进程和线程

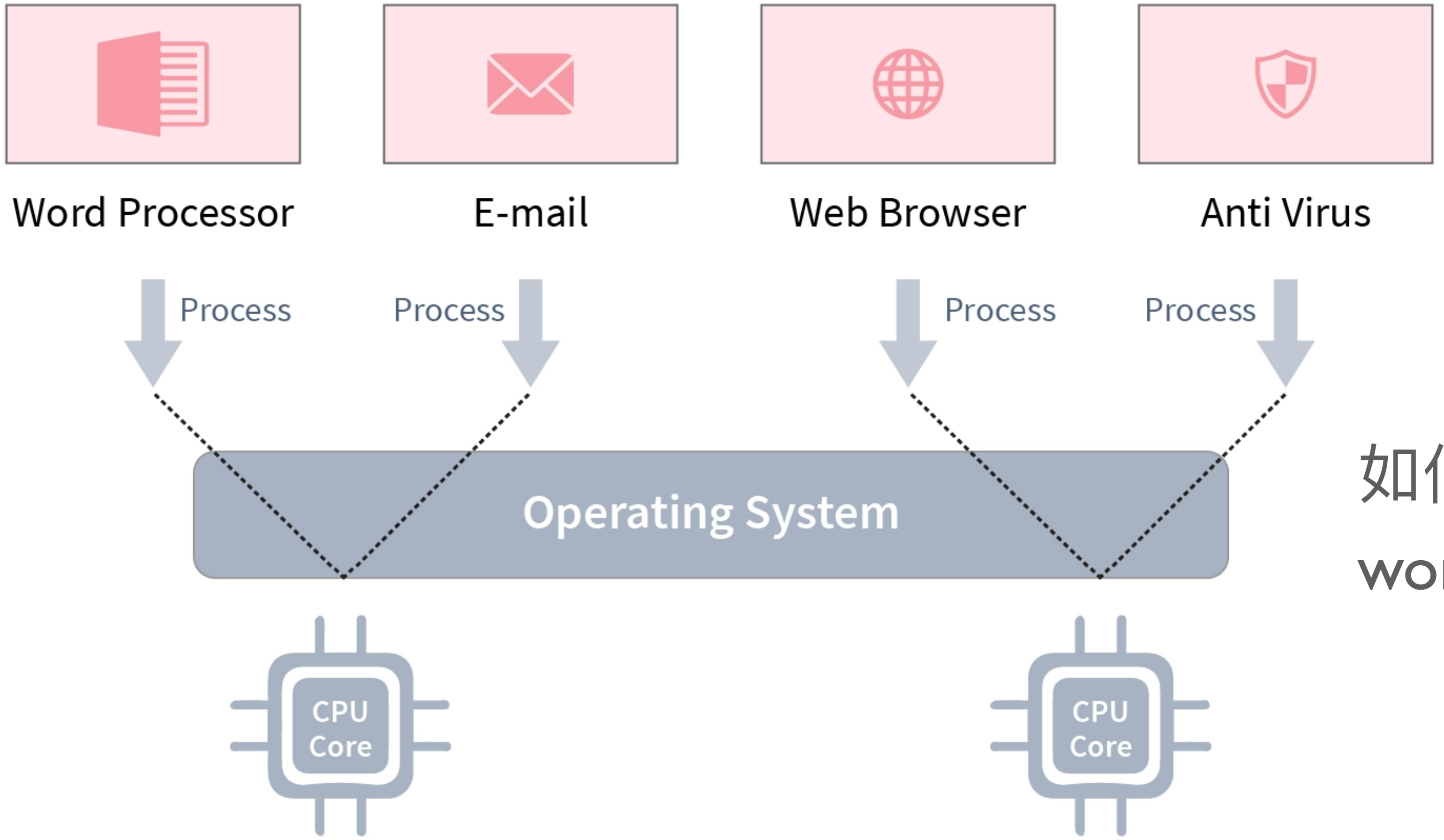
Section 2: Part I

2025

进程

操作系统为正在运行的程序 (a running program) 提供的抽象

- 刻画多道程序 (multi-programming)
- 对 CPU 的分时共享 (time sharing)

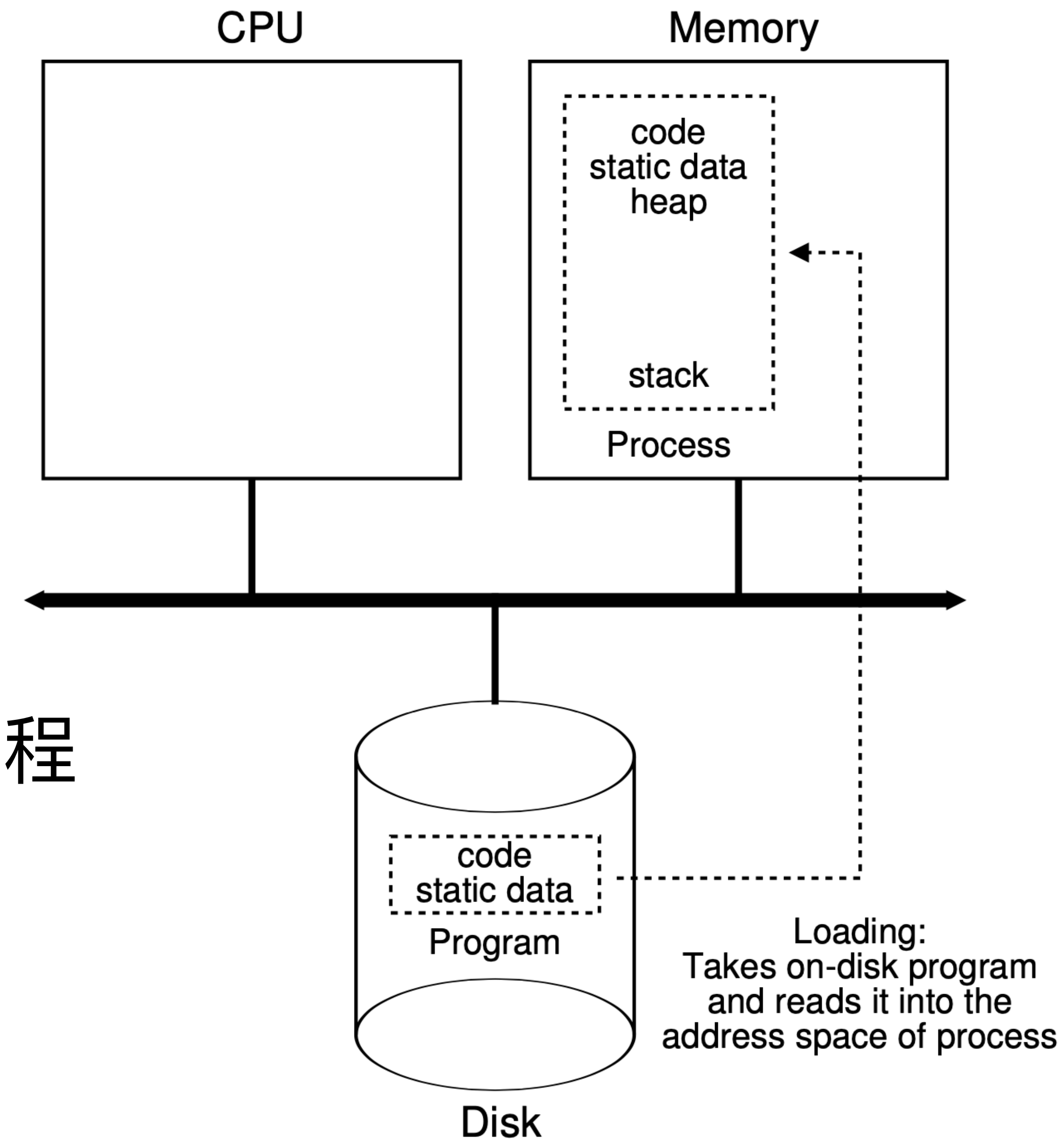


如何对正在运行的多个 hello world 程序进行抽象和管理?

进程

操作系统为正在运行的程序 (a running program) 提供的抽象

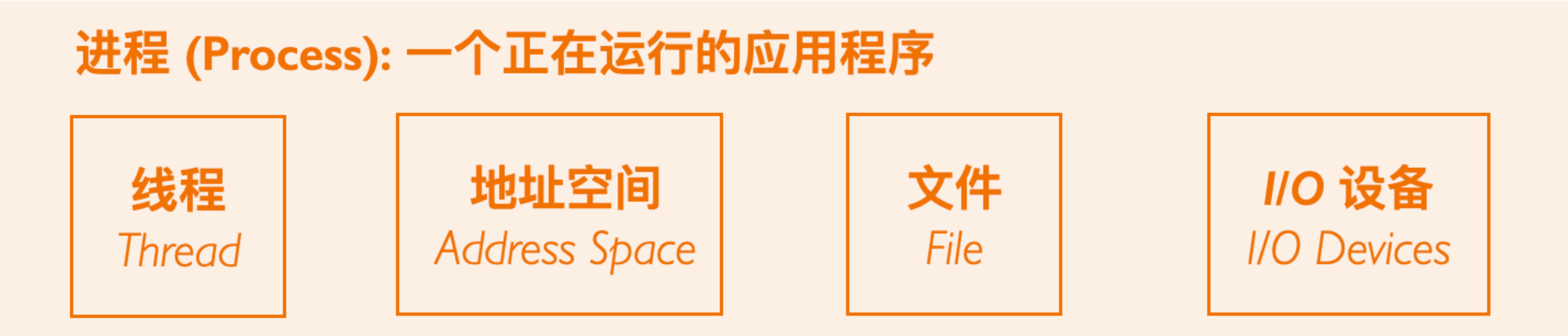
- **静态部分**: 程序运行所需的代码和数据
- **动态部分**: 程序运行期间的状态
- 从程序 (program) 到进程 (process)
  - 将代码和数据加载到内存中
  - 分配内存空间、初始化必要信息
  - 将 CPU 的控制权移交给新创建的进程



进程

一个进程应包含正在执行的程序的所有机器状态

- 寄存器 (Registers): CPU 运行时状态
  program counter, stack pointer, a set of general purpose registers
- 内存 (Memory): 进程可以访问的地址空间
  code, data, stack, and heap
- I/O 信息: 若干资源
  e.g., a list of the files the process currently has open



进程 (Process): 一个正在运行的应用程序

线程 Thread

地址空间 Address Space

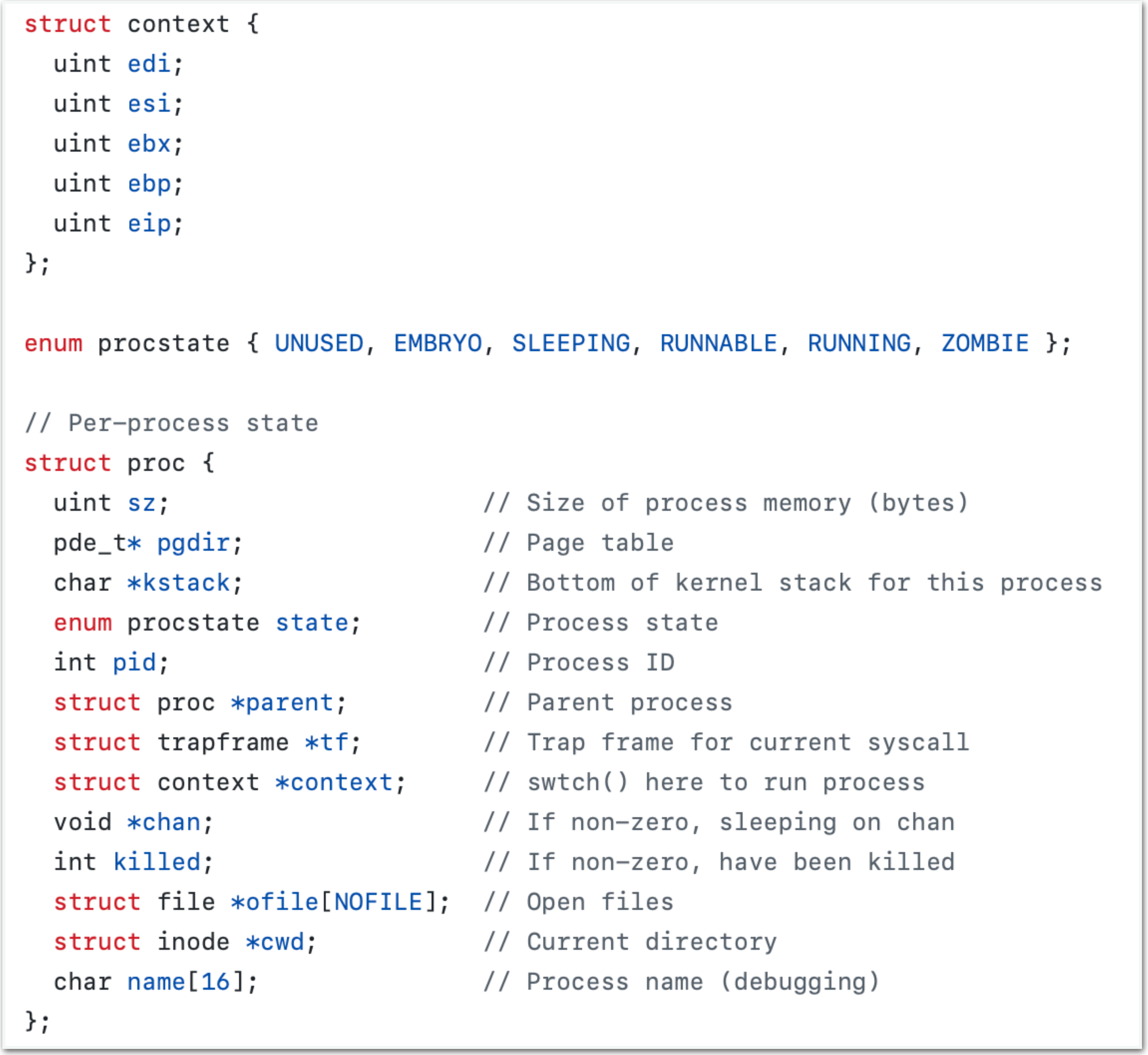
文件 File

I/O 设备 I/O Devices

进程

为了管理系统中的进程，操作系统需要在内核中维护一些数据结构

- 进程控制块 Process Control Block (PCB): 记录关于一个进程的所有相关信息 (Identification, Context, and Management)
  - 进程 ID (Process Identifier, PID)、当前进程的状态
  - CPU 寄存器和调度信息 (e.g., 优先级)
  - 内存管理 (e.g., 物理内存分配情况、地址转换相关信息)
  - I/O 信息 (e.g., 打开文件表)
  - 其它统计信息 (e.g., 已使用 CPU 时间)
- 在进程运行的任何时刻，我们都可以用上述 PCB 中的信息来唯一刻画该进程



```c
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

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 (x86)
proc structure

进程

Linux 系统中提供 `/proc/{PID}` 接口来获取进程相关的信息

- `cmdline`: Command line arguments
- `cwd`: Link to the current working directory
- `environ`:Values of environment variables
- `fd`: Directory, which contains all file descriptors
- `status`: Process status in human readable form
- `maps`: Memory maps to executables and library files
- `mem`: Memory held by this process
- `pagemap`: page table
- …

https://docs.kernel.org/filesystems/proc.html

进程操作

- Create: 创建一个进程以运行某个程序
- Wait: 等待一个进程运行结束
- Destroy: 终止一个进程
- Status: 获取一个进程的相关信息
- Control: 暂停和恢复进程运行等

进程创建相关的 APIs

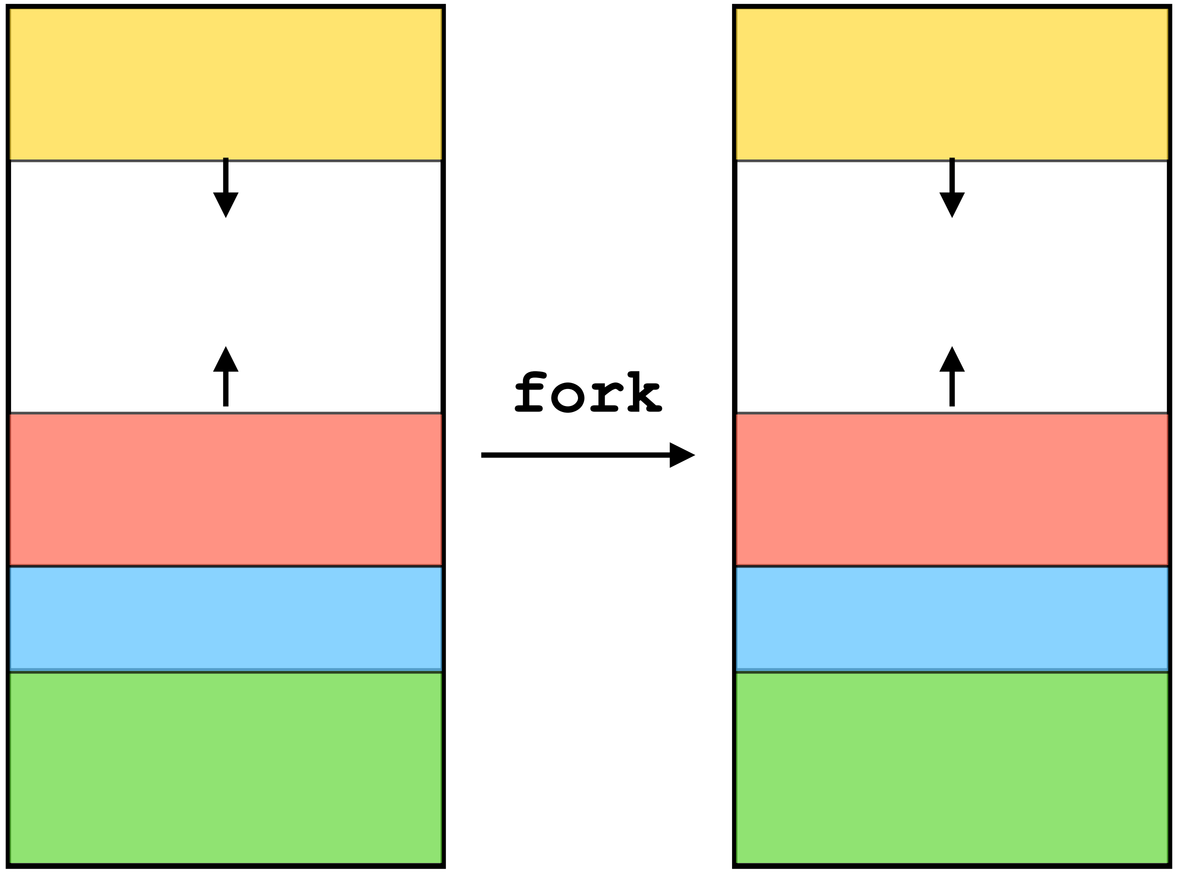
Unix Systems

和进程创建相关的三个系统调用

- `fork()`:创建父进程的完整副本
- `execve()`:将新的可执行代码载入内存并开始运行
- `wait()`:等待创建的进程执行完成

进程创建相关的 APIs

`fork()`



Parent Process
`return P`

Child Process
(PID = P)
`return 0`

创建一个父进程的完整拷贝
(包括内存和 PCB)

- 子进程会继承父进程的 execution context
- 为子进程分配一个不同的 PID
- 子进程和父进程在 `fork()` 返回后各自独立执行后续指令

进程创建相关的 APIs

fork()

- **调用一次、返回两次**
  - 父进程收到新创建进程的 PID 作为返回值
  - 子进程收到 0 作为返回值
  - 创建进程失败时返回 -1 (例如内存不足)
- **相同但分离的地址空间**: 父子进程随后各自拥有独立私有的地址空间
- **共享文件**: 子进程继承父进程已打开文件
- **并发执行**: 父子进程在返回后的执行顺序存在不确定性 (取决于调度器)
  - 父进程可使用 wait() 来等待子进程执行完成

进程创建相关的 APIs

fork()

```
int main() {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    exit(1);
  } else if (rc == 0) {
    // child process goes down this path
    printf(“[%d] child process\n”, (int) getpid());
  } else {
    // parent goes down this path
    printf(“[%d] parent process, rc = %d\n”,
           (int) getpid(), rc);
  }
  return 0;
}
```

进程创建相关的 APIs

wait()

```
int main() {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    exit(1);
  } else if (rc == 0) {
    // child process goes down this path
    printf(“[%d] child process\n”, (int) getpid());
  } else {
    // the wait system call will not return
    // until the child had exited
    int status; // exit status
    int wc = wait(&status);
    printf(“[%d] parent process, rc = %d, status = %d\n”,
        (int) getpid(), rc, WEXITSTATUS(status));
  }
  return 0;
}
```

Quiz

下述代码运行过程中总共创建了几个进程？输出是什么？

```
int count = 0;

int main() {
  fork();
  fork();

  for (int i = 0 ; i < 1000 ; i++)
    count++;

  printf(“%d\n”, count);
  return 0;
}
```

Quiz

运行下述代码可能 (或不可能) 的输出有哪些？

```
int main() {
  if (fork() == 0) {
    printf(“a”);
  } else {
    printf(“b”);
    wait();
  }
  printf(“c”);

  return 0;
}
```

进程创建相关的 APIs

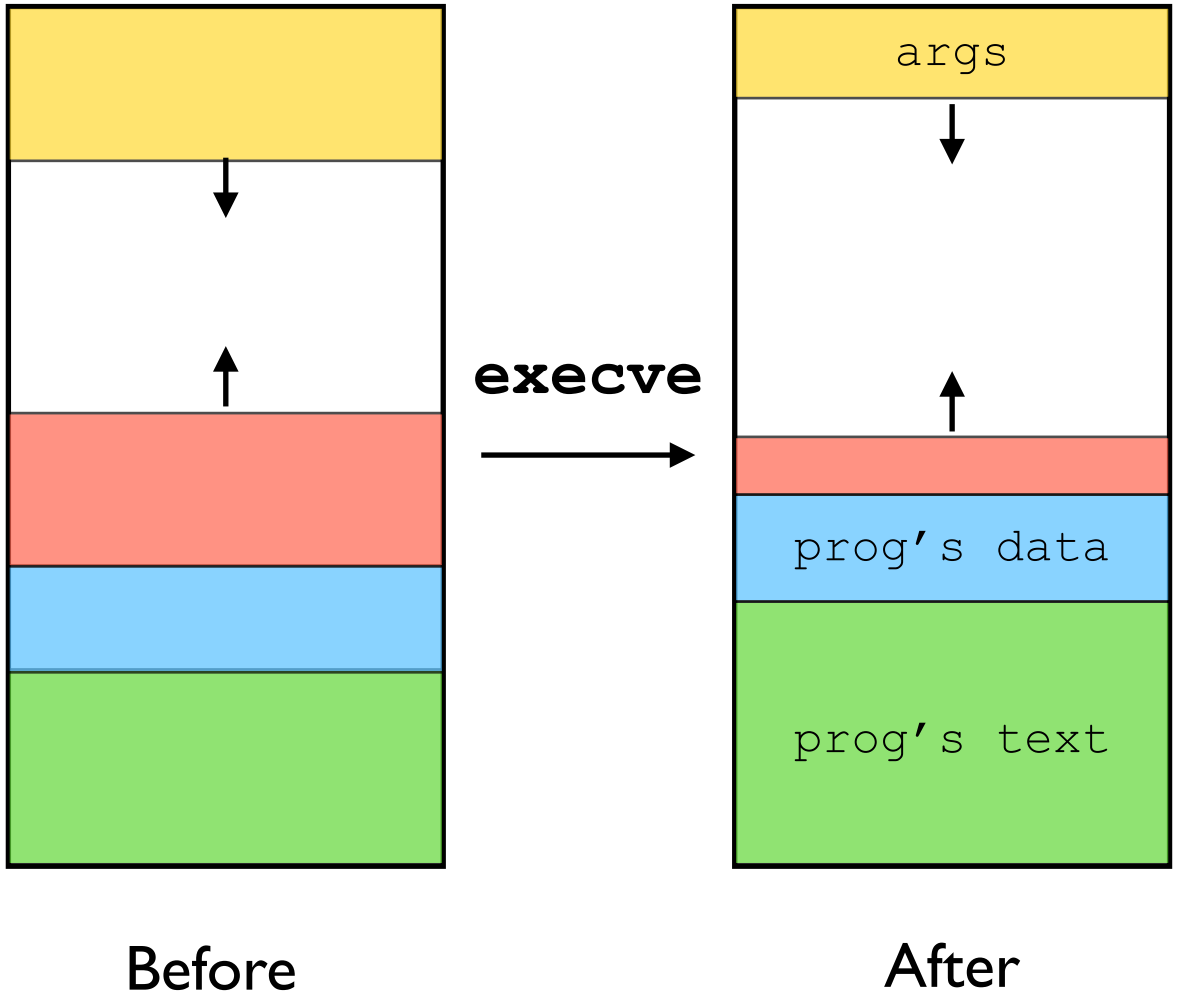
`fork()`

使用 `fork()` 可以帮助我们创建当前运行进程的精确副本

- 子进程依赖父进程的代码来完成某项任务，例如运行同一个可执行文件的不同函数 (e.g., a web browser)
- 亦可用于给进程创建一个 “快照”
  - 主进程 crash 了，启动快照重新执行
- 但是，如果我们想运行一个不同的程序?

进程创建相关的 APIs

execve()



```
int execve(const char *pathname,
           char *const argv[],
           char *const envp[]);
```

加载一个可执行文件并运行

- 使用 pathname 指定的可执行文件重写地址空间
  - 替换数据/代码、初始化堆/栈、设置 PC 为代码段入口点
  - PCB 中相应的信息也会改变
- 使用 envp 作为新进程执行的环境变量
- 并没有创建一个新进程，而是将当前运行的程序转换为另一个程序 **(调用一次、永不返回)**

进程创建相关的 APIs

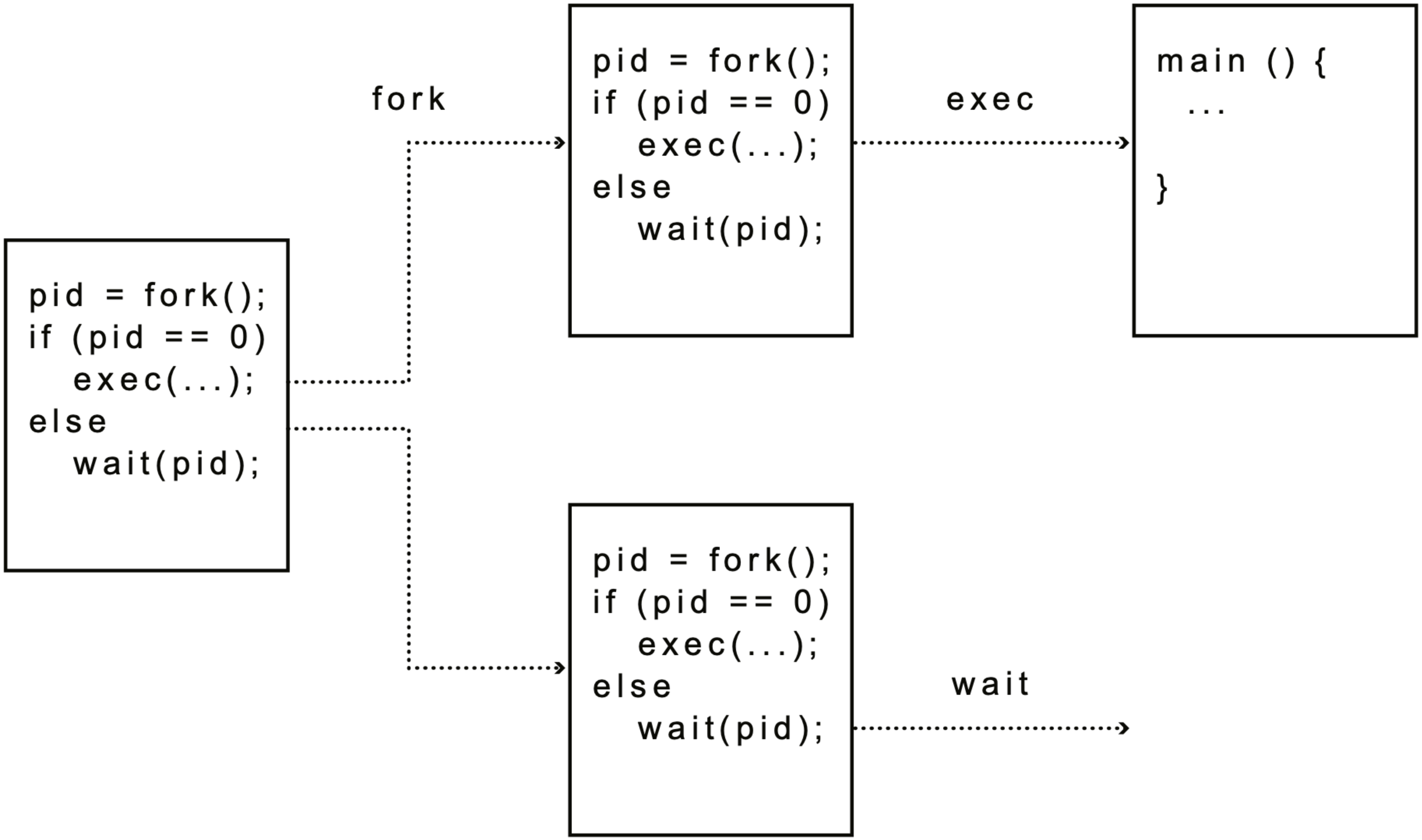
execve()

```
int main() {
  int rc = fork();
  if (rc < 0) {
    exit(1);
  } else if (rc == 0) {
    printf(“[%d] child process\n”, (int) getpid());
    // run a new program
    char *args[] = {“/bin/wc”, “fork.c”, NULL};
    char *envp[] = {“KEY=Value”, NULL};
    execve(args[0], args, envp);

    printf(“this should not print out\n”);
  } else {
    wait(NULL);
    printf(”[%d] parent process\n”, (int) getpid());
  }
  return 0;
}
```

进程创建相关的 APIs

execve ( )



```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

```
main () {
  ...

}
```

```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

进程创建相关的 APIs

execve()

- execve() 是唯一能够 “执行” 程序的系统调用
  - 因此也是一切进程 strace 的第一个系统调用
- libc 中提供了构建于 execve() 之上的更多选择
  - exec[l/v][p][e]

```
int execl(const char *pathname, const char *arg, ...
                /*, (char *) NULL */);
int execlp(const char *file, const char *arg, ...
                /*, (char *) NULL */);
int execle(const char *pathname, const char *arg, ...
                /*, (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

https://man7.org/linux/man-pages/man2/execve.2.html

进程创建相关的 APIs

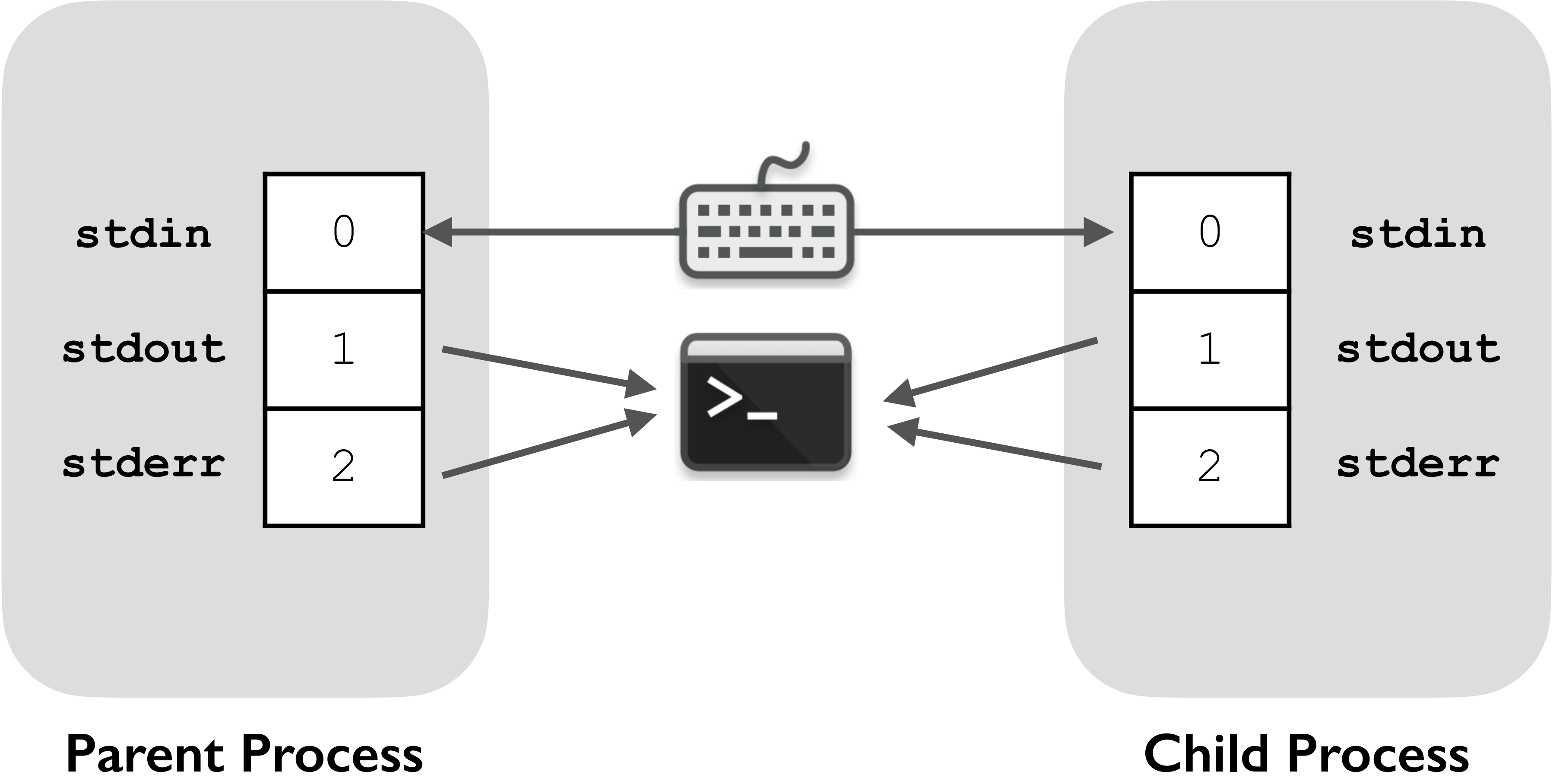
`fork` + `execve` 的设计

能灵活地改变即将要运行程序的环境

- 可以在调用 `fork()` 之后、执行 `execve()` 之前运行特定的代码
  - 继承父进程的一部分信息、同时改变另一部分信息
  - 例如，打开或关闭某些文件、修改环境变量等
- 构建 Unix Shell 的必要能力
  - 重定向 (redirection): `wc fork.c > out.txt`
  - 管道 (pipe): `cat a.txt b.txt | sort`

进程创建相关的 APIs

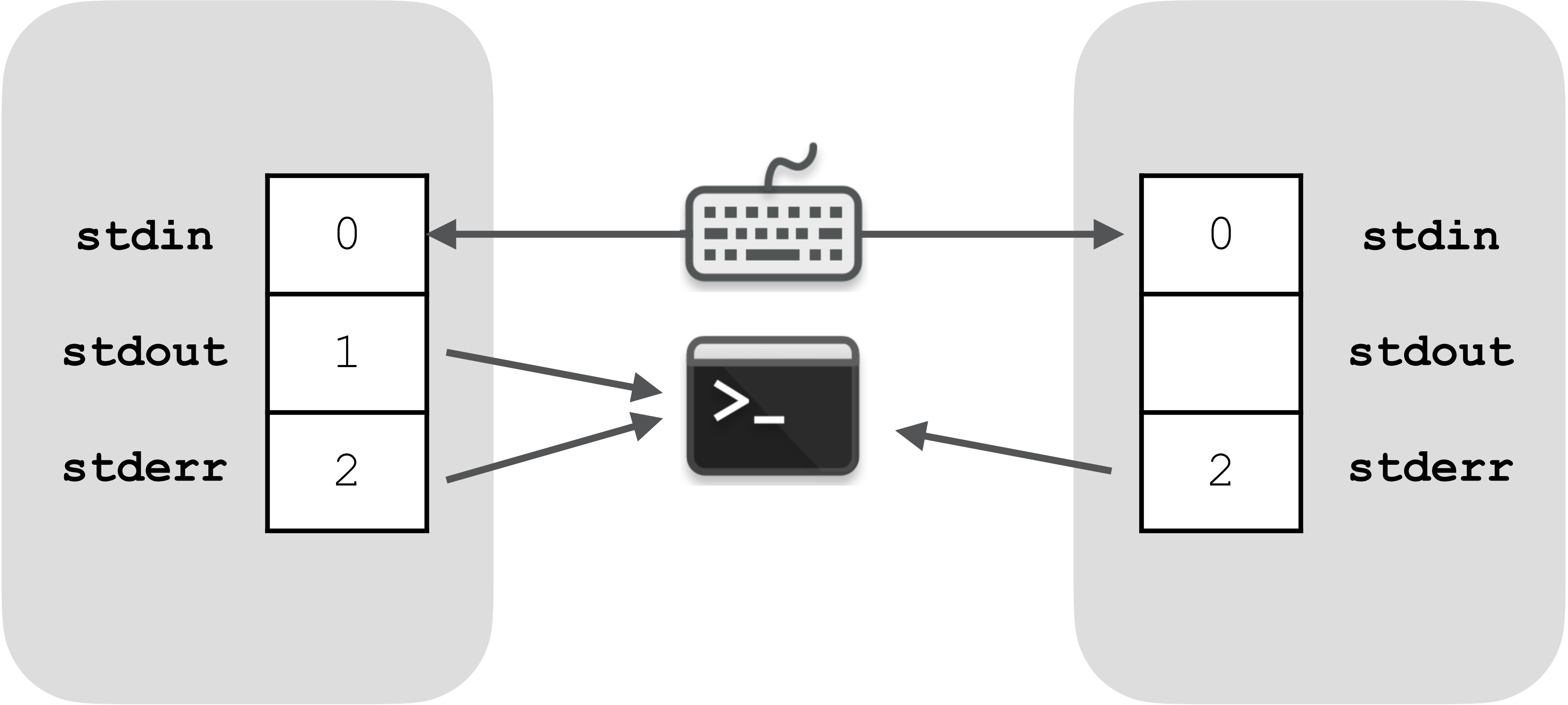
重定向 (redirection)



- 子进程会继承父进程的文件描述符 (File Descriptor)
    - 指向操作系统内部对象的 “指针”
    - 以操作系统所允许的方式来访问不同对象

进程创建相关的 APIs

重定向 (redirection)

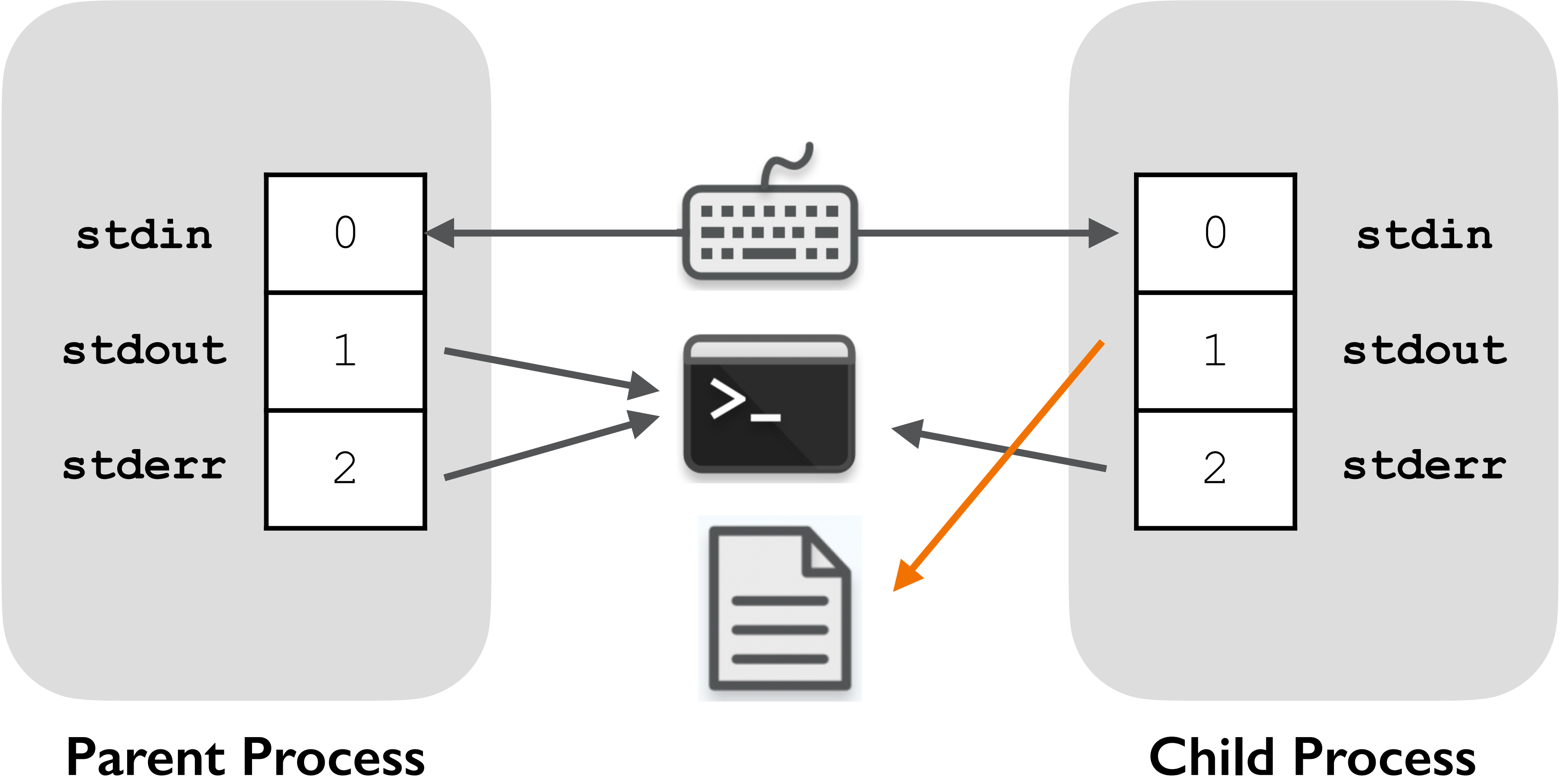


Parent Process **Child Process**

- 在执行 `execve()` 之前关闭 stdout

进程创建相关的 APIs

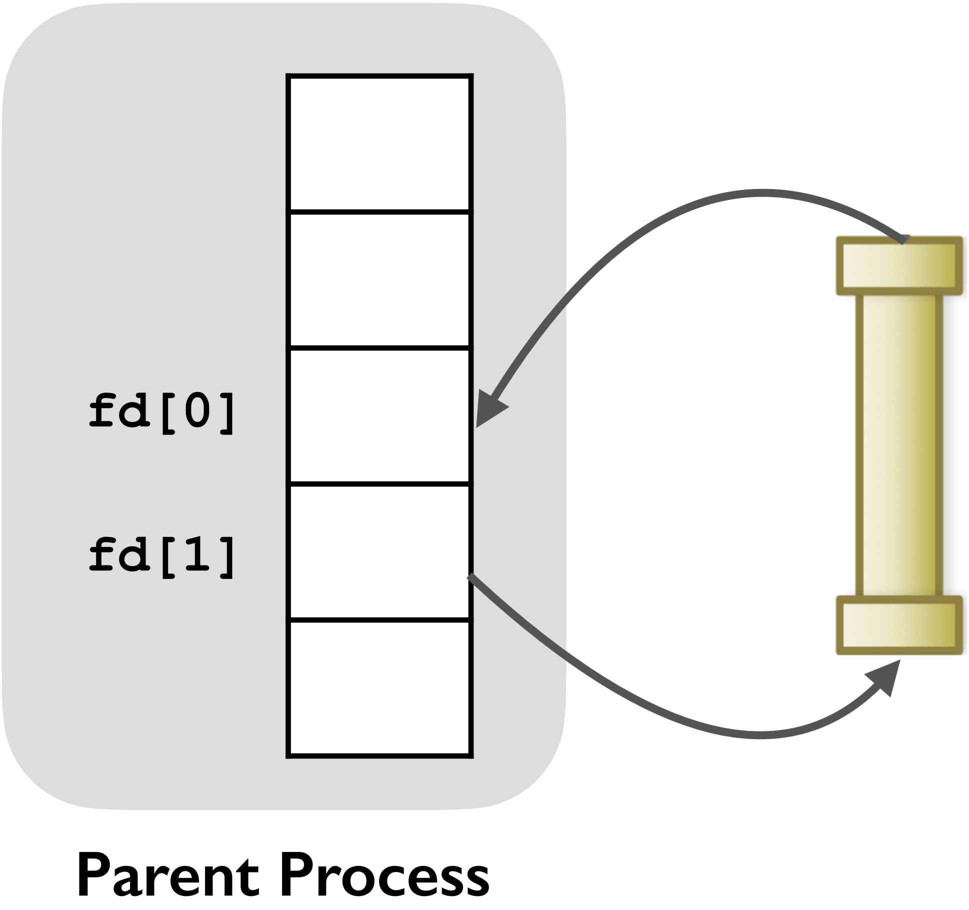
重定向 (redirection)



- 然后打开一个文件，会分配当前最小空闲位置的文件描述符
- 随后程序执行的所有 printf 都将重定向到该文件 (无需修改程序代码)

进程创建相关的 APIs

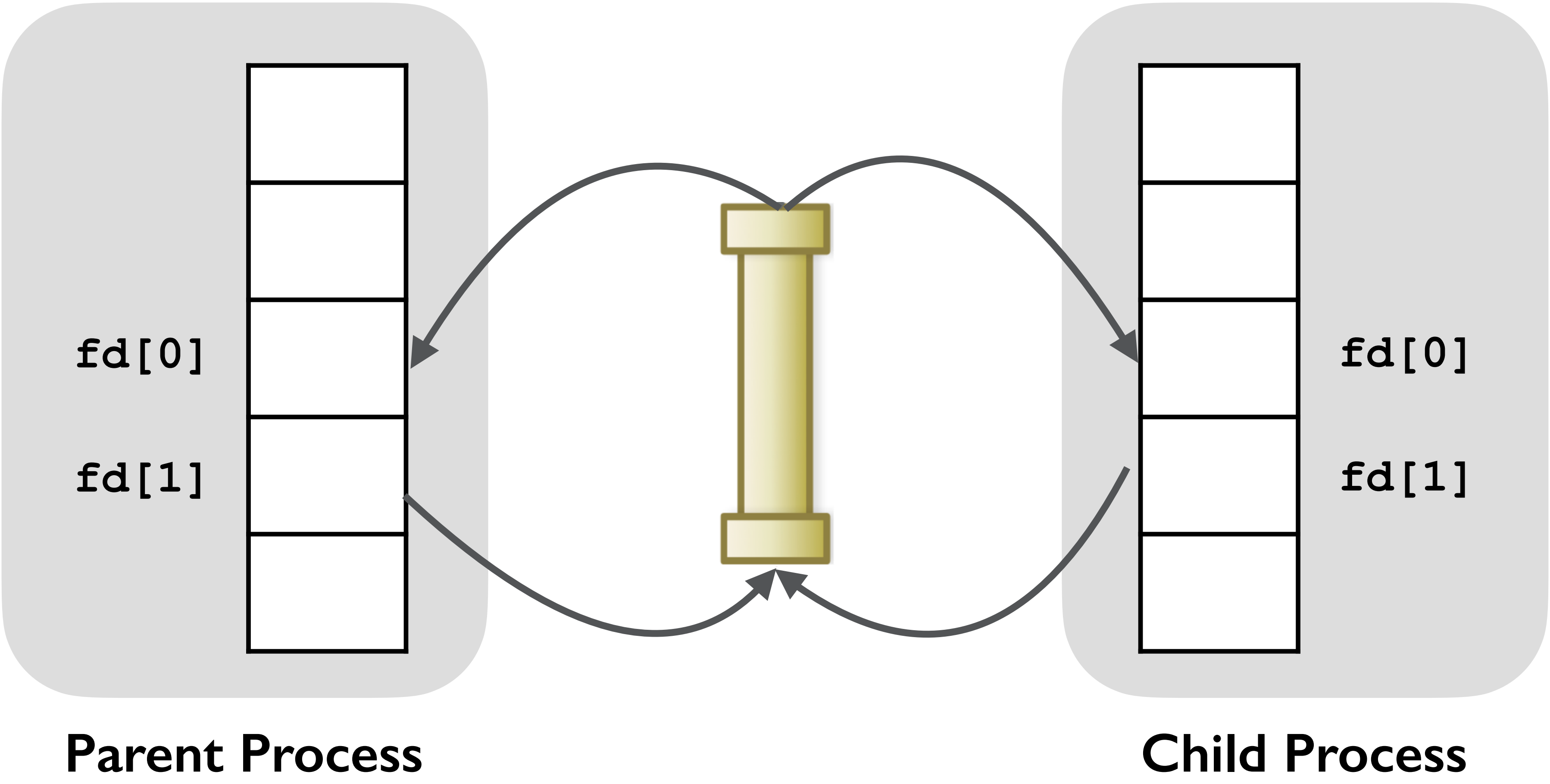
管道 (pipe)



- 在 `fork()` 之前，父进程使用 `pipe(int fd[2])` 创建一个管道对象
- 返回两个文件描述符，分别用于 read-end 和 write-end

进程创建相关的 APIs

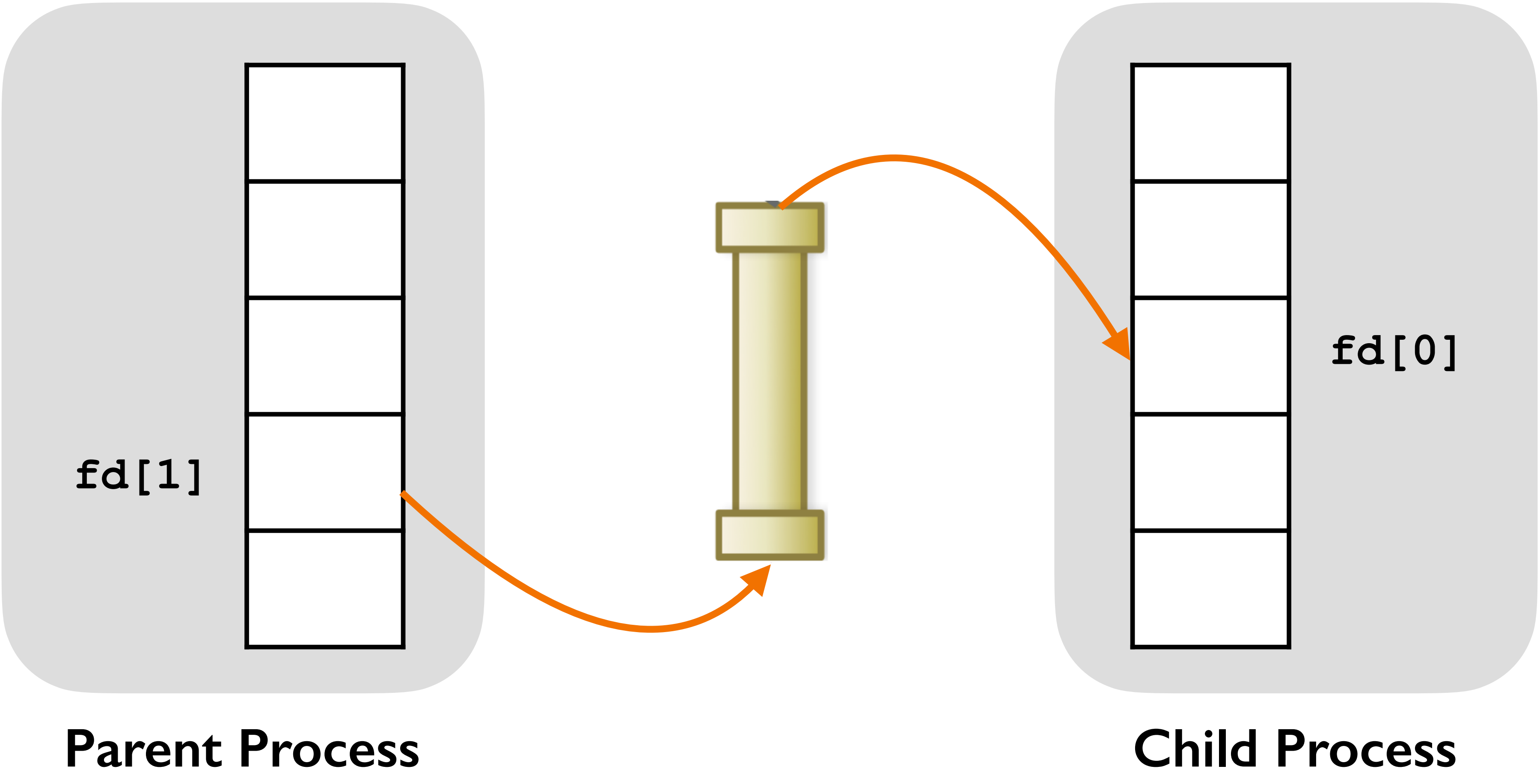
管道 (pipe)



- 执行 `fork()`，子进程具有同样的 read-end 和 write-end

进程创建相关的 APIs

管道 (pipe)



- 在 `execve()` 之前，父进程关闭 read-end，子进程关闭 write-end
- 随后，父进程就可以通过该管道向子进程传递信息
  - 分别调用 `write()` 和 `read()`

进程创建相关的 APIs

管道 (pipe)

```
#define MSGSIZE 10
char *msg = “Message”;

int main() {
  char inbuf[MSGSIZE];
  int fd[2];
  pipe(&fd[0]);      // ignore error handling
  int pid = fork();
  if (pid > 0) {     // parent process (writer)
    close(fd[0]);
    write(fd[1], msg, MSGSIZE);
    printf("Send: %s\n", msg);
    wait(NULL);
  } else {           // child process (reader)
    close(fd[1]);
    read(fd[0], inbuf, MSGSIZE);
    printf("Receive: %s\n", inbuf);
  }
}
```

进程创建相关的 APIs

管道 (pipe)

为什么父子进程要分别关闭 read-end 和 write-end?

- 如果父进程没有关闭 `fd[0]`
    - 无法知道是不是还有进程想读，而 `write()` 将在管道满时阻塞
    - 如果所有 read-end 都已关闭，父进程将收到 SIGPIPE 信号 (broken pipe)，在默认情况下会终止父进程
- 如果子进程没有关闭 `fd[1]`
    - 子进程可能阻塞在 `read()` 上，而父进程又在 `wait()` 子进程
    - 如果所有 write-end 都已关闭，`read()` 返回 EOF (End-Of-File)

进程创建相关的 APIs

管道 (pipe)

管道就是一个特殊的 “文件”

- 普通管道 (匿名管道) 允许父子进程以 Producer-Consumer 模式进行进程间通信 (Inter-Process Communication, IPC)
  - 一种单向 (unidirectional) 的数据传输通道
  - 分别利用 `pipe(int fd[2])` 返回的 `fd[0]` 和 `fd[1]` 进行读写
- 可以利用 `mkfifo()` 创建命名管道
  - 可在文件系统中看到
  - 任意进程都可以 `open()` 该管道 “文件” 进行读写

进程创建相关的 APIs

Windows

不同的操作系统往往使用不同的 APIs 设计

- Windows 提供一个具有很多参数的系统调用来创建进程

```
// Start the child process
if (!CreateProcess(NULL,   // No module name (use command line)
    argv[1],        // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,              // No creation flags
    NULL,           // Use parent's environment block
    NULL,           // Use parent's starting directory
    &si,            // Pointer to STARTUPINFO structure
    &pi )           // Pointer to PROCESS_INFORMATION structure
)
```

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 to process creation in systems developed independently of Unix [e.g., 1, 30, 33, 54].

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 implementations using our experiences with prior research systems (§5). Fork limits the ability of OS researchers and developers to innovate because any new abstraction must be special-cased for it. Systems that support fork and exec efficiently are forced to duplicate per-process state lazily. This encourages the centralisation of state, a major problem for systems not structured using monolithic kernels. On the other hand, research systems that avoid implementing fork are unable to run the enormous body of software that uses it.

We end with a discussion of alternatives (§6) and a call to action (§7): fork should be removed as a first-class primitive of our systems, and replaced with good-enough emulation for legacy applications. It is not enough to add new primitives to the OS—fork must be removed from the kernel.

HotOS '19, May 13–15, 2019, Bertinoro, Italy
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321435

Baumann A, Appavoo J, Krieger O, et al. A fork () in the road. Proceedings of the Workshop on Hot Topics in Operating Systems. 2019: 14-22.

进程创建相关的 APIs

POSIX

POSIX 标准提供了 `posix_spawn()` 接口来创建子进程并执行新程序

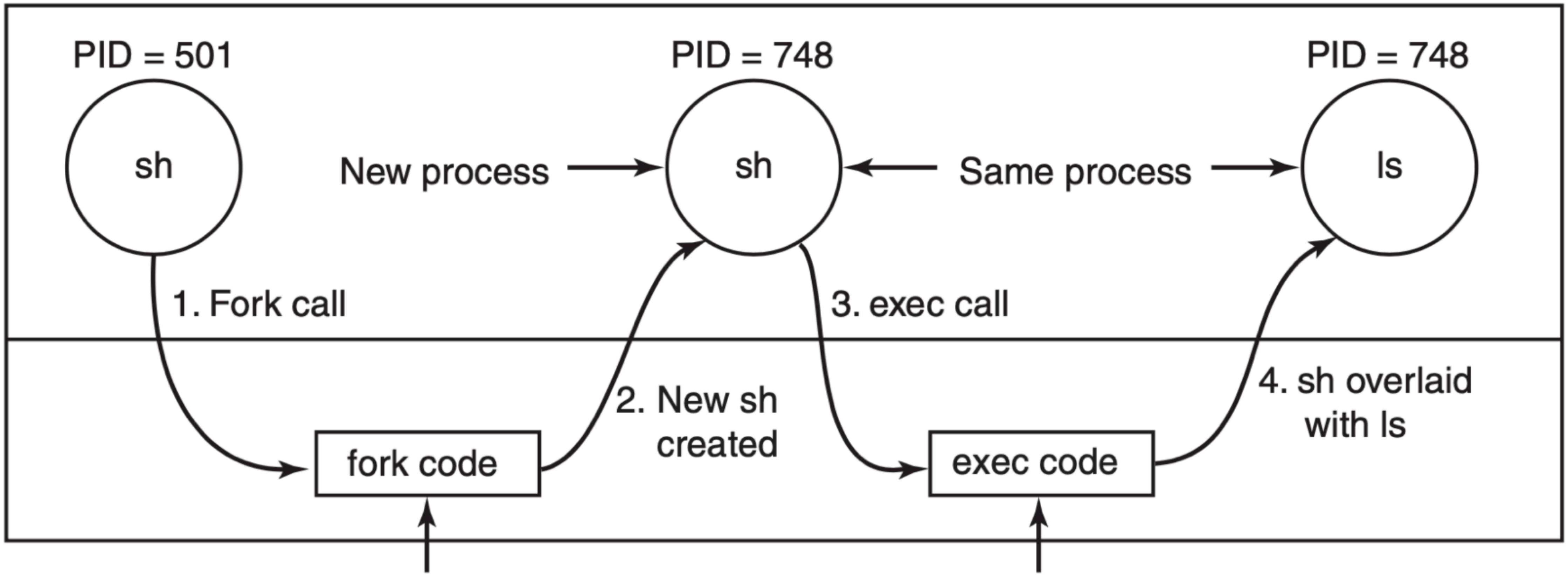
- 通过一个 file actions object (`*file_actions`) 指明在 `fork` 和 `execve` 之间需进行的文件相关操作 (e.g., redirection)
- 通过一个 attributes object (`*attrp`) 指明所创建子进程的属性 (e.g., priority, signal handling, …)

```
int posix_spawn(pid_t *restrict pid, const char *restrict path,
                const posix_spawn_file_actions_t *restrict file_actions,
                const posix_spawnattr_t *restrict attrp,
                char *const argv[restrict],
                char *const envp[restrict]);
```

https://www.man7.org/linux/man-pages/man3/posix_spawn.3.html

进程创建相关的 APIs

A Simple Shell



Allocate child's task structure
Fill child's task structure from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

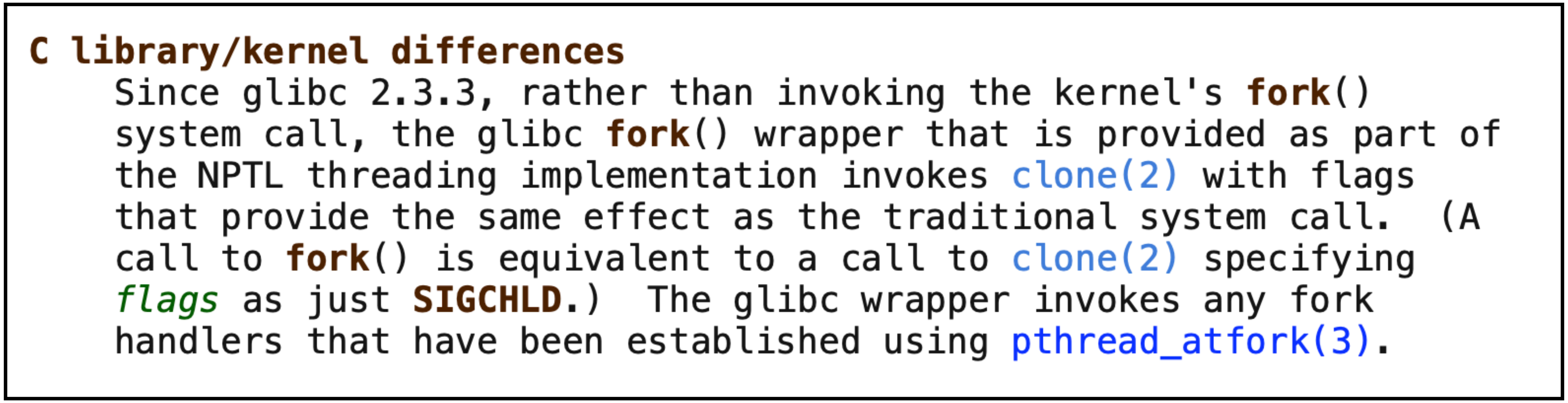
Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

进程创建相关的 APIs

The Real System

`strace -f ./fork` 中没有 `fork()`？

- Linux 提供 `clone()` 系统调用来创建进程 (与线程实现有关)，而 `fork()`"系统调用"只是 libc 中对 `clone()` 的封装
- 我们可以使用 `ltrace` 来追踪库函数调用



```
C library/kernel differences
    Since glibc 2.3.3, rather than invoking the kernel's fork()
    system call, the glibc fork() wrapper that is provided as part of
    the NPTL threading implementation invokes clone(2) with flags
    that provide the same effect as the traditional system call.  (A
    call to fork() is equivalent to a call to clone(2) specifying
    flags as just SIGCHLD.)  The glibc wrapper invokes any fork
    handlers that have been established using pthread_atfork(3).
```

https://man7.org/linux/man-pages/man2/fork.2.html

https://thorstenball.com/blog/2014/06/13/where-did-fork-go/

进程创建相关的 APIs

The Real System

./a.out 和 ./a.out | cat 分别会输出几个 hello？

- 终端上的 stdout 默认是 line buffered (在新的一行时 flush)
- 如果输出不是终端，则会使用 fully buffered

```
int main() {
  for (int i = 0 ; i < 2 ; i++) {
    fork();
    printf(“hello\n”);
  }
  return 0;
}
```

https://man7.org/linux/man-pages/man3/setbuf.3.html