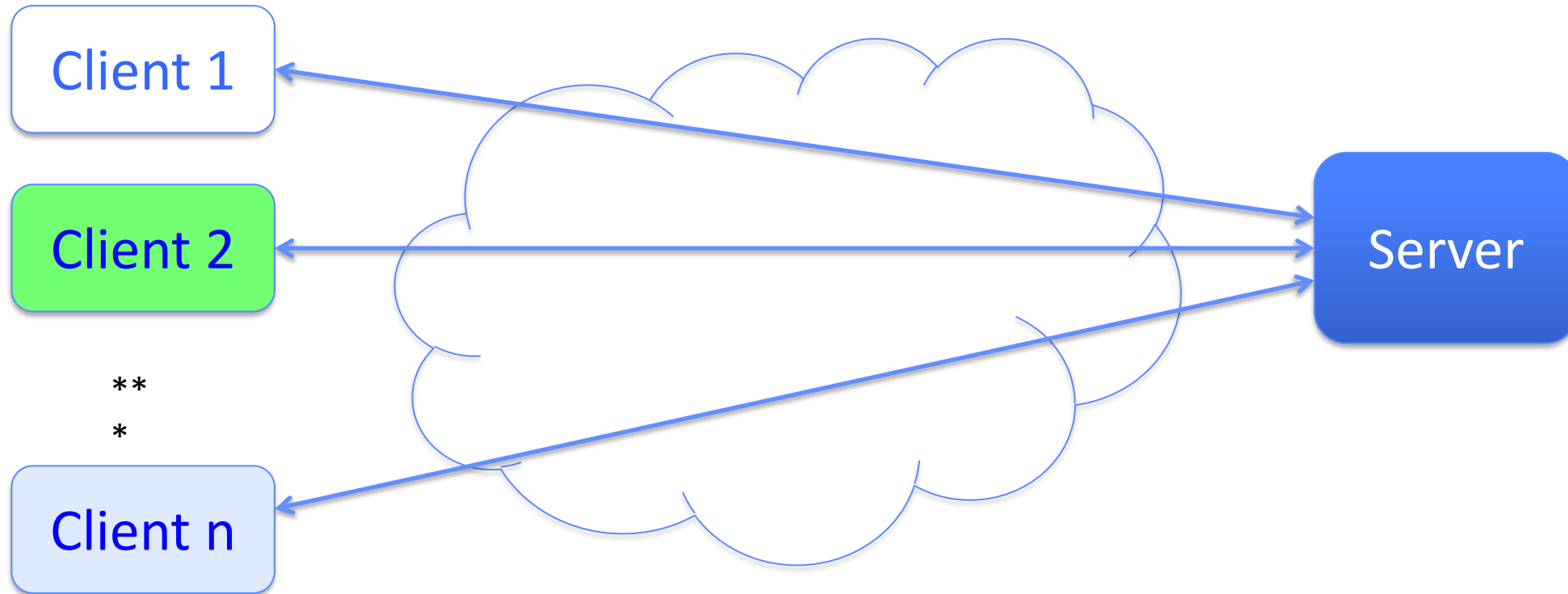
- 文件抽象 The File Abstraction
 - High-Level File I/O: Streams
 - Low-Level File I/O: File Descriptors
- 进程间的通信 Interprocess Communication” (IPC)
 - 管道 pipe
 - 套接字 socket

`write(wfd, wbuf, wlen);`



`n = read(rfd, rbuf, rmax);`

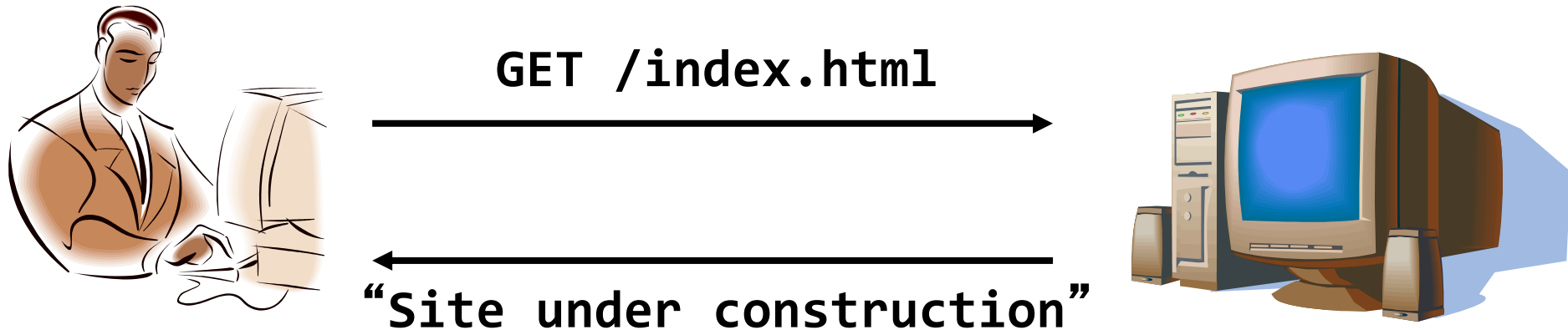
Client-Server Protocols: Cross-Network IPC



- Many clients accessing a common server
- File servers, www, FTP, databases

Client-Server Communication

- Client is “sometimes on”
 - Sends the server requests for services when interested
 - E.g., Web browser on laptop/phone
 - Doesn’t communicate directly with other clients
 - Needs to know server’s address
- Server is “always on”
 - Services requests from many clients
 - E.g., Web server for `www.cnn.com`
 - Doesn’t initiate contact with clients
 - Needs a fixed, well-known address



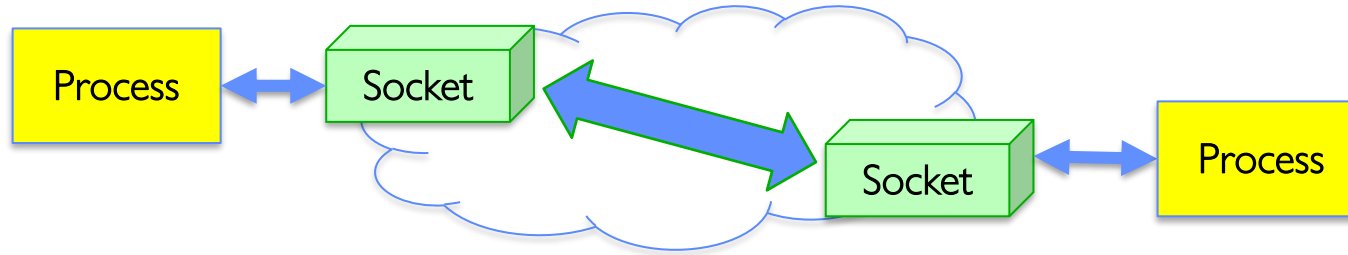
What is a Network Connection?

- Bidirectional *stream* of bytes between two processes on possibly different machines
 - For now, we are discussing “TCP Connections”
- Abstractly, a connection between two endpoints A and B consists of:
 - A queue (bounded buffer) for data sent from A to B
 - A queue (bounded buffer) for data sent from B to A

The Socket Abstraction: Endpoint for Communication

- **Key Idea:** Communication across the world looks like File I/O

`write(wfd, wbuf, wlen);`



`n = read(rfd, rbuf, rmax);`

- Sockets: Endpoint for Communication
 - Queues to temporarily hold results
- Connection: Two Sockets Connected Over the network \Rightarrow IPC over network!
 - How to **`open()`**?
 - What is the namespace?
 - How are they connected in time?

Sockets: More Details

- **Socket:** An abstraction for one endpoint of a network connection
 - Another mechanism for **inter-process communication**
 - Most operating systems (Linux, Mac OS X, Windows) provide this, even if they don't copy rest of UNIX I/O
 - Standardized by POSIX
- First introduced in 4.2 BSD (Berkeley Standard Distribution) Unix
 - This release had some huge benefits (and excitement from potential users)
 - Runners waiting at release time to get release on tape and take to businesses
- Same abstraction for any kind of network
 - Local (within same machine)
 - The Internet (TCP/IP, UDP/IP)
 - Things “no one” uses anymore (OSI, Appletalk, IPX, ...)

Sockets: More Details

- Looks just like a file with a **file descriptor**
 - Corresponds to a network connection (*two* queues)
 - **write** adds to output queue (queue of data destined for other side)
 - **read** removes from it input queue (queue of data destined for this side)
 - Some operations do not work, e.g. **lseek**
- How can we use sockets to support real applications?
 - A bidirectional byte stream isn't useful on its own...
 - May need messaging facility to partition stream into chunks
 - May need RPC facility to translate one environment to another and provide the abstraction of a function call over the network

Simple Example: Echo Server



Simple Example: Echo Server

Client (issues requests)

Server (services requests)

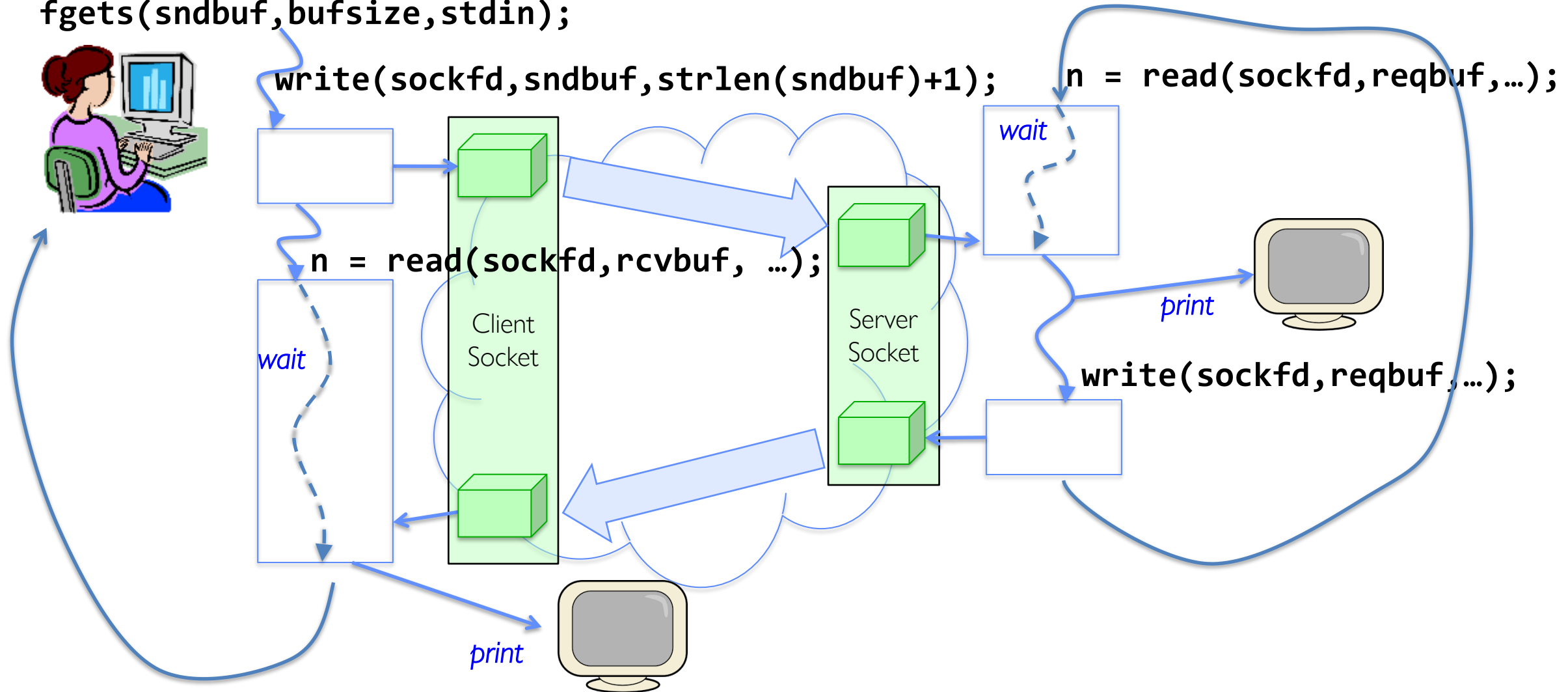
```
fgets(sndbuf, bufsize, stdin);
```

```
write(sockfd, sndbuf, strlen(sndbuf)+1);
```

```
n = read(sockfd, reqbuf, ...);
```

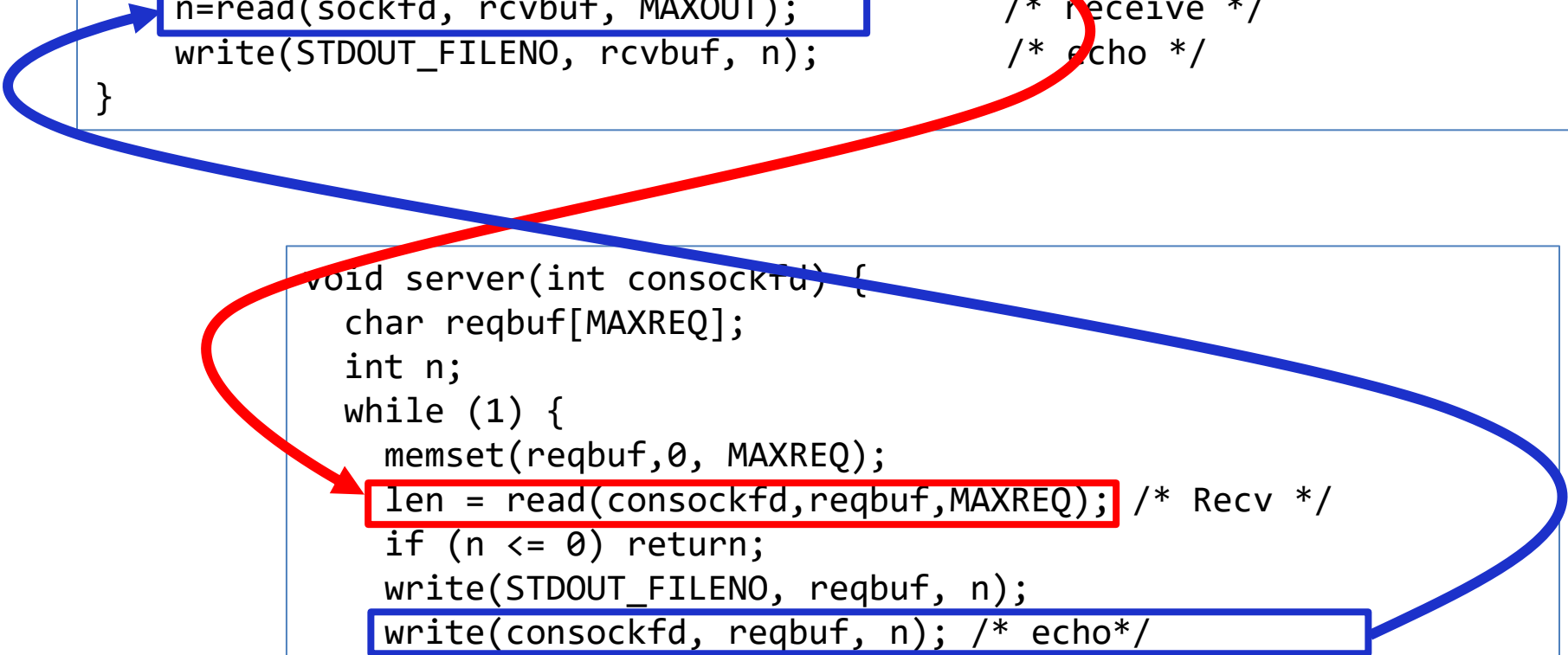
```
n = read(sockfd, rcvbuf, ...);
```

```
write(sockfd, reqbuf, ...);
```



Echo client-server example

```
void client(int sockfd) {
    int n;
    char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
    while (1) {
        fgets(sndbuf, MAXIN, stdin);           /* prompt */
        write(sockfd, sndbuf, strlen(sndbuf)+1); /* send (including null terminator) */
        memset(rcvbuf, 0, MAXOUT);             /* clear */
        n=read(sockfd, rcvbuf, MAXOUT);         /* receive */
        write(STDOUT_FILENO, rcvbuf, n);        /* echo */
    }
}
```



```
void server(int consockfd) {
    char reqbuf[MAXREQ];
    int n;
    while (1) {
        memset(reqbuf, 0, MAXREQ);
        len = read(consockfd, reqbuf, MAXREQ); /* Recv */
        if (n <= 0) return;
        write(STDOUT_FILENO, reqbuf, n);
        write(consockfd, reqbuf, n); /* echo */
    }
}
```

Summary

- POSIX idea: “everything is a file”
- All sorts of I/O managed by open/read/write/close
- Interprocess Communication (IPC)
 - Communication facility between protected environments (i.e. processes)
- Pipes are an abstraction of a single queue
 - One end write-only, another end read-only
 - Used for communication between multiple processes on one machine
 - File descriptors obtained via inheritance
- 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
 - Inheritance of file descriptors on fork() facilitates handling each connection in a separate process
- Both support read/write system calls, just like File I/O