操作系统接口

进程和内存

一个 xv6 的进程包含以下几个部分

- 用户空间的内存：指令、数据等
- 内核中的一个状态信息

当一个进程不执行的时候，xv6 将它运行时 CPU 的寄存器的值保存起来，当进程再次执行的时候再放回去。每一个进程有一个唯一的 identifier，我们称之为 PID

!!! note Fork 进程可以用系统调用 `fork` 创建一个新的进程。`fork` 使得新的进程和原来的进程有和调用者同样的内存内容。 `fork` 将结果同时返回给原来的进程和新的进程，原来的进程会收到新进程的 PID，新的进程会收到一个 0。

!!! note exec

系统调用 `exec` 会把调用它的进程的内存换成一个新从文件系统中的某个文件加载进来的进程。可执行文件要满足一定的格式，xv6 中应用的是 ELF 格式。当 `exec` 成功执行的时候，原始进程不会收到返回值，但是新的进程会收到参数，用法为

`int exec(char *file, char *argv[])`

!!! note xv6 的 Shell

代码位于 `user./sh.c` 中，总的来说，这个 Shell 分为三个部分

- 预处理器
    - 判断是不是 `cd` 之类的命令
- 命令构造器
    - 一堆结构体，一个套一个
- 命令执行器

Shell 大概是这样工作的

- 读一个用户命令
- fork
- fork 出来的子进程调用 exec
- 父进程 wait

!!! 用户空间的内存管理

在 xv6 中，用户空间的内存分配大多数是通过隐式的方式进行的，比如 `fork`、`exec`。当需要更多内存的时候，可以调用 `sbrk` 来实现。`sbrk` 也是一个系统调用，它会将进程的空间扩大 n，同时返回新开辟的内存的指针。

I/O 和文件描述符

一个文件描述符实际上就是一个整数，代表了一个由内核控制的对象，进程可能会对它进行读写之类的操作。一个继承可以通过打开文件，打开文件夹或访问一个设备、创建管道来获得一个文件描述符。

在 xv6 中，每个进程会拥有一个表，fd 就是这个表的一个索引，这个表的索引从零开始，为各个进程私有。默认进程会从文件描述符 0 读入，向文件描述符 1 输出，也就是标准输入和标准输出。标准错误输出是文件描述符 2。在 Shell 中，默认会确保自己有 3 个文件描述符。

!!! note read and write

`read` 和 `write` 这两个系统调用通过文件描述符从文件中读或写数据，

`read(fd, buf, n)` 从 fd 中读最多 n 个 byte，把它们扔到 5buf 中。每个文件描述符指向的文件都会维护一个 offset，`read` 会从 offset 所在处开始读取。

`write(fd, buf, n)` 会从 `buf` 中读入 n 个 byte 并将其写入 fd 中。返回值为所写入的 byte 数。当返回值小于 n 的时候，说明写入出现了错误。和 `read` 相同， `write` 也是从 offset 处开始的。

!!! note close

`close` 系统调用会释放一个文件描述符，使得它能够被其它的 `open`，`pipe`，`dup` 等系统调用所使用。新释放的文件描述符总是**当前进程中最小的未使用编号**

根据这一规则，下面这一段代码是合理的

```
char *argv[2];
argv[0] = "cat";
argv[1] = 0;
if(fork() == 0) {
    close(0);
    open("input.txt", O_RDONLY);
    exec("cat", argv);
}
```

首先将标准输入关掉，`open` 打开一个文件，根据上述规则，这个新的文件描述符会是代表标准输入的 0

在 xv6 的 shell 中，也是这样处理命令的执行的。

!!! note fcntl.h

这里保存了一系列参数，什么只读，只写之类的。

!!! note dup

系统调用 `dup` 会复制一个已经存在的文件描述符，也就是说返回一个新的文件描述符，这两个文件描述符共享同一个 offset，比如下面的代码会输出 offset

```
fd = dup(1);
write(1, "hello ", 6);
write(fd, "world\n", 6);
```

Pipes

书中是这么说的

> A pipe is a small kernel buffer exposed to processes as a pair of file descriptors, one for reading and one for writing.

管道也是进程间相互通信的一种办法：一个进程向管道一端写数据，另一个进程在管道的另一端读数据。

!!! wc 程序

用 `pipe(p)` 创建个管道，并将管道用于读写的文件描述符放在数组 p中。在 `fork` 之后，子进程调用 `close` 和 `dup` 把文件描述符 0 关闭，整成 `p[0]`，随后 close 和 `p[0...1]` 再用 `exec` 执行 wc 程序（数文件有多少行）

在很多场景中，管道都是比临时文件更加好用的功能，例如下面两个命令有相同的功能

!!! example Pipe and Redirection - `echo hello world | wc` - `echo hello world >/tmp/xyz; wc </tmp/xyz`

!!! note 管道相对临时文件的优势

- 可以被自动清理，不需要重定向到文件后再手动删除文件
- 可以传输任意长度的数据，但是文件重定向需要足够的磁盘空间来存储临时数据
- 方便并行
- 在进程内通信的时候，管道的分块读写比起文件读写更有效率

文件系统

xv6 的文件系统支持数据文件和文件夹，文件夹形成一个树形结构，起始是一个特殊的文件夹 `root`，和正常的认知一样，有路径，当前文件夹的概念

!!! chdir 系统调用 `chdir` 可以用来切换当前进程的当前文件夹

!!! mkdir, open and mknod

```
- mkdir: 创建一个新的文件夹
- open:  用 `O_CREATE` 的时候会创建一个新的文件
- `mknod` 创建一个指向设备的文件，需要给定设备编号
```

注意文件名和文件是不一样的，一个文件底层的“文件”，我们称之为 inode，一个 inode 可以有很多个名字，我们称之为 links。每个 link 由一个文件夹中的一个 entry（一个file name 和一个 inode） 组成，inode 存储了文件的元数据，包括类型（文件还是文件夹还是设备？）长度、磁盘上的位置、有关的 link 数目等。

!!! fstat

```
系统调用 `fstat` 会从一个文件描述符对应的 inode 中取出信息，并保存到一个结构体 `struct stat` 中，这个东西定义在 `stat.h` 中。
```

!!! link

```
系统调用 `link` 可以用来创建新的 link，比如下面的代码就会创建具有两个文件名的文件

```
open("a", O_CREATE|O_WRONLY);
link("a", "b");
```
```

!!! unlink

```
系统调用 `unlink` 会将一个文件名从文件系统中删除，但是文件的 inode 和硬盘占用还保留，除非没有相关的 link。
```

这些系统调用都是可以由用户层程序调用的，unix 类的系统将它们嵌入到了 shell 中。值得注意的是 cd 是一个例外，当 cd 的时候不应该 fork 一个子进程再改变它的当前路径，在 shell 的代码中对 cd 进行了特殊处理。

操作系统的组织

操作系统的一个重要需求就是支持同时进行多个任务。操作系统必须要做到在各个进程之间共享系统资源但是又不引起重读。例如在一个有多个 CPU 的设备上，操作系统必须保证所有的进程都得到运行的机会。操作系统同时还应该做好进程之间的隔离，也就是说一个进程出错的时候，整个操作系统不会受到严重的影响。虽然如此，操作系统不能将所有进程完全隔离开来，因为进程间的通信时必要的。

Xv6 系统执行在一个多核的 RISC-V 微处理器上，大多数底层的功能只适用于 RISC-V.

在我们使用的完整的计算机中，除了 CPU 还有许多周边设备，它们中的的许多都有 I/O 的需求和接口。xv6 支持 qemu 所摩西的一些涉笔，包括 RAM、存有启动代码的ROM、用户键盘和屏幕、硬盘等。

Abstracting Physical Resources

!!! Question

> **为什么不把操作系统做成一个 `library`? **
>
> 如果有多个进程，这样做的缺点就会显现出来。

为了实现 strong isolation，禁止应用访问敏感的硬件资源是必要的，取而代之的是，我们可以将硬件设备抽象成服务。例如在 Unix 中，我们不是直接读写磁盘上的文件，而是通过 `open`，`read` 之类的 system calls 来完成操作。

相似地，Unix 类系统隐藏了进程间切换，保存和读取状态寄存器等过程，使得应用程序不能感知到它在和其它程序一起执行。

另一个例子：Unix 进程用 exec 来建立自己的内存镜像，而不是直接和物理内存交互，这样使得操作系统能够决定将进程放在物理内存的那个位置，如果内存空间很紧张，操作系统甚至还可以将一个进程的数据等信息放在磁盘中。

Unix 类系统中很多形式的交互都依赖于文件描述符的读写，文件描述符也是一种高度的资源抽象。

Kernel organization

一个主要的设计问题就是操作系统的哪一个部分应该被允许在 supervisor mode 中运行。一种方式是将整个操作系统完全塞到 supervisor mode 中，这种组织方式我们称之为 monolithic kernel

在这种组织方式的操作系统拥有所有的硬件优先权限，同时操作系统的各个部分可以不受限制地进行合作，例如和以在文件系统和虚拟内存系统中不受限制地共享 buffer。

但是这种组织方式的缺点是各个部分之间的接口可能会十分复杂，因此更容易造成开发时的错误。值得关心的时，kernel mode 中的bug 通常是知名度，如果 kernel 寄了，计算机和上面的所有应用程序都会 fail，计算机必须重新启动。

为了解决这些问题，操作系统的设计者缩小了运行在 supervisor mode 中的系统代码的范围，这种内核组织方式我们称之为 微内核（microkernel）。

在一个以微内核方式组织的操作系统中，内核接口由底层的函数组成。例如启动应用程序，发送信息，和硬件设备交互等。这种组织方式使得内核可以相对比较简单。

Xv6 实际上是以 monolithic kernel 的范式实现的。因此 xv6 的内核接口对应着操作系统的接口，kernel 中实现了完整的操作系统。因此 xv6 没有提供服务机制。虽然 xv6 的内核很小，但是概念上确实是 monolithic kernel

Code: xv6 organization

注意各个模块间接口的代码放在 `kernel/defs.h` 中。

Process overview

xv6 采用了页式存储管理，每个进程都有自己的虚拟地址，页表将给出虚拟地址和物理地址之间的关系。

xv6 对于每个进程维护一个独立的页表，它定义了进程的地址空间。

xv6 对于每个进程还需要维护一些状态信息，都放在一个 `struct proc` 的结构体中。进程比较重要的内核状态，比如页表、内核栈、运行状态等，都保存在 `struct proc` 中，例如对于一个 `struct proc` 的指针 `p`，我们可以用 `p->pagetable` 访问到该进程的页表

每个进程又有一个执行的线程，用于执行进程的指令，一个线程可以被暂停和恢复。为了在进程之间能够做到不被察觉的切换，内核会暂停当前正在执行的线程并恢复另一个进程的线程。多数线程的状态信息存储于线程的栈上，每个进程一般有两个栈，一个 user stack，一个 kernel stack。

进程可以用过 RISC-V 的 `ecall` 指令进行系统调用。这个指令会提高程序的硬件优先级，并从此时切换到 内核栈并执行内核指令。当系统调用结束后，内核会切换回用户栈，

Code: starting xv6, the first process and system call

当一个 RISC-V 计算机通电的时候，他会初始化并运行一个存在ROM中的 boot loader。这个 boot loader 会将 xv6 内核加载到内存中，在 machine mode 中执行位于 `_entry` 的 xv6。此时页表系统还未启用，虚拟地址直接映射到物理地址。

在 `_entry` 处的代码会设置一个栈，使得 xv6 可以在上面运行 C 代码。Xv6 在 `start.c` 中声明了它的初始栈 `stack0`。这里的代码还会继续加载栈指针 `sp`。

函数 `start` 进行一些只在 machine mode 中允许的配置操作，之后切换到 supervisor mode 中。这一步靠的是 `mret` 这个 RISC-V 指令。并设置好进入 `main` 的寄存器。

切到 supervisor mode 之前，`start` 还要处理一下 timer interrupt，这在 设备驱动程序那一节也有提到。

随后程序跳入 `main`，这里对一些设别进行初始化，使用 `userinit` 创建第一个进程并在这个进程中执行一小段 RISC-V 汇编指令（`initcode.S`），执行第一个系统调用，把系统调用中的 `exec` 加载到寄存器中，调用 RISC-V 指令 `ecall` 回到 kernel

kernel 用寄存器中的数字来执行对应的系统调用。

一旦kernel完成了 `exec`，系统会在 `/init` 进程中返回用户空间，创建一个 console 设备并打开三个文件描述符，最后在 console 上启动 shell，系统启动成功。