

Table of Contents

概述

课程介绍	1.1
------	-----

文档翻译

book-riscv-rev1	2.1
第一章 操作系统接口	2.1.1
1.1 进程和内存	2.1.1.1
1.2 I/O和文件描述符	2.1.1.2
1.3 管道	2.1.1.3
1.4 文件系统	2.1.1.4
1.5 真实世界	2.1.1.5
1.6 练习	2.1.1.6
第二章 操作系统架构	2.1.2
2.1 抽象系统资源	2.1.2.1
2.2 用户态, 核心态, 以及系统调用	2.1.2.2
2.3 内核组织	2.1.2.3
2.4 代码: XV6架构篇	2.1.2.4
2.5 进程概述	2.1.2.5
2.6 代码: 启动XV6和第一个进程	2.1.2.6
2.7 真实世界	2.1.2.7
2.8 练习	2.1.2.8
第三章 页表	2.1.3
3.1 页式硬件	2.1.3.1
3.2 内核地址空间	2.1.3.2
3.3 代码: 创建一个地址空间	2.1.3.3
3.4 物理内存分配	2.1.3.4
3.5 代码: 物理内存分配	2.1.3.5
3.6 进程地址空间	2.1.3.6
3.7 代码: sbrk	2.1.3.7
3.8 代码: exec	2.1.3.8
3.9 真实世界	2.1.3.9
3.10 练习	2.1.3.10
第四章 陷阱指令和系统调用	2.1.4

4.1 RISC-V陷入机制	2.1.4.1
4.2 从用户空间陷入	2.1.4.2
4.3 代码：调用系统调用	2.1.4.3
4.4 系统调用参数	2.1.4.4
4.5 从内核空间陷入	2.1.4.5
4.6 页面错误异常	2.1.4.6
4.7 真实世界	2.1.4.7
4.8 练习	2.1.4.8
第五章 中断和设备驱动	2.1.5
5.1 代码：控制台输入	2.1.5.1
5.2 代码：控制台输出	2.1.5.2
5.3 驱动中的并发	2.1.5.3
5.4 定时器中断	2.1.5.4
5.5 真实世界	2.1.5.5
5.6 练习	2.1.5.6
第六章 锁	2.1.6
6.1 竞态条件	2.1.6.1
6.2 代码：Locks	2.1.6.2
6.3 代码：使用锁	2.1.6.3
6.4 死锁和锁排序	2.1.6.4
6.5 锁和中断处理函数	2.1.6.5
6.6 指令和内存访问排序	2.1.6.6
6.7 睡眠锁	2.1.6.7
6.8 真实世界	2.1.6.8
6.9 练习	2.1.6.9
第七章 调度	2.1.7
7.1 多路复用	2.1.7.1
7.2 代码：上下文切换	2.1.7.2
7.3 代码：调度	2.1.7.3
7.4 代码：mycpu和myproc	2.1.7.4
7.5 sleep与wakeup	2.1.7.5
7.6 代码：sleep和wakeup	2.1.7.6
7.7 代码：Pipes	2.1.7.7
7.8 代码：wait, exit和kill	2.1.7.8
7.9 真实世界	2.1.7.9
7.10 练习	2.1.7.10
第八章 文件系统	2.1.8
8.1 概述	2.1.8.1

8.2 Buffer cache层	2.1.8.2
8.3 代码: Buffer cache	2.1.8.3
8.4 日志层	2.1.8.4
8.5 日志设计	2.1.8.5
8.6 代码: 日志	2.1.8.6
8.7 代码: 块分配器	2.1.8.7
8.8 索引结点层	2.1.8.8
8.9 代码: Inodes	2.1.8.9
8.10 代码: Inode包含内容	2.1.8.10
8.11 代码: 目录层	2.1.8.11
8.12 代码: 路径名	2.1.8.12
8.13 文件描述符层	2.1.8.13
8.14 代码: 系统调用	2.1.8.14
8.15 真实世界	2.1.8.15
8.16 练习	2.1.8.16
Introduction	2.2
使用GNU Debugger	2.3
C Pointers, gdb	2.4
Calling Convention	2.5
Journaling the Linux ext2fs Filesystem	2.6

实验记录

版本控制	3.1
实验内容	3.2
Lab1: Xv6 and Unix utilities	3.2.1
Lab2: System calls	3.2.2
Lab3: Page tables	3.2.3
Lab4: Traps	3.2.4
Lab5: Xv6 lazy page allocation	3.2.5
Lab6: Copy-on-Write Fork for xv6	3.2.6
Lab7: Multithreading	3.2.7
Lab8: Locks	3.2.8
Lab9: File system	3.2.9
Lab10: Mmap	3.2.10
Lab11: Network	3.2.11
实验解析	3.3
Lab1: Util	3.3.1
Lab2: Syscall	3.3.2

Lab4: Traps	3.3.3
Lab5: Xv6 lazy page allocation	3.3.4
Lab6: Copy-on-Write Fork for xv6	3.3.5
Lab7: Multithreading	3.3.6
Lab8: Locks	3.3.7
Lab9: File system	3.3.8
Lab10: Mmap	3.3.9

- 课程介绍
- 常用网址
- [GITBOOK](#)浏览
- [BY ME A COFFEE](#)

课程介绍

6.S081 Fall2020是麻省理工2020年秋季的操作系统课程，MIT将学习相关的资源全部公开并放到了官网。本课程中共涉及11个实验，需要花费一定时间来完成。由于是国外的课程，文档资料均为英文，为方便自己后续查阅和其他英文水平不足以流畅阅读英文文献的同学也能上手本课程，计划将资料全部翻译为中文。并分享课程笔记和实验记录。

由于水平有限，翻译中难免有错误或词不达意，还请见谅。

The screenshot shows the homepage of the 6.S081 course. At the top, there's a dark navigation bar with the course name and links to Schedule, Class, Labs, xv6, References, and Piazza. The year '2020' is also present. Below the bar, the main content area has a title 'Xv6, a simple Unix-like teaching operating system' and a sub-section 'Introduction'. It includes a brief description of Xv6 and its porting to RISC-V. There are two sections for 'Xv6 sources and text', each with a code block showing Git clone commands. The first command is for the xv6 source code, and the second is for the xv6 book.

常用网址

- 课程官网：[6.S081 Fall 2020](#)
- 课程视频：[6.S081--bilibili](#)
- 视频翻译：[6.S081课程翻译--gitbook](#)
- 我的实验仓库：[xv6-labs-2020--Github](#)

GITBOOK浏览

更好的浏览体验，请查看[6.S081-All-In-One-Gitbook\(xv6.dgs.zone\)](#)

这些笔记开始时写在了飞书上，飞书不支持导出markdown，因此是从飞书上逐篇复制下来的，二者存在一些格式差异，我已经进行了修改，但仍然可能有部分没有注意到~

BY ME A COFFEE

如果你觉得翻译有用而且愿意的话，欢迎请我一杯咖啡或者是一包辣条（doge）



copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间： 2022-02-24 11:55:45

- [BOOK-RISCV-REV1](#)

BOOK-RISCV-REV1

XV6: 一个简单，类UNIX的教学用操作系统

本书是6.S081最重要的参考书目，其中详细的介绍了XV6的设计细节，在课程前往往需要阅读对应章节，做实验时也要回来查看。

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间： 2021-08-19 09:43:44

第一章 操作系统接口

操作系统的任务是在多个程序之间共享一台计算机，并提供比硬件本身支持的更有用的服务。操作系统管理和抽象底层硬件，例如文字处理器不需要关心使用哪种类型的磁盘硬件。一个操作系统在多个程序之间共享硬件，这样它们就可以(或者看起来可以)同时运行。最后，操作系统为程序提供了可控的交互方式，这样它们就可以共享数据或者一起工作。

操作系统通过接口向用户程序提供服务。设计良好的接口是很困难的。一方面，我们希望接口简单明了，因为这样更利于正确使用。另一方面，我们可能倾向于为应用程序提供许多复杂的特性。解决这个问题的诀窍在于设计接口时，依赖一些可结合的机制，以此来提供更好的通用性。

本书使用单一的操作系统作为具体的例子来说明操作系统的概念。**xv6**这个操作系统提供了Ken Thompson和Dennis Ritchie的Unix介绍的基本接口，并且模仿了Unix的内部设计。Unix提供了一个窄接口，其机制表现突出，提供了令人惊讶的通用程度。这个接口非常成功，甚至现代操作系统BSD、Linux、Mac OSX、Solaris，甚至在一定程度上，Microsoft windows都有类Unix的接口。理解**xv6**是理解这些系统和其他系统的一个良好开端。

如下图1.1所示，**xv6**采用传统的内核形式（内核是一个特殊的程序，为正在运行的程序提供服务）。每个正在运行的程序，称为进程，都有包含指令、数据和堆栈的内存。指令实现了程序的运算，数据是计算所依赖的变量，堆栈组织程序的过程调用。一台给定的计算机通常有许多进程，但只有一个内核。

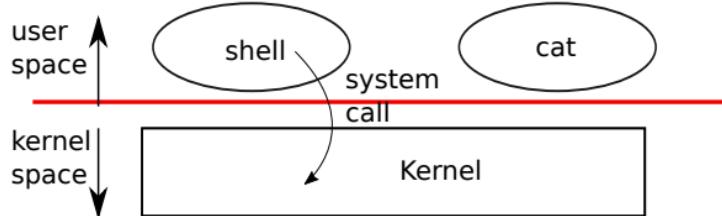


Figure 1.1: A kernel and two user processes.

当一个进程需要调用一个内核服务时，它会调用一个系统调用，这是操作系统接口中的一个调用。系统调用进入内核；内核执行服务并返回。因此，一个进程在用户空间和内核空间之间交替执行。

内核使用CPU提供的硬件保护机制来确保每个在用户空间执行的进程只能访问它自己的内存。内核程序的执行拥有操控硬件的权限，它需要实现这些保护；而用户程序执行时没有这些特权。当用户程序调用系统调用时，硬件会提升权限级别，并开始执行内核中预先安排好的函数。

内核提供的系统调用集合是用户程序看到的接口。**Xv6**内核提供了Unix内核传统上提供的服务和系统调用的子集。表1.2列出了**xv6**的所有系统调用。

系统调用	描述
<code>int fork()</code>	创建一个进程，返回子进程的PID
<code>int exit(int status)</code>	终止当前进程，并将状态报告给 <code>wait()</code> 函数。 无返回
<code>int wait(int *status)</code>	等待一个子进程退出；将退出状态存入 <code>*status</code> ； 返回子进程PID。
<code>int kill(int pid)</code>	终止对应PID的进程，返回0，或返回-1表示错误
<code>int getpid()</code>	返回当前进程的PID
<code>int sleep(int n)</code>	暂停n个时钟节拍
<code>int exec(char *file, char *argv[])</code>	加载一个文件并使用参数执行它；只有在出错时才返回
<code>char *sbrk(int n)</code>	按n字节增长进程的内存。返回新内存的开始
<code>int open(char *file, int flags)</code>	打开一个文件； <code>flags</code> 表示read/write；返回一个fd（文件描述符）
<code>int write(int fd, char *buf, int n)</code>	从buf写n个字节到文件描述符fd；返回n
<code>int read(int fd, char *buf, int n)</code>	将n个字节读入buf；返回读取的字节数；如果文件结束，返回0
<code>int close(int fd)</code>	释放打开的文件fd
<code>int dup(int fd)</code>	返回一个新的文件描述符，指向与fd相同的文件
<code>int pipe(int p[])</code>	创建一个管道，把read/write文件描述符放在p[0]和p[1]中
<code>int chdir(char *dir)</code>	改变当前的工作目录
<code>int mkdir(char *dir)</code>	创建一个新目录
<code>int mknod(char *file, int, int)</code>	创建一个设备文件
<code>int fstat(int fd, struct stat *st)</code>	将打开文件fd的信息放入*st
<code>int stat(char *file, struct stat *st)</code>	将指定名称的文件信息放入*st
<code>int link(char *file1, char *file2)</code>	为文件file1创建另一个名称(file2)
<code>int unlink(char *file)</code>	删除一个文件

表1.2: xv6系统调用（除非另外声明，这些系统调用返回0表示无误，返回-1表示出错）

本章的其余部分概述了xv6的服务——进程、内存、文件描述符、管道和文件系统——并用代码片段和关于**shell**（Unix的命令行用户界面）如何使用它们的讨论来阐释。**Shell**对系统调用的使用说明了它们是如何被精心设计的。

Shell是一个普通的程序，它从用户那里读取命令并执行它们。**Shell**是一个用户程序，而不是内核的一部分，这一事实说明了系统调用接口的强大之处：**shell**没有什么特别之处。这也意味着**shell**很容易替换；因此，现代Unix系统有多种**shell**可供选择，每种**shell**都有自己的用户界面和脚本特性。**Xv6 Shell**是Unix Bourne shell本质的简单实现。它的实现可以在([user/sh.c:1](#))中找到.

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2022-04-09 17:41:58

- 1.1 进程和内存

1.1 进程和内存

Xv6进程由用户空间内存(指令、数据和堆栈)和对内核私有的每个进程状态组成。

Xv6分时进程: 它透明地在等待执行的进程集合中切换可用的CPU。当一个进程没有执行时，xv6保存它的CPU寄存器，并在下一次运行该进程时恢复它们。内核利用进程id或PID标识每个进程。

一个进程可以使用fork系统调用创建一个新的进程。Fork创建了一个新的进程，其内存内容与调用进程（称为父进程）完全相同，称其为子进程。Fork在父子进程中都返回值。在父进程中，fork返回子类的PID；在子进程中，fork返回零。例如，考虑下面用C语言编写的程序片段

```
// fork()在父进程中返回子进程的PID
// 在子进程中返回0
int pid = fork();
if(pid > 0) {
    printf("parent: child=%d\n", pid);
    pid = wait((int *) 0);
    printf("child %d is done\n", pid);
} else if(pid == 0) {
    printf("child: exiting\n");
    exit(0);
} else {
    printf("fork error\n");
}
```

`exit` 系统调用导致调用进程停止执行并释放资源（如内存和打开的文件）。`exit` 接受一个整数状态参数，通常0表示成功，1表示失败。`wait` 系统调用返回当前进程的已退出(或已杀死)子进程的PID，并将子进程的退出状态复制到传递给 `wait` 的地址；如果调用方的子进程都没有退出，那么`wait`等待一个子进程退出。如果调用者没有子级，`wait` 立即返回-1。如果父进程不关心子进程的退出状态，它可以传递一个0地址给 `wait`。

在这个例子中，输出

```
parent: child=1234
child: exiting
```

可能以任何一种顺序出来，这取决于父或子谁先到达 `printf` 调用。子进程退出后，父进程的 `wait` 返回，导致父进程打印

```
parent: child 1234 is done
```

尽管最初子进程与父进程有着相同的内存内容，但是二者在运行中拥有不同的内存空间和寄存器：在一个进程中改变变量不会影响到另一个进程。例如当 `wait` 的返回值存入父进程的变量 `pid` 中时，并不会影响子进程中的 `pid`，子进程中 `pid` 仍然为0。

`exec` 系统调用使用从文件系统中存储的文件所加载的新内存映像替换调用进程的内存。（百度百科：根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件）该文件必须有特殊的格式，它指定文件的哪部分存放指令，哪部分是数据，以及哪一条指令用于启动等等。`xv6` 使用 `ELF` 格式（将会在第三章详细讨论）。当 `exec` 执行成功，它不向调用进程返回数据，而是使加载自文件的指令在 `ELF header` 中声明的程序入口处开始执行。`exec` 有两个参数：可执行文件的文件名和字符串参数数组。例如

```
char* argv[3];
argv[0] = "echo";
argv[1] = "hello";
argv[2] = 0;
exec("/bin/echo", argv);
printf("exec error\n");
```

这个代码片段将调用程序替换为了参数列表为 `echo hello` 的 `/bin/echo` 程序运行，多数程序忽略参数数组中的第一个元素，它通常是程序名。

`xv6` 的 `shell` 使用上述调用为用户运行程序。`shell` 的主要结构很简单，请参见 `main (user/sh.c:145)`。主循环使用 `getcmd` 函数从用户的输入中读取一行，然后调用 `fork` 创建一个 `shell` 进程的副本。父进程调用 `wait`，子进程执行命令。例如：当用户向 `shell` 输入 `echo hello` 时，`runcmd (user/sh.c:58)` 将以 `echo hello` 为参数被调用来执行实际命令。对于“`echo hello`”，它将调用 `exec (user/sh.c:78)`。如果 `exec` 成功，那么子进程将从 `echo` 而不是 `runcmd` 执行命令，在某刻 `echo` 会调用 `exit`，这将导致父进程从 `main (user/sh.c:78)` 中的 `wait` 返回。

你或许想知道为什么 `exec` 和 `fork` 没有组合成为一个系统调用，稍后我们将会看到 `shell` 在其 I/O 重定向的实现中利用了这种分离。为了避免创建一个重复的进程然后立即替换它(使用 `exec`)的浪费，操作内核通过使用虚拟内存技术(如 `copy-on-write`)优化 `fork` 在这个用例中的实现(见第 4.6 节)。

`Xv6` 隐式地分配大多数用户空间内存：`fork` 分配父内存的子副本所需的内存，`exec` 分配足够的内存来保存可执行文件。在运行时需要更多内存的进程(可能是 `malloc`)可以调用 `sbrk(n)` 将其数据内存增加 `n` 个字节；`sbrk` 返回新内存的位置。

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间：2021-08-18 11:55:13

- 1.2 I/O和文件描述符

1.2 I/O和文件描述符

文件描述符是一个小整数(**small integer**)，表示进程可以读取或写入的由内核管理的对象。进程可以通过打开一个文件、目录、设备，或创建一个管道，或复制一个已存在的描述符来获得一个文件描述符。为了简单起见，我们通常将文件描述符所指的对象称为“文件”；文件描述符接口将文件、管道和设备之间的差异抽象出来，使它们看起来都像字节流。我们将输入和输出称为 I/O。

在内部，**xv6**内核使用文件描述符作为每个进程表的索引，这样每个进程都有一个从零开始的文件描述符的私有空间。按照惯例，进程从文件描述符0读取（标准输入），将输出写入文件描述符1（标准输出），并将错误消息写入文件描述符2（标准错误）。正如我们将看到的，**shell**利用这个约定来实现I/O重定向和管道。**shell**确保它始终有三个打开的文件描述符（**user/sh.c:151**），这是控制台的默认文件描述符。

`read` 和 `write` 系统调用以字节为单位读取或写入已打开的以文件描述符命名的文件。`read(fd, buf, n)` 从文件描述符`fd`读取最多`n`字节，将它们复制到`buf`，并返回读取的字节数，引用文件的每个文件描述符都有一个与之关联的偏移量。`read` 从当前文件偏移量开始读取数据，然后将该偏移量前进所读取的字节数：（也就是说）后续读取将返回第一次读取返回的字节之后的字节。当没有更多的字节可读时，`read` 返回0来表示文件的结束。

系统调用 `write(fd, buf, n)` 将`buf`中的`n`字节写入文件描述符，并返回写入的字节数。只有发生错误时才会写入小于`n`字节的数据。与读一样，`write` 在当前文件偏移量处写入数据，然后将该偏移量向前推进写入的字节数：每个 `write` 从上一个偏移量停止的地方开始写入。

以下程序片段（构成程序 `cat` 的本质）将数据从其标准输入复制到其标准输出。如果发生错误，它将消息写入标准错误：

```
char buf[512];
int n;
for (;;) {
    n = read(0, buf, sizeof buf);
    if (n == 0)
        break;
    if (n < 0) {
        fprintf(2, "read error\n");
        exit(1);
    }
    if (write(1, buf, n) != n) {
        fprintf(2, "write error\n");
        exit(1);
    }
}
```

代码片段中需要注意的重要一点是，`cat` 不知道它是从文件、控制台还是管道读取。同样也不知道它是打印到控制台、文件还是其他什么地方。文件描述符的使用以及文件描述符0是输入而文件描述符1是输出的约定允许了 `cat` 的简单实现。

`close` 系统调用释放一个文件描述符，使其可以被未来使用的 `open`、`pipe` 或 `dup` 系统调用重用（见下文）。新分配的文件描述符总是当前进程中编号最小的未使用描述符。

文件描述符和 `fork` 相互作用，使 I/O 重定向更容易实现。`fork` 复制父进程的文件描述符表及其内存，以便子级以与父级在开始时拥有完全相同的打开文件。系统调用 `exec` 替换了调用进程的内存，但保留其文件表。此行为允许 `shell` 通过 `fork` 实现 I/O 重定向，在子进程中重新打开选定的文件描述符，然后调用 `exec` 来运行新程序。下面是 `shell` 运行命令 `cat < input.txt` 的代码的简化版本。

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

在子进程关闭文件描述符 0 之后，`open` 保证使用新打开的 `input.txt`: 0 的文件描述符作为最小的可用文件描述符。`cat` 然后执行文件描述符 0 (标准输入)，但引用的是 `input.txt`。父进程的文件描述符不会被这个序列改变，因为它只修改子进程的描述符。

Xv6shell 中的 I/O 重定向代码就是这样工作的(`user/sh.c:82`)。回想一下，在代码执行到这里时，`shell` 已经 `fork` 出了子 `shell`，`runcmd` 将调用 `exec` 来加载新程序。

`open` 的第二个参数由一组标志组成，这些标志以位表示，用于控制打开的操作。可能的值定义在文件控制(`fcntl`)头文件(`kernel/fcntl.h:1-5`)中

宏定义	功能说明
O_RDONLY	只读
O_WRONLY	只写
O_RDWR	可读可写
O_CREATE	如果文件不存在则创建文件
O_TRUNC	将文件截断为零长度

现在应该很清楚为什么 `fork` 和 `exec` 分离的用处了：在这两个调用之间，`shell` 有机会对子进程进行 I/O 重定向，而不会干扰主 `shell` 的 I/O 设置。我们可以想象一个假设的 `forkexec` 系统调用组合，但是用这样的调用进行 I/O 重定向是很笨拙的。`Shell` 可以在调用 `forkexec` 之前修改自己的 I/O 设置(然后撤销这些修改);或者 `forkexec` 可以将 I/O 重定向的指令作为参数;或者(最不吸引人的是)可以让每个程序(如 `cat`)执行自己的 I/O 重定向。

尽管 `fork` 复制了文件描述符表，但是每个基础文件偏移量在父文件和子文件之间是共享的，比如下面的程序：

```
if (fork() == 0) {
    write(1, "hello ", 6);
    exit(0);
} else {
    wait(0);
    write(1, "world\n", 6);
}
```

在这个片段的末尾，附加到文件描述符1的文件将包含数据 `hello world`。父进程中的写操作(由于等待，只有在子进程完成后才运行)在子进程停止写入的位置进行。这种行为有助于从shell命令序列产生顺序输出，比如 `(echo hello;echo world)>output.txt`。

`dup` 系统调用复制一个现有的文件描述符，返回一个引用自同一个底层I/O对象的新文件描述符。两个文件描述符共享一个偏移量，就像`fork`复制的文件描述符一样。这是另一种将“hello world”写入文件的方法：

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

如果两个文件描述符是通过一系列 `fork` 和 `dup` 调用从同一个原始文件描述符派生出来的，那么它们共享一个偏移量。否则，文件描述符不会共享偏移量，即使它们来自于对同一文件的打开调用。`dup` 允许shell执行这样的命令：`ls existing-file non-existing-file > tmp1 2>&1`。`2>&1` 告诉shell给命令的文件描述符2是描述符1的副本。现有文件的名称和不存在文件的错误信息都会显示在tmp1文件中。**Xv6 shell**不支持错误文件描述符的I/O重定向，但是现在你知道如何实现它了。

文件描述符是一个强大的抽象，因为它们隐藏了它们所连接的细节：写入文件描述符1的进程可能写入文件、设备（如控制台）或管道。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-18 11:58:39

- 1.3 管道

1.3 管道

管道是作为一对文件描述符公开给进程的小型内核缓冲区，一个用于读取，一个用于写入。将数据写入管道的一端使得这些数据可以从管道的另一端读取。管道为进程提供了一种通信方式。

下面的示例代码使用连接到管道读端的标准输入来运行程序 `wc`。

```
int p[2];
char *argv[2];
argv[0] = "wc";
argv[1] = 0;
pipe(p);
if (fork() == 0) {
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    exec("/bin/wc", argv);
} else {
    close(p[0]);
    write(p[1], "hello world\n", 12);
    close(p[1]);
}
```

程序调用 `pipe`，创建一个新的管道，并在数组 `p` 中记录读写文件描述符。

在 `fork` 之后，父子进程都有指向管道的文件描述符。子进程调用 `close` 和 `dup` 使文件描述符 0 指向管道的读取端（前面说过优先分配最小的未使用的描述符），然后关闭 `p` 中所存的文件描述符，并调用 `exec` 运行 `wc`。当 `wc` 从它的标准输入读取时，就是从管道读取。父进程关闭管道的读取端，写入管道，然后关闭写入端。

如果没有可用的数据，则管道上的 `read` 操作将会进入等待，直到有新数据写入或所有指向写入端的文件描述符都被关闭，在后一种情况下，`read` 将返回 0，就像到达数据文件的末尾一样。事实上，`read` 在新数据不可能到达前会一直阻塞，这是子进程在执行上面的 `wc` 之前关闭管道的写入端非常重要的一个原因：如果 `wc` 的文件描述符之一指向管道的写入端，`wc` 将永远看不到文件的结束。

Xv6 shell 以类似于上面代码(`user/sh.c:100`)的方式实现了诸如 `grep fork sh.c | wc -l` 之类的管道。子进程创建一个管道将管道的左端和右端连接起来。然后对管道的左端调用 `fork` 和 `runcmd`，对管道的右端调用 `fork` 和 `runcmd`，并等待两者都完成。管道的右端可能是一个命令，该命令本身包含一个管道(例如，`a | b | c`)，该管道本身 `fork` 为两个新的子进程(一个用于 `b`，一个用于 `c`)。因此，`shell` 可以创建一个进程树。这个树的叶子是命令，内部节点是等待左右两个子进程完成的进程。

原则上，可以让内部节点在管道的左端运行，但是正确地这样做会使实现复杂化。考虑进行以下修改：将 `sh.c` 更改为不对 `p->left` 进行 `fork`，并在内部进程中运行 `runcmd(p->left)`。然后，例如，`echo hi | wc` 将不会产生输出，因为当 `echo hi` 在 `runcmd` 中退出时，内部进程将退出，而不会调用 `fork` 来运行管道的右端。这个不正确的行为可以通过不调用内部进程的 `runcmd` 中的 `exit` 来修复，但是这个修复使代码复杂化：现在 `runcmd` 需要知道它是否是一个内部进程。同样的，当没有对 `(p->right)` 执行 `fork` 时也会更加复杂。例如，只需进行上述的修改，`sleep 10 |`

`echo hi` 将立即打印“hi”，而不是在10秒后，因为 `echo` 将立即运行并退出，而不是等待 `sleep` 完成。因为**sh.c**的目标是尽可能的简单，所以它不会试图避免创建内部进程。

管道看起来并不比临时文件更强大：下面的管道命令行

```
echo hello world | wc
```

可以不通过管道实现，如下

```
echo hello world > /tmp/xyz; wc < /tmp/xyz
```

在这种情况下，管道相比临时文件至少有四个优势

- 首先，管道会自动清理自己；在文件重定向时，`shell`使用完 `/tmp/xyz` 后必须小心删除
- 其次，管道可以任意传递长的数据流，而文件重定向需要磁盘上足够的空闲空间来存储所有的数据。
- 第三，管道允许并行执行管道阶段，而文件方法要求第一个程序在第二个程序启动之前完成。
- 第四，如果实现进程间通讯，管道的块读写比文件的非块语义更有效率。

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间： 2022-04-24 17:58:43

- 1.4 文件系统

1.4 文件系统

Xv6文件系统提供数据文件（包含未解释的字节数组）和目录（包含对数据文件和其他目录的命名引用）。这些目录形成一个树，从一个叫做根的特殊目录开始。

像 `/a/b/c` 这样的路径是指在根目录 `/` 下名为 `a` 的目录中名为 `b` 的目录中名为 `c` 的文件或目录。不以 `/` 开始的路径相对于调用进程的当前工作目录进行计算，当前工作目录可以通过 `chdir` 系统调用进行更改。下面两个代码片段打开相同的文件(假设所有相关的目录都存在)

```
chdir("/a");
chdir("b");
open("c", O_RDONLY);
open("/a/b/c", O_RDONLY);
```

上面代码将进程的当前目录更改为 `/a/b`；下面代码既不引用也不更改进程的当前目录

还有创建新文件和目录的系统调用：

- `mkdir` 创建一个新目录
- `open` 中若使用 `O_CREATE` 标志将会创建一个新的数据文件
- `mknod` 创建一个新的设备文件

这个例子说明了这三点：

```
mkdir("/dir");
fd = open("/dir/file", O_CREATE | O_WRONLY);
close(fd);
mknod("/console", 1, 1);
```

`mknod` 创建一个引用设备的特殊文件。与设备文件相关联的是主设备号和次设备号(`mknod`的两个参数)，它们唯一地标识了一个内核设备。当进程稍后打开设备文件时，内核将使用内核设备实现 `read` 和 `write` 系统调用，而不是使用文件系统。

一个文件的名字和文件本身是不同的；同一个底层文件（叫做`inode`，索引结点）可以有多个名字（叫做`link`，链接）。每个链接都由目录中的一个条目组成；该条目包含一个文件名和一个`inode`引用。`inode`保存有关文件的元数据（用于解释或帮助理解信息的数据），包括其类型(文件/目录/设备)、长度、文件内容在磁盘上的位置以及指向文件的链接数。

`fstat` 系统调用从文件描述符所引用的`inode`中检索信息。它填充一个 `stat` 类型的结构体，`struct stat` 在 `stat.h(kernel/stat.h)` 中定义为

```
#define T_DIR 1    // Directory
#define T_FILE 2    // File
#define T_DEVICE 3 // Device
struct stat {
    int dev;      // 文件系统的磁盘设备
    uint ino;     // Inode编号
    short type;   // 文件类型
    short nlink;  // 指向文件的链接数
    uint64 size;  // 文件字节数
};
```

`link` 系统调用创建另一个文件名，该文件名指向与现有文件相同的`inode`。下面的代码片段创建了一个名字既为`a`又为`b`的新文件

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

从`a`读取或写入与从`b`读取或写入是相同的操作。每个`inode`由唯一的`inode`编号标识。在上面的代码序列之后，可以通过检查 `fstat` 的结果来确定`a`和`b`引用相同的底层内容：两者都将返回相同的`inode`号(`ino`)，并且 `nlink` 计数将被设置为2。

`unlink` 系统调用从文件系统中删除一个名称。只有当文件的链接数为零且没有文件描述符引用时，文件的`inode`和包含其内容的磁盘空间才会被释放，因此添加

```
unlink("a");
```

最后一行代码序列中会使`inode`和文件内容可以作为`b`访问。此外

```
fd = open("/tmp/xyz", O_CREATE | O_RDWR);
unlink("/tmp/xyz");
```

是创建没有名称的临时`inode`的惯用方法，该临时`inode`将在进程关闭`fd`或退出时被清理。

`Unix`以用户级程序的形式提供了可从`shell`调用的文件实用程序，例如 `mkdir`、`ln` 和 `rm`。这种设计允许任何人通过添加新的用户级程序来扩展命令行接口。事后看来，这个计划似乎是显而易见的，但是在`Unix`时代设计的其他系统经常将这样的命令构建到`shell`中(并将`shell`构建到内核中)

一个例外是 `cd`，它是内置在`shell`(`user/sh.c:160`)。`cd` 必须更改`shell`本身的当前工作目录。如果 `cd` 作为常规命令运行，那么`shell`将分出一个子进程，子进程将运行 `cd`，`cd` 将更改子进程的工作目录。父目录(即`shell`的)的工作目录不会改变。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-18 12:04:10

- 1.5 真实世界

1.5 真实世界

Unix将“标准”文件描述符、管道和方便的shell语法结合起来进行操作，这是编写通用可重用程序方面的一大进步。这个想法引发了一种“软件工具”的文化，这种文化对Unix的强大和流行做出了卓越贡献，shell是第一个所谓的“脚本语言”。Unix系统调用接口今天仍然存在于BSD、Linux和MacOSx等系统中。

Unix系统调用接口已经通过便携式操作系统接口(POSIX)标准进行了标准化。Xv6与POSIX不兼容:它缺少许多系统调用(包括lseek等基本系统调用)，并且它提供的许多系统调用与标准不同。我们xv6的主要目标是简单明了，同时提供一个简单的类unix系统调用接口。为了运行基本的Unix程序，有些人扩展了xv6，增加了一些系统调用和一个简单的c库。然而，现代内核比xv6提供了更多的系统调用和更多种类的内核服务。例如，它们支持网络工作、窗口系统、用户级线程、许多设备的驱动程序等等。现代内核不断快速发展，提供了许多超越POSIX的特性。

Unix通过一组文件名和文件描述符接口统一访问多种类型的资源(文件、目录和设备)。这个想法可以扩展到更多种类的资源;一个很好的例子是Plan9，它将“资源是文件”的概念应用到网络、图形等等。然而，大多数unix衍生的操作系统并没有遵循这条路。

文件系统和文件描述符是强大的抽象。即便如此，还有其他的操作系统接口模型。Multics，Unix的前身，以一种看起来像内存的方式抽象了文件存储，产生了一种非常不同的接口风格。Multics设计的复杂性直接影响了Unix的设计者，他们试图使设计更简单。

Xv6没有提供一个用户概念或者保护一个用户不受另一个用户的伤害;用Unix的术语来说，所有的Xv6进程都作为root运行。

本书研究了xv6如何实现其类Unix接口，但这些思想和概念不仅仅适用于Unix。任何操作系统都必须在底层硬件上复用进程，彼此隔离进程，并提供受控制的进程间通讯机制。在学习了xv6之后，你应该去看看更复杂的操作系统，以及这些系统中与xv6相同的底层基本概念。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-18 12:05:11

- [1.6 练习](#)

1.6 练习

编写一个使用**UNIX**系统调用的程序，通过一对管道在两个进程之间“ping-pong”一个字节（也就是像打乒乓球一样来回传递），每个方向一个管道。以每秒的交换次数为单位，测量程序的性能。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-18 12:05:39

- 第二章 操作系统架构

第二章 操作系统架构

操作系统的一个关键要求是同时支持多个活动。例如，使用第1章中描述的系统调用接口，一个进程可以用 `fork` 启动新进程。操作系统必须在这些进程之间分时使用计算机资源。例如，即使进程比硬件处理器多，操作系统也必须确保所有进程都有机会执行。操作系统还必须安排进程之间的隔离。也就是说，如果一个进程有错误和故障，它不应该影响不依赖于有错误的进程的进程。然而，完全隔离又太过头了，进程之间应当可以进行刻意为之的交互；管道就是一个例子。因此，操作系统必须满足三个要求：多路复用、隔离和交互。

本章概述了如何组织操作系统来实现这三个要求。事实证明，有很多方法可以做到这一点，但是本文侧重于以宏内核为中心的主流设计，许多Unix操作系统都使用这种内核。本章还概述了xv6进程（它是xv6中的隔离单元）以及xv6启动时第一个进程的创建。

Xv6运行在多核RISC-V微处理器上，它的许多低级功能（例如，它的进程实现）是特定于RISC-V的。RISC-V是一个64位的中央处理器，xv6是用基于“LP64”的C语言编写的，这意味着C语言中的 `long` (L) 和指针 (P) 变量都是64位的，但 `int` 是32位的。这本书假设读者已经在一些架构上做了一些机器级编程，并将在出现时介绍RISC-V特定的想法。RISC-V的一个有用的参考文献是《The RISC-V Reader: An Open Architecture Atlas》。用户级ISA和特权指令架构均是官方规范。

完整计算机中的CPU被支撑硬件包围，其中大部分是以I/O接口的形式。Xv6是以qemu的“-machine virt”选项模拟的支撑硬件编写的。这包括RAM、包含引导代码的ROM、一个到用户键盘/屏幕的串行连接，以及一个用于存储的磁盘。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-18 22:36:12

- 2.1 抽象系统资源

2.1 抽象系统资源

当谈及操作系统时，人们可能会问的第一个问题是为什么需要它？也就是说，我们可以将图1.2中的系统调用实现为一个库，应用程序可以与之链接。在此方案中，每个应用程序甚至可以根据自己的需求定制自己的库。应用程序可以直接与硬件资源交互，并以应用程序的最佳方式使用这些资源（例如，实现高性能或可预测的性能）。一些嵌入式设备或实时系统的操作系统就是这样组织的。

这种库函数方法的缺点是，如果有多个应用程序在运行，这些应用程序必须表现良好。例如，每个应用程序必须定期放弃中央处理器，以便其他应用程序能够运行。如果所有应用程序都相互信任并且没有错误，这种协同操作的分时方案可能是可以的。然而更典型的情况是，应用程序互不信任且存在bug，所以人们通常希望提供比合作方案更强的隔离。

为了实现强隔离，最好禁止应用程序直接访问敏感的硬件资源，而是将资源抽象为服务。例如，**Unix**应用程序只通过文件系统的 `open`、`read`、`write` 和 `close` 系统调用与存储交互，而不是直接读写磁盘。这为应用程序提供了方便实用的路径名，并允许操作系统（作为接口的实现者）管理磁盘。即使隔离不是一个问题，有意交互（或者只是希望互不干扰）的程序可能会发现文件系统比直接使用磁盘更方便。

同样，**Unix**在进程之间透明地切换硬件处理器，根据需要保存和恢复寄存器状态，这样应用程序就不必意识到分时共享的存在。这种透明性允许操作系统共享处理器，即使有些应用程序处于无限循环中。

另一个例子是，**Unix**进程使用 `exec` 来构建它们的内存映像，而不是直接与物理内存交互。这允许操作系统决定将一个进程放在内存中的哪里；如果内存很紧张，操作系统甚至可以将一个进程的一些数据存储在磁盘上。`exec` 还为用户提供了存储可执行程序映像的文件系统的便利。

Unix进程之间的许多交互形式都是通过文件描述符实现的。文件描述符不仅抽象了许多细节（例如，管道或文件中的数据存储在哪里），而且还可以简化交互的方式进行了定义。例如，如果流水线中的一个应用程序失败了，内核会为流水线中的下一个进程生成文件结束信号（EOF）。

图1.2中的系统调用接口是精心设计的，既为程序员提供了便利，又提供了强隔离的可能性。**Unix**接口不是抽象资源的唯一方法，但它已经被证明是一个非常好的方法

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 09:44:42

- 2.2 用户态，核心态，以及系统调用

2.2 用户态，核心态，以及系统调用

[!NOTE] 用户态=用户模式=目态

核心态=管理模式=管态

强隔离需要应用程序和操作系统之间的硬边界，如果应用程序出错，我们不希望操作系统失败或其他应用程序失败，相反，操作系统应该能够清理失败的应用程序，并继续运行其他应用程序，要实现强隔离，操作系统必须保证应用程序不能修改（甚至读取）操作系统的数据结构和指令，以及应用程序不能访问其他进程的内存。

CPU为强隔离提供硬件支持。例如，RISC-V有三种CPU可以执行指令的模式：机器模式(**Machine Mode**)、用户模式(**User Mode**)和管理模式(**Supervisor Mode**)。在机器模式下执行的指令具有完全特权；CPU在机器模式下启动。机器模式主要用于配置计算机。Xv6在机器模式下执行很少的几行代码，然后更改为管理模式。

在管理模式下，CPU被允许执行特权指令：例如，启用和禁用中断、读取和写入保存页表地址的寄存器等。如果用户模式下的应用程序试图执行特权指令，那么CPU不会执行该指令，而是切换到管理模式，以便管理模式代码可以终止应用程序，因为它做了它不应该做的事情。第1章中的图1.1说明了这种组织。应用程序只能执行用户模式的指令（例如，数字相加等），并被称为在用户空间中运行，而此时处于管理模式下的软件可以执行特权指令，并被称为在内核空间中运行。在内核空间（或管理模式）中运行的软件被称为内核。

想要调用内核函数的应用程序（例如xv6中的 `read` 系统调用）必须过渡到内核。CPU提供一个特殊的指令，将CPU从用户模式切换到管理模式，并在内核指定的入口点进入内核（RISC-V为此提供 `ecall` 指令）。一旦CPU切换到管理模式，内核就可以验证系统调用的参数，决定是否允许应用程序执行请求的操作，然后拒绝它或执行它。由内核控制转换到管理模式的入口点是很重要的；如果应用程序可以决定内核入口点，那么恶意应用程序可以在跳过参数验证的地方进入内核。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间：2021-08-19 10:28:57

- 2.3 内核组织

2.3 内核组织

一个关键的设计问题是操作系统的哪些部分应该以管理模式运行。一种可能是整个操作系统都驻留在内核中，这样所有系统调用的实现都以管理模式运行。这种组织被称为宏内核（**monolithic kernel**）。

在这种组织中，整个操作系统以完全的硬件特权运行。这个组织很方便，因为操作系统设计者不必考虑操作系统的哪一部分不需要完全的硬件特权。此外，操作系统的不同部分更容易合作。例如，一个操作系统可能有一个可以由文件系统和虚拟内存系统共享的数据缓存区。

宏组织的一个缺点是操作系统不同部分之间的接口通常很复杂（正如我们将在本文的其余部分中看到的），因此操作系统开发人员很容易犯错误。在宏内核中，一个错误就可能是致命的，因为管理模式中的错误经常会导致内核失败。如果内核失败，计算机停止工作，因此所有应用程序也会失败。计算机必须重启才能再次使用。

为了降低内核出错的风险，操作系统设计者可以最大限度地减少在管理模式下运行的操作系统代码量，并在用户模式下执行大部分操作系统。这种内核组织被称为微内核（**microkernel**）。

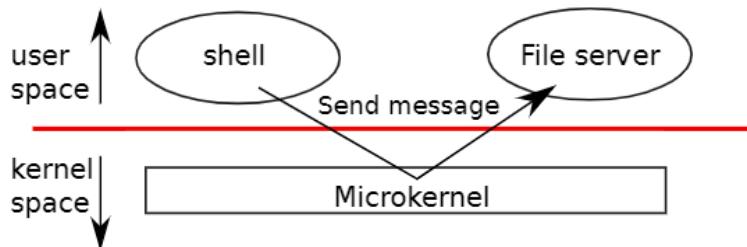


Figure 2.1: A microkernel with a file-system server

图2.1说明了这种微内核设计。在图中，文件系统作为用户级进程运行。作为进程运行的操作系统服务被称为服务器。为了允许应用程序与文件服务器交互，内核提供了允许从一个用户态进程向另一个用户态进程发送消息的进程间通信机制。例如，如果像**shell**这样的应用程序想要读取或写入文件，它会向文件服务器发送消息并等待响应。

[!TIP|label:TIPS] 由于客户/服务器（**Client/Server**）模式，具有非常多的优点，故在单机**微内核**操作系统中几乎无一例外地都采用客户/服务器模式，将操作系统中最基本的部分放入内核中，而把操作系统的绝大部分功能都放在微内核外面的一组服务器(**进程**)中实现。

在微内核中，内核接口由一些用于启动应用程序、发送消息、访问设备硬件等的低级功能组成。这种组织允许内核相对简单，因为大多数操作系统驻留在用户级服务器中。

像大多数Unix操作系统一样，Xv6是作为一个宏内核实现的。因此，xv6内核接口对应于操作系统接口，内核实现了完整的操作系统。由于xv6不提供太多服务，它的内核可以比一些微内核还小，但从概念上说xv6属于宏内核

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 09:55:13

- 2.4 代码（XV6架构篇）

2.4 代码（XV6架构篇）

XV6的源代码位于`kernel/`子目录中，源代码按照模块化的概念划分为多个文件，图2.2列出了这些文件，模块间的接口都被定义在了`def.h` (`kernel/defs.h`)。

文件	描述
<code>bio.c</code>	文件系统的磁盘块缓存
<code>console.c</code>	连接到用户的键盘和屏幕
<code>entry.S</code>	首次启动指令
<code>exec.c</code>	<code>exec()</code> 系统调用
<code>file.c</code>	文件描述符支持
<code>fs.c</code>	文件系统
<code>kalloc.c</code>	物理页面分配器
<code>kernelvec.S</code>	处理来自内核的陷入指令以及计时器中断
<code>log.c</code>	文件系统日志记录以及崩溃修复
<code>main.c</code>	在启动过程中控制其他模块初始化
<code>pipe.c</code>	管道
<code>plic.c</code>	RISC-V中断控制器
<code>printf.c</code>	格式化输出到控制台
<code>proc.c</code>	进程和调度
<code>sleeplock.c</code>	Locks that yield the CPU
<code>spinlock.c</code>	Locks that don't yield the CPU.
<code>start.c</code>	早期机器模式启动代码
<code>string.c</code>	字符串和字节数组库
<code>swtch.c</code>	线程切换
<code>syscall.c</code>	Dispatch system calls to handling function.
<code>sysfile.c</code>	文件相关的系统调用
<code>sysproc.c</code>	进程相关的系统调用
<code>trampoline.S</code>	用于在用户和内核之间切换的汇编代码
<code>trap.c</code>	对陷入指令和中断进行处理并返回的C代码
<code>uart.c</code>	串口控制台设备驱动程序
<code>virtio_disk.c</code>	磁盘设备驱动程序
<code>vm.c</code>	管理页表和地址空间

图2.2: XV6内核源文件

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 09:50:27

- 2.5 进程概述

2.5 进程概述

Xv6（和其他Unix操作系统一样）中的隔离单位是一个进程。进程抽象防止一个进程破坏或监视另一个进程的内存、CPU、文件描述符等。它还防止一个进程破坏内核本身，这样一个进程就不能破坏内核的隔离机制。内核必须小心地实现进程抽象，因为一个有缺陷或恶意的应用程序可能会欺骗内核或硬件做坏事（例如，绕过隔离）。内核用来实现进程的机制包括用户/管理模式标志、地址空间和线程的时间切片。

为了帮助加强隔离，进程抽象给程序提供了一种错觉，即它有自己的专用机器。进程为程序提供了一个看起来像是私有内存系统或地址空间的东西，其他进程不能读取或写入。进程还为程序提供了看起来像是自己的CPU来执行程序的指令。

Xv6使用页表（由硬件实现）为每个进程提供自己的地址空间。RISC-V页表将虚拟地址（RISC-V指令操纵的地址）转换（或“映射”）为物理地址（CPU芯片发送到主存储器的地址）。

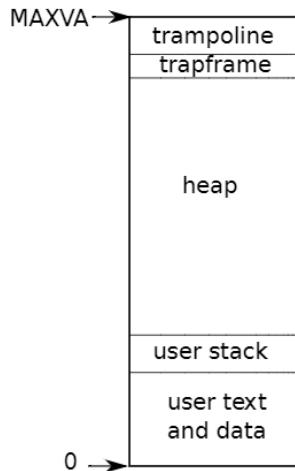


Figure 2.3: Layout of a process's virtual address space

Xv6为每个进程维护一个单独的页表，定义了该进程的地址空间。如图2.3所示，以虚拟内存地址0开始的进程的用户内存地址空间。首先是指令，然后是全局变量，然后是栈区，最后是一个堆区域（用于`malloc`）以供进程根据需要进行扩展。有许多因素限制了进程地址空间的最大范围：RISC-V上的指针有64位宽；硬件在页表中查找虚拟地址时只使用低39位；xv6只使用这39位中的38位。因此，最大地址是 $2^{38}-1=0x3fffffff$ ，即`MAXVA`（定义在`kernel/riscv.h:348`）。在地址空间的顶部，xv6为`trampoline`（用于在用户和内核之间切换）和映射进程切换到内核的`trapframe`分别保留了一个页面，正如我们将在第4章中解释的那样。

xv6内核为每个进程维护许多状态片段，并将它们聚集到一个`proc` (`kernel/proc.h:86`)结构体中。一个进程最重要的内核状态片段是它的页表、内核栈区和运行状态。我们将使用符号`p->xxx`来引用`proc`结构体的元素；例如，`p->pagetable`是一个指向该进程页表的指针。

每个进程都有一个执行线程（或简称线程）来执行进程的指令。一个线程可以挂起并且稍后再恢复。为了透明地在进程之间切换，内核挂起当前运行的线程，并恢复另一个进程的线程。线程的大部分状态（本地变量、函数调用返回地址）存储在线程的栈区上。每个进程有两个栈区：一个用户栈区和一个内核栈区（`p->kstack`）。当进程执行用户指令时，只有它的用户栈在使用，它的内核栈是空的。当进程进入内核（由于系统调用或中断）时，内核代码在进程的内核堆栈上执行；当一个进程在内核中时，它的用户堆栈仍然包含保存的数据，只是不处于活动状态。进程的线程在主动使用它的用户栈和内核栈之间交替。内核栈是独立的（并且不受用户代码的保护），因此即使一个进程破坏了它的用户栈，内核依然可以正常运行。

一个进程可以通过执行RISC-V的 `ecall` 指令进行系统调用，该指令提升硬件特权级别，并将程序计数器（PC）更改为内核定义的入口点，入口点的代码切换到内核栈，执行实现系统调用的内核指令，当系统调用完成时，内核切换回用户栈，并通过调用 `sret` 指令返回用户空间，该指令降低了硬件特权级别，并在系统调用指令刚结束时恢复执行用户指令。进程的线程可以在内核中“阻塞”等待I/O，并在I/O完成后恢复到中断的位置。

`p->state` 表明进程是已分配、就绪态、运行态、等待I/O中（阻塞态）还是退出。

`p->pagetable` 以RISC-V硬件所期望的格式保存进程的页表。当在用户空间执行进程时，Xv6让分页硬件使用进程的 `p->pagetable`。一个进程的页表也可以作为已分配给该进程用于存储进程内存的物理页面地址的记录。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 09:51:20

- 2.6 代码（启动XV6和第一个进程）

2.6 代码（启动XV6和第一个进程）

为了使xv6更加具体，我们将概述内核如何启动和运行第一个进程。接下来的章节将更详细地描述本概述中显示的机制。

当RISC-V计算机上电时，它会初始化自己并运行一个存储在只读内存中的引导加载程序。引导加载程序将xv6内核加载到内存中。然后，在机器模式下，中央处理器从 `_entry (kernel/entry.S:6)` 开始运行xv6。Xv6启动时页式硬件（`paging hardware`）处于禁用模式：也就是说虚拟地址将直接映射到物理地址。

加载程序将xv6内核加载到物理地址为 `0x80000000` 的内存中。它将内核放在 `0x80000000` 而不是 `0x0` 的原因是地址范围 `0x0:0x80000000` 包含I/O设备。

`_entry` 的指令设置了一个栈区，这样xv6就可以运行C代码。`Xv6`在 `start.c (kernel/start.c:11)` 文件中为初始栈 `stack0` 声明了空间。由于RISC-V上的栈是向下扩展的，所以 `_entry` 的代码将栈顶地址 `stack0+4096` 加载到栈顶指针寄存器 `sp` 中。现在内核有了栈区，`_entry` 便调用C代码 `start (kernel/start.c:21)`。

函数 `start` 执行一些仅在机器模式下允许的配置，然后切换到管理模式。RISC-V提供指令 `mret` 以进入管理模式，该指令最常用于将管理模式切换到机器模式的调用中返回。而 `start` 并非从这样的调用返回，而是执行以下操作：它在寄存器 `mstatus` 中将先前的运行模式改为管理模式，它通过将 `main` 函数的地址写入寄存器 `mepc` 将返回地址设为 `main`，它通过向页表寄存器 `satp` 写入0来在管理模式下禁用虚拟地址转换，并将所有的中断和异常委托给管理模式。

在进入管理模式之前，`start` 还要执行另一项任务：对时钟芯片进行编程以产生计时器中断。清理完这些“家务”后，`start` 通过调用 `mret` “返回”到管理模式。这将导致程序计数器（PC）的值更改为 `main (kernel/main.c:11)` 函数地址。

[!TIP][label:TIPS]注：`mret` 执行返回，返回到先前状态，由于 `start` 函数将前模式改为了管理模式且返回地址改为了 `main`，因此 `mret` 将返回到 `main` 函数，并以管理模式运行

在 `main (kernel/main.c:11)` 初始化几个设备和子系统后，便通过调用 `userinit (kernel/proc.c:212)` 创建第一个进程，第一个进程执行一个用RISC-V程序集写的小型程序：`initcode.S (user/initcode.S:1)`，它通过调用 `exec` 系统调用重新进入内核。正如我们在第1章中看到的，`exec` 用一个新程序（本例中为 `/init`）替换当前进程的内存和寄存器。一旦内核完成 `exec`，它就返回 `/init` 进程中的用户空间。如果需要，`init (user/init.c:15)` 将创建一个新的控制台设备文件，然后以文件描述符0、1和2打开它。然后它在控制台上启动一个`shell`。系统就这样启动了。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间：2021-08-19 09:53:20

- 2.7 真实世界

2.7 真实世界

在现实中，人们可以同时看到宏内核和微内核。许多Unix都采用宏内核。例如，尽管Linux的一些操作系统功能作为用户级服务器运行（例如窗口系统），但它是宏内核架构。而如L4、Minix和QNX的内核都被组织成一个带有多个服务器的微内核，微内核在嵌入式设备中得到了广泛的应用。

大多数操作系统都采用了进程的概念，并且大多数操作系统的进程看起来与xv6相似。然而，现代操作系统支持在一个进程中创建多个线程，使得一个进程能够利用多个处理器。在一个进程中支持多个线程涉及许多XV6缺乏的机制，包括潜在的接口更改（例如，Linux下 fork 的变体 clone），以控制进程线程共享哪些内容。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 09:55:46

- 2.8 练习

2.8 练习

你可以使用**gdb**来观察最开始的“内核空间到用户空间”的转换。

1. 请运行 `make qemu-gdb`（如果想以单线程方式，则输入 `make CPUS=1 qemu-gdb`）。
2. 打开另一个窗口，并在相同的目录下运行 `gdb`（注：应当使用`riscv64-linux-gnu-gdb`）。
3. 键入`gdb`命令 `break*0x3fffff10e`，这将在内核中的 `sret` 指令处设置一个断点，该指令从内核空间跳入用户空间。
4. 键入`gdb`命令 `continue`。`gdb`应当会停留在即将执行 `sret` 的断点处。
5. 键入 `stepi`。`gdb`现在应当会指示目前在地址为 `0x0` 处执行，该地址就是以 **initcode.S** 开始的用户空间的起始地址

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间： 2021-08-19 09:56:24

- 第三章 页表

第三章 页表

页表是操作系统为每个进程提供自己的私有地址空间和内存的机制。页表决定了内存地址的含义，以及物理内存的哪些部分可以访问。它们允许xv6隔离不同进程的地址空间，并将它们复用到单个物理内存上。页表还提供了一层抽象（**a level of indirection**），这允许xv6执行一些特殊操作：在若干个地址空间中映射相同的内存（**a trampoline page**），并用一个未映射的页面保护内核和用户栈区。本章的其余部分解释了RISC-V硬件提供的页表以及xv6如何使用它们。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 10:43:11

- 3.1 页式硬件

3.1 页式硬件

提醒一下，RISC-V指令（用户和内核指令都包括）使用的是虚拟地址。而机器的RAM或物理内存是由物理地址索引的。RISC-V页表硬件通过将每个虚拟地址映射到物理地址来为这两种地址建立联系。

XV6基于Sv39 RISC-V运行，这意味着它只使用64位虚拟地址的低39位；而高25位不使用。在这种Sv39配置中，RISC-V页表在逻辑上是一个由 $\$2^{27}$ （134,217,728）个页表条目（Page Table Entries/PTE）组成的数组。每个PTE包含一个44位的物理页码（Physical Page Number/PPN）和一些标志。页式硬件通过使用虚拟地址39位中的前27位索引页表，以找到该虚拟地址对应的一个PTE，然后生成一个56位的物理地址，其前44位来自PTE中的PPN，其后12位来自原始虚拟地址。图3.1显示了这个过程，页表的逻辑视图是一个简单的PTE数组（参见图3.2进行更详细的了解）。页表通过逻辑到物理地址的转换给了操作系统控制权，转换的粒度是一个个对齐的物理块（一个物理块包含 $\$2^{12}=4096$ 字节），这样的块称为页面。

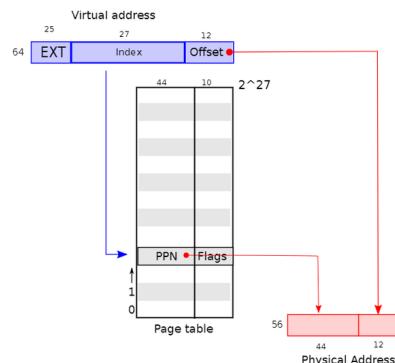


Figure 3.1: RISC-V virtual and physical addresses, with a simplified logical page table.

在Sv39 RISC-V中，虚拟地址的前25位不用于转换；将来RISC-V可能会使用那些位来定义更多级别的转换。另外物理地址也是有增长空间的：PTE格式中有空间让物理地址长度再增长10个比特位。

如图3.2所示，实际的转换分三个步骤进行。页表以三级的树型结构存储在物理内存中。该树的根是一个4096字节的页表页，其中包含512个PTE，其中包含该树下一级页表页的物理地址。这些页中的每一个都包含该树最后一级的512个PTE（也就是说每个PTE占8个字节，正如图3.2最下面所描绘的）。分页硬件使用27位中的前9位在根页表页面中选择PTE，中间9位在树的下一级页表页面中选择PTE，最后9位选择最终的PTE。

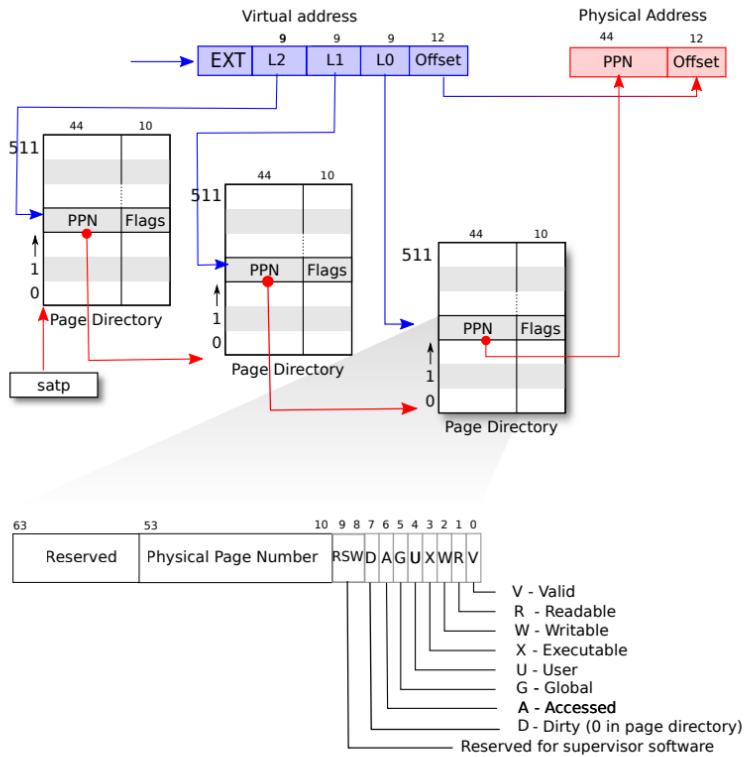


Figure 3.2: RISC-V address translation details.

如果转换地址所需的三个PTE中的任何一个不存在，页式硬件就会引发页面故障异常，并让内核来处理该异常（参见第4章）。这种三级结构允许页表在大范围虚拟地址没有映射的情况下忽略整个页表页面。

每个PTE包含标志位，这些标志位告诉分页硬件允许如何使用关联的虚拟地址。`PTE_V` 指示PTE是否存在：如果它没有被设置，对页面的引用会导致异常（即不允许）。`PTE_R` 控制是否允许指令读取到页面。`PTE_W` 控制是否允许指令写入到页面。`PTE_X` 控制CPU是否可以将页面内容解释为指令并执行它们。`PTE_U` 控制用户模式下的指令是否被允许访问页面；如果没有设置 `PTE_U`，PTE只能在管理模式下使用。图3.2显示了它是如何工作的。标志和所有其他与页面硬件相关的结构在 (*kernel/riscv.h*) 中定义。

为了告诉硬件使用页表，内核必须将根页表页的物理地址写入到 `satp` 寄存器中（`satp` 的作用是存放根页表页在物理内存中的地址）。每个CPU都有自己 `satp`。一个CPU将使用自己的 `satp` 指向的页表转换后续指令生成的所有地址。每个CPU都有自己的 `satp`，这样不同的CPU就可以运行不同的进程，每个CPU都有自己的页表描述的私有地址空间。

关于术语的一些注意事项。物理内存是指DRAM中的存储单元。物理内存以一个字节为单位划为地址，称为物理地址。指令只使用虚拟地址，页式硬件将其转换为物理地址，然后将其发送到DRAM硬件来进行读写。与物理内存和虚拟地址不同，虚拟内存不是物理对象，而是指内核提供的管理物理内存和虚拟地址的抽象和机制的集合。

- 3.2 内核地址空间

3.2 内核地址空间

Xv6为每个进程维护一个页表，用以描述每个进程的用户地址空间，外加一个单独描述内核地址空间的页表。内核配置其地址空间的布局，以允许自己以可预测的虚拟地址访问物理内存和各种硬件资源。图3.3显示了这种布局如何将内核虚拟地址映射到物理地址。文件(**kernel/memlayout.h**)声明了xv6内核内存布局的常量。

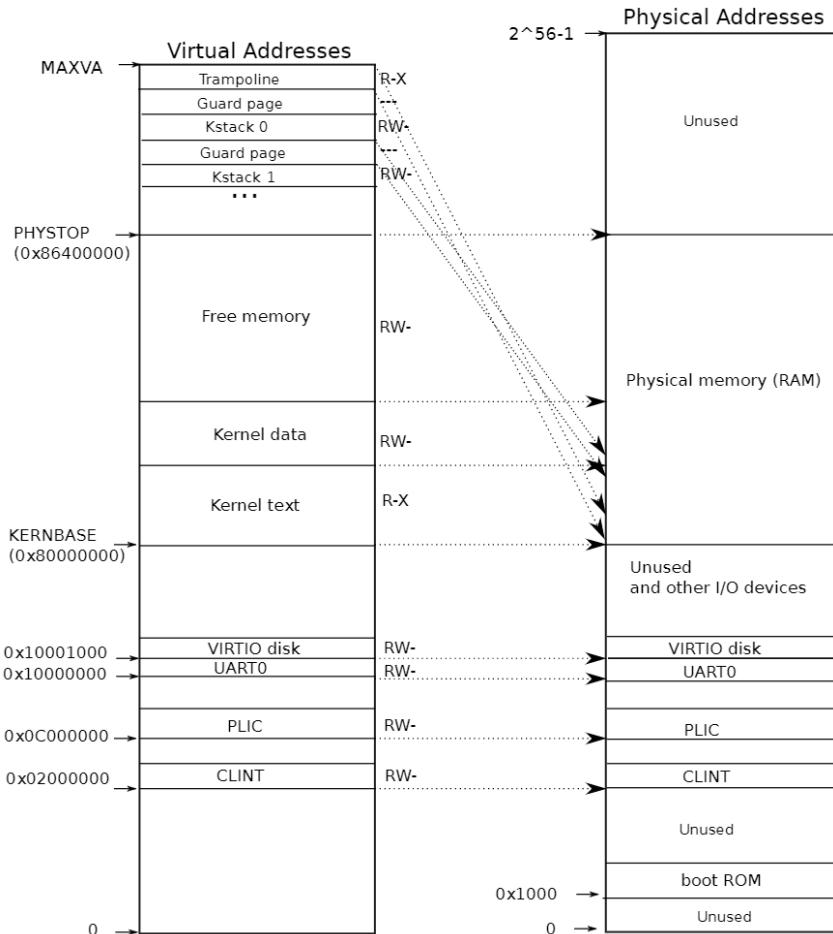


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

QEMU模拟了一台计算机，它包括从物理地址 `0x80000000` 开始并至少到 `0x86400000` 结束的RAM（物理内存），xv6称结束地址为 `PHYSTOP`。QEMU模拟还包括I/O设备，如磁盘接口。QEMU将设备接口作为内存映射控制寄存器暴露给软件，这些寄存器位于物理地址空间 `0x80000000` 以下。内核可以通过读取/写入这些特殊的物理地址与设备交互；这种读取和写入与设备硬件而不是RAM通信。第4章解释了xv6如何与设备进行交互。

内核使用“直接映射”获取内存和内存映射设备寄存器；也就是说，将资源映射到等于物理地址的虚拟地址。例如，内核本身在虚拟地址空间和物理内存中都位于 `KERNBASE` = 0x80000000`。直接映射简化了读取或写入物理内存的内核代码。例如，当 `fork` 为子进程分配用户内存时，分配器返回该内存的物理地址；`fork` 在将父进程的用户内存复制到子进程时直接将该地址用作虚拟地址。

有几个内核虚拟地址不是直接映射：

- 跳床页面(**trampoline page**)。它映射在虚拟地址空间的顶部；用户页表具有相同的映射。第4章讨论了跳床页面的作用，但我们在那里看到了一个有趣的页表用例；一个物理页面（持有跳床代码）在内核的虚拟地址空间中映射了两次：一次在虚拟地址空间的顶部，一次直接映射。
- 内核栈页面。每个进程都有自己的内核栈，它将映射到偏高一些的地址，这样xv6在它之下就可以留下一个未映射的保护页(**guard page**)。保护页的PTE是无效的（也就是说 `PTE_V` 没有设置），所以如果内核溢出内核栈就会引发一个异常，内核触发 `panic`。如果没有保护页，栈溢出将会覆盖其他内核内存，引发错误操作。恐慌崩溃（**panic crash**）是更可取的方案。（注：**Guard page**不会浪费物理内存，它只是占据了虚拟地址空间的一段靠后的地址，但并不映射到物理地址空间。）

虽然内核通过高地址内存映射使用内核栈，是它们也可以通过直接映射的地址进入内核。另一种设计可能只有直接映射，并在直接映射的地址使用栈。然而，在这种安排中，提供保护页将涉及取消映射虚拟地址，否则虚拟地址将引用物理内存，这将很难使用。

内核在权限 `PTE_R` 和 `PTE_X` 下映射跳床页面和内核文本页面。内核从这些页面读取和执行指令。内核在权限 `PTE_R` 和 `PTE_W` 下映射其他页面，这样它就可以读写那些页面中的内存。对于保护页面的映射是无效的。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 10:48:06

- 3.3 代码：创建一个地址空间

3.3 代码：创建一个地址空间

大多数用于操作地址空间和页表的xv6代码都写在 **vm.c** (*kernel/vm.c:1*) 中。其核心数据结构是 `pagetable_t`，它实际上是指向RISC-V根页表页的指针；一个 `pagetable_t` 可以是内核页表，也可以是一个进程页表。最核心的函数是 `walk` 和 `mappages`，前者为虚拟地址找到PTE，后者为新映射装载PTE。名称以 `kvm` 开头的函数操作内核页表；以 `uvm` 开头的函数操作用户页表；其他函数用于二者。`copyout` 和 `copyin` 复制数据到用户虚拟地址或从用户虚拟地址复制数据，这些虚拟地址作为系统调用参数提供；由于它们需要显式地翻译这些地址，以便找到相应的物理内存，故将它们写在 **vm.c** 中。

在按序启动的前期，`main` 调用 `kvminit` (*kernel/vm.c:22*) 来创建内核的页表。这个调用发生在xv6使能RISC-V分页之前，所以地址直接引用物理内存。`kvminit` 首先分配一个物理内存页面来保存根页表页。然后它调用 `kvmmap` 来装载内核需要的转换。转换包括内核的指令和数据、物理内存的上限是 `PHYSTOP`，并包括实际上是设备的内存。

`kvmmap` (*kernel/vm.c:118*) 调用 `mappages` (*kernel/vm.c:149*)，`mappages` 将范围虚拟地址到同等范围物理地址的映射装载到一个页表中。它以页面大小为间隔，为范围内的每个虚拟地址单独执行此操作。对于要映射的每个虚拟地址，`mappages` 调用 `walk` 来查找该地址的PTE地址。然后，它初始化PTE以保存相关的物理页号、所需权限（`PTE_W`、`PTE_X` 和/或 `PTE_R`）以及用于标记PTE有效的 `PTE_V` (*kernel/vm.c:161*)。

在查找PTE中的虚拟地址（参见图3.2）时，`walk` (*kernel/vm.c:72*) 模仿RISC-V分页硬件。`walk` 一次从3级页表中获取9个比特位。它使用上一级的9位虚拟地址来查找下一级页表或最终页面的PTE (*kernel/vm.c:78*)。如果PTE无效，则所需的页面还没有分配；如果设置了 `alloc` 参数，`walk` 就会分配一个新的页表页面，并将其物理地址放在PTE中。它返回树中最低一级的PTE地址(*kernel/vm.c:88*)。

上面的代码依赖于直接映射到内核虚拟地址空间中的物理内存。例如，当 `walk` 降低页表的级别时，它从PTE (*kernel/vm.c:80*) 中提取下一级页表的（物理）地址，然后使用该地址作为虚拟地址来获取下一级的PTE (*kernel/vm.c:78*)。

`main` 调用 `kvminit hart` (*kernel/vm.c:53*) 来安装内核页表。它将根页表页的物理地址写入寄存器 `satp`。之后，CPU将使用内核页表转换地址。由于内核使用标识映射，下一条指令的当前虚拟地址将映射到正确的物理内存地址。

`main` 中调用的 `procinit` (*kernel/proc.c:26*) 为每个进程分配一个内核栈。它将每个栈映射到 `KSTACK` 生成的虚拟地址，这为无效的栈保护页面留下了空间。`kvmmap` 将映射的PTE添加到内核页表中，对 `kvminit hart` 的调用将内核页表重新加载到 `satp` 中，以便硬件知道新的PTE。

每个RISC-V CPU都将页表条目缓存在转译后备缓冲器（快表/TLB）中，当xv6更改页表时，它必须告诉CPU使相应的缓存TLB条目无效。如果没有这么做，那么在某个时候TLB可能会使用旧的缓存映射，指向一个在此期间已分配给另一个进程的物理页面，这样会导致一个进程可能能够在其他进程的内存上涂鸦。RISC-V有一

个指令 `sfence.vma`，用于刷新当前CPU的TLB。xv6在重新加载 `satp` 寄存器后，在 `kvm_inithart` 中执行 `sfence.vma`，并在返回用户空间之前在用于切换至一个用户页表的 `trampoline` 代码中执行 `sfence.vma` (***kernel/trampoline.S:79***)。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 11:44:27

- 3.4 物理内存分配

3.4 物理内存分配

内核必须在运行时为页表、用户内存、内核栈和管道缓冲区分配和释放物理内存。
`xv6`使用内核末尾到 `PHYSTOP` 之间的物理内存进行运行时分配。它一次分配和释放整个4096字节的页面。它使用链表的数据结构将空闲页面记录下来。分配时需要从链表中删除页面；释放时需要将释放的页面添加到链表中。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 10:49:09

- 3.5 代码（物理内存分配）

3.5 代码（物理内存分配）

分配器(allocator)位于 `kalloc.c`(*kernel/kalloc.c:1*)中。分配器的数据结构是可供分配的物理内存页的空闲列表。每个空闲页的列表元素是一个 `struct run` (*kernel/kalloc.c:17*)。分配器从哪里获得内存来填充该数据结构呢？它将每个空闲页的 `run` 结构存储在空闲页本身，因为在那没有存储其他东西。空闲列表受到自旋锁（spin lock）的保护(*kernel/kalloc.c:21-24*)。列表和锁被封装在一个结构体中，以明确锁在结构体中保护的字段。现在，忽略锁以及对 `acquire` 和 `release` 的调用：第6章将详细查看有关锁的细节。

[!TIP] 对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，“自旋”一词就是因此而得名。

自旋锁比较适用于锁使用者保持锁时间比较短的情况。正是由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。

`main` 函数调用 `kinit` (*kernel/kalloc.c:27*)来初始化分配器。`kinit` 初始化空闲列表以保存从内核结束到 `PHYSTOP` 之间的每一页。`xv6`应该通过解析硬件提供的配置信息来确定有多少物理内存可用。然而，`xv6`假设机器有128兆字节的RAM。`kinit` 调用 `freerange` 将内存添加到空闲列表中，在 `freerange` 中每页都会调用 `kfree`。PTE 只能引用在4096字节边界上对齐的物理地址（是4096的倍数），所以 `freerange` 使用 `PGROUNDUP` 来确保它只释放对齐的物理地址。分配器开始时没有内存；这些对 `kfree` 的调用给了它一些管理空间。

分配器有时将地址视为整数，以便对其执行算术运算（例如，在 `freerange` 中遍历所有页面），有时将地址用作读写内存的指针（例如，操纵存储在每个页面中的 `run` 结构）；这种地址的双重用途是分配器代码充满C类型转换的主要原因。另一个原因是释放和分配从本质上改变了内存的类型。

函数 `kfree` (*kernel/kalloc.c:47*)首先将内存中的每一个字节设置为1。这将导致使用释放后的内存的代码（使用“悬空引用”）读取到垃圾信息而不是旧的有效内容，从而希望这样的代码更快崩溃。然后 `kfree` 将页面前置（头插法）到空闲列表中：它将 `pa` 转换为一个指向 `struct run` 的指针 `r`，在 `r->next` 中记录空闲列表的旧开始，并将空闲列表设置为等于 `r`。

`kalloc` 删除并返回空闲列表中的第一个元素。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 10:50:40

- 3.6 进程地址空间

3.6 进程地址空间

每个进程都有一个单独的页表，当xv6在进程之间切换时，也会更改页表。如图2.3所示，一个进程的用户内存从虚拟地址零开始，可以增长到MAXVA (*kernel/riscv.h:348*)，原则上允许一个进程内存寻址空间为256G。

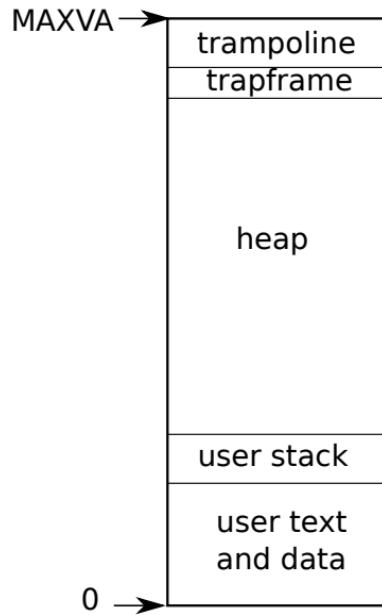


Figure 2.3: Layout of a process's virtual address space

当进程向xv6请求更多的用户内存时，xv6首先使用 `kalloc` 来分配物理页面。然后，它将PTE添加到进程的页表中，指向新的物理页面。Xv6在这些PTE中设置 `PTE_W`、`PTE_X`、`PTE_R`、`PTE_U` 和 `PTE_V` 标志。大多数进程不使用整个用户地址空间；xv6在未使用的PTE中留空 `PTE_V`。

我们在这里看到了一些使用页表的很好的例子。首先，不同进程的页表将用户地址转换为物理内存的不同页面，这样每个进程都拥有私有内存。第二，每个进程看到自己的内存空间都是以0地址起始的连续虚拟地址，而进程的物理内存可以是非连续的。第三，内核在用户地址空间的顶部映射一个带有蹦床（trampoline）代码的页面，这样在所有地址空间都可以看到一个单独的物理内存页面。

图3.4更详细地显示了xv6中执行态进程的用户内存布局。栈是单独一个页面，显示的是由 `exec` 创建后的初始内容。包含命令行参数的字符串以及指向它们的指针数组位于栈的最顶部。再往下是允许程序在 `main` 处开始启动的值（即 `main` 的地址、`argc`、`argv`），这些值产生的效果就像刚刚调用了 `main(argc, argv)` 一样。

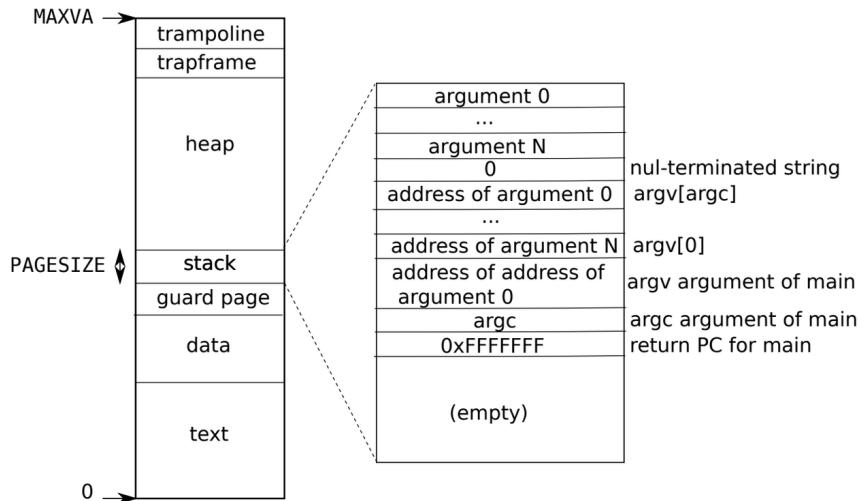


Figure 3.4: A process's user address space, with its initial stack.

为了检测用户栈是否溢出了所分配栈内存，`xv6`在栈正下方放置了一个无效的保护页（guard page）。如果用户栈溢出并且进程试图使用栈下方的地址，那么由于映射无效（`PTE_V` 为0）硬件将生成一个页面故障异常。当用户栈溢出时，实际的操作系统可能会自动为其分配更多内存。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 10:52:21

- 3.7 代码: `sbrk`

3.7 代码: `sbrk`

`sbrk` 是一个用于进程减少或增长其内存的系统调用。这个系统调用由函数 `growproc` 实现(**kernel/proc.c:239**)。`growproc` 根据 `n` 是正的还是负的调用 `uvmalloc` 或 `uvmdealloc`。`uvmalloc` (**kernel/vm.c:229**)用 `kalloc` 分配物理内存，并用 `mappages` 将PTE添加到用户页表中。`uvmdealloc` 调用 `uvmunmap` (**kernel/vm.c:174**)，`uvmunmap` 使用 `walk` 来查找对应的PTE，并使用 `kfree` 来释放PTE引用的物理内存。

XV6使用进程的页表，不仅是告诉硬件如何映射用户虚拟地址，也是明晰哪一个物理页面已经被分配给该进程的唯一记录。这就是为什么释放用户内存（在 `uvmunmap` 中）需要检查用户页表的原因。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 10:52:54

- 3.8 代码: exec

3.8 代码: exec

`exec` 是创建地址空间的用户部分的系统调用。它使用一个存储在文件系统中的文件初始化地址空间的用户部分。`exec` (*kernel/exec.c:13*) 使用 `namei` (*kernel/exec.c:26*) 打开指定的二进制 `path`，这在第8章中有解释。然后，它读取 ELF头。Xv6应用程序以广泛使用的ELF格式描述，定义于(*kernel/elf.h*)。ELF二进制文件由ELF头、`struct elfhdr` (*kernel/elf.h:6*)，后面一系列的程序节头 (`section headers`)、`struct proghdr` (*kernel/elf.h:25*)组成。每个 `proghdr` 描述程序中必须加载到内存中的一节 (`section`)；xv6程序只有一个程序节头，但是其他系统对于指令和数据部分可能各有单独的节。

[!NOTE] ELF文件格式: 在计算机科学中，是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储格式文件。ELF是UNIX系统实验室（USL）作为应用程序二进制接口（Application Binary Interface, ABI）而开发和发布的，也是Linux的主要可执行文件格式。ELF文件由4部分组成，分别是ELF头（ELF header）、程序头表（Program header table）、节（Section）和节头表（Section header table）。实际上，一个文件中不一定包含全部内容，而且它们的位置也未必如同所示这样安排，只有ELF头的位置是固定的，其余各部分的位置、大小等信息由ELF头中的各项值来决定。

第一步是快速检查文件可能包含ELF二进制的文件。ELF二进制文件以四个字节的“幻数”`0x7F`、“`E`”、“`L`”、“`F`”或 `ELF_MAGIC` 开始(*kernel/elf.h:3*)。如果ELF头有正确的幻数，`exec` 假设二进制文件格式良好。

`exec` 使用 `proc_pagetable` (*kernel/exec.c:38*)分配一个没有用户映射的新页表，使用 `uvmalloc` (*kernel/exec.c:52*)为每个ELF段分配内存，并使用 `loadseg` (*kernel/exec.c:10*)将每个段加载到内存中。`loadseg` 使用 `walkaddr` 找到分配内存的物理地址，在该地址写入ELF段的每一页，并使用 `readi` 从文件中读取。

使用 `exec` 创建的第一个用户程序 `/init` 的程序节标题如下：

```
# objdump -p _init
user/_init: file format elf64-littleriscv
Program Header:
LOAD off 0x00000000000000b0 vaddr 0x0000000000000000
          paddr 0x0000000000000000 align 2**3
          filesz 0x000000000000840 memsz 0x000000000000858 flags rwx
STACK off 0x0000000000000000 vaddr 0x0000000000000000
          paddr 0x0000000000000000 align 2**4
          filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
```

程序节头的 `filesz` 可能小于 `memsz`，这表明它们之间的间隙应该用零来填充（对于C全局变量），而不是从文件中读取。对于`/init`，`filesz` 是2112字节，`memsz` 是2136字节，因此 `uvmalloc` 分配了足够的物理内存来保存2136字节，但只从文件`/init`中读取2112字节。

现在 `exec` 分配并初始化用户栈。它只分配一个栈页面。`exec` 一次将参数中的一个字符串复制到栈顶，并在 `ustack` 中记录指向它们的指针。它在传递给 `main` 的 `argv` 列表的末尾放置一个空指针。`ustack` 中的前三个条目是伪返回程序

计数器（fake return program counter）、`argc` 和 `argv` 指针。

`exec` 在栈页面的正下方放置了一个不可访问的页面，这样试图使用超过一个页面的程序就会出错。这个不可访问的页面还允许 `exec` 处理过大的参数；在这种情况下，被 `exec` 用来将参数复制到栈的函数 `copyout` (*kernel/vm.c:355*) 将会注意到目标页面不可访问，并返回-1。

在准备新内存映像的过程中，如果 `exec` 检测到像无效程序段这样的错误，它会跳到标签 `bad`，释放新映像，并返回-1。`exec` 必须等待系统调用会成功后再释放旧映像：因为如果旧映像消失了，系统调用将无法返回-1。`exec` 中唯一的错误情况发生在映像的创建过程中。一旦映像完成，`exec` 就可以提交到新的页表 (*kernel/exec.c:113*) 并释放旧的页表 (*kernel/exec.c:117*)。

`exec` 将 ELF 文件中的字节加载到 ELF 文件指定地址的内存中。用户或进程可以将他们想要的任何地址放入 ELF 文件中。因此 `exec` 是有风险的，因为 ELF 文件中的地址可能会意外或故意的引用内核。对一个设计拙劣的内核来说，后果可能是一次崩溃，甚至是内核的隔离机制被恶意破坏（即安全漏洞）。xv6 执行许多检查来避免这些风险。例如，`if(ph.vaddr + ph.memsz < ph.vaddr)` 检查总和是否溢出 64 位整数，危险在于用户可能会构造一个 ELF 二进制文件，其中的 `ph.vaddr` 指向用户选择的地址，而 `ph.memsz` 足够大，使总和溢出到 0x1000，这看起来像是一个有效的值。在 xv6 的旧版本中，用户地址空间也包含内核（但在用户模式下不可读写），用户可以选择一个与内核内存相对应的地址，从而将 ELF 二进制文件中的数据复制到内核中。在 xv6 的 RISC-V 版本中，这是不可能的，因为内核有自己独立的页表：`loadseg` 加载到进程的页表中，而不是内核的页表中。

内核开发人员很容易省略关键的检查，而现实世界中的内核有很长一段丢失检查的历史，用户程序可以利用这些检查的缺失来获得内核特权。xv6 可能没有完成验证提供给内核的用户级数据的全部工作，恶意用户程序可以利用这些数据来绕过 xv6 的隔离。

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间：2021-08-19 10:54:29

3.9 真实世界

像大多数操作系统一样，**xv6**使用分页硬件进行内存保护和映射。大多数操作系统通过结合分页和页面故障异常使用分页，比**xv6**复杂得多，我们将在第4章讨论这一点。

内核通过使用虚拟地址和物理地址之间的直接映射，以及假设在地址 `0x8000000` 处有物理**RAM**（内核期望加载的位置），**Xv6**得到了简化。这在**QEMU**中很有效，但在实际硬件上却是个坏主意：实际硬件将**RAM**和设备置于不可预测的物理地址，因此（例如）在**xv6**期望能够存储内核的 `0x8000000` 地址处可能没有**RAM**。更严肃的内核设计利用页表将任意硬件物理内存布局转换为可预测的内核虚拟地址布局。

RISC-V支持物理地址级别的保护，但**xv6**没有使用这个特性。

在有大量内存的机器上，使用**RISC-V**对“超级页面”的支持可能很有意义。而当物理内存较小时，小页面更有用，这样可以以精细的粒度向磁盘分配和输出页面。例如，如果一个程序只使用8KB内存，给它一个4MB的物理内存超级页面是浪费。在有大量内存的机器上，较大的页面是有意义的，并且可以减少页表操作的开销。

xv6内核缺少一个类似 `malloc` 可以为小对象提供内存的分配器，这使得内核无法使用需要动态分配的复杂数据结构。

内存分配是一个长期的热门话题，基本问题是有效使用有限的内存并为将来的未知请求做好准备。今天，人们更关心速度而不是空间效率。此外，一个更复杂的内核可能会分配许多不同大小的小块，而不是（如**xv6**中）只有4096字节的块；一个真正的内核分配器需要处理小分配和大分配。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 10:55:09

- 3.10 练习

3.10 练习

1. 分析RISC-V的设备树以找到计算机拥有的物理内存量。
2. 编写一个用户程序，通过调用 `sbrk(1)` 为其地址空间增加一个字节。运行该程序并研究调用 `sbrk` 之前和调用 `sbrk` 之后该程序的页表。内核分配了多少空间？新内存的PTE包含什么？
3. 修改xv6来为内核使用超级页面。
4. 修改xv6，这样当用户程序解引用空指针时会收到一个异常。也就是说，修改xv6使得虚拟地址0不被用户程序映射。
5. 传统上，`exec` 的Unix实现包括对shell脚本的特殊处理。如果要执行的文件以文本 `#!` 开头，那么第一行将被视为解释此文件的程序来运行。例如，如果调用 `exec` 来运行 `myprog arg1`，而 `myprog` 的第一行是 `#!/interp`，那么 `exec` 将使用命令行 `/interp myprog arg1` 运行 `/interp`。在xv6中实现对该约定的支持。
6. 为内核实现地址空间随机化

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 10:55:38

- 第四章 陷阱指令和系统调用

第四章 陷阱指令和系统调用

有三种事件会导致中央处理器搁置普通指令的执行，并强制将控制权转移到处理该事件的特殊代码上。一种情况是系统调用，当用户程序执行 `ecall` 指令要求内核为其做些什么时；另一种情况是异常：（用户或内核）指令做了一些非法的事情，例如除以零或使用无效的虚拟地址；第三种情况是设备中断，一个设备，例如当磁盘硬件完成读或写请求时，向系统表明它需要被关注。

本书使用陷阱（`trap`）作为这些情况的通用术语。通常，陷阱发生时正在执行的任何代码都需要稍后恢复，并且不需要意识到发生了任何特殊的事情。也就是说，我们经常希望陷阱是透明的：这对于中断尤其重要，中断代码通常难以预料。通常的顺序是陷阱强制将控制权转移到内核；内核保存寄存器和其他状态，以便可以恢复执行；内核执行适当的处理程序代码（例如，系统调用接口或设备驱动程序）；内核恢复保存的状态并从陷阱中返回；原始代码从它停止的地方恢复。

`xv6` 内核处理所有陷阱。这对于系统调用来说是顺理成章的。由于隔离性要求用户进程不直接使用设备，而且只有内核具有设备处理所需的状态，因而对中断也是有意义的。因为 `xv6` 通过杀死违规程序来响应用户空间中的所有异常，它也对异常有意义。

`Xv6` 陷阱处理分为四个阶段：RISC-V CPU采取的硬件操作、为内核C代码执行而准备的汇编程序集“向量”、决定如何处理陷阱的C陷阱处理程序以及系统调用或设备驱动程序服务例程。虽然三种陷阱类型之间的共性表明内核可以用一个代码路径处理所有陷阱，但对于三种不同的情况：来自用户空间的陷阱、来自内核空间的陷阱和定时器中断，分别使用单独的程序集向量和C陷阱处理程序更加方便。

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间：2021-08-19 11:37:46

- 4.1 RISC-V陷入机制

4.1 RISC-V陷入机制

每个RISC-V CPU都有一组控制寄存器，内核通过向这些寄存器写入内容来告诉CPU如何处理陷阱，内核可以读取这些寄存器来明确已经发生的陷阱。RISC-V文档包含了完整的内容。*riscv.h*(*kernel/riscv.h*:1)包含在xv6中使用到的内容的定义。以下是最重要的寄存器概述：

- `stvec`：内核在这里写入其陷阱处理程序的地址；RISC-V跳转到这里处理陷阱。
- `sepc`：当发生陷阱时，RISC-V会在这里保存程序计数器 `pc`（因为 `pc` 会被 `stvec` 覆盖）。`sret`（从陷阱返回）指令会将 `sepc` 复制到 `pc`。内核可以写入 `sepc` 来控制 `sret` 的去向。
- `scause`：RISC-V在这里放置一个描述陷阱原因的数字。
- `sscratch`：内核在这里放置了一个值，这个值在陷阱处理程序一开始就会派上用场。
- `sstatus`：其中的**SIE**位控制设备中断是否启用。如果内核清空**SIE**，RISC-V将推迟设备中断，直到内核重新设置**SIE**。**SPP**位指示陷阱是来自用户模式还是管理模式，并控制 `sret` 返回的模式。

上述寄存器都用于在管理模式下处理陷阱，在用户模式下不能读取或写入。在机器模式下处理陷阱有一组等效的控制寄存器，xv6仅在计时器中断的特殊情况下使用它们。

多核芯片上的每个CPU都有自己的这些寄存器集，并且在任何给定时间都可能有多个CPU在处理陷阱。

当需要强制执行陷阱时，RISC-V硬件对所有陷阱类型（计时器中断除外）执行以下操作：

1. 如果陷阱是设备中断，并且状态**SIE**位被清空，则不执行以下任何操作。
2. 清除**SIE**以禁用中断。
3. 将 `pc` 复制到 `sepc`。
4. 将当前模式（用户或管理）保存在状态的**SPP**位中。
5. 设置 `scause` 以反映产生陷阱的原因。
6. 将模式设置为管理模式。
7. 将 `stvec` 复制到 `pc`。
8. 在新的 `pc` 上开始执行。

请注意，CPU不会切换到内核页表，不会切换到内核栈，也不会保存除 `pc` 之外的任何寄存器。内核软件必须执行这些任务。CPU在陷阱期间执行尽可能少量工作的一个原因是为软件提供灵活性；例如，一些操作系统在某些情况下不需要页表切换，这可以提高性能。

你可能想知道CPU硬件的陷阱处理顺序是否可以进一步简化。例如，假设CPU不切换程序计数器。那么陷阱可以在仍然运行用户指令的情况下切换到管理模式。但因此这些用户指令可以打破用户/内核的隔离机制，例如通过修改 `satp` 寄存器来指向允许访问所有物理内存的页表。因此，CPU使用专门的寄存器切换到内核指定的指令地址，即 `stvec`，是很重要的。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 11:38:22

- 4.2 从用户空间陷入

4.2 从用户空间陷入

如果用户程序发出系统调用（`ecall` 指令），或者做了一些非法的事情，或者设备中断，那么在用户空间中执行时就可能会产生陷阱。来自用户空间的陷阱的高级路径是 `uservec` (*kernel/trampoline.S:16*)，然后是 `usertrap` (*kernel/trap.c:37*)；返回时，先是 `usertrapret` (*kernel/trap.c:90*)，然后是 `userret` (*kernel/trampoline.S:16*)。

来自用户代码的陷阱比来自内核的陷阱更具挑战性，因为 `satp` 指向不映射内核的用户页表，栈指针可能包含无效甚至恶意的值。

由于RISC-V硬件在陷阱期间不会切换页表，所以用户页表必须包括 `uservec` (`stvec`指向的陷阱向量指令) 的映射。`uservec` 必须切换 `satp` 以指向内核页表；为了在切换后继续执行指令，`uservec` 必须在内核页表中与用户页表中映射相同的地址。

`xv6` 使用包含 `uservec` 的蹦床页面（trampoline page）来满足这些约束。`xv6` 将蹦床页面映射到内核页表和每个用户页表中相同的虚拟地址。这个虚拟地址是 `TRAMPOLINE`（如图2.3和图3.3所示）。蹦床内容在 `trampoline.S` 中设置，并且（当执行用户代码时）`stvec` 设置为 `uservec` (*kernel/trampoline.S:16*)。

当 `uservec` 启动时，所有32个寄存器都包含被中断代码所拥有的值。但是 `uservec` 需要能够修改一些寄存器，以便设置 `satp` 并生成保存寄存器的地址。RISC-V以 `sscratch` 寄存器的形式提供了帮助。`uservec` 开始时的 `csrrw` 指令交换了 `a0` 和 `sscratch` 的内容。现在用户代码的 `a0` 被保存了；`uservec` 有一个寄存器（`a0`）可以使用；`a0` 包含内核以前放在 `sscratch` 中的值。

`uservec` 的下一个任务是保存用户寄存器。在进入用户空间之前，内核先前将 `sscratch` 设置为指向一个每个进程的陷阱帧，该帧（除此之外）具有保存所有用户寄存器的空间(*kernel/proc.h:44*)。因为 `satp` 仍然指向用户页表，所以 `uservec` 需要将陷阱帧映射到用户地址空间中。每当创建一个进程时，`xv6` 就为该进程的陷阱帧分配一个页面，并安排它始终映射在用户虚拟地址 `TRAPFRAME`，该地址就在 `TRAMPOLINE` 下面。尽管使用物理地址，该进程的 `p->trapframe` 仍指向陷阱帧，这样内核就可以通过内核页表使用它。

因此在交换 `a0` 和 `sscratch` 之后，`a0` 持有指向当前进程陷阱帧的指针。`uservec` 现在保存那里的所有用户寄存器，包括从 `sscratch` 读取的用户的 `a0`。

陷阱帧包含指向当前进程内核栈的指针、当前CPU的 `hartid`、`usertrap` 的地址和内核页表的地址。`uservec` 取得这些值，将 `satp` 切换到内核页表，并调用 `usertrap`。

`usertrap` 的任务是确定陷阱的原因，处理并返回(*kernel/trap.c:37*)。如上所述，它首先改变 `stvec`，这样内核中的陷阱将由 `kernelvec` 处理。它保存了 `sepc`（保存的用户程序计数器），再次保存是因为 `usertrap` 中可能有一个进程切换，可能导致 `sepc` 被覆盖。如果陷阱来自系统调用，`syscall` 会处理它；如果是设备中断，`devintr` 会处理；否则它是一个异常，内核会杀死错误进程。系统调用路径在保存的用户程序计数器 `pc` 上加4，因为在系统调用的情况下，RISC-V会留下指

向 `ecall` 指令的程序指针（返回后需要执行 `ecall` 之后的下一条指令）。在退出的过程中，`usertrap` 检查进程是已经被杀死还是应该让出CPU（如果这个陷阱是计时器中断）。

返回用户空间的第一步是调用 `usertrapret` (*kernel/trap.c:90*)。该函数设置RISC-V控制寄存器，为将来来自用户空间的陷阱做准备。这涉及到将 `stvec` 更改为指向 `uservec`，准备 `uservec` 所依赖的陷阱帧字段，并将 `sepc` 设置为之前保存的用户程序计数器。最后，`usertrapret` 在用户和内核页表中都映射的蹦床页面上调用 `userret`；原因是 `userret` 中的汇编代码会切换页表。

`usertrapret` 对 `userret` 的调用将指针传递到 `a0` 中的进程用户页表和 `a1` 中的 `TRAPFRAME` (*kernel/trampoline.S:88*)。`userret` 将 `satp` 切换到进程的用户页表。回想一下，用户页表同时映射蹦床页面和 `TRAPFRAME`，但没有从内核映射其他内容。同样，蹦床页面映射在用户和内核页表中的同一个虚拟地址上的事实允许用户在更改 `satp` 后继续执行。`userret` 复制陷阱帧保存的用户 `a0` 到 `sscratch`，为以后与 `TRAPFRAME` 的交换做准备。从此刻开始，`userret` 可以使用的唯一数据是寄存器内容和陷阱帧的内容。下一个 `userret` 从陷阱帧中恢复保存的用户寄存器，做 `a0` 与 `sscratch` 的最后一次交换来恢复用户 `a0` 并为下一个陷阱保存 `TRAPFRAME`，并使用 `sret` 返回用户空间。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 11:38:14

- 4.3 代码：调用系统调用

4.3 代码：调用系统调用

第2章以 `initcode.S` 调用 `exec` 系统调用 (`user/initcode.S:11`) 结束。让我们看看用户调用是如何在内核中实现 `exec` 系统调用的。

用户代码将 `exec` 需要的参数放在寄存器 `a0` 和 `a1` 中，并将系统调用号放在 `a7` 中。系统调用号与 `syscalls` 数组中的条目相匹配，`syscalls` 数组是一个函数指针表（`kernel/syscall.c:108`）。`ecall` 指令陷入(trap)到内核中，执行 `uservec`、`usertrap` 和 `syscall`，和我们之前看到的一样。

`syscall`（`kernel/syscall.c:133`）从陷阱帧（`trapframe`）中保存的 `a7` 中检索系统调用号（`p->trapframe->a7`），并用它索引到 `syscalls` 中，对于第一次系统调用，`a7` 中的内容是 `SYS_exec`（`kernel/syscall.h:8`），导致了对系统调用接口函数 `sys_exec` 的调用。

当系统调用接口函数返回时，`syscall` 将其返回值记录在 `p->trapframe->a0` 中。这将导致原始用户空间对 `exec()` 的调用返回该值，因为RISC-V上的C调用约定将返回值放在 `a0` 中。系统调用通常返回负数表示错误，返回零或正数表示成功。如果系统调用号无效，`syscall` 打印错误并返回-1。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 11:38:44

- 4.4 系统调用参数

4.4 系统调用参数

内核中的系统调用接口需要找到用户代码传递的参数。因为用户代码调用了系统调用封装函数，所以参数最初被放置在RISC-V C调用所约定的地方：寄存器。内核陷阱代码将用户寄存器保存到当前进程的陷阱框架中，内核代码可以在那里找到它们。函数 `artint`、`artaddr` 和 `artfd` 从陷阱框架中检索第n个系统调用参数并以整数、指针或文件描述符的形式保存。他们都调用 `argraw` 来检索相应的保存的用户寄存器（*kernel/syscall.c:35*）。

有些系统调用传递指针作为参数，内核必须使用这些指针来读取或写入用户内存。例如：`exec` 系统调用传递给内核一个指向用户空间中字符串参数的指针数组。这些指针带来了两个挑战。首先，用户程序可能有缺陷或恶意，可能会传递给内核一个无效的指针，或者一个旨在欺骗内核访问内核内存而不是用户内存的指针。其次，`xv6` 内核页表映射与用户页表映射不同，因此内核不能使用普通指令从用户提供的地址加载或存储。

内核实现了安全地将数据传输到用户提供的地址和从用户提供的地址传输数据的功能。`fetchstr` 是一个例子（*kernel/syscall.c:25*）。文件系统调用，如 `exec`，使用 `fetchstr` 从用户空间检索字符串文件名参数。`fetchstr` 调用 `copyinstr` 来完成这项困难的工作。

`copyinstr`（*kernel/vm.c:406*）从用户页表表中的虚拟地址 `srcva` 复制 `max` 字节到 `dst`。它使用 `walkaddr`（它又调用 `walk`）在软件中遍历页表，以确定 `srcva` 的物理地址 `pa0`。由于内核将所有物理RAM地址映射到同一个内核虚拟地址，`copyinstr` 可以直接将字符串字节从 `pa0` 复制到 `dst`。`walkaddr`（*kernel/vm.c:95*）检查用户提供的虚拟地址是否为进程用户地址空间的一部分，因此程序不能欺骗内核读取其他内存。一个类似的函数 `copyout`，将数据从内核复制到用户提供的地址。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间：2021-08-19 11:39:13

- 4.5 从内核空间陷入

4.5 从内核空间陷入

xv6根据执行的是用户代码还是内核代码，对CPU陷阱寄存器的配置有所不同。当在CPU上执行内核时，内核将 `stvec` 指向 `kernelvec` (*kernel/kernelvec.S:10*)的汇编代码。由于xv6已经在内核中，`kernelvec` 可以依赖于设置为内核页表的 `satp`，以及指向有效内核栈的栈指针。`kernelvec` 保存所有寄存器，以便被中断的代码最终可以不受干扰地恢复。

`kernelvec` 将寄存器保存在被中断的内核线程的栈上，这是有意义的，因为寄存器值属于该线程。如果陷阱导致切换到不同的线程，那这一点就显得尤为重要——在这种情况下，陷阱将实际返回到新线程的栈上，将被中断线程保存的寄存器安全地保存在其栈上。

`Kernelvec` 在保存寄存器后跳转到 `kerneltrap` (*kernel/trap.c:134*)。`kerneltrap` 为两种类型的陷阱做好了准备：设备中断和异常。它调用 `devintr` (*kernel/trap.c:177*) 来检查和处理前者。如果陷阱不是设备中断，则必定是一个异常，内核中的异常将是一个致命的错误；内核调用 `panic` 并停止执行。

如果由于计时器中断而调用了 `kerneltrap`，并且一个进程的内核线程正在运行（而不是调度程序线程），`kerneltrap` 会调用 `yield`，给其他线程一个运行的机会。在某个时刻，其中一个线程会让步，让我们的线程和它的 `kerneltrap` 再次恢复。第7章解释了 `yield` 中发生的事情。

当 `kerneltrap` 的工作完成后，它需要返回到任何被陷阱中断的代码。因为一个 `yield` 可能已经破坏了保存的 `sepc` 和在 `sstatus` 中保存的前一个状态模式，因此 `kerneltrap` 在启动时保存它们。它现在恢复这些控制寄存器并返回到 `kernelvec` (*kernel/kernelvec.S:48*)。`kernelvec` 从栈中弹出保存的寄存器并执行 `sret`，将 `sepc` 复制到 `pc` 并恢复中断的内核代码。

值得思考的是，如果内核陷阱由于计时器中断而调用 `yield`，陷阱返回是如何发生的。

当CPU从用户空间进入内核时，xv6将CPU的 `stvec` 设置为 `kernelvec`；您可以在 `usertrap` (*kernel/trap.c:29*) 中看到这一点。内核执行时有一个时间窗口，但 `stvec` 设置为 `uservec`，在该窗口中禁用设备中断至关重要。幸运的是，RISC-V总是在开始设置陷阱时禁用中断，xv6在设置 `stvec` 之前不会再次启用中断。

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间：2021-08-19 11:41:29

- 4.6 页面错误异常

4.6 页面错误异常

Xv6对异常的响应相当无趣: 如果用户空间中发生异常, 内核将终止故障进程。如果内核中发生异常, 则内核会崩溃。真正的操作系统通常以更有趣的方式做出反应。

例如, 许多内核使用页面错误来实现写时拷贝版本的 `fork` ——*copy on write (COW fork)*。要解释*COW fork*, 请回忆第3章内容: xv6的 `fork` 通过调用 `uvncpy` (*kernel/vm.c*:309) 为子级分配物理内存, 并将父级的内存复制到其中, 使子级具有与父级相同的内存内容。如果父子进程可以共享父级的物理内存, 则效率会更高。然而武断地实现这种方法是行不通的, 因为它会导致父级和子级通过对共享栈和堆的写入来中断彼此的执行。

由页面错误驱动的*COW fork*可以使父级和子级安全地共享物理内存。当CPU无法将虚拟地址转换为物理地址时, CPU会生成页面错误异常。Risc-v有三种不同的页面错误: 加载页面错误(当加载指令无法转换其虚拟地址时), 存储页面错误(当存储指令无法转换其虚拟地址时)和指令页面错误(当指令的地址无法转换时)。`scause` 寄存器中的值指示页面错误的类型, `stval` 寄存器包含无法翻译的地址。

*COW fork*中的基本计划是让父子最初共享所有物理页面, 但将它们映射为只读。因此, 当子级或父级执行存储指令时, risc-v CPU引发页面错误异常。为了响应此异常, 内核复制了包含错误地址的页面。它在子级的地址空间中映射一个权限为读/写的副本, 在父级的地址空间中映射另一个权限为读/写的副本。更新页表后, 内核会在导致故障的指令处恢复故障进程的执行。由于内核已经更新了相关的PTE以允许写入, 所以错误指令现在将正确执行。

*COW*策略对 `fork` 很有效, 因为通常子进程会在 `fork` 之后立即调用 `exec`, 用新的地址空间替换其地址空间。在这种常见情况下, 子级只会触发很少的页面错误, 内核可以避免拷贝父进程内存完整的副本。此外, *COW fork*是透明的: 无需对应用程序进行任何修改即可使其受益。

除*COW fork*以外, 页表和页面错误的结合还开发出了广泛有趣的可能性。另一个广泛使用的特性叫做惰性分配——*lazy allocation*。它包括两部分内容: 首先, 当应用程序调用 `sbrk` 时, 内核增加地址空间, 但在页表中将新地址标记为无效。其次, 对于包含于其中的地址的页面错误, 内核分配物理内存并将其映射到页表中。由于应用程序通常要求比他们需要的更多的内存, 惰性分配可以称得上一次胜利: 内核仅在应用程序实际使用它时才分配内存。像*COW fork*一样, 内核可以对应用程序透明地实现此功能。

利用页面故障的另一个广泛使用的功能是从磁盘分页。如果应用程序需要比可用物理RAM更多的内存, 内核可以换出一些页面: 将它们写入存储设备(如磁盘), 并将它们的PTE标记为无效。如果应用程序读取或写入被换出的页面, 则CPU将触发页面错误。然后内核可以检查故障地址。如果该地址属于磁盘上的页面, 则内核分配物理内存页面, 将该页面从磁盘读取到该内存, 将PTE更新为有效并引用该内存, 然后恢复应用程序。为了给页面腾出空间, 内核可能需要换出另一个页面。此功能不需要对应用程序进行更改, 并且如果应用程序具有引用的地址(即, 它们在任何给定时间仅使用其内存的子集), 则该功能可以很好地工作。

结合分页和页面错误异常的其他功能包括自动扩展栈空间和内存映射文件。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 11:41:43

- 4.7 真实世界

4.7 真实世界

如果内核内存被映射到每个进程的用户页表中（带有适当的**PTE**权限标志），就可以消除对特殊蹦床页面的需求。这也将消除在从用户空间捕获到内核时对页表切换的需求。这反过来也将允许内核中的系统调用实现利用当前进程正在映射的用户内存，允许内核代码直接解引用用户指针。许多操作系统已经使用这些想法来提高效率。**Xv6**避免了这些漏洞，以减少由于无意中使用用户指针而导致内核中出现安全漏洞的可能性，并降低了确保用户和内核虚拟地址不重叠所需的一些复杂性。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 11:42:01

- 4.8 练习

4.8 练习

1. 函数 `copyin` 和 `copyinstr` 在软件中遍历用户页表。设置内核页表，使内核拥有用户程序的映射，这样 `copyin` 和 `copyinstr` 可以使用 `memcpy` 将系统调用参数复制到内核空间，依靠硬件进行页表遍历
2. 实现惰性内存分配(*lazy allocation*)
3. 实现写时拷贝版本的 `fork` (*copy on write fork*)

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间： 2021-08-19 11:42:19

- 第五章 中断和设备驱动

第五章 中断和设备驱动

驱动程序是操作系统中管理特定设备的代码：它配置硬件设备，告诉设备执行操作，处理由此产生的中断，并与可能正在等待设备输入/输出的进程进行交互。编写驱动可能很棘手，因为驱动程序与它管理的设备同时运行。此外，驱动程序必须理解设备的硬件接口，这可能很复杂，而且缺乏文档。

需要操作系统关注的设备通常可以被配置为生成中断，这是陷阱的一种。内核陷阱处理代码识别设备何时引发中断，并调用驱动程序的中断处理程序；在xv6中，这种调度发生在 `devintr` 中（***kernel/trap.c:177***）。

许多设备驱动程序在两种环境中执行代码：上半部分在进程的内核线程中运行，下半部分在中断时执行。上半部分通过系统调用进行调用，如希望设备执行I/O操作的 `read` 和 `write`。这段代码可能会要求硬件执行操作（例如，要求磁盘读取块）；然后代码等待操作完成。最终设备完成操作并引发中断。驱动程序的中断处理程序充当下半部分，计算出已经完成的操作，如果合适的话唤醒等待中的进程，并告诉硬件开始执行下一个正在等待的操作。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 13:46:55

- 5.1 代码：控制台输入

5.1 代码：控制台输入

控制台驱动程序（**console.c**）是驱动程序结构的简单说明。控制台驱动程序通过连接到RISC-V的UART串口硬件接受人们键入的字符。控制台驱动程序一次累积一行输入，处理如 `backspace` 和 `Ctrl-u` 的特殊输入字符。用户进程，如**Shell**，使用 `read` 系统调用从控制台获取输入行。当您在QEMU中通过键盘输入到xv6时，您的按键将通过QEMU模拟的UART硬件传递到xv6。

驱动程序管理的UART硬件是由QEMU仿真的16550芯片。在真正的计算机上，16550将管理连接到终端或其他计算机的RS232串行链路。运行QEMU时，它连接到键盘和显示器。

UART硬件在软件中看起来是一组内存映射的控制寄存器。也就是说，存在一些RISC-V硬件连接到UART的物理地址，以便载入(`load`)和存储(`store`)操作与设备硬件而不是内存交互。UART的内存映射地址起始于 `0x10000000` 或 `UART0` (*kernel/memlayout.h:21*)。有几个宽度为一字节的UART控制寄存器，它们关于UART0的偏移量在(*kernel/uart.c:22*)中定义。例如，LSR寄存器包含指示输入字符是否正在等待软件读取的位。这些字符（如果有的话）可用于从RHR寄存器读取。每次读取一个字符，UART硬件都会从等待字符的内部FIFO寄存器中删除它，并在FIFO为空时清除LSR中的“就绪”位。UART传输硬件在很大程度上独立于接收硬件；如果软件向THR写入一个字节，则UART传输该字节。

Xv6的 `main` 函数调用 `consoleinit` (*kernel/console.c:184*) 来初始化UART硬件。该代码配置UART：UART对接收到的每个字节的输入生成一个接收中断，对发送完的每个字节的输出生成一个发送完成中断 (*kernel/uart.c:53*)。

xv6的shell通过 `init.c` (*user/init.c:19*) 中打开的文件描述符从控制台读取输入。对 `read` 的调用实现了从内核流向 `consoleread` (*kernel/console.c:82*) 的数据通路。`consoleread` 等待输入到达（通过中断）并在 `cons.buf` 中缓冲，将输入复制到用户空间，然后（在整行到达后）返回给用户进程。如果用户还没有键入整行，任何读取进程都将在 `sleep` 系统调用中等待 (*kernel/console.c:98*)（第7章解释了 `sleep` 的细节）。

当用户输入一个字符时，UART硬件要求RISC-V发出一个中断，从而激活xv6的陷阱处理程序。陷阱处理程序调用 `devintr` (*kernel/trap.c:177*)，它查看RISC-V的 `scause` 寄存器，发现中断来自外部设备。然后它要求一个称为PLIC的硬件单元告诉它哪个设备中断了 (*kernel/trap.c:186*)。如果是UART，`devintr` 调用 `uartintr`。

`uartintr` (*kernel/uart.c:180*) 从UART硬件读取所有等待输入的字符，并将它们交给 `consoleintr` (*kernel/console.c:138*)；它不会等待字符，因为未来的输入将引发一个新的中断。`consoleintr` 的工作是在 `cons.buf` 中积累输入字符，直到一整行到达。`consoleintr` 对 `backspace` 和其他少量字符进行特殊处理。当换行符到达时，`consoleintr` 唤醒一个等待的 `consoleread`（如果有的话）。

一旦被唤醒，`consoleread` 将监视 `cons.buf` 中的一整行，将其复制到用户空间，并返回（通过系统调用机制）到用户空间。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 13:47:08

- 5.2 代码：控制台输出

5.2 代码：控制台输出

在连接到控制台的文件描述符上执行 `write` 系统调用，最终将到达 `uartputc` (*kernel/uart.c:87*)。设备驱动程序维护一个输出缓冲区 (`uart_tx_buf`)，这样写进程就不必等待UART完成发送；相反，`uartputc` 将每个字符附加到缓冲区，调用 `uartstart` 来启动设备传输（如果还未启动），然后返回。导致 `uartputc` 等待的唯一情况是缓冲区已满。

每当UART发送完一个字节，它就会产生一个中断。`uartintr` 调用 `uartstart`，检查设备是否真的完成了发送，并将下一个缓冲的输出字符交给设备。因此，如果一个进程向控制台写入多个字节，通常第一个字节将由 `uartputc` 调用 `uartstart` 发送，而剩余的缓冲字节将由 `uartintr` 调用 `uartstart` 发送，直到传输完成中断到来。

需要注意，这里的一般模式是通过缓冲区和中断机制将设备活动与进程活动解耦。即使没有进程等待读取输入，控制台驱动程序仍然可以处理输入，而后续的读取将看到这些输入。类似地，进程无需等待设备就可以发送输出。这种解耦可以通过允许进程与设备I/O并发执行来提高性能，当设备很慢（如UART）或需要立即关注（如回声型字符(echoing typed characters)）时，这种解耦尤为重要。这种想法有时被称为I/O并发

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 13:49:47

- 5.3 驱动中的并发

5.3 驱动中的并发

你或许注意到了在 `consoleread` 和 `consoleintr` 中对 `acquire` 的调用。这些调用获得了一个保护控制台驱动程序的数据结构不受并发访问的锁。这里有三种并发风险：运行在不同CPU上的两个进程可能同时调用 `consoleread`；硬件或许会在 `consoleread` 正在执行时要求CPU传递控制台中断；并且硬件可能在当前CPU正在执行 `consoleread` 时向其他CPU传递控制台中断。第6章探讨了锁在这些场景中的作用。

在驱动程序中需要注意并发的另一种场景是，一个进程可能正在等待来自设备的输入，但是输入的中断信号可能是在另一个进程（或者根本没有进程）正在运行时到达的。因此中断处理程序不允许考虑他们已经中断的进程或代码。例如，中断处理程序不能安全地使用当前进程的页表调用 `copyout`（注：因为你不知道是否发生了进程切换，当前进程可能并不是原先的进程）。中断处理程序通常做相对较少的工作（例如，只需将输入数据复制到缓冲区），并唤醒上半部分代码来完成其余工作。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 13:50:00

- 5.4 定时器中断

5.4 定时器中断

Xv6使用定时器中断来维持其时钟，并使其能够在受计算量限制的进程（compute-bound processes）之间切换；`usertrap` 和 `kerneltrap` 中的 `yield` 调用会导致这种切换。定时器中断来自附加到每个RISC-V CPU上的时钟硬件。Xv6对该时钟硬件进行编程，以定期中断每个CPU。

RISC-V要求定时器中断在机器模式而不是管理模式下进行。RISC-V机器模式无需分页即可执行，并且有一组单独的控制寄存器，因此在机器模式下运行普通的xv6内核代码是不实际的。因此，xv6处理定时器中断完全不同于上面列出的陷阱机制。

机器模式下执行的代码位于 `main` 之前的 `start.c` 中，它设置了接收定时器中断（`kernel/start.c:57`）。工作的一部分是对CLINT（core-local interruptor）硬件编程，以在特定延迟后生成中断。另一部分是设置一个`scratch`区域，类似于`trapframe`，以帮助定时器中断处理程序保存寄存器和CLINT寄存器的地址。最后，`start` 将 `mtvec` 设置为 `timervec`，并使能定时器中断。

计时器中断可能发生在用户或内核代码正在执行的任何时候；内核无法在临界区操作期间禁用计时器中断。因此，计时器中断处理程序必须保证不干扰中断的内核代码。基本策略是处理程序要求RISC-V发出“软件中断”并立即返回。RISC-V用普通陷阱机制将软件中断传递给内核，并允许内核禁用它们。处理由定时器中断产生的软件中断的代码可以在 `devintr` (`kernel/trap.c:204`)中看到。

机器模式定时器中断向量是 `timervec` (`kernel/kernelvec.S:93`)。它在 `start` 准备的`scratch`区域中保存一些寄存器，以告诉CLINT何时生成下一个定时器中断，要求RISC-V引发软件中断，恢复寄存器，并且返回。定时器中断处理程序中没有C代码。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间：2021-08-19 13:50:09

5.5 真实世界

Xv6允许在内核中执行时以及在执行用户程序时触发设备和定时器中断。定时器中断迫使定时器中断处理程序进行线程切换（调用 `yield`），即使在内核中执行时也是如此。如果内核线程有时花费大量时间计算而不返回用户空间，则在内核线程之间公平地对CPU进行时间分割的能力非常有用。然而，内核代码需要注意它可能被挂起（由于计时器中断），然后在不同的CPU上恢复，这是xv6中一些复杂性的来源。如果设备和计时器中断只在执行用户代码时发生，内核可以变得简单一些。

在一台典型的计算机上支持所有设备是一项艰巨的工作，因为有许多设备，这些设备有许多特性，设备和驱动程序之间的协议可能很复杂，而且缺乏文档。在许多操作系统中，驱动程序比核心内核占用更多的代码。

UART驱动程序读取UART控制寄存器，一次检索一字节的数据；因为软件驱动数据移动，这种模式被称为程序I/O（Programmed I/O）。程序I/O很简单，但速度太慢，无法在高数据速率下使用。需要高速移动大量数据的设备通常使用直接内存访问（DMA）。DMA设备硬件直接将传入数据写入内存，并从内存中读取传出数据。现代磁盘和网络设备使用DMA。DMA设备的驱动程序将在RAM中准备数据，然后使用对控制寄存器的单次写入来告诉设备处理准备好的数据。

当一个设备在不可预知的时间需要注意时，中断是有意义的，而且不是太频繁。但是中断有很高的CPU开销。因此，如网络和磁盘控制器的高速设备，使用一些技巧减少中断需求。一个技巧是对整批传入或传出的请求发出单个中断。另一个技巧是驱动程序完全禁用中断，并定期检查设备是否需要注意。这种技术被称为轮询（polling）。如果设备执行操作非常快，轮询是有意义的，但是如果设备大部分空闲，轮询会浪费CPU时间。一些驱动程序根据当前设备负载在轮询和中断之间动态切换。

UART驱动程序首先将传入的数据复制到内核中的缓冲区，然后复制到用户空间。这在低数据速率下是可行的，但是这种双重复制会显著降低快速生成或消耗数据的设备的性能。一些操作系统能够直接在用户空间缓冲区和设备硬件之间移动数据，通常带有DMA。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 13:50:18

- 5.6 练习

5.6 练习

1. 修改**uart.c**以完全不使用中断。您可能还需要修改**console.c**
2. 为以太网卡添加驱动程序

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间： 2021-08-19 13:50:46

第六章 锁

大多数内核，包括xv6，交错执行多个活动。交错的一个来源是多处理器硬件：计算机的多个CPU之间独立执行，如xv6的RISC-V。多个处理器共享物理内存，xv6利用共享（sharing）来维护所有CPU进行读写的数据结构。这种共享增加了一种可能性，即一个CPU读取数据结构，而另一个CPU正在更新它，甚至多个CPU同时更新相同的数据；如果不仔细设计，这种并行访问可能会产生不正确的结果或损坏数据结构。即使在单处理器上，内核也可能在许多线程之间切换CPU，导致它们的执行交错。最后，如果中断发生在错误的时间，设备中断处理程序修改与某些可中断代码相同的数据，可能导致数据损坏。单词并发（concurrency）是指由于多处理器并行、线程切换或中断，多个指令流交错的情况。

内核中充满了并发访问数据（concurrently-accessed data）。例如，两个CPU可以同时调用 `kalloc`，从而从空闲列表的头部弹出。内核设计者希望允许大量的并发，因为这样可通过并行性提高性能，并提高响应能力。然而，结果是，尽管存在这种并发性，内核设计者还是花费了大量的精力来使其正确运行。有许多方法可以得到正确的代码，有些方法比其他方法更容易。以并发下的正确性为目标的策略和支持它们的抽象称为并发控制技术（concurrency control techniques）。

Xv6使用了许多并发控制技术，这取决于不同的情况。本章重点介绍了一种广泛使用的技术：锁。锁提供了互斥，确保一次只有一个CPU可以持有锁。如果程序员将每个共享数据项关联一个锁，并且代码在使用一个数据项时总是持有相关联的锁，那么该项一次将只被一个CPU使用。在这种情况下，我们说锁保护数据项。尽管锁是一种易于理解的并发控制机制，但锁的缺点是它们会扼杀性能，因为它们会串行化并发操作。

本章的其余部分解释了为什么xv6需要锁，xv6如何实现它们，以及如何使用它们。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间：2021-08-19 13:51:10

- 6.1 竞态条件

6.1 竞态条件

作为我们为什么需要锁的一个例子，考虑两个进程在两个不同的CPU上调用 `wait`。`wait` 释放了子进程的内存。因此，在每个CPU上，内核将调用 `kfree` 来释放子进程的页面。内核分配器维护一个链接列表：`kalloc()` (*kernel/kalloc.c:69*) 从空闲页面列表中取出（`pop`）一个内存页面；`kfree()` (*kernel/kalloc.c:47*) 将一个内存页面添加（`push`）到空闲列表上。为了获得最佳性能，我们可能希望两个父进程的 `kfree` 可以并行执行，而不必等待另一个进程，但是考虑到xv6的 `kfree` 实现，这将导致错误。

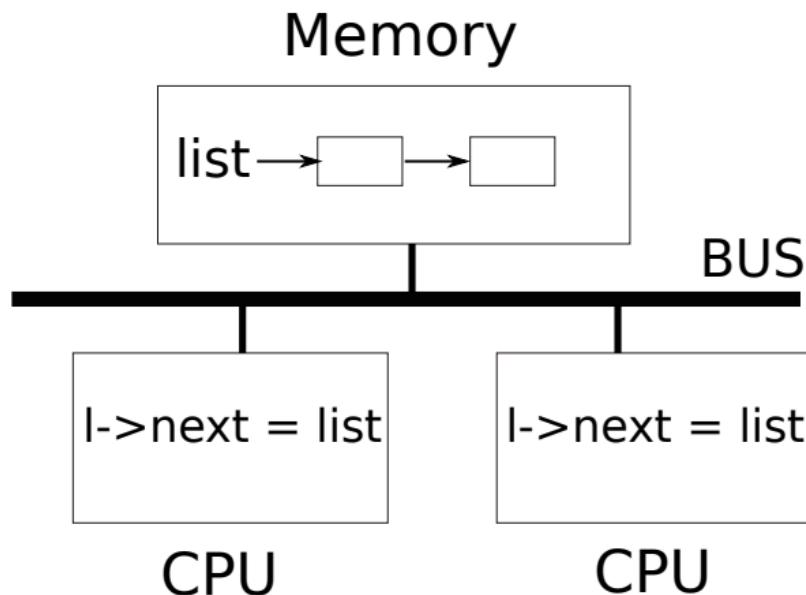


Figure 6.1: Simplified SMP architecture

图6.1更详细地说明了这项设定：链表位于两个CPU共享的内存中，这两个CPU使用 `load` 和 `store` 指令操作链表。（实际上，每个处理器都有 `cache`，但从概念上讲，多处理器系统的行为就像所有CPU共享一块单独的内存一样）如果没有并发请求，您可能以如下方式实现列表push操作：

```

struct element {
    int data;
    struct element *next;
};

struct element *list = 0;

void
push(int data)
{
    struct element *l;

    l = malloc(sizeof *l);
    l->data = data;
    l->next = list;
    list = l;
}

```

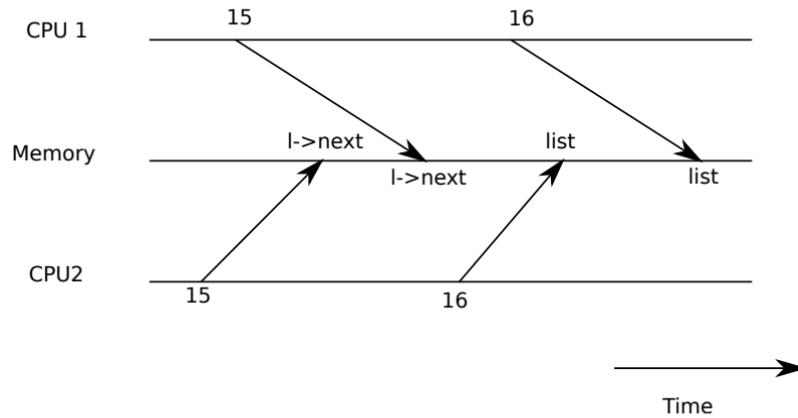


Figure 6.2: Example race

如果存在隔离性，那么这个实现是正确的。但是，如果多个副本并发执行，代码就会出错。如果两个CPU同时执行 `push`，如图6.1所示，两个CPU都可能在执行第16行之前执行第15行，这会导致如图6.2所示的不正确的结果。然后会有两个类型为 `element` 的列表元素使用 `next` 指针设置为 `list` 的前一个值。当两次执行位于第16行的对 `list` 的赋值时，第二次赋值将覆盖第一次赋值；第一次赋值中涉及的元素将丢失。

第16行丢失的更新是竞态条件（race condition）的一个例子。竞态条件是指多个进程读写某些共享数据（至少有一个访问是写入）的情况。竞争通常包含bug，要么丢失更新（如果访问是写入的），要么读取未完成更新的数据结构。竞争的结果取决于进程在处理器运行的确切时机以及内存系统如何排序它们的内存操作，这可能会使竞争引起的错误难以复现和调试。例如，在调试 `push` 时添加 `printf` 语句可能会改变执行的时间，从而使竞争消失。

避免竞争的通常方法是使用锁。锁确保互斥，这样一次只有一个CPU可以执行 `push` 中敏感的代码行；这使得上述情况不可能发生。上面代码的正确上锁版本只添加了几行（用黄色突出显示）：

```

struct element {
    int data;
    struct element *next;
};

struct element *list = 0;
struct lock listlock;

void
push(int data)
{
    struct element *l;

    l = malloc(sizeof *l);
    l->data = data;
    acquire(&listlock);
    l->next = list;
    list = l;
    release(&listlock);
}

```

`acquire` 和 `release` 之间的指令序列通常被称为临界区域（critical section）。锁的作用通常被称为保护 `list`。

当我们说锁保护数据时，我们实际上是指锁保护适用于数据的某些不变量集合。不变量是跨操作维护的数据结构的属性。通常，操作的正确行为取决于操作开始时不变量是否为真。操作可能暂时违反不变量，但必须在完成之前重新建立它们。例如，在链表的例子中，不变量是 `list` 指向列表中的第一个元素，以及每个元素的 `next` 字段指向下一个元素。`push` 的实现暂时违反了这个不变量：在第17行，`l->next` 指向 `list`（注：则此时 `list` 不再指向列表中的第一个元素，即违反了不变量），但是 `list` 还没有指向 `l`（在第18行重新建立）。我们上面检查的竞争条件发生了，因为第二个CPU执行了依赖于列表不变量的代码，而这些代码（暂时）被违反了。正确使用锁可以确保每次只有一个CPU可以对临界区域中的数据结构进行操作，因此当数据结构的不变量不成立时，将没有其他CPU对数据结构执行操作。

您可以将锁视为串行化（serializing）并发的临界区域，以便同时只有一个进程在运行这部分代码，从而维护不变量（假设临界区域设定了正确的隔离性）。您还可以将由同一锁保护的临界区域视为彼此之间的原子，即彼此之间只能看到之前临界区域的完整更改集，而永远看不到部分完成的更新。

尽管正确使用锁可以改正不正确的代码，但锁限制了性能。例如，如果两个进程并发调用 `kfree`，锁将串行化这两个调用，我们在不同的CPU上运行它们没有任何好处。如果多个进程同时想要相同的锁或者锁经历了争用，则称之为发生冲突（conflict）。内核设计中的一个主要挑战是避免锁争用。Xv6为此几乎没做任何工作，但是复杂的内核会精心设计数据结构和算法来避免锁的争用。在链表示例中，内核可能会为每个CPU维护一个空闲列表，并且只有当CPU的列表为空并且必须从另一个CPU挪用内存时才会触及另一个CPU的空闲列表。其他用例可能需要更复杂的设计。

锁的位置对性能也很重要。例如，在 `push` 中把 `acquire` 的位置提前也是正确的：将 `acquire` 移动到第13行之前完全没问题。但这样对 `malloc` 的调用也会被串行化，从而降低了性能。下面的《使用锁》一节提供了一些关于在哪里插入 `acquire` 和 `release` 调用的指导方针。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 13:52:30

- 6.2 代码: Locks

6.2 代码: Locks

Xv6有两种类型的锁：自旋锁（spinlocks）和睡眠锁（sleep-locks）。我们将从自旋锁（注：自旋，即循环等待）开始。Xv6将自旋锁表示为 `struct spinlock` (*kernel/spinlock.h:2*)。结构体中的重要字段是 `locked`，当锁可用时为零，当它被持有时为非零。从逻辑上讲，xv6应该通过执行以下代码来获取锁

```
void
acquire(struct spinlock* lk) // does not work!
{
    for(;;) {
        if(lk->locked == 0) {
            lk->locked = 1;
            break;
        }
    }
}
```

不幸的是，这种实现不能保证多处理器上的互斥。可能会发生两个CPU同时到达第5行，看到 `lk->locked` 为零，然后都通过执行第6行占有锁。此时就有两个不同的CPU持有锁，从而违反了互斥属性。我们需要的是一种方法，使第5行和第6行作为原子（即不可分割）步骤执行。

因为锁被广泛使用，多核处理器通常提供实现第5行和第6行的原子版本的指令。在RISC-V上，这条指令是 `amoswap r, a`。`amoswap` 读取内存地址 `a` 处的值，将寄存器 `r` 的内容写入该地址，并将其读取的值放入 `r` 中。也就是说，它交换寄存器和指定内存地址的内容。它原子地执行这个指令序列，使用特殊的硬件来防止任何其他CPU在读取和写入之间使用内存地址。

Xv6的 `acquire` (*kernel/spinlock.c:22*) 使用可移植的C库调用归结为 `amoswap` 的指令 `_sync_lock_test_and_set`；返回值是 `lk->locked` 的旧（交换了的）内容。`acquire` 函数将 `swap` 包装在一个循环中，直到它获得了锁前一直重试（自旋）。每次迭代将1与 `lk->locked` 进行 `swap` 操作，并检查 `lk->locked` 之前的价值。如果之前为0，`swap` 已经把 `lk->locked` 设置为1，那么我们就获得了锁；如果前一个值是1，那么另一个CPU持有锁，我们原子地将1与 `lk->locked` 进行 `swap` 的事实并没有改变它的值。

获取锁后，用于调试，`acquire` 将记录下来获取锁的CPU。`lk->cpu` 字段受锁保护，只能在保持锁时更改。

函数 `release` (*kernel/spinlock.c:47*) 与 `acquire` 相反：它清除 `lk->cpu` 字段，然后释放锁。从概念上讲，`release` 只需要将0分配给 `lk->locked`。C标准允许编译器用多个存储指令实现赋值，因此对于并发代码，C赋值可能是非原子的。因此 `release` 使用执行原子赋值的C库函数 `_sync_lock_release`。该函数也可以归结为RISC-V的 `amoswap` 指令。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间：2021-08-19 13:53:14

- 6.3 代码：使用锁

6.3 代码：使用锁

Xv6在许多地方使用锁来避免竞争条件（race conditions）。如上所述，`kalloc` (*kernel/kalloc.c:69*)和`kfree` (*kernel/kalloc.c:47*)就是一个很好的例子。尝试练习1和练习2，看看如果这些函数省略了锁会发生什么。你可能会发现很难触发不正确的行为，这表明很难可靠地测试代码是否经历了锁错误和竞争后被释放。`xv6`有一些竞争是有可能发生的。

使用锁的一个困难部分是决定要使用多少锁，以及每个锁应该保护哪些数据和不变量。有几个基本原则。首先，任何时候可以被一个CPU写入，同时又可以被另一个CPU读写的变量，都应该使用锁来防止两个操作重叠。其次，请记住锁保护不变量（invariants）：如果一个不变量涉及多个内存位置，通常所有这些位置都需要由一个锁来保护，以确保不变量不被改变。

上面的规则说什么时候需要锁，但没有说什么时候不需要锁。为了提高效率，不要向太多地方上锁是很重要的，因为锁会降低并行性。如果并行性不重要，那么可以安排只拥有一个线程，而不用担心锁。一个简单的内核可以在多处理器上做到这一点，方法是拥有一个锁，这个锁必须在进入内核时获得，并在退出内核时释放（尽管如管道读取或`wait`的系统调用会带来问题）。许多单处理器操作系统已经被转换为使用这种方法在多处理器上运行，有时被称为“大内核锁（big kernel lock）”，但是这种方法牺牲了并行性：一次只能有一个CPU运行在内核中。如果内核做一些繁重的计算，使用一组更细粒度的锁的集合会更有效率，这样内核就可以同时在多个处理器上执行。

作为粗粒度锁的一个例子，`xv6`的`kalloc.c`分配器有一个由单个锁保护的空闲列表。如果不同CPU上的多个进程试图同时分配页面，每个进程在获得锁之前将必须在`acquire`中自旋等待。自旋会降低性能，因为它只是无用的等待。如果对锁的竞争浪费了很大一部分CPU时间，也许可以通过改变分配器的设计来提高性能，使其拥有多个空闲列表，每个列表都有自己的锁，以允许真正的并行分配。

作为细粒度锁定的一个例子，`xv6`对每个文件都有一个单独的锁，这样操作不同文件的进程通常可以不需等待彼此的锁而继续进行。文件锁的粒度可以进一步细化，以允许进程同时写入同一个文件的不同区域。最终的锁粒度决策需要由性能测试和复杂性考量来驱动。

在后面的章节解释`xv6`的每个部分时，他们将提到`xv6`使用锁来处理并发的例子。作为预览，表6.3列出了`xv6`中的所有锁。

锁	描述
<code>bcache.lock</code>	保护块缓冲区缓存项（ block buffer cache entries ）的分配
<code>cons.lock</code>	串行化对控制台硬件的访问，避免混合输出
<code>ftable.lock</code>	串行化文件表中文件结构体的分配
<code>icache.lock</code>	保护索引结点缓存项（ inode cache entries ）的分配
<code>vdisk_lock</code>	串行化对磁盘硬件和 DMA 描述符队列的访问
<code>kmem.lock</code>	串行化内存分配
<code>log.lock</code>	串行化事务日志操作
管道的 <code>pi->lock</code>	串行化每个管道的操作
<code>pid_lock</code>	串行化 next_pid 的增量
进程的 <code>p->lock</code>	串行化进程状态的改变
<code>tickslock</code>	串行化时钟计数操作
索引结点的 <code>ip->lock</code>	串行化索引结点及其内容的操作
缓冲区的 <code>b->lock</code>	串行化每个块缓冲区的操作

Figure 6.3: Locks in xv6

copyright by duguosheng all right reserved, powered by Gitbook
 该文件修订时间： 2021-08-19 13:53:42

- 6.4 死锁和锁排序

6.4 死锁和锁排序

如果在内核中执行的代码路径必须同时持有数个锁，那么所有代码路径以相同的顺序获取这些锁是很重要的。如果它们不这样做，就有死锁的风险。假设xv6中的两个代码路径需要锁A和B，但是代码路径1按照先A后B的顺序获取锁，另一个路径按照先B后A的顺序获取锁。假设线程T1执行代码路径1并获取锁A，线程T2执行代码路径2并获取锁B。接下来T1将尝试获取锁B，T2将尝试获取锁A。两个获取都将无限期阻塞，因为在这两种情况下，另一个线程都持有所需的锁，并且不会释放它，直到它的获取返回。为了避免这种死锁，所有代码路径必须以相同的顺序获取锁。全局锁获取顺序的需求意味着锁实际上是每个函数规范的一部分：调用者必须以一种使锁按照约定顺序被获取的方式调用函数。

由于 `sleep` 的工作方式（见第7章），Xv6有许多包含每个进程的锁（每个 `struct proc` 中的锁）在内的长度为2的锁顺序链。例如，`consoleintr` (*kernel/console.c:138*) 是处理键入字符的中断例程。当换行符到达时，任何等待控制台输入的进程都应该被唤醒。为此，`consoleintr` 在调用 `wakeup` 时持有 `cons.lock`，`wakeup` 获取等待进程的锁以唤醒它。因此，全局避免死锁的锁顺序包括必须在任何进程锁之前获取 `cons.lock` 的规则。文件系统代码包含xv6最长的锁链。例如，创建一个文件需要同时持有目录上的锁、新文件inode上的锁、磁盘块缓冲区上的锁、磁盘驱动程序的 `vdisk_lock` 和调用进程的 `p->lock`。为了避免死锁，文件系统代码总是按照前一句中提到的顺序获取锁。

遵守全局死锁避免的顺序可能会出人意料地困难。有时锁顺序与逻辑程序结构相冲突，例如，也许代码模块M1调用模块M2，但是锁顺序要求在M1中的锁之前获取M2中的锁。有时锁的身份是事先不知道的，也许是因为必须持有一个锁才能发现下一个要获取的锁的身份。这种情况在文件系统中出现，因为它在路径名称中查找连续的组件，也在 `wait` 和 `exit` 代码中出现，因为它们在进程表中寻找子进程。最后，死锁的危险通常是对细粒度锁定方案的限制，因为更多的锁通常意味着更多的死锁可能性。避免死锁的需求通常是内核实现中的一个主要因素。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 13:53:56

- 6.5 锁和中断处理函数

6.5 锁和中断处理函数

一些xv6自旋锁保护线程和中断处理程序共用的数据。例如，`clockintr` 定时器中断处理程序在增加 `ticks` (*kernel/trap.c:163*) 的同时内核线程可能在 `sys_sleep` (*kernel/sysproc.c:64*) 中读取 `ticks`。锁 `tickslock` 串行化这两个访问。

自旋锁和中断的交互引发了潜在的危险。假设 `sys_sleep` 持有 `tickslock`，并且它的 CPU 被计时器中断中断。`clockintr` 会尝试获取 `tickslock`，意识到它被持有后等待释放。在这种情况下，`tickslock` 永远不会被释放：只有 `sys_sleep` 可以释放它，但是 `sys_sleep` 直到 `clockintr` 返回前不能继续运行。所以 CPU 会死锁，任何需要锁的代码也会冻结。

为了避免这种情况，如果一个自旋锁被中断处理程序所使用，那么 CPU 必须保证在启用中断的情况下永远不能持有该锁。Xv6 更保守：当 CPU 获取任何锁时，xv6 总是禁用该 CPU 上的中断。中断仍然可能发生在其他 CPU 上，此时中断的 `acquire` 可以等待线程释放自旋锁；由于不在同一 CPU 上，不会造成死锁。

当 CPU 未持有自旋锁时，xv6 重新启用中断；它必须做一些记录来处理嵌套的临界区域。`acquire` 调用 `push_off` (*kernel/spinlock.c:89*) 并且 `release` 调用 `pop_off` (*kernel/spinlock.c:100*) 来跟踪当前 CPU 上锁的嵌套级别。当计数达到零时，`pop_off` 恢复最外层临界区域开始时存在的中断使能状态。`intr_off` 和 `intr_on` 函数执行 RISC-V 指令分别用来禁用和启用中断。

严格的在设置 `lk->locked` (*kernel/spinlock.c:28*) 之前让 `acquire` 调用 `push_off` 是很重要的。如果两者颠倒，会存在一个既持有锁又启用了中断的短暂窗口期，不幸的话定时器中断会使系统死锁。同样，只有在释放锁之后，`release` 才调用 `pop_off` 也是很重要的(*kernel/spinlock.c:66*)。

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间：2021-08-19 13:54:09

- 6.6 指令和内存访问排序

6.6 指令和内存访问排序

人们很自然地会想到程序是按照源代码语句出现的顺序执行的。然而，许多编译器和中央处理器为了获得更高的性能而不按顺序执行代码。如果一条指令需要许多周期才能完成，中央处理器可能会提前发出指令，这样它就可以与其他指令重叠，避免中央处理器停顿。例如，中央处理器可能会注意到在顺序指令序列A和B中彼此不存在依赖。CPU也许首先启动指令B，或者是因为它的输入先于A的输入准备就绪，或者是为了重叠执行A和B。编译器可以执行类似的重新排序，方法是在源代码中一条语句的指令发出之前，先发出另一条语句的指令。

编译器和CPU在重新排序时需要遵循一定规则，以确保它们不会改变正确编写的串行代码的结果。然而，规则确实允许重新排序后改变并发代码的结果，并且很容易导致多处理器上的不正确行为。CPU的排序规则称为内存模型（memory model）。

例如，在 `push` 的代码中，如果编译器或CPU将对应于第4行的存储指令移动到第6行 `release` 后的某个地方，那将是一场灾难：

```
l = malloc(sizeof *l);
l->data = data;
acquire(&listlock);
l->next = list;
list = l;
release(&listlock);
```

如果发生这样的重新排序，将会有个窗口期，另一个CPU可以获取锁并查看更新后的 `list`，但却看到一个未初始化的 `list->next`。

为了告诉硬件和编译器不要执行这样的重新排序，xv6在 `acquire (kernel/spinlock.c:22)` 和 `release (kernel/spinlock.c:47)` 中都使用了 `_sync_synchronize()`。`_sync_synchronize()` 是一个内存障碍：它告诉编译器和CPU不要跨障碍重新排序 `load` 或 `store` 指令。因为xv6在访问共享数据时使用了锁，xv6的 `acquire` 和 `release` 中的障碍在几乎所有重要的情况下都会强制顺序执行。第9章讨论了一些例外。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间：2021-08-19 13:55:14

- 6.7 睡眠锁

6.7 睡眠锁

有时xv6需要长时间保持锁。例如，文件系统（第8章）在磁盘上读写文件内容时保持文件锁定，这些磁盘操作可能需要几十毫秒。如果另一个进程想要获取自旋锁，那么长时间保持自旋锁会导致获取进程在自旋时浪费很长时间的CPU。自旋锁的另一个缺点是，一个进程在持有自旋锁的同时不能让出（`yield`）CPU，然而我们希望持有锁的进程等待磁盘I/O的时候其他进程可以使用CPU。持有自旋锁时让步是非法的，因为如果第二个线程试图获取自旋锁，就可能导致死锁：因为 `acquire` 不会让出CPU，第二个线程的自旋可能会阻止第一个线程运行并释放锁。在持有锁时让步也违反了在持有自旋锁时中断必须关闭的要求。因此，我们想要一种锁，它在等待获取锁时让出CPU，并允许在持有锁时让步（以及中断）。

Xv6以睡眠锁（**sleep-locks**）的形式提供了这种锁。`acquiresleep` (*kernel/sleeplock.c:22*) 在等待时让步CPU，使用的技术将在第7章中解释。在更高层次上，睡眠锁有一个被自旋锁保护的锁定字段，`acquiresleep` 对 `sleep` 的调用原子地让出CPU并释放自旋锁。结果是其他线程可以在 `acquiresleep` 等待时执行。

因为睡眠锁保持中断使能，所以它们不能用在中断处理程序中。因为 `acquiresleep` 可能让出CPU，所以睡眠锁不能在自旋锁临界区域中使用（尽管自旋锁可以在睡眠锁临界区域中使用）。

因为等待会浪费CPU时间，所以自旋锁最适合短的临界区域；睡眠锁对于冗长的操作效果很好。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 13:54:32

6.8 真实世界

尽管对并发原语和并行性进行了多年的研究，但使用锁进行编程仍然具有挑战性。通常最好将锁隐藏在更高级别的结构中，如同步队列，尽管xv6没有这样做。如果您使用锁进行编程，明智的做法是使用试图识别竞争条件（race conditions）的工具，因为很容易错过需要锁的不变量。

大多数操作系统都支持POSIX线程（Pthread），它允许一个用户进程在不同的CPU上同时运行几个线程。Pthread支持用户级锁（user-level locks）、障碍（barriers）等。支持Pthread需要操作系统的支持。例如，应该是这样的情况，如果一个Pthread在系统调用中阻塞，同一进程的另一个Pthread应当能够在该CPU上运行。另一个例子是，如果一个线程改变了其进程的地址空间（例如，映射或取消映射内存），内核必须安排运行同一进程下的线程的其他CPU更新其硬件页表，以反映地址空间的变化。

没有原子指令实现锁是可能的，但是代价昂贵，并且大多数操作系统使用原子指令。

如果许多CPU试图同时获取相同的锁，可能会付出昂贵的开销。如果一个CPU在其本地cache中缓存了一个锁，而另一个CPU必须获取该锁，那么更新保存该锁的cache行的原子指令必须将该行从一个CPU的cache移动到另一个CPU的cache中，并且可能会使cache行的任何其他副本无效。从另一个CPU的cache中获取cache行可能比从本地cache中获取一行的代价要高几个数量级。

为了避免与锁相关的开销，许多操作系统使用无锁的数据结构和算法。例如，可以实现一个像本章开头那样的链表，在列表搜索期间不需要锁，并且使用一个原子指令在一个列表中插入一个条目。然而，无锁编程比有锁编程更复杂；例如，人们必须担心指令和内存重新排序。有锁编程已经很难了，所以xv6避免了无锁编程的额外复杂性。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 13:54:41

- 6.9 练习

6.9 练习

1. 注释掉在 `kalloc` 中对 `acquire` 和 `release` 的调用。这似乎会给调用 `kalloc` 的内核代码带来问题；你希望看到什么症状？当你运行 `xv6` 时，你看到这些症状了吗？运行 `usertests` 时呢？如果你没有看到问题是为什么呢？看看你是否可以通过在 `kalloc` 的临界区域插入虚拟循环来引发问题。
2. 假设您将 `kfree` 中的锁注释掉（在 `kalloc` 中恢复锁之后）。现在可能会出什么问题？`kfree` 中缺少锁比 `kalloc` 中缺少锁的危害小吗？
3. 如果两个CPU同时调用 `kalloc`，则其中一个不得不等待另一个，这对性能不利。修改 `kalloc.c` 以具有更多的并行性，这样不同CPU对 `kalloc` 的同时调用就可以进行，而不需要相互等待。
4. 使用POSIX线程编写一个并行程序，大多数操作系统都支持这种程序。例如，实现一个并行哈希表，并测量 `puts/gets` 的数量是否随着内核数量的增加而缩放。
5. 在 `xv6` 中实现 `Pthread` 的一个子集。也就是说，实现一个用户级线程库，这样一个用户进程可以有1个以上的线程，并安排这些线程可以在不同的CPU上并行运行。想出一个正确处理线程发出阻塞系统调用并改变其共享地址空间的方案。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 13:55:03

- 第七章 调度

第七章 调度

任何操作系统都可能运行比CPU数量更多的进程，所以需要一个进程间分时共享CPU的方案。这种共享最好对用户进程透明。一种常见的方法是，通过将进程多路复用到硬件CPU上，使每个进程产生一种错觉，即它有自己的虚拟CPU。本章解释了XV6如何实现这种多路复用。

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间： 2021-08-19 13:55:41

- 7.1 多路复用

7.1 多路复用

Xv6通过在两种情况下将每个CPU从一个进程切换到另一个进程来实现多路复用（Multiplexing）。第一：当进程等待设备或管道I/O完成，或等待子进程退出，或在 sleep 系统调用中等待时，xv6使用睡眠（sleep）和唤醒（wakeup）机制切换。第二：xv6周期性地强制切换以处理长时间计算而不睡眠的进程。这种多路复用产生了每个进程都有自己的CPU的错觉，就像xv6使用内存分配器和硬件页表来产生每个进程都有自己内存的错觉一样。

实现多路复用带来了一些挑战。首先，如何从一个进程切换到另一个进程？尽管上下文切换的思想很简单，但它的实现是xv6中最不透明的代码之一。第二，如何以对用户进程透明的方式强制切换？Xv6使用标准技术，通过定时器中断驱动上下文切换。第三，许多CPU可能同时在进程之间切换，使用一个用锁方案来避免争用是很有必要的。第四，进程退出时必须释放进程的内存以及其他资源，但它不能自己完成所有这一切，因为（例如）它不能在仍然使用自己内核栈的情况下释放它。第五，多核机器的每个核心必须记住它正在执行哪个进程，以便系统调用正确影响对应进程的内核状态。最后，sleep 允许一个进程放弃CPU，wakeup 允许另一个进程唤醒第一个进程。需要小心避免导致唤醒通知丢失的竞争。Xv6试图尽可能简单地解决这些问题，但结果代码很复杂。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 13:55:56

- 7.2 代码：上下文切换

7.2 代码：上下文切换

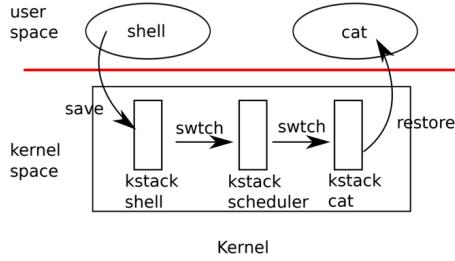


Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

图7.1概述了从一个用户进程（旧进程）切换到另一个用户进程（新进程）所涉及的步骤：一个到旧进程内核线程的用户-内核转换（系统调用或中断），一个到当前CPU调度程序线程的上下文切换，一个到新进程内核线程的上下文切换，以及一个返回到用户级进程的陷阱。调度程序在旧进程的内核栈上执行是不安全的：其他一些核心可能会唤醒进程并运行它，而在两个不同的核心上使用同一个栈将是一场灾难，因此xv6调度程序在每个CPU上都有一个专用线程（保存寄存器和栈）。在本节中，我们将研究在内核线程和调度程序线程之间切换的机制。

从一个线程切换到另一个线程需要保存旧线程的CPU寄存器，并恢复新线程先前保存的寄存器；栈指针和程序计数器被保存和恢复的事实意味着CPU将切换栈和执行中的代码。

函数 `swtch` 为内核线程切换执行保存和恢复操作。`swtch` 对线程没有直接的了解；它只是保存和恢复寄存器集，称为上下文（contexts）。当某个进程要放弃CPU时，该进程的内核线程调用 `swtch` 来保存自己的上下文并返回到调度程序的上下文。每个上下文都包含在一个 `struct context` (*kernel/proc.h:2*) 中，这个结构体本身包含在一个进程的 `struct proc` 或一个CPU的 `struct cpu` 中。`Swtch` 接受两个参数：`struct context *old` 和 `struct context *new`。它将当前寄存器保存在 `old` 中，从 `new` 中加载寄存器，然后返回。

让我们跟随一个进程通过 `swtch` 进入调度程序。我们在第4章中看到，中断结束时的一种可能性是 `usertrap` 调用了 `yield`。依次地：`yield` 调用 `sched`，`sched` 调用 `swtch` 将当前上下文保存在 `p->context` 中，并切换到先前保存在 `cpu->scheduler` (*kernel/proc.c:517*) 中的调度程序上下文。

注：当前版本的XV6中调度程序上下文是 `cpu->context`

`Swtch` (*kernel/swtch.S:3*) 只保存被调用方保存的寄存器（*callee-saved registers*）；调用方保存的寄存器（*caller-saved registers*）通过调用C代码保存在栈上（如果需要）。`Swtch` 知道 `struct context` 中每个寄存器字段的偏移量。它不保存程序计数器。但 `swtch` 保存 `ra` 寄存器，该寄存器保存调用 `swtch` 的返回地址。现在，`swtch` 从新进程的上下文中恢复寄存器，该上下文保存前一个 `swtch` 保存的寄存器值。当 `swtch` 返回时，它返回到由 `ra` 寄存器指定的指令，即新线程以前调用 `swtch` 的指令。另外，它在新线程的栈上返回。

注：关于 callee-saved registers 和 caller-saved registers 请回看视频课程 LEC5 以及文档《Calling Convention》

[!NOTE] 这里不太容易理解，这里举个课程视频中的例子：

以 `cc` 切换到 `ls` 为例，且 `ls` 此前运行过

1. XV6 将 `cc` 程序的内核线程的内核寄存器保存在一个 `context` 对象中
2. 因为要切换到 `ls` 程序的内核线程，那么 `ls` 程序现在的状态必然 是 `RUNABLE`，表明 `ls` 程序之前运行了一半。这同时也意味着：
 - a. `ls` 程序的用户空间状态已经保存在了对应的 `trapframe` 中
 - b. `ls` 程序的内核线程对应的内核寄存器已经保存在对应的 `context` 对象中
- 所以接下来，XV6 会恢复 `ls` 程序的内核线程的 `context` 对象，也就是恢复内核线程的寄存器。
3. 之后 `ls` 会继续在它的内核线程栈上，完成它的中断处理程序
4. 恢复 `ls` 程序的 `trapframe` 中的用户进程状态，返回到用户空间的 `ls` 程序中
5. 最后恢复执行 `ls`

在我们的示例中，`sched` 调用 `swtch` 切换到 `cpu->scheduler`，即每个 CPU 的调度程序上下文。调度程序上下文之前通过 `scheduler` 对 `swtch` (*kernel/proc.c:475*) 的调用进行了保存。当我们追踪 `swtch` 到返回时，他返回到 `scheduler` 而不是 `sched`，并且它的栈指针指向当前 CPU 的调用程序栈（`scheduler stack`）。

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间：2021-08-19 14:01:56

- 7.3 代码：调度

7.3 代码：调度

上一节介绍了 `swtch` 的底层细节；现在，让我们以 `swtch` 为给定对象，检查从一个进程的内核线程通过调度程序切换到另一个进程的情况。调度器（`scheduler`）以每个CPU上一个特殊线程的形式存在，每个线程都运行 `scheduler` 函数。此函数负责选择下一个要运行的进程。想要放弃CPU的进程必须先获得自己的进程锁 `p->lock`，并释放它持有的任何其他锁，更新自己的状态（`p->state`），然后调用 `sched`。`Yield` (*kernel/proc.c:515*) 遵循这个约定，`sleep` 和 `exit` 也遵循这个约定，我们将在后面进行研究。`Sched` 对这些条件再次进行检查（*kernel/proc.c:499-504*），并检查这些条件的隐含条件：由于锁被持有，中断应该被禁用。最后，`sched` 调用 `swtch` 将当前上下文保存在 `p->context` 中，并切换到 `cpu->scheduler` 中的调度程序上下文。`Swtch` 在调度程序的栈上返回，就像是 `scheduler` 的 `swtch` 返回一样。`scheduler` 继续 `for` 循环，找到要运行的进程，切换到该进程，重复循环。

我们刚刚看到，`xv6` 在对 `swtch` 的调用中持有 `p->lock`：`swtch` 的调用者必须已经持有了锁，并且锁的控制权传递给切换到的代码。这种约定在锁上是不寻常的；通常，获取锁的线程还负责释放锁，这使得对正确性进行推理更加容易。对于上下文切换，有必要打破这个惯例，因为 `p->lock` 保护进程 `state` 和 `context` 字段上的不变量，而这些不变量在 `swtch` 中执行时不成立。如果在 `swtch` 期间没有保持 `p->lock`，可能会出现一个问题：在 `yield` 将其状态设置为 `RUNNABLE` 之后，但在 `swtch` 使其停止使用自己的内核栈之前，另一个CPU可能会决定运行该进程。结果将是两个CPU在同一栈上运行，这不可能是正确的。

内核线程总是在 `sched` 中放弃其CPU，并总是切换到调度程序中的同一位置，而调度程序（几乎）总是切换到以前调用 `sched` 的某个内核线程。因此，如果要打印 `xv6` 切换线程处的行号，将观察到以下简单模式：（*kernel/proc.c:475*），（*kernel/proc.c:509*），（*kernel/proc.c:475*），（*kernel/proc.c:509*）等等。在两个线程之间进行这种样式化切换的过程有时被称为协程（`coroutines`）；在本例中，`sched` 和 `scheduler` 是彼此的协同程序。

存在一种情况使得调度程序对 `swtch` 的调用没有以 `sched` 结束。一个新进程第一次被调度时，它从 `forkret` (*kernel/proc.c:527*) 开始。`Forkret` 存在以释放 `p->lock`；否则，新进程可以从 `usertrapret` 开始。

`scheduler` (*kernel/proc.c:457*) 运行一个简单的循环：找到要运行的进程，运行它直到它让步，然后重复循环。`scheduler` 在进程表上循环查找可运行的进程，该进程具有 `p->state == RUNNABLE`。一旦找到一个进程，它将设置CPU当前进程变量 `c->proc`，将该进程标记为 `RUNNING`，然后调用 `swtch` 开始运行它（*kernel/proc.c:470-475*）。

考虑调度代码结构的一种方法是，它为每个进程强制维持一个不变量的集合，并在这些不变量不成立时持有 `p->lock`。其中一个不变量是：如果进程是 `RUNNING` 状态，计时器中断的 `yield` 必须能够安全地从进程中切换出去；这意味着CPU寄存器必须保存进程的寄存器值（即 `swtch` 没有将它们移动到 `context` 中），并且 `c->proc` 必须指向进程。另一个不变量是：如果进程是 `RUNNABLE` 状态，空闲CPU的调

度程序必须安全地运行它；这意味着 `p->context` 必须保存进程的寄存器（即，它们实际上不在实际寄存器中），没有CPU在进程的内核栈上执行，并且没有CPU的 `c->proc` 引用进程。请注意，在保持 `p->lock` 时，这些属性通常不成立。

维护上述不变量是xv6经常在一个线程中获取 `p->lock` 并在另一个线程中释放它的原因，例如在 `yield` 中获取并在 `scheduler` 中释放。一旦 `yield` 开始修改一个 `RUNNING` 进程的状态为 `RUNNABLE`，锁必须保持被持有状态，直到不变量恢复：最早的正确释放点是 `scheduler`（在其自身栈上运行）清除 `c->proc` 之后。类似地，一旦 `scheduler` 开始将 `RUNNABLE` 进程转换为 `RUNNING`，在内核线程完全运行之前（在 `swtch` 之后，例如在 `yield` 中）绝不能释放锁。

`p->lock` 还保护其他东西：`exit` 和 `wait` 之间的相互作用，避免丢失 `wakeup` 的机制（参见第7.5节），以及避免一个进程退出和其他进程读写其状态之间的争用（例如，`exit` 系统调用查看 `p->pid` 并设置 `p->killed` (*kernel/proc.c:611*)）。为了清晰起见，也许为了性能起见，有必要考虑一下 `p->lock` 的不同功能是否可以拆分。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:01:40

- 7.4 代码: mycpu和myproc

7.4 代码: mycpu和myproc

Xv6通常需要指向当前进程的 `proc` 结构体的指针。在单处理器系统上，可以有一个指向当前 `proc` 的全局变量。但这不能用于多核系统，因为每个核执行的进程不同。解决这个问题的方法是基于每个核心都有自己的寄存器集，从而使用其中一个寄存器来帮助查找每个核心的信息。

Xv6为每个CPU维护一个 `struct cpu`，它记录当前在该CPU上运行的进程（如果有的话），为CPU的调度线程保存寄存器，以及管理中断禁用所需的嵌套自旋锁的计数。函数 `mycpu` (*kernel/proc.c:60*)返回一个指向当前CPU的 `struct cpu` 的指针。RISC-V给它的CPU编号，给每个CPU一个 `hartid`。Xv6确保每个CPU的 `hartid` 在内核中存储在该CPU的 `tp` 寄存器中。这允许 `mycpu` 使用 `tp` 对一个`cpu`结构体数组（即 `cpus` 数组，*kernel/proc.c:9*）进行索引，以找到正确的那个。

确保CPU的 `tp` 始终保存CPU的 `hartid` 有点麻烦。`mstart` 在CPU启动次序的早期设置 `tp` 寄存器，此时仍处于机器模式（*kernel/start.c:46*）。因为用户进程可能会修改 `tp`，`usertrapret` 在蹦床页面（*trampoline page*）中保存 `tp`。最后，`uservec` 在从用户空间（*kernel/trampoline.S:70*）进入内核时恢复保存的 `tp`。编译器保证永远不会使用 `tp` 寄存器。如果RISC-V允许xv6 直接读取当前 `hartid`会更方便，但这只允许在机器模式下，而不允许在管理模式下。

`cpuid` 和 `mycpu` 的返回值很脆弱：如果定时器中断并导致线程让步（`yield`），然后移动到另一个CPU，以前返回的值将不再正确。为了避免这个问题，xv6要求调用者禁用中断，并且只有在使用完返回的 `struct cpu` 后才重新启用。

函数 `myproc` (*kernel/proc.c:68*)返回当前CPU上运行进程 `struct proc` 的指针。`myproc` 禁用中断，调用 `mycpu`，从 `struct cpu` 中取出当前进程指针（`c->proc`），然后启用中断。即使启用中断，`myproc` 的返回值也可以安全使用：如果计时器中断将调用进程移动到另一个CPU，其 `struct proc` 指针不会改变。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:02:20

- 7.5 sleep与wakeup

7.5 sleep与wakeup

调度和锁有助于隐藏一个进程对另一个进程的存在，但到目前为止，我们还没有帮助进程进行有意交互的抽象。为解决这个问题已经发明了许多机制。Xv6使用了一种称为 `sleep` 和 `wakeup` 的方法，它允许一个进程在等待事件时休眠，而另一个进程在事件发生后将其唤醒。睡眠和唤醒通常被称为序列协调（**sequence coordination**）或条件同步机制（**conditional synchronization mechanisms**）。

为了说明，让我们考虑一个称为信号量（**semaphore**）的同步机制，它可以协调生产者和消费者。信号量维护一个计数并提供两个操作。“V”操作（对于生产者）增加计数。“P”操作（对于使用者）等待计数为非零，然后递减并返回。如果只有一个生产者线程和一个消费者线程，并且它们在不同的CPU上执行，并且编译器没有进行过积极的优化，那么此实现将是正确的：

```
struct semaphore {
    struct spinlock lock;
    int count;
};

void V(struct semaphore* s) {
    acquire(&s->lock);
    s->count += 1;
    release(&s->lock);
}

void P(struct semaphore* s) {
    while (s->count == 0)
        ;
    acquire(&s->lock);
    s->count -= 1;
    release(&s->lock);
}
```

上面的实现代价昂贵。如果生产者很少采取行动，消费者将把大部分时间花在 `while` 循环中，希望得到非零计数。消费者的CPU可以找到比通过反复轮询 `s->count` 繁忙等待更有成效的工作。要避免繁忙等待，消费者需要一种方法来释放CPU，并且只有在 `v` 增加计数后才能恢复。

这是朝着这个方向迈出的一步，尽管我们将看到这是不够的。让我们想象一对调用，`sleep` 和 `wakeup`，工作流程如下。`sleep(chan)` 在任意值 `chan` 上睡眠，称为等待通道（**wait channel**）。`sleep` 将调用进程置于睡眠状态，释放CPU用于其他工作。`wakeup(chan)` 唤醒所有在 `chan` 上睡眠的进程（如果有），使其 `sleep` 调用返回。如果没有进程在 `chan` 上等待，则 `wakeup` 不执行任何操作。我们可以将信号量实现更改为使用 `sleep` 和 `wakeup`（更改的行添加了注释）：

```

void V(struct semaphore* s) {
    acquire(&s->lock);
    s->count += 1;
    wakeup(s); // !pay attention
    release(&s->lock);
}

void P(struct semaphore* s) {
    while (s->count == 0)
        sleep(s); // !pay attention
    acquire(&s->lock);
    s->count -= 1;
    release(&s->lock);
}

```

`P` 现在放弃CPU而不是自旋，这很好。然而，事实证明，使用此接口设计 `sleep` 和 `wakeup` 而不遭受所谓的丢失唤醒（*lost wake-up*）问题并非易事。假设 `P` 在第9行发现 `s->count==0`。当 `P` 在第9行和第10行之间时，`V` 在另一个CPU上运行：它将 `s->count` 更改为非零，并调用 `wakeup`，这样就不会发现进程处于休眠状态，因此不会执行任何操作。现在 `P` 继续在第10行执行：它调用 `sleep` 并进入睡眠。这会导致一个问题：`P` 正在休眠，等待调用 `V`，而 `V` 已经被调用。除非我们运气好，生产者再次呼叫 `V`，否则消费者将永远等待，即使 `count` 为非零。

这个问题的根源是 `V` 在错误的时刻运行，违反了 `P` 仅在 `s->count==0` 时才休眠的不变量。保护不变量的一种不正确的方法是将锁的获取（下面以黄色突出显示）移动到 `P` 中，以便其检查 `count` 和调用 `sleep` 是原子的：

```

void V(struct semaphore* s) {
    acquire(&s->lock);
    s->count += 1;
    wakeup(s);
    release(&s->lock);
}

void P(struct semaphore* s) {
    acquire(&s->lock); // !pay attention
    while (s->count == 0)
        sleep(s);
    s->count -= 1;
    release(&s->lock);
}

```

人们可能希望这个版本的 `P` 能够避免丢失唤醒，因为锁阻止 `V` 在第10行和第11行之间执行。它确实这样做了，但它会导致死锁：`P` 在睡眠时持有锁，因此 `V` 将永远阻塞等待锁。

我们将通过更改 `sleep` 的接口来修复前面的方案：调用方必须将条件锁（*condition lock*）传递给 `sleep`，以便在调用进程被标记为 *asleep* 并在睡眠通道上等待后 `sleep` 可以释放锁。如果有一个并发的 `V` 操作，锁将强制它在 `P` 将自己置于睡眠状态前一直等待，因此 `wakeup` 将找到睡眠的消费者并将其唤醒。一旦消费者再次醒来，`sleep` 会在返回前重新获得锁。我们新的正确的 `sleep/wakeup` 方案可用如下（更改以黄色突出显示）：

```
void V(struct semaphore* s) {
    acquire(&s->lock);
    s->count += 1;
    wakeup(s);
    release(&s->lock);
}

void P(struct semaphore* s) {
    acquire(&s->lock);

    while (s->count == 0)
        sleep(s, &s->lock); // !pay attention
    s->count -= 1;
    release(&s->lock);
}
```

P 持有 s->lock 的事实阻止 V 在 P 检查 s->count 和调用 sleep 之间试图唤醒它。
然而请注意，我们需要 sleep 释放 s->lock 并使消费者进程进入睡眠状态的操作是原子的。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:05:07

- 7.6 代码: sleep和wakeup

7.6 代码: sleep和wakeup

让我们看看 `sleep` (**kernel/proc.c:548**) 和 `wakeup` (**kernel/proc.c:582**) 的实现。其基本思想是让 `sleep` 将当前进程标记为 `SLEEPING`，然后调用 `sched` 释放 CPU；`wakeup` 查找在给定等待通道上休眠的进程，并将其标记为 `RUNNABLE`。`sleep` 和 `wakeup` 的调用者可以使用任何相互间方便的数字作为通道。`Xv6` 通常使用等待过程中涉及的内核数据结构的地址。

`sleep` 获得 `p->lock` (**kernel/proc.c:559**)。要进入睡眠的进程现在同时持有 `p->lock` 和 `lk`。在调用者（示例中为 `p`）中持有 `lk` 是必要的：它确保没有其他进程（在示例中指一个运行的 `v`）可以启动 `wakeup(chan)` 调用。既然 `sleep` 持有 `p->lock`，那么释放 `lk` 是安全的：其他进程可能会启动对 `wakeup(chan)` 的调用，但是 `wakeup` 将等待获取 `p->lock`，因此将等待 `sleep` 把进程置于睡眠状态的完成，以防止 `wakeup` 错过 `sleep`。

还有一个小问题：如果 `lk` 和 `p->lock` 是同一个锁，那么如果 `sleep` 试图获取 `p->lock` 就会自身死锁。但是，如果调用 `sleep` 的进程已经持有 `p->lock`，那么它不需要做更多的事情来避免错过并发的 `wakeup`。当 `wait` (**kernel/proc.c:582**) 持有 `p->lock` 调用 `sleep` 时，就会出现这种情况。

由于 `sleep` 只持有 `p->lock` 而无其他，它可以通过记录睡眠通道、将进程状态更改为 `SLEEPING` 并调用 `sched` (**kernel/proc.c:564-567**) 将进程置于睡眠状态。过一会儿，我们就会明白为什么在进程被标记为 `SLEEPING` 之前不将 `p->lock` 释放（由 `scheduler`）是至关重要的。

在某个时刻，一个进程将获取条件锁，设置睡眠者正在等待的条件，并调用 `wakeup(chan)`。在持有状态锁时调用 `wakeup` 非常重要[注]。`wakeup` 遍历进程表 (**kernel/proc.c:582**)。它获取它所检查的每个进程的 `p->lock`，这既是因为它可能会操纵该进程的状态，也是因为 `p->lock` 确保 `sleep` 和 `wakeup` 不会彼此错过。当 `wakeup` 发现一个 `SLEEPING` 的进程且 `chan` 相匹配时，它会将该进程的状态更改为 `RUNNABLE`。调度器下次运行时，将看到进程已准备运行。

注：严格地说，`wakeup` 只需跟在 `acquire` 之后就足够了（也就是说，可以在 `release` 之后调用 `wakeup`）

为什么 `sleep` 和 `wakeup` 的用锁规则能确保睡眠进程不会错过唤醒？休眠进程从检查条件之前的某处到标记为休眠之后的某处，要么持有条件锁，要么持有其自身的 `p->lock` 或同时持有两者。调用 `wakeup` 的进程在 `wakeup` 的循环中同时持有这两个锁。因此，要么唤醒器 (`waker`) 在消费者线程检查条件之前使条件为真；要么唤醒器的 `wakeup` 在睡眠线程标记为 `SLEEPING` 后对其进行严格检查。然后 `wakeup` 将看到睡眠进程并将其唤醒（除非有其他东西首先将其唤醒）。

有时，多个进程在同一个通道上睡眠；例如，多个进程读取同一个管道。一个单独的 `wakeup` 调用就能把他们全部唤醒。其中一个将首先运行并获取与 `sleep` 一同调用的锁，并且（在管道例子中）读取在管道中等待的任何数据。尽管被唤醒，其他进程将发现没有要读取的数据。从他们的角度来看，醒来是“虚假的”，他们必须再次睡眠。因此，在检查条件的循环中总是调用 `sleep`。

如果两次使用 `sleep/wakeup` 时意外选择了相同的通道，则不会造成任何伤害：它们将看到虚假的唤醒，但如上所述的循环将容忍此问题。`sleep/wakeup` 的魅力在于它既轻量级（不需要创建特殊的数据结构来充当睡眠通道），又提供了一层抽象（调用者不需要知道他们正在与哪个特定进程进行交互）。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:05:29

- 7.7 代码: Pipes

7.7 代码: Pipes

使用睡眠和唤醒来同步生产者和消费者的一个更复杂的例子是xv6的管道实现。我们在第1章中看到了管道接口：写入管道一端的字节被复制到内核缓冲区，然后可以从管道的另一端读取。以后的章节将研究围绕管道的文件描述符支持，但现在让我们看看 `pipewrite` 和 `piperead` 的实现。

每个管道都由一个 `struct pipe` 表示，其中包含一个锁 `lock` 和一个数据缓冲区 `data`。字段 `nread` 和 `nwrite` 统计从缓冲区读取和写入缓冲区的总字节数。缓冲区是环形的：在 `buf[PIPESIZE-1]` 之后写入的下一个字节是 `buf[0]`。而计数不是环形。此约定允许实现区分完整缓冲区（`nwrite==nread+PIPESIZE`）和空缓冲区（`nwrite==nread`），但这意味着对缓冲区的索引必须使用 `buf[nread%PIPESIZE]`，而不仅仅是 `buf[nread]`（对于 `nwrite` 也是如此）。

让我们假设对 `piperead` 和 `pipewrite` 的调用同时发生在两个不同的CPU上。`Pipewrite` (*kernel/pipe.c:77*) 从获取管道锁开始，它保护计数、数据及其相关不变量。`Piperead` (*kernel/pipe.c:103*) 然后也尝试获取锁，但无法实现。它在 `acquire` (*kernel/spinlock.c:22*) 中旋转等待锁。当 `piperead` 等待时，`pipewrite` 遍历被写入的字节 (`addr[0..n-1]`)，依次将每个字节添加到管道中 (*kernel/pipe.c:95*)。在这个循环中缓冲区可能会被填满 (*kernel/pipe.c:85*)。在这种情况下，`pipewrite` 调用 `wakeup` 来提醒所有处于睡眠状态的读进程缓冲区中有数据等待，然后在 `&pi->nwrite` 上睡眠，等待读进程从缓冲区中取出一些字节。作为使 `pipewrite` 进程进入睡眠状态的一部分，`Sleep` 释放 `pi->lock`。

现在 `pi->lock` 可用，`piperead` 设法获取它并进入其临界区域：它发现 `pi->nread != pi->nwrite` (*kernel/pipe.c:110*)（`pipewrite` 进入睡眠状态是因为 `pi->nwrite == pi->nread+PIPESIZE` (*kernel/pipe.c:85*)），因此它进入 `for` 循环，从管道中复制数据 (*kernel/pipe.c:117*)，并根据复制的字节数增加 `nread`。那些读出的字节就可供写入，因此 `piperead` 调用 `wakeup` (*kernel/pipe.c:124*) 返回之前唤醒所有休眠的写进程。`Wakeup` 寻找一个在 `&pi->nwrite` 上休眠的进程，该进程正在运行 `pipewrite`，但在缓冲区填满时停止。它将该进程标记为 `RUNNABLE`。

管道代码为读者和写者使用单独的睡眠通道 (`pi->nread` 和 `pi->nwrite`)；这可能会使系统在有许多读者和写者等待同一管道这种不太可能的情况下更加高效。管道代码在检查休眠条件的循环中休眠；如果有多个读者或写者，那么除了第一个醒来的进程之外，所有进程都会看到条件仍然错误，并再次睡眠。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间：2021-08-19 14:05:42

- 7.8 代码: `wait`, `exit`和`kill`

7.8 代码: `wait`, `exit`和`kill`

`Sleep` 和 `wakeup` 可用于多种等待。第一章介绍的一个有趣的例子是子进程 `exit` 和父进程 `wait` 之间的交互。在子进程死亡时，父进程可能已经在 `wait` 中休眠，或者正在做其他事情；在后一种情况下，随后的 `wait` 调用必须观察到子进程的死亡，可能是在子进程调用 `exit` 后很久。`xv6` 记录子进程终止直到 `wait` 观察到它的方式是让 `exit` 将调用方置于 `ZOMBIE` 状态，在那里它一直保持到父进程的 `wait` 注意到它，将子进程的状态更改为 `UNUSED`，复制子进程的 `exit` 状态码，并将子进程 ID 返回给父进程。如果父进程在子进程之前退出，则父进程将子进程交给 `init` 进程，`init` 进程将永久调用 `wait`；因此，每个子进程退出后都有一个父进程进行清理。主要的实现挑战是父级和子级 `wait` 和 `exit`，以及 `exit` 和 `exit` 之间可能存在竞争和死锁。

`Wait` 使用调用进程的 `p->lock` 作为条件锁，以避免丢失唤醒，并在开始时获取该锁（[kernel/proc.c:398](#)）。然后它扫描进程表。如果它发现一个子进程处于 `ZOMBIE` 状态，它将释放该子进程的资源及其 `proc` 结构体，将该子进程的退出状态码复制到提供给 `wait` 的地址（如果不是 0），并返回该子进程的进程 ID。如果 `wait` 找到子进程但没有子进程退出，它将调用 `sleep` 以等待其中一个退出（[kernel/proc.c:445](#)），然后再次扫描。这里，`sleep` 中释放的条件锁是等待进程的 `p->lock`，这是上面提到的特例。注意，`wait` 通常持有两个锁：它在试图获得任何子进程的锁之前先获得自己的锁；因此，整个 `xv6` 都必须遵守相同的锁定顺序（父级，然后是子级），以避免死锁。

`Wait` 查看每个进程的 `np->parent` 以查找其子进程。它使用 `np->parent` 而不持有 `np->lock`，这违反了通常的规则，即共享变量必须受到锁的保护。`np` 可能是当前进程的祖先，在这种情况下，获取 `np->lock` 可能会导致死锁，因为这将违反上述顺序。这种情况下无锁检查 `np->parent` 似乎是安全的：进程的 `parent` 字段仅由其父进程更改，因此如果 `np->parent==p` 为 `true`，除非当前流程更改它，否则该值无法被更改，

`Exit`（[kernel/proc.c:333](#)）记录退出状态码，释放一些资源，将所有子进程提供给 `init` 进程，在父进程处于等待状态时唤醒父进程，将调用方标记为僵尸进程（`zombie`），并永久地让出 CPU。最后的顺序有点棘手。退出进程必须在将其状态设置为 `ZOMBIE` 并唤醒父进程时持有其父进程的锁，因为父进程的锁是防止在 `wait` 中丢失唤醒的条件锁。子级还必须持有自己的 `p->lock`，否则父级可能会看到它处于 `ZOMBIE` 状态，并在它仍运行时释放它。锁获取顺序对于避免死锁很重要：因为 `wait` 先获取父锁再获取子锁，所以 `exit` 必须使用相同的顺序。

`Exit` 调用一个专门的唤醒函数 `wakeup1`，该函数仅唤醒父进程，且父进程必须正在 `wait` 中休眠（[kernel/proc.c:598](#)）。在将自身状态设置为 `ZOMBIE` 之前，子进程唤醒父进程可能看起来不正确，但这是安全的：虽然 `wakeup1` 可能会导致父进程运行，但 `wait` 中的循环在 `scheduler` 释放子进程的 `p->lock` 之前无法检查子进程，所以 `wait` 在 `exit` 将其状态设置为 `ZOMBIE`（[kernel/proc.c:386](#)）之前不能查看退出进程。

`exit` 允许进程自行终止，而 `kill`（[kernel/proc.c:611](#)）允许一个进程请求另一个进程终止。对于 `kill` 来说，直接销毁受害者进程（即要杀死的进程）太复杂了，因为受害者可能在另一个 CPU 上执行，也许是在更新内核数据结构的敏感序列中

间。因此，`kill` 的工作量很小：它只是设置受害者的 `p->killed`，如果它正在睡眠，则唤醒它。受害者进程终将进入或离开内核，此时，如果设置了 `p->killed`，`usertrap` 中的代码将调用 `exit`。如果受害者在用户空间中运行，它将很快通过进行系统调用或由于计时器（或其他设备）中断而进入内核。

如果受害者进程在 `sleep` 中，`kill` 对 `wakeup` 的调用将导致受害者从 `sleep` 中返回。这存在潜在的危险，因为等待的条件可能不为真。但是，`xv6` 对 `sleep` 的调用总是封装在 `while` 循环中，该循环在 `sleep` 返回后重新测试条件。一些对 `sleep` 的调用还在循环中测试 `p->killed`，如果它被设置，则放弃当前活动。只有在这种放弃是正确的情况下才能这样做。例如，如果设置了 `killed` 标志，则管道读写代码返回；最终代码将返回到陷阱，陷阱将再次检查标志并退出。

一些 `XV6` 的 `sleep` 循环不检查 `p->killed`，因为代码在应该是原子操作的多步系统调用的中间。`virtio` 驱动程序（`kernel/virtio_disk.c:242`）就是一个例子：它不检查 `p->killed`，因为一个磁盘操作可能是文件系统保持正确状态所需的一组写入操作之一。等待磁盘 I/O 时被杀死的进程将不会退出，直到它完成当前系统调用并且 `usertrap` 看到 `killed` 标志。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:05:52

7.9 真实世界

xv6调度器实现了一个简单的调度策略：它依次运行每个进程。这一策略被称为轮询调度（round robin）。真实的操作系统实施更复杂的策略，例如，允许进程具有优先级。其思想是调度器将优先选择可运行的高优先级进程，而不是可运行的低优先级进程。这些策略可能变得很复杂，因为常常存在相互竞争的目标：例如，操作系统可能希望保证公平性和高吞吐量。此外，复杂的策略可能会导致意外的交互，例如优先级反转（priority inversion）和航队（convoys）。当低优先级进程和高优先级进程共享一个锁时，可能会发生优先级反转，当低优先级进程持有该锁时，可能会阻止高优先级进程前进。当许多高优先级进程正在等待一个获得共享锁的低优先级进程时，可能会形成一个长的等待进程航队；一旦航队形成，它可以持续很长时间。为了避免此类问题，在复杂的调度器中需要额外的机制。

睡眠和唤醒是一种简单有效的同步方法，但还有很多其他方法。所有这些问题中的第一个挑战是避免我们在本章开头看到的“丢失唤醒”问题。原始Unix内核的 `sleep` 只是禁用了中断，这就足够了，因为Unix运行在单CPU系统上。因为xv6在多处理器上运行，所以它为 `sleep` 添加了一个显式锁。FreeBSD的 `msleep` 采用了同样的方法。Plan 9的 `sleep` 使用一个回调函数，该函数在马上睡眠时获取调度锁，并在运行中持有；该函数用于在最后时刻检查睡眠条件，以避免丢失唤醒。Linux内核的 `sleep` 使用一个显式的进程队列，称为等待队列，而不是等待通道；队列有自己内部的锁。

在 `wakeup` 中扫描整个进程列表以查找具有匹配 `chan` 的进程效率低下。一个更好的解决方案是用一个数据结构替换 `sleep` 和 `wakeup` 中的 `chan`，该数据结构包含在该结构上休眠的进程列表，例如Linux的等待队列。Plan 9的 `sleep` 和 `wakeup` 将该结构称为集结点（rendezvous point）或Rendez。许多线程库引用与条件变量相同的结构；在这种情况下，`sleep` 和 `wakeup` 操作称为 `wait` 和 `signal`。所有这些机制都有一个共同的特点：睡眠条件受到某种在睡眠过程中原子级释放的锁的保护。

`wakeup` 的实现会唤醒在特定通道上等待的所有进程，可能有许多进程在等待该特定通道。操作系统将安排所有这些进程，它们将竞相检查睡眠条件。进程的这种行为有时被称为惊群效应（thundering herd），最好避免。大多数条件变量都有两个用于唤醒的原语：`signal` 用于唤醒一个进程；`broadcast` 用于唤醒所有等待进程。

信号量（Semaphores）通常用于同步。计数 `count` 通常对应于管道缓冲区中可用的字节数或进程具有的僵尸子进程数。使用显式计数作为抽象的一部分可以避免“丢失唤醒”问题：使用显式计数记录已经发生 `wakeup` 的次数。计数还避免了虚假唤醒和惊群效应问题。

终止进程并清理它们在xv6中引入了很多复杂性。在大多数操作系统中甚至更复杂，因为，例如，受害者进程可能在内核深处休眠，而展开其栈空间需要非常仔细的编程。许多操作系统使用显式异常处理机制（如 `longjmp`）来展开栈。此外，还有其他事件可能导致睡眠进程被唤醒，即使它等待的事件尚未发生。例如，当一个 Unix进程处于休眠状态时，另一个进程可能会向它发送一个 `signal`。在这种情况下，进程将从中断的系统调用返回，返回值为-1，错误代码设置为 `EINTR`。应用程序可以检查这些值并决定执行什么操作。Xv6不支持信号，因此不会出现这种复杂性。

Xv6对 kill 的支持并不完全令人满意：有一些 sleep 循环可能应该检查 p->killed。一个相关的问题是，即使对于检查 p->killed 的 sleep 循环，sleep 和 kill 之间也存在竞争；后者可能会设置 p->killed，并试图在受害者的循环检查 p->killed 之后但在调用 sleep 之前尝试唤醒受害者。如果出现此问题，受害者将不会注意到 p->killed，直到其等待的条件发生。这可能比正常情况要晚一点（例如，当virtio驱动程序返回受害者正在等待的磁盘块时）或永远不会发生（例如，如果受害者正在等待来自控制台的输入，但用户没有键入任何输入）。

注：上节中说到kill的工作方式，kill 设置 p->killed，如果遇到进程正在休眠，则会唤醒它，此后在 usertrap 中检测 p->killed，并使进程退出

而如果像上面说的，在检查 p->killed 之后调用 sleep 之前唤醒受害者进程，那么接下来执行 sleep 就会导致进程无法进入内核，无法在 usertrap 中退出，而必须等待所需事件的发生再次唤醒

一个实际的操作系统将在固定时间内使用空闲列表找到自由的 proc 结构体，而不是 allocproc 中的线性时间搜索；xv6使用线性扫描是为了简单起见。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间：2021-08-19 14:06:02

- 7.10 练习

7.10 练习

1. `sleep` 必须检查 `lk != &p->lock` 来避免死锁(*kernel/proc.c:558-561*). 假设通过将

```
if(lk != &p->lock) {  
    acquire(&p->lock);  
    release(lk);  
}
```

替换为

```
release(lk);  
acquire(&p->lock);
```

来消除特殊情况，这样做将会破坏 `sleep`。是如何破坏的呢？

1. 大多数进程清理可以通过 `exit` 或 `wait` 来完成。事实证明，必须是 `exit` 作为关闭打开的文件的那个。为什么？答案涉及管道。
2. 在xv6中实现信号量而不使用 `sleep` 和 `wakeup`（但可以使用自旋锁）。用信号量取代xv6中 `sleep` 和 `wakeup` 的使用。判断结果。
3. 修复上面提到的 `kill` 和 `sleep` 之间的竞争，这样在受害者的 `sleep` 循环检查 `p->killed` 之后但在调用 `sleep` 之前发生的 `kill` 会导致受害者放弃当前系统调用。
4. 设计一个计划，使每个睡眠循环检查 `p->killed`，这样，例如，`virtio`驱动程序中的一个进程可以在被另一个进程终止时从 `while` 循环快速返回。
5. 修改xv6，使其在从一个进程的内核线程切换到另一个线程时仅使用一次上下文切换，而不是通过调度器线程进行切换。屈服（`yield`）线程需要选择下一个线程本身并调用 `swtch`。挑战在于：防止多个内核意外执行同一个线程；获得正确的锁；避免死锁。
6. 修改xv6的调度程序，以便在没有进程可运行时使用RISC-V的 `WFI`（`wait for interrupt`，等待中断）指令。尽量确保在任何时候有可运行的进程等待运行时，没有核心在 `WFI` 中暂停。
7. 锁 `p->lock` 保护许多不变量，当查看受 `p->lock` 保护的特定xv6代码段时，可能很难确定保护的是哪个不变量。通过将 `p->lock` 拆分为多个锁，设计一个更清晰的计划。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:06:46

第八章 文件系统

文件系统的目的是组织和存储数据。文件系统通常支持用户和应用程序之间的数据共享，以及持久性，以便在重新启动后数据仍然可用。

xv6文件系统提供类似于Unix的文件、目录和路径名（参见第1章），并将其数据存储在virtio磁盘上以便持久化（参见第4章）。文件系统解决了几个难题：

注：完整计算机中的CPU被支撑硬件包围，其中大部分是以I/O接口的形式。

Xv6是以qemu的“-machine virt”选项模拟的支撑硬件编写的。这包括RAM、包含引导代码的ROM、一个到用户键盘/屏幕的串行连接，以及一个用于存储的磁盘。

- 文件系统需要磁盘上的数据结构来表示目录和文件名称树，记录保存每个文件内容的块的标识，以及记录磁盘的哪些区域是空闲的。
- 文件系统必须支持崩溃恢复（**crash recovery**）。也就是说，如果发生崩溃（例如，电源故障），文件系统必须在重新启动后仍能正常工作。风险在于崩溃可能会中断一系列更新，并使磁盘上的数据结构不一致（例如，一个块在某个文件中使用但同时仍被标记为空闲）。
- 不同的进程可能同时在文件系统上运行，因此文件系统代码必须协调以保持不变量。
- 访问磁盘的速度比访问内存慢几个数量级，因此文件系统必须保持常用块的内存缓存。

本章的其余部分将解释xv6如何应对这些挑战。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 14:07:05

- 8.1 概述

8.1 概述

xv6文件系统实现分为七层，如图8.1所示。磁盘层读取和写入virtio硬盘上的块。缓冲区高速缓存层缓存磁盘块并同步对它们的访问，确保每次只有一个内核进程可以修改存储在任何特定块中的数据。日志记录层允许更高层在一次事务（transaction）中将更新包装到多个块，并确保在遇到崩溃时自动更新这些块（即，所有块都已更新或无更新）。索引结点层提供单独的文件，每个文件表示为一个索引结点，其中包含唯一的索引号（i-number）和一些保存文件数据的块。目录层将每个目录实现为一种特殊的索引结点，其内容是一系列目录项，每个目录项包含一个文件名和索引号。路径名层提供了分层路径名，如`/usr/rtm/xv6/fs.c`，并通过递归查找来解析它们。文件描述符层使用文件系统接口抽象了许多Unix资源（例如，管道、设备、文件等），简化了应用程员的工作。



图8.1 XV6文件系统的层级

文件系统必须有将索引节点和内容块存储在磁盘上哪些位置的方案。为此，xv6将磁盘划分为几个部分，如图8.2所示。文件系统不使用块0（它保存引导扇区）。块1称为超级块：它包含有关文件系统的元数据（文件系统大小（以块为单位）、数据块数、索引节点数和日志中的块数）。从2开始的块保存日志。日志之后是索引节点，每个块有多个索引节点。然后是位图块，跟踪正在使用的数据块。其余的块是数据块：每个都要么在位图块中标记为空闲，要么保存文件或目录的内容。超级块由一个名为 `mkfs` 的单独的程序填充，该程序构建初始文件系统。

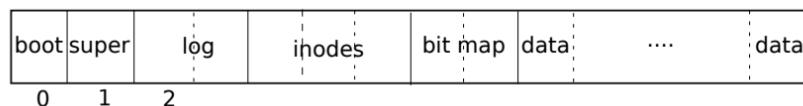


Figure 8.2: Structure of the xv6 file system.

本章的其余部分将从缓冲区高速缓存层开始讨论每一层。注意那些在较低层次上精心选择的抽象可以简化较高层次的设计的情况。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:08:19

- 8.2 Buffer cache层

8.2 Buffer cache层

Buffer cache有两个任务：

1. 同步对磁盘块的访问，以确保磁盘块在内存中只有一个副本，并且一次只有一个内核线程使用该副本
2. 缓存常用块，以便不需要从慢速磁盘重新读取它们。代码在**bio.c**中。

Buffer cache层导出的主接口主要是 `bread` 和 `bwrite`；前者获取一个 `buf`，其中包含一个可以在内存中读取或修改的块的副本，后者将修改后的缓冲区写入磁盘上的相应块。内核线程必须通过调用 `brelease` 释放缓冲区。Buffer cache每个缓冲区使用一个睡眠锁，以确保每个缓冲区（因此也是每个磁盘块）每次只被一个线程使用；`bread` 返回一个上锁的缓冲区，`brelease` 释放该锁。

让我们回到Buffer cache。Buffer cache中保存磁盘块的缓冲区数量固定，这意味着如果文件系统请求还未存放在缓存中的块，Buffer cache必须回收当前保存其他块内容的缓冲区。Buffer cache为新块回收最近使用最少的缓冲区。这样做的原因是认为最近使用最少的缓冲区是最不可能近期再次使用的缓冲区。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:08:31

- 8.3 代码: Buffer cache

8.3 代码: Buffer cache

Buffer cache是以双链表表示的缓冲区。`main` (*kernel/main.c:27*) 调用的函数 `binit` 使用静态数组 `buf` (*kernel/bio.c:43-52*) 中的 `NBUF` 个缓冲区初始化列表。对Buffer cache的所有其他访问都通过 `bcache.head` 引用链表，而不是 `buf` 数组。

缓冲区有两个与之关联的状态字段。字段 `valid` 表示缓冲区是否包含块的副本。字段 `disk` 表示缓冲区内容是否已交给磁盘，这可能会更改缓冲区（例如，将数据从磁盘写入 `data`）。

`Bread` (*kernel/bio.c:93*) 调用 `bget` 为给定扇区 (*kernel/bio.c:97*) 获取缓冲区。如果缓冲区需要从磁盘进行读取，`bread` 会在返回缓冲区之前调用 `virtio_disk_rw` 来执行此操作。

`Bget` (*kernel/bio.c:59*) 扫描缓冲区列表，查找具有给定设备和扇区号 (*kernel/bio.c:65-73*) 的缓冲区。如果存在这样的缓冲区，`bget` 将获取缓冲区的睡眠锁。然后 `Bget` 返回锁定的缓冲区。

如果对于给定的扇区没有缓冲区，`bget` 必须创建一个，这可能会重用包含其他扇区的缓冲区。它再次扫描缓冲区列表，查找未在使用中的缓冲区 (`b->refcnt = 0`)：任何这样的缓冲区都可以使用。`Bget` 编辑缓冲区元数据以记录新设备和扇区号，并获取其睡眠锁。注意，`b->valid = 0` 的布置确保了 `bread` 将从磁盘读取块数据，而不是错误地使用缓冲区以前的内容。

每个磁盘扇区最多有一个缓存缓冲区是非常重要的，并且因为文件系统使用缓冲区上的锁进行同步，可以确保读者看到写操作。`Bget` 的从第一个检查块是否缓存的循环到第二个声明块现在已缓存（通过设置 `dev`、`blockno` 和 `refcnt`）的循环，一直持有 `bcache.lock` 来确保此不变量。这会导致检查块是否存在以及（如果不存）指定一个缓冲区来存储块具有原子性。

`bget` 在 `bcache.lock` 临界区域之外获取缓冲区的睡眠锁是安全的，因为非零 `b->refcnt` 防止缓冲区被重新用于不同的磁盘块。睡眠锁保护块缓冲内容的读写，而 `bcache.lock` 保护有关缓存哪些块的信息。

如果所有缓冲区都处于忙碌，那么太多进程同时执行文件系统调用：`bget` 将会 `panic`。一个更优雅的响应可能是在缓冲区空闲之前休眠，尽管这样可能会出现死锁。

一旦 `bread` 读取了磁盘（如果需要）并将缓冲区返回给其调用者，调用者就可以独占使用缓冲区，并可以读取或写入数据字节。如果调用者确实修改了缓冲区，则必须在释放缓冲区之前调用 `bwrite` 将更改的数据写入磁盘。`Bwrite` (*kernel/bio.c:107*) 调用 `virtio_disk_rw` 与磁盘硬件对话。

当调用方使用完缓冲区后，它必须调用 `brelse` 来释放缓冲区(`brelse` 是 `b-release` 的缩写，这个名字很隐晦，但值得学习：它起源于 Unix，也用于 BSD、Linux 和 Solaris)。`brelse` (*kernel/bio.c:117*) 释放睡眠锁并将缓冲区移动到链表的前面 (*kernel/bio.c:128-133*)。移动缓冲区会使列表按缓冲区的使用频率排序（意思是释放）：列表中的第一个缓冲区是最近使用的，最后一个是最使用最少的。`bget` 中的两个循环利用了这一点：在最坏的情况下，对现有缓冲区的扫描必

须处理整个列表，但首先检查最新使用的缓冲区（从 `bcache.head` 开始，然后是下一个指针），在引用局部性良好的情况下将减少扫描时间。选择要重用的缓冲区时，通过自后向前扫描（跟随 `prev` 指针）选择最近使用最少的缓冲区。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:08:40

- 8.4 日志层

8.4 日志层

文件系统设计中最有趣的问题之一是崩溃恢复。出现此问题的原因是，许多文件系统操作都涉及到对磁盘的多次写入，并且在完成写操作的部分子集后崩溃可能会使磁盘上的文件系统处于不一致的状态。例如，假设在文件截断（将文件长度设置为零并释放其内容块）期间发生崩溃。根据磁盘写入的顺序，崩溃可能会留下对标记为空闲的内容块的引用的inode，也可能留下已分配但未引用的内容块。

后者相对来说是良性的，但引用已释放块的inode在重新启动后可能会导致严重问题。重新启动后，内核可能会将该块分配给另一个文件，现在我们有两个不同的文件无意中指向同一块。如果xv6支持多个用户，这种情况可能是一个安全问题，因为旧文件的所有者将能够读取和写入新文件中的块，而新文件的所有者是另一个用户。

Xv6通过简单的日志记录形式解决了文件系统操作期间的崩溃问题。xv6系统调用不会直接写入磁盘上的文件系统数据结构。相反，它会在磁盘上的*log*（日志）中放置它希望进行的所有磁盘写入的描述。一旦系统调用记录了它的所有写入操作，它就会向磁盘写入一条特殊的*commit*（提交）记录，表明日志包含一个完整的操作。此时，系统调用将写操作复制到磁盘上的文件系统数据结构。完成这些写入后，系统调用将擦除磁盘上的日志。

如果系统崩溃并重新启动，则在运行任何进程之前，文件系统代码将按如下方式从崩溃中恢复。如果日志标记为包含完整操作，则恢复代码会将写操作复制到磁盘文件系统中它们所属的位置。如果日志没有标记为包含完整操作，则恢复代码将忽略该日志。恢复代码通过擦除日志完成。

为什么xv6的日志解决了文件系统操作期间的崩溃问题？如果崩溃发生在操作提交之前，那么磁盘上的登录将不会被标记为已完成，恢复代码将忽略它，并且磁盘的状态将如同操作尚未启动一样。如果崩溃发生在操作提交之后，则恢复将重播操作的所有写入操作，如果操作已经开始将它们写入磁盘数据结构，则可能会重复这些操作。在任何一种情况下，日志都会使操作在崩溃时成为原子操作：恢复后，要么操作的所有写入都显示在磁盘上，要都不显示。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:08:51

- 8.5 日志设计

8.5 日志设计

日志驻留在超级块中指定的已知固定位置。它由一个头块（**header block**）和一系列更新块的副本（**logged block**）组成。头块包含一个扇区号数组（每个**logged block**对应一个扇区号）以及日志块的计数。磁盘上的头块中的计数或者为零，表示日志中没有事务；或者为非零，表示日志包含一个完整的已提交事务，并具有指定数量的**logged block**。在事务提交（**commit**）时Xv6才向头块写入数据，在此之前不会写入，并在将**logged blocks**复制到文件系统后将计数设置为零。因此，事务中途崩溃将导致日志头块中的计数为零；提交后的崩溃将导致非零计数。

注：**logged block**表示已经记录了操作信息的日志块，而**log block**仅表示日志块

每个系统调用的代码都指示写入序列的起止，考虑到崩溃，写入序列必须具有原子性。为了允许不同进程并发执行文件系统操作，日志系统可以将多个系统调用的写入累积到一个事务中。因此，单个提交可能涉及多个完整系统调用的写入。为了避免在事务之间拆分系统调用，日志系统仅在没有文件系统调用进行时提交。

同时提交多个事务的想法称为组提交（**group commit**）。组提交减少了磁盘操作的数量，因为成本固定的一次提交分摊了多个操作。组提交还同时为磁盘系统提供更多并发写操作，可能允许磁盘在一个磁盘旋转时间内写入所有这些操作。Xv6的**virtio**驱动程序不支持这种批处理，但是Xv6的文件系统设计允许这样做。

Xv6在磁盘上留出固定的空间来保存日志。事务中系统调用写入的块总数必须可容纳于该空间。这导致两个后果：任何单个系统调用都不允许写入超过日志空间的不同块。这对于大多数系统调用来说都不是问题，但其中两个可能会写入许多块：**write** 和 **unlink**。一个大文件的 **write** 可以写入多个数据块和多个位图块以及一个**inode**块；**unlink** 大文件可能会写入许多位图块和**inode**。Xv6的 **write** 系统调用将大的写入分解为适合日志的多个较小的写入，**unlink** 不会导致此问题，因为实际上Xv6文件系统只使用一个位图块。日志空间有限的另一个后果是，除非确定系统调用的写入将可容纳于日志中剩余的空间，否则日志系统无法允许启动系统调用。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:09:01

- 8.6 代码：日志

8.6 代码：日志

在系统调用中一个典型的日志使用就像这样：

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

`begin_op` (*kernel/log.c:126*) 等待直到日志系统当前未处于提交中，并且直到有足够的未被占用的日志空间来保存此调用的写入。`log.outstanding` 统计预定了日志空间的系统调用数；为此保留的总空间为 `log.outstanding` 乘以 `MAXOPBLOCKS`。递增 `log.outstanding` 会预定空间并防止在此系统调用期间发生提交。代码保守地假设每个系统调用最多可以写入 `MAXOPBLOCKS` 个不同的块。

`log_write` (*kernel/log.c:214*) 充当 `bwrite` 的代理。它将块的扇区号记录在内存中，在磁盘上的日志中预定一个槽位，并调用 `bpin` 将缓存固定在 `block cache` 中，以防止 `block cache` 将其逐出。

注：固定在 `block cache` 是指在缓存不足需要考虑替换时，不会将这个 `block` 换出，因为事务具有原子性：假设块 45 被写入，将其换出的话需要写入磁盘中文件系统对应的位置，而日志系统要求所有内存必须都存入日志，最后才能写入文件系统。

`bpin` 是通过增加引用计数防止块被换出的，之后需要再调用 `bunpin`

在提交之前，块必须留在缓存中：在提交之前，缓存的副本是修改的唯一记录；只有在提交后才能将其写入磁盘上的位置；同一事务中的其他读取必须看到修改。`log_write` 会注意到在单个事务中多次写入一个块的情况，并在日志中为该块分配相同的槽位。这种优化通常称为合并（**absorption**）。例如，包含多个文件 `inode` 的磁盘块在一个事务中被多次写入是很常见的。通过将多个磁盘写入合并到一个磁盘中，文件系统可以节省日志空间并实现更好的性能，因为只有一个磁盘块副本必须写入磁盘。

注：日志需要写入磁盘，以便重启后读取，但日志头块和日志数据块也会在 `block cache` 中有一个副本

`end_op` (*kernel/log.c:146*) 首先减少未完成系统调用的计数。如果计数现在为零，则通过调用 `commit()` 提交当前事务。这一过程分为四个阶段。`write_log()` (*kernel/log.c:178*) 将事务中修改的每个块从缓冲区缓存复制到磁盘上日志槽位中。`write_head()` (*kernel/log.c:102*) 将头块写入磁盘：这是提交点，写入后的崩溃将导致从日志恢复重演事务的写入操作。`install_trans` (*kernel/log.c:69*) 从日志中读取每个块，并将其写入文件系统中的适当位置。最后，`end_op` 写入计数为零的日志头；这必须在下一个事务开始写入日志块之前发生，以便崩溃不会导致使用一个事务的头块和后续事务的日志块进行恢复。

`recover_from_log` (**kernel/log.c:116**) 是由 `initlog` (**kernel/log.c:55**) 调用的，而它又是在第一个用户进程运行 (**kernel/proc.c:539**) 之前的引导期间由 `fsinit` (**kernel/fs.c:42**) 调用的。它读取日志头，如果头中指示日志包含已提交的事务，则模拟 `end_op` 的操作。

日志的一个示例使用发生在 `filewrite` (**kernel/file.c:135**) 中。事务如下所示：

```
begin_op();
ilock(f->ip);
r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```

这段代码被包装在一个循环中，该循环一次将大的写操作分解为几个扇区的单个事务，以避免日志溢出。作为此事务的一部分，对 `writei` 的调用写入许多块：文件的 `inode`、一个或多个位图块以及一些数据块。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:10:18

- 8.7 代码：块分配器

8.7 代码：块分配器

文件和目录内容存储在磁盘块中，磁盘块必须从空闲池中分配。`xv6`的块分配器在磁盘上维护一个空闲位图，每一位代表一个块。0表示对应的块是空闲的；1表示它正在使用中。程序 `mkfs` 设置对应于引导扇区、超级块、日志块、`inode`块和位图块的比特位。

块分配器提供两个功能：`balloc` 分配一个新的磁盘块，`bfree` 释放一个块。`Balloc` 中位于 `kernel/fs.c:71` 的循环从块0到 `sb.size`（文件系统中的块数）遍历每个块。它查找位图中位为零的空闲块。如果 `balloc` 找到这样一个块，它将更新位图并返回该块。为了提高效率，循环被分成两部分。外部循环读取位图中的每个块。内部循环检查单个位图块中的所有BPB位。由于任何一个位图块在 `buffer cache` 中一次只允许一个进程使用，因此，如果两个进程同时尝试分配一个块，可能会发生争用。

`Bfree`（`kernel/fs.c:90`）找到正确的位图块并清除正确的位。同样，`bread` 和 `brelse` 隐含的独占使用避免了显式锁定的需要。

与本章其余部分描述的大部分代码一样，必须在事务内部调用 `balloc` 和 `bfree`。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间：2021-08-19 14:10:47

- 8.8 索引结点层

8.8 索引结点层

术语`inode`（即索引结点）可以具有两种相关含义之一。它可能是指包含文件大小和数据块编号列表的磁盘上的数据结构。或者“`inode`”可能指内存中的`inode`，它包含磁盘上`inode`的副本以及内核中所需的额外信息。

磁盘上的`inode`都被打包到一个称为`inode`块的连续磁盘区域中。每个`inode`的大小都相同，因此在给定数字n的情况下，很容易在磁盘上找到第n个`inode`。事实上，这个编号n，称为`inode number`或*i-number*，是在具体实现中标识`inode`的方式。

磁盘上的`inode`由 `struct dinode` (*kernel/fs.h:32*) 定义。字段 `type` 区分文件、目录和特殊文件（设备）。`type` 为零表示磁盘`inode`是空闲的。字段 `nlink` 统计引用此`inode`的目录条目数，以便识别何时应释放磁盘上的`inode`及其数据块。字段 `size` 记录文件中内容的字节数。`addr` 数组记录保存文件内容的磁盘块的块号。

内核将活动的`inode`集合保存在内存中；`struct inode` (*kernel/file.h:17*) 是磁盘上 `struct dinode` 的内存副本。只有当有C指针引用某个`inode`时，内核才会在内存中存储该`inode`。`ref` 字段统计引用内存中`inode`的C指针的数量，如果引用计数降至零，内核将从内存中丢弃该`inode`。`iget` 和 `iput` 函数分别获取和释放指向`inode`的指针，修改引用计数。指向`inode`的指针可以来自文件描述符、当前工作目录和如 `exec` 的瞬态内核代码。

`xv6`的`inode`代码中有四种锁或类似锁的机制。`icache.lock` 保护以下两个不变量：`inode`最多在缓存中出现一次；缓存`inode`的 `ref` 字段记录指向缓存`inode`的内存指针数量。每个内存中的`inode`都有一个包含睡眠锁的 `lock` 字段，它确保以独占方式访问`inode`的字段（如文件长度）以及`inode`的文件或目录内容块。如果`inode`的 `ref` 大于零，则会导致系统在`cache`中维护`inode`，而不会对其他`inode`重用此缓存项。最后，每个`inode`都包含一个 `nlink` 字段（在磁盘上，如果已缓存则复制到内存中），该字段统计引用文件的目录项的数量；如果`inode`的链接计数大于零，`xv6`将不会释放`inode`。

`iget()` 返回的 `struct inode` 指针在相应的 `iput()` 调用之前保证有效：`inode`不会被删除，指针引用的内存也不会被其他`inode`重用。`iget()` 提供对`inode`的非独占访问，因此可以有许多指向同一`inode`的指针。文件系统代码的许多部分都依赖于 `iget()` 的这种行为，既可以保存对`inode`的长期引用（如打开的文件和当前目录），也可以防止争用，同时避免操纵多个`inode`（如路径名查找）的代码产生死锁。

`iget` 返回的 `struct inode` 可能没有任何有用的内容。为了确保它保存磁盘`inode`的副本，代码必须调用 `ilock`。这将锁定`inode`（以便没有其他进程可以对其进行 `ilock`），并从磁盘读取尚未读取的`inode`。`iunlock` 释放`inode`上的锁。将`inode`指针的获取与锁定分离有助于在某些情况下避免死锁，例如在目录查找期间。多个进程可以持有指向 `iget` 返回的`inode`的C指针，但一次只能有一个进程锁定`inode`。

`inode`缓存只缓存内核代码或数据结构持有C指针的`inode`。它的主要工作实际上是对多个进程的同步访问；缓存是次要的。如果经常使用`inode`，在`inode`缓存不保留它的情况下`buffer cache`可能会将其保留在内存中。`inode`缓存是直写的，这意味着修改已缓存`inode`的代码必须立即使用 `iupdate` 将其写入磁盘。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 14:11:01

- 8.9 代码: Inodes

8.9 代码: Inodes

为了分配新的inode（例如，在创建文件时），`xv6`调用`ialloc`（**kernel/fs.c:196**）。`ialloc`类似于`balloc`：它一次一个块地遍历磁盘上的索引节点结构体，查找标记为空闲的一个。当它找到一个时，它通过将新`type`写入磁盘来声明它，然后末尾通过调用`iget`（**kernel/fs.c:210**）从inode缓存返回一个条目。`ialloc`的正确操作取决于这样一个事实：一次只有一个进程可以保存对`bp`的引用：`ialloc`可以确保其他进程不会同时看到inode可用并尝试声明它。

`iget`（**kernel/fs.c:243**）在inode缓存中查找具有所需设备和inode编号的活动条目（`ip->ref > 0`）。如果找到一个，它将返回对该inode的新引用（**kernel/fs.c:252-256**）。在`iget`扫描时，它会记录第一个空槽（**kernel/fs.c:257-258**）的位置，如果需要分配缓存项，它会使用这个槽。

在读取或写入inode的元数据或内容之前，代码必须使用`ilock`锁定inode。`ilock`（**kernel/fs.c:289**）为此使用睡眠锁。一旦`ilock`以独占方式访问inode，它将根据需要从磁盘（更可能是buffer cache）读取inode。函数`iunlock`（**kernel/fs.c:317**）释放睡眠锁，这可能会导致任何睡眠进程被唤醒。

`iput`（**kernel/fs.c:333**）通过减少引用计数（**kernel/fs.c:356**）释放指向inode的C指针。如果这是最后一次引用，inode缓存中该inode的槽现在将是空闲的，可以重用于其他inode。

如果`iput`发现没有指向inode的C指针引用，并且inode没有指向它的链接（发生于无目录），则必须释放inode及其数据块。`iput`调用`itrunc`将文件截断为零字节，释放数据块；将索引节点类型设置为0（未分配）；并将inode写入磁盘（**kernel/fs.c:338**）。

`iput`中释放inode的锁定协议值得仔细研究。一个危险是并发线程可能正在`ilock`中等待使用该inode（例如，读取文件或列出目录），并且不会做好该inode已不再被分配的准备。这不可能发生，因为如果缓存的inode没有链接，并且`ip->ref`为1，那么系统调用就无法获取指向该inode的指针。那一个引用是调用`iput`的线程所拥有的引用。的确，`iput`在`icache.lock`的临界区域之外检查引用计数是否为1，但此时已知链接计数为零，因此没有线程会尝试获取新引用。另一个主要危险是，对`ialloc`的并发调用可能会选择`iput`正在释放的同一个inode。这只能在`iupdate`写入磁盘以使inode的`type`为零后发生。这个争用是良性的：分配线程将客气地等待获取inode的睡眠锁，然后再读取或写入inode，此时`iput`已完成。

`iput()`可以写入磁盘。这意味着任何使用文件系统的系统调用都可能写入磁盘，因为系统调用可能是最后一个引用该文件的系统调用。即使像`read()`这样看起来是只读的调用，也可能最终调用`iput()`。这反过来意味着，即使是只读系统调用，如果它们使用文件系统，也必须在事务中进行包装。

`iput()`和崩溃之间存在一种具有挑战性的交互。`iput()`不会在文件的链接计数降至零时立即截断文件，因为某些进程可能仍在内存中保留对inode的引用：进程可能仍在读取和写入该文件，因为它已成功打开该文件。但是，如果在最后一个进程

关闭该文件的文件描述符之前发生崩溃，则该文件将被标记为已在磁盘上分配，但没有目录项指向它。

文件系统以两种方式之一处理这种情况。简单的解决方案用于恢复时：重新启动后，文件系统会扫描整个文件系统，以查找标记为已分配但没有指向它们的目录项的文件。如果存在任何此类文件，接下来可以将其释放。

第二种解决方案不需要扫描文件系统。在此解决方案中，文件系统在磁盘（例如在超级块中）上记录链接计数降至零但引用计数不为零的文件的i-number。如果文件系统在其引用计数达到0时删除该文件，则会通过从列表中删除该inode来更新磁盘列表。恢复时，文件系统将释放列表中的任何文件。

Xv6没有实现这两种解决方案，这意味着inode可能被标记为已在磁盘上分配，即使它们不再使用。这意味着随着时间的推移，xv6可能会面临磁盘空间不足的风险。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:11:13

- 8.10 代码: Inode包含内容

8.10 代码: Inode包含内容

磁盘上的inode结构体 `struct dinode` 包含一个 `size` 和一个块号数组（见图8.3）。`inode`数据可以在 `dinode` 的 `addr`s 数组列出的块中找到。前面的 `NDIRECT` 个数据块被列在数组中的前 `NDIRECT` 个元素中；这些块称为直接块（`direct blocks`）。接下来的 `NINDIRECT` 个数据块不在`inode`中列出，而是在称为间接块（`indirect block`）的数据块中列出。`addr`s 数组中的最后一个元素给出了间接块的地址。因此，可以从`inode`中列出的块加载文件的前 12 KB ($NDIRECT \times BSIZE$) 字节，而只有在查阅间接块后才能加载下一个 256 KB ($NINDIRECT \times BSIZE$) 字节。这是一个很好的磁盘表示，但对于客户端来说较复杂。函数 `bmap` 管理这种表示，以便实现我们将很快看到的如 `readi` 和 `writei` 这样的更高级例程。`bmap(struct inode *ip, uint bn)` 返回索引结点 `ip` 的第 `bn` 个数据块的磁盘块号。如果 `ip` 还没有这样的块，`bmap` 会分配一个。

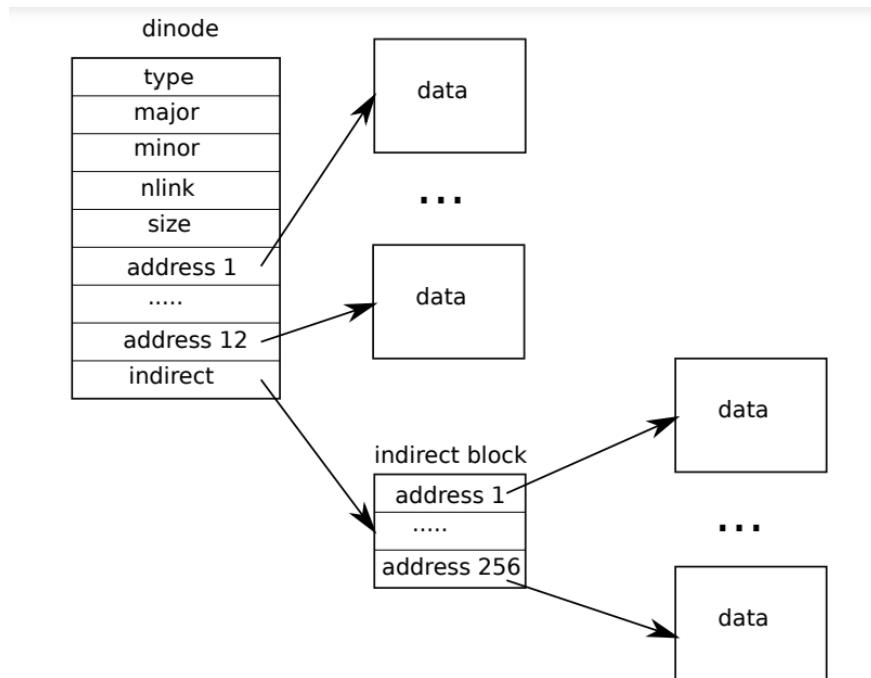


Figure 8.3: The representation of a file on disk.

函数 `bmap` (*kernel/fs.c:378*) 从简单的情况开始：前面的 `NDIRECT` 个块在`inode`本身中列出 (*kernel/fs.c:383-387*) 中。下面 `NINDIRECT` 个块在 `ip->addr[NDIRECT]` 的间接块中列出。`Bmap` 读取间接块 (*kernel/fs.c:394*)，然后从块内的正确位置 (*kernel/fs.c:395*) 读取块号。如果块号超过 `NDIRECT+NINDIRECT`，则 `bmap` 调用 `panic` 崩溃；`writei` 包含防止这种情况发生的检查 (*kernel/fs.c:490*)。

`Bmap` 根据需要分配块。`ip->addr[]` 或间接块中条目为零表示未分配块。当 `bmap` 遇到零时，它会用按需分配的新块 (*kernel/fs.c:384-385*) (*kernel/fs.c:392-393*) 替换它们。

`itrunc` 释放文件的块，将`inode`的 `size` 重置为零。`Itrunc` (**kernel/fs.c:410**) 首先释放直接块 (**kernel/fs.c:416-421**)，然后释放间接块中列出的块 (**kernel/fs.c:426-429**)，最后释放间接块本身 (**kernel/fs.c:431-432**)。

`Bmap` 使 `readi` 和 `writei` 很容易获取`inode`的数据。`Readi` (**kernel/fs.c:456**) 首先确保偏移量和计数不超过文件的末尾。开始于超过文件末尾的地方读取将返回错误 (**kernel/fs.c:461-462**)，而从文件末尾开始或穿过文件末尾的读取返回的字节数少于请求的字节数 (**kernel/fs.c:463-464**)。主循环处理文件的每个块，将数据从缓冲区复制到 `dst` (**kernel/fs.c:466-474**)。`writei` (**kernel/fs.c:483**) 与 `readi` 相同，但有三个例外：从文件末尾开始或穿过文件末尾的写操作会使文件增长到最大文件大小 (**kernel/fs.c:490-491**)；循环将数据复制到缓冲区而不是输出 (**kernel/fs.c:36**)；如果写入扩展了文件，`writei` 必须更新其大小 (**kernel/fs.c:504-511**)。

`readi` 和 `writei` 都是从检查 `ip->type == T_DEV` 开始的。这种情况处理的是数据不在文件系统中的特殊设备；我们将在文件描述符层返回到这种情况。

函数 `stati` (**kernel/fs.c:442**) 将`inode`元数据复制到 `stat` 结构体中，该结构通过 `stat` 系统调用向用户程序公开。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 14:12:30

- 8.11 代码：目录层

8.11 代码：目录层

目录的内部实现很像文件。其inode的 type 为 `T_DIR`，其数据是一系列目录条目（directory entries）。每个条目（entry）都是一个 `struct dirent` (*kernel/fs.h:56*)，其中包含一个名称 `name` 和一个inode编号 `inum`。名称最多为 `DIRSIZ` (14) 个字符；如果较短，则以 `NUL` (0) 字节终止。inode编号为零的条目是空的。

函数 `dirlookup` (*kernel/fs.c:527*) 在目录中搜索具有给定名称的条目。如果找到一个，它将返回一个指向相应inode的指针，解开锁定，并将 `*poff` 设置为目录中条目的字节偏移量，以满足调用方希望对其进行编辑的情形。如果 `dirlookup` 找到具有正确名称的条目，它将更新 `*poff` 并返回通过 `iget` 获得的未锁定的 inode。`Dirlookup` 是 `iget` 返回未锁定inode的原因。调用者已锁定 `dp`，因此，如果对`.`，当前目录的别名，进行查找，则在返回之前尝试锁定inode将导致重新锁定 `dp` 并产生死锁(还有更复杂的死锁场景，涉及多个进程和`..`，父目录的别名。`.`不是唯一的问题。) 调用者可以解锁 `dp`，然后锁定 `ip`，确保它一次只持有一个锁。

函数 `dirlink` (*kernel/fs.c:554*) 将给定名称和inode编号的新目录条目写入目录 `dp`。如果名称已经存在，`dirlink` 将返回一个错误 (*kernel/fs.c:560-564*)。主循环读取目录条目，查找未分配的条目。当找到一个时，它会提前停止循环 (*kernel/fs.c:538-539*)，并将 `off` 设置为可用条目的偏移量。否则，循环结束时会将 `off` 设置为 `dp->size`。无论哪种方式，`dirlink` 都会通过在偏移 `off` 处写入 (*kernel/fs.c:574-577*) 来向目录添加一个新条目。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 14:13:00

- 8.12 代码：路径名

8.12 代码：路径名

路径名查找涉及一系列对 `dirlookup` 的调用，每个路径组件调用一个。`Namei` (*kernel/fs.c:661*) 计算 `path` 并返回相应的`inode`。函数 `nameiparent` 是一个变体：它在最后一个元素之前停止，返回父目录的`inode`并将最后一个元素复制到 `name` 中。两者都调用通用函数 `namex` 来完成实际工作。

`Namex` (*kernel/fs.c:626*) 首先决定路径计算的开始位置。如果路径以斜线开始，则计算从根目录开始；否则，从当前目录开始 (*kernel/fs.c:630-633*)。然后，它使用 `skipelent` 依次考察路径的每个元素 (*kernel/fs.c:635*)。循环的每次迭代都必须在当前索引结点 `ip` 中查找 `name`。迭代首先给 `ip` 上锁并检查它是否是一个目录。如果不是，则查找失败 (*kernel/fs.c:636-640*) (锁定 `ip` 是必要的，不是因为 `ip->type` 可以被更改，而是因为在 `ilock` 运行之前，`ip->type` 不能保证已从磁盘加载。) 如果调用是 `nameiparent`，并且这是最后一个路径元素，则根据 `nameiparent` 的定义，循环会提前停止；最后一个路径元素已经复制到 `name` 中，因此 `namex` 只需返回解锁的 `ip` (*kernel/fs.c:641-645*)。最后，循环将使用 `dirlookup` 查找路径元素，并通过设置 `ip = next` (*kernel/fs.c:646-651*) 为下一次迭代做准备。当循环用完路径元素时，它返回 `ip`。

`namex` 过程可能需要很长时间才能完成：它可能涉及多个磁盘操作来读取路径名中所遍历目录的索引节点和目录块（如果它们不在`buffer cache`中）。Xv6经过精心设计，如果一个内核线程对 `namex` 的调用在磁盘I/O上阻塞，另一个查找不同路径名的内核线程可以同时进行。`Namex` 分别锁定路径中的每个目录，以便在不同目录中进行并行查找。

这种并发性带来了一些挑战。例如，当一个内核线程正在查找路径名时，另一个内核线程可能正在通过取消目录链接来更改目录树。一个潜在的风险是，查找可能正在搜索已被另一个内核线程删除且其块已被重新用于另一个目录或文件的目录。

Xv6避免了这种竞争。例如，在 `namex` 中执行 `dirlookup` 时，`lookup`线程持有目录上的锁，`dirlookup` 返回使用 `iget` 获得的`inode`。`Iget`增加索引节点的引用计数。只有在从 `dirlookup` 接收`inode`之后，`namex` 才会释放目录上的锁。现在，另一个线程可以从目录中取消`inode`的链接，但是xv6还不会删除`inode`，因为`inode`的引用计数仍然大于零。

另一个风险是死锁。例如，查找“.”时，`next` 指向与 `ip` 相同的`inode`。在释放 `ip` 上的锁之前锁定 `next` 将导致死锁。为了避免这种死锁，`namex` 在获得下一个目录的锁之前解锁该目录。这里我们再次看到为什么 `iget` 和 `ilock` 之间的分离很重要。

- 8.13 文件描述符层

8.13 文件描述符层

Unix界面的一个很酷的方面是， Unix中的大多数资源都表示为文件，包括控制台、管道等设备，当然还有真实文件。文件描述符层是实现这种一致性的层。

正如我们在第1章中看到的， Xv6为每个进程提供了自己的打开文件表或文件描述符。每个打开的文件都由一个 `struct file` (`kernel/file.h:1`) 表示，它是`inode`或管道的封装，加上一个I/O偏移量。每次调用 `open` 都会创建一个新的打开文件（一个新的 `struct file`）：如果多个进程独立地打开同一个文件，那么不同的实例将具有不同的I/O偏移量。另一方面，单个打开的文件（同一个 `struct file`）可以多次出现在一个进程的文件表中，也可以出现在多个进程的文件表中。如果一个进程使用 `open` 打开文件，然后使用 `dup` 创建别名，或使用 `fork` 与子进程共享，就会发生这种情况。引用计数跟踪对特定打开文件的引用数。可以打开文件进行读取或写入，也可以同时进行读取和写入。`readable` 和 `writable` 字段可跟踪此操作。

系统中所有打开的文件都保存在全局文件表 `ftable` 中。文件表具有分配文件（`filealloc`）、创建重复引用（`filedup`）、释放引用（`fileclose`）以及读取和写入数据（`fileread` 和 `filewrite`）的函数。

前三个函数遵循现在熟悉的形式。`Filealloc` (`kernel/file.c:30`) 扫描文件表以查找未引用的文件（`f->ref == 0`），并返回一个新的引用；`filedup` (`kernel/file.c:48`) 增加引用计数；`fileclose` (`kernel/file.c:60`) 将其递减。当文件的引用计数达到零时，`fileclose` 会根据 `type` 释放底层管道或 `inode`。

函数 `filestat`、`fileread` 和 `filewrite` 实现对文件的 `stat`、`read` 和 `write` 操作。`Filestat` (`kernel/file.c:88`) 只允许在`inode`上操作并且调用了 `stati`。`Fileread` 和 `filewrite` 检查打开模式是否允许该操作，然后将调用传递给管道或`inode`的实现。如果文件表示`inode`，`fileread` 和 `filewrite` 使用I/O偏移量作为操作的偏移量，然后将文件指针前进该偏移量 (`kernel/file.c:122-123`) (`kernel/file.c:153-154`)。管道没有偏移的概念。回想一下，`inode`的函数要求调用方处理锁 (`kernel/file.c:94-96`) (`kernel/file.c:121-124`) (`kernel/file.c:163-166`)。`inode`锁定有一个方便的副作用，即读取和写入偏移量以原子方式更新，因此，对同一文件的同时多次写入不能覆盖彼此的数据，尽管他们的写入最终可能是交错的。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 19:16:00

- 8.14 代码：系统调用

8.14 代码：系统调用

通过使用底层提供的函数，大多数系统调用的实现都很简单（请参阅 `kernel/sysfile.c`）。有几个调用值得仔细看看。

函数 `sys_link` 和 `sys_unlink` 编辑目录，创建或删除索引节点的引用。它们是使用事务能力的另一个很好的例子。`sys_link` (`kernel/sysfile.c:120`) 从获取其参数开始，两个字符串分别是 `old` 和 `new` (`kernel/sysfile.c:125`)。假设 `old` 存在并且不是一个目录 (`kernel/sysfile.c:129-132`)，`sys_link` 会增加其 `ip->nlink` 计数。然后 `sys_link` 调用 `nameiparent` 来查找 `new` (`kernel/sysfile.c:145`) 的父目录和最终路径元素，并创建一个指向 `old` 的 `inode` (`kernel/sysfile.c:148`) 的新目录条目。`new` 的父目录必须存在并且与现有 `inode` 位于同一设备上：`inode` 编号在一个磁盘上只有唯一的含义。如果出现这样的错误，`sys_link` 必须返回并减少 `ip->nlink`。

事务简化了实现，因为它需要更新多个磁盘块，但我们不必担心更新的顺序。他们要么全部成功，要么什么都不做。例如在没有事务的情况下，在创建一个链接之前更新 `ip->nlink` 会使文件系统暂时处于不安全状态，而在这两者之间发生的崩溃可能会造成严重破坏。对于事务，我们不必担心这一点

`Sys_link` 为现有 `inode` 创建一个新名称。函数 `create` (`kernel/sysfile.c:242`) 为新 `inode` 创建一个新名称。它是三个文件创建系统调用的泛化：带有 `O_CREATE` 标志的 `open` 生成一个新的普通文件，`mkdir` 生成一个新目录，`mkdev` 生成一个新的设备文件。与 `sys_link` 一样，`create` 从调用 `nameiparent` 开始，以获取父目录的 `inode`。然后调用 `dirlookup` 检查名称是否已经存在 (`kernel/sysfile.c:252`)。如果名称确实存在，`create` 的行为取决于它用于哪个系统调用：`open` 的语义与 `mkdir` 和 `mkdev` 不同。如果 `create` 是代表 `open` (`type == T_FILE`) 使用的，并且存在的名称本身是一个常规文件，那么 `open` 会将其视为成功，`create` 也会这样做 (`kernel/sysfile.c:256`)。否则，这是一个错误 (`kernel/sysfile.c:257-258`)。如果名称不存在，`create` 现在将使用 `alloc` (`kernel/sysfile.c:261`) 分配一个新的 `inode`。如果新 `inode` 是目录，`create` 将使用 `.` 和 `..` 条目对它进行初始化。最后，既然数据已正确初始化，`create` 可以将其链接到父目录 (`kernel/sysfile.c:274`)。`create` 与 `sys_link` 一样，同时持有两个 `inode` 锁：`ip` 和 `dp`。不存在死锁的可能性，因为索引结点 `ip` 是新分配的：系统中没有其他进程会持有 `ip` 的锁，然后尝试锁定 `dp`。

使用 `create`，很容易实

现 `sys_open`、`sys_mkdir` 和 `sys_mknod`。`Sys_open` (`kernel/sysfile.c:287`) 是最复杂的，因为创建一个新文件只是它能做的一小部分。如果 `open` 被传递了 `O_CREATE` 标志，它将调用 `create` (`kernel/sysfile.c:301`)。否则，它将调用 `namei` (`kernel/sysfile.c:307`)。`Create` 返回一个锁定的 `inode`，但 `namei` 不锁定，因此 `sys_open` 必须锁定 `inode` 本身。这提供了一个方便的地方来检查目录是否仅为读取打开，而不是写入。假设 `inode` 是以某种方式获得的，`sys_open` 分配一个文件和一个文件描述符 (`kernel/sysfile.c:325`)，然后填充该文件 (`kernel/sysfile.c:337-342`)。请注意，没有其他进程可以访问部分初始化的文件，因为它仅位于当前进程的表中。

在我们还没有文件系统之前，第7章就研究了管道的实现。函数 `sys_pipe` 通过提供创建管道对的方法将该实现连接到文件系统。它的参数是一个指向两个整数的指针，它将在其中记录两个新的文件描述符。然后分配管道并安装文件描述符。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 19:16:16

8.15 真实世界

实际操作系统中的buffer cache比xv6复杂得多，但它有两个相同的用途：缓存和同步对磁盘的访问。与UNIX V6一样，Xv6的buffer cache使用简单的最近最少使用（LRU）替换策略；有许多更复杂的策略可以实现，每种策略都适用于某些工作场景，而不适用于其他工作场景。更高效的LRU缓存将消除链表，而改为使用哈希表进行查找，并使用堆进行LRU替换。现代buffer cache通常与虚拟内存系统集成，以支持内存映射文件。

Xv6的日志系统效率低下。提交不能与文件系统调用同时发生。系统记录整个块，即使一个块中只有几个字节被更改。它执行同步日志写入，每次写入一个块，每个块可能需要整个磁盘旋转时间。真正的日志系统解决了所有这些问题。

日志记录不是提供崩溃恢复的唯一方法。早期的文件系统在重新启动期间使用了一个清道夫程序（例如，UNIX的fsck程序）来检查每个文件和目录以及块和索引节点空闲列表，查找并解决不一致的问题。清理大型文件系统可能需要数小时的时间，而且在某些情况下，无法以导致原始系统调用原子化的方式解决不一致问题。从日志中恢复要快得多，并且在崩溃时会导致系统调用原子化。

Xv6使用的索引节点和目录的基础磁盘布局与早期UNIX相同；这一方案多年来经久不衰。BSD的UFS/FFS和Linux的ext2/ext3使用基本相同的数据结构。文件系统布局中最无效的部分是目录，它要求在每次查找期间对所有磁盘块进行线性扫描。当目录只有几个磁盘块时，这是合理的，但对于包含许多文件的目录来说，开销巨大。Microsoft Windows的NTFS、Mac OS X的HFS和Solaris的ZFS（仅举几例）将目录实现为磁盘上块的平衡树。这很复杂，但可以保证目录查找在对数时间内完成（即时间复杂度为 $O(\log n)$ ）。

Xv6对于磁盘故障的解决很初级：如果磁盘操作失败，Xv6就会调用panic。这是否合理取决于硬件：如果操作系统位于使用冗余屏蔽磁盘故障的特殊硬件之上，那么操作系统可能很少看到故障，因此panic是可以的。另一方面，使用普通磁盘的操作系统应该预料到会出现故障，并能更优雅地处理它们，这样一个文件中的块丢失不会影响文件系统其余部分的使用。

Xv6要求文件系统安装在单个磁盘设备上，且大小不变。随着大型数据库和多媒体文件对存储的要求越来越高，操作系统正在开发各种方法来消除“每个文件系统一个磁盘”的瓶颈。基本方法是将多个物理磁盘组合成一个逻辑磁盘。RAID等硬件解决方案仍然是最流行的，但当前的趋势是在软件中尽可能多地实现这种逻辑。这些软件实现通常允许通过动态添加或删除磁盘来扩展或缩小逻辑设备等丰富功能。当然，一个能够动态增长或收缩的存储层需要一个能够做到这一点的文件系统：xv6使用的固定大小的inode块阵列在这样的环境中无法正常工作。将磁盘管理与文件系统分离可能是最干净的设计，但两者之间复杂的接口导致了一些系统（如Sun的ZFS）将它们结合起来。

Xv6的文件系统缺少现代文件系统的许多其他功能；例如，它缺乏对快照和增量备份的支持。

现代Unix系统允许使用与磁盘存储相同的系统调用访问多种资源：命名管道、网络连接、远程访问的网络文件系统以及监视和控制接口，如/proc（注：Linux内核提供了一种通过/proc文件系统，在运行时访问内核内部数据结构、改变内核设置

的机制。`proc`文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。）。不同于xv6中`fileread`和`filewrite`的`if`语句，这些系统通常为每个打开的文件提供一个函数指针表，每个操作一个，并通过函数指针来援引`inode`的调用实现。网络文件系统和用户级文件系统提供了将这些调用转换为网络RPC并在返回之前等待响应的函数。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 14:13:56

- 8.16 练习

8.16 练习

1. 为什么要在 `balloc` 中 `panic` ? xv6可以恢复吗?
2. 为什么要在 `ialloc` 中 `panic` ? xv6可以恢复吗?
3. 当文件用完时, `filealloc` 为什么不 `panic` ? 为什么这更常见, 因此值得处理?
4. 假设在 `sys_link` 调用 `iunlock(ip)` 和 `dirlink` 之间, 与 `ip` 对应的文件被另一个进程解除链接。链接是否正确创建? 为什么?
5. `create` 需要四个函数调用都成功 (一次调用 `ialloc`, 三次调用 `dirlink`) 。如果未成功, `create` 调用 `panic` 。为什么这是可以接受的? 为什么这四个调用都不能失败?
6. `sys_chdir` 在 `iput(cp->cwd)` 之前调用 `iunlock(ip)`, 这可能会尝试锁定 `cp->cwd`, 但将 `iunlock(ip)` 延迟到 `iput` 之后不会导致死锁。为什么不这样做?
7. 实现 `lseek` 系统调用。支持 `lseek` 还需要修改 `filewrite`, 以便在 `lseek` 设置 `off` 超过 `f->ip->size` 时, 用零填充文件中的空缺。
8. 将 `O_TRUNC` 和 `O_APPEND` 添加到 `open`, 以便 `>` 和 `>>` 操作符在 `shell` 中工作。
9. 修改文件系统以支持符号链接。
10. 修改文件系统以支持命名管道。
11. 修改文件和VM系统以支持内存映射文件。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间: 2021-08-19 19:15:26

- [Introduction](#)
- [课程介绍](#)
 - 概述
 - 课程结构
 - [UNIX系统调用简介](#)
 - 例子:`copy.c`: 复制输入到输出
 - 例子:`open.c`, 创建一个文件
 - 例子: `fork.c`: 创建一个新的过程
 - 例子:`exec.c`: 用可执行文件替换调用进程
 - 例子:`forkexec.c`。`fork()`一个新进程, `exec()`一个程序
 - 例子:`redirect.c`, 重定向命令的输出
 - 例子:`pipe1.c`, 通过管道交流
 - 例子:`pipe2.c`, 进程之间的通信
 - 例子:`list.c`, 列出目录中的文件
 - [总结](#)

Introduction

课程介绍

6.S081 2020 Lecture 1: OS概述

概述

- 6.S081目标
 - 了解操作系统(OS)的设计和实现
- 动手扩展小型操作系统的实践经验
- 有编写系统软件的实际操作经验
- 操作系统的目的是什么?
 - 为方便和可移植性而对硬件进行抽象
- 在多种应用中实现硬件的多路复用
- 隔离应用程序, 多个程序互不干扰
- 允许在合作的应用程序之间共享
- 控制共享安全
- 不要妨碍高效
- 支持广泛的应用
- 组织方式:分层结构
 - 用户应用层:`vi`、`gcc`、`DB`和`c`
- 内核服务层
- 硬件层: CPU、RAM、磁盘、网络等

我们非常关心接口和内部内核结构

- 操作系统内核通常提供什么服务?
 - 进程(一个正在运行的程序)
- 内存分配
- 文件内容
- 文件名, 目录
- 访问控制(安全性)
- 其他:用户、IPC (进程间通信)、网络、时间、终端
- 什么是应用程序/内核接口?
 - “系统调用”
- 例子, 在UNIX(如Linux, macOS, FreeBSD)中的C语言中:

```
fd = open("out", 1);
write(fd, "hello\n", 6);
pid = fork();
```

- 这些看起来像函数调用, 但实际上不是
- 为什么操作系统的[设计和实现是困难和有趣的?](#)
 - 恶劣的环境:古怪的硬件, 很难调试
- 许多设计张力:
 - 高效vs抽象/便携/通用
- 强大的vs简单的接口
- 灵活vs安全
- 功能交互:`'fd = open(); fork()'`
- 用途多种多样:笔记本电脑、智能手机、云计算、虚拟机、嵌入式
- 不断发展的硬件:NVRAM、多核、高速网络
- 你会很高兴你选了这门课, 如果你...
 - 关心计算机运行的背后发生了什么
- 喜欢基础架构
- 需要追踪漏洞或安全问题
- 注重高性能

课程结构

- 网上课程信息:
 - [6.S081官网](#), 网站中包含了课程表, 作业, 实验
- [Piazza](#): 公告, 讨论, 实验帮助

- 视频课程
 - 操作系统的想法
- 通过代码和xv6的书, xv6 (一个小的操作系统) 的案例研究
- 实验背景
- 操作系统相关论文
- 上课前提交一个关于阅读材料的问题。
- 实验:
 - 重点是:实践经验 (基本上每个星期一个实验)
- 实验的三种类型:
 - 系统编程(下周截止...)
 - OS原语, 例如线程切换。
 - xv6的OS内核扩展, 例如网络。
 - 使用piazza提问/回答实验室的问题。
 - 讨论很好, 但请不要看别人的解决方案!
 - 评分:
 - 70%的实验室, 基于测试(与运行的测试相同)。
 - 20%的实验室检查会议:我们会问你关于随机选择的实验室的问题。
 - 10%的家庭作业和课堂/广场讨论。
 - 没有考试, 没有小测验。
 - 请注意, 大部分成绩来自实验室, 请尽早开始!

UNIX系统调用简介

- 应用程序通过系统调用查看操作系统;这种接口将是我们关注的重点。
 - 让我们从查看程序如何使用系统调用开始。
- 您将在第一个实验室中使用这些系统调用。
- 并在随后的实验室中进行扩展和改进。
- 我将展示一些示例, 并在xv6上运行它们。

xv6的结构与UNIX系统(如Linux)类似。但是要简单得多——您将能够理解xv6的全部内容附带的书解释了xv6的工作原理和原因

- 为什么选择UNIX ?
 - 开源代码, 有良好的文档, 干净的设计, 广泛使用
- 如果您需要了解Linux内部, 学习xv6将有所帮助
- xv6在6.S081中有两个角色:
 - 核心函数的例子:虚拟内存, 多核, 中断, 等等
 - 大多数实验的起点

- xv6运行在RISC-V上，就像当前的6.004一样
- 您将在qemu机器仿真器下运行xv6

例子: `copy.c` : 复制输入到输出

```
char buf[64];

while(1){
    int n = read(0, buf, sizeof(buf));
    if(n <= 0)
        break;
    write(1, buf, n);
}

exit(0);
```

从输入中读取字节，并将其写入输出

`copy.c` 是用C语言写的，克尼根和里奇(K&R)的《C程序设计语言》可以帮助你学习C语言，另外你可以通过官网上时间表中的 `example` 指向的链接找到这些示例程序

其中 `read()` 和 `write()` 是系统调用

- `read()/write()`的第一个参数是一个“文件描述符”(fd)

它传递给内核，告诉系统调用要读/写哪个“打开的文件”，fd必须是已经打开过的，可以指向文件/设备/套接字等等。一个进程可以打开很多文件，有很多 fd，UNIX约定:fd 0是“标准输入”，1是“标准输出”

- 第二个`read()`参数是一个指向要读取的内存的指针
- 第三个参数是要读取的最大字节数

`read()`可以读得更少，但不能读得更多

- 返回值:实际读取的字节数，或-1表示错误

注意:`copy.c`不关心数据的格式，UNIX I/O是8位字节，解释是特定于应用程序的，例如数据库记录，C源，等等

文件描述符来自哪里？

例子: `open.c` , 创建一个文件

```

// open.c: create a file, write to it.

#include "kernel/types.h"
#include "user/user.h"
#include "kernel/fcntl.h"

int
main()
{
    int fd = open("output.txt", O_WRONLY | O_CREATE);
    write(fd, "ooo\n", 4);

    exit(0);
}

```

`open()` 创建文件，返回文件描述符`fd`(或-1表示错误)，`fd` 是一个短整数，`fd` 索引到内核维护的每个进程表中

不同的进程具有不同的`fd` 命名空间，也就是说，文件描述符1对于不同的进程通常意味着不同的东西，然而这些例子忽略了可能的错误——但你不要这么草率！

《xv6》中的图1.2列出了系统调用的参数、返回值，或者你查看UNIX手册页，例如。`man 2 open`

- 当程序调用像`open()`这样的系统调用时会发生什么？

`open` 看起来像一个函数调用，但实际上是一个特殊的指令

- 硬件保存一些用户寄存器
- 硬件增加特权级别
- 硬件会跳转到内核中一个已知的“入口点”
- 现在在内核中运行C代码
- 内核调用系统调用执行
 - `open()`在文件系统中查找文件名
 - 它可能会等待磁盘
 - 它更新内核数据结构(缓存，FD表)
- 恢复用户寄存器
- 减少特权级别
- 回到程序中的调用点，它将继续运行

我们将在后面的课程中看到更多细节

- `Shell`是UNIX系统上的命令行界面。

`shell`会打印“\$”提示符提示输入命令，它允许您运行UNIX命令行实用程序，这对系统管理、文件处理、开发、脚本编写非常有用

```

$ ls
$ ls > out
$ grep x < out

```

UNIX也支持其他类型的交互，例如窗口系统，图形用户界面，服务器，路由器，等等。但是，通过`shell`实现分时是UNIX最初的重点。我们可以通过`shell`执行许多系统调用。

例子：`fork.c`：创建一个新的过程

```

// fork.c: create a new process

#include "kernel/types.h"
#include "user/user.h"

int
main()
{
    int pid;

    pid = fork();

    printf("fork() returned %d\n", pid);

    if(pid == 0){
        printf("child\n");
    } else {
        printf("parent\n");
    }

    exit(0);
}

```

shell会为您键入的每个命令创建一个新进程，例如，对于

```
$ echo hello
```

来说 `fork()` 系统调用创建一个新进程，内核复制调用进程的指令、数据、寄存器、文件描述符、当前目录

“父进程”和“子进程”的唯一的区别是：`fork()` 在父进程中返回`pid`，在子进程中返回0，`pid`(进程ID)是一个整数，内核给每个进程一个不同的`pid`

因此：`fork.c` 的 `printf("fork() returned %d\n", pid);` 会在父子两个进程中执行

“`if(pid == 0)` ”允许代码进行区分父子进程

`fork`让我们创建一个新进程，那么我们如何在这个进程中运行一个程序呢？

例子：exec.c：用可执行文件替换调用进程

```

// exec.c: replace a process with an executable file

#include "kernel/types.h"
#include "user/user.h"

int
main()
{
    char *argv[] = { "echo", "this", "is", "echo", 0 };

    exec("echo", argv);

    printf("exec failed!\n");

    exit(0);
}

```

shell是如何运行程序的？例如

```
$ echo a b c
```

程序存储在一个文件中，指令和初始内存由编译器和链接器创建

有一个叫echo的文件，包含指令

```
$ echo
```

echo.c 文件内容如下

```
#include "kernel/types.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
    int i;

    for(i = 1; i < argc; i++){
        write(1, argv[i], strlen(argv[i]));
        if(i + 1 < argc){
            write(1, " ", 1);
        } else {
            write(1, "\n", 1);
        }
    }
    exit(0);
}
```

- exec() 系统调用
 - 用可执行文件替换当前进程
 - 丢弃指令和数据存储器
 - 从文件中加载指令和内存
 - 保存文件描述符
- exec(filename, argument-array)
 - argument-array保存命令行参数;exec将参数传递给main()
 - 执行 cat user/echo.c
 - echo.c 程序演示了如何查看命令行参数

例子: forkexec.c 。 fork() 一个新进程， exec() 一个程序

```

#include "kernel/types.h"
#include "user/user.h"

// forkexec.c: fork then exec

int
main()
{
    int pid, status;

    pid = fork();
    if(pid == 0){
        char *argv[] = { "echo", "THIS", "IS", "ECHO", 0 };
        exec("echo", argv);
        printf("exec failed!\n");
        exit(1);
    } else {
        printf("parent waiting\n");
        wait(&status);
        printf("the child exited with status %d\n", status);
    }

    exit(0);
}

```

- `forkexec.c` 包含一个常见的UNIX习惯用法:
 - `fork()`一个子进程
 - `exec()`子进程中的命令
 - 父进程调用 `wait()` 等待子进程结束
- 对于您键入的每个命令，`shell`都会`fork/exec/wait`
 - 在 `wait()` 完成之后，`shell`打印下一个提示符
 - 若想让程序在后台运行，可在命令的最后加上符号 `&`，这样`shell`会跳过 `wait()`
- `exec(status) -> wait(&status)`
 - 状态约定:0表示成功，1表示命令遇到错误
- 注意: `fork()` 会复制，但是 `exec()` 会丢弃复制的内存

这似乎很浪费，在“copy-on-write”实验室中，你将透明的删除复制

例子: `redirect.c`，重定向命令的输出

```

#include "kernel/types.h"
#include "user/user.h"
#include "kernel/fcntl.h"

// redirect.c: run a command with output redirected

int
main()
{
    int pid;

    pid = fork();
    if(pid == 0){
        close(1);
        open("output.txt", O_WRONLY|O_CREATE);

        char *argv[] = { "echo", "this", "is", "redirected", "echo", 0 };
        exec("echo", argv);
        printf("exec failed!\n");
        exit(1);
    } else {
        wait((int *) 0);
    }

    exit(0);
}

```

shell如何完成重定向呢?

```
$ echo hello > out
```

答案是:通过 `fork` 产生子进程, 然后在子进程中改变文件描述符1, 再调用 `exec` 执行**echo**

- 注意:`open()`总是选择最小的未使用文件描述符;在重定向中, 由于 `close(1)` 使得1成为了最小的文件描述符
- `fork` 、`FDs` (文件描述符) 和 `exec` 可以很好地交互以实现I/O重定向
 - 将 `fork` 和 `exec` 分离给了子进程一个在 `exec` 之前更改文件描述符的机会

文件描述符提供了一种间接性: 命令只使用描述符0和1, 而不需要知道文件描述符到底指向去哪里

`exec` 会保存**shell**所设置的文件描述符

- 因此:只有**shell**需要知道I/O重定向, 而不是每个程序
- 关于设计决策, 有必要问一下“为什么”:
 - 为什么要采用这些I/O和流程抽象?为什么不是别的?
 - 为什么要提供文件系统?为什么不让程序以自己的方式使用磁盘呢?
 - 为什么使用文件描述符?为什么不将文件名传递给**write()**?
 - 为什么文件是字节流, 而不是磁盘块或格式化的记录?
 - 为什么不合并**fork()**和**exec()**呢?
 - UNIX设计工作得很好, 但我们将看到其他设计!

例子: **pipe1.c** , 通过管道交流

```

// pipe1.c: communication over a pipe

#include "kernel/types.h"
#include "user/user.h"

int
main()
{
    int fds[2];
    char buf[100];
    int n;

    // create a pipe, with two FDs in fds[0], fds[1].
    pipe(fds);

    write(fds[1], "this is pipe1\n", 14);
    n = read(fds[0], buf, sizeof(buf));

    write(1, buf, n);

    exit(0);
}

```

shell是如何实现管道的呢

```
$ ls | grep x
```

- 文件描述符可以指向“管道”，也可以指向文件
- `pipe()` 系统调用创建两个文件描述符
 - 第一个用于读取
 - 第二个用于写入
- 内核为每个管道维护一个缓冲区
 - `write()` 追加到缓冲区
 - `read()` 等待数据

例子: **pipe2.c** , 进程之间的通信

```

#include "kernel/types.h"
#include "user/user.h"

// pipe2.c: communication between two processes

int
main()
{
    int n, pid;
    int fds[2];
    char buf[100];

    // create a pipe, with two FDs in fds[0], fds[1].
    pipe(fds);

    pid = fork();
    if (pid == 0) {
        write(fds[1], "this is pipe2\n", 14);
    } else {
        n = read(fds[0], buf, sizeof(buf));
        write(1, buf, n);
    }

    exit(0);
}

```

- 管道和 `fork()` 很好地结合在一起实现 `ls | grep x`
 - shell 创建一个管道,
 - 然后执行两次 `fork`
 - 然后将 `ls` 的文件描述符 1 连接到管道的写文件描述符
 - 将 `grep` 的文件描述符 0 连接到管道的读文件描述符

管道是一个单独的抽象，但与 `fork()` 结合得很好。

例子: `list.c` , 列出目录中的文件

```

#include "kernel/types.h"
#include "user/user.h"

// list.c: list file names in the current directory

struct dirent {
    ushort inum;
    char name[14];
};

int
main()
{
    int fd;
    struct dirent e;

    fd = open(".", 0);
    while(read(fd, &e, sizeof(e)) == sizeof(e)){
        if(e.name[0] != '\0'){
            printf("%s\n", e.name);
        }
    }
    exit(0);
}

```

`ls`如何得到一个目录中的文件列表呢？

你可以打开一个目录并读取它->文件名

"`.`"是进程当前目录的伪名称

请参阅 `ls.c` 了解更多细节

总结

- 我们已经了解了**UNIX**的I/O、文件系统和进程抽象。
- 接口很简单——只有整数和I/O缓冲区。
- 抽象组合得很好，例如I/O重定向。

你们将在下周的第一个实验中使用这些系统调用。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 15:32:55

- 使用GUN Debugger
- 6.828的GDB
- GDB命令
 - 单步调试
 - 运行调试
 - 断点
 - 条件断点
 - 监视点
 - 检查命令
 - 其他检查命令
 - 布局
 - 其他技巧
- 其他
- QEMU使用
- 总结

使用GUN Debugger

6.828的GDB

- 我们提供一个名为`.gdbinit`的文件，自动设置GDB以用于QEMU
 - 必须在`lab`或`xv6`目录中运行GDB
 - 编辑`~/.gdbinit`以执行其他需要的GDB初始化
- 以带有或不带有GDB的方式使用`make`指令启动QEMU
 - 带有GDB：运行`make qemu[-nox]-gdb`，然后在第二个Shell中启动GDB（`iscv64-linux-gnu-gdb`）
 - 如果以单核方式启动，则使用`make CPUS=1 qemu-gdb`
 - 不带有GDB：当不需要GDB时使用`make qemu[-nox]`命令

GDB命令

- 当你不知道某个命令怎么使用时，运行`help <命令名称>`来获取帮助
- 在无歧义的情况下，所有命令都可以被简写

```
c`='co`='cont`='continue
```

- 一些额外的简写已经被定义,例如

```
s`='step` 以及 `si`='stepl
```

单步调试

- `step`一次运行一行代码。当有函数调用时，它将步进到被调用的对象函数。

- `next` 也是一次运行一行代码。但当有函数调用时，它不会进入该函数。
- `stepi` 和 `nexti` 对于汇编指令是单步调试。

所有命令都可以采用一个数字参数来指定重复执行的次数。按回车键将重复上一个命令。

运行调试

- `continue` 运行代码，直到遇到断点或使用 `<Ctrl-c>` 中断它
- `finish` 运行代码，直到当前函数返回
- `advance <location>` 运行代码，直到指令指针到达指定位置

断点

- `break <location>` 在指定的位置设置断点。位置可以是内存地址(`*0x7c00`)或名称(`monbacktrace` , `monitor.c:71`)
- 如需修改断点请使用 `delete` , `disable` , `enable`

条件断点

- `break <location> if <condition>` 在指定位置设置断点，但仅在满足条件时中断。
- `cond <number> <condition>` 在现有断点上添加条件。

监视点

类似于断点，但条件更为复杂。

- `watch <expression>` 每当表达式的值更改时，将停止执行
- `watch -l <address>` 每当指定内存地址的内容发生变化时，就会停止执行。
 - 命令 `wa var` 和 `wa -l &var` 有什么不同呢？
- `rwatch [-l] <expression>` 将在读取表达式的值时停止执行。

检查命令

- `x` 以您指定格式 (`x/x` 表示十六进制, `x/i` 表示汇编, 等等) 打印内存的原始内容。
- `print` 计算一个C表达式并将结果以合适的类型打印。它通常比 `x` 更有用
- 使用 `p *((struct elfhdr *) 0x10000)` 的输出比 `x/13x 0x10000` 的输出好得多

其他检查命令

- `info registers` 打印每个寄存器的值
- `info frame` 打印当前栈帧
- `list <location>` 在指定位置打印函数的源代码

- `backtrace` 或许对于你的lab1中的工作很有用处

布局

GDB有一个文本用户界面，在curses用户界面中显示有用的信息，如代码列表、反汇编和寄存器内容

- `layout <name>` 切换到给定的用户界面

例如 `layout split`，效果如下

```

kernel/main.c
12 {
B+>13     if(cpuid() == 0){
14         consoleinit();
15         printinit();
16         printf("\n");
}
17 void <main>      addi    sp,sp,-16
18 ma<main+2>      sd      ra,8(sp)
19 { 0x80000eac <main+4>      sd      s0,0(sp)
20 0x80000eae <main+6>      addi    s0,sp,16
B+>0x80000eb0 <main+8>      auipc   ra,0x1
21 0x80000eb4 <main+12>      jalr   -1298(ra)
22 0x80000eb8 <main+16>      auipc   a4,0x8
23     printf("Hello world\n");
remote Thread 1.1 In: main
(gdb) info reg          // physical page allocator
(gdb) kvminit();        // create kernel page table
L13  PC: 0x80000eb0

```

其他技巧

- 你可以使用 `set` 命令在执行期间更改变量的值。
- 你必须切换符号文件才能获得除内核以外环境的函数和变量名。例如，当调试 JOS时：

```

symbol-file obj/user/<name>
symbol-file obj/kern/kernel

```

符号文件（Symbol Files）是一个数据信息文件，它包含了应用程序二进制文件（比如：EXE、DLL等）调试信息，专门用来作调试之用，最终生成的可执行文件在运行时并不需要这个符号文件，但你的程序中所有的变量信息都记录在这个文件中。所以调试应用程序时，这个文件是非常重要的。用Visual C++ 和 WinDbg 调试程序时都要用到这个文件。

其他

```

layout asm : 查看汇编  layout reg : 查看寄存器  info reg : 查看寄存器 b
*0x1234 : 在指定地址设定断点

```

QEMU使用

`Ctrl+a c` : 进入控制模式 `info mem` : 打印页表

总结

- 使用 `help` 命令查看使用手册
- GDB是非常强大的，我们今天只触及表面
- 花费一个小时的时间学习如何使用GDB是非常值得的

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间： 2021-08-19 15:47:34

- C, Pointers, gdb
- 我的第一个内存bug
- C语言中的内存
- C语言指针
 - 指针语法
 - 回到内存
 - 指针算术运算，耶！
 - 指针算术运算，唉...
- C语言数组
 - C语言数组的缺陷
- C语言的位运算符
- C语言类型转换
- C语言的#include
- 一个关于指针的示例

C, Pointers, gdb

我的第一个内存bug

<pre>one = 1 two = one two += 1 print(one) print(two) ----- 1 2</pre>	<pre>abc = ['a', 'b', 'c'] abcdef = abc abcdef += ['d', 'e', 'f'] print(abc) print(abcdef) ----- ['a', 'b', 'c', 'd', 'e', 'f'] ['a', 'b', 'c', 'd', 'e', 'f']</pre>
---	--

C语言中的内存

1. 静态内存
2. 全局变量，可在整个程序中访问
3. 使用 `static` 关键字定义，和在全局范围中定义的变量一样。
4. 栈内存
5. 函数中的局部变量。函数退出后自动销毁。
6. 堆内存
7. 您使用 `malloc()` 和 `free()` 控制这些变量的创建和销毁

8. 由于使用结束后必须调用 `free()`，可能会导致内存泄漏

C语言指针

指针是一个64位整数，其值是内存中的地址。

每个变量都有一个地址，所以每个变量都会有对应的指针，包括指向指针的指针，指向指针的指针的指针，以此类推。

指针可以处理 `++`, `--`, `+`, `-` 这样的算数运算

指针语法

```
int x = 5;
int *x_addr = &x;           //等同于int* x_addr = &x; 例如值为0x7ffd2766a948
*x_addr = 6;               //可以使用*运算符访问基础值
int x_value = *x_addr;    //解引用，这将得到6
int arr1[10];             //数组隐含了指针！稍后将详细介绍。
int *arr2[20];            //指针数组，使arr2成为指向指针的指针
void *myPtr;
```

试试这些！在 `user/` 下创建一个新的程序，就像在 `Util` 中一样

回到内存

```
char *makeABC() {
    char y[3] = {'a', 'b', 'c'};
    return y;
}
```

这有什么错误？

指针算术运算，耶！

假设我们有一些值为 `0x100002` 的 `char *c`。

```
c++;      // 0x100003
c += 4;   // 0x100007
```

就该如此啊！

指针算术运算，唉...

假设我们有一些值为 `0x100002` 的 `int *i`。

```
i++;      //0x100006
i += 4;   //0x100016
```

指针以基本数据类型的长度（以字节为单位）进行加减。

C语言数组

C数组是存储特定数据类型的连续内存块。变量实际就是数组起始位置的指针。

```
char myString[40];           // myString的类型是char*
char *myArrayOfStrings[20];   // myArrayOfStrings的类型是char**
int counting[5] = {1, 2, 3, 4, 5}; // counting类型为int*
```

括号运算符 [] （例如访问 arr[1] ）只是指针算法的语法糖。

假设我们定义了 int arr[4] = {5, 6, 7, 8}; 那么下面这些是等价的：

```
arr[2] = 50;
*(arr + 2) = 50; // 记住指针的算术运算!
2[arr] = 50;     // 加法是交换的(排列次序不影响结果)!
```

C语言数组的缺陷

我们可以通过越界访问数组来访问或修改非法内存。C不提供任何检查。

这种行为可能是意想不到的。

需要时使用您的size变量！

C语言的位运算符

一切最终都是比特位，C语言允许我们操纵这些比特。

以下均为二进制数：

```
& (and/与): 10001 & 10000 -> 10000
| (or/或): 10001 | 10000 -> 10001
^ (xor/异或): 10001 ^ 10000 -> 00001
~ (complement/取反): ~10000 -> 01111
<< (left shift/左移): 1 << 4 -> 10000 (binary) -> 16 (decimal)
>> (right shfit/右移): 10101 >> 3 -> 10 (binary)
```

我们可以将这些运算符组合起来，使标志设置变得简单：

定义位偏移 flag0 = 0 , flag1 =1 , flag2 = 2 .

要设置标志 flag0 和 flag2 :

```
flags = (1 << flag0) | (1 << flag2) -> 101
```

要检查在整型标志变量中标志是否被设置：

```
if(flags & flag1) -> 101 & 010 == 0 (false!)
```

C语言类型转换

在C语言中进行类型转换的语法是: `(newType)variable`

将 `void*` 转换为 `char* : (char*)myVoidPtr`

从表达式转换为 `uint64 : (uint64)(2 + 3), (uint64)myVoidPtr`

关于一些好的例子, 请参见[kalloc.c](#)和[vm.c](#)。

```
extern char end[]; // first address after kernel.  
void kfree(void *pa) {  
    struct run *r;  
    if (((uint64)pa % PGSIZE) != 0 || ((char *)pa < end || (uint64)pa >= PHYSTOP)  
        panic("kfree");  
    ...  
}
```

C语言的#include

`.h` 文件包含声明 (构成)

`.c` 文件包含定义 (实现)

基本上从不 `#include .c` 类型的文件!

[include卫兵](#)帮助处理嵌套/重复 `#include` (在xv6中没有使用太多)

[!NOTE] include卫兵是指这样的结构

```
#ifndef XXXX_H  
#define XXXX_H  
...  
#endif
```

使用 `extern` 关键字! 将函数的可见性扩展到程序中的所有文件。

一个关于指针的示例

```

#include <stdio.h>
#include <stdlib.h>

void
f(void)
{
    int a[4];
    int *b = malloc(16);
    int *c;
    int i;

    printf("1: a = %p, b = %p, c = %p\n", a, b, c);

    c = a;
    for (i = 0; i < 4; i++)
        a[i] = 100 + i;
    c[0] = 200;
    printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);

    c[1] = 300;
    *(c + 2) = 301;
    3[c] = 302;
    printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);

    c = c + 1;
    *c = 400;
    printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);

    c = (int *) ((char *) c + 1);
    *c = 500;
    printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
           a[0], a[1], a[2], a[3]);

    b = (int *) a + 1;
    c = (int *) ((char *) a + 1);
    printf("6: a = %p, b = %p, c = %p\n", a, b, c);
}

int
main(int ac, char **av)
{
    f();
    return 0;
}

```

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 15:19:15

- [Calling Convention](#)
- [18.1 C语言的数据类型和对齐方式](#)
- [18.2 RVG调用协定](#)
- [18.3 软浮点数调用协定](#)

Calling Convention

本章描述了RV32和RV64程序的C编译器标准和两个调用约定：附加标准通用扩展（RV32G/RV64G）的基础ISA约定，以及缺乏浮点单元（例如RV32I/RV64I）实现的软浮点约定。

使用ISA扩展的实现可能需要扩展调用约定。

18.1 C语言的数据类型和对齐方式

表18.1总结了RISC-V C程序本机支持的数据类型。在RV32和RV64 C编译器中，C中的`int`类型都是32位。另一方面，`long`和指针都与整数寄存器位数一致，所以在RV32中，两者都是32位，而在RV64中，两者都是64位。同样，RV32采用ILP32整数模型，而RV64是LP64。在RV32和RV64中，C类型`long long`是64位整数，`float`是遵循IEEE754-2008标准的32位浮点数，`double`是遵循IEEE754-2008标准的64位浮点数，`long double`是遵循IEEE754-2008标准的128位浮点数。

C类型`char`和`unsigned char`都是8位无符号整数，当存储在RISC-V整数寄存器中时是零扩展。`unsigned short`是16位无符号整数，当存储在RISC-V整数寄存器中时是零扩展。`signed char`是8位有符号整数，当存储在RISC-V整数寄存器中时是符号扩展的，即比特位从(XLEN-1)到7都是相等的。`short`是16位有符号整数，当存储在寄存器中时是符号扩展的。

在RV64中，32位的数据类型（如`int`）以合适的符号扩展存储在整数寄存器中；也就是说，比特位从63到31都是相等的。即使是无符号的32位类型，这个限制也适用。

RV32和RV64 C编译器和兼容软件将所有上述数据类型存储在内存中时保持自然对齐。

C数据类型	描述	RV32中字节数	RV64中字节数
<code>char</code>	字符值/字节	1	1
<code>short</code>	短整型	2	2
<code>int</code>	整型	4	4
<code>long</code>	长整型	4	8
<code>long long</code>	超长整型	8	8
<code>void*</code>	指针	4	8
<code>float</code>	单精度浮点型	4	4
<code>double</code>	双精度浮点型	8	8
<code>long double</code>	扩展精度浮点型	16	16

表18.1: 基于RISC-V指令集的C编译器数据类型

18.2 RVG调用协定

RISC-V调用约定尽可能在寄存器中传递参数。为此，最多使用八个整数寄存器**a0-a7**和八个浮点寄存器**fa0-fa7**。

如果函数的参数被概念化为C结构体的字段，结构体中的每个字段都按指针长度对齐，则参数寄存器是该结构体中前八个指针字长参数的副本。如果第*i*(*i*<8)个参数是浮点类型，则在浮点寄存器**fa*i***中传递；否则，在整数寄存器**ai**中传递。但是，浮点参数如果属于 `union` 或结构体中数组字段的一部分，就会在整数寄存器中传递。此外，变参函数的浮点参数（未显式命名参数列表的函数）在整数寄存器中传递。

小于指针字长的参数在参数寄存器的最低有效位(**LSB**)中传递。相应地，栈上传递的小于指针字长的参数出现在指针字的较低地址中，因为RISC-V有一个小端存储系统。

当在堆栈上传递两倍于指针字大小的基本参数时，它们是自然对齐的。当它们在整数寄存器中传递时，它们驻留在对齐的偶数号-奇数号寄存器对中，偶数寄存器保存最低有效位。例如，在RV32中，函数 `void foo(int, long long)` 的第一个参数在**a0**中传递，第二个参数在**a2**和**a3**中传递。**a1**中不传递任何内容。

大于指针字大小两倍的参数通过引用传递。

结构体中未在参数寄存器中传递的部分在栈上传递。栈指针**sp**指向未在寄存器中传递的第一个参数。

函数在整数寄存器**a0**和**a1**以及浮点寄存器**fa0**和**fa1**中返回值。只有当浮点值是原始值（传入时**fa0**和**fa1**作为参数寄存器，原始值是指该参数不改变而直接返回）或作为仅有一两个浮点值组成的结构体的成员时，才会从浮点寄存器中返回。长度恰好为两个指针字长的其他返回值将在**a0**和**a1**中返回。较大的返回值完全在内存中传递；调用方分配此内存区域，并将指针作为隐式的第一个参数传递给被调用方。

在标准的RISC-V调用约定中，栈向下增长，栈指针始终保持16字节对齐。

除了自变量和返回值寄存器之外，还有在调用中不稳定的七个整数寄存器**t0-t6**和十二个浮点寄存器**ft0-ft11**作为临时寄存器，如果之后使用，调用者必须保存它们。十二个整数寄存器**s0-s11**和十二个浮点寄存器**fs0-fs11**在调用中受保护，如果使用，被调用者必须保存它们。表18.2显示了调用约定中每个整数和浮点寄存器的作用。

寄存器	ABI名称	描述	保存者
x0	zero	硬布线零	
x1	ra	返回地址	调用者
x2	sp	栈指针	被调用者
x3	gp	全局指针	
x4	tp	线程指针	
x5-7	t0-2	临时暂存单元	调用者
x8	s0/fp	保留寄存器/帧指针	被调用者
x9	s1	保留寄存器	被调用者
x10-11	a0-1	函数参数/返回值	调用者
x12-17	a2-7	函数参数	调用者
x18-27	s2-11	保留寄存器	被调用者
x28-31	t3-6	临时暂存单元	调用者
f0-7	ft0-7	浮点临时暂存单元	调用者
f8-9	fs0-1	浮点保留寄存器	被调用者
f10-11	fa0-1	浮点参数/返回值	调用者
f12-17	fa2-7	浮点参数	调用者
f18-27	fs2-11	浮点保留寄存器	被调用者
f28-31	ft8-11	浮点临时暂存单元	调用者

表18.2 RISC-V调用协定寄存器的使用

18.3 软浮点数调用协定

软浮点调用约定用于缺乏浮点硬件的RV32和RV64实现。它避免使用了F、D和Q标准扩展中的所有指令，从而避免使用f寄存器。

完整参数的传递和返回方式与RVG约定相同，栈规则也相同。浮点参数使用长度相同的整型参数的规则在整数寄存器中传递和返回。例如，在RV32中，函数 `double foo(int, double, long double)` 的第一个参数在a0中传递，第二个参数在a2和a3中传递，第三个参数通过a4传递引用；其结果在a0和a1中返回。在RV64中，参数以a0、a1和a2-a3对形式传递，结果以a0形式返回。

动态舍入模式和累计异常标志可以通过C99头文件 `fenv.h` 提供的程序访问。

注：为了编写高精度浮点数的运算，编程人员需要控制浮点数环境的各个方面：结果如何舍入，浮点数表达式如何简化与变换，如何处理浮点数异常（如下溢之类的浮点数异常是忽略还是产生错误）等等。C99引入了 `fenv.h` 来控制浮点数环境。

- Journaling the Linux ext2fs Filesystem
- 摘要
- 介绍
- 什么是文件系统
- 文件系统可靠性
- 现有实现
- 为Linux设计一个新的文件系统
- 事务剖析
- 事务合并
- 磁盘表示
- 文件系统日志的格式
- 日志的提交和检查点
- 事务间冲突
- 项目现状和未来的工作
- 结论

Journaling the Linux ext2fs Filesystem

摘要

本文描述了为Linux ext2fs文件系统设计和实现事务元数据日志的工作进展。我们回顾了崩溃后恢复文件系统的问题，并描述了一种旨在通过向文件系统添加事务日志来提高ext2fs崩溃恢复速度和可靠性的设计。

介绍

文件系统是任何现代操作系统的核心部分，人们期望它既快速又非常可靠。但是，由于硬件、软件或电源故障，问题仍然存在，机器可能会意外停机。

在一次意料之外的重启后，系统可能需要一些时间才能恢复文件系统的一致性状态。随着磁盘大小的增长，这一时间可能会成为一个严重的问题，在扫描、检查和修复磁盘时，系统会离线一小时或更长时间。尽管磁盘驱动器的速度每年都在加快，但与容量的巨大增长相比，这一速度的增长并不明显。不幸的是，在使用传统的文件系统检查技术时，磁盘容量每增加一倍，恢复时间就会增加一倍。

在系统可用性很重要的情况下，这可能是无法节省的时间，因此需要一种机制，以避免每次机器重新启动时都需要昂贵的恢复阶段。

什么是文件系统

对于任何文件系统，我们都需要什么功能？文件系统所服务的操作系统有明确的要求。文件系统对应用程序的表现方式是：一个操作系统通常需要遵守某些约定的文件名，并且文件具有某些以特定方式解释的属性。

然而，文件系统的许多内部方面没有那么受约束，文件系统实现者可以在一定程度上自由地设计这些方面。磁盘上数据的布局（或者，如果文件系统不是本地的，它的网络协议）、内部缓存的细节以及用于调度磁盘IO的算法——在不违反文件系统应用程序接口规范的前提下，这些都是可以改变的。

我们可能选择一种而不是另一种设计的原因有很多。与旧文件系统的兼容性可能是一个问题：例如，Linux提供了一个UMSDOS文件系统，它在标准MSDOS磁盘文件结构的基础上实现了POSIX文件系统的语义学。

当试图解决Linux上文件系统恢复时间过长的问题时，我们牢记许多目标：

- 使用新文件系统不会严重影响性能；
- 不得破坏与现有应用程序的兼容性
- 文件系统的可靠性不得以任何方式受到损害。

文件系统可靠性

当我们谈论文件系统的可靠性时，有许多问题利害攸关。就本特定项目而言，我们主要关心的是恢复崩溃文件系统内容的可靠性，我们可以确定其中的几个方面：

保持（Preservation）：崩溃前磁盘上稳定的数据永远会被损坏。显然，崩溃时正在写入的文件不能保证完全完好无损，但是恢复系统不能碰磁盘上已经安全的任何文件。

可预测性（Predictability）：我们必须恢复的故障模式应该是可预测的，以便我们可靠地恢复。

原子性（Atomicity）：许多文件系统操作需要大量独立的IO来完成。一个很好的例子是将文件从一个目录重命名到另一个目录。如果这样的文件系统操作在磁盘上完全完成，或者在恢复完成后完全撤销，恢复就是原子性的。（对于重命名的例子，恢复应该在崩溃后保留提交给磁盘的旧文件名或新文件名，但不能两者都保留。）

现有实现

Linux ext2fs文件系统提供了保留恢复（preserving recovery），但它是非原子的，不可预测。事实上，可预测性比乍一看要复杂得多。为了能够在崩溃后进行可预测的清理，恢复阶段必须能够确定文件系统在遇到表现为一次不完整操作的磁盘不一致性时试图做什么。通常，这要求在一次涉及磁盘上多个块更改的更新操作时，文件系统必须以可预测的顺序写入磁盘。

实现磁盘写入之间的这种排序有许多方法。最简单的方法是简单地等待第一次写入完成，然后再将下一次写入提交给设备驱动程序——“同步元数据更新（**synchronous metadata update**）”方法。这是BSD快速文件系统采取的方法，出现在4.2BSD中，它启发了随后的许多Unix文件系统，包括ext2fs。

然而，同步元数据更新的一大缺点是它的性能。如果文件系统操作要求我们等待磁盘IO完成，那么我们就不能将多个文件系统更新批处理成单个磁盘写入。例如，如果我们在磁盘上的同一个目录块中创建十几个目录项，那么同步更新需要我们将该块写回磁盘十几次。

有一些方法可以解决这个性能问题。一种保持磁盘写入顺序而不实际等待IO完成的方法是在内存中的磁盘缓冲区之间保持顺序，并确保当我们最终去写回数据时，在一个块的所有前置块都安全地写回磁盘前，我们永远都不会写该块，——“延迟有序写入”技术。

延迟有序写入的一个复杂性是很容易陷入缓存缓冲区之间存在循环依赖的情况。例如，如果我们试图在两个目录之间重命名一个文件，同时将第二个目录中的另一个文件重命名至第一个目录，那么我们最终会遇到两个目录块相互依赖的情况：两个目录块都在等待对方写回磁盘，最终都不能写入。

Ganger的“软更新”机制巧妙地避开了这个问题，当我们第一次尝试将缓冲区写入磁盘时，如果这些更新仍然有未完成的依赖关系，我们会有选择地回滚缓冲区中的特定更新。丢失的更新将在所有的依赖关系都得到满足后恢复。这使得我们可以在有循环依赖关系时以我们选择的任何顺序写入缓冲区。软更新机制已经被**FreeBSD**采用，并将作为他们下一个主要内核版本的一部分提供。

然而，所有这些方法都有一个共同的问题。尽管它们确保磁盘的状态在文件系统操作过程中一直处于可预测的状态，但恢复过程仍然必须扫描整个磁盘，以便找到和修复任何未完成的操作。恢复变得更加可靠，但不一定更快。

然而，在不牺牲可靠性和可预测性的情况下快速恢复文件系统是可能的。这通常由保证文件系统更新原子完成的文件系统来完成（在这样的系统中，单个文件系统更新通常被称为事务）。原子更新背后的基本原则是文件系统可以将整批新数据写入磁盘，但是这些更新在磁盘上进行最终提交更新之前不会生效。如果提交涉及到对磁盘的单个块的写入，那么崩溃只能导致两种情况：要么提交记录已经写入磁盘，在这种情况下，所有提交的文件系统操作都可以假设是完整的，并且在磁盘上是一致的；要么提交记录丢失，这种情况下，由于在崩溃时部分尚未提交的更新仍未完成，我们必须忽略任何其他写入操作。这自然需要文件系统更新，以将更新数据的新旧内容保存在磁盘上的某个地方，直到提交。

有许多方法可以实现这一点。在某些情况下，文件系统将更新数据的新副本保存在与旧副本不同的位置，并最终在更新提交到磁盘后重用旧空间。**WAFL**文件系统是这样工作的，维护一个文件系统数据树，它可以通过将树节点复制到新的位置，然后更新树根部的单个磁盘块来进行原子更新。（注：在**WAFL**中，如果它也修改一个数据，他可能不管以前的数据的位置，直接把新数据与新校验写到新的位置，之后更改指针，告诉文件系统说，新的数据在这里，而不是原来那里了。）

日志结构化文件系统通过将所有文件系统数据——包括文件内容和元数据——以连续流（“日志”）写入磁盘来实现相同的目的。使用这种方案查找一段数据的位置可能比在传统文件系统中更复杂，但是日志有一个很大的优势，那就是在日志中放置标记相对容易，以指示直到某个点的所有数据都已提交并在磁盘上保持一致。写入这样的文件系统也特别快，因为日志的性质使得大多数写入发生在没有磁盘查找的连续流中。许多文件系统都是基于这种设计编写的，包括**Sprite LFS**和**Berkeley LFS**。**Linux**上也有一个原型**LFS**实现。

最后，还有一类原子更新的文件系统，它将新版本写入磁盘上的单独位置，并在更新提交前保留旧版本和更新不完整的新版本。提交后，文件系统可以自由地将更新磁盘块的新版本写回磁盘上的原始位置。

这是日志记录journaling（有时称为日志增强版log enhanced）文件系统的工作方式。当磁盘上的元数据被更新时，更新被记录在磁盘上用作日志保留的单独区域中。完成的文件系统事务将提交记录添加到日志中，只有在提交安全地存储在磁盘上后，文件系统才能将元数据写回其原始位置。事务是原子的，因为我们总是可以在崩溃后根据日志是否包含事务的提交记录撤销事务（丢弃日志中的新数据）或重做事务（将日志副本复制回原始副本）。许多现代文件系统采用了这种设计的变体。

为Linux设计一个新的文件系统

Linux新文件系统设计背后的主要动机是消除崩溃后大型文件系统恢复时间。出于这个原因，我们选择了文件系统日志计划作为这项工作的基础。日志实现了快速的文件系统恢复，因为我们知道在任何时候，磁盘上可能不一致的所有数据都必须记录在日志中。因此，可以通过扫描日志并将所有提交的数据复制回主文件系统区域来实现文件系统恢复。这很快，因为日志通常比完整的文件系统小得多。它只需要足够记录几秒钟的未提交更新的容量。

选择日志记录还有另一个重要优势。日志记录文件系统不同于传统文件系统，因为它将临时数据保存在一个新的位置，独立于磁盘上的永久数据和元数据。正因为如此，这样的文件系统并不要求永久数据必须以任何特定的方式存储。特别是，ext2fs文件系统的磁盘结构很有可能在新文件系统中使用，现有的ext2fs代码也很有可能用作日志记录版本的基础。

因此，我们不是在为Linux设计一个新的文件系统。相反，我们在现有的ext2fs中添加了一个新的特性——事务性文件系统日志记录

事务剖析

当考虑日志文件系统时，一个核心概念是事务，对应于文件系统的单个更新。应用程序发出的任何单个文件系统请求都会产生一个事务，并且包含该请求产生的所有更改的元数据。例如，对文件的写入将导致对文件在磁盘上的索引节点中的修改时间戳的更新，如果文件被写操作扩展，还可能更新长度信息和块映射信息。配额信息、空闲磁盘空间和记录使用块的位图都必须更新，如果给文件分配新的块，所有这些都必须记录在事务中。

在事务中还有一个我们必须注意的隐藏操作。事务还包括读取文件系统的现有内容，这在事务之间强加了顺序。修改磁盘上块的事务不能在读取新数据并根据读取的内容更新磁盘的事务之后提交。即使两个事务从来没有尝试写回相同的块，依赖性也是存在的——想象一个事务从目录中的一个块中删除文件名，另一个事务将相同的文件名插入到不同的块中。这两个操作在它们写入的块中可能不会重叠，但是第二个操作只有在第一个操作成功后才有效（违反这一操作将导致重复的目录条目）。

最后，除了元数据更新之间的排序之外，还有一个排序要求。在我们提交将新块分配给文件的事务之前，我们必须绝对确保事务创建的所有数据块实际上都已写入磁盘（我们称这些数据块为依赖数据dependent data）。忽略此要求实际上不会损害文件系统元数据的完整性，但它可能会导致新文件崩溃恢复后仍包含以前的文件内容，这是一个安全风险，也是一个一致性问题。

事务合并

日志文件系统中使用的许多术语和技术来自数据库世界，日志是确保复杂事务原子提交的标准机制。然而，传统数据库事务和文件系统之间有许多不同之处，其中一些允许我们大大简化事情。

两个最大的区别是文件系统没有事务中止，所有文件系统事务都相对短暂。而在数据库中，我们有时想中途止事务，丢弃我们迄今为止所做的任何更改，在ext2fs中情况并非如此——当我们开始对文件系统进行任何更改时，我们已经检查了更改是否可以合法完成。在我们开始写入更改之前中止事务（例如，如果一个创建文件操作找到一个相同名称的现有文件，它可能会中止）不会带来任何问题，因为在这种情况下，我们可以简单地提交事务而不做任何更改，并实现相同的效果。

第二个区别——文件系统事务存在期很短——这很重要，因为这意味着我们可以极大地简化事务之间的依赖关系。如果我们必须满足一些非常长期的事务，那么我们需要允许事务以任何顺序独立提交，只要它们彼此不冲突，否则一个停滞不前的事务可能会拖累整个系统。然而，如果所有事务都足够快，那么我们可以要求事务以严格的顺序提交到磁盘，而不会明显损害性能。

通过这个观察，我们可以对事务模型进行简化，从而大大降低实现的复杂性，同时提高性能。与为每个文件系统更新创建单独的事务不同，我们只是经常创建一个新事务，并允许所有文件系统服务调用将它们的更新添加到单个系统范围的复合事务中。

这种机制有一个很大的优点。因为复合事务中的所有操作都将一起提交到日志中，所以我们不必为任何经常更新的元数据块编写单独的副本。特别是，这有助于创建新文件等操作，在这些操作中，对文件的每次写入都会导致文件被扩展，从而连续更新相同的配额、位图块和索引节点块。在复合事务的生命周期中，任何多次更新的块只需要提交到磁盘一次。

关于何时提交当前复合事务并启动新事务的决定是一个应该由用户控制的策略决定，因为它涉及到影响系统性能的权衡。提交等待的时间越长，可以在日志中合并的文件系统操作就越多，因此从长远来看需要的IO操作就越少。然而，更长的提交占用了大量的内存和磁盘空间，并在崩溃发生时留下了更大的更新丢失窗口。它们还可能导致磁盘活动的骤变，从而使文件系统响应时间难以预测。

磁盘表示

磁盘上记录的ext2fs文件系统的布局将与现有的ext2fs内核完全兼容。传统的UNIX文件系统通过将每个文件与磁盘上唯一编号的inode关联起来，将数据存储在磁盘上，而ext2fs设计已经包含了许多保留的inode编号。我们使用其中一个保留索引节点来存储文件系统日志，并且在所有其他方面，文件系统都将与现有的Linux内核兼容。现有的ext2fs设计包括一组兼容性位图，其中可以设置位来指示文件系统是否使用特定扩展。通过为日志扩展分配一个新的兼容性位，我们可以确保即使旧内核能够成功挂载一个新的、日志记录的ext2fs文件系统，它们也不会被允许以任何方式写入文件系统。

文件系统日志的格式

日志文件的工作很简单：它在我们提交事务的过程中记录文件系统元数据块的新内容。日志的唯一其他要求是我们必须能够原子地提交它包含的事务。

我们向日志写入三种不同类型的数据块：元数据块、描述符块和头块（**metadata, descriptor and header blocks**）。

日志元数据块包含由事务更新的单个文件系统元数据块的全部内容。这意味着，无论我们对文件系统元数据块做了多么小的更改，我们都必须写出整个日志块来记录更改。然而，由于两个原因，这一成本相对较低：

- 无论如何，日志写入非常快，因为对日志的大多数写入都是顺序的，我们可以很容易地将日志IO批处理成大型集群，磁盘控制器可以有效地处理这些集群；
- 通过将更改后的元数据缓冲区的全部内容从文件系统缓存写入日志，我们可以避免在日志代码中执行大量CPU工作。

Linux内核已经为我们提供了一种非常有效的机制，可以将**buffer cache**中现有块的内容写到磁盘上的不同位置。**buffer cache**中的每个缓冲区都由一个名为 **buffer_head** 的结构体描述，该结构体包括缓冲区的数据要写到哪个磁盘块的信息。如果我们想将整个缓冲区块在不干扰 **buffer_head** 的情况下写入新位置，我们可以简单地创建一个新的临时 **buffer_head**，将旧的描述复制到其中，然后编辑临时 **buffer_head** 中的设备块编号字段，以指向日志文件中的块。然后，我们可以将临时 **buffer_head** 直接提交给设备IO系统，并在IO完成后丢弃它。

描述符块是描述其他日志元数据块的日志块，每当我们需要将元数据块写出到日志时，我们需要记录下元数据通常安置在哪些磁盘块，这样恢复机制就可以将元数据复制回主文件系统中。在日志中的每一组元数据块之前都会写出一个描述符块，其中包含要写入的元数据块的数量加上它们的磁盘块号。

描述符块和元数据块都按顺序写入日志，每当我们运行超过末尾时，都会从日志的开头重新开始。在任何时候，我们都维护当前的日志头（最后写入的块的块号）和尾部（日志中尚未取消固定的最老的块，如下所述）。每当我们用完日志空间时——日志的头部已经循环回来并赶上了尾部——我们会停止新的日志写入，直到日志的尾部被清理干净，以释放更多的空间。

最后，日志文件包含一些位于固定位置的头块。这些头块记录了日志的当前头部和尾部，加上序列号。在恢复时，头块被扫描以找到序列号最高的块，当我们在恢复过程中扫描日志时，我们只是运行从尾部到头部的所有日志块，就像头块中记录的那样。

日志的提交和检查点

在某个时候，要么是因为上次提交后我们已经等了足够长的时间，要么是因为日志中的空间不足，我们希望将未完成的文件系统更新作为一个新的复合事务提交到日志中。

复合事务被完全提交后，我们仍然没有完成它。我们需要跟踪记录在事务中的元数据缓冲区，这样我们就可以注意到它们何时被写回磁盘上的主位置。

回想一下，当我们提交事务时，新更新的文件系统块位于日志中，但尚未同步回磁盘上的永久家块（家块就是写入操作对应的磁盘中文件系统对应的块，我们需要保持旧块的这种不同步，以防在提交日志之前崩溃）。一旦提交了日志，磁盘上的旧

版本就不再重要，我们可以在闲暇时将缓冲区写回它们的主位置。但是，在同步完这些缓冲区之前，我们不能删除日志中数据的副本。

要完全提交并完成事务的检查点，我们将经历以下阶段：

1. 关闭事务。在此刻，我们会建立一个新的事务以记录未来开始的任何文件系统操作。任何现有的、不完整的操作仍然会使用现有的事务：我们不能在多个事务上拆分单个文件系统操作！
2. 开始将事务刷新到磁盘。在一个单独的log-writer内核线程的上下文中，我们开始向日志写入所有被事务修改过的元数据缓冲区。在这个阶段，我们还必须写入任何依赖数据（参见上面的部分：事务解剖）。
3. 提交缓冲区后，将其标记以固定事务，直到它不再脏（它已通过通常的写回机制写回主存储）。
4. 等待此事务中所有未完成的文件系统操作完成。我们可以在所有操作完成之前安全地开始写日志，允许这两个步骤在某种程度上重叠会更快。
5. 等待所有未完成的事务更新完全记录在日志中。
6. 更新日志头块以记录日志的新头部和尾部，将事务提交到磁盘。**space released in the journal can now be reused by a later transaction.**
7. 当我们将事务的更新缓冲区写到日志中时，我们将它们标记以将事务固定在日志中。只有当这些缓冲区已同步到磁盘上的主缓冲区时，它们才会解除固定。只有当事务的最后一个缓冲区取消固定时，我们才能重用事务占用的日志块。当发生这种情况时，写入另一组日志头，记录日志尾部的新位置。日志中释放的空间现在可以由以后的事务重用。

事务间冲突

为了提高性能，我们在提交事务时不会完全暂停文件系统更新。相反，我们创建一个新的复合事务，在其中记录提交旧事务时到达的更新。

这就留下了一个问题，如果一个更新想要访问被另一个更新所占有的元数据缓冲区，而另一个更新包含于当前正在提交的旧事务，此时该怎么办。为了提交旧事务，我们需要将其缓冲区写入日志，但是我们不能在日志中写入任何不属于事务的更改，因为这将导致我们提交不完整的更新。

如果新事务只想读取有问题的缓冲区，那么没有问题：我们已经在两个事务之间创建了读/写依赖关系，但是由于复合事务总是以严格的顺序提交，我们可以安全地忽略冲突。

如果新事务想要写入缓冲区，事情就比较复杂了，我们需要缓冲区的旧副本来自提交第一个事务，但是我们不能让新事务在不让它修改缓冲区的情况下继续进行。

这里的解决方案是在这种情况下创建缓冲区的新副本。将一份副本提供给新事务以进行修改。另一个由旧事务保留，并将像往常一样提交到日志。一旦事务提交，此副本将被删除。当然，在文件系统中的其他地方安全地记录此缓冲区之前，我们无法回收旧事务的日志空间，但由于必须将缓冲区提交到下一个事务的日志记录中，这一点会自动得到处理。

项目现状和未来的工作

这仍然是一项正在进行的工作。初始实现的设计既稳定又简单，我们不期望为了完成实现而需要对设计进行任何重大修改。

上述设计相对简单，只需对现有**ext2fs**代码进行少量修改，即可处理日志文件的管理、缓冲区和事务之间的关联以及不干净关闭后的文件系统恢复。

一旦我们有了一个稳定的代码库来测试，我们可以在许多可能的方向上扩展基本设计。最重要的是文件系统性能的调优。这将要求我们研究日志系统中任意参数的影响，如提交频率和日志大小。它还将涉及瓶颈研究，以确定是否可以通过修改系统设计来提高性能，并且已经有几个可能的设计扩展。

一个研究领域可能是考虑压缩更新中的日志更新。目前的方案要求我们向日志写入整个元数据块，即使块中只有一个比特被修改。我们可以通过只记录缓冲区中更改的值而不是记录整个缓冲区来非常容易地压缩这些更新。然而，目前还不清楚这是否会带来任何重大的性能优势。目前的方案对大多数写入来说不需要内存到内存的拷贝，这在CPU和总线利用率方面是一个巨大的性能优势。写入整个缓冲区产生的IO开销很低——因为更新是连续的，在现代磁盘IO系统中，它们直接从主存储器传输到磁盘控制器，而不经过缓存或CPU。

另一个重要的可能扩展领域是对快速**NFS**服务器的支持。**NFS**设计允许客户端在服务器崩溃时正常恢复：客户端将在服务器重新启动时重新连接。如果发生这种崩溃，服务器尚未安全写入磁盘的任何客户端数据都将丢失，因此**NFS**要求服务器在将客户端的文件系统请求提交到服务器磁盘之前，不得确认该请求已完成。

对于通用文件系统来说，这可能是一个难以支持的特性。**NFS**服务器的性能通常通过对客户端请求的响应时间来衡量，如果这些响应必须等待文件系统更新与磁盘同步，则总体性能会受到磁盘上文件系统更新延迟的限制。这与文件系统的大多数其他用途不同，在文件系统中，性能是根据缓存内更新的延迟而不是磁盘上更新的延迟来衡量的。

有些文件系统是专门设计来解决这个问题的。**WAFL**是一个基于事务树的文件系统，它可以在磁盘上的任何地方写入更新，但是**Calaveras**文件系统通过使用类似于上面建议的日志来达到同样的目的。不同之处在于，**Calaveras**将每个应用程序的文件系统请求在日志中记录为一个单独的事务，从而尽可能快地在磁盘上完成单独的更新。建议的**ext2fs**日志记录中的批处理提交牺牲了快速提交，而倾向于一次提交多个更新，从而以延迟为代价获得吞吐量(由于缓存的影响，磁盘上的延迟对应用程序是隐藏的)。

ext2fs日志记录有两种方式更适合在**NFS**服务器上使用，一种是使用较小的事务，另一种是记录文件数据和元数据。通过调整提交到日志的事务的大小，我们可能能够显著提高提交单个更新的周转时间。**NFS**还要求尽快将数据写入提交到磁盘，原则上没有理由不扩展日志文件以覆盖正常文件数据的写入。

最后，值得注意的是，这个方案中没有任何东西会阻止我们在几个不同的文件系统中共享一个日志文件。允许多个文件系统被记录到完全为此目的保留的单独磁盘上的日志中不需要太多额外的工作，并且在有许多日志文件系统都经历高负载的情况下，这可能会大大提高性能。单独的日志磁盘将几乎完全按顺序写入，因此可以保持高吞吐量，而不会损害主文件系统磁盘上的可用带宽。

结论

本文中概述的文件系统设计应该比Linux上现有的**ext2fs**文件系统提供显著的优势。它应该通过使文件系统在崩溃后更可预测和更快地恢复来提高可用性和可靠性，并且在正常操作中不应该导致太多的性能损失。

对日常性能最重要的影响是，新创建的文件必须快速同步到磁盘，以便将创建的文件提交到日志，而不是允许内核通常支持的数据延迟写回。这可能使日志文件系统不适合在**/tmp**文件系统上使用。

设计应该只需要对现有的**ext2fs**代码库进行最小的更改：大多数功能都是由新的日志机制提供的，该机制将通过一个简单的事务缓冲区IO接口与**ext2fs**主代码交互。

最后，这里介绍的设计构建在现有**ext2fs**磁盘上文件系统布局的基础上，因此可以在现有**ext2fs**文件系统中添加事务日志，无需重新格式化文件系统就可使用这些新特性。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-24 15:01:18

- 将实验代码提交到github

将实验代码提交到github

[!DANGER] MIT 6.S081 这门课程每个lab对应一个git分支，所以请不要擅自将.git目录删除或更改origin指向的仓库地址

- (1). 首先将mit的实验代码克隆到本地

```
git clone git://g.csail.mit.edu/xv6-labs-2020
```

- (2). 在github创建一个新的空仓库

创建完成后会有提示代码，请不要根据提示代码操作，并且记下右图中红色标注的仓库地址

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?
[Import a repository.](#)

Owner *



Repository name *

xv6-labs-2020



Great repository names are short and memorable. Need inspiration? How about [crispy-train](#)?

Description (optional)

MIT 6.S081 实验代码

Public

Anyone on the internet can see this repository. You choose who can commit.

Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

Add a README file

This is where you can write a long description for your project. [Learn more](#)

Add .gitignore

Choose which files not to track from a list of templates. [Learn more](#).

Choose a license

A license tells others what they can and can't do with your code. [Learn more](#).

[Create repository](#)

Quick setup — if you've done this kind of thing before

Set up in Desktop

or

HTTPS SSH

<https://github.com/duguosheng/xv6-labs-2020.git>

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#)

...or create a new repository on the command line

```
echo "# xv6-labs-2020" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/duguosheng/xv6-labs-2020.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/duguosheng/xv6-labs-2020.git
git branch -M main
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

(3). 添加git仓库地址

查看本地仓库的git配置文件，可以看到**origin**主机名下已经有了对应的上游仓库地址

```
cd xv6-labs-2020/
cat .git/config
```

```
[dgs@manjaro Projects]$ cd xv6-labs-2020/
[dgs@manjaro xv6-labs-2020]$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = git://g.csail.mit.edu/xv6-labs-2020
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
[dgs@manjaro xv6-labs-2020]$
```

因此我们不要使用**origin**, 可以使用其他主机名对应到github仓库, 例如, 我使用**github**

```
git remote add github 你的仓库地址
cat .git/config
```

```
[dgs@manjaro xv6-labs-2020]$ git remote add github https://github.com/duguosheng/xv6-labs-2020.git
[dgs@manjaro xv6-labs-2020]$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = git://g.csail.mit.edu/xv6-labs-2020
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
[remote "github"]
    url = https://github.com/duguosheng/xv6-labs-2020.git
    fetch = +refs/heads/*:refs/remotes/github/*
[dgs@manjaro xv6-labs-2020]$
```

(4). `git push` 命令

- 功能: `git push` 命令用于将本地的分支版本上传到远程并合并。
- 命令格式:

```
git push <远程主机名> <本地分支名>:<远程分支名>
```

如果本地分支名与远程分支名相同, 则可以省略冒号:

```
git push <远程主机名> <本地分支名>
```

更多用法请自行搜索

(5). 将实验代码推送github仓库

例如: 将实验1用到的**util**分支推送到github

```
git checkout util
git push github util:util
```

需要你输入账户密码, 提交就成功了

其他实验仓库的提交同理

(6). xv6实验git分支建议

建议是每个实验创建一个测试分支，例如对于util来说

```
git checkout util      # 切换到util分支  
git checkout -b util_test # 建立并切换到util的测试分支
```

当你在util_test分支中每测试通过一个作业，请提交（git commit）你的代码，并将所做的修改合并（git merge）到util中，然后提交（git push）到github

```
git add .  
git commit -m "完成了第一个作业"  
git checkout util  
git merge util_test  
git push github util:util
```

(7). 其他

你还可以添加gitee，防止github有时无法访问的问题

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-20 09:07:03

- 实验说明
 - 实验难度
 - 调试技巧

实验说明

实验难度

每个实验都具有相应的难度

- Easy: 不到一个小时。这些锻炼通常是为后续锻炼做的热身运动。
- Moderate: 1-2小时。
- Hard: 超过2个小时。这些练习通常不需要很多代码，但是代码很难正确。

实验往往不需要很多行代码(几十到几百行)，但是代码在概念上很复杂，而且细节往往很重要。所以，在你写任何代码之前，一定要完成实验室指定的阅读，通读相关文件，查阅文档([RISC-V手册等存放在了参考页面上](#))。只有当你确定掌握了任务和解决方案，再开始编码。当你开始编写代码的时候，一小步一小步地实现你的解决方案(作业通常会建议如何将问题分解为更小的步骤)，并且在继续下一个步骤之前测试每个步骤是否正常工作。

调试技巧

确保你理解了c和指针。[Kernighan和Ritchie的《c程序设计语言》](#)一书对C语言进行了简要的描述。这里有一些有用的指针练习。除非你已经完全掌握了C语言，不要跳过或略读上面的指针练习。如果你不能真正理解C语言中的指针，你将在实验室中遭受难以言喻的痛苦，然后最终以一种艰难的方式来理解它们。相信我们，你不会想知道什么是“艰难的路”的。

一些常见的习惯用法特别值得记住：

- 如果 `int *p = (int*)100`，那么 `(int)p + 1` 及 `(int)(p + 1)` 是不同的数字，第一个是101，但第二个是104。当向指针添加一个整数时，如第二种情况，整数被隐式地乘以指针指向的对象的大小。
- `p[i]` 被定义为与 `*(p+i)` 相同，指向内存中p指向的第i个对象，当对象大于1字节时，上面所说的加法规则有利于此定义工作

虽然大多数C程序不需要在指针和整数之间进行强制转换，但操作系统经常需要这样做。每当您看到一个包含内存地址的加法时，问问自己它是整数加法还是指针加法，并确保所添加的值是否适当地相乘。

- 如果你有一个部分工作的练习，请通过提交代码来检查你的进度。如果您稍后破坏了某些东西，那么您可以回滚到您的检查点，然后以较小的步骤继续前进。要了解关于Git的更多信息，请查看[Git用户手册](#)，或者您可能会发现这个[面向计算机科学家的Git概述](#)非常有用。
- 如果您没有通过测试，确保您了解为什么您的代码没有通过测试。插入打印(`printf`)语句，直到您理解正在发生的事情。

- 您可能会发现您的print语句可能会产生许多您想要搜索的输出；其中一种方法是在script内部运行make qemu（在您的机器上运行man script），它将所有控制台输出记录到一个文件中，然后您可以搜索该文件。别忘了退出script。
- 在许多情况下，print语句就足够了，但有时能够单步遍历一些汇编代码或检查堆栈上的变量是有帮助的。要在xv6中使用gdb，请在一个窗口中运行make qemu-gdb，在另一个窗口中运行gdb（或riscv64-linux-gnu-gdb），设置断点，后跟“c”（continue），xv6将一直运行，直到到达断点。（有关有用的GDB提示，请参阅[使用GNU调试器](#)。）
- 如果要查看编译器为内核生成的程序集是什么，或者要找出特定内核地址的指令是什么，请参阅文件kernel.asm，该文件在编译内核时由Makefile生成。（Makefile同时也为所有用户程序生成.asm文件。）
- 如果内核崩溃，它将打印一条错误消息，列出崩溃时程序计数器的值；您可以进行搜索kernel.asm找出程序计数器崩溃时在哪个函数中，或者可以运行addr2line -e kernel/kernel pc-value（有关详细信息，请运行man addr2line）。如果要获取回溯，请使用gdb重新启动：在一个窗口中运行'make qemu-gdb'，在另一个窗口中运行gdb（或riscv64-linux-gnu-gdb），在panic中设置断点（“b panic”），后跟“c”（continue）。当内核到达断点时，键入“bt”以获取回溯跟踪。
- 如果您的内核挂起（例如，由于死锁）或无法进一步执行（例如，由于在执行内核指令时出现页面错误），您可以使用gdb查找挂起的位置。在一个窗口中运行“make qemu-gdb”，在另一个窗口中运行gdb（riscv64-linux-gnu-gdb），后跟“c”（continue）。当内核出现挂起时，在qemu-gdb窗口中按Ctrl-C并键入“bt”以获得回溯跟踪。
- qemu有一个“监视器”，允许您查询模拟机器的状态。您可以通过键入<Ctrl>+a c（c表示控制台）来获得它。一个特别有用的monitor命令是info mem，用于打印页表。您可能需要使用cpu命令来选择info mem查看哪一个核心，或者可以使用make CPUS=1 qemu启动qemu，以使其只有一个核心。

花时间学习上述工具是非常值得的

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间：2021-08-19 10:08:45

- [Lab1: Xv6 and Unix utilities](#)
- 实验任务
 - 启动xv6(难度: Easy)
 - sleep(难度: Easy)
 - pingpong (难度: Easy)
 - Primes(素数, 难度: Moderate/Hard)
 - find (难度: Moderate)
 - xargs (难度: Moderate)
- 提交实验
- 可选的挑战练习

Lab1: Xv6 and Unix utilities

实验任务

启动xv6(难度: Easy)

获取实验室的xv6源代码并切换到util分支

```
$ git clone git://g.csail.mit.edu/xv6-labs-2020
Cloning into 'xv6-labs-2020'...
...
$ cd xv6-labs-2020
$ git checkout util
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
```

Xv6-labs-2020存储库与本书的xv6-riscv稍有不同;它主要添加一些文件。如果你好奇的话, 可以执行 `git log`:

```
$ git log
```

您将需要使用Git版本控制系统管理和提交文件以及后续的实验室作业。接下来, 切换到一个分支(执行 `git checkout util`), 其中包含针对该实验室定制的xv6版本。要了解关于Git的更多信息, 请查看Git用户手册。Git允许您跟踪对代码所做的更改。例如, 如果你完成了其中一个练习, 并且想检查你的进度, 你可以通过运行以下命令来提交你的变化:

```
$ git commit -am 'my solution for util lab exercise 1'
Created commit 60d2135: my solution for util lab exercise 1
 1 files changed, 1 insertions(+), 0 deletions(-)
$
```

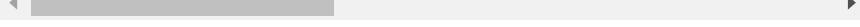
您可以使用 `git diff` 命令跟踪您的更改。运行 `git diff` 将显示自上次提交以来对代码的更改, `git diff origin/util` 将显示相对于初始xv6-labs-2020代码的更改。这里, **origin/xv6-labs-2020**是git分支的名称, 它是包含您下载的初始代码分支。

- 构建并运行xv6

```

$ make qemu
riscv64-unknown-elf-gcc -c -o kernel/entry.o kernel/entry.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -MD
...
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_zombie user/
riscv64-unknown-elf-objdump -S user/_zombie &gt; user/zombie.asm
riscv64-unknown-elf-objdump -t user/_zombie | sed '1,/SYMBOL TABLE/d; s/ .* / /; /$/d'
mkfs/mkfs fs.img README user/xargstest.sh user/_cat user/_echo user/_forktest user/_fi
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 tota
balloc: first 591 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nog
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$
```



如果你在提示符下输入 `ls`，你会看到类似如下的输出：

```

$ ls
.
.. 1 1 1024
README 2 2 2059
xargstest.sh 2 3 93
cat 2 4 24256
echo 2 5 23080
forktest 2 6 13272
grep 2 7 27560
init 2 8 23816
kill 2 9 23024
ln 2 10 22880
ls 2 11 26448
mkdir 2 12 23176
rm 2 13 23160
sh 2 14 41976
stressfs 2 15 24016
usertests 2 16 148456
grind 2 17 38144
wc 2 18 25344
zombie 2 19 22408
console 3 20 0
```

这些是 `mkfs` 在初始文件系统中包含的文件；大多数是可以运行的程序。你刚刚跑了其中一个：`ls`。

`xv6` 没有 `ps` 命令，但是如果您键入 `Ctrl-p`，内核将打印每个进程的信息。如果现在尝试，您将看到两行：一行用于 `init`，另一行用于 `sh`。

退出 `qemu`：`Ctrl-a x`。

sleep(难度：Easy)

[!TIP|label:YOUR JOB] 实现xv6的UNIX程序 `sleep`：您的 `sleep` 应该暂停到用户指定的计时数。一个滴答(tick)是由xv6内核定义的时间概念，即来自定时器芯片的两个中断之间的时间。您的解决方案应该在文件 `user/sleep.c` 中

提示：

- 在你开始编码之前，请阅读《book-riscv-rev1》的第一章
- 看看其他的一些程序（如`/user/echo.c`, `/user/grep.c`, `/user/rm.c`）查看如何获取传递给程序的命令行参数
- 如果用户忘记传递参数，`sleep` 应该打印一条错误信息
- 命令行参数作为字符串传递；您可以使用 `atoi` 将其转换为数字（详见`/user/ulib.c`）
- 使用系统调用 `sleep`
- 请参阅`kernel/sysproc.c`以获取实现 `sleep` 系统调用的xv6内核代码（查找 `sys_sleep`），`user/user.h` 提供了 `sleep` 的声明以便其他程序调用，用汇编程序编写的`user/usys.S`可以帮助 `sleep` 从用户区跳转到内核区。
- 确保 `main` 函数调用 `exit()` 以退出程序。
- 将你的 `sleep` 程序添加到`Makefile`中的 `UPROGS` 中；完成之后，`make qemu` 将编译您的程序，并且您可以从xv6的shell运行它。
- 看看Kernighan和Ritchie编著的《C程序设计语言》（第二版）来了解C语言。

从xv6 shell运行程序：

```
$ make qemu
...
init: starting sh
$ sleep 10
(nothing happens for a little while)
$
```

如果程序在如上所示运行时暂停，则解决方案是正确的。运行 `make grade` 看看您是否真的通过了睡眠测试。

请注意，`make grade` 运行所有测试，包括下面作业的测试。如果要对一项作业运行成绩测试，请键入（不要启动XV6，在外部终端下使用）：

```
$ ./grade-lab-util sleep
```

这将运行与 `sleep` 匹配的成绩测试。或者，您可以键入：

```
$ make GRADEFLAGS=sleep grade
```

效果是一样的。

pingpong (难度: Easy)

[!TIP|label:YOUR JOB] 编写一个使用**UNIX**系统调用的程序来在两个进程之间“**ping-pong**”一个字节，请使用两个管道，每个方向一个。父进程应该向子进程发送一个字节；子进程应该打印“`<pid>: received ping`”，其中`<pid>`是进程ID，并在管道中写入字节发送给父进程，然后退出；父级应该从读取从子进程而来的字节，打印“`<pid>: received pong`”，然后退出。您的解决方案应该在文件**`user/pingpong.c`**中。

提示：

- 使用 `pipe` 来创造管道
- 使用 `fork` 创建子进程
- 使用 `read` 从管道中读取数据，并且使用 `write` 向管道中写入数据
- 使用 `getpid` 获取调用进程的pid
- 将程序加入到**Makefile**的 `UPROGS`
- xv6上的用户程序有一组有限的可用库函数。您可以在**`user/user.h`**中看到可调用的程序列表；源代码（系统调用除外）位于**`user/ulib.c`**、**`user/printf.c`**和**`user/umalloc.c`**中。

运行程序应得到下面的输出

```
$ make qemu
...
init: starting sh
$ pingpong
4: received ping
3: received pong
$
```

如果您的程序在两个进程之间交换一个字节并产生如上所示的输出，那么您的解决方案是正确的。

Primes(素数， 难度： Moderate/Hard)

[!TIP|label:YOUR JOB] 使用管道编写 `prime sieve` (筛选素数)的并发版本。这个想法是由**Unix**管道的发明者**Doug McIlroy**提出的。请查看[这个网站](#)(翻译在下面)，该网页中间的图片和周围的文字解释了如何做到这一点。您的解决方案应该在**`user/primes.c`**文件中。

您的目标是使用 `pipe` 和 `fork` 来设置管道。第一个进程将数字2到35输入管道。对于每个素数，您将安排创建一个进程，该进程通过一个管道从其左邻居读取数据，并通过另一个管道向其右邻居写入数据。由于xv6的文件描述符和进程数量有限，因此第一个进程可以在35处停止。

提示：

- 请仔细关闭进程不需要的文件描述符，否则您的程序将在第一个进程达到35之前就会导致xv6系统资源不足。

- 一旦第一个进程达到35，它应该使用 `wait` 等待整个管道终止，包括所有子孙进程等等。因此，主 `primes` 进程应该只在打印完所有输出之后，并且在所有其他 `primes` 进程退出之后退出。
- 提示：当管道的 `write` 端关闭时，`read` 返回零。
- 最简单的方法是直接将32位（4字节）`int`写入管道，而不是使用格式化的 ASCII I/O。
- 您应该仅在需要时在管线中创建进程。
- 将程序添加到**Makefile**中的 `UPROGS`

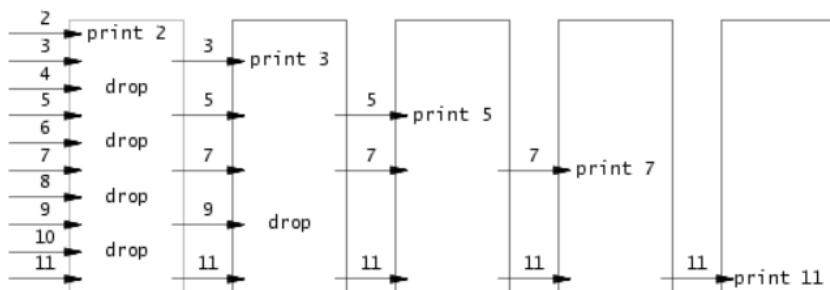
如果您的解决方案实现了基于管道的筛选并产生以下输出，则是正确的：

```
$ make qemu
...
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$
```

参考资料翻译：

考虑所有小于1000的素数的生成。Eratosthenes的筛选法可以通过执行以下伪代码的进程管线来模拟：

```
p = get a number from left neighbor
print p
loop:
    n = get a number from left neighbor
    if (p does not divide n)
        send n to right neighbor
p = 从左邻居中获取一个数
print p
loop:
    n = 从左邻居中获取一个数
    if (n不能被p整除)
        将n发送给右邻居
```



生成进程可以将数字2、3、4、...、1000输入管道的左端：行中的第一个进程消除2的倍数，第二个进程消除3的倍数，第三个进程消除5的倍数，依此类推。

find (难度: Moderate)

[!TIP|label:YOUR JOB] 写一个简化版本的UNIX的 `find` 程序：查找目录树中具有特定名称的所有文件，你的解决方案应该放在 `user/find.c`

提示：

- 查看 `user/ls.c` 文件学习如何读取目录
- 使用递归允许 `find` 下降到子目录中
- 不要在“.”和“..”目录中递归
- 对文件系统的更改会在 `qemu` 的运行过程中一直保持；要获得一个干净的文件系统，请运行 `make clean`，然后 `make qemu`
- 你将会使用到C语言的字符串，要学习它请看《C程序设计语言》（K&R），例如第5.5节
- 注意在C语言中不能像 `python` 一样使用“==”对字符串进行比较，而应当使用 `strcmp()`
- 将程序加入到 `Makefile` 的 `UPROGS`

如果你的程序输出下面的内容，那么它是正确的（当文件系统中包含文件 `b` 和 `a/b` 的时候）

```
$ make qemu
...
init: starting sh
$ echo > b
$ mkdir a
$ echo > a/b
$ find . b
./b
./a/b
$
```

xargs (难度: Moderate)

[!TIP|label:YOUR JOB] 编写一个简化版UNIX的 `xargs` 程序：它从标准输入中按行读取，并且为每一行执行一个命令，将行作为参数提供给命令。你的解决方案应该在 `user/xargs.c`

下面的例子解释了 `xargs` 的行为

```
$ echo hello too | xargs echo bye
bye hello too
$
```

注意，这里的命令是 `echo bye`，额外的参数是 `hello too`，这样就组成了命令 `echo bye hello too`，此命令输出 `bye hello too`

请注意，**UNIX**上的 `xargs` 进行了优化，一次可以向该命令提供更多的参数。我们不需要您进行此优化。要使**UNIX**上的 `xargs` 表现出本实验所实现的方式，请将 `-n` 选项设置为1。例如

```
$ echo "1\n2" | xargs -n 1 echo line
line 1
line 2
$
```

提示：

- 使用 `fork` 和 `exec` 对每行输入调用命令，在父进程中使用 `wait` 等待子进程完成命令。
- 要读取单个输入行，请一次读取一个字符，直到出现换行符（'\n'）。
- **kernel/param.h** 声明 `MAXARG`，如果需要声明 `argv` 数组，这可能很有用。
- 将程序添加到**Makefile**中的 `UPROGS`。
- 对文件系统的更改会在**qemu**的运行过程中保持不变；要获得一个干净的文件系统，请运行 `make clean`，然后 `make qemu`

`xargs`、`find` 和 `grep` 结合得很好

```
$ find . b | xargs grep hello
```

将对“.”下面的目录中名为**b**的每个文件运行 `grep hello`。

要测试您的 `xargs` 方案是否正确，请运行**shell**脚本**xargstest.sh**。如果您的解决方案产生以下输出，则是正确的：

```
$ make qemu
...
init: starting sh
$ sh < xargstest.sh
$ $ $ $ $ hello
hello
hello
$ $
```

你可能不得不回去修复你的 `find` 程序中的bug。输出有许多 `$`，因为**xv6 shell**没有意识到它正在处理来自文件而不是控制台的命令，并为文件中的每个命令打印 `$`。

提交实验

这就完成了实验。确保你通过了所有的成绩测试。如果这个实验有问题，别忘了把你的答案写在**answers-lab-name.txt**中。提交你的更改（包括**answers-lab-name.txt**），然后在实验目录中键入 `make handin` 以提交实验。

花费的时间

创建一个命名为**time.txt**的新文件，并在其中输入一个整数，即您在实验室花费的小时数。不要忘记 `git add` 和 `git commit` 文件。

提交

你将使用[提交网站](#)提交作业。您需要从提交网站请求一次API密钥，然后才能提交任何作业或实验。

将最终更改提交到实验后，键入`make handin`以提交实验。

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting https://6828.scripts.mit.edu/2020/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXX
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload   Total Spent    Left  Speed
100 79258  100    239  100 79019    853   275k --::-- --::-- --::--  276k
$
```

`make handin`将把你的API密钥存储在`myapi.key`中。如果需要更改API密钥，只需删除此文件并让`make handin`再次生成它(`myapi.key`不得包含换行符)。

如果你运行了`make handin`，并且你有未提交的更改或未跟踪的文件，则会看到类似于以下内容的输出：

```
M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in. Continue? [y/N]
```

检查上述行，确保跟踪了您的实验解决方案所需的所有文件，即以`??`开头的行中所显示的文件。您可以使用`git add filename`命令使git追踪创建的新文件。

如果`make handin`无法正常工作，请尝试使用curl或Git命令修复该问题。或者你可以运行`make tarball`。这将为您制作一个tar文件，然后您可以通过我们的web界面上传。

- 请运行“`make grade`”以确保您的代码通过所有测试
- 在运行“`make handin`”之前提交任何修改过的源代码`
- 您可以检查提交的状态，并在以下位置下载提交的代码:

<https://6828.scripts.mit.edu/2020/handin.py/>

可选的挑战练习

- 编写一个`uptime`程序，使用`uptime`系统调用以滴答为单位打印计算机正常运行时间。（easy）
- 在`find`程序的名称匹配中支持正则表达式。`grep.c`对正则表达式有一些基本的支持。（easy）
- `xv6 shell`（`user/sh.c`）只是另一个用户程序，您可以对其进行改进。它是一个最小的shell，缺少建立在真实shell中的许多特性。例如，
 - 在处理文件中的shell命令时，将shell修改为不打印\$（moderate）
 - 将shell修改为支持`wait`（easy）

- 将shell修改为支持用“ ; ”分隔的命令列表（**moderate**）
 - 通过实现在左括号“ (”以及右括号“) ”来修改shell以支持子shell（**moderate**）
 - 将shell修改为支持 tab 键补全（**easy**）
 - 修改shell使其支持命令历史记录（**moderate**）
 - 或者您希望shell执行的任何其他操作。
- 如果您非常雄心勃勃，可能需要修改内核以支持所需的内核特性；xv6支持的并不多。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 13:45:54

- [Lab2: system calls](#)
- [System call tracing \(moderate\)](#)
- [Sysinfo \(moderate\)](#)
- [可选的挑战](#)

Lab2: system calls

在上一个实验室中，您使用系统调用编写了一些实用程序。在本实验室中，您将向 xv6 添加一些新的系统调用，这将帮助您了解它们是如何工作的，并使您了解 xv6 内核的一些内部结构。您将在以后的实验室中添加更多系统调用。

[!WARNING|label:Attention] 在你开始写代码之前，请阅读 xv6 手册《book-riscv-rev1》的第2章、第4章的第4.3节和第4.4节以及相关源代码文件：

- 系统调用的用户空间代码在 `user/user.h` 和 `user/usys.pl` 中。
- 内核空间代码是 `kernel/syscall.h`、`kernel/syscall.c`。
- 与进程相关的代码是 `kernel/proc.h` 和 `kernel/proc.c`。

要开始本章实验，请将代码切换到 `syscall` 分支：

```
$ git fetch  
$ git checkout syscall  
$ make clean
```

如果运行 `make grade`，您将看到测试分数的脚本无法执行 `trace` 和 `sysinfotest`。您的工作是添加必要的系统调用和存根（stubs）以使它们工作。

System call tracing (moderate)

[!TIP|label:YOUR JOB] 在本作业中，您将添加一个系统调用跟踪功能，该功能可能会在以后调试实验时对您有所帮助。您将创建一个新的 `trace` 系统调用来控制跟踪。它应该有一个参数，这个参数是一个整数“掩码”（mask），它的比特位指定要跟踪的系统调用。例如，要跟踪 `fork` 系统调用，程序调用 `trace(1 << SYS_fork)`，其中 `SYS_fork` 是 `kernel/syscall.h` 中的系统调用编号。如果在掩码中设置了系统调用的编号，则必须修改 xv6 内核，以便在每个系统调用即将返回时打印出一行。该行应该包含进程id、系统调用的名称和返回值；您不需要打印系统调用参数。`trace` 系统调用应启用对调用它的进程及其随后派生的任何子进程的跟踪，但不应影响其他进程。

我们提供了一个用户级程序版本的 `trace`，它运行另一个启用了跟踪的程序（参见 `user/trace.c`）。完成后，您应该看到如下输出：

```

$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
$
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
$
$ grep hello README
$
$ trace 2 usertests forkforkfork
usertests starting
test forkforkfork: 407: syscall fork -> 408
408: syscall fork -> 409
409: syscall fork -> 410
410: syscall fork -> 411
409: syscall fork -> 412
410: syscall fork -> 413
409: syscall fork -> 414
411: syscall fork -> 415
...
$
```

在上面的第一个例子中，`trace` 调用 `grep`，仅跟踪了 `read` 系统调用。`32` 是 `1<<SYS_read`。在第二个示例中，`trace` 在运行 `grep` 时跟踪所有系统调用；`2147483647` 将所有31个低位置为1。在第三个示例中，程序没有被跟踪，因此没有打印跟踪输出。在第四个示例中，在 `usertests` 中测试的 `forkforkfork` 中所有子孙进程的 `fork` 系统调用都被追踪。如果程序的行为如上所示，则解决方案是正确的（尽管进程ID可能不同）

提示：

- 在 **Makefile** 的 **UPROGS** 中添加 `$U/_trace`
- 运行 `make qemu`，您将看到编译器无法编译 **user/trace.c**，因为系统调用的用户空间存根还不存在：将系统调用的原型添加到 **user/user.h**，存根添加到 **user/usys.pl**，以及将系统调用编号添加到 **kernel/syscall.h**，**Makefile** 调用 perl脚本 **user/usys.pl**，它生成实际的系统调用存根 **user/usys.S**，这个文件中的汇编代码使用 RISC-V 的 `ecall` 指令转换到内核。一旦修复了编译问题（注：如果编译还未通过，尝试先 `make clean`，再执行 `make qemu`），就运行 `trace 32 grep hello README`；但由于您还没有在内核中实现系统调用，执行将失败。
- 在 **kernel/sysproc.c** 中添加一个 `sys_trace()` 函数，它通过将参数保存到 `proc` 结构体（请参见 **kernel/proc.h**）里的一个新变量中来实现新的系统调用。从用户空间检索系统调用参数的函数在 **kernel/syscall.c** 中，您可以在 **kernel/sysproc.c** 中看到它们的使用示例。
- 修改 `fork()`（请参阅 **kernel/proc.c**）将跟踪掩码从父进程复制到子进程。
- 修改 **kernel/syscall.c** 中的 `syscall()` 函数以打印跟踪输出。您将需要添加一个系统调用名称数组以建立索引。

Sysinfo (moderate)

[!TIP|label:YOUR JOB] 在这个作业中，您将添加一个系统调用 `sysinfo`，它收集有关正在运行的系统的信息。系统调用采用一个参数：一个指向 `struct sysinfo` 的指针（参见 `kernel/sysinfo.h`）。内核应该填写这个结构的字段：`freemem` 字段应该设置为空闲内存的字节数，`nproc` 字段应该设置为 `state` 字段不为 `UNUSED` 的进程数。我们提供了一个测试程序 `sysinfotest`；如果输出“`sysinfotest: OK`”则通过。

提示：

- 在 `Makefile` 的 `UPROGS` 中添加 `$U/_sysinfotest`
- 当运行 `make qemu` 时，`user/sysinfotest.c` 将会编译失败，遵循和上一个作业一样的步骤添加 `sysinfo` 系统调用。要在 `user/user.h` 中声明 `sysinfo()` 的原型，需要预先声明 `struct sysinfo` 的存在：

```
struct sysinfo;
int sysinfo(struct sysinfo *);
```

一旦修复了编译问题，就运行 `sysinfotest`；但由于您还没有在内核中实现系统调用，执行将失败。

- `sysinfo` 需要将一个 `struct sysinfo` 复制回用户空间；请参考 `sys_fstat()` (`kernel/sysfile.c`) 和 `filestat()` (`kernel/file.c`) 以获取如何使用 `copyout()` 执行此操作的示例。
- 要获取空闲内存量，请在 `kernel/kalloc.c` 中添加一个函数
- 要获取进程数，请在 `kernel/proc.c` 中添加一个函数

可选的挑战

- 打印所跟踪的系统调用的参数（`easy`）。
- 计算平均负载并通过 `sysinfo` 导出（`moderate`）。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2022-04-09 12:36:00

- [Lab3: page tables](#)
- [Print a page table \(easy\)](#)
- [A kernel page table per process \(hard\)](#)
- [Simplify copyin/copyinstr \(hard\)](#)
- 可选的挑战练习

Lab3: page tables

在本实验室中，您将探索页表并对其进行修改，以简化将数据从用户空间复制到内核空间的函数。

[!WARNING|label:Attention] 开始编码之前，请阅读xv6手册的第3章和相关文件：

- *kernel/memlayout.h*, 它捕获了内存的布局。
- *kernel/vm.c*, 其中包含大多数虚拟内存（VM）代码。
- *kernel/kalloc.c*, 它包含分配和释放物理内存的代码。

要启动实验，请切换到pgtbl分支：

```
$ git fetch
$ git checkout pgtbl
$ make clean
```

Print a page table (easy)

为了帮助您了解RISC-V页表，也许为了帮助将来的调试，您的第一个任务是编写一个打印页表内容的函数。

[!TIP|label:YOUR JOB] 定义一个名为 `vmprint()` 的函数。它应当接收一个 `pagetable_t` 作为参数，并以下面描述的格式打印该页表。在 `exec.c` 中的 `return argc` 之前插入 `if(p->pid==1) vmprint(p->pagetable)`，以打印第一个进程的页表。如果你通过了 `pte printout` 测试的 `make grade`，你将获得此作业的满分。

现在，当您启动xv6时，它应该像这样打印输出来描述第一个进程刚刚完成 `exec()` 时的页表：

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
... .0: pte 0x0000000021fda401 pa 0x0000000087f69000
... ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
... ...1: pte 0x0000000021fda00f pa 0x0000000087f68000
... ...2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
... .511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
... ...510: pte 0x0000000021fdd807 pa 0x0000000087f76000
... ...511: pte 0x0000000020001c0b pa 0x0000000080007000
```

第一行显示 `vmprint` 的参数。之后的每行对应一个PTE，包含树中指向页表页的 PTE。每个PTE行都有一些“...”的缩进表明它在树中的深度。每个PTE行显示其在页表页中的PTE索引、PTE比特位以及从PTE提取的物理地址。不要打印无效的 PTE。在上面的示例中，顶级页表页具有条目0和255的映射。条目0的下一级只映射了索引0，该索引0的下一级映射了条目0、1和2。

您的代码可能会发出与上面显示的不同的物理地址。条目数和虚拟地址应相同。

一些提示：

- 你可以将 `vmprint()` 放在 `kernel/vm.c` 中
- 使用定义在 `kernel/riscv.h` 末尾处的宏
- 函数 `freewalk` 可能会对你有所启发
- 将 `vmprint` 的原型定义在 `kernel/defs.h` 中，这样你就可以在 `exec.c` 中调用它了
- 在你的 `printf` 调用中使用 `%p` 来打印像上面示例中的完成的64比特的十六进制 PTE 和地址

[!NOTE|label:QUESTION] 根据文本中的图3-4解释 `vmprint` 的输出。page 0 包含什么？page 2 中是什么？在用户模式下运行时，进程是否可以读取/写入 page 1 映射的内存？

A kernel page table per process (hard)

Xv6有一个单独的用于在内核中执行程序时的内核页表。内核页表直接映射（恒等映射）到物理地址，也就是说内核虚拟地址 `x` 映射到物理地址仍然是 `x`。Xv6还为每个进程的用户地址空间提供了一个单独的页表，只包含该进程用户内存的映射，从虚拟地址0开始。因为内核页表不包含这些映射，所以用户地址在内核中无效。因此，当内核需要使用在系统调用中传递的用户指针（例如，传递给 `write()` 的缓冲区指针）时，内核必须首先将指针转换为物理地址。本节和下一节的目标是允许内核直接解引用用户指针。

[!TIP|label:YOUR JOB] 你的第一项工作是修改内核来让每一个进程在内核中执行时使用它自己的内核页表的副本。修改 `struct proc` 来为每一个进程维护一个内核页表，修改调度程序使得切换进程时也切换内核页表。对于这个步骤，每个进程的内核页表都应当与现有的全局内核页表完全一致。如果你的 `usertests` 程序正确运行了，那么你就通过了这个实验。

阅读本作业开头提到的章节和代码；了解虚拟内存代码的工作原理后，正确修改虚拟内存代码将更容易。页表设置中的错误可能会由于缺少映射而导致陷阱，可能会导致加载和存储影响到意料之外的物理页存页面，并且可能会导致执行来自错误内存页的指令。

提示：

- 在 `struct proc` 中为进程的内核页表增加一个字段
- 为一个新进程生成一个内核页表的合理方案是实现一个修改版的 `kvminit`，这个版本中应当创造一个新的页表而不是修改 `kernel_pagetable`。你将会考虑在 `allocproc` 中调用这个函数
- 确保每一个进程的内核页表都关于该进程的内核栈有一个映射。在未修改的 XV6 中，所有的内核栈都在 `procinit` 中设置。你将要把这个功能部分或全部的

迁移到 `allocproc` 中

- 修改 `scheduler()` 来加载进程的内核页表到核心的 `satp` 寄存器(参阅 `kvminitart` 来获取启发)。不要忘记在调用完 `w_satp()` 后调用 `sfence_vma()`
- 没有进程运行时 `scheduler()` 应当使用 `kernel_pagetable`
- 在 `freeproc` 中释放一个进程的内核页表
- 你需要一种方法来释放页表，而不必释放叶子物理内存页面。
- 调式页表时，也许 `vmprint` 能派上用场
- 修改XV6本来的函数或新增函数都是允许的；你或许至少需要在 `kernel/vm.c` 和 `kernel/proc.c` 中这样做（但不要修改 `kernel/vmcopyin.c`, `kernel/stats.c`, `user/usertests.c`, 和 `user/stats.c`）
- 页表映射丢失很可能导致内核遭遇页面错误。这将导致打印一段包含 `sepc=0x00000000XXXXXXXX` 的错误提示。你可以在 `kernel/kernel.asm` 通过查询 `XXXXXXXX` 来定位错误。

Simplify `copyin` / `copyinstr` (hard)

内核的 `copyin` 函数读取用户指针指向的内存。它通过将用户指针转换为内核可以直接解引用的物理地址来实现这一点。这个转换是通过在软件中遍历进程页表来执行的。在本部分的实验中，您的工作是将用户空间的映射添加到每个进程的内核页表（上一节中创建），以允许 `copyin`（和相关的字符串函数 `copyinstr`）直接解引用用户指针。

[!TIP|label:YOUR JOB] 将定义在 `kernel/vm.c` 中的 `copyin` 的主题内容替换为对 `copyin_new` 的调用（在 `kernel/vmcopyin.c` 中定义）；对 `copyinstr` 和 `copyinstr_new` 执行相同的操作。为每个进程的内核页表添加用户地址映射，以便 `copyin_new` 和 `copyinstr_new` 工作。如果 `usertests` 正确运行并且所有 `make grade` 测试都通过，那么你就完成了此项作业。

此方案依赖于用户的虚拟地址范围不与内核用于自身指令和数据的虚拟地址范围重叠。Xv6 使用从零开始的虚拟地址作为用户地址空间，幸运的是内核的内存从更高的地址开始。然而，这个方案将用户进程的最大大小限制为小于内核的最低虚拟地址。内核启动后，在 XV6 中该地址是 `0xc000000`，即 PLIC 寄存器的地址；请参见 `kernel/vm.c` 中的 `kvminit()`、`kernel/memlayout.h` 和文中的图 3-4。您需要修改 `xv6`，以防止用户进程增长到超过 PLIC 的地址。

一些提示：

- 先用对 `copyin_new` 的调用替换 `copyin()`，确保正常工作后再去修改 `copyinstr`
- 在内核更改进程的用户映射的每一处，都以相同的方式更改进程的内核页表。包括 `fork()`, `exec()`, 和 `sbrk()`。
- 不要忘记在 `userinit` 的内核页表中包含第一个进程的用户页表
- 用户地址的 PTE 在进程的内核页表中需要什么权限？（在内核模式下，无法访问设置了 `PTE_U` 的页面）
- 别忘了上面提到的 PLIC 限制

Linux 使用的技术与您已经实现的技术类似。直到几年前，许多内核在用户和内核空间中都为当前进程使用相同的自身进程页表，并为用户和内核地址进行映射以避免在用户和内核空间之间切换时必须切换页表。然而，这种设置允许边信道攻击，如 Meltdown 和 Spectre。

[!NOTE|label:QUESTION] 解释为什么在 `copyin_new()` 中需要第三个测试 `srcva + len < srcva`：给出 `srcva` 和 `len` 值的例子，这样的值将使前两个测试为假（即它们不会导致返回-1），但是第三个测试为真（导致返回-1）。

可选的挑战练习

- 使用超级页来减少页表中PTE的数量
- 扩展您的解决方案以支持尽可能大的用户程序；也就是说，消除用户程序小于PLIC的限制
- 取消映射用户进程的第一页，以便使对空指针的解引用将导致错误。用户文本段必须从非0处开始，例如4096

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 16:08:02

- [Lab4: traps](#)
- [RISC-V assembly \(easy\)](#)
- [Backtrace\(moderate\)](#)
- [Alarm\(Hard\)](#)
 - `test0: invoke handler`(调用处理程序)
 - `test1/test2(): resume interrupted code`(恢复被中断的代码)
- [可选的挑战练习](#)

Lab4: traps

本实验探索如何使用陷阱实现系统调用。您将首先使用栈做一个热身练习，然后实现一个用户级陷阱处理的示例。

[!WARNING|label:Attention] 开始编码之前，请阅读xv6手册的第4章和相关源文件：

- ***kernel/trampoline.S***: 涉及从用户空间到内核空间再到内核空间的转换的程序集
- ***kernel/trap.c***: 处理所有中断的代码

要启动实验，请切换到 `traps` 分支：

```
$ git fetch
$ git checkout traps
$ make clean
```

RISC-V assembly (easy)

理解一点RISC-V汇编是很重要的，你应该在6.004中接触过。xv6仓库中有一个文件[**user/call.c**](#)。执行 `make fs.img` 编译它，并在[**user/call.asm**](#)中生成可读的汇编版本。

阅读[**call.asm**](#)中函数 `g`、`f` 和 `main` 的代码。RISC-V的使用手册在[参考页](#)上。以下是您应该回答的一些问题（将答案存储在[**answers-traps.txt**](#)文件中）：

1. 哪些寄存器保存函数的参数？例如，在 `main` 对 `printf` 的调用中，哪个寄存器保存13？
2. `main` 的汇编代码中对函数 `f` 的调用在哪里？对 `g` 的调用在哪里(提示：编译器可能会将函数内联)
3. `printf` 函数位于哪个地址？
4. 在 `main` 中 `printf` 的 `jalr` 之后的寄存器 `ra` 中有什么值？
5. 运行以下代码。

```
unsigned int i = 0x000646c72;
printf("H%w Wo%s", 57616, &i);
```

程序的输出是什么？这是将字节映射到字符的[ASCII码表](#)。

输出取决于RISC-V小端存储的事实。如果RISC-V是大端存储，为了得到相同的输出，你会把 `i` 设置成什么？是否需要将 `57616` 更改为其他值？

这里有一个小端和大端存储的描述和一个更异想天开的描述。

1. 在下面的代码中，“`y=`”之后将打印什么(注：答案不是一个特定的值)？为什么会发生这种情况？

```
printf("x=%d y=%d", 3);
```

Backtrace(moderate)

回溯(Backtrace)通常对于调试很有用：它是一个存放于栈上用于指示错误发生位置的函数调用列表。

在 `kernel/printf.c` 中实现名为 `backtrace()` 的函数。在 `sys_sleep` 中插入一个对此函数的调用，然后运行 `bttest`，它将会调用 `sys_sleep`。你的输出应该如下所示：

```
backtrace:  
0x0000000080002cda  
0x0000000080002bb6  
0x0000000080002898
```

在 `bttest` 退出qemu后。在你的终端：地址或许会稍有不同，但如果你运行 `addr2line -e kernel/kernel`（或 `riscv64-unknown-elf-addr2line -e kernel/kernel`），并将上面的地址剪切粘贴如下：

```
$ addr2line -e kernel/kernel  
0x0000000080002de2  
0x0000000080002f4a  
0x0000000080002bfc  
Ctrl-D
```

你应该看到类似下面的输出：

```
kernel/sysproc.c:74  
kernel/syscall.c:224  
kernel/trap.c:85
```

编译器向每一个栈帧中放置一个帧指针（frame pointer）保存调用者帧指针的地址。你的 `backtrace` 应当使用这些帧指针来遍历栈，并在每个栈帧中打印保存的返回地址。

提示：

- 在 `kernel/defs.h` 中添加 `backtrace` 的原型，那样你就能在 `sys_sleep` 中引用 `backtrace`
- GCC编译器将当前正在执行的函数的帧指针保存在 `s0` 寄存器，将下面的函数添加到 `kernel/riscv.h`

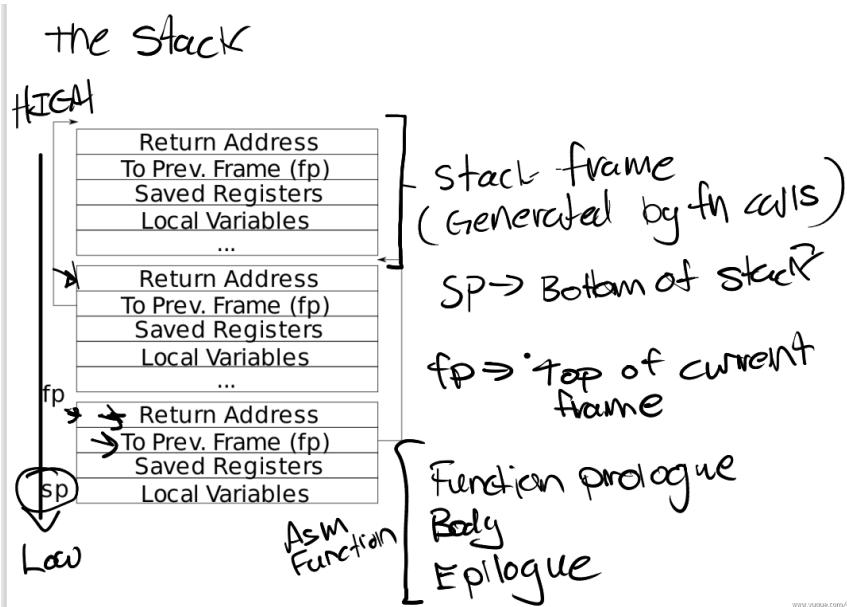
```

static inline uint64
r_fp()
{
    uint64 x;
    asm volatile("mv %0, $0" : "=r" (x) );
    return x;
}

```

并在 `backtrace` 中调用此函数来读取当前的帧指针。这个函数使用[内联汇编](#)来读取 `s0`

- 这个[课堂笔记](#)中有张栈帧布局图。注意返回地址位于栈帧帧指针的固定偏移(-8)位置，并且保存的帧指针位于帧指针的固定偏移(-16)位置



- XV6在内核中以页面对齐的地址为每个栈分配一个页面。你可以通过 `PGRONDOW(fp)` 和 `PGRONDUP(fp)`（参见[kernel/riscv.h](#)）来计算栈页面的顶部和底部地址。这些数字对于 `backtrace` 终止循环是有帮助的。

一旦你的 `backtrace` 能够运行，就在[kernel/printf.c](#)的 `panic` 中调用它，那样你就可以在 `panic` 发生时看到内核的 `backtrace`。

Alarm(Hard)

[!TIP|label:YOUR JOB] 在这个练习中你将向XV6添加一个特性，在进程使用CPU的时间内，XV6定期向进程发出警报。这对于那些希望限制CPU时间消耗的受计算限制的进程，或者对于那些计算的同时执行某些周期性操作的进程可能很有用。更普遍的来说，你将实现用户级中断/故障处理程序的一种初级形式。例如，你可以在应用程序中使用类似的一些东西处理页面故障。如果你的解决方案通过了 `alarmtest` 和 `usertests` 就是正确的。

你应当添加一个新的 `sigalarm(interval, handler)` 系统调用，如果一个程序调用了 `sigalarm(n, fn)`，那么每当程序消耗了CPU时间达到n个“滴答”，内核应当使应用程序函数 `fn` 被调用。当 `fn` 返回时，应用应当在它离开的地方恢复执行。在XV6

中，一个滴答是一段相当任意的时间单元，取决于硬件计时器生成中断的频率。如果一个程序调用了 `sigalarm(0, 0)`，系统应当停止生成周期性的报警调用。

你将在XV6的存储库中找到名为 `user/alarmtest.c` 的文件。将其添加到 `Makefile`。

注意：你必须添加了 `sigalarm` 和 `sigreturn` 系统调用后才能正确编译（往下看）。

`alarmtest` 在 `test0` 中调用了 `sigalarm(2, periodic)` 来要求内核每隔两个滴答强制调用 `periodic()`，然后旋转一段时间。你可以在 `user/alarmtest.asm` 中看到 `alarmtest` 的汇编代码，这或许会便于调试。当 `alarmtest` 产生如下输出并且 `usertests` 也能正常运行时，你的方案就是正确的：

```
$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
..alarm!
test1 passed
test2 start
.....alarm!
test2 passed
$ usertests
...
ALL TESTS PASSED
$
```

当你完成后，你的方案也许仅有几行代码，但如何正确运行是一个棘手的问题。我们将使用原始存储库中的 `alarmtest.c` 版本测试您的代码。你可以修改 `alarmtest.c` 来帮助调试，但是要确保原来的 `alarmtest` 显示所有的测试都通过了。

test0: invoke handler(调用处理程序)

首先修改内核以跳转到用户空间中的报警处理程序，这将导致 `test0` 打印“`alarm!`”。不用担心输出“`alarm!`”之后会发生什么；如果您的程序在打印“`alarm!`”后崩溃，对于目前来说也是正常的。以下是一些提示：

- 您需要修改 `Makefile` 以使 `alarmtest.c` 被编译为 xv6 用户程序。
- 放入 `user/user.h` 的正确声明是：

```
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);
```

- 更新 `user/usys.pl`（此文件生成 `user/usys.S`）、`kernel/syscall.h` 和 `kernel/syscall.c` 以允许 `alarmtest` 调用 `sigalarm` 和 `sigreturn` 系统调用。
- 目前来说，你的 `sys_sigreturn` 系统调用返回应该是零。

- 你的 `sys_sigalarm()` 应该将报警间隔和指向处理程序函数的指针存储在 `struct proc` 的新字段中（位于 `kernel/proc.h`）。
- 你也需要在 `struct proc` 新增一个新字段。用于跟踪自上一次调用（或直到下一次调用）到进程的报警处理程序间经历了多少滴答；您可以在 `proc.c` 的 `allocproc()` 中初始化 `proc` 字段。
- 每一个滴答声，硬件时钟就会强制一个中断，这个中断在 `kernel/trap.c` 中的 `usertrap()` 中处理。
- 如果产生了计时器中断，您只想操纵进程的报警滴答；你需要写类似下面的代码

```
if(which_dev == 2) ...
```

- 仅当进程有未完成的计时器时才调用报警函数。请注意，用户报警函数的地址可能是0（例如，在 `user/alarmtest.asm` 中，`periodic` 位于地址0）。
- 您需要修改 `usertrap()`，以便当进程的报警间隔期满时，用户进程执行处理程序函数。当RISC-V上的陷阱返回到用户空间时，什么决定了用户空间代码恢复执行的指令地址？
- 如果您告诉 `qemu` 只使用一个CPU，那么使用 `gdb` 查看陷阱会更容易，这可以通过运行

```
make CPUS=1 qemu-gdb
```

- 如果 `alarmtest` 打印“alarm!”，则您已成功。

test1/test2(): resume interrupted code(恢复被中断的代码)

`alarmtest` 打印“alarm!”后，很可能会在 `test0` 或 `test1` 中崩溃，或者 `alarmtest`（最后）打印“`test1 failed`”，或者 `alarmtest` 未打印“`test1 passed`”就退出。要解决此问题，必须确保完成报警处理程序后返回到用户程序最初被计时器中断的指令执行。必须确保寄存器内容恢复到中断时的值，以便用户程序在报警后可以不受干扰地继续运行。最后，您应该在每次报警计数器关闭后“重新配置”它，以便周期性地调用处理程序。

作为一个起始点，我们为您做了一个设计决策：用户报警处理程序需要在完成后调用 `sigreturn` 系统调用。请查看 `alarmtest.c` 中的 `periodic` 作为示例。这意味着您可以将代码添加到 `usertrap` 和 `sys_sigreturn` 中，这两个代码协同工作，以使用户进程在处理完警报后正确恢复。

提示：

- 您的解决方案将要求您保存和恢复寄存器——您需要保存和恢复哪些寄存器才能正确恢复中断的代码？（提示：会有很多）
- 当计时器关闭时，让 `usertrap` 在 `struct proc` 中保存足够的状态，以使 `sigreturn` 可以正确返回中断的用户代码。

- 防止对处理程序的重复调用——如果处理程序还没有返回，内核就不应该再次调用它。`test2` 测试这个。
- 一旦通过 `test0`、`test1` 和 `test2`，就运行 `usertests` 以确保没有破坏内核的任何其他部分。

可选的挑战练习

- 在 `backtrace()` 中打印函数的名称和行号，而不仅仅是数字化的地址。(hard)

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 16:07:51

- [Lab5: xv6 lazy page allocation](#)
- [Eliminate allocation from sbrk\(\) \(easy\)](#)
- [Lazy allocation \(moderate\)](#)
- [Lazytests and Usertests \(moderate\)](#)
- 可选的挑战练习

Lab5: xv6 lazy page allocation

操作系统可以使用页表硬件的技巧之一是延迟分配用户空间堆内存（**lazy allocation of user-space heap memory**）。Xv6应用程序使用 `sbrk()` 系统调用向内核请求堆内存。在我们给出的内核中，`sbrk()` 分配物理内存并将其映射到进程的虚拟地址空间。内核为一个大请求分配和映射内存可能需要很长时间。例如，考虑由262144个4096字节的页组成的千兆字节；即使单独一个页面的分配开销很低，但合起来如此大的分配数量将不可忽视。此外，有些程序申请分配的内存比实际使用的要多（例如，实现稀疏数组），或者为了以后的不时之需而分配内存。为了让 `sbrk()` 在这些情况下更快地完成，复杂的内核会延迟分配用户内存。也就是说，`sbrk()` 不分配物理内存，只是记住分配了哪些用户地址，并在用户页表中将这些地址标记为无效。当进程第一次尝试使用延迟分配中给定的页面时，CPU生成一个页面错误（**page fault**），内核通过分配物理内存、置零并添加映射来处理该错误。您将在这个实验室中向xv6添加这个延迟分配特性。

[!WARNING|label:Attention] 在开始编码之前，请阅读xv6手册的第4章（特别是4.6），以及可能要修改的相关文件：

- `kernel/trap.c`
- `kernel/vm.c`
- `kernel/sysproc.c`

要启动实验，请切换到 `lazy` 分支：

```
$ git fetch
$ git checkout lazy
$ make clean
```

Eliminate allocation from sbrk() (easy)

[!TIP|label:YOUR JOB] 你的首项任务是删除 `sbrk(n)` 系统调用中的页面分配代码（位于`sysproc.c`中的函数 `sys_sbrk()`）。`sbrk(n)` 系统调用将进程的内存大小增加n个字节，然后返回新分配区域的开始部分（即旧的大小）。新的`sbrk(n)` 应该只将进程的大小（`myproc()->sz`）增加n，然后返回旧的大小。它不应该分配内存——因此您应该删除对 `growproc()` 的调用（但是您仍然需要增加进程的大小！）。

试着猜猜这个修改的结果是什么：将会破坏什么？

进行此修改，启动xv6，并在shell中键入 `echo hi`。你应该看到这样的输出：

```
init: starting sh
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
    sepc=0x000000000001258 stval=0x000000000004008
va=0x000000000004000 pte=0x0000000000000000
panic: uvmunmap: not mapped
```

“`usertrap(): ...`”这条消息来自`trap.c`中的用户陷阱处理程序；它捕获了一个不知道如何处理的异常。请确保您了解发生此页面错误的原因。“`stval=0x0..04008`”表示导致页面错误的虚拟地址是`0x4008`。

Lazy allocation (moderate)

[!TIP|label:YOUR JOB] 修改`trap.c`中的代码以响应来自用户空间的页面错误，方法是新分配一个物理页面并映射到发生错误的地址，然后返回到用户空间，让进程继续执行。您应该在生成“`usertrap(): ...`”消息的`printf`调用之前添加代码。你可以修改任何其他xv6内核代码，以使`echo hi`正常工作。

提示：

- 你可以在`usertrap()`中查看`r_scause()`的返回值是否为13或15来判断该错误是否为页面错误
- `stval`寄存器中保存了造成页面错误的虚拟地址，你可以通过`r_stval()`读取
- 参考`vm.c`中的`uvmalloc()`中的代码，那是一个`sbrk()`通过`growproc()`调用的函数。你将需要对`kalloc()`和`mappages()`进行调用
- 使用`PGROUNDOWN(va)`将出错的虚拟地址向下舍入到页面边界
- 当前`uvmunmap()`会导致系统`panic`崩溃；请修改程序保证正常运行
- 如果内核崩溃，请在`kernel/kernel.asm`中查看`sepc`
- 使用`pgtbl lab`的`vmpprint`函数打印页表的内容
- 如果您看到错误“`incomplete type proc`”，请include“`spinlock.h`”然后是“`proc.h`”。

如果一切正常，你的lazy allocation应该使`echo hi`正常运行。您应该至少有一个页面错误（因为延迟分配），也许有两个。

Lazytests and Usertests (moderate)

我们为您提供了一个`lazystests`，这是一个xv6用户程序，它测试一些可能会给您的惰性内存分配器带来压力的特定情况。修改内核代码，使所有`lazystests`和`usertests`都通过。

- 处理`sbrk()`参数为负的情况。
- 如果某个进程在高于`sbrk()`分配的任何虚拟内存地址上出现页错误，则终止该进程。
- 在`fork()`中正确处理父到子内存拷贝。
- 处理这种情形：进程从`sbrk()`向系统调用（如`read`或`write`）传递有效地址，但尚未分配该地址的内存。
- 正确处理内存不足：如果在页面错误处理程序中执行`kalloc()`失败，则终止当前进程。

- 处理用户栈下面的无效页面上发生的错误。

如果内核通过 `lazytests` 和 `usertests`，那么您的解决方案是可以接受的：

```
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap...
usertrap(): ...
test lazy unmap: OK
running test out of memory
usertrap(): ...
test out of memory: OK
ALL TESTS PASSED
$ usertests
...
ALL TESTS PASSED
$
```

可选的挑战练习

- 让延时分配协同上一个实验中简化版的 `copyin` 一起工作。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 16:10:36

- [Lab6: Copy-on-Write Fork for xv6](#)
 - 问题
 - 解决方案
- [Implement copy-on write \(hard\)](#)
- [可选的挑战练习](#)

Lab6: Copy-on-Write Fork for xv6

虚拟内存提供了一定程度的间接寻址：内核可以通过将PTE标记为无效或只读来拦截内存引用，从而导致页面错误，还可以通过修改PTE来更改地址的含义。在计算机系统中有一种说法，任何系统问题都可以用某种程度的抽象方法来解决。[Lazy allocation](#)实验中提供了一个例子。这个实验探索了另一个例子：写时复制分支（copy-on write fork）。

在开始本实验前，将仓库切换到cow分支

```
$ git fetch  
$ git checkout cow  
$ make clean
```

问题

xv6中的 `fork()` 系统调用将父进程的所有用户空间内存复制到子进程中。如果父进程较大，则复制可能需要很长时间。更糟糕的是，这项工作经常造成大量浪费；例如，子进程中的 `fork()` 后跟 `exec()` 将导致子进程丢弃复制的内存，而其中的大部分可能都从未使用过。另一方面，如果父子进程都使用一个页面，并且其中一个或两个对该页面有写操作，则确实需要复制。

解决方案

`copy-on-write (COW) fork()` 的目标是推迟到子进程实际需要物理内存拷贝时再进行分配和复制物理内存页面。

`COW fork()` 只为子进程创建一个页表，用户内存的PTE指向父进程的物理页。
`COW fork()` 将父进程和子进程中的所有用户PTE标记为不可写。当任一进程试图写入其中一个COW页时，CPU将强制产生页面错误。内核页面错误处理程序检测到这种情况将为出错进程分配一页物理内存，将原始页复制到新页中，并修改出错进程中的相关PTE指向新的页面，将PTE标记为可写。当页面错误处理程序返回时，用户进程将能够写入其页面副本。

`COW fork()` 将使得释放用户内存的物理页面变得更加棘手。给定的物理页可能会被多个进程的页表引用，并且只有在最后一个引用消失时才应该被释放。

Implement copy-on write (hard)

[!TIP|label:YOUR JOB] 您的任务是在xv6内核中实现copy-on-write fork。如果修改后的内核同时成功执行 `cowtest` 和 `usertest` 程序就完成了。

为了帮助测试你的实现方案，我们提供了一个名为 `cowtest` 的 xv6 程序（源代码位于 `user/cowtest.c`）。`cowtest` 运行各种测试，但在未修改的 xv6 上，即使是第一个测试也会失败。因此，最初您将看到：

```
$ cowtest
simple: fork() failed
$
```

“simple”测试分配超过一半的可用物理内存，然后执行一系列的 `fork()`。`fork` 失败的原因是没有足够的可用物理内存来为子进程提供父进程内存的完整副本。

完成本实验后，内核应该通过 `cowtest` 和 `usertests` 中的所有测试。即：

```
$ cowtest
simple: ok
simple: ok
three: zombie!
ok
three: zombie!
ok
three: zombie!
ok
file: ok
ALL COW TESTS PASSED
$ usertests
...
ALL TESTS PASSED
$
```

这是一个合理的攻克计划：

1. 修改 `uvmcopy()` 将父进程的物理页映射到子进程，而不是分配新页。在子进程和父进程的 PTE 中清除 `PTE_W` 标志。
2. 修改 `usertrap()` 以识别页面错误。当 COW 页面出现页面错误时，使用 `kalloc()` 分配一个新页面，并将旧页面复制到新页面，然后将新页面添加到 PTE 中并设置 `PTE_W`。
3. 确保每个物理页在最后一个 PTE 对它的引用撤销时被释放——而不是在此之前。这样做了一个好方法是为每个物理页保留引用该页面的用户页表数的“引用计数”。当 `kalloc()` 分配页时，将页的引用计数设置为 1。当 `fork` 导致子进程共享页面时，增加页的引用计数；每当任何进程从其页表中删除页面时，减少页的引用计数。`kfree()` 只应在引用计数为零时将页面放回空闲列表。可以将这些计数保存在一个固定大小的整型数组中。你必须制定一个如何索引数组以及如何选择数组大小的方案。例如，您可以用页的物理地址除以 4096 对数组进行索引，并为数组提供等同于 `kalloc.c` 中 `kinit()` 在空闲列表中放置的所有页面的最高物理地址的元素数。
4. 修改 `copyout()` 在遇到 COW 页面时使用与页面错误相同的方案。

提示：

- **lazy page allocation** 实验可能已经让您熟悉了许多与 copy-on-write 相关的 xv6 内核代码。但是，您不应该将这个实验室建立在您的 **lazy allocation** 解决方案的基础上；相反，请按照上面的说明从一个新的 xv6 开始。
- 有一种可能很有用的方法来记录每个 PTE 是否是 COW 映射。您可以使用 RISC-V PTE 中的 RSW（reserved for software，即为软件保留的）位来实现此目

的。

- `usertests` 检查 `cowtest` 不测试的场景，所以别忘两个测试都需要完全通过。
- ***kernel/riscv.h***的末尾有一些有用的宏和页表标志位的定义。
- 如果出现COW页面错误并且没有可用内存，则应终止进程。

可选的挑战练习

- 修改xv6以同时支持lazy allocation和COW。
- 测量您的COW实现减少了多少xv6拷贝的字节数以及分配的物理页数。寻找并利用机会进一步减少这些数字。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 16:24:14

- [Lab7: Multithreading](#)
- [Uthread: switching between threads \(moderate\)](#)
- [Using threads \(moderate\)](#)
- [Barrier\(moderate\)](#)

Lab7: Multithreading

本实验将使您熟悉多线程。您将在用户级线程包中实现线程之间的切换，使用多个线程来加速程序，并实现一个屏障。

[!WARNING|label:Attention] 在编写代码之前，您应该确保已经阅读了xv6手册中的“第7章：调度”，并研究了相应的代码。

要启动实验，请切换到**thread**分支：

```
$ git fetch  
$ git checkout thread  
$ make clean
```

Uthread: switching between threads (moderate)

在本练习中，您将为用户级线程系统设计上下文切换机制，然后实现它。为了让您开始，您的xv6有两个文件：**user/uthread.c**和**user/uthread_switch.S**，以及一个规则：运行在**Makefile**中以构建 **uthread** 程序。**uthread.c**包含大多数用户级线程包，以及三个简单测试线程的代码。线程包缺少一些用于创建线程和在线程之间切换的代码。

[!TIP|label:YOUR JOB] 您的工作是提出一个创建线程和保存/恢复寄存器以在线程之间切换的计划，并实现该计划。完成后，`make grade` 应该表明您的解决方案通过了 `uthread` 测试。

完成后，在xv6上运行 `uthread` 时应该会看到以下输出（三个线程可能以不同的顺序启动）：

```

$ make qemu
...
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
...
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$
```

该输出来自三个测试线程，每个线程都有一个循环，该循环打印一行，然后将CPU让出给其他线程。

然而在此时还没有上下文切换的代码，您将看不到任何输出。

您需要将代码添加到 **user/uthread.c** 中的 `thread_create()` 和 `thread_schedule()`，以及 **user/uthread_switch.S** 中的 `thread_switch`。一个目标是确保当 `thread_schedule()` 第一次运行给定线程时，该线程在自己的栈上执行传递给 `thread_create()` 的函数。另一个目标是确保 `thread_switch` 保存被切换线程的寄存器，恢复切换到线程的寄存器，并返回到后一个线程指令中最后停止的点。您必须决定保存/恢复寄存器的位置：修改 `struct thread` 以保存寄存器是一个很好的计划。您需要在 `thread_schedule` 中添加对 `thread_switch` 的调用；您可以将需要的任何参数传递给 `thread_switch`，但目的是将线程从 `t` 切换到 `next_thread`。

提示：

- `thread_switch` 只需要保存/还原被调用方保存的寄存器（**callee-save register**，参见 LEC5 使用的文档《Calling Convention》）。为什么？
- 您可以在 **user/uthread.asm** 中看到 `uthread` 的汇编代码，这对于调试可能很方便。
- 这可能对于测试你的代码很有用，使用 `riscv64-linux-gnu-gdb` 的单步调试通过你的 `thread_switch`，你可以按这种方法开始：

```

(gdb) file user/_uthread
Reading symbols from user/_uthread...
(gdb) b uthread.c:60
```

这将在 **uthread.c** 的第60行设置断点。断点可能会（也可能不会）在运行 `uthread` 之前触发。为什么会出现这种情况？

一旦您的 xv6 shell 运行，键入“`uthread`”，`gdb` 将在第60行停止。现在您可以键入如下命令来检查 `uthread` 的状态：

```
(gdb) p/x *next_thread
```

使用“`x`”，您可以检查内存位置的内容：

```
(gdb) x/x next_thread->stack
```

您可以跳到`thread_switch`的开头，如下：

```
(gdb) b thread_switch
```

```
(gdb) c
```

您可以使用以下方法单步执行汇编指令：

```
(gdb) si
```

`gdb`的在线文档在[这里](#)。

Using threads (moderate)

在本作业中，您将探索使用哈希表的线程和锁的并行编程。您应该在具有多个内核的真实Linux或MacOS计算机（不是xv6，不是qemu）上执行此任务。最新的笔记本电脑都有多核处理器。

这个作业使用UNIX的man pthreads在手册页面上找到关于它的信息，您可以在web上查看，例如[这里](#)、[这里](#)和[这里](#)。

文件**notxv6/ph.c**包含一个简单的哈希表，如果单个线程使用，该哈希表是正确的，但是多个线程使用时，该哈希表是不正确的。在您的xv6主目录（可能是`~/xv6-labs-2020`）中，键入以下内容：

```
$ make ph  
$ ./ph 1
```

请注意，要构建`ph`，**Makefile**使用操作系统的gcc，而不是6.S081的工具。`ph`的参数指定在哈希表上执行`put`和`get`操作的线程数。运行一段时间后，`ph 1`将产生与以下类似的输出：

```
100000 puts, 3.991 seconds, 25056 puts/second  
0: 0 keys missing  
100000 gets, 3.981 seconds, 25118 gets/second
```

您看到的数字可能与此示例输出的数字相差两倍或更多，这取决于您计算机的速度、是否有多个核心以及是否正在忙于做其他事情。

`ph`运行两个基准程序。首先，它通过调用`put()`将许多键添加到哈希表中，并以每秒为单位打印`puts`的接收速率。之后它使用`get()`从哈希表中获取键。它打印由于`puts`而应该在哈希表中但丢失的键的数量（在本例中为0），并以每秒为单位打印`gets`的接收数量。

通过给`ph`一个大于1的参数，可以告诉它同时从多个线程使用其哈希表。试试`ph 2`：

```
$ ./ph 2
100000 puts, 1.885 seconds, 53044 puts/second
1: 16579 keys missing
0: 16579 keys missing
200000 gets, 4.322 seconds, 46274 gets/second
```

这个 `ph 2` 输出的第一行表明，当两个线程同时向哈希表添加条目时，它们达到每秒53044次插入的总速率。这大约是运行 `ph 1` 的单线程速度的两倍。这是一个优秀的“并行加速”，大约达到了人们希望的2倍（即两倍数量的核心每单位时间产出两倍的工作）。

然而，声明 `16579 keys missing` 的两行表示散列表中本应存在的大量键不存在。也就是说，`puts`应该将这些键添加到哈希表中，但出现了一些问题。请看一下

notxv6/ph.c，特别是 `put()` 和 `insert()`。

[!TIP|label:YOUR JOB] 为什么两个线程都丢失了键，而不是一个线程？确定可能导致键丢失的具有2个线程的事件序列。在 **answers-thread.txt** 中提交您的序列和简短解释。

[!TIP] 为了避免这种事件序列，请在 **notxv6/ph.c** 中的 `put` 和 `get` 中插入 `lock` 和 `unlock` 语句，以便在两个线程中丢失的键数始终为0。相关的 `pthread` 调用包括：

- `pthread_mutex_t lock; // declare a lock`
- `pthread_mutex_init(&lock, NULL); // initialize the lock`
- `pthread_mutex_lock(&lock); // acquire lock`
- `pthread_mutex_unlock(&lock); // release lock`

当 `make grade` 说您的代码通过 `ph_safe` 测试时，您就完成了，该测试需要两个线程的键缺失数为0。在此时，`ph_fast` 测试失败是正常的。

不要忘记调用 `pthread_mutex_init()`。首先用1个线程测试代码，然后用2个线程测试代码。您主要需要测试：程序运行是否正确呢（即，您是否消除了丢失的键？）？与单线程版本相比，双线程版本是否实现了并行加速（即单位时间内的工作量更多）？

在某些情况下，并发 `put()` 在哈希表中读取或写入的内存中没有重叠，因此不需要锁来相互保护。您能否更改 **ph.c** 以利用这种情况为某些 `put()` 获得并行加速？提示：每个散列桶加一个锁怎么样？

[!TIP|label:YOUR JOB] 修改代码，使某些 `put` 操作在保持正确性的同时并行运行。当 `make grade` 说你的代码通过了 `ph_safe` 和 `ph_fast` 测试时，你就完成了。`ph_fast` 测试要求两个线程每秒产生的 `put` 数至少是一个线程的1.25倍。

Barrier(moderate)

在本作业中，您将实现一个**屏障**(Barrier)：应用程序中的一个点，所有参与的线程在此点上必须等待，直到所有其他参与线程也达到该点。您将使用 `pthread` 条件变量，这是一种序列协调技术，类似于xv6的 `sleep` 和 `wakeup`。

您应该在真正的计算机（不是xv6，不是qemu）上完成此任务。

文件**notxv6/barrier.c**包含一个残缺的屏障实现。

```
$ make barrier
$ ./barrier 2
barrier: notxv6/barrier.c:42: thread: Assertion `i == t' failed.
```

2指定在屏障上同步的线程数（**barrier.c**中的 `nthread`）。每个线程执行一个循环。在每次循环迭代中，线程都会调用 `barrier()`，然后以随机微秒数休眠。如果一个线程在另一个线程到达屏障之前离开屏障将触发断言（`assert`）。期望的行为是每个线程在 `barrier()` 中阻塞，直到 `nthreads` 的所有线程都调用了 `barrier()`。

[!TIP|label:YOUR JOB] 您的目标是实现期望的屏障行为。除了在 ph 作业中看到的lock原语外，还需要以下新的pthread原语；详情请看[这里](#)和[这里](#)。

- // 在cond上进入睡眠，释放锁mutex，在醒来时重新获取
- `pthread_cond_wait(&cond, &mutex);`
- // 唤醒睡在cond的所有线程
- `pthread_cond_broadcast(&cond);`

确保您的方案通过 `make grade` 的 `barrier` 测试。

`pthread_cond_wait` 在调用时释放 `mutex`，并在返回前重新获取 `mutex`。

我们已经为您提供提供了 `barrier_init()`。您的工作是实现 `barrier()`，这样panic就不会发生。我们为您定义了 `struct barrier`；它的字段供您使用。

有两个问题使您的任务变得复杂：

- 你必须处理一系列的 `barrier` 调用，我们称每一连串的调用为一轮（`round`）。`bstate.round` 记录当前轮数。每次当所有线程都到达屏障时，都应增加 `bstate.round`。
- 您必须处理这样的情况：一个线程在其他线程退出 `barrier` 之前进入了下一轮循环。特别是，您在前后两轮中重复使用 `bstate.nthread` 变量。确保在前一轮仍在使用 `bstate.nthread` 时，离开 `barrier` 并循环运行的线程不会增加 `bstate.nthread`。

使用一个、两个和两个以上的线程测试代码。

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 22:42:38

- [Lab8: locks](#)
- [Memory allocator\(moderate\)](#)
- [Buffer cache\(hard\)](#)
- 可选的挑战练习

Lab8: locks

在本实验中，您将获得重新设计代码以提高并行性的经验。多核机器上并行性差的一个常见症状是频繁的锁争用。提高并行性通常涉及更改数据结构和锁定策略以减少争用。您将对xv6内存分配器和块缓存执行此操作。

[!WARNING|label:Attention] 在编写代码之前，请确保阅读xv6手册中的以下部分：

- 第6章：《锁》和相应的代码。
- 第3.5节：《代码：物理内存分配》
- 第8.1节至第8.3节：《概述》、《Buffer cache层》和《代码：Buffer cache》

要开始本实验，请将代码切换到 lock 分支

```
$ git fetch
$ git checkout lock
$ make clean
```

Memory allocator(moderate)

程序 `user/kalloc` 强调了 xv6 的内存分配器：三个进程增长和缩小地址空间，导致对 `kalloc` 和 `kfree` 的多次调用。`kalloc` 和 `kfree` 获得 `kmem.lock`。`kalloc` 打印（作为“#fetch-and-add”）在 `acquire` 中由于尝试获取另一个内核已经持有的锁而进行的循环迭代次数，如 `kmem` 锁和其他锁。`acquire` 中的循环迭代次数是锁争用的粗略度量。完成实验前，`kalloc` 的输出与此类似：

```
$ kalloc
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 83375 #acquire() 433015
lock: bcache: #fetch-and-add 0 #acquire() 1260
--- top 5 contended locks:
lock: kmem: #fetch-and-add 83375 #acquire() 433015
lock: proc: #fetch-and-add 23737 #acquire() 130718
lock: virtio_disk: #fetch-and-add 11159 #acquire() 114
lock: proc: #fetch-and-add 5937 #acquire() 130786
lock: proc: #fetch-and-add 4080 #acquire() 130786
tot= 83375
test1 FAIL
```

`acquire` 为每个锁维护要获取该锁的 `acquire` 调用计数，以及 `acquire` 中循环尝试但未能设置锁的次数。`kalloc` 调用一个系统调用，使内核打印 `kmem` 和 `bcache` 锁（这是本实验的重点）以及5个最有竞争的锁的计数。如果存在锁争用，则 `acquire` 循环迭代的次数将很大。系统调用返回 `kmem` 和 `bcache` 锁的循环迭代次数之和。

对于本实验，您必须使用具有多个内核的专用空载机器。如果你使用一台正在做其他事情的机器，`kalloc` 打印的计数将毫无意义。你可以使用专用的Athena 工作站或你自己的笔记本电脑，但不要使用拨号机。

`kalloc` 中锁争用的根本原因是 `kalloc()` 有一个空闲列表，由一个锁保护。要消除锁争用，您必须重新设计内存分配器，以避免使用单个锁和列表。基本思想是为每个CPU维护一个空闲列表，每个列表都有自己的锁。因为每个CPU将在不同的列表上运行，不同CPU上的分配和释放可以并行运行。主要的挑战将是处理一个CPU的空闲列表为空，而另一个CPU的列表有空闲内存的情况；在这种情况下，一个CPU必须“窃取”另一个CPU空闲列表的一部分。窃取可能会引入锁争用，但这种情况希望不会经常发生。

[!TIP|label:YOUR JOB] 您的工作是实现每个CPU的空闲列表，并在CPU的空闲列表为空时进行窃取。所有锁的命名必须以“`kmem`”开头。也就是说，您应该为每个锁调用 `initlock`，并传递一个以“`kmem`”开头的名称。运行 `kalloc` 以查看您的实现是否减少了锁争用。要检查它是否仍然可以分配所有内存，请运行 `usertests sbrkmuch`。您的输出将与下面所示的类似，在 `kmem` 锁上的争用总数将大大减少，尽管具体的数字会有所不同。确保 `usertests` 中的所有测试都通过。评分应该表明考试通过。

```
$ kalloc
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 42843
lock: kmem: #fetch-and-add 0 #acquire() 198674
lock: kmem: #fetch-and-add 0 #acquire() 191534
lock: bcache: #fetch-and-add 0 #acquire() 1242
--- top 5 contended locks:
lock: proc: #fetch-and-add 43861 #acquire() 117281
lock: virtio_disk: #fetch-and-add 5347 #acquire() 114
lock: proc: #fetch-and-add 4856 #acquire() 117312
lock: proc: #fetch-and-add 4168 #acquire() 117316
lock: proc: #fetch-and-add 2797 #acquire() 117266
tot= 0
test1 OK
start test2
total free number of pages: 32499 (out of 32768)
.....
test2 OK
$ usertests sbrkmuch
usertests starting
test sbrkmuch: OK
ALL TESTS PASSED
$ usertests
...
ALL TESTS PASSED
$
```

提示：

- 您可以使用 ***kernel/param.h*** 中的常量 `NCPU`
- 让 `freerange` 将所有可用内存分配给运行 `freerange` 的 CPU。
- 函数 `cpuid` 返回当前的核心编号，但只有在中断关闭时调用它并使用其结果才是安全的。您应该使用 `push_off()` 和 `pop_off()` 来关闭和打开中断。
- 看看 ***kernel/sprintf.c*** 中的 `sprintf` 函数，了解字符串如何进行格式化。尽管可以将所有锁命名为“`kmem`”。

Buffer cache(hard)

这一半作业独立于前一半；不管你是否完成了前半部分，你都可以完成这半部分（并通过测试）。

如果多个进程密集地使用文件系统，它们可能会争夺 `bcache.lock`，它保护 ***kernel/bio.c*** 中的磁盘块缓存。`bchacetest` 创建多个进程，这些进程重复读取不同的文件，以便在 `bcache.lock` 上生成争用；（在完成本实验之前）其输出如下所示：

```
$ bchacetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 33035
lock: bcache: #fetch-and-add 16142 #acquire() 65978
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 162870 #acquire() 1188
lock: proc: #fetch-and-add 51936 #acquire() 73732
lock: bcache: #fetch-and-add 16142 #acquire() 65978
lock: uart: #fetch-and-add 7505 #acquire() 117
lock: proc: #fetch-and-add 6937 #acquire() 73420
tot= 16142
test0: FAIL
start test1
test1 OK
```

您可能会看到不同的输出，但 `bcache` 锁的 `acquire` 循环迭代次数将很高。如果查看 ***kernel/bio.c*** 中的代码，您将看到 `bcache.lock` 保护已缓存的块缓冲区的列表、每个块缓冲区中的引用计数（`b->refcnt`）以及缓存块的标识（`b->dev` 和 `b->blockno`）。

[!TIP|label:YOUR JOB] 修改块缓存，以便在运行 `bchacetest` 时，`bcache`（buffer cache的缩写）中所有锁的 `acquire` 循环迭代次数接近于零。理想情况下，块缓存中涉及的所有锁的计数总和应为零，但只要总和小于 500 就可以。修改 `bget` 和 `brelse`，以便 `bcache` 中不同块的并发查找和释放不太可能在锁上发生冲突（例如，不必全部等待 `bcache.lock`）。你必须保护每个块最多缓存一个副本的不变量。完成后，您的输出应该与下面显示的类似（尽管不完全相同）。确保 `usertests` 仍然通过。完成后，`make grade` 应该通过所有测试。

```

$ bcachetest
start test0
test0 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 0 #acquire() 32954
lock: kmem: #fetch-and-add 0 #acquire() 75
lock: kmem: #fetch-and-add 0 #acquire() 73
lock: bcache: #fetch-and-add 0 #acquire() 85
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4159
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2118
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4274
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4326
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6334
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6321
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6704
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6696
lock: bcache.bucket: #fetch-and-add 0 #acquire() 7757
lock: bcache.bucket: #fetch-and-add 0 #acquire() 6199
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4136
lock: bcache.bucket: #fetch-and-add 0 #acquire() 4136
lock: bcache.bucket: #fetch-and-add 0 #acquire() 2123
--- top 5 contended locks:
lock: virtio_disk: #fetch-and-add 158235 #acquire() 1193
lock: proc: #fetch-and-add 117563 #acquire() 3708493
lock: proc: #fetch-and-add 65921 #acquire() 3710254
lock: proc: #fetch-and-add 44090 #acquire() 3708607
lock: proc: #fetch-and-add 43252 #acquire() 3708521
tot= 128
test0: OK
start test1
test1 OK
$ usertests
...
ALL TESTS PASSED
$
```

请将你所有的锁以“`bcache`”开头进行命名。也就是说，您应该为每个锁调用`initlock`，并传递一个以“`bcache`”开头的名称。

减少块缓存中的争用比`kalloc`更复杂，因为`bcache`缓冲区真正的在进程（以及CPU）之间共享。对于`kalloc`，可以通过给每个CPU设置自己的分配器来消除大部分争用；这对块缓存不起作用。我们建议您使用每个哈希桶都有一个锁的哈希表在缓存中查找块号。

在您的解决方案中，以下是一些存在锁冲突但可以接受的情形：

- 当两个进程同时使用相同的块号时。`bcachetest test0`始终不会这样做。
- 当两个进程同时在`cache`中未命中时，需要找到一个未使用的块进行替换。`bcachetest test0`始终不会这样做。
- 在你用来划分块和锁的方案中某些块可能会发生冲突，当两个进程同时使用冲突的块时。例如，如果两个进程使用的块，其块号散列到哈希表中相同的槽。`bcachetest test0`可能会执行此操作，具体取决于您的设计，但您应该尝试调整方案的细节以避免冲突（例如，更改哈希表的大小）。

`bcachetest`的`test1`使用的块比缓冲区更多，并且执行大量文件系统代码路径。

提示：

- 请阅读xv6手册中对块缓存的描述（第8.1-8.3节）。

- 可以使用固定数量的散列桶，而不动态调整哈希表的大小。使用素数个存储桶（例如13）来降低散列冲突的可能性。
- 在哈希表中搜索缓冲区并在找不到缓冲区时为该缓冲区分配条目必须是原子的。
- 删除保存了所有缓冲区的列表（`bcache.head` 等），改为标记上次使用时间的时间戳缓冲区（即使用`kernel/trap.c`中的`ticks`）。通过此更改，`brelse` 不需 要获取`bcache`锁，并且`bget`可以根据时间戳选择最近使用最少的块。
- 可以在`bget`中串行化回收（即`bget`中的一部分：当缓存中的查找未命中时，它选择要复用的缓冲区）。
- 在某些情况下，您的解决方案可能需要持有两个锁；例如，在回收过程中，您可能需要持有`bcache`锁和每个`bucket`（散列桶）一个锁。确保避免死锁。
- 替换块时，您可能会将`struct buf`从一个`bucket`移动到另一个`bucket`，因为新块散列到不同的`bucket`。您可能会遇到一个棘手的情况：新块可能会散列到与旧块相同的`bucket`中。在这种情况下，请确保避免死锁。
- 一些调试技巧：实现`bucket`锁，但将全局`bcache.lock`的`acquire / release`保留在`bget`的开头/结尾，以串行化代码。一旦您确定它在没有竞争条件的情况下是正确的，请移除全局锁并处理并发性问题。您还可以运行`make CPUS=1 qemu`以使用一个内核进行测试。

可选的挑战练习

在buffer cache中进行无锁查找。提示：使用gcc的`__sync_*`函数。您如何证明自己的实现是正确的？

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 16:38:19

- [Lab9: file system](#)
- [Large files\(moderate\)](#)
 - 预备
 - 看什么
 - 你的工作
- [Symbolic links\(moderate\)](#)
 - 你的工作
- [可选的挑战练习](#)

Lab9: file system

在本实验室中，您将向xv6文件系统添加大型文件和符号链接。

[!WARNING|label:Attention] 在编写代码之前，您应该阅读《xv6手册》中的《第八章：文件系统》，并学习相应的代码。

获取实验室的xv6源代码并切换到fs分支：

```
$ git fetch  
$ git checkout fs  
$ make clean
```

Large files(moderate)

在本作业中，您将增加xv6文件的最大大小。目前，xv6文件限制为268个块或 $268 \times \text{BSIZE}$ 字节（在xv6中`BSIZE`为1024）。此限制来自以下事实：一个xv6`inode`包含12个“直接”块号和一个“间接”块号，“一级间接”块指一个最多可容纳256个块号的块，总共 $12+256=268$ 个块。

`bigfile` 命令可以创建最长的文件，并报告其大小：

```
$ bigfile  
..  
wrote 268 blocks  
bigfile: file is too small  
$
```

测试失败，因为`bigfile`希望能够创建一个包含65803个块的文件，但未修改的xv6将文件限制为268个块。

您将更改xv6文件系统代码，以支持每个`inode`中可包含256个一级间接块地址的“二级间接”块，每个一级间接块最多可以包含256个数据块地址。结果将是一个文件将能够包含多达65803个块，或 $256 \times 256 + 256 + 11$ 个块（11而不是12，因为我们将为二级间接块牺牲一个直接块号）。

预备

`mkfs` 程序创建xv6文件系统磁盘映像，并确定文件系统的总块数；此大小由 **kernel/param.h** 中的 `FSSIZE` 控制。您将看到，该实验室存储库中的 `FSSIZE` 设置为 200000 个块。您应该在 `make` 输出中看到来自 `mkfs/mkfs` 的以下输出：

```
nmeta 70 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 25) blocks 199930
```

这一行描述了 `mkfs/mkfs` 构建的文件系统：它有 70 个元数据块（用于描述文件系统的块）和 199930 个数据块，总计 200000 个块。

如果在实验期间的任何时候，您发现自己必须从头开始重建文件系统，您可以运行 `make clean`，强制 `make` 重建 **fs.img**。

看什么

磁盘索引节点的格式由 **fs.h** 中的 `struct dinode` 定义。您应当尤其对 `NDIRECT`、`NINDIRECT`、`MAXFILE` 和 `struct dinode` 的 `addr[]` 元素感兴趣。查看《XV6 手册》中的图 8.3，了解标准 xv6 索引结点的示意图。

在磁盘上查找文件数据的代码位于 **fs.c** 的 `bmap()` 中。看看它，确保你明白它在做什么。在读取和写入文件时都会调用 `bmap()`。写入时，`bmap()` 会根据需要分配新块以保存文件内容，如果需要，还会分配间接块以保存块地址。

`bmap()` 处理两种类型的块编号。`bn` 参数是一个“逻辑块号”——文件中相对于文件开头的块号。`ip->addr[]` 中的块号和 `bread()` 的参数都是磁盘块号。您可以将 `bmap()` 视为将文件的逻辑块号映射到磁盘块号。

你的工作

修改 `bmap()`，以便除了直接块和一级间接块之外，它还实现二级间接块。你只需要有 11 个直接块，而不是 12 个，为你的新的二级间接块腾出空间；不允许更改磁盘 `inode` 的大小。`ip->addr[]` 的前 11 个元素应该是直接块；第 12 个应该是一个一级间接块（与当前的一样）；13 号应该是你的新二级间接块。当 `bigfile` 写入 65803 个块并成功运行 `usertests` 时，此练习完成：

```
$ bigfile
.....
wrote 65803 blocks
done; ok
$ usertests
...
ALL TESTS PASSED
$
```

运行 `bigfile` 至少需要一分钟半的时间。

提示：

- 确保您理解 `bmap()`。写出 `ip->addr[]`、间接块、二级间接块和它所指向的一级间接块以及数据块之间的关系图。确保您理解为什么添加二级间接块会将最

大文件大小增加`256*256`个块（实际上要-1，因为您必须将直接块的数量减少一个）。

- 考虑如何使用逻辑块号索引二级间接块及其指向的间接块。
- 如果更改 `NDIRECT` 的定义，则可能必须更改 `file.h` 文件中 `struct inode` 中 `addr[]` 的声明。确保 `struct inode` 和 `struct dinode` 在其 `addr[]` 数组中具有相同数量的元素。
- 如果更改 `NDIRECT` 的定义，请确保创建一个新的 `fs.img`，因为 `mkfs` 使用 `NDIRECT` 构建文件系统。
- 如果您的文件系统进入坏状态，可能是由于崩溃，请删除 `fs.img`（从 Unix 而不是 xv6 执行此操作）。`make` 将为您构建一个新的干净文件系统映像。
- 别忘了把你 `bread()` 的每一个块都 `brelse()`。
- 您应该仅根据需要分配间接块和二级间接块，就像原始的 `bmap()`。
- 确保 `itrunc` 释放文件的所有块，包括二级间接块。

Symbolic links(moderate)

在本练习中，您将向 xv6 添加符号链接。符号链接（或软链接）是指按路径名链接的文件；当一个符号链接打开时，内核跟随该链接指向引用的文件。符号链接类似于硬链接，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备。尽管 xv6 不支持多个设备，但实现此系统调用是了解路径名查找工作原理的一个很好的练习。

你的工作

[!TIP|label:YOUR JOB] 您将实现 `symlink(char *target, char *path)` 系统调用，该调用在引用由 `target` 命名的文件的路径处创建一个新的符号链接。有关更多信息，请参阅 `symlink` 手册页（注：执行 `man symlink`）。要进行测试，请将 `symlinktest` 添加到 `Makefile` 并运行它。当测试产生以下输出（包括 `usertests` 运行成功）时，您就完成本作业了。

```
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$ usertests
...
ALL TESTS PASSED
$
```

提示：

- 首先，为 `symlink` 创建一个新的系统调用号，在 `user/usys.pl`、`user/user.h` 中添加一个条目，并在 `kernel/sysfile.c` 中实现一个空的 `sys_symlink`。
- 向 `kernel/stat.h` 添加新的文件类型 (`T_SYMLINK`) 以表示符号链接。
- 在 `kernel/fcntl.h` 中添加一个新标志 (`O_NOFOLLOW`)，该标志可用于 `open` 系统调用。请注意，传递给 `open` 的标志使用按位或运算符组合，因此新标志不应与任何现有标志重叠。一旦将 `user/symlinktest.c` 添加到 `Makefile` 中，您就可以编译它。

- 实现 `symlink(target, path)` 系统调用，以在 `path` 处创建一个新的指向 `target` 的符号链接。请注意，系统调用的成功不需要 `target` 已经存在。您需要选择存储符号链接目标路径的位置，例如在`inode`的数据块中。`symlink` 应返回一个表示成功（0）或失败（-1）的整数，类似于 `link` 和 `unlink`。
- 修改 `open` 系统调用以处理路径指向符号链接的情况。如果文件不存在，则打开必须失败。当进程向 `open` 传递 `O_NOFOLLOW` 标志时，`open` 应打开符号链接（而不是跟随符号链接）。
- 如果链接文件也是符号链接，则必须递归地跟随它，直到到达非链接文件为止。如果链接形成循环，则必须返回错误代码。你可以通过以下方式估算存在循环：通过在链接深度达到某个阈值（例如10）时返回错误代码。
- 其他系统调用（如 `link` 和 `unlink`）不得跟随符号链接；这些系统调用对符号链接本身进行操作。
- 您不必处理指向此实验的目录的符号链接。

可选的挑战练习

实现三级间接块

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 16:40:21

- [Lab10: mmap](#)
- [mmap\(hard\)](#)

Lab10: mmap

mmap(hard)

`mmap` 和 `munmap` 系统调用允许UNIX程序对其地址空间进行详细控制。它们可用于在进程之间共享内存，将文件映射到进程地址空间，并作为用户级页面错误方案的一部分，如本课程中讨论的垃圾收集算法。在本实验室中，您将把 `mmap` 和 `munmap` 添加到xv6中，重点关注内存映射文件（**memory-mapped files**）。

获取实验室的xv6源代码并切换到 `mmap` 分支：

```
$ git fetch
$ git checkout mmap
$ make clean
```

手册页面（运行 `man 2 mmap`）显示了 `mmap` 的以下声明：

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

可以通过多种方式调用 `mmap`，但本实验只需要与内存映射文件相关功能子集。您可以假设 `addr` 始终为零，这意味着内核应该决定映射文件的虚拟地址。`mmap` 返回该地址，如果失败则返回 `0xffffffffffffffffff`。`length` 是要映射的字节数；它可能与文件的长度不同。`prot` 指示内存是否应映射为可读、可写，以及/或者可执行的；您可以认为 `prot` 是 `PROT_READ` 或 `PROT_WRITE` 或两者兼有。`flags` 要么是 `MAP_SHARED`（映射内存的修改应写回文件），要么是 `MAP_PRIVATE`（映射内存的修改不应写回文件）。您不必在 `flags` 中实现任何其他位。`fd` 是要映射的文件的打开文件描述符。可以假定 `offset` 为零（它是要映射的文件的起点）。

允许进程映射同一个 `MAP_SHARED` 文件而不共享物理页面。

`munmap(addr, length)` 应删除指定地址范围内的 `mmap` 映射。如果进程修改了内存并将其映射为 `MAP_SHARED`，则应首先将修改写入文件。`munmap` 调用可能只覆盖 `mmap` 区域的一部分，但您可以认为它取消映射的位置要么在区域起始位置，要么在区域结束位置，要么就是整个区域（但不会在区域中间“打洞”）。

[!TIP|label:YOUR JOB] 您应该实现足够的 `mmap` 和 `munmap` 功能，以使 `mmaptest` 测试程序正常工作。如果 `mmaptest` 不会用到某个 `mmap` 的特性，则不需要实现该特性。

完成后，您应该会看到以下输出：

```

$ mmapttest
mmap_test starting
test mmap f
test mmap f: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test mmap two files
test mmap two files: OK
mmapttest: ALL OK
fork_test starting
fork_test OK
mmapttest: all tests succeeded
$ usertests
usertests starting
...
ALL TESTS PASSED
$
```

提示：

- 首先，向 `UPROGS` 添加 `_mmapttest`，以及 `mmap` 和 `munmap` 系统调用，以便让 `user/mmapttest.c` 进行编译。现在，只需从 `mmap` 和 `munmap` 返回错误。我们在 `kernel/fcntl.h` 中为您定义了 `PROT_READ` 等。运行 `mmapttest`，它将在第一次 `mmap` 调用时失败。
- 惰性地填写页表，以响应页错误。也就是说，`mmap` 不应该分配物理内存或读取文件。相反，在 `usertrap` 中（或由 `usertrap` 调用）的页面错误处理代码中执行此操作，就像在 `lazy page allocation` 实验中一样。惰性分配的原因是确保大文件的 `mmap` 是快速的，并且比物理内存大的文件的 `mmap` 是可能的。
- 跟踪 `mmap` 为每个进程映射的内容。定义与第15课中描述的 `VMA`（虚拟内存区域）对应的结构体，记录 `mmap` 创建的虚拟内存范围的地址、长度、权限、文件等。由于 `xv6` 内核中没有内存分配器，因此可以声明一个固定大小的 `VMA` 数组，并根据需要从该数组进行分配。大小为 16 应该就足够了。
- 实现 `mmap`：在进程的地址空间中找到一个未使用的区域来映射文件，并将 `VMA` 添加到进程的映射区域表中。`VMA` 应该包含指向映射文件对应 `struct file` 的指针；`mmap` 应该增加文件的引用计数，以便在文件关闭时结构体不会消失（提示：请参阅 `filedup`）。运行 `mmapttest`：第一次 `mmap` 应该成功，但是第一次访问被 `mmap` 的内存将导致页面错误并终止 `mmapttest`。
- 添加代码以导致在 `mmap` 的区域中产生页面错误，从而分配一页物理内存，将 4096 字节的相关文件读入该页面，并将其映射到用户地址空间。使用 `readi` 读取文件，它接受一个偏移量参数，在该偏移处读取文件（但必须 `lock/unlock` 传递给 `readi` 的索引结点）。不要忘记在页面上正确设置权限。运行 `mmapttest`；它应该到达第一个 `munmap`。
- 实现 `munmap`：找到地址范围的 `VMA` 并取消映射指定页面（提示：使用 `uvmunmap`）。如果 `munmap` 删除了先前 `mmap` 的所有页面，它应该减少相应 `struct file` 的引用计数。如果未映射的页面已被修改，并且文件已映射到 `MAP_SHARED`，请将页面写回该文件。查看 `filewrite` 以获得灵感。

- 理想情况下，您的实现将只写回程序实际修改的 `MAP_SHARED` 页面。RISC-V PTE中的脏位（`D`）表示是否已写入页面。但是，`mmaptest` 不检查非脏页是否有回写；因此，您可以不用看 `D` 位就写回页面。
- 修改 `exit` 将进程的已映射区域取消映射，就像调用了 `munmap` 一样。运行 `mmaptest ; mmap_test` 应该通过，但可能不会通过 `fork_test`。
- 修改 `fork` 以确保子对象具有与父对象相同的映射区域。不要忘记增加VMA 的 `struct file` 的引用计数。在子进程的页面错误处理程序中，可以分配新的物理页面，而不是与父级共享页面。后者会更酷，但需要更多的实施工作。运行 `mmaptest ;` 它应该通过 `mmap_test` 和 `fork_test`。

运行 `usertests` 以确保一切正常。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-09-06 17:15:49

- [Lab11: Network](#)
- [背景](#)
- [你的工作\(hard\)](#)
- [提示](#)

Lab11: Network

在本实验室中，您将为网络接口卡（NIC）编写一个xv6设备驱动程序。

获取xv6实验的源代码并切换到 `net` 分支：

```
$ git fetch  
$ git checkout net  
$ make clean
```

背景

[!TIP] 在编写代码之前，您可能会发现阅读xv6手册中的《第5章：中断和设备驱动》很有帮助。

您将使用名为E1000的网络设备来处理网络通信。对于xv6（以及您编写的驱动程序），E1000看起来像是连接到真正以太网局域网（LAN）的真正硬件。事实上，用于与您的驱动程序对话的E1000是qemu提供的模拟，连接到的LAN也由qemu模拟。在这个模拟LAN上，xv6（“来宾”）的IP地址为10.0.2.15。Qemu还安排运行Qemu的计算机出现在IP地址为10.0.2.2的LAN上。当xv6使用E1000将数据包发送到10.0.2.2时，qemu会将数据包发送到运行qemu的（真实）计算机上的相应应用程序（“主机”）。

您将使用QEMU的“用户模式网络栈（user-mode network stack）”。[QEMU的文档](#)中有更多关于用户模式栈的内容。我们已经更新了[Makefile](#)以启用QEMU的用户模式网络栈和E1000网卡。

[Makefile](#)将QEMU配置为将所有传入和传出数据包记录到实验目录中的 `packets.pcap` 文件中。查看这些记录可能有助于确认xv6正在发送和接收您期望的数据包。要显示记录的数据包，请执行以下操作：

```
tcpdump -XXnr packets.pcap
```

我们已将一些文件添加到本实验的xv6存储库中。`kernel/e1000.c`文件包含E1000的初始化代码以及用于发送和接收数据包的空函数，您将填写这些函数。

`kernel/e1000_dev.h`包含E1000定义的寄存器和标志位的定义，并在[《英特尔E1000软件开发人员手册》](#)中进行了描述。`kernel/net.c`和`kernel/net.h`包含一个实现IP、UDP和ARP协议的简单网络栈。这些文件还包含用于保存数据包的灵活数据结构（称为 `mbuf`）的代码。最后，`kernel/pci.c`包含在xv6引导时在PCI总线上搜索E1000卡的代码。

你的工作(hard)

[!TIP][label:YOUR JOB] 您的工作是在 ***kernel/e1000.c*** 中完成 `e1000_transmit()` 和 `e1000_recv()`，以便驱动程序可以发送和接收数据包。当 `make grade` 表示您的解决方案通过了所有测试时，您就完成了。

[!TIP] 在编写代码时，您会发现自己参考了《[E1000软件开发人员手册](#)》。以下部分可能特别有用：

- Section 2是必不可少的，它概述了整个设备。
- Section 3.2概述了数据包接收。
- Section 3.3与Section 3.4一起概述了数据包传输。
- Section 13概述了E1000使用的寄存器。
- Section 14可能会帮助您理解我们提供的init代码。

浏览《[E1000软件开发人员手册](#)》。本手册涵盖了几个密切相关的以太网控制器。QEMU模拟82540EM。现在浏览第2章，了解该设备。要编写驱动程序，您需要熟悉第3章和第14章以及第4.1节（虽然不包括4.1的子节）。你还需要参考第13章。其他章节主要介绍你的驱动程序不必与之交互的E1000组件。一开始不要担心细节；只需了解文档的结构，就可以在以后找到内容。E1000具有许多高级功能，其中大部分您可以忽略。完成这个实验只需要一小部分基本功能。

我们在***e1000.c***中提供的 `e1000_init()` 函数将E1000配置为读取要从RAM传输的数据包，并将接收到的数据包写入RAM。这种技术称为DMA，用于直接内存访问，指的是E1000硬件直接向RAM写入和读取数据包。

由于数据包突发到达的速度可能快于驱动程序处理数据包的速度，因此 `e1000_init()` 为E1000提供了多个缓冲区，E1000可以将数据包写入其中。E1000要求这些缓冲区由RAM中的“描述符”数组描述；每个描述符在RAM中都包含一个地址，E1000可以在其中写入接收到的数据包。`struct rx_desc` 描述描述符格式。描述符数组称为接收环或接收队列。它是一个圆环，在这个意义上，当网卡或驱动程序到达队列的末尾时，它会绕回到数组的开头。`e1000_init()` 使用 `mbufalloc()` 为要进行DMA的E1000分配 `mbuf` 数据包缓冲区。此外还有一个传输环，驱动程序将需要E1000发送的数据包放入其中。`e1000_init()` 将两个环的大小配置为 `RX_RING_SIZE` 和 `TX_RING_SIZE`。

当***net.c***中的网络栈需要发送数据包时，它会调用 `e1000_transmit()`，并使用一个保存要发送的数据包的 `mbuf` 作为参数。传输代码必须在TX（传输）环的描述符中放置指向数据包数据的指针。`struct tx_desc` 描述了描述符的格式。您需要确保每个 `mbuf` 最终被释放，但只能在E1000完成数据包传输之后（E1000在描述符中设置 `E1000_TXD_STAT_DD` 位以指示此情况）。

当E1000从以太网接收到每个包时，它首先将包DMA到下一个RX(接收)环描述符指向的 `mbuf`，然后产生一个中断。`e1000_recv()` 代码必须扫描RX环，并通过调用 `net_rx()` 将每个新数据包的 `mbuf` 发送到网络栈（在***net.c***中）。然后，您需要分配一个新的 `mbuf` 并将其放入描述符中，以便当E1000再次到达RX环中的该点时，它会找到一个新的缓冲区，以便DMA新数据包。

除了在RAM中读取和写入描述符环外，您的驱动程序还需要通过其内存映射控制寄存器与E1000交互，以检测接收到数据包何时可用，并通知E1000驱动程序已经用要发送的数据包填充了一些TX描述符。全局变量 `regs` 包含指向E1000第一个控制寄存器的指针；您的驱动程序可以通过将 `regs` 索引为数组来获取其他寄存器。您需要特别使用索引 `E1000_RDT` 和 `E1000_TDT`。

要测试驱动程序，请在一个窗口中运行 `make server`，在另一个窗口中运行 `make qemu`，然后在 `xv6` 中运行 `nettests`。`nettests` 中的第一个测试尝试将 UDP 数据包发送到主机操作系统，地址是 `make server` 运行的程序。如果您还没有完成实验，`E1000` 驱动程序实际上不会发送数据包，也不会发生什么事情。

完成实验后，`E1000` 驱动程序将发送数据包，`qemu` 将其发送到主机，`make server` 将看到它并发送响应数据包，然后 `E1000` 驱动程序和 `nettests` 将看到响应数据包。但是，在主机发送应答之前，它会向 `xv6` 发送一个“ARP”请求包，以找出其 48 位以太网地址，并期望 `xv6` 以 ARP 应答进行响应。一旦您完成了对 `E1000` 驱动程序的工作，`kernel/net.c` 就会处理这个问题。如果一切顺利，`nettests` 将打印 `testing ping: OK`，`make server` 将打印 `a message from xv6!`。

`tcpdump -XXnr packets.pcap` 应该生成这样的输出：

```
reading from file packets.pcap, link-type EN10MB (Ethernet)
15:27:40.861988 IP 10.0.2.15.2000 > 10.0.2.2.25603: UDP, length 19
    0x0000: ffff ffff ffff 5254 0012 3456 0800 4500 .....RT..4V..E.
    0x0010: 002f 0000 0000 6411 3eae 0a00 020f 0a00 ./....d.>.....
    0x0020: 0202 07d0 6403 001b 0000 6120 6d65 7373 ....d....a.mess
    0x0030: 6167 6520 6672 6f6d 2078 7636 21     age.from.xv6!
15:27:40.862370 ARP, Request who-has 10.0.2.15 tell 10.0.2.2., length 28
    0x0000: ffff ffff ffff 5255 0a00 0202 0806 0001 .....RU.....
    0x0010: 0800 0604 0001 5255 0a00 0202 0a00 0202 .....RU.....
    0x0020: 0000 0000 0000 0a00 020f ..... .
15:27:40.862844 ARP, Reply 10.0.2.15 is-at 52:54:00:12:34:56, length 28
    0x0000: ffff ffff ffff 5254 0012 3456 0806 0001 .....RT..4V....
    0x0010: 0800 0604 0002 5254 0012 3456 0a00 020f .....RT..4V....
    0x0020: 5255 0a00 0202 0a00 0202 RU.....
15:27:40.863036 IP 10.0.2.2.25603 > 10.0.2.15.2000: UDP, length 17
    0x0000: 5254 0012 3456 5255 0a00 0202 0800 4500 RT..4VRU.....E.
    0x0010: 002d 0000 0000 4011 62b0 0a00 0202 0a00 ..-.@.b.....
    0x0020: 020f 6403 07d0 0019 3406 7468 6973 2069 ..d....4.this.i
    0x0030: 7320 7468 6520 686f 7374 21           s.the.host!
```

您的输出看起来会有些不同，但它应该包含字符串“ARP, Request”，“ARP, Reply”，“UDP”，“a.message.from.xv6”和“this.is.the.host”。

`nettests` 执行一些其他测试，最终通过（真实的）互联网将 DNS 请求发送到谷歌的一个名称服务器。您应该确保您的代码通过所有这些测试，然后您应该看到以下输出：

```
$ nettests
nettests running on port 25603
testing ping: OK
testing single-process pings: OK
testing multi-process pings: OK
testing DNS
DNS arecord for pdos.csail.mit.edu. is 128.52.129.126
DNS OK
all tests passed.
```

您应该确保 `make grade` 同意您的解决方案通过。

提示

首先，将打印语句添加到 `e1000_transmit()` 和 `e1000_recv()`，然后运行 `make server` 和（在xv6中）`nettests`。您应该从打印语句中看到，`nettests` 生成对 `e1000_transmit` 的调用。

实现 `e1000_transmit` 的一些提示：

- 首先，通过读取 `E1000_TDT` 控制寄存器，向E1000询问等待下一个数据包的TX环索引。
- 然后检查环是否溢出。如果 `E1000_RXD_STAT_DD` 未在 `E1000_TDT` 索引的描述符中设置，则E1000尚未完成先前相应的传输请求，因此返回错误。
- 否则，使用 `mbufalloc()` 释放从该描述符传输的最后一个 `mbuf`（如果有）。
- 然后填写描述符。`m->head` 指向内存中数据包的内容，`m->len` 是数据包的长度。设置必要的cmd标志（请参阅E1000手册的第3.3节），并保存指向 `mbuf` 的指针，以便稍后释放。
- 最后，通过将一加到 `E1000_TDT` 再对 `RX_RING_SIZE` 取模来更新环位置。
- 如果 `e1000_transmit()` 成功地将 `mbuf` 添加到环中，则返回0。如果失败（例如，没有可用的描述符来传输 `mbuf`），则返回-1，以便调用方知道应该释放 `mbuf`。

实现 `e1000_recv` 的一些提示：

- 首先通过提取 `E1000_RDT` 控制寄存器并加一对 `RX_RING_SIZE` 取模，向E1000询问下一个等待接收数据包（如果有）所在的环索引。
- 然后通过检查描述符 `status` 部分中的 `E1000_RXD_STAT_DD` 位来检查新数据包是否可用。如果不可用，请停止。
- 否则，将 `mbuf` 的 `m->len` 更新为描述符中报告的长度。使用 `net_rx()` 将 `mbuf` 传递到网络栈。
- 然后使用 `mbufalloc()` 分配一个新的 `mbuf`，以替换刚刚给 `net_rx()` 的 `mbuf`。将其数据指针（`m->head`）编程到描述符中。将描述符的状态位清除为零。
- 最后，将 `E1000_RDT` 寄存器更新为最后处理的环描述符的索引。
- `e1000_init()` 使用 `mbufs` 初始化RX环，您需要通过浏览代码来了解它是如何做到这一点的。
- 在某刻，曾经到达的数据包总数将超过环大小（16）；确保你的代码可以处理这个问题。

您将需要锁来应对xv6可能从多个进程使用E1000，或者在中断到达时在内核线程中使用E1000的可能性。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间：2021-09-06 17:17:52

- 实验解析

实验解析

这里存放了各个实验的逐步解析，水平有限，可能并非最优解。

前几个实验当时没有做记录，以后再补上

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间： 2021-08-19 17:10:11

- lab1: Util
 - sleep
 - pingpong
 - primes
 - find
 - xargs

lab1: Util

sleep

这个简单，不多说了

```
#include "kernel/types.h"
#include "user/user.h"

int main(int argc, char const *argv[])
{
    if (argc != 2) { //参数错误
        fprintf(2, "usage: sleep <time>\n");
        exit(1);
    }
    sleep(atoi(argv[1]));
    exit(0);
}
```

pingpong

使用两个管道进行父子进程通信，需要注意的是如果管道的写端没有 `close`，那么管道中数据为空时对管道的读取将会阻塞。因此对于不需要的管道描述符，要尽可能早的关闭。

```

#include "kernel/types.h"
#include "user/user.h"

#define RD 0 //pipe的read端
#define WR 1 //pipe的write端

int main(int argc, char const *argv[]) {
    char buf = 'P'; //用于传送的字节

    int fd_c2p[2]; //子进程->父进程
    int fd_p2c[2]; //父进程->子进程
    pipe(fd_c2p);
    pipe(fd_p2c);

    int pid = fork();
    int exit_status = 0;

    if (pid < 0) {
        fprintf(2, "fork() error!\n");
        close(fd_c2p[RD]);
        close(fd_c2p[WR]);
        close(fd_p2c[RD]);
        close(fd_p2c[WR]);
        exit(1);
    } else if (pid == 0) { //子进程
        close(fd_p2c[WR]);
        close(fd_c2p[RD]);

        if (read(fd_p2c[RD], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "child read() error!\n");
            exit_status = 1; //标记出错
        } else {
            fprintf(1, "%d: received ping\n", getpid());
        }

        if (write(fd_c2p[WR], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "child write() error!\n");
            exit_status = 1;
        }

        close(fd_p2c[RD]);
        close(fd_c2p[WR]);

        exit(exit_status);
    } else { //父进程
        close(fd_p2c[RD]);
        close(fd_c2p[WR]);

        if (write(fd_p2c[WR], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "parent write() error!\n");
            exit_status = 1;
        }

        if (read(fd_c2p[RD], &buf, sizeof(char)) != sizeof(char)) {
            fprintf(2, "parent read() error!\n");
            exit_status = 1; //标记出错
        } else {
            fprintf(1, "%d: received pong\n", getpid());
        }

        close(fd_p2c[WR]);
        close(fd_c2p[RD]);

        exit(exit_status);
    }
}

```

```
    }  
}
```

primes

这个感觉还是有些难度的，它的思想是多进程版本的递归，不断地将左邻居管道中的数据筛选后传送给右邻居，每次传送的第一个数据都将是一个素数。

具体还是看代码吧，里面注释应该还是比较清楚的

```

#include "kernel/types.h"
#include "user/user.h"

#define RD 0
#define WR 1

const uint INT_LEN = sizeof(int);

/***
 * @brief 读取左邻居的第一个数据
 * @param lpipe 左邻居的管道符
 * @param pfist 用于存储第一个数据的地址
 * @return 如果没有数据返回-1,有数据返回0
 */
int lpipe_first_data(int lpipe[2], int *dst)
{
    if (read(lpipe[RD], dst, sizeof(int)) == sizeof(int)) {
        printf("prime %d\n", *dst);
        return 0;
    }
    return -1;
}

/***
 * @brief 读取左邻居的数据，将不能被first整除的写入右邻居
 * @param lpipe 左邻居的管道符
 * @param rpipe 右邻居的管道符
 * @param first 左邻居的第一个数据
 */
void transmit_data(int lpipe[2], int rpipe[2], int first)
{
    int data;
    // 从左管道读取数据
    while (read(lpipe[RD], &data, sizeof(int)) == sizeof(int)) {
        // 将无法整除的数据传递入右管道
        if (data % first)
            write(rpipe[WR], &data, sizeof(int));
    }
    close(lpipe[RD]);
    close(rpipe[WR]);
}

/***
 * @brief 寻找素数
 * @param lpipe 左邻居管道
 */
void primes(int lpipe[2])
{
    close(lpipe[WR]);
    int first;
    if (lpipe_first_data(lpipe, &first) == 0) {
        int p[2];
        pipe(p); // 当前的管道
        transmit_data(lpipe, p, first);

        if (fork() == 0) {
            primes(p); // 递归的思想，但这将在一个新的进程中调用
        } else {
            close(p[RD]);
            wait(0);
        }
    }
    exit(0);
}

```

```
int main(int argc, char const *argv[])
{
    int p[2];
    pipe(p);

    for (int i = 2; i <= 35; ++i) //写入初始数据
        write(p[WR], &i, INT_LEN);

    if (fork() == 0) {
        primes(p);
    } else {
        close(p[WR]);
        close(p[RD]);
        wait(0);
    }

    exit(0);
}
```

find

感觉没什么好说的0.0，代码基本上都是COPY的ls.c中的内容

```

#include "kernel/types.h"

#include "kernel/fs.h"
#include "kernel/stat.h"
#include "user/user.h"

void find(char *path, const char *filename)
{
    char buf[512], *p;
    int fd;
    struct dirent de;
    struct stat st;

    if ((fd = open(path, 0)) < 0) {
        fprintf(2, "find: cannot open %s\n", path);
        return;
    }

    if (fstat(fd, &st) < 0) {
        fprintf(2, "find: cannot fstat %s\n", path);
        close(fd);
        return;
    }

    //参数错误, find的第一个参数必须是目录
    if (st.type != T_DIR) {
        fprintf(2, "usage: find <DIRECTORY> <filename>\n");
        return;
    }

    if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf) {
        fprintf(2, "find: path too long\n");
        return;
    }
    strcpy(buf, path);
    p = buf + strlen(buf);
    *p++ = '/'; //p指向最后一个'/'之后
    while (read(fd, &de, sizeof de) == sizeof de) {
        if (de.inum == 0)
            continue;
        memmove(p, de.name, DIRSIZ); //添加路径名称
        p[DIRSIZ] = 0;           //字符串结束标志
        if (stat(buf, &st) < 0) {
            fprintf(2, "find: cannot stat %s\n", buf);
            continue;
        }
        //不要在“.”和“..”目录中递归
        if (st.type == T_DIR && strcmp(p, ".") != 0 && strcmp(p, "..") != 0) {
            find(buf, filename);
        } else if (strcmp(filename, p) == 0)
            printf("%s\n", buf);
    }

    close(fd);
}

int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(2, "usage: find <directory> <filename>\n");
        exit(1);
    }
    find(argv[1], argv[2]);
    exit(0);
}

```

xargs

之前这个题目我是做错的，但是仍然通过了测试。需要注意的是题目中要求为每一行执行一个命令。之前刷力扣的时候处理字符串好几次用到了有限状态自动机，虽然写的代码比较多，但是只要搞清楚逻辑，这种方法反而比较容易写出来。

xv6中的 echo 命令并不能输出换行符，例如在xv6和linux中执行命令

xv6执行 echo "1\n2" 输出

```
"1\n2"
```

linux执行 echo -e "1\n2" 输出（-e 启用转义）

```
1
```

```
2
```

因此没有想到如何在xv6中验证 xargs 的正确性，于是将类似（类似是因为头文件不同，且将 exec 替换为了 execvp ）的代码xargs.c在linux下编译并测试运行，如下图所示：第一条命令是使用linux中 xargs 的输出，第二条命令是使用自己写的 xargs 的输出，二者是一致的。

```
~/T/c >>> echo -e " 1 \n 2\n 3456 " | xargs -n 1 echo line
line 1
line 2
line 3456
~/T/c >>> echo -e " 1 \n 2\n 3456 " | ./xargs echo line
line 1
line 2
line 3456
```

有限状态自动机主要就是一系列的状态转换，例如对于

```
1 \n 23
0123 456
```

来说，第一行是待读取的字符串，第二行是字符下标，起始时状态为 S_WAIT，状态转换如下

```
读取到0处的空格 状态由S_WAIT变为S_WAIT，继续等待参数到来（arg_beg前移）
读取到1处的字符1 状态由S_WAIT变为S_ARG，开始读取参数（arg_beg不动）
读取到2处的空格 状态由S_ARG变为S_ARG_END，储存参数地址（将lines[arg_beg]的地址存入x_argv中）
读取到3处的换行 状态由S_ARG_END变为S_LINE_END，fork后执行程序
...以此类推
```

```

#include "kernel/types.h"
#include "user/user.h"
#include "kernel/param.h"

#define MAXSZ 512
// 有限状态自动机状态定义
enum state {
    S_WAIT,           // 等待参数输入，此状态为初始状态或当前字符为空格
    S_ARG,            // 参数内
    S_ARG_END,        // 参数结束
    S_ARG_LINE_END,   // 左侧有参数的换行，例如"arg\n"
    S_LINE_END,       // 左侧为空格的换行，例如"\n"
    S_END             // 结束，EOF
};

// 字符类型定义
enum char_type {
    C_SPACE,
    C_CHAR,
    C_LINE_END
};

/***
 * @brief 获取字符类型
 *
 * @param c 待判定的字符
 * @return enum char_type 字符类型
 */
enum char_type get_char_type(char c)
{
    switch (c) {
    case ' ':
        return C_SPACE;
    case '\n':
        return C_LINE_END;
    default:
        return C_CHAR;
    }
}

/***
 * @brief 状态转换
 *
 * @param cur 当前的状态
 * @param ct 将要读取的字符
 * @return enum state 转换后的状态
 */
enum state transform_state(enum state cur, enum char_type ct)
{
    switch (cur) {
    case S_WAIT:
        if (ct == C_SPACE)    return S_WAIT;
        if (ct == C_LINE_END) return S_LINE_END;
        if (ct == C_CHAR)     return S_ARG;
        break;
    case S_ARG:
        if (ct == C_SPACE)    return S_ARG_END;
        if (ct == C_LINE_END) return S_ARG_LINE_END;
        if (ct == C_CHAR)     return S_ARG;
        break;
    case S_ARG_END:
    case S_ARG_LINE_END:
    case S_LINE_END:
        if (ct == C_SPACE)    return S_WAIT;
        if (ct == C_LINE_END) return S_LINE_END;

```

```

        if (ct == C_CHAR)      return S_ARG;
        break;
    default:
        break;
    }
    return S_END;
}

/**
 * @brief 将参数列表后面的元素全部置为空
 *         用于换行时，重新赋予参数
 *
 * @param x_argv 参数指针数组
 * @param beg 要清空的起始下标
 */
void clearArgv(char *x_argv[MAXARG], int beg)
{
    for (int i = beg; i < MAXARG; ++i)
        x_argv[i] = 0;
}

int main(int argc, char *argv[])
{
    if (argc - 1 >= MAXARG) {
        fprintf(2, "xargs: too many arguments.\n");
        exit(1);
    }
    char lines[MAXSZ];
    char *p = lines;
    char *x_argv[MAXARG] = {0}; // 参数指针数组，全部初始化为空指针

    // 存储原有的参数
    for (int i = 1; i < argc; ++i) {
        x_argv[i - 1] = argv[i];
    }
    int arg_beg = 0;           // 参数起始下标
    int arg_end = 0;           // 参数结束下标
    int arg_cnt = argc - 1;   // 当前参数索引
    enum state st = S_WAIT;  // 起始状态置为S_WAIT

    while (st != S_END) {
        // 读取为空则退出
        if (read(0, p, sizeof(char)) != sizeof(char)) {
            st = S_END;
        } else {
            st = transform_state(st, get_char_type(*p));
        }

        if (++arg_end >= MAXSZ) {
            fprintf(2, "xargs: arguments too long.\n");
            exit(1);
        }

        switch (st) {
        case S_WAIT:           // 这种情况下只需要让参数起始指针前移
            ++arg_beg;
            break;
        case S_ARG_END:         // 参数结束，将参数地址存入x_argv数组中
            x_argv[arg_cnt++] = &lines[arg_beg];
            arg_beg = arg_end;
            *p = '\0';          // 替换为字符串结束符
            break;
        case S_ARG_LINE_END:   // 将参数地址存入x_argv数组中同时执行指令
            x_argv[arg_cnt++] = &lines[arg_beg];
            // 不加break，因为后续处理同S_LINE_END
        }
    }
}

```

```
case S_LINE_END:      // 行结束，则为当前行执行指令
    arg_beg = arg_end;
    *p = '\0';
    if (fork() == 0) {
        exec(argv[1], x_argv);
    }
    arg_cnt = argc - 1;
    clearArgv(x_argv, arg_cnt);
    wait(0);
    break;
default:
    break;
}

++p;      // 下一个字符的存储位置后移
}
exit(0);
}
```

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2022-03-09 22:04:39

- lab2: syscall
 - trace
 - sysinfo

lab2: syscall

trace

本实验主要是实现一个追踪系统调用的函数，那么首先根据提示定义 `trace` 系统调用，并修复编译错误。

首先看一下 `user/trace.c` 的内容，主要的代码如下

```

if (trace(atoi(argv[1])) < 0) {
    fprintf(2, "%s: trace failed\n", argv[0]);
    exit(1);
}
for(i = 2; i < argc && i < MAXARG; i++){
    nargv[i-2] = argv[i];
}
exec(nargv[0], nargv);

```

它首先调用 `trace(int)`，然后将命令行中的参数 `argv` 复制到 `nargv` 中，同时删去前两个参数，例如

```

argv = trace 32 grep hello README
nargv = grep hello README

```

那么，根据提示，我们首先再 `proc` 结构体中添加一个数据字段，用于保存 `trace` 的参数。并在 `sys_trace()` 的实现中实现参数的保存

```

// kernel/proc.h
struct proc {
    // ...
    int trace_mask;    // trace系统调用参数
};

// kernel/sysproc.c
uint64
sys_trace(void)
{
    // 获取系统调用的参数
    argint(0, &(myproc()->trace_mask));
    return 0;
}

```

接下来应当考虑如何进行系统调用追踪了，根据提示，这将在 `syscall()` 函数中实现。下面是实现代码，需要注意的是条件判断中使用了 `&` 而不是 `==`，这是因为在实验说明书的例子中，`trace 2147483647 grep hello README` 将所有31个低位置为1，使得其可以追踪所有的系统调用。

```

void
syscall(void)
{
    int num;
    struct proc *p = myproc();

    num = p->trapframe->a7; // 系统调用编号, 参见书中4.3节
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        p->trapframe->a0 = syscalls[num](); // 执行系统调用, 然后将返回值存入a0

        // 系统调用是否匹配
        if ((1 << num) & p->trace_mask)
            printf("%d: syscall %s -> %d\n", p->pid, syscalls_name[num], p->trapframe->a0);
    } else {
        printf("%d %s: unknown sys call %d\n",
               p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}

```

在上面的代码中，我们还有一些引用的变量尚未定义，在`syscall.c`中定义他们

```

// ...
extern uint64 sys_trace(void);

static uint64 (*syscalls[])(void) = {
// ...
[SYS_trace]    sys_trace,
};

static char *syscalls_name[] = {
[SYS_fork]      "fork",
[SYS_exit]      "exit",
[SYS_wait]      "wait",
[SYS_pipe]      "pipe",
[SYS_read]      "read",
[SYS_kill]      "kill",
[SYS_exec]      "exec",
[SYS_fstat]     "fstat",
[SYS_chdir]     "chdir",
[SYS_dup]       "dup",
[SYS_getpid]    "getpid",
[SYS_sbrk]       "sbrk",
[SYS_sleep]     "sleep",
[SYS_uptime]    "uptime",
[SYS_open]      "open",
[SYS_write]     "write",
[SYS_mknod]     "mknod",
[SYS_unlink]    "unlink",
[SYS_link]      "link",
[SYS_mkdir]     "mkdir",
[SYS_close]     "close",
[SYS_trace]     "trace",
};

```

sysinfo

- 在`kernel/kalloc.c`中添加一个函数用于获取空闲内存量

```

struct run {
    struct run *next;
};

struct {
    spinlock lock;
    struct run *freelist;
} kmem;

```

内存是使用链表进行管理的，因此遍历 `kmem` 中的空闲链表就能够获取所有的空闲内存，如下

```

void
freebytes(uint64 *dst)
{
    *dst = 0;
    struct run *p = kmem.freelist; // 用于遍历

    acquire(&kmem.lock);
    while (p) {
        *dst += PGSIZE;
        p = p->next;
    }
    release(&kmem.lock);
}

```

- 在 `kernel/proc.c` 中添加一个函数获取进程数

遍历 `proc` 数组，统计处于活动状态的进程即可，循环的写法参考 `scheduler` 函数

```

void
procnum(uint64 *dst)
{
    *dst = 0;
    struct proc *p;
    for (p = proc; p < &proc[NPROC]; p++) {
        if (p->state != UNUSED)
            (*dst)++;
    }
}

```

- 实现 `sys_sysinfo`，将数据写入结构体并传递到用户空间

```
uint64
sys_sysinfo(void)
{
    struct sysinfo info;
    freebytes(&info.freemem);
    procnum(&info.nproc);

    // 获取虚拟地址
    uint64 dstaddr;
    argaddr(0, &dstaddr);

    // 从内核空间拷贝数据到用户空间
    if (copyout(myproc()->pagetable, dstaddr, (char *)&info, sizeof info) < 0)
        return -1;

    return 0;
}
```

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2022-04-09 17:33:19

- [Lab4: Traps](#)
- [1. RISC-V assembly](#)
- [2. Backtrace](#)
- [3. Alarm](#)

Lab4: Traps

1. RISC-V assembly

(1). 在a0-a7中存放参数，13存放在a2中

(2). 在C代码中，main调用f，f调用g。而在生成的汇编中，main函数进行了内联优化处理。

从代码 `li a1,12` 可以看出，main直接计算出了结果并储存

(3). 在 `0x630`

(4). `auipc` (Add Upper Immediate to PC): `auipc rd imm`，将高位立即数加到PC上，从下面的指令格式可以看出，该指令将20位的立即数左移12位之后（右侧补0）加上PC的值，将结果保存到dest位置，图中为 `rd` 寄存器

31	imm[31:12]	20	U-immediate[31:12]	dest	7 6	opcode	0
12 11	rd	5	U-immediate[31:12]	dest	7	LUI	
						AUIPC	

下面来看 `jalr` (jump and link register): `jalr rd, offset(rs1)` 跳转并链接寄存器。
`jalr` 指令会将当前PC+4保存在rd中，然后跳转到指定的偏移地址 `offset(rs1)`。

31	imm[11:0]	20 19	15 14	12 11	7 6	0
12	offset[11:0]	5	base	0	dest	opcode
						JALR

来看XV6的代码：

```
30: 00000097      auipc ra,0x0
34: 600080e7      jalr  1536(ra) # 630 <printf>
```

第一行代码：`00000097H=00...0 0000 1001 0111B`，对比指令格式，可见imm=0，dest=00001，opcode=0010111，对比汇编指令可知，`auipc`的操作码是0010111，ra寄存器代码是00001。这行代码将0x0左移12位（还是0x0）加到PC（当前为0x30）上并存入ra中，即ra中保存的是0x30

第二行代码：`600080e7H=0110 0...0 1000 0000 1110 0111B`，可见imm=0110 0000 0000，rs1=00001，funct3=000，rd=00001，opcode=1100111，rs1和rd的知识码都是00001，即都为寄存器 `ra`。这对比 `jalr` 的标准格式有所不同，可能是此两处使用寄存器相同时，汇编中可以省略 `rd` 部分。

`ra`中保存的是`0x30`, 加上`0x600`后为`0x630`, 即 `printf` 的地址, 执行此行代码后, 将跳转到`printf`函数执行, 并将`PC+4=0X34+0X4=0X38`保存到 `ra` 中, 供之后返回使用。

(5). `57616=0xE110, 0x00646c72`小端存储为`72-6c-64-00`, 对照ASCII码表

`72:r 6c:l 64:d 00:充当字符串结尾标识`

因此输出为: `HE110 World`

若为大端存储, `i`应改为`0x726c6400`, 不需改变`57616`

(6). 原本需要两个参数, 却只传入了一个, 因此`y`=后面打印的结果取决于之前`a2`中保存的数据

2. Backtrace

这个函数就是实现曾经调用函数地址的回溯, 这个功能在日常的编程中也经常见到, 编译器报错时就是类似的逻辑, 只不过题目的要求较为简单, 只用打印程序地址, 而实际的报错中往往打印程序文件名, 函数名以及行号等信息(最后的可选练习就是实现这样的功能)。

```
/**  
 * @brief backtrace 回溯函数调用的返回地址  
 */  
void  
backtrace(void) {  
    printf("backtrace:\n");  
    // 读取当前帧指针  
    uint64 fp = r_fp();  
    while (PGROUNDUP(fp) - PGROUNDDOWN(fp) == PGSIZE) {  
        // 返回地址保存在-8偏移的位置  
        uint64 ret_addr = *(uint64*)(fp - 8);  
        printf("%p\n", ret_addr);  
        // 前一个帧指针保存在-16偏移的位置  
        fp = *(uint64*)(fp - 16);  
    }  
}
```

根据提示: 返回地址位于栈帧帧指针的固定偏移(-8)位置, 并且保存的帧指针位于帧指针的固定偏移(-16)位置。先使用 `r_fp()` 读取当前的帧指针, 然后读出返回地址并打印, 再将 `fp` 定位到前一个帧指针的位置继续读取即可。

根据提示: XV6在内核中以页面对齐的地址为每个栈分配一个页面。使用 `PGROUNDUP(fp) - PGROUNDDOWN(fp) == PGSIZE` 判断当前的 `fp` 是否被分配了一个页面来终止循环。

3. Alarm

这项练习要实现定期的警报。首先是要通过 `test0`, 如何调用处理程序是主要的问题。程序计数器的过程是这样的:

1. `ecall` 指令中将`PC`保存到`SEPC`
2. 在 `usertrap` 中将`SEPC`保存到 `p->trapframe->epc`

3. `p->trapframe->epc` 加4指向下一条指令
4. 执行系统调用
5. 在 `usertrapret` 中将SEPC改写为 `p->trapframe->epc` 中的值
6. 在 `sret` 中将PC设置为SEPC的值

可见执行系统调用后返回到用户空间继续执行的指令地址是由 `p->trapframe->epc` 决定的，因此在 `usertrap` 中主要就是完成它的设置工作。

- (1).** 在 `struct proc` 中增加字段，同时记得在 `allocproc` 中将它们初始化为0，并在 `freeproc` 中也设为0

```
int alarm_interval;           // 报警间隔
void (*alarm_handler)();      // 报警处理函数
int ticks_count;              // 两次报警间的滴答计数
```

- (2).** 在 `sys_sigalarm` 中读取参数

```
uint64
sys_sigalarm(void) {
    if(argint(0, &myproc()->alarm_interval) < 0 || 
        argaddr(1, (uint64*)&myproc()->alarm_handler) < 0)
        return -1;

    return 0;
}
```

- (3).** 修改 `usertrap()`

```
// give up the CPU if this is a timer interrupt.
if(which_dev == 2) {
    if(++p->ticks_count == p->alarm_interval) {
        // 更改陷阱帧中保留的程序计数器
        p->trapframe->epc = (uint64)p->alarm_handler;
        p->ticks_count = 0;
    }
    yield();
}
```

接下来要通过 `test1` 和 `test2`，要解决的主要问题是寄存器保存恢复和防止重复执行的问题。考虑一下没有 `alarm` 时运行的大致过程

1. 进入内核空间，保存用户寄存器到进程陷阱帧
2. 陷阱处理过程
3. 恢复用户寄存器，返回用户空间

而当添加了 `alarm` 后，变成了以下过程

1. 进入内核空间，保存用户寄存器到进程陷阱帧
2. 陷阱处理过程
3. 恢复用户寄存器，返回用户空间，但此时返回的并不是进入陷阱时的程序地址，而是处理函数 `handler` 的地址，而 `handler` 可能会改变用户寄存器

因此我们要在 `usertrap` 中再次保存用户寄存器，当 `handler` 调用 `sigreturn` 时将其恢复，并且要防止在 `handler` 执行过程中重复调用，过程如下

- (1).** 再在 `struct proc` 中新增两个字段

```
int is_alarmming; // 是否正在执行告警处理函数
struct trapframe* alarm_trapframe; // 告警陷阱帧
```

(2). 在allocproc和freeproc中设定好相关分配，回收内存的代码

```
/*
 * allocproc.c
 */
// 初始化告警字段
if((p->alarm_trapframe = (struct trapframe*)kalloc()) == 0) {
    freeproc(p);
    release(&p->lock);
    return 0;
}
p->is_alarmming = 0;
p->alarm_interval = 0;
p->alarm_handler = 0;
p->ticks_count = 0;

/*
 * freeproc.c
 */
if(p->alarm_trapframe)
    kfree((void*)p->alarm_trapframe);
p->alarm_trapframe = 0;
p->is_alarmming = 0;
p->alarm_interval = 0;
p->alarm_handler = 0;
p->ticks_count = 0;
```

(3). 更改usertrap函数，保存进程陷阱帧 p->trapframe 到 p->alarm_trapframe

```
// give up the CPU if this is a timer interrupt.
if(which_dev == 2) {
    if(p->alarm_interval != 0 && ++p->ticks_count == p->alarm_interval && p->is_alarmming
        // 保存寄存器内容
        memmove(p->alarm_trapframe, p->trapframe, sizeof(struct trapframe));
        // 更改陷阱帧中保留的程序计数器，注意一定要在保存寄存器内容后再设置epc
        p->trapframe->epc = (uint64)p->alarm_handler;
        p->ticks_count = 0;
        p->is_alarmming = 1;
    }
    yield();
}
```

(4). 更改 sys_sigreturn，恢复陷阱帧

```
uint64
sys_sigreturn(void) {
    memmove(myproc()->trapframe, myproc()->alarm_trapframe, sizeof(struct trapframe));
    myproc()->is_alarmming = 0;
    return 0;
}
```

- Lab5: xv6 lazy page allocation
- Eliminate allocation from sbrk()
- Lazy allocation
- Lazytests and Usertests
 - 为什么使用两个continue

Lab5: xv6 lazy page allocation

Eliminate allocation from sbrk()

这个实验很简单，就仅仅改动 `sys_sbrk()` 函数即可，将实际分配内存的函数删除，而仅仅改变进程的 `sz` 属性

```
uint64
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;

    // lazy allocation
    myproc()->sz += n;

    return addr;
}
```

Lazy allocation

根据提示来做就好，另外6.S081对应的视频课程中对这部分代码做出了很大一部分的解答。

(1). 修改 `usertrap()` (***kernel/trap.c***) 函数，使用 `r_scause()` 判断是否为页面错误，在页面错误处理的过程中，先判断发生错误的虚拟地址（`r_stval()` 读取）是否位于栈空间之上，进程大小（虚拟地址从0开始，进程大小表征了进程的最高虚拟地址）之下，然后分配物理内存并添加映射

```

uint64 cause = r_scause();
if(cause == 8) {
    ...
} else if((which_dev = devintr()) != 0) {
    // ok
} else if(cause == 13 || cause == 15) {
    // 处理页面错误
    uint64 fault_va = r_stval(); // 产生页面错误的虚拟地址
    char* pa; // 分配的物理地址
    if(PGROUNDUP(p->trapframe->sp) - 1 < fault_va && fault_va < p->sz &&
        (pa = kalloc()) != 0) {
        memset(pa, 0, PGSIZE);
        if(mappages(p->pagetable, PGROUNDDOWN(fault_va), PGSIZE, (uint64)pa, PTE_R | PTE_W)) {
            kfree(pa);
            p->killed = 1;
        }
    } else {
        // printf("usertrap(): out of memory!\n");
        p->killed = 1;
    }
} else {
    ...
}

```

(2). 修改 `uvmunmap()` (*kernel/vm.c*)，之所以修改这部分代码是因为lazy allocation中首先并未实际分配内存，所以当解除映射关系的时候对于这部分内存要略过，而不是使系统崩溃，这部分在课程视频中已经解答。

```

void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
    ...

    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
        if((pte = walk(pagetable, a, 0)) == 0)
            panic("uvmunmap: walk");
        if((*pte & PTE_V) == 0)
            continue;

        ...
    }
}

```

Lazytests and Usertests

(1). 处理 `sbrk()` 参数为负数的情况，参考之前 `sbrk()` 调用的 `growproc()` 程序，如果为负数，就调用 `uvmdealloc()` 函数，但需要限制缩减后的内存空间不能小于0

```

uint64
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;

    struct proc* p = myproc();
    addr = p->sz;
    uint64 sz = p->sz;

    if(n > 0) {
        // lazy allocation
        p->sz += n;
    } else if(sz + n > 0) {
        sz = uvmdealloc(p->pagetable, sz, sz + n);
        p->sz = sz;
    } else {
        return -1;
    }
    return addr;
}

```

(2). 正确处理 fork 的内存拷贝： fork 调用了 uvmcopy 进行内存拷贝，所以修改 uvmcopy 如下

```

int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    ...
    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walk(old, i, 0)) == 0)
            continue;
        if((*pte & PTE_V) == 0)
            continue;
        ...
    }
    ...
}

```

(3). 还需要继续修改 uvmunmap，否则会运行出错，关于为什么要使用两个 continue，请看本文最下面

```

void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
    ...

    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
        if((pte = walk(pagetable, a, 0)) == 0)
            continue;
        if((*pte & PTE_V) == 0)
            continue;

        ...
    }
}

```

(4). 处理通过 `sbrk` 申请内存后还未实际分配就传给系统调用使用的情况，系统调用的处理会陷入内核，`scause` 寄存器存储的值是 8，如果此时传入的地址还未实际分配，就不能走到上文 `usertrap` 中判断 `scause` 是 13 或 15 后进行内存分配的代码，`syscall` 执行就会失败

- 系统调用流程：

- 陷入内核 ==> `usertrap` 中 `r_scause() == 8` 的分支 ==> `syscall()` ==> 回到用户空间

- 页面错误流程：

- 陷入内核 ==> `usertrap` 中 `r_scause() == 13 || r_scause() == 15` 的分支 ==> 分配内存 ==> 回到用户空间

因此就需要找到在何时系统调用会使用这些地址，将地址传入系统调用后，会通过 `argaddr` 函数(**kernel/syscall.c**)从寄存器中读取，因此在这里添加物理内存分配的代码

```
int
argaddr(int n, uint64 *ip)
{
    *ip = argraw(n);
    struct proc* p = myproc();

    // 处理向系统调用传入 lazy allocation 地址的情况
    if(walkaddr(p->pagetable, *ip) == 0) {
        if(PGROUNDDOWN(p->trapframe->sp) - 1 < *ip && *ip < p->sz) {
            char* pa = kalloc();
            if(pa == 0)
                return -1;
            memset(pa, 0, PGSIZE);

            if(mappages(p->pagetable, PGROUNDDOWN(*ip), PGSIZE, (uint64)pa, PTE_R | PTE_W | PTE_U))
                kfree(pa);
            return -1;
        }
    } else {
        return -1;
    }
}

return 0;
}
```

为什么使用两个 `continue`

这里需要解释一下为什么在两个判断中使用了 `continue` 语句，在课程视频中仅仅添加了第二个 `continue`，利用 `vmprint` 打印出来初始时刻用户进程的页表如下

```

page table 0x0000000087f55000
..0: pte 0x0000000021fd3c01 pa 0x0000000087f4f000
... ..0: pte 0x0000000021fd4001 pa 0x0000000087f50000
... ... ..0: pte 0x0000000021fd445f pa 0x0000000087f51000
... ... ...1: pte 0x0000000021fd4cdf pa 0x0000000087f53000
... ... ...2: pte 0x0000000021fd900f pa 0x0000000087f64000
... ... ...3: pte 0x0000000021fd5cdf pa 0x0000000087f57000
...255: pte 0x0000000021fd5001 pa 0x0000000087f54000
... ...511: pte 0x0000000021fd4801 pa 0x0000000087f52000
... ... ...510: pte 0x0000000021fd58c7 pa 0x0000000087f56000
... ... ...511: pte 0x0000000020001c4b pa 0x0000000080007000

```

除去高地址的trapframe和_trampoline页面，进程共计映射了4个有效页面，即添加了映射关系的虚拟地址范围是 `0x0000~0x3fff`，假如使用 `sbrk` 又申请了一个页面，由于**lazy allocation**，页表暂时不会改变，而不经过读写操作后直接释放进程，进程将会调用 `uvmunmap` 函数，此时将会发生什么呢？

`uvmunmap` 首先使用 `walk` 找到虚拟地址对应的PTE地址，虚拟地址的最后12位表征了偏移量，前面每9位索引一级页表，将 `0x4000` 的虚拟地址写为二进制（省略前面的无效位）：

```
{000 0000 00}[00 0000 000](0 0000 0100) 0000 0000 0000
```

- {} : 页目录表索引(`level==2`)，为0
- [] : 二级页表索引(`level==1`)，为0
- () : 三级页表索引(`level==0`)，为4

我们来看一下 `walk` 函数，`walk` 返回指定虚拟地址的PTE，但我认为这个程序存在一定的不足。`walk`函数的代码如下所示

```

pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
    if(va >= MAXVA)
        panic("walk");

    for(int level = 2; level > 0; level--) {
        pte_t *pte = &pagetable[PX(level, va)];
        if(*pte & PTE_V) {
            pagetable = (pagetable_t)PTE2PA(*pte);
        } else {
            if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
                return 0;
            memset(pagetable, 0, PGSIZE);
            *pte = PA2PTE(pagetable) | PTE_V;
        }
    }
    return &pagetable[PX(0, va)];
}

```

这段代码中 `for` 循环执行 `level==2` 和 `level==1` 的情况，而对照刚才打印的页表，`level==2` 时索引为0的项是存在的，`level==1` 时索引为0的项也是存在的，最后执行 `return` 语句，然而`level==0`时索引为4的项却是不存在的，此时 `walk` 不再检查 `PTE_V` 标志等信息，而是直接返回，因此即使虚拟地址对应的PTE实际不存在，`walk` 函数的返回值也可能不为0！

那么返回的这个地址是什么呢？`level`为0时

有效索引为0~3，因此索引为4时返回的是最后一个有效PTE后面的一个地址。

因此我们不能仅靠PTE为0来判断虚拟地址无效，还需要再次检查返回的PTE中是否设置了 PTE_V 标志位。

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 18:06:16

- [Lab6: Copy-on-Write Fork for xv6](#)

Lab6: Copy-on-Write Fork for xv6

跟着提示一步一步来

- (1). 在 ***kernel/riscv.h*** 中选取PTE中的保留位定义标记一个页面是否为COW Fork页面的标志位

```
// 记录应用了COW策略后fork的页面
#define PTE_F (1L << 8)
```

- (2). 在 ***kalloc.c*** 中进行如下修改

- 定义引用计数的全局变量 `ref`，其中包含了一个自旋锁和一个引用计数数组，由于 `ref` 是全局变量，会被自动初始化为全0。

这里使用自旋锁是考虑到这种情况：进程P1和P2共用内存M，M引用计数为2，此时CPU1要执行 `fork` 产生P1的子进程，CPU2要终止P2，那么假设两个CPU同时读取引用计数为2，执行完成后CPU1中保存的引用计数为3，CPU2保存的计数为1，那么后赋值的语句会覆盖掉先赋值的语句，从而产生错误

```
struct ref_struct {
    struct spinlock lock;
    int cnt[PHYSTOP / PGSIZE]; // 引用计数
} ref;
```

- 在 `kinit` 中初始化 `ref` 的自旋锁

```
void
kinit()
{
    initlock(&kmem.lock, "kmem");
    initlock(&ref.lock, "ref");
    freerange(end, (void*)PHYSTOP);
}
```

- 修改 `kalloc` 和 `kfree` 函数，在 `kalloc` 中初始化内存引用计数为1，在 `kfree` 函数中对内存引用计数减1，如果引用计数为0时才真正删除

```

void
kfree(void *pa)
{
    struct run *r;

    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("kfree");

    // 只有当引用计数为0了才回收空间
    // 否则只是将引用计数减1
    acquire(&ref.lock);
    if(--ref.cnt[(uint64)pa / PGSIZE] == 0) {
        release(&ref.lock);

        r = (struct run*)pa;

        // Fill with junk to catch dangling refs.
        memset(pa, 1, PGSIZE);

        acquire(&kmem.lock);
        r->next = kmem.freelist;
        kmem.freelist = r;
        release(&kmem.lock);
    } else {
        release(&ref.lock);
    }
}
void *
kalloc(void)
{
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r) {
        kmem.freelist = r->next;
        acquire(&ref.lock);
        ref.cnt[(uint64)r / PGSIZE] = 1; // 将引用计数初始化为1
        release(&ref.lock);
    }
    release(&kmem.lock);

    if(r)
        memset((char*)r, 5, PGSIZE); // fill with junk
    return (void*)r;
}

```

- 添加如下四个函数，详细说明已在注释中，这些函数中用到了 `walk`，记得在 `defs.h` 中添加声明，最后也需要将这些函数的声明添加到 `defs.h`，在 `cowalloc` 中，读取内存引用计数，如果为1，说明只有当前进程引用了该物理内存（其他进程此前已经被分配到了其他物理页面），就只需要改变PTE使能 `PTE_W`；否则就分配物理页面，并将原来的内存引用计数减1。该函数需要返回物理地址，这将在 `copyout` 中使用到。

```

/**
 * @brief cowpage 判断一个页面是否为COW页面
 * @param pagetable 指定查询的页表
 * @param va 虚拟地址
 * @return 0 是 -1 不是
 */
int cowpage(pagetable_t pagetable, uint64 va) {
    if(va >= MAXVA)
        return -1;
    pte_t* pte = walk(pagetable, va, 0);
    if(pte == 0)
        return -1;
    if((*pte & PTE_V) == 0)
        return -1;
    return (*pte & PTE_F ? 0 : -1);
}

/**
 * @brief cowalloc copy-on-write分配器
 * @param pagetable 指定页表
 * @param va 指定的虚拟地址,必须页面对齐
 * @return 分配后va对应的物理地址,如果返回0则分配失败
 */
void* cowalloc(pagetable_t pagetable, uint64 va) {
    if(va % PGSIZE != 0)
        return 0;

    uint64 pa = walkaddr(pagetable, va); // 获取对应的物理地址
    if(pa == 0)
        return 0;

    pte_t* pte = walk(pagetable, va, 0); // 获取对应的PTE

    if(krefcnt((char*)pa) == 1) {
        // 只剩一个进程对此物理地址存在引用
        // 则直接修改对应的PTE即可
        *pte |= PTE_W;
        *pte &= ~PTE_F;
        return (void*)pa;
    } else {
        // 多个进程对物理内存存在引用
        // 需要分配新的页面,并拷贝旧页面的内容
        char* mem = kalloc();
        if(mem == 0)
            return 0;

        // 复制旧页面内容到新页
        memmove(mem, (char*)pa, PGSIZE);

        // 清除PTE_V,否则在mappages中会判定为remap
        *pte &= ~PTE_V;

        // 为新页面添加映射
        if(mappages(pagetable, va, PGSIZE, (uint64)mem, (PTE_FLAGS(*pte) | PTE_W) & ~PTE_F
                    kfree(mem));
        *pte |= PTE_V;
        return 0;
    }

    // 将原来的物理内存引用计数减1
    kfree((char*)PGROUNDDOWN(pa));
    return mem;
}
}

```

```

/**
 * @brief krefcnt 获取内存的引用计数
 * @param pa 指定的内存地址
 * @return 引用计数
 */
int krefcnt(void* pa) {
    return ref.cnt[(uint64)pa / PGSIZE];
}

/**
 * @brief kaddrefcnt 增加内存的引用计数
 * @param pa 指定的内存地址
 * @return 0:成功 -1:失败
 */
int kaddrefcnt(void* pa) {
    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        return -1;
    acquire(&ref.lock);
    ++ref.cnt[(uint64)pa / PGSIZE];
    release(&ref.lock);
    return 0;
}

```



- 修改 freerange

```

void
freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start);
    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE) {
        // 在kfree中将会对cnt[]减1，这里要先设为1，否则就会减成负数
        ref.cnt[(uint64)p / PGSIZE] = 1;
        kfree(p);
    }
}

```

(3). 修改 `uvncpy`，不为子进程分配内存，而是使父子进程共享内存，但禁用 `PTE_W`，同时标记 `PTE_F`，记得调用 `kaddrefcnt` 增加引用计数

```

int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;

    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walk(old, i, 0)) == 0)
            panic("uvmcopy: pte should exist");
        if((*pte & PTE_V) == 0)
            panic("uvmcopy: page not present");
        pa = PTE2PA(*pte);
        flags = PTE_FLAGS(*pte);

        // 仅对可写页面设置COW标记
        if(flags & PTE_W) {
            // 禁用写并设置COW Fork标记
            flags = (flags | PTE_F) & ~PTE_W;
            *pte = PA2PTE(pa) | flags;
        }

        if(mappages(new, i, PGSIZE, pa, flags) != 0) {
            uvmunmap(new, 0, i / PGSIZE, 1);
            return -1;
        }
        // 增加内存的引用计数
        kaddrrefcnt((char*)pa);
    }
    return 0;
}

```

(4). 修改 usertrap , 处理页面错误

```

uint64 cause = r_scause();
if(cause == 8) {
    ...
} else if((which_dev = devintr()) != 0){
    // ok
} else if(cause == 13 || cause == 15) {
    uint64 fault_va = r_stval(); // 获取出错的虚拟地址
    if(fault_va >= p->sz
        || cowpage(p->pagetable, fault_va) != 0
        || cowalloc(p->pagetable, PGROUNDDOWN(fault_va)) == 0)
        p->killed = 1;
} else {
    ...
}

```

(5). 在 copyout 中处理相同的情况，如果是COW页面，需要更换 pa0 指向的物理地址

```
while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);

    // 处理COW页面的情况
    if(cowpage(pagetable, va0) == 0) {
        // 更换目标物理地址
        pa0 = (uint64)cowalloc(pagetable, va0);
    }

    if(pa0 == 0)
        return -1;

    ...
}
```

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 18:07:55

- [Lab7: Multithreading](#)
- [Uthread: switching between threads](#)
- [Using threads](#)
- [Barrier](#)

Lab7: Multithreading

Uthread: switching between threads

本实验是在给定的代码基础上实现用户级线程切换，相比于XV6中实现的内核级线程，这个要简单许多。因为是用户级线程，不需要设计用户栈和内核栈，用户页表和内核页表等等切换，所以本实验中只需要一个类似于 `context` 的结构，而不需要费尽心机的维护 `trapframe`

(1). 定义存储上下文的结构体 `tcontext`

```
// 用户线程的上下文结构体
struct tcontext {
    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;
};
```

(2). 修改 `thread` 结构体，添加 `context` 字段

```
struct thread {
    char          stack[STACK_SIZE]; /* the thread's stack */
    int           state;            /* FREE, RUNNING, RUNNABLE */
    struct tcontext context;       /* 用户进程上下文 */
};
```

(3). 模仿 `kernel/swtch.S`, 在 `kernel/uthread_switch.S` 中写入如下代码

```

.text

/*
 * save the old thread's registers,
 * restore the new thread's registers.
 */

.globl thread_switch
thread_switch:
    /* YOUR CODE HERE */
    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     /* return to ra */

```

(4). 修改 `thread_scheduler`，添加线程切换语句

```

...
if (current_thread != next_thread) {           /* switch threads? */
...
/* YOUR CODE HERE */
thread_switch((uint64)&t->context, (uint64)&current_thread->context);
} else
next_thread = 0;

```

(5). 在 `thread_create` 中对 `thread` 结构体做一些初始化设定，主要是 `ra` 返回地址和 `sp` 栈指针，其他的都不重要

```

// YOUR CODE HERE
t->context.ra = (uint64)func;                  // 设定函数返回地址
t->context.sp = (uint64)t->stack + STACK_SIZE; // 设定栈指针

```

Using threads

来看一下程序的运行过程：设定了五个散列桶，根据键除以5的余数决定插入到哪一个散列桶中，插入方法是头插法，下面是图示

不支持在 Docs 外粘贴 block

这个实验比较简单，首先是问为什么造成数据丢失：

假设现在有两个线程T1和T2，两个线程都走到put函数，且假设两个线程中key%NBUCKET相等，即要插入同一个散列桶中。两个线程同时调用insert(key, value, &table[i], table[i])，insert是通过头插法实现的。如果先insert的线程还未返回另一个线程就开始insert，那么前面的数据会被覆盖

因此只需要对插入操作上锁即可

(1). 为每个散列桶定义一个锁，将五个锁放在一个数组中，并进行初始化

```
pthread_mutex_t lock[NBUCKET] = { PTHREAD_MUTEX_INITIALIZER }; // 每个散列桶一把锁
```

(2). 在 put 函数中对 insert 上锁

```
if(e){
    // update the existing key.
    e->value = value;
} else {
    pthread_mutex_lock(&lock[i]);
    // the new is new.
    insert(key, value, &table[i], table[i]);
    pthread_mutex_unlock(&lock[i]);
}
```

Barrier

额。。。这个也比较简单，只要保证下一个round的操作不会影响到上一个还未结束的round中的数据就可

```
static void
barrier()
{
    // 申请持有锁
    pthread_mutex_lock(&bstate.barrier_mutex);

    bstate.nthread++;
    if(bstate.nthread == nthread) {
        // 所有线程已到达
        bstate.round++;
        bstate.nthread = 0;
        pthread_cond_broadcast(&bstate.barrier_cond);
    } else {
        // 等待其他线程
        // 调用pthread_cond_wait时，mutex必须已经持有
        pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
    }
    // 释放锁
    pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-19 18:08:34

- [Lab8: Locks](#)
- [Memory allocator](#)
- [Buffer cache](#)

Lab8: Locks

Memory allocator

本实验完成的任务是为每个CPU都维护一个空闲列表，初始时将所有的空闲内存分配到某个CPU，此后各个CPU需要内存时，如果当前CPU的空闲列表上没有，则窃取其他CPU的。例如，所有的空闲内存初始分配到CPU0，当CPU1需要内存时就会窃取CPU0的，而使用完成后就挂在CPU1的空闲列表，此后CPU1再次需要内存时就可以从自己的空闲列表中取。

(1). 将 `kmem` 定义为一个数组，包含 `NCPU` 个元素，即每个CPU对应一个

```
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem[NCPU];
```

(2). 修改 `kinit`，为所有锁初始化以“`kmem`”开头的名称，该函数只会被一个CPU调用，`freerange` 调用 `kfree` 将所有空闲内存挂在该CPU的空闲列表上

```
void
kinit()
{
    char lockname[8];
    for(int i = 0; i < NCPU; i++) {
        snprintf(lockname, sizeof(lockname), "kmem_%d", i);
        initlock(&kmem[i].lock, lockname);
    }
    freerange(end, (void*)PHYSTOP);
}
```

(3). 修改 `kfree`，使用 `cpuid()` 和它返回的结果时必须关中断，请参考《XV6使用手册》第7.4节

```

void
kfree(void *pa)
{
    struct run *r;

    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("kfree");

    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);

    r = (struct run*)pa;

    push_off(); // 关中断
    int id = cpuid();
    acquire(&kmem[id].lock);
    r->next = kmem[id].freelist;
    kmem[id].freelist = r;
    release(&kmem[id].lock);
    pop_off(); //开中断
}

```

(4). 修改 `kalloc`，使得在当前CPU的空闲列表没有可分配内存时窃取其他内存的

```

void *
kalloc(void)
{
    struct run *r;

    push_off(); // 关中断
    int id = cpuid();
    acquire(&kmem[id].lock);
    r = kmem[id].freelist;
    if(r)
        kmem[id].freelist = r->next;
    else {
        int antid; // another id
        // 遍历所有CPU的空闲列表
        for(antid = 0; antid < NCPU; ++antid) {
            if(antid == id)
                continue;
            acquire(&kmem[antid].lock);
            r = kmem[antid].freelist;
            if(r) {
                kmem[antid].freelist = r->next;
                release(&kmem[antid].lock);
                break;
            }
            release(&kmem[antid].lock);
        }
        release(&kmem[id].lock);
        pop_off(); //开中断
    }

    if(r)
        memset((char*)r, 5, PGSIZE); // fill with junk
    return (void*)r;
}

```

Buffer cache

这个实验的目的是将缓冲区的分配与回收并行化以提高效率，这个实验折腾了一天，有些内容还是比较绕的，

(1). 定义哈希桶结构，并在 `bcache` 中删除全局缓冲区链表，改为使用素数个散列桶

```
#define NBUCKET 13
#define HASH(id) (id % NBUCKET)

struct hashbuf {
    struct buf head;           // 头节点
    struct spinlock lock;     // 锁
};

struct {
    struct buf buf[NBUF];
    struct hashbuf buckets[NBUCKET]; // 散列桶
} bcache;
```

(2). 在 `binit` 中，(1) 初始化散列桶的锁，(2) 将所有散列桶的 `head->prev`、`head->next` 都指向自身表示为空，(3) 将所有的缓冲区挂载到 `bucket[0]` 桶上，代码如下

```
void
binit(void) {
    struct buf* b;
    char lockname[16];

    for(int i = 0; i < NBUCKET; ++i) {
        // 初始化散列桶的自旋锁
        snprintf(lockname, sizeof(lockname), "bcache_%d", i);
        initlock(&bcache.buckets[i].lock, lockname);

        // 初始化散列桶的头节点
        bcache.buckets[i].head.prev = &bcache.buckets[i].head;
        bcache.buckets[i].head.next = &bcache.buckets[i].head;
    }

    // Create linked list of buffers
    for(b = bcache.buf; b < bcache.buf + NBUF; b++) {
        // 利用头插法初始化缓冲区列表，全部放到散列桶0上
        b->next = bcache.buckets[0].head.next;
        b->prev = &bcache.buckets[0].head;
        initsleeplock(&b->lock, "buffer");
        bcache.buckets[0].head.next->prev = b;
        bcache.buckets[0].head.next = b;
    }
}
```

(3). 在 `buf.h` 中增加新字段 `timestamp`，这里来理解一下这个字段的用途：在原始方案中，每次 `brelse` 都将被释放的缓冲区挂载到链表头，表明这个缓冲区最近刚刚被使用过，在 `bget` 中分配时从链表尾向前查找，这样符合条件的第一个就是最久未使用的。而在提示中建议使用时间戳作为LRU判定的法则，这样我们就无需在 `brelse` 中进行头插法更改结点位置

```
struct buf {
    ...
    ...
    uint timestamp; // 时间戳
};
```

(4). 更改 `brelse`，不再获取全局锁

```
void
brelse(struct buf* b) {
    if(!holdingsleep(&b->lock))
        panic("brelse");

    int bid = HASH(b->blockno);

    releasesleep(&b->lock);

    acquire(&bcache.buckets[bid].lock);
    b->refcnt--;

    // 更新时间戳
    // 由于LRU改为使用时间戳判定，不再需要头插法
    acquire(&tickslock);
    b->timestamp = ticks;
    release(&tickslock);

    release(&bcache.buckets[bid].lock);
}
```

(5). 更改 `bget`，当没有找到指定的缓冲区时进行分配，分配方式是优先从当前列表遍历，找到一个没有引用且 `timestamp` 最小的缓冲区，如果没有就申请下一个桶的锁，并遍历该桶，找到后将该缓冲区从原来的桶移动到当前桶中，最多将所有桶都遍历完。在代码中要注意锁的释放

```

static struct buf*
bget(uint dev, uint blockno) {
    struct buf* b;

    int bid = HASH(blockno);
    acquire(&bcache.buckets[bid].lock);

    // Is the block already cached?
    for(b = bcache.buckets[bid].head.next; b != &bcache.buckets[bid].head; b = b->next)
        if(b->dev == dev && b->blockno == blockno) {
            b->refcnt++;

            // 记录使用时间戳
            acquire(&tickslock);
            b->timestamp = ticks;
            release(&tickslock);

            release(&bcache.buckets[bid].lock);
            acquiresleep(&b->lock);
            return b;
        }
    }

    // Not cached.
    b = 0;
    struct buf* tmp;

    // Recycle the least recently used (LRU) unused buffer.
    // 从当前散列桶开始查找
    for(int i = bid, cycle = 0; cycle != NBUCKET; i = (i + 1) % NBUCKET) {
        ++cycle;
        // 如果遍历到当前散列桶，则不重新获取锁
        if(i != bid) {
            if(!holding(&bcache.buckets[i].lock))
                acquire(&bcache.buckets[i].lock);
            else
                continue;
        }

        for(tmp = bcache.buckets[i].head.next; tmp != &bcache.buckets[i].head; tmp = tmp->
            // 使用时间戳进行LRU算法，而不是根据结点在链表中的位置
            if(tmp->refcnt == 0 && (b == 0 || tmp->timestamp < b->timestamp))
                b = tmp;

            if(b) {
                // 如果是从其他散列桶窃取的，则将其以头插法插入到当前桶
                if(i != bid) {
                    b->next->prev = b->prev;
                    b->prev->next = b->next;
                    release(&bcache.buckets[i].lock);

                    b->next = bcache.buckets[bid].head.next;
                    b->prev = &bcache.buckets[bid].head;
                    bcache.buckets[bid].head.next->prev = b;
                    bcache.buckets[bid].head.next = b;
                }

                b->dev = dev;
                b->blockno = blockno;
                b->valid = 0;
                b->refcnt = 1;

                acquire(&tickslock);
                b->timestamp = ticks;
                release(&tickslock);
            }
        }
    }
}

```

```

        release(&bcache.buckets[bid].lock);
        acquiresleep(&b->lock);
        return b;
    } else {
        // 在当前散列桶中未找到，则直接释放锁
        if(i != bid)
            release(&bcache.buckets[i].lock);
    }

    panic("bget: no buffers");
}

```

(6). 最后将末尾的两个小函数也改一下

```

void
bpin(struct buf* b) {
    int bid = HASH(b->blockno);
    acquire(&bcache.buckets[bid].lock);
    b->refcnt++;
    release(&bcache.buckets[bid].lock);
}

void
bunpin(struct buf* b) {
    int bid = HASH(b->blockno);
    acquire(&bcache.buckets[bid].lock);
    b->refcnt--;
    release(&bcache.buckets[bid].lock);
}

```

踩过的坑：

1. `bget` 中重新分配可能要持有两个锁，如果桶 `a` 持有自己的锁，再申请桶 `b` 的锁，与此同时如果桶 `b` 持有自己的锁，再申请桶 `a` 的锁就会造成死锁！因此代码中使用了 `if(!holding(&bcache.bucket[i].lock))` 来进行检查。此外，代码优先从自己的桶中获取缓冲区，如果自身没有依次向后查找这样的方式也尽可能地避免了前面的情况。
2. 在 `bget` 中搜索缓冲区并在找不到缓冲区时为该缓冲区分配条目必须是原子的！在提示中说 `bget` 如果未找到而进行分配的操作可以是串行化的，也就是说多个CPU中未找到，应当串行的执行分配，同时还应当避免死锁。于是在发现未命中（`Not cached`）后，我写了如下的代码（此时未删除 `bcache.lock`）

```

// 前半部分查找缓冲区的代码
// Not cached
release(&bcache.buckets[bid].lock);
acquire(&bcache.lock);
acquire(&bcache.buckets[bid].lock);
// 后半部分分配缓冲区的代码

```

这段代码中先释放了散列桶的锁之后再重新获取，之所以这样做是为了让所有代码都保证申请锁的顺序：先获取整个缓冲区的大锁再获取散列桶的小锁，这样才能避免死锁。但是这样做却破坏了程序执行的原子性。

在 `release` 桶的锁并重新 `acquire` 的这段时间，另一个CPU可能也以相同的参数调用了 `bget`，也发现没有该缓冲区并想要执行分配。最终的结果是一个磁盘块对应了两个缓冲区，破坏了最重要的不变量，即每个块最多缓存一个副本。这样会导致 `usertests` 中的 `manywrites` 测试报错：`panic: freeing free block`

copyright by duguosheng all right reserved, powered by Gitbook
该文件修订时间： 2021-08-19 18:09:44

- [Lab9: file system](#)
- [Large files](#)
- [Symbolic links](#)

Lab9: file system

Large files

(1). 在fs.h中添加宏定义

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDINDIRECT ((BSIZE / sizeof(uint)) * (BSIZE / sizeof(uint)))
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
#define NADDR_PER_BLOCK (BSIZE / sizeof(uint)) // 一个块中的地址数量
```

(2). 由于 NDIRECT 定义改变，其中一个直接块变为了二级间接块，需要修改inode结构体中 addrs 元素数量

```
// fs.h
struct dinode {
    ...
    uint addrs[NDIRECT + 2]; // Data block addresses
};

// file.h
struct inode {
    ...
    uint addrs[NDIRECT + 2];
};
```

(3). 修改 bmap 支持二级索引

```

static uint
bmap(struct inode *ip, uint bn)
{
    uint addr, *a;
    struct buf *bp;

    if(bn < NDIRECT){
        ...
    }
    bn -= NDIRECT;

    if(bn < NINDIRECT){
        ...
    }
    bn -= NINDIRECT;

    // 二级间接块的情况
    if(bn < NDINDIRECT) {
        int level2_idx = bn / NADDR_PER_BLOCK; // 要查找的块号位于二级间接块中的位置
        int level1_idx = bn % NADDR_PER_BLOCK; // 要查找的块号位于一级间接块中的位置
        // 读出二级间接块
        if((addr = ip->addrs[NDIRECT + 1]) == 0)
            ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;

        if((addr = a[level2_idx]) == 0) {
            a[level2_idx] = addr = balloc(ip->dev);
            // 更改了当前块的内容，标记以供后续写回磁盘
            log_write(bp);
        }
        brelse(bp);

        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;
        if((addr = a[level1_idx]) == 0) {
            a[level1_idx] = addr = balloc(ip->dev);
            log_write(bp);
        }
        brelse(bp);
        return addr;
    }

    panic("bmap: out of range");
}

```

(4). 修改 `itrunc` 释放所有块

```

void
itrunc(struct inode *ip)
{
    int i, j;
    struct buf *bp;
    uint *a;

    for(i = 0; i < NDIRECT; i++){
        ...
    }

    if(ip->addrs[NDIRECT]){
        ...
    }

    struct buf* bp1;
    uint* a1;
    if(ip->addrs[NDIRECT + 1]) {
        bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
        a = (uint*)bp->data;
        for(i = 0; i < NADDR_PER_BLOCK; i++) {
            // 每一个一级间接块的操作都类似于上面的
            // if(ip->addrs[NDIRECT])中的内容
            if(a[i]) {
                bp1 = bread(ip->dev, a[i]);
                a1 = (uint*)bp1->data;
                for(j = 0; j < NADDR_PER_BLOCK; j++) {
                    if(a1[j])
                        bfree(ip->dev, a1[j]);
                }
                brelse(bp1);
                bfree(ip->dev, a[i]);
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT + 1]);
        ip->addrs[NDIRECT + 1] = 0;
    }

    ip->size = 0;
    iupdate(ip);
}

```

Symbolic links

(1). 配置系统调用的常规操作，如在 **user/usys.pl**、**user/user.h** 中添加一个条目，在 **kernel/syscall.c**、**kernel/syscall.h** 中添加相关内容

(2). 添加提示中的相关定义，`T_SYMLINK` 以及 `O_NOFOLLOW`

```

// fcntl.h
#define O_NOFOLLOW 0x004
// stat.h
#define T_SYMLINK 4

```

(3). 在 **kernel/sysfile.c** 中实现 `sys_symlink`，这里需要注意的是 `create` 返回已加锁的 `inode`，此外 `iunlockput` 既对 `inode` 解锁，还将其引用计数减1，计数为0时回收此 `inode`

```
uint64
sys_symlink(void) {
    char target[MAXPATH], path[MAXPATH];
    struct inode* ip_path;

    if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0) {
        return -1;
    }

    begin_op();
    // 分配一个inode结点, create返回锁定的inode
    ip_path = create(path, T_SYMLINK, 0, 0);
    if(ip_path == 0) {
        end_op();
        return -1;
    }
    // 向inode数据块中写入target路径
    if(writei(ip_path, 0, (uint64)target, 0, MAXPATH) < MAXPATH) {
        iunlockput(ip_path);
        end_op();
        return -1;
    }

    iunlockput(ip_path);
    end_op();
    return 0;
}
```

(4). 修改 sys_open 支持打开符号链接

```

uint64
sys_open(void)
{
    ...

    if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
        ...
    }

    // 处理符号链接
    if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
        // 若符号链接指向的仍然是符号链接，则递归的跟随它
        // 直到找到真正指向的文件
        // 但深度不能超过MAX_SYMLINK_DEPTH
        for(int i = 0; i < MAX_SYMLINK_DEPTH; ++i) {
            // 读出符号链接指向的路径
            if(readi(ip, 0, (uint64)path, 0, MAXPATH) != MAXPATH) {
                iunlockput(ip);
                end_op();
                return -1;
            }
            iunlockput(ip);
            ip = namei(path);
            if(ip == 0) {
                end_op();
                return -1;
            }
            ilock(ip);
            if(ip->type != T_SYMLINK)
                break;
        }
        // 超过最大允许深度后仍然为符号链接，则返回错误
        if(ip->type == T_SYMLINK) {
            iunlockput(ip);
            end_op();
            return -1;
        }
    }

    if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
        ...
    }

    ...
    return fd;
}

```

copyright by duguosheng all right reserved, powered by Gitbook 该文件修订时间： 2021-08-19 18:17:45

- [Lab10: mmap](#)

Lab10: mmap

本实验是实现一个内存映射文件的功能，将文件映射到内存中，从而在与文件交互时减少磁盘操作。

(1). 根据提示1，首先是配置 `mmap` 和 `munmap` 系统调用，此前已进行过多次类似流程，不再赘述。在 `kernel/fcntl.h` 中定义了宏，只有在定义了 `LAB_MMAP` 时这些宏才生效，而 `LAB_MMAP` 是在编译时在命令行通过 `gcc` 的 `-D` 参数定义的

```
void* mmap(void* addr, int length, int prot, int flags, int fd, int offset);
int munmap(void* addr, int length);
```

(2). 根据提示3，定义VMA结构体，并添加到进程结构体中

```
#define NVMA 16
// 虚拟内存区域结构体
struct vm_area {
    int used;           // 是否已被使用
    uint64 addr;        // 起始地址
    int len;            // 长度
    int prot;           // 权限
    int flags;          // 标志位
    int vfd;            // 对应的文件描述符
    struct file* vfile; // 对应文件
    int offset;         // 文件偏移，本实验中一直为0
};

struct proc {
    ...
    struct vm_area vma[NVMA]; // 虚拟内存区域
}
```

(3). 在 `allocproc` 中将 `vma` 数组初始化为全0

```
static struct proc*
allocproc(void)
{
    ...

found:
    ...

    memset(&p->vma, 0, sizeof(p->vma));
    return p;
}
```

(4). 根据提示2、3、4，参考 `lazy` 实验中的分配方法（将当前 `p->sz` 作为分配的虚拟起始地址，但不实际分配物理页面），此函数写在 `sysfile.c` 中就可以使用静态函数 `argfd` 同时解析文件描述符和 `struct file`

```

uint64
sys_mmap(void) {
    uint64 addr;
    int length;
    int prot;
    int flags;
    int vfd;
    struct file* vfile;
    int offset;
    uint64 err = 0xffffffffffffffffffff;

    // 获取系统调用参数
    if(argaddr(0, &addr) < 0 || argint(1, &length) < 0 || argint(2, &prot) < 0 ||
       argint(3, &flags) < 0 || argfd(4, &vfd, &vfile) < 0 || argint(5, &offset) < 0)
        return err;

    // 实验提示中假定addr和offset为0, 简化程序可能发生的情况
    if(addr != 0 || offset != 0 || length < 0)
        return err;

    // 文件不可写则不允许拥有PROT_WRITE权限时映射为MAP_SHARED
    if(vfile->writable == 0 && (prot & PROT_WRITE) != 0 && flags == MAP_SHARED)
        return err;

    struct proc* p = myproc();
    // 没有足够的虚拟地址空间
    if(p->sz + length > MAXVA)
        return err;

    // 遍历查找未使用的VMA结构体
    for(int i = 0; i < NVMA; ++i) {
        if(p->vma[i].used == 0) {
            p->vma[i].used = 1;
            p->vma[i].addr = p->sz;
            p->vma[i].len = length;
            p->vma[i].flags = flags;
            p->vma[i].prot = prot;
            p->vma[i].vfile = vfile;
            p->vma[i].vfd = vfd;
            p->vma[i].offset = offset;

            // 增加文件的引用计数
            filedup(vfile);

            p->sz += length;
            return p->vma[i].addr;
        }
    }

    return err;
}

```

(5). 根据提示5, 此时访问对应的页面就会产生页面错误, 需要在 `usertrap` 中进行处理, 主要完成三项工作: 分配物理页面, 读取文件内容, 添加映射关系

```

void
usertrap(void)
{
    ...
    if(cause == 8) {
        ...
    } else if((which_dev = devintr()) != 0){
        // ok
    } else if(cause == 13 || cause == 15) {
#define LAB_MMAP
        // 读取产生页面故障的虚拟地址，并判断是否位于有效区间
        uint64 fault_va = r_stval();
        if(PGROUNDDOWN(p->trapframe->sp) - 1 < fault_va && fault_va < p->sz) {
            if(mmap_handler(r_stval(), cause) != 0) p->killed = 1;
        } else
            p->killed = 1;
#endif
    } else {
        ...
    }
    ...
}

/**
 * @brief mmap_handler 处理mmap惰性分配导致的页面错误
 * @param va 页面故障虚拟地址
 * @param cause 页面故障原因
 * @return 0成功, -1失败
 */
int mmap_handler(int va, int cause) {
    int i;
    struct proc* p = myproc();
    // 根据地址查找属于哪一个VMA
    for(i = 0; i < NVMA; ++i) {
        if(p->vma[i].used && p->vma[i].addr <= va && va <= p->vma[i].addr + p->vma[i].len
           break;
    }
    if(i == NVMA)
        return -1;

    int pte_flags = PTE_U;
    if(p->vma[i].prot & PROT_READ) pte_flags |= PTE_R;
    if(p->vma[i].prot & PROT_WRITE) pte_flags |= PTE_W;
    if(p->vma[i].prot & PROT_EXEC) pte_flags |= PTE_X;

    struct file* vf = p->vma[i].vfile;
    // 读导致的页面错误
    if(cause == 13 && vf->readable == 0) return -1;
    // 写导致的页面错误
    if(cause == 15 && vf->writable == 0) return -1;

    void* pa = kalloc();
    if(pa == 0)
        return -1;
    memset(pa, 0, PGSIZE);

    // 读取文件内容
    ilock(vf->ip);
    // 计算当前页面读取文件的偏移量，实验中p->vma[i].offset总是0
    // 要按顺序读读取，例如内存页面A,B和文件块a,b
    // 则A读取a, B读取b, 而不能A读取b, B读取a
    int offset = p->vma[i].offset + PGROUNDDOWN(va - p->vma[i].addr);
}

```

```
int readbytes = readi(vf->ip, 0, (uint64)pa, offset, PGSIZE);
// 什么都没有读到
if(readbytes == 0) {
    iunlock(vf->ip);
    kfree(pa);
    return -1;
}
iunlock(vf->ip);

// 添加页面映射
if(mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)pa, pte_flags) != 0) {
    kfree(pa);
    return -1;
}

return 0;
}
```

(6). 根据提示6实现 `munmap`，且提示7中说明无需查看脏位就可写回

```

uint64
sys_munmap(void) {
    uint64 addr;
    int length;
    if(argaddr(0, &addr) < 0 || argint(1, &length) < 0)
        return -1;

    int i;
    struct proc* p = myproc();
    for(i = 0; i < NVMA; ++i) {
        if(p->vma[i].used && p->vma[i].len >= length) {
            // 根据提示, munmap的地址范围只能是
            // 1. 起始位置
            if(p->vma[i].addr == addr) {
                p->vma[i].addr += length;
                p->vma[i].len -= length;
                break;
            }
            // 2. 结束位置
            if(addr + length == p->vma[i].addr + p->vma[i].len) {
                p->vma[i].len -= length;
                break;
            }
        }
    }
    if(i == NVMA)
        return -1;

    // 将MAP_SHARED页面写回文件系统
    if(p->vma[i].flags == MAP_SHARED && (p->vma[i].prot & PROT_WRITE) != 0) {
        filewrite(p->vma[i].vfile, addr, length);
    }

    // 判断此页面是否存在映射
    uvmunmap(p->pagetable, addr, length / PGSIZE, 1);

    // 当前VMA中全部映射都被取消
    if(p->vma[i].len == 0) {
        fileclose(p->vma[i].vfile);
        p->vma[i].used = 0;
    }

    return 0;
}

```

(7). 回忆lazy实验中, 如果对惰性分配的页面调用了 `uvmunmap`, 或者子进程在`fork`中调用 `uvmcopy` 复制了父进程惰性分配的页面都会导致panic, 因此需要修改 `uvmunmap` 和 `uvmcopy` 检查 `PTE_V` 后不再 panic

```

if((*pte & PTE_V) == 0)
    continue;

```

(8). 根据提示8修改 `exit`, 将进程的已映射区域取消映射

```

void
exit(int status)
{
    // Close all open files.
    for(int fd = 0; fd < NOFILE; fd++){
        ...
    }

    // 将进程的已映射区域取消映射
    for(int i = 0; i < NVMA; ++i) {
        if(p->vma[i].used) {
            if(p->vma[i].flags == MAP_SHARED && (p->vma[i].prot & PROT_WRITE) != 0) {
                filewrite(p->vma[i].vfile, p->vma[i].addr, p->vma[i].len);
            }
            fileclose(p->vma[i].vfile);
            uvmunmap(p->pagetable, p->vma[i].addr, p->vma[i].len / PGSIZE, 1);
            p->vma[i].used = 0;
        }
    }

    begin_op();
    iput(p->cwd);
    end_op();
    ...
}

```

(9). 根据提示9，修改 fork， 复制父进程的VMA并增加文件引用计数

```

int
fork(void)
{
    // increment reference counts on open file descriptors.
    for(i = 0; i < NOFILE; i++)
        ...

    // 复制父进程的VMA
    for(i = 0; i < NVMA; ++i) {
        if(p->vma[i].used) {
            memmove(&np->vma[i], &p->vma[i], sizeof(p->vma[i]));
            filedup(p->vma[i].vfile);
        }
    }

    safestrcpy(np->name, p->name, sizeof(p->name));
    ...
}

```

copyright by duguosheng all right reserved, powered by Gitbook该文件修订时间： 2021-08-28 23:31:52