进程和线程

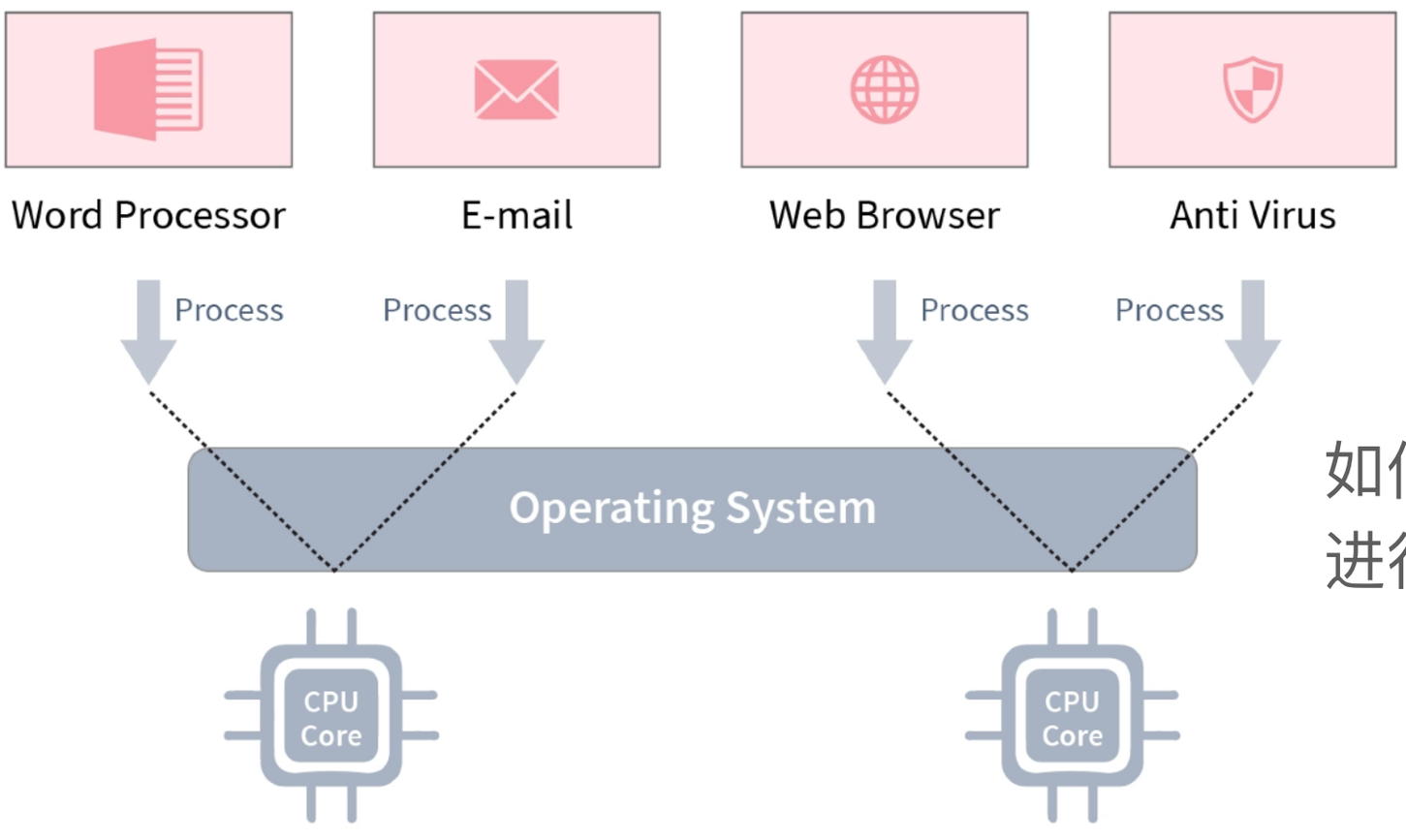
Section 2: Part I

2026

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)
- 实现对 CPU 的分时共享 (time sharing)

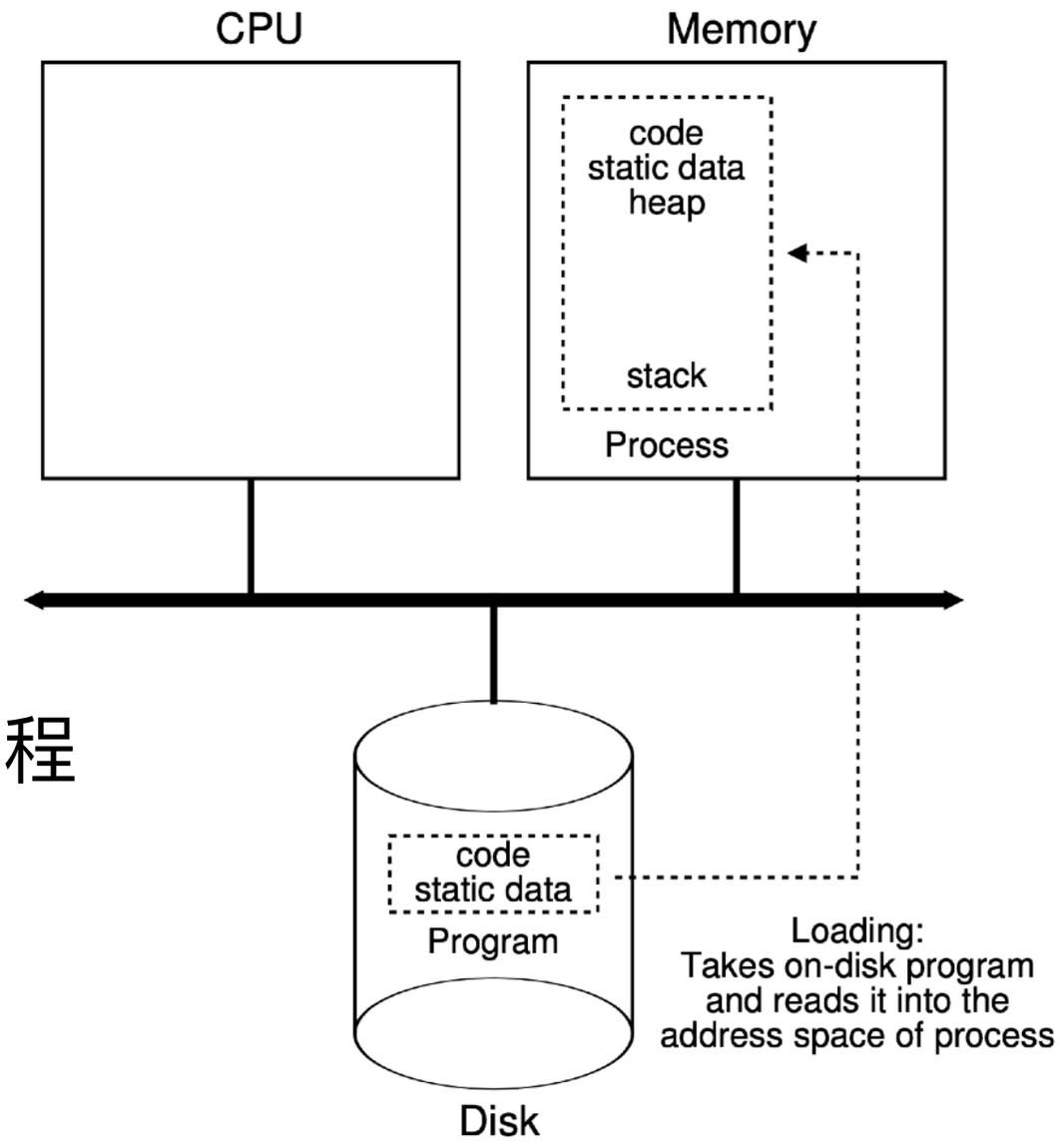


如何对正在运行的多个程序进行抽象和管理?

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

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

```
enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

xv6 (risc-v)
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

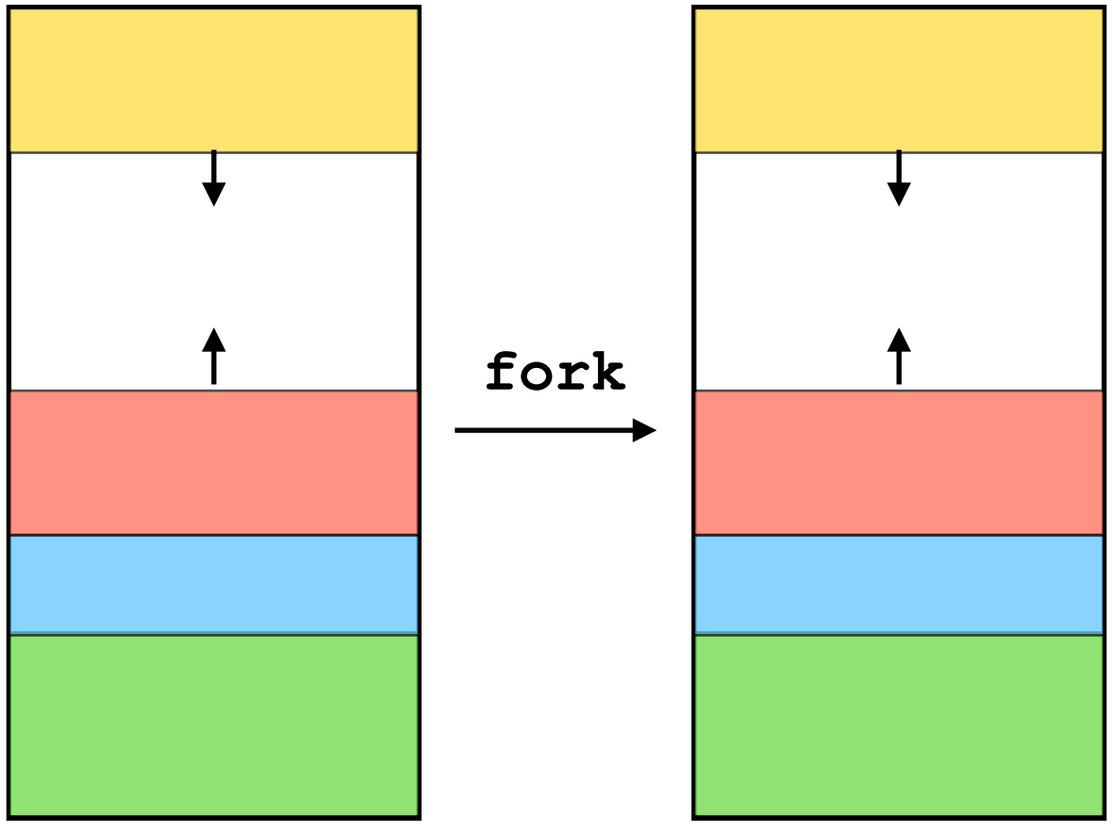
Unix Systems

Unix 中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()

```
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

`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

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

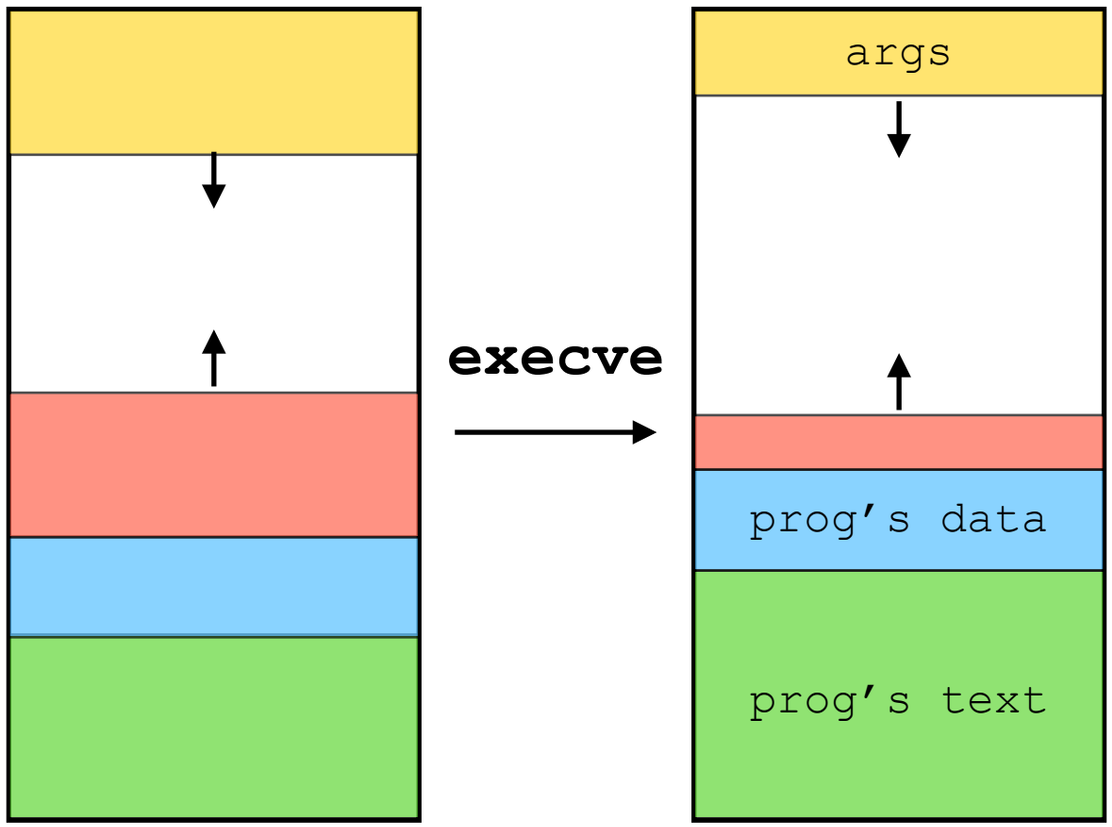
`fork()`

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()`



Before　　After

```
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

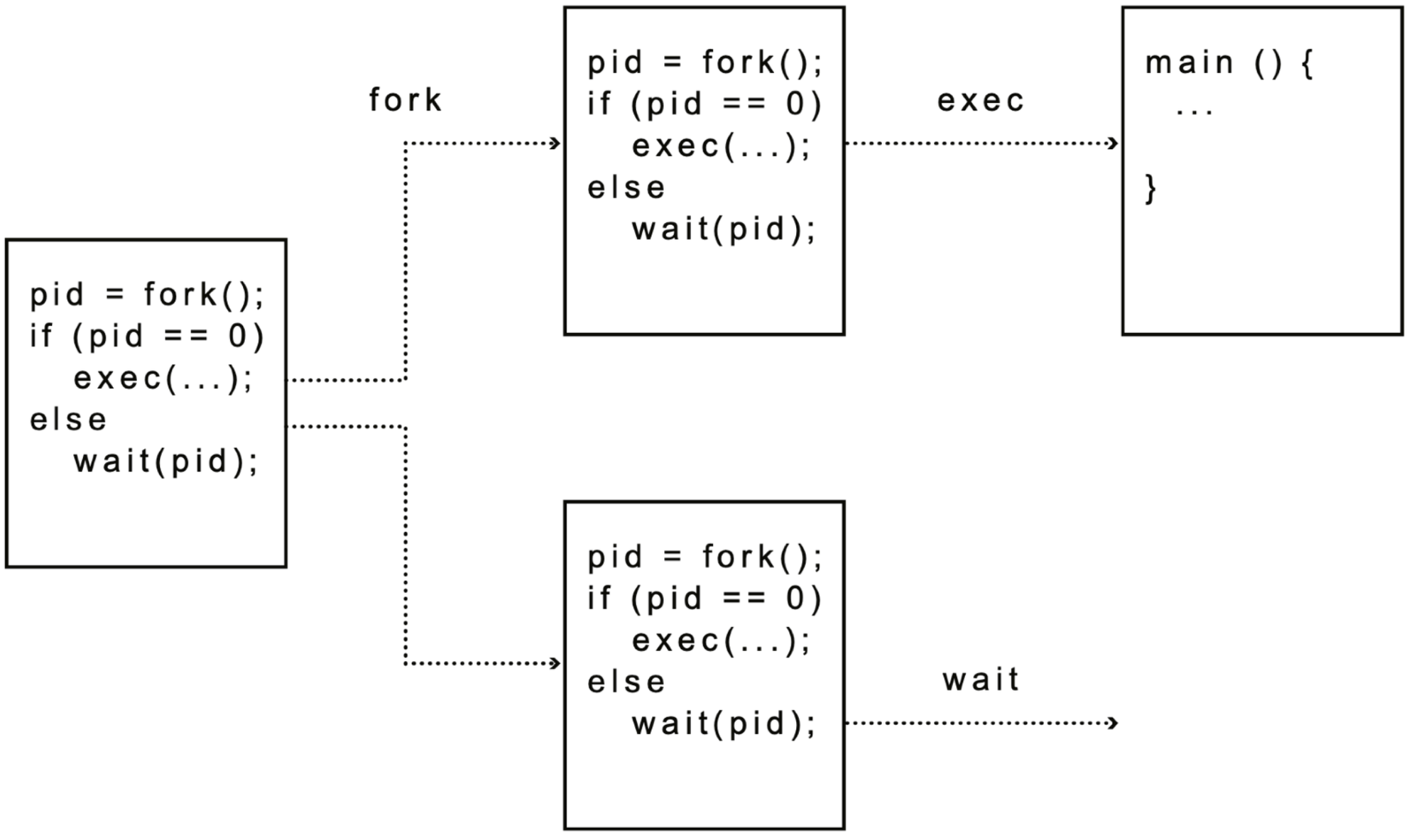
execve()

```
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()



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

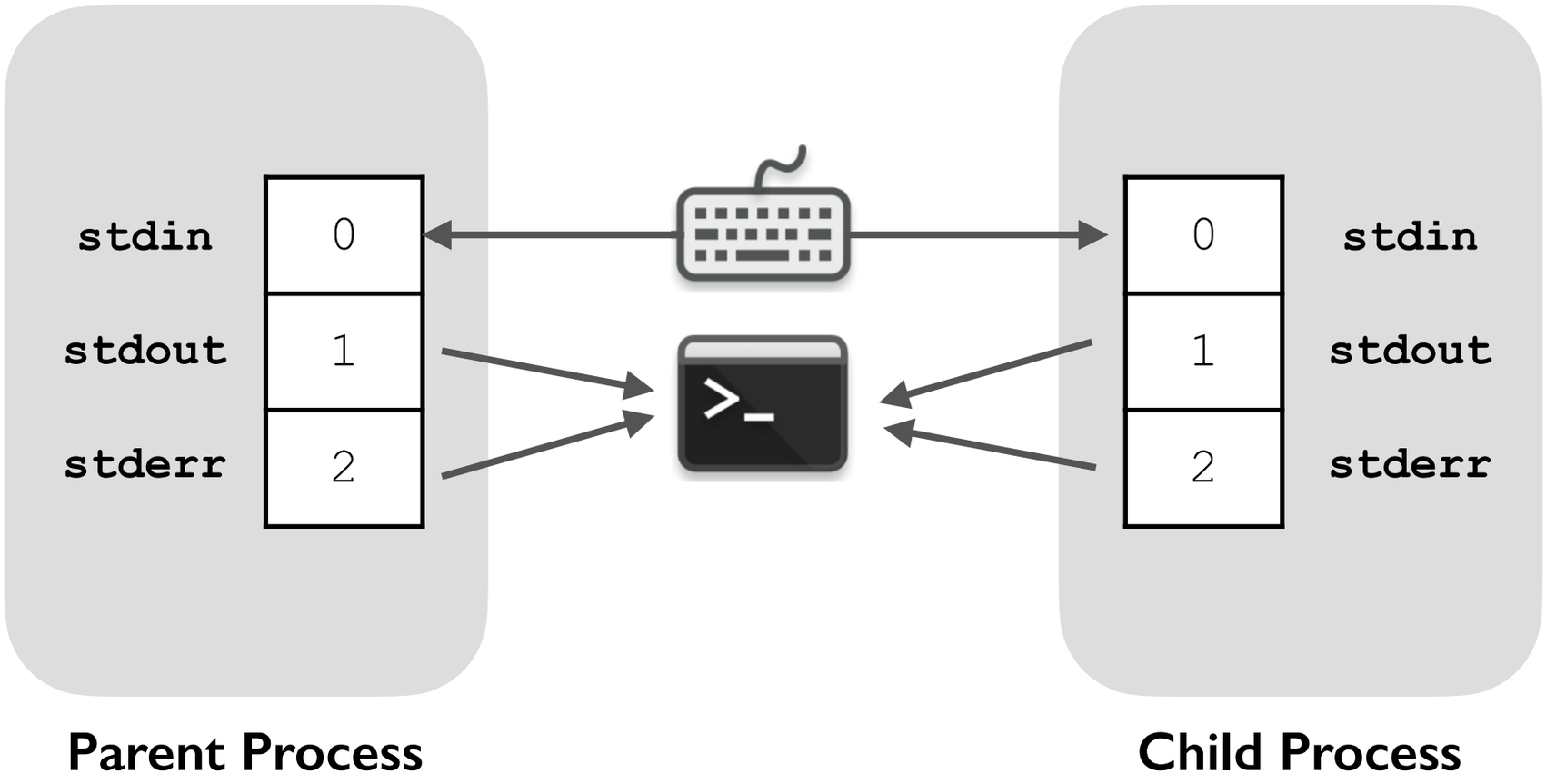
能灵活地改变即将要运行程序的环境

- 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

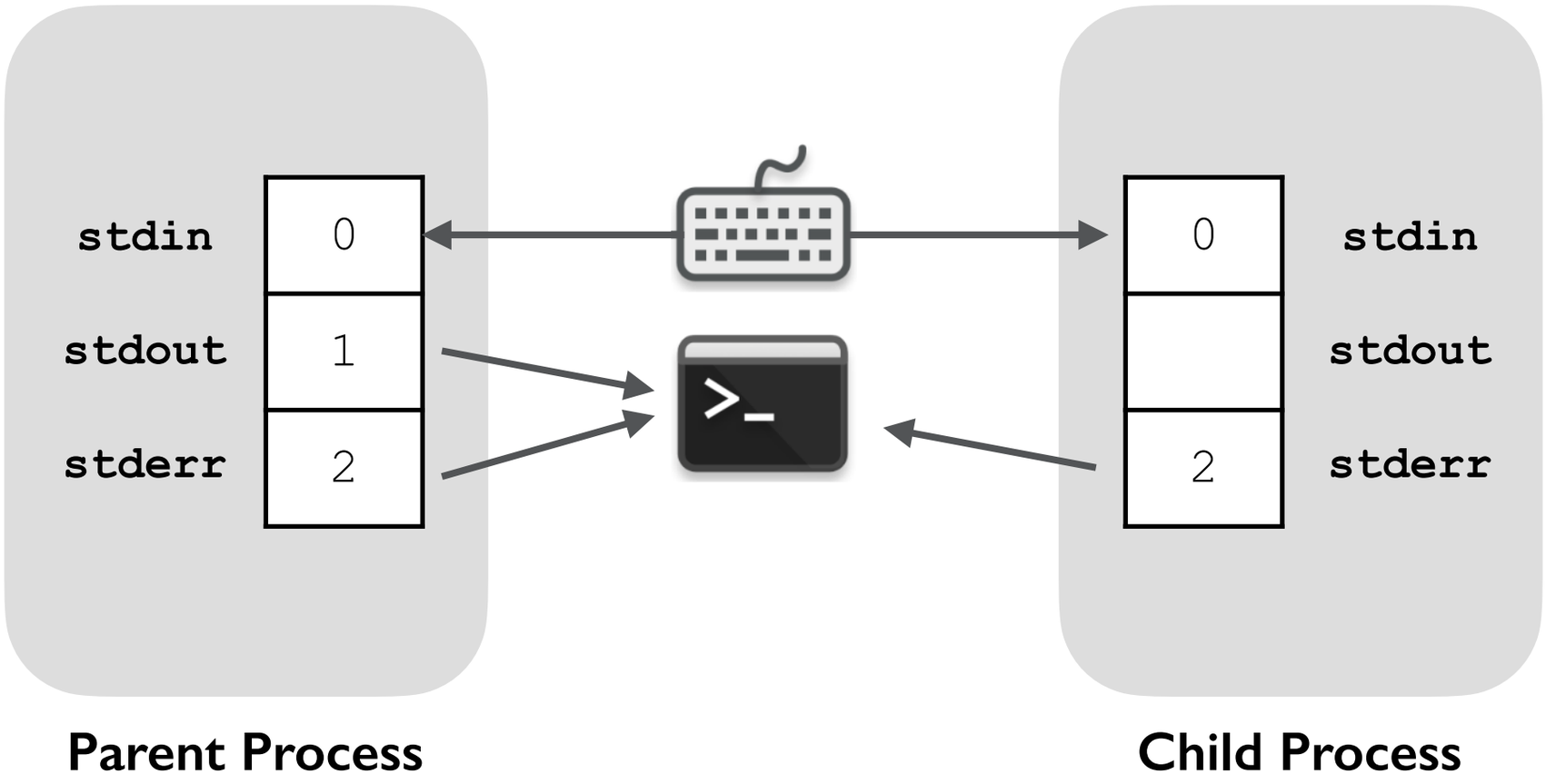
重定向 (redirection)



- 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

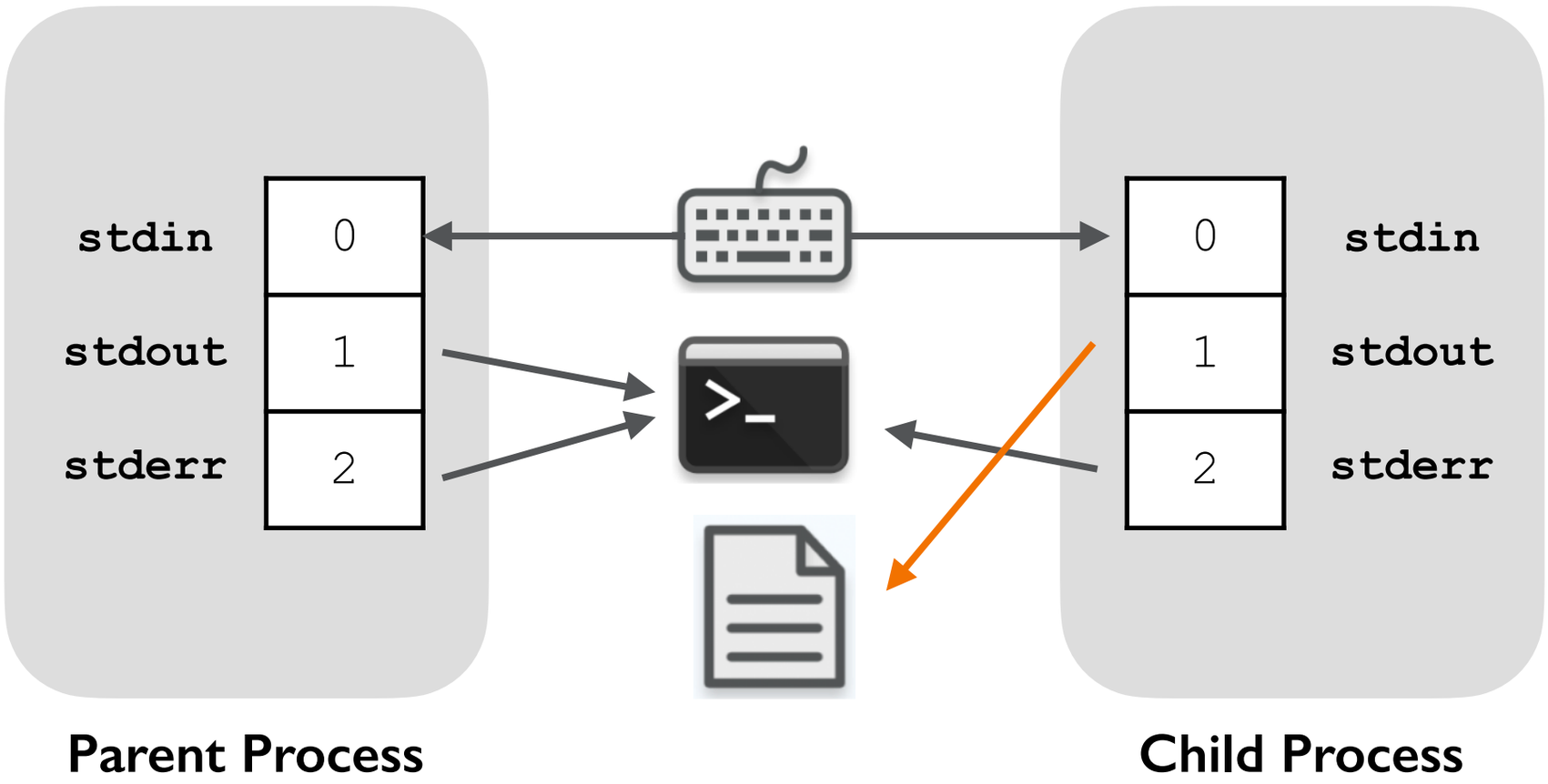
重定向 (redirection)



- 在执行 `execve()` 之前关闭 stdout

进程创建相关的 APIs

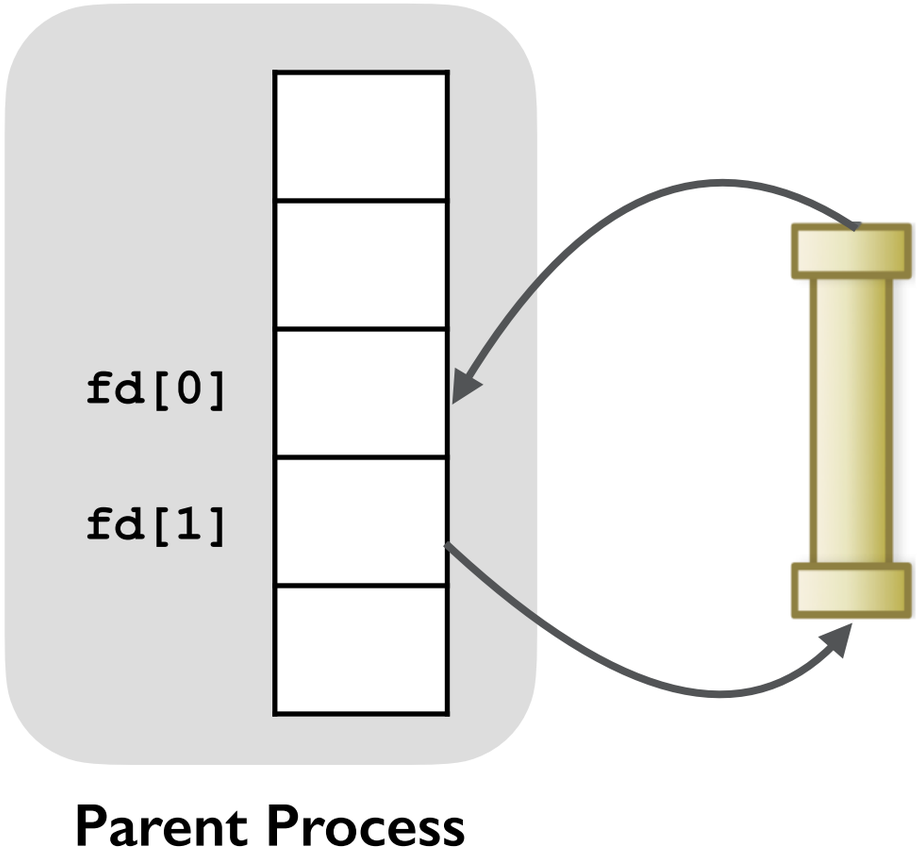
重定向 (redirection)



- 然后打开一个文件，默认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

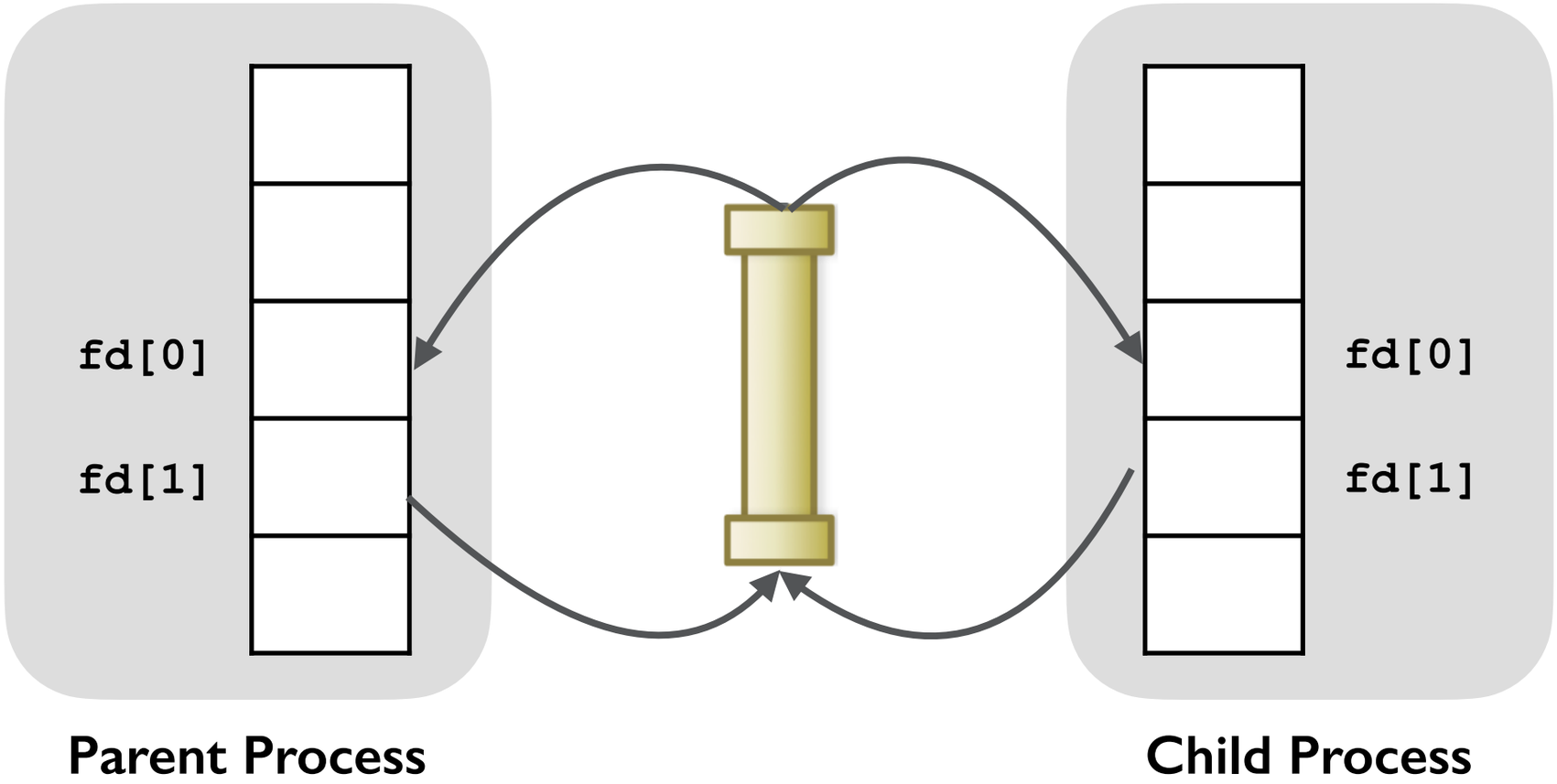
管道 (pipe)



- 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

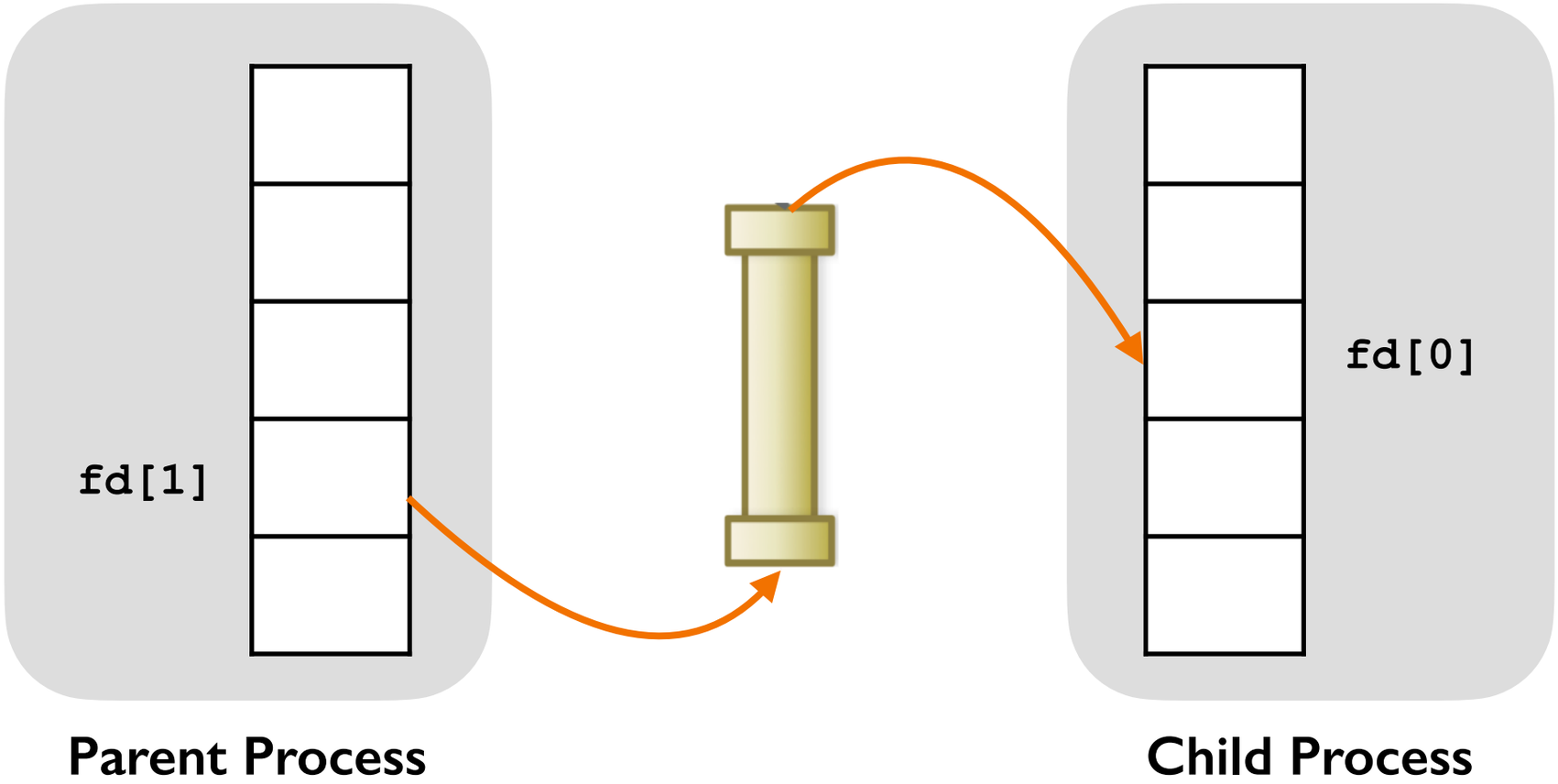
管道 (pipe)



- 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
  - 父子进程分别调用 `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

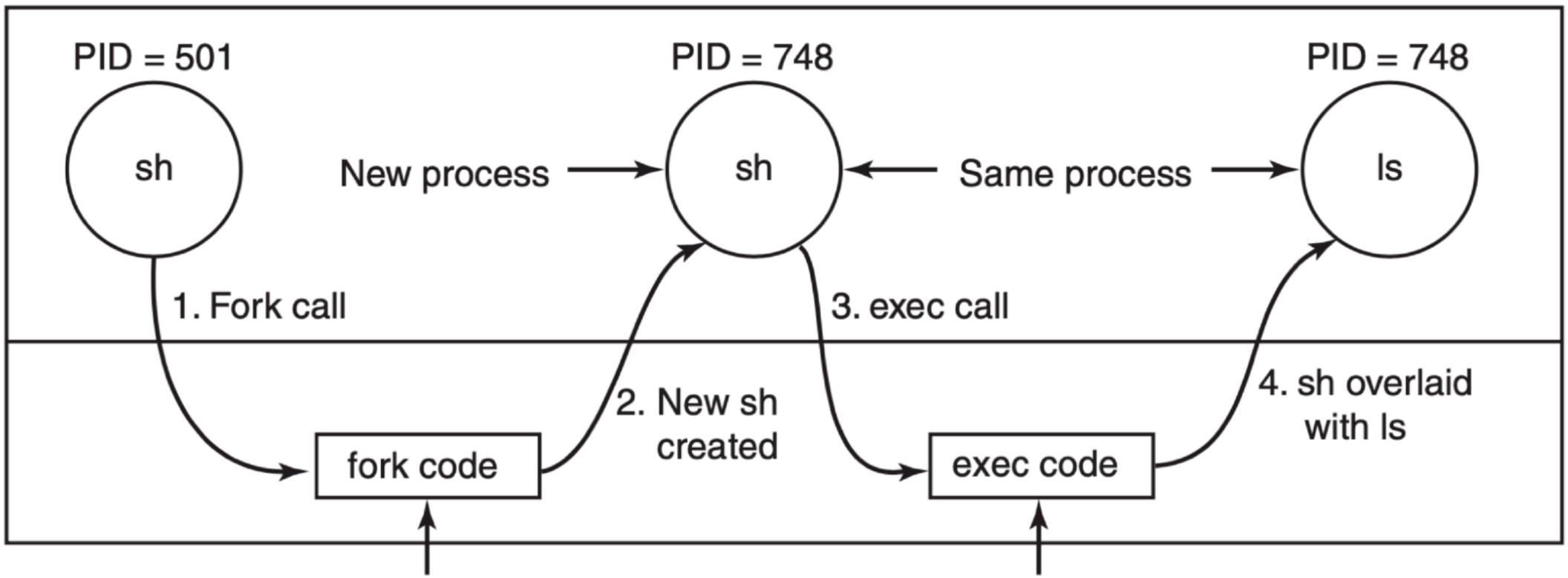
管道 (pipe)

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)
  - 分别利用 `pipe(int fd[2])` 返回的 `fd[0]` 和 `fd[1]` 进行读写
  - 实现一种单向的数据传输通道
- 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

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

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].

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321435

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.

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

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

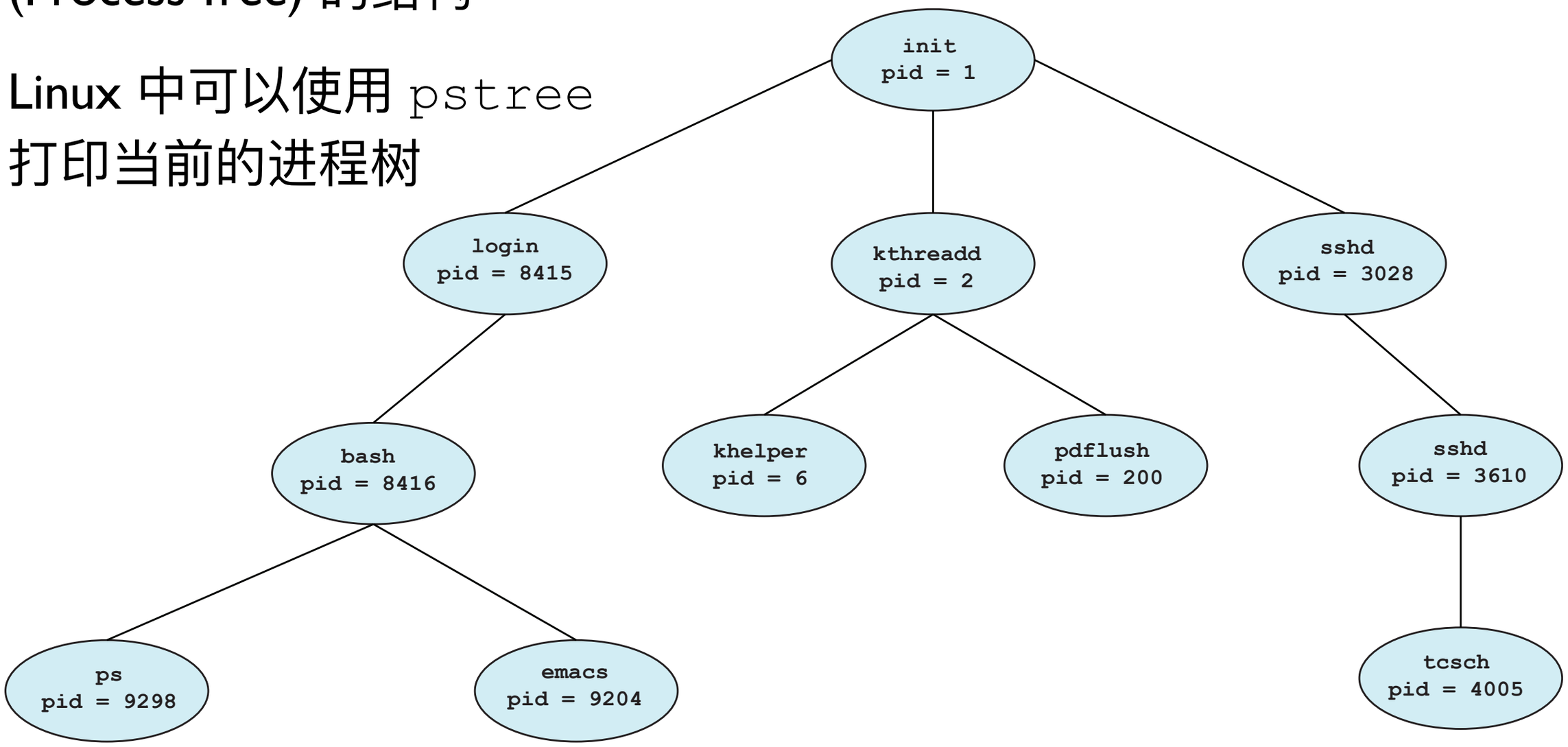
```
int main() {
  for (int i = 0 ; i < 2 ; i++) {
    fork();
    printf("hello\n");
  }
  return 0;
}
```

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

进程的层级关系

新进程都是通过现有进程执行创建进程操作来产生的

- 父进程创建子进程，子进程又创建其他进程，最终形成一种进程树 (Process Tree) 的结构
- Linux 中可以使用 `pstree` 打印当前的进程树



进程的终止

除了创建进程外，进程也需要能够终止运行

- 需要由操作系统来回收其占有的资源 (例如所占用物理内存)
- 从 `main` 函数返回是最常见的终止方式
  - 返回值 `0` 代表符合预期终止

进程的终止

其它一些方终止式

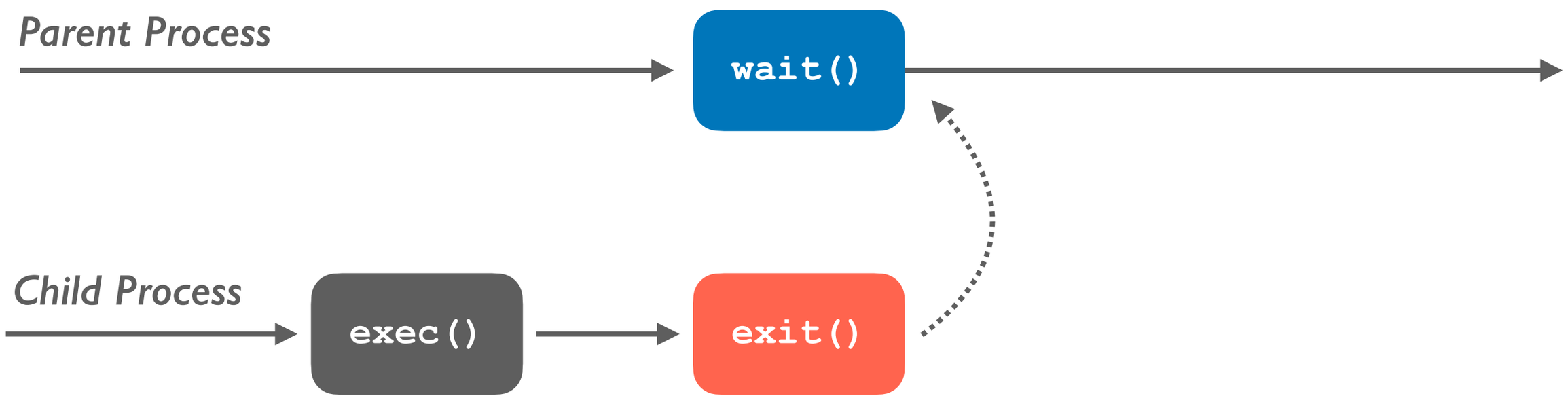
- 使用库函数 `exit()`，其在 `_exit()` 基础上做了一些封装
  - 会执行进程通过 `atexit()` 注册的清理动作
  - 关闭并 flush 所有打开的 stdio streams
- 使用库函数封装的系统调用 `_exit()`
  - 立即停止进程运行，并回收资源 (e.g., 关闭文件描述符)
  - 调用 `exit_group()` 来结束所有线程
- 使用系统调用 `syscall(SYS_exit,0)`
  - 只会终止当前线程
- 以及异常终止: 调用 `abort()`

进程的终止

Zombie Process

如果一个进程已结束执行，但其 PCB 仍在进程表 (process table) 中，则该进程进入僵尸状态 (a living dead)

- 父进程 `wait()`: 内核在子进程结束时将其退出状态 (exit status) 传递给父进程，同时移除子进程的 PCB
- 如果父进程不 `wait()` 子进程一直执行？

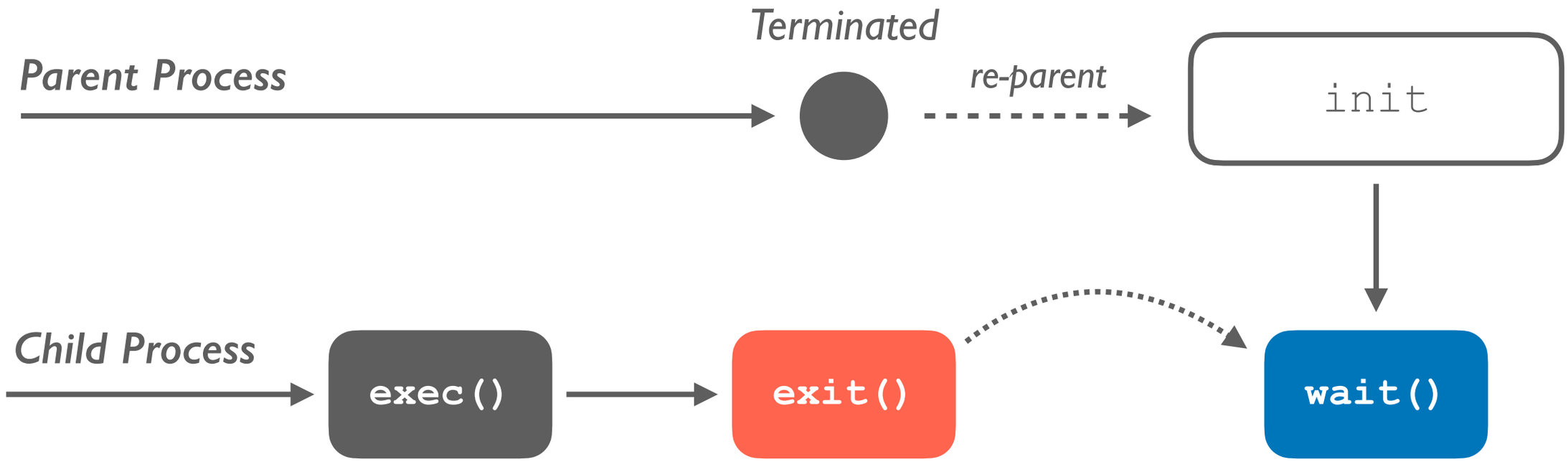


进程的终止

Orphan Process

如果一个进程的父进程先于其结束运行 (在 `wait()` 子进程之前就正常退出、或被终止运行)，则该进程成为孤儿进程

- 在 Linux 中 `init/systemd` 将成为所有孤儿进程的父进程，从而能在这些孤儿进程执行结束时回收资源



信号 (Signal)

`kill -l`

Number	Name	Default action	Corresponding event
1	SIGHUP	Terminate	Terminal line hangup
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Terminate	Quit from keyboard
4	SIGILL	Terminate	Illegal instruction
5	SIGTRAP	Terminate and dump core [a]	Trace trap
6	SIGABRT	Terminate and dump core [a]	Abort signal from `abort` function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core [a]	Floating-point exception
9	SIGKILL	Terminate [b]	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core [a]	Invalid memory reference (seg fault)
12	SIGUSR2	Terminate	User-defined signal 2
13	SIGPIPE	Terminate	Wrote to a pipe with no reader
14	SIGALRM	Terminate	Timer signal from `alarm` function
15	SIGTERM	Terminate	Software termination signal
16	SIGSTKFLT	Terminate	Stack fault on coprocessor
17	SIGCHLD	Ignore	A child process has stopped or terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop until next SIGCONT [b]	Stop signal not from terminal
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal
21	SIGTTIN	Stop until next SIGCONT	Background process read from terminal

Ctrl+C

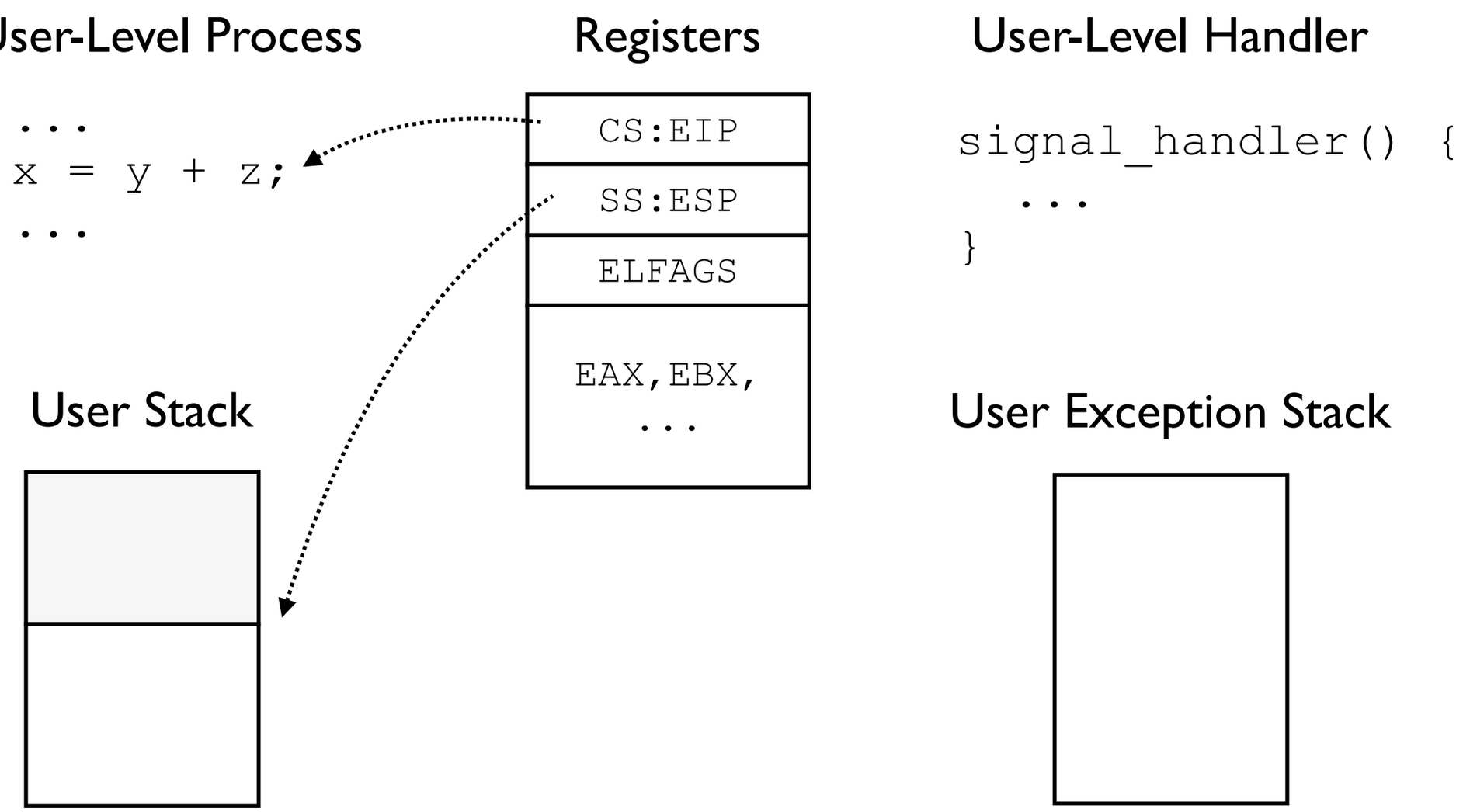
信号 (Signal)

一种通知进程发生了某个事件 (并以进程定义的方式处理) 的机制

- 用户态的 “中断”
  - 用户程序可以通过 `signal()` 注册信号处理函数
  - 如果没有为信号定义处理函数，则内核将调用默认处理程序
- 例如，Ctrl+C 就是向进程发送 `SIGINT` 信号
  - 如何让 Ctrl+C 不起作用?

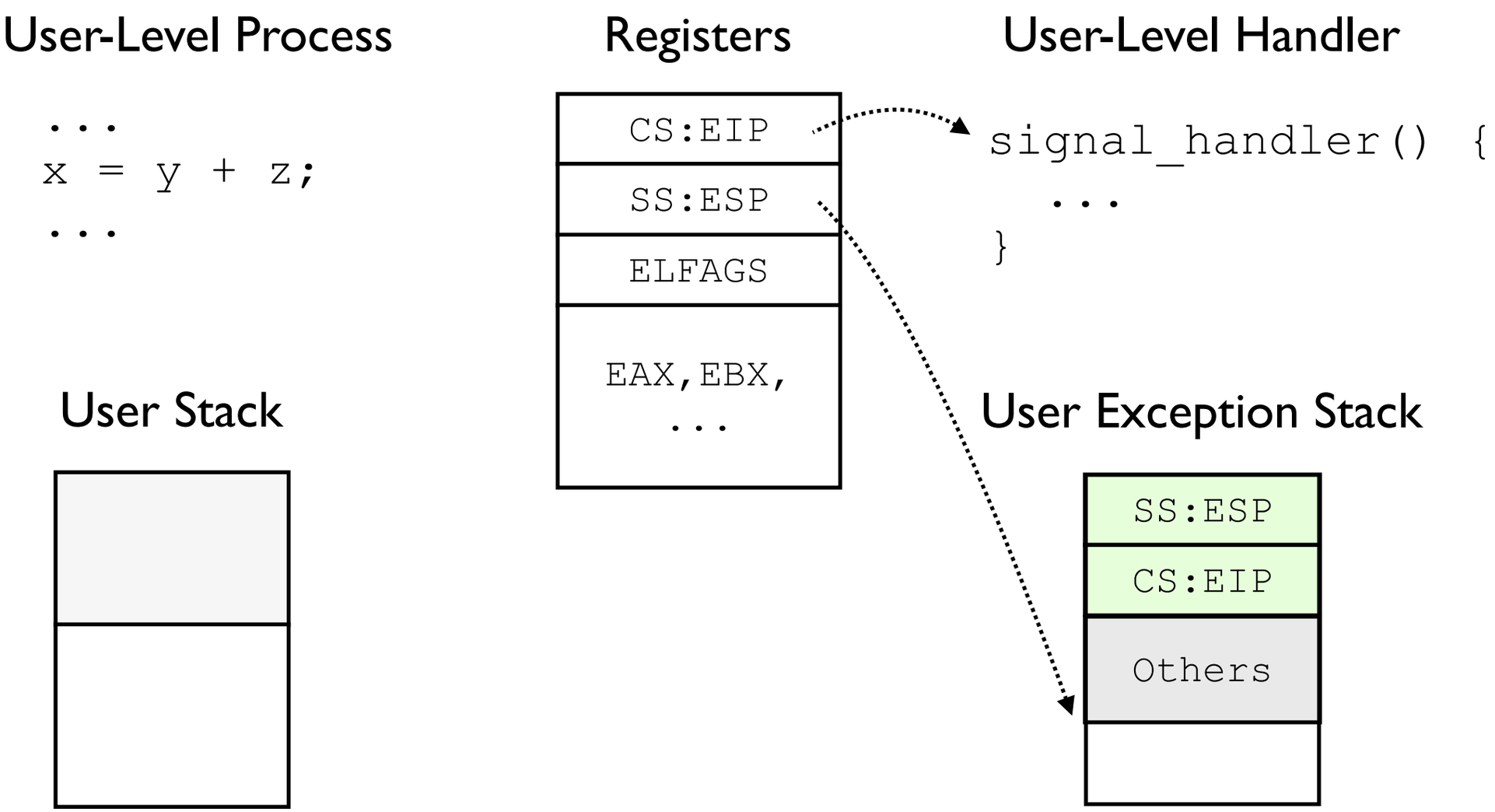
信号 (Signal)

信号的实现 (user level) 与中断 (kernel) 有很多相似之处



信号 (Signal)

信号的实现 (user level) 与中断 (kernel) 有很多相似之处



信号 (Signal)

一种通知进程发生了某个事件 (并以进程定义的方式处理) 的机制

- 可以帮助实现异步操作，例如:
  - `SIGIO` (数据传输完成信号): 可以不用阻塞等待 I/O 完成，通过该信号让回调函数 handler 来处理
  - `SIGCHLD` (子进程终止信号): 可以在 handler 里 `wait()`，而不需要在 main 里 `wait()` 从而阻塞父进程
- 也是一种进程间通信的机制

进程的生命周期

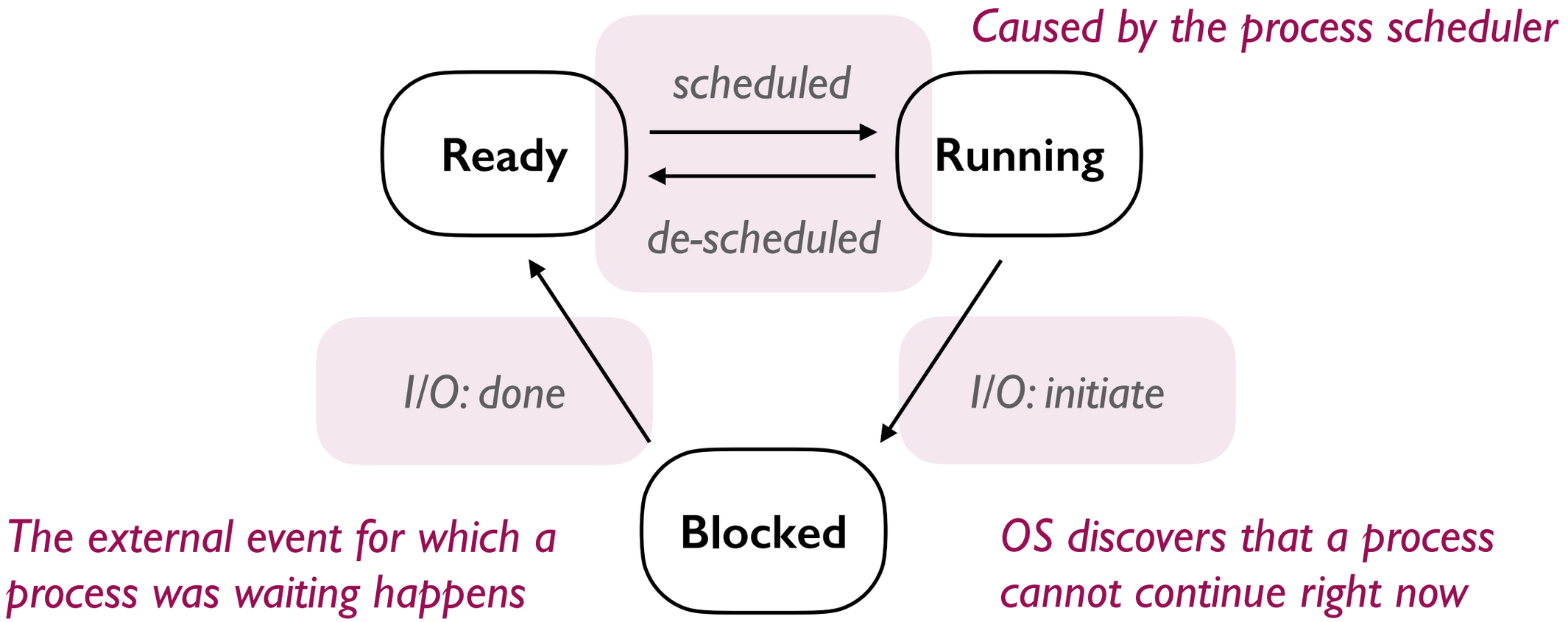
进程在从创建到终止的整个生命周期中会处于不同的执行状态

- 进程执行的三种基本状态
  - Running: 进程当前正在 CPU 上运行 (正在占有 CPU)
  - Ready: 进程已准备就绪但尚未被调度到 CPU 上运行 (等待 CPU)
  - Blocked: 进程执行了某种操作，需要等到某个事件发生后才能准备运行 (等待某个事件)

进程的生命周期

进程在从创建到终止的整个生命周期中会处于不同的执行状态

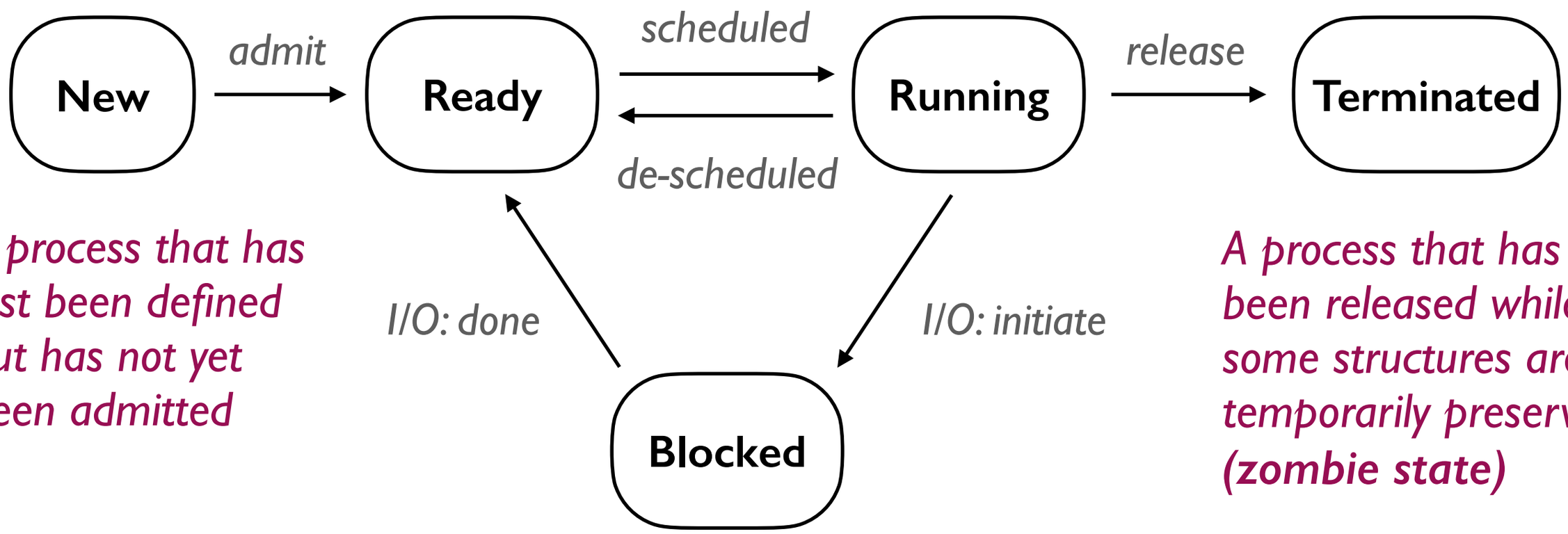
- 随着进程的运行，其状态会不断发生变化



进程的生命周期

进程在从创建到终止的整个生命周期中会处于不同的执行状态

- 新增 New 和 Terminated (Zombie) 状态



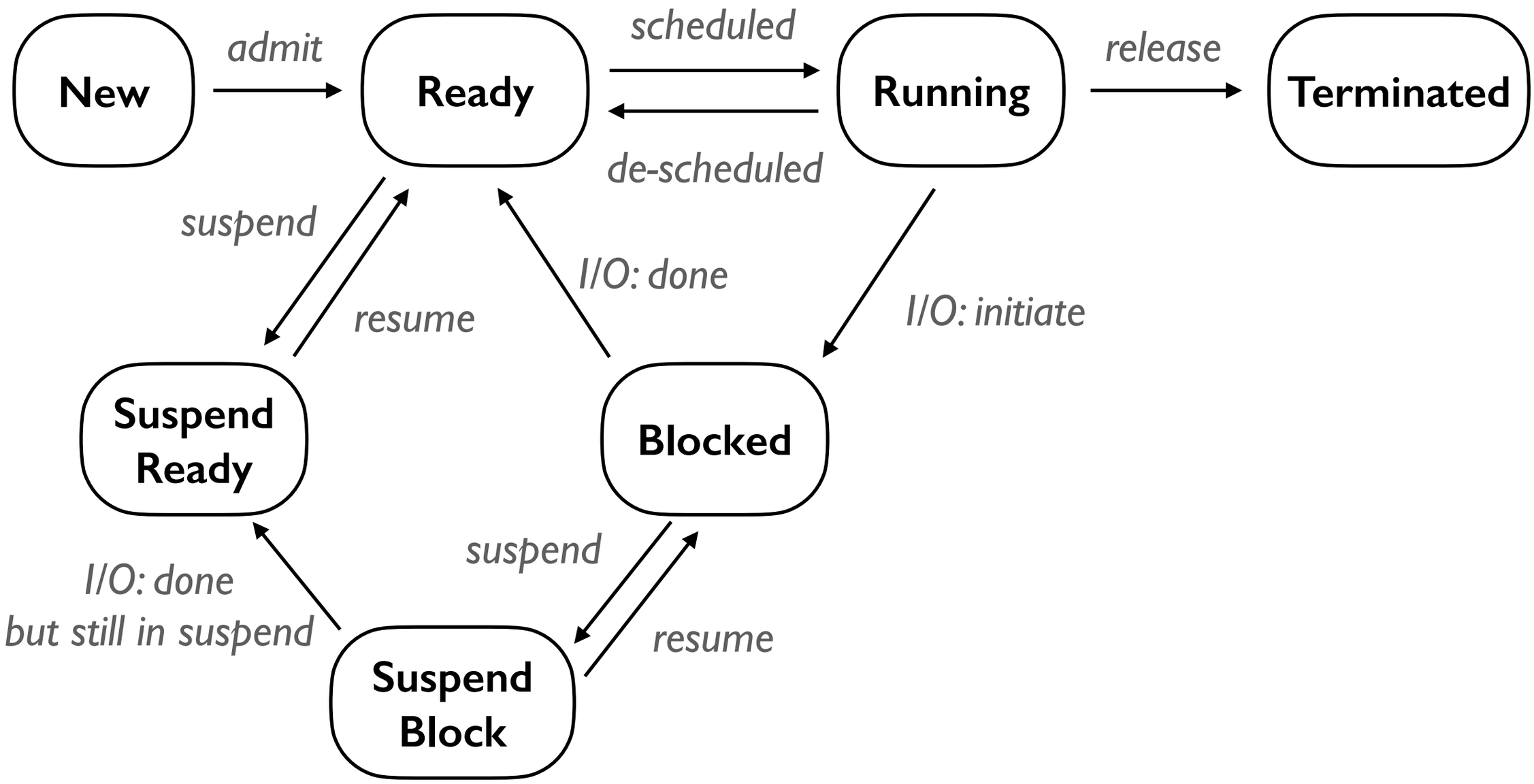
A process that has just been defined but has not yet been admitted

A process that has been released while some structures are temporarily preserved ***(zombie state)***

进程的生命周期

进程在从创建到终止的整个生命周期中会处于不同的执行状态

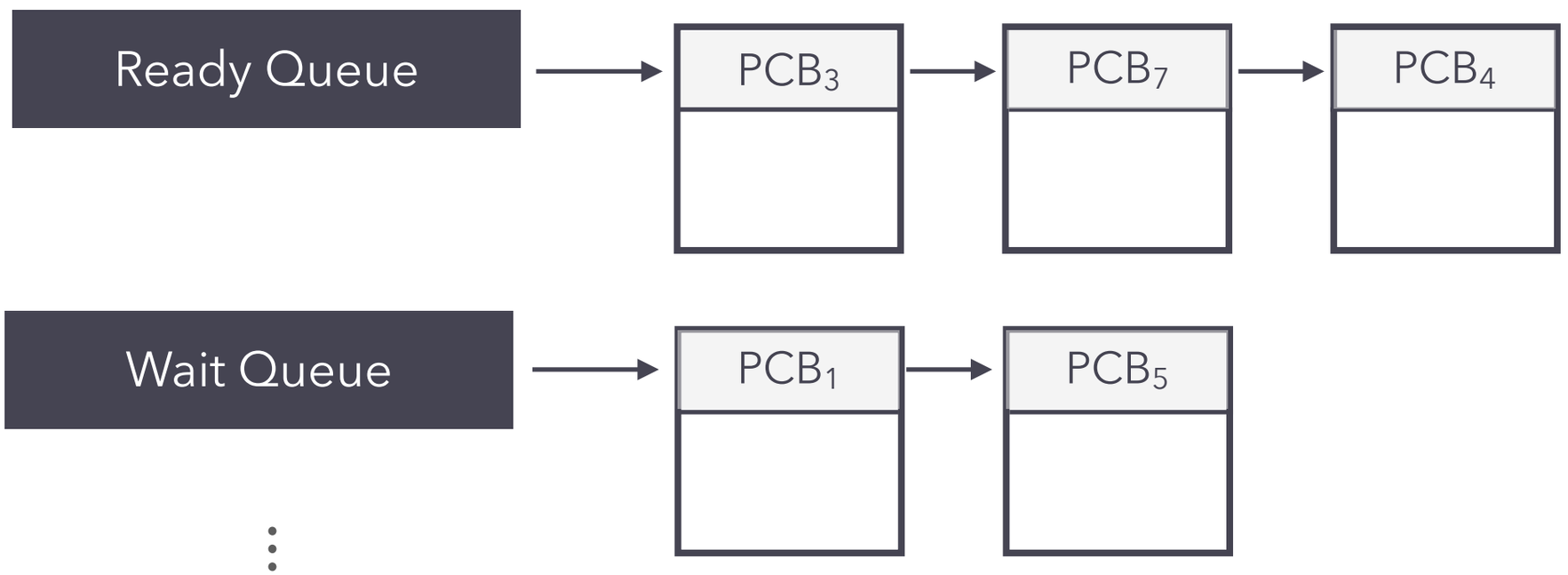
- 新增 Suspend Ready 和 Suspend Block 状态 (考虑在内存和磁盘间 Swap)



进程的生命周期

操作系统需要追踪当前系统中所有进程的状态

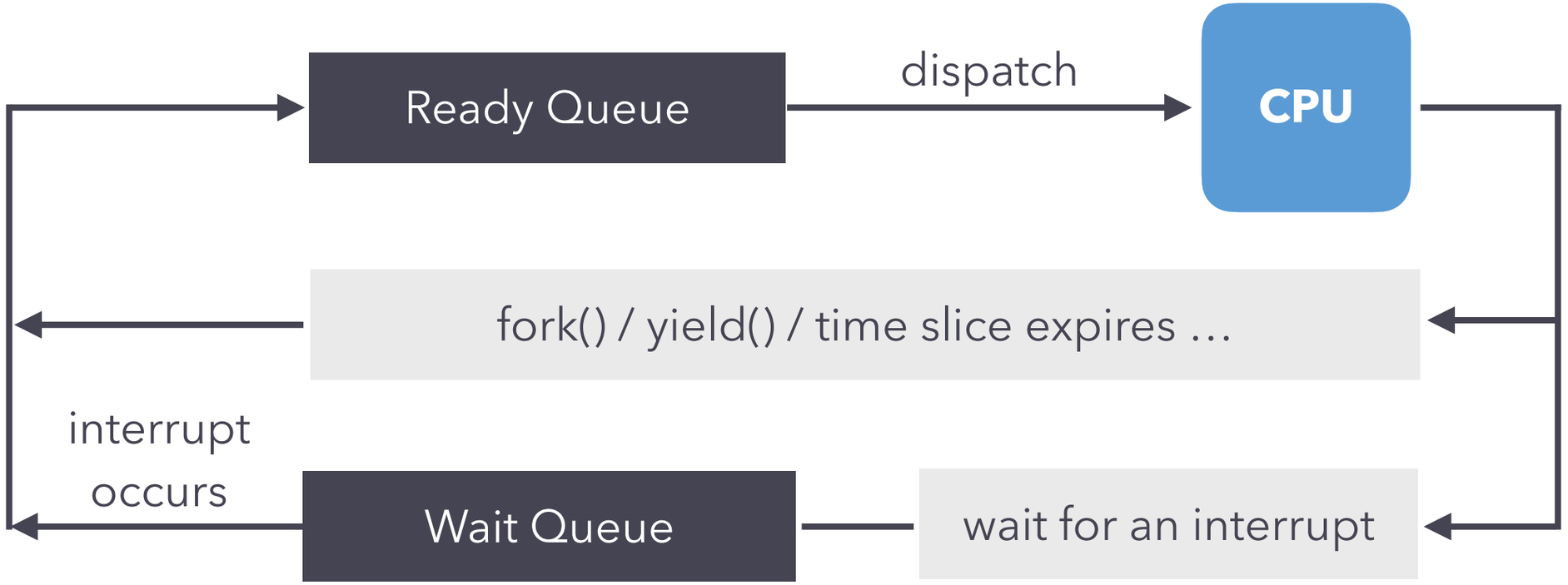
- 所有进程 PCB 放在一个全局数组中 (xv6)
- 可以为每个状态维护一个队列
  - 当创建一个进程时，将其 PCB 加入 Ready 队列中
  - 当进程状态变化时，将其从一个队列移入另一个队列



进程调度

为了实现 CPU 的分时共享，操作系统需要在进程间不断切换 (暂时停止当前进程的运行，并继续运行另一个进程)

- 在进程 `exit()`,`yield()`,`sleep()` 等时都会调用 Scheduler 来选择下一个执行的进程



进程调度

为了实现 CPU 的分时共享，操作系统需要在进程间不断切换 (暂时停止当前进程的运行，并继续运行另一个进程)

- 如果当前 Ready Queue 为空 (没有 runnable 的进程)？
- 操作系统调度一个 Idle Process 执行
  - 循环执行 `HLT` 指令 (x86)，暂停 CPU 执行直到下一个中断到来

上下文切换

如果调度器决定执行另一个进程，则需进行上下文切换 (Context Switch)

- 保存旧进程的 Execution Context 并加载新进程的 Execution Context
  - 当一个进程正在运行时，它的状态在 CPU 寄存器和内存中
  - 当 OS 暂停一个进程的运行时，其设法将该进程 CPU 寄存器的值保存在该进程的 PCB / Kernel Stack 中 (在内存中)
  - 当 OS 新启动 / 恢复一个进程的执行时，从内存中恢复对应进程 CPU 寄存器的值
- 通常通过一段精心编写 (复杂晦涩) 的汇编指令来完成
  - xv6 中的 `swtch()`

上下文切换

user mode

kernel mode

timer interrupt
save A's user registers
move to kernel mode
jump to trap handler

running A

`swtch()`
save A's kernel registers
switch to B's kennel stack
restore B's kernel registers

running B

return from trap
restore B's user registers
move to user mode
jump to B's PC

```
// Saved registers for kernel context switches.
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

```
enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

xv6 (risc-v) context switch

```
.globl swtch
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)

        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret
```

```
# Context switch
#
#   void swtch(struct context *old, struct context *new);
#
# Save current registers in old. Load from new.
```

xv6 (risc-v)
context switch

A's User Stack

B's User Stack

`trapframe`

`trapframe`

save / restore
on `trap()`

A's saved
user registers

B's saved
user registers

A's Kernel Stack

B's Kernel Stack

…

…

A's PCB

B's PCB

`context`

`context`

save / restore
on `swtch()`

A's saved
kernel registers

B's saved
kernel registers

```
79      // give up the CPU if this is a timer interrupt.
80      if(which_dev == 2)
81        yield();
82
```

trap.c

Timer Interrupt

```
511   void
512   yield(void)
513   {
514     struct proc *p = myproc();
515     acquire(&p->lock);
516     p->state = RUNNABLE;
517     sched();
518     release(&p->lock);
519   }
```

proc.c

修改进程状态

```
490   void
491   sched(void)
492   {
493     int intena;
494     struct proc *p = myproc();
495
496     if(!holding(&p->lock))
497       panic("sched p->lock");
498     if(mycpu()->noff != 1)
499       panic("sched locks");
500     if(p->state == RUNNING)
501       panic("sched running");
502     if(intr_get())
503       panic("sched interruptible");
504
505     intena = mycpu()->intena;
506     swtch(&p->context, &mycpu()->context);
507     mycpu()->intena = intena;
508   }
```

proc.c

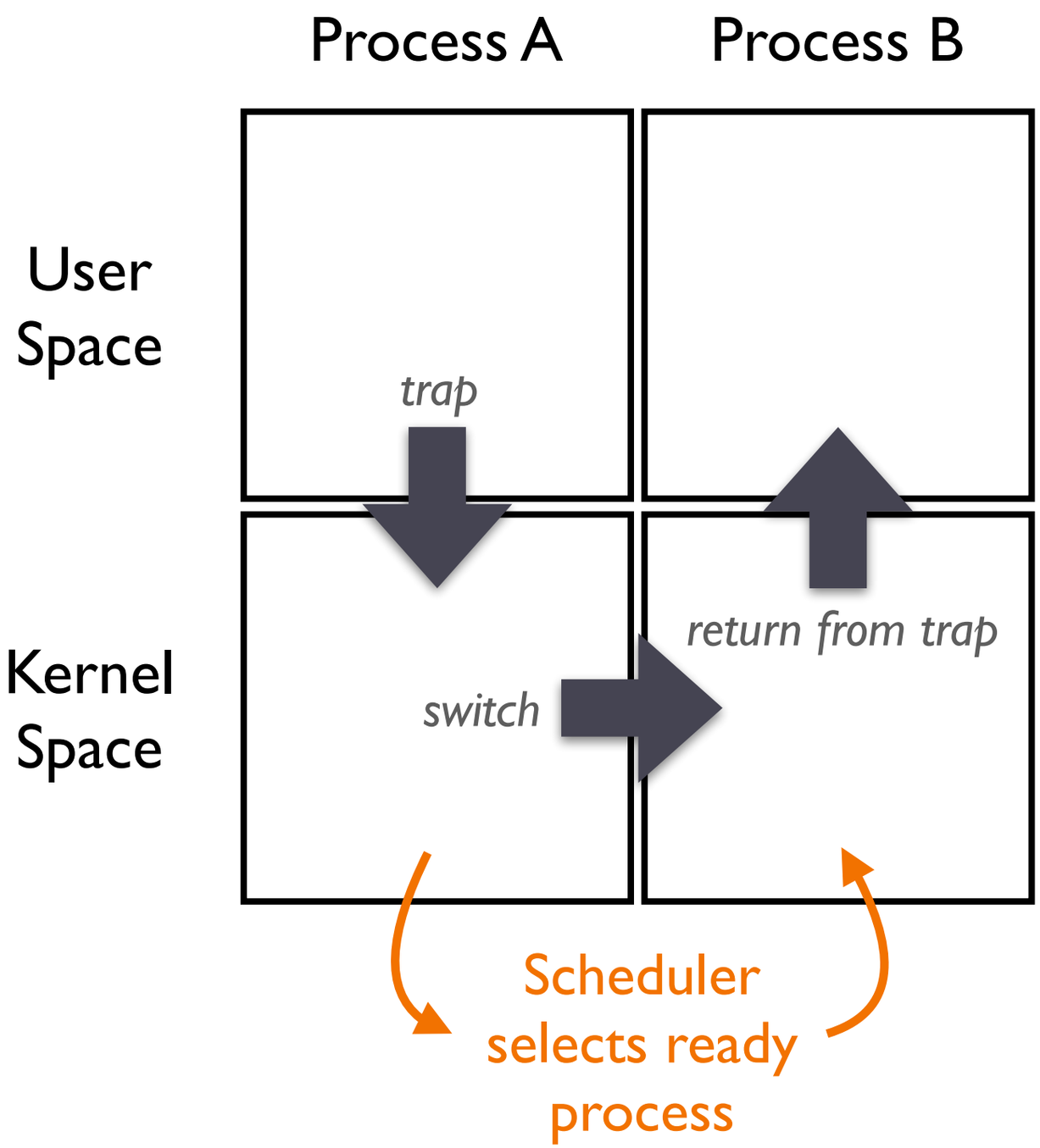
通过 swtch() 从进程 A 切换到 Scheduler

```
444   void
445   scheduler(void)
446   {
447     struct proc *p;
448     struct cpu *c = mycpu();
449
450     c->proc = 0;
451     for(;;){
452       // The most recent process to run may have had interrupts
453       // turned off; enable them to avoid a deadlock if all
454       // processes are waiting.
455       intr_on();
456
457       int found = 0;
458       for(p = proc; p < &proc[NPROC]; p++) {
459         acquire(&p->lock);
460         if(p->state == RUNNABLE) {
461           // Switch to chosen process.  It is the process's job
462           // to release its lock and then reacquire it
463           // before jumping back to us.
464           p->state = RUNNING;
465           c->proc = p;
466           swtch(&c->context, &p->context);
467
468           // Process is done running for now.
469           // It should have changed its p->state before coming back.
470           c->proc = 0;
471           found = 1;
472         }
473         release(&p->lock);
474       }
475       if(found == 0) {
476         // nothing to run; stop running on this core until an interrupt.
477         intr_on();
478         asm volatile("wfi");
479       }
480     }
481   }
```

proc.c

Scheduler 决定要运行的进程，
并通过 swtch() 切换到进程 B

上下文切换



模式切换和进程切换

- 进程 A 陷入内核，interrupt handler 保存其 user registers (user stack → kernel stack)
- 从运行进程 A 切换到运行进程 B (A's kernel stack → B's kernel stack)
- 从进程 B 内核态返回 (*return-from-trap*)，恢复 user registers (kernel stack → user stack)

重新思考进程

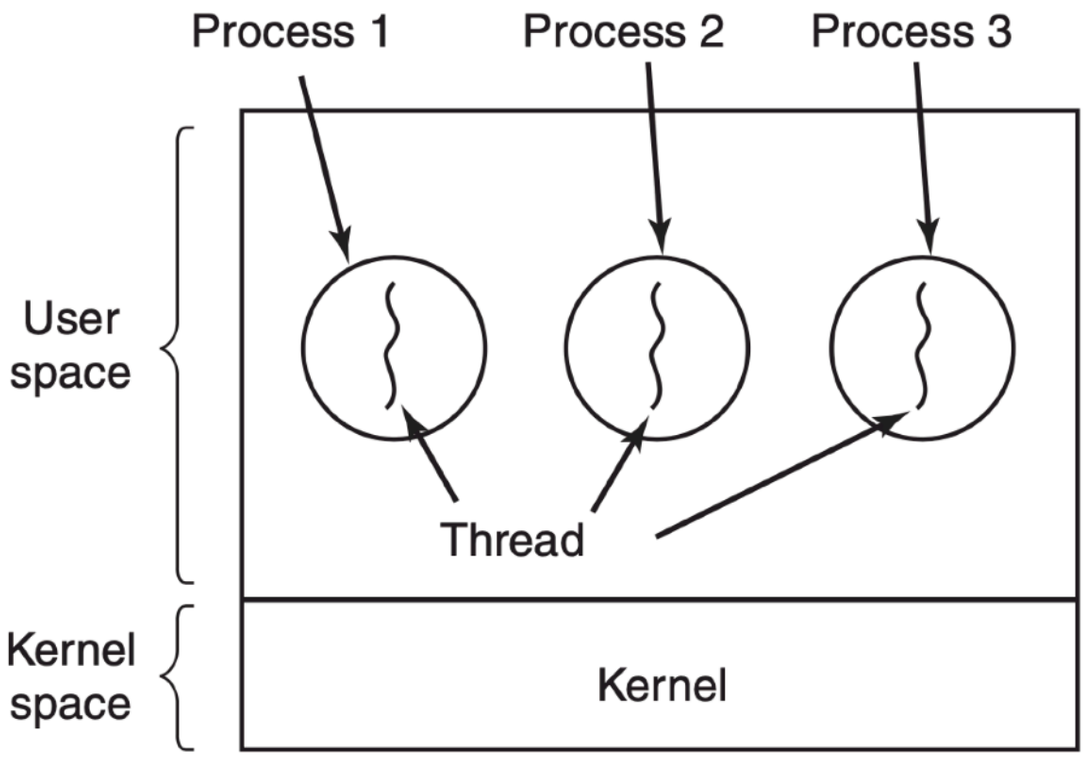
进程 (Process) 这一概念服务于两个目的:

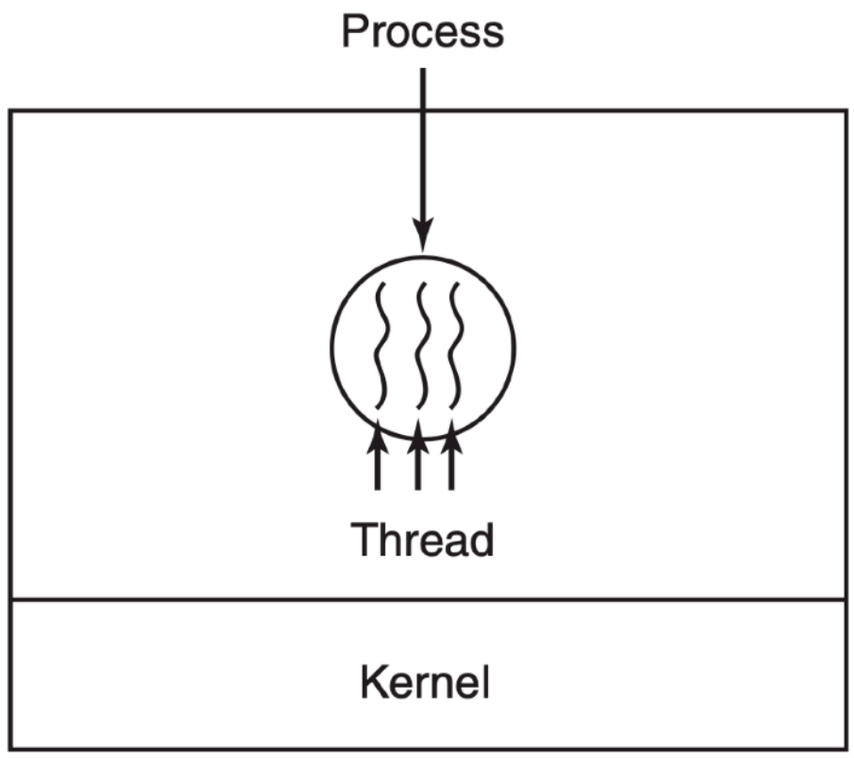
- 定义操作系统提供资源分配和隔离的粒度 (resource ownership)
  - 为不同进程分配不同的地址空间
  - 为不同进程维护不同的打开文件表
- 定义操作系统刻画并发执行以及提供调度的粒度 (execution)
  - 一个可以与其它任务交替执行的指令序列 (a stream of instructions)
  - 实现在不同指令序列之间的切换 (context switch)

线程

引入线程 (Thread) 来刻画一个可单独调度的指令执行序列

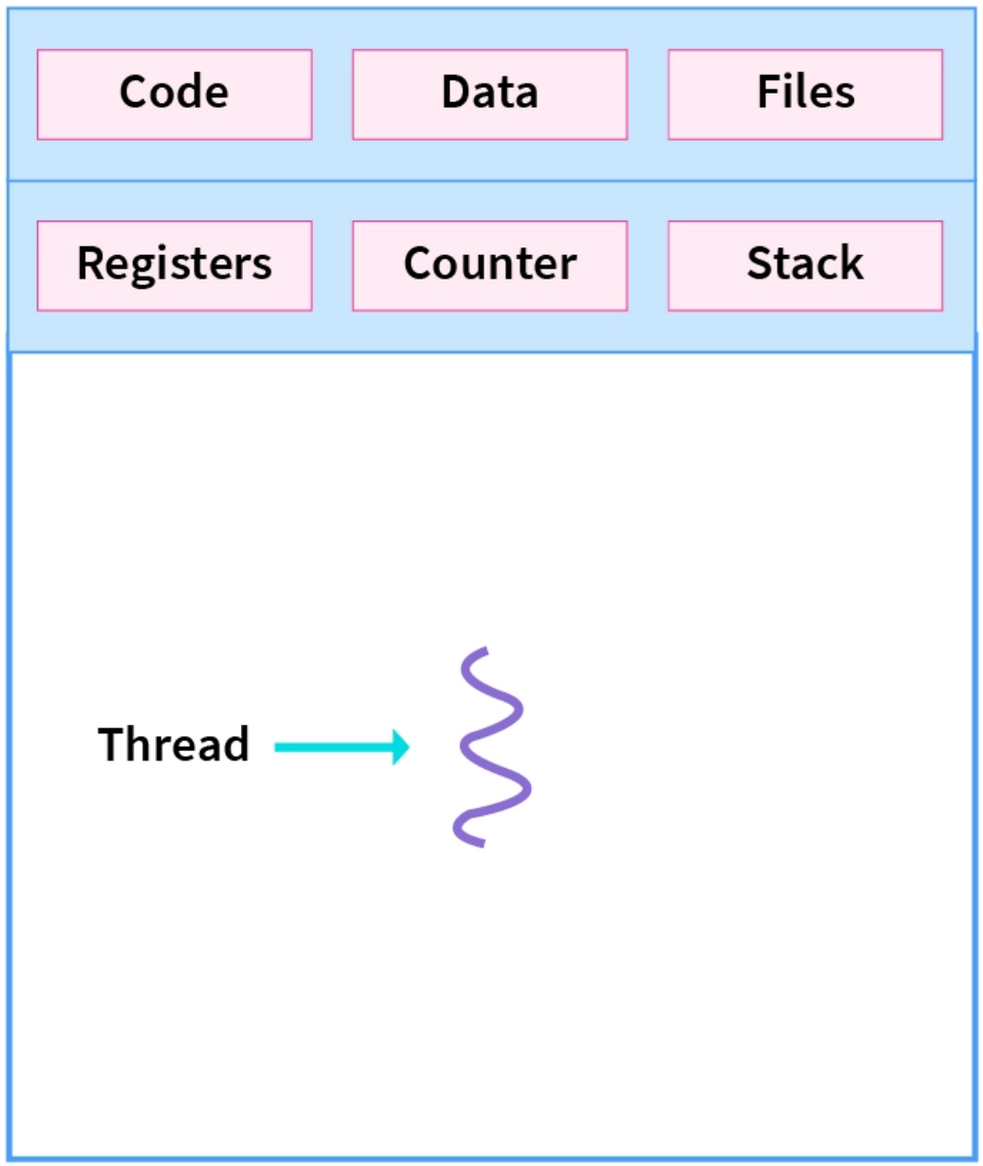
- 将进程概念中的资源分配和执行序列分开
  - 进程负责资源的分配和保护，线程负责指令序列的并发执行
  - 更好地刻画多处理器环境下的程序执行



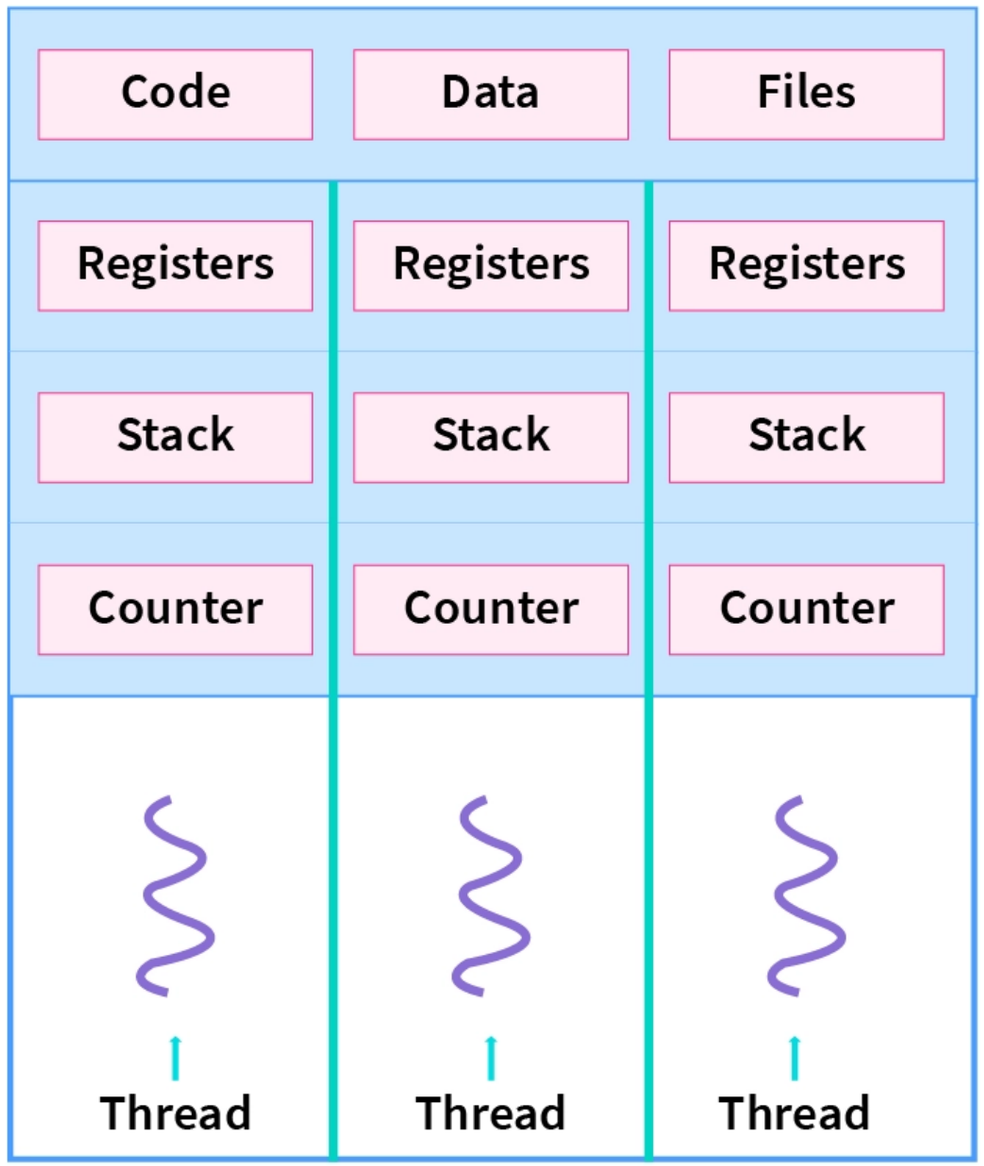


线程

引入线程 (Thread) 来刻画一个可单独调度的指令执行序列



Single-threaded process



Multithreaded process

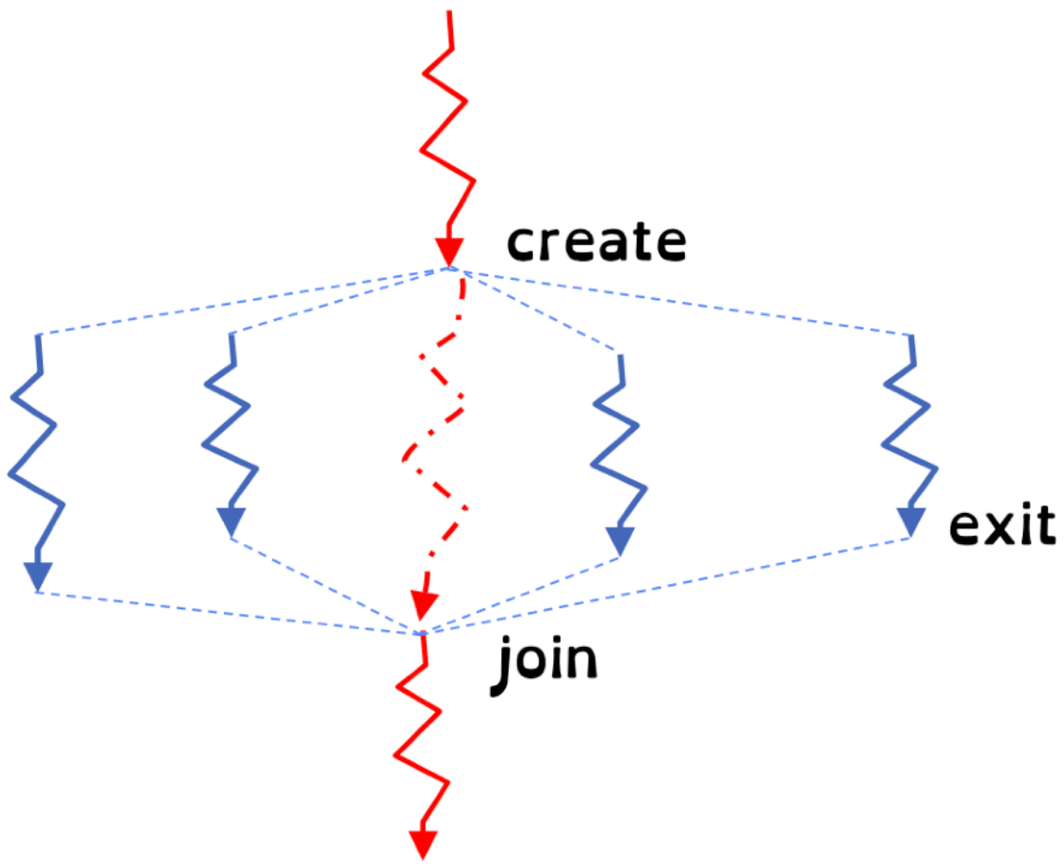
线程

- 每个线程代表一个可并发执行的任务
  - 有自己的 Program Counter / CPU Registers 和 Stack
  - 设计 Thread Control Block (TCB) 来存储线程 execution context
- 同一个进程的多个线程之间共享该进程的地址空间 (Address Space)
  - 所有线程共享该进程的 Code / Data、以及相关资源
  - 相比于进程，能更加高效的创建、终止和切换线程
  - 多个线程之间的通信也更加方便
    - 但对全局变量的修改会影响同一进程中的其他线程 **!!**

与线程相关的 APIs

POSIX Threads (pthreads)

- `pthread_create()`:
  创建一个新线程
- `pthread_exit()`:
  终止当前的线程执行
- `pthread_join()`:
  等待某个指定的线程执行结束
- `pthread_yield()`:
  放弃 CPU 以让其它线程执行

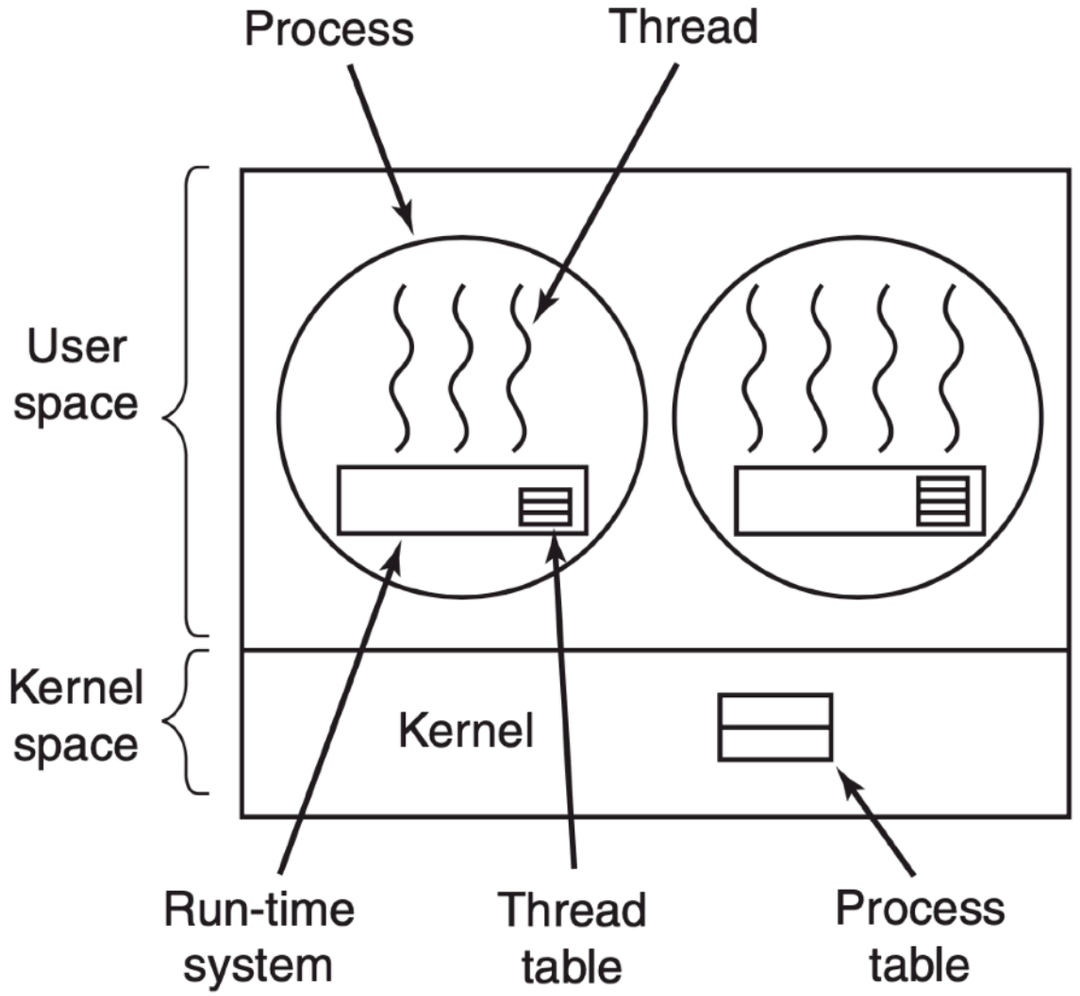


线程的实现

User-Level Thread

由应用程序支持的线程实现，对操作系统内核不可见

- 线程由运行在用户态的库函数来管理 (TCB in user space)
- 线程相关操作 (创建、销毁、切换和调度) 都发生在用户态
- Kernel 仅能感知进程

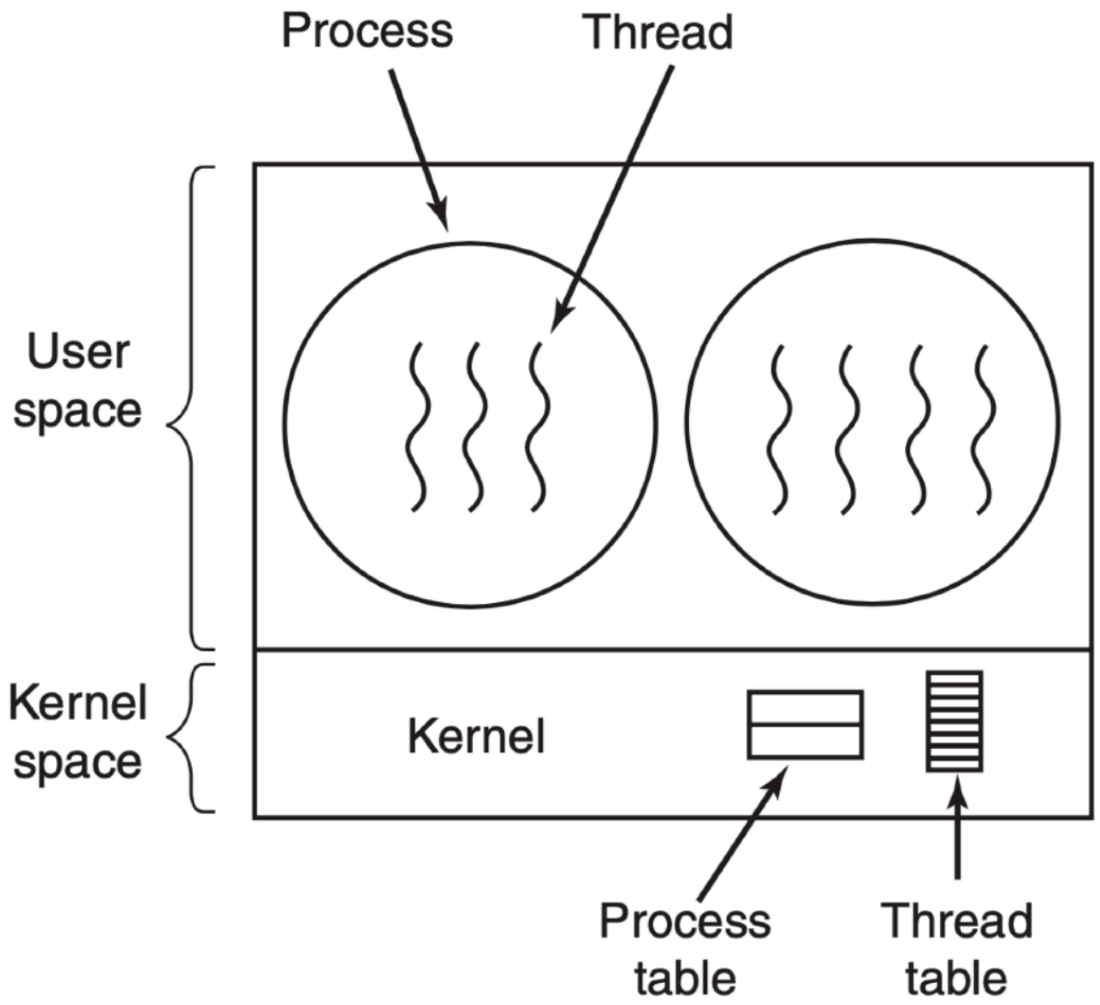


线程的实现

Kernel Thread

由操作系统内核支持的线程

- 线程由 Kernel 来管理 (TCB in kernel space)
- 线程相关操作 (创建、销毁、切换和调度) 通过系统调用实现



线程的实现

User-Level & Kernel Thread

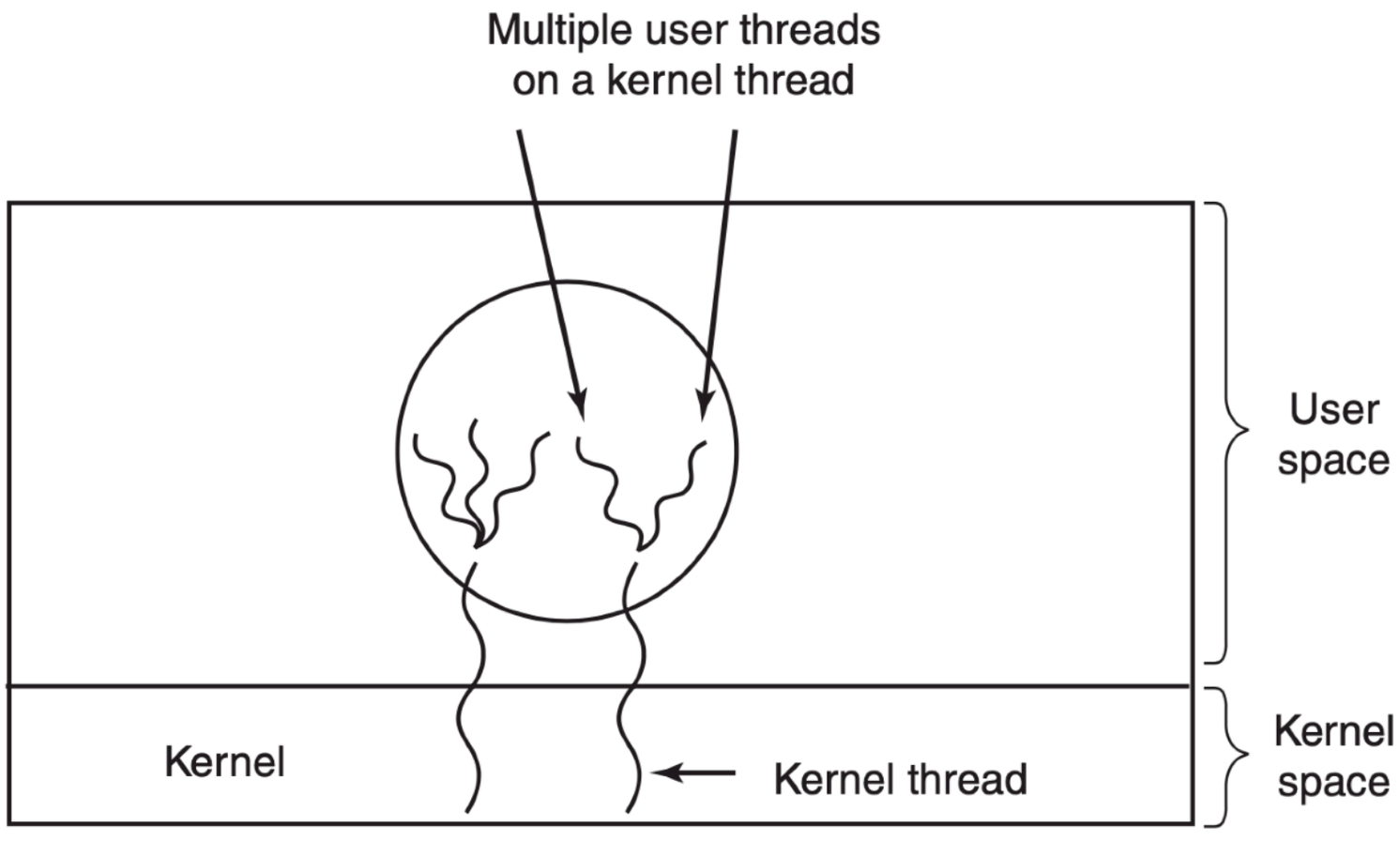
- 线程操作由用户态实现
  - 速度快 (不涉及模式转换)、可以在不支持线程的 OS 上实现
  - 每个进程仅有一个线程会被 OS 分配到 CPU 上执行
    - 当一个线程阻塞时，同一进程的所有其它线程也一起阻塞
      - 只有一个线程主动 yield 或 exit 时才有机会调度其它线程运行 (用户线程没有 timer interrupt)
- 线程操作由系统调用实现
  - 运行开销大 (涉及模式转换)
  - OS 可将多个线程分配到不同 CPU 上执行

线程的实现

Hybrid Implementation

将一个进程中的 N 个用户线程映射到 M ≤ N 个内核线程上

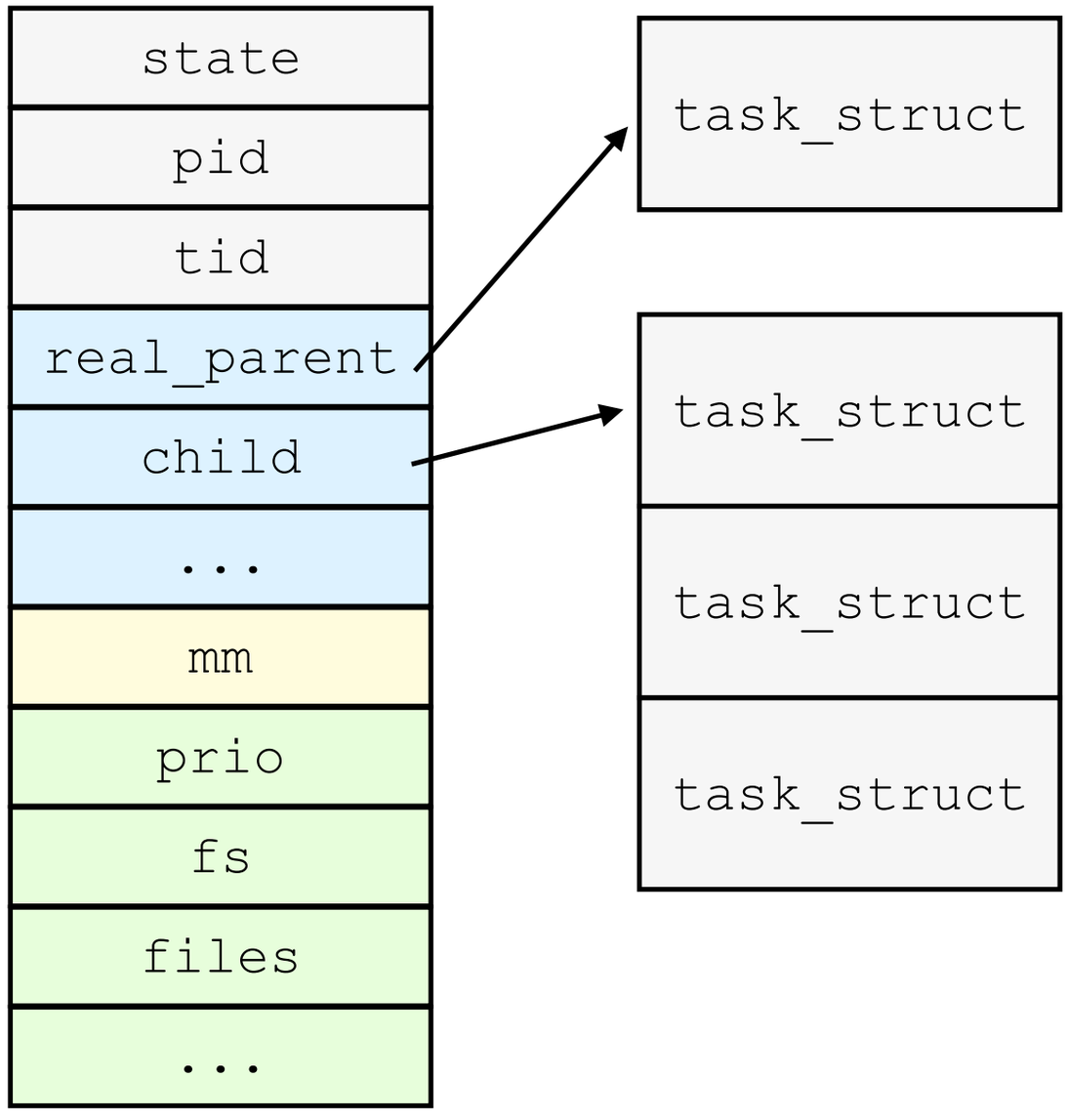
- 每个内核线程可以看作一个 “虚拟处理器”



Linux 中的线程

Linux 并不刻意区分进程和线程，而是使用 `task_struct` 数据结构来表示任一 Execution Context

- 如果两个 `task_struct` 共享同一地址空间，他们就是同一个进程中的两个线程
- 反之，就是两个不同进程



task_struct

Linux 中的线程

Linux 并不刻意区分进程和线程，而是使用 `task_struct` 数据结构来表示任一 Execution Context

- 在执行 `clone(func, stack_ptr, sharing_flags, arg)` 时
  - 根据 `sharing_flags` 确定新创建一个进程还是线程 (共享地址空间的 Light Weight Process)
- 同一个进程中的所有线程将分配相同的 PID，但具有不同的 LWP ID / Thread ID

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PARENT	New thread has same parent as the caller	New thread's parent is caller

总结

- 进程 (process)
  - 进程的基本概念 
  - 进程的创建和结束: `fork()`, `execve()`, `wait()`, `exit()` 
  - 重定向 (redirection) 和管道 (pipe) 的实现 
  - 进程的生命周期 (process state) 
  - 进程的上下文切换 (context switch)
- 线程 (thread)
  - 引入线程的动机、线程的基本模型 
  - 线程 user-level 和 kernel 实现的区别