
刑 天

----基于 xv6 的操作系统

参赛队伍：刑 天

2024 全国大学生计算机系统能力大赛

华东区域赛操作系统设计赛--OS 原理

内核实现

2024 年 12 月

目录

1. 设计目标及要求.....	4
1.1. 设计目标.....	4
1.2. 设计要求.....	4
2. 开发环境.....	4
3. 需求分析与总体设计.....	4
3.1. 需求分析.....	4
3.1.1. 引导加载需求.....	4
3.1.2. 键盘输入支持.....	5
3.1.3. 存储管理.....	5
3.1.4. 进程管理.....	5
3.1.5. 文件系统.....	5
3.1.6. 系统调用接口.....	5
3.1.7. 内核功能增强与修改.....	5
3.2. 总体设计.....	5
3.3. boot 引导模块.....	6
3.4. 内存管理模块.....	7
3.5. 文件系统模块.....	8
4. 内存管理.....	8
4.1. kmem	9
4.2. 内存的初始化.....	9
4.3. 分配内存.....	11
5. 文件系统.....	11
5.1. 系统结构与数据结构.....	12
5.1.1. 文件系统结构.....	12
5.1.2. 目录项.....	12
5.1.3. 目录项缓存.....	13
5.2. 关键程序实现.....	13
5.2.1. 路径查找 (env_lookup_path)	13
5.2.2. ealloc - 分配文件/目录条目.....	14
5.2.3. ewrite 和 eread - 文件读写	15
5.2.4. isdirempty - 判断目录是否为空	15
5.2.5. emount 和 eumount - 文件系统挂载与卸载	15
5.3. 算法与系统实现思路.....	16
6. 进程管理.....	16
6.1. 相关原理.....	16
6.1.1. 进程.....	16
6.1.2. 进程 API	17
6.1.3. 进程创建.....	17
6.1.4. 进程状态.....	18
6.1.5. 数据结构.....	18
6.2. 相关函数实现.....	19
6.2.1. 进程控制块 PCB	19

6.2.2. 进程初始化.....	21
6.2.3. 进程分配.....	22
6.2.4. 进程的上下文切换.....	24
6.2.5. init 进程	28
7. 系统调用.....	29
7.1. 系统调用内核的原理.....	29
7.1.1. 在 RISC-V 中请求系统调用	29
7.1.2. 系统调用的处理过程.....	30
7.2. 相关系统调用的实现.....	31
7.2.1. 进程管理系统调用	31
7.2.2. 内存管理系统调用	37
7.2.3. 文件系统系统调用	41
7.2.4. 其它系统调用.....	46
8. 运行界面示例.....	49
8.1. 单个命令测试.....	49
8.2. 多命令同时测试.....	51
9. 总结体会及未来设计方向.....	54
9.1. 总结体会.....	54
9.2. 未来设计方向.....	55
参考文献:	57

1. 设计目标及要求

1.1. 设计目标

深入了解操作系统的工作原理和实现方法，提高编程能力和系统思维，设计基于 xv6 的操作系统内核，能够实现一些常见的系统调用。

1.2. 设计要求

- 调试操作系统启动的引导程序,使得操作系统多核启动能够顺利执行。
- 调试 SBI 输入输出函数,使得操作系统能够正常进行键盘输入。
- 实现存储管理、进程管理、文件系统等操作系统内核的基本管理功能。
- 实现一些系统调用的调试。
- 修改多个内核相关部分，在原有的功能上增强功能的实现。

2. 开发环境

使用系统: ubuntu-20.04.6-desktop-amd64

装载系统软件: VMware Workstation Pro

项目环境依赖: qemu-system-riscv64、riscv-gnu-toolchain

编译配置: riscv64-unknown-elf-gcc

调试配置: riscv64-unknown-elf-gdb

编辑软件: vscode 且能进行远程连接、调试

我们的内核代码是在合肥工业大学 [oskernel2024-ywd](<https://gitlab.eduxiji.net/T202410359992654/oskernel2024-ywd/-/tree/main>) 和 [能 run 就行-2958](<https://gitlab.eduxiji.net/educg-group-14238-914330/853476998-2958>)团队的内核代码的基础上实现的，团队在配置开发环境时主要参考了 [合肥工业大学-赤道会师团队的环境配置手册](<https://gitlab.eduxiji.net/educg-group-22024-2210153/T202410359992686-915/-/blob/main/docs/documents/%E7%8E%AF%E5%A2%83%E9%85%8D%E7%BD%AE%E6%89%8B%E5%86%8C.pdf>), 这份环境配置写的很详细，在环境配置的过程中，我们也遇到了一些手册中没有出现得到问题，我们将一些遇到的问题及相应的解决办法写在了我们对的另一份“开发中遇到的问题及解决”文档中。

3. 需求分析与总体设计

3.1. 需求分析

要目标是设计和实现一个基于 xv6 的操作系统内核，深入理解操作系统的基本原理及其实现过程。为了完成该项目，系统需要实现以下关键功能，并满足相应的需求。

3.1.1. 引导加载需求

本项目需要实现操作系统的引导加载，即将操作系统的内核从磁盘加载到内

存中,并跳转到保护模式,初始化全局描述符表和中断描述符表,以及内核分页。

3.1.2. 键盘输入支持

在操作系统中,键盘输入是用户与系统交互的基本方式之一。系统需要实现基本的输入输出功能,要求如下:

实现简单的 SBI (Serial Basic Input) 接口,确保操作系统能够正确地接收来自键盘的输入。

在调试过程中,系统应能够正常响应键盘的按键事件,并将其传递给操作系统内核进行处理。

3.1.3. 存储管理

存储管理是操作系统内核中至关重要的一部分,涉及内存的分配与回收。该功能的需求分析如下:

实现内存的分配和管理,确保每个进程都能获得合适的内存空间。

支持页表管理和虚拟内存机制,实现虚拟内存的映射和地址转换。

在内存管理中,考虑如何处理内存碎片,确保系统的内存利用率高效。

3.1.4. 进程管理

支持创建和销毁进程的系统调用,实现进程的生命周期管理。

实现进程调度机制,确保操作系统能够公平、高效地调度多个进程。

提供基本的进程同步与互斥机制,以确保进程间的资源共享与安全。

3.1.5. 文件系统

实现一个简单的文件系统,能够支持文件的创建、删除、读写操作。

提供基本的文件目录结构,支持目录的创建、删除及文件的查找。

实现系统调用接口,使得用户能够通过命令操作文件系统。

3.1.6. 系统调用接口

实现基本的系统调用,包括进程管理、文件操作、内存管理等功能。

系统调用接口应简单易用,能够提供高效的执行机制。

3.1.7. 内核功能增强与修改

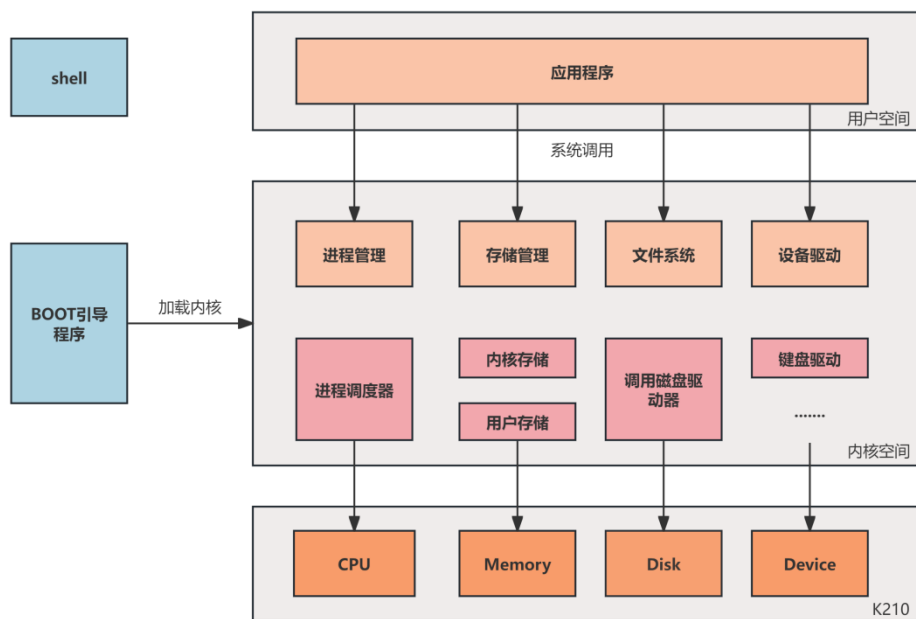
在完成上述基本功能的基础上,还需要对操作系统内核进行功能增强和修改。

在现有的 xv6 内核功能基础上,针对项目要求进行必要的修改与增强。

调试和优化内核代码,解决可能出现的性能瓶颈和系统稳定性问题。

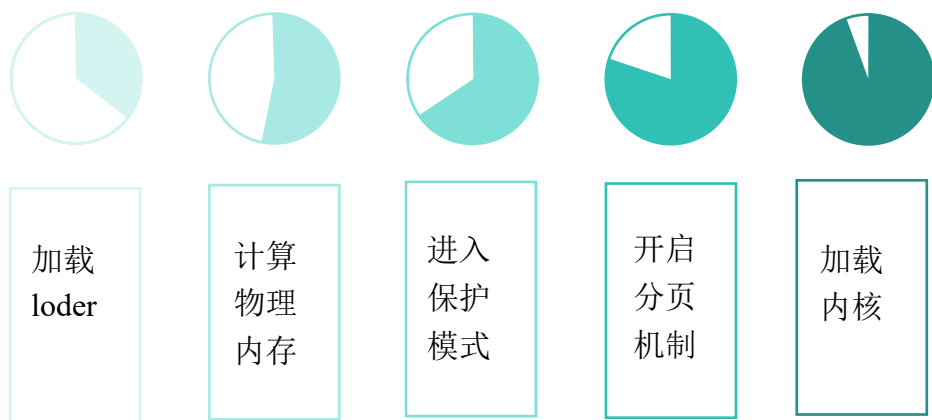
3.2. 总体设计

总体设计框架:

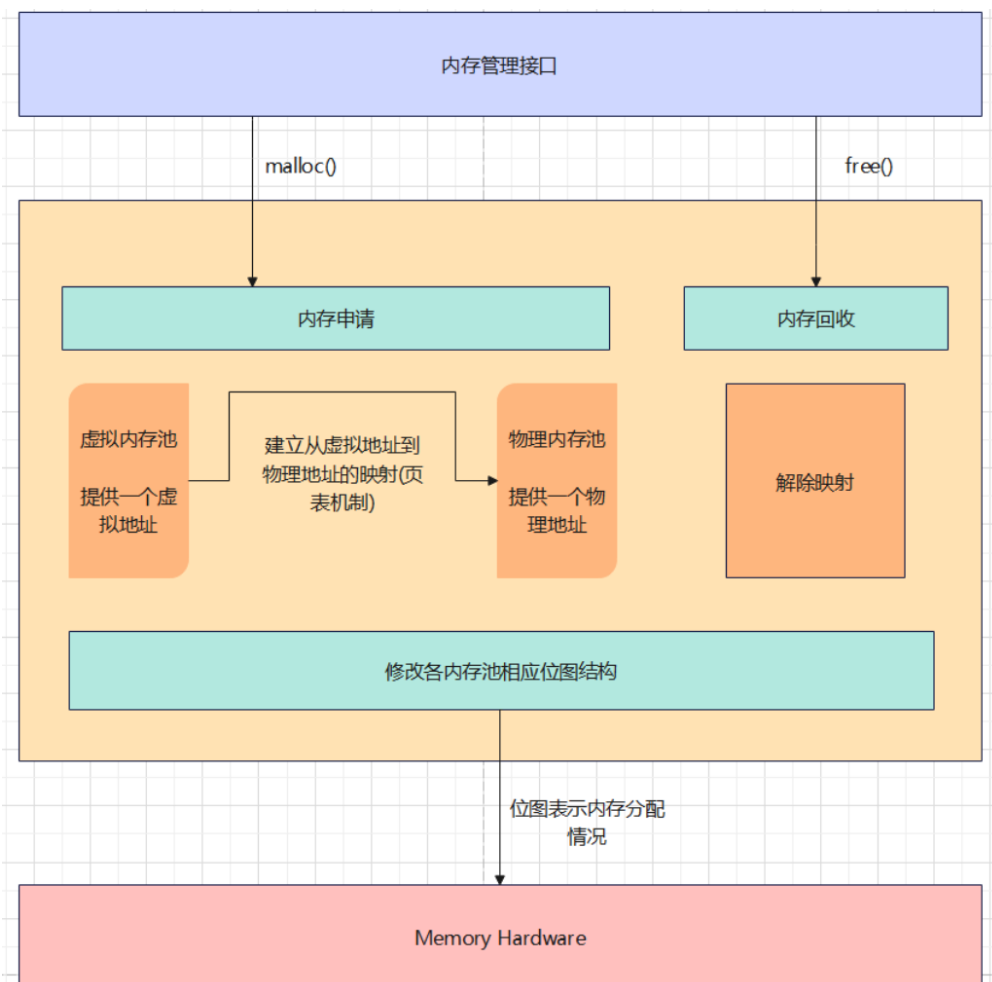


系统整体功能大致可以分为 6 个部分，负责启动加载内核的 boot 模块，负责提供用户交互的 Shell 模块，以及进程管理、内存管理、文件系统和设备驱动 4 个子系统模块。

3.3. boot 引导模块

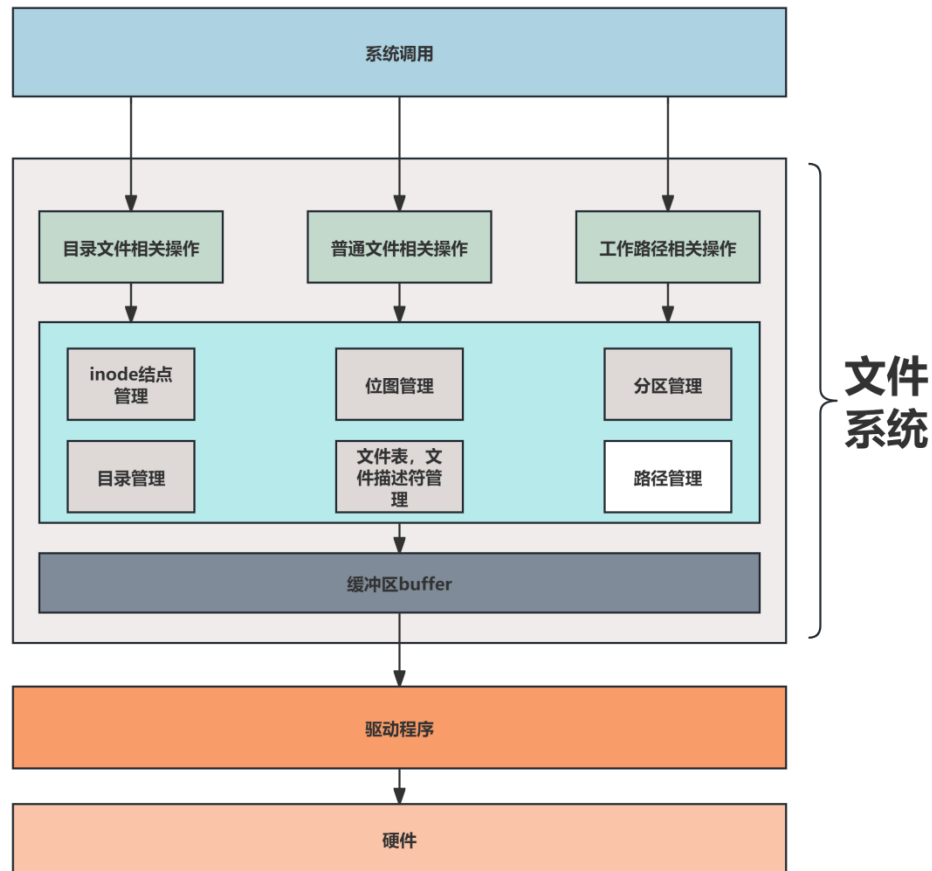


3.4. 内存管理模块



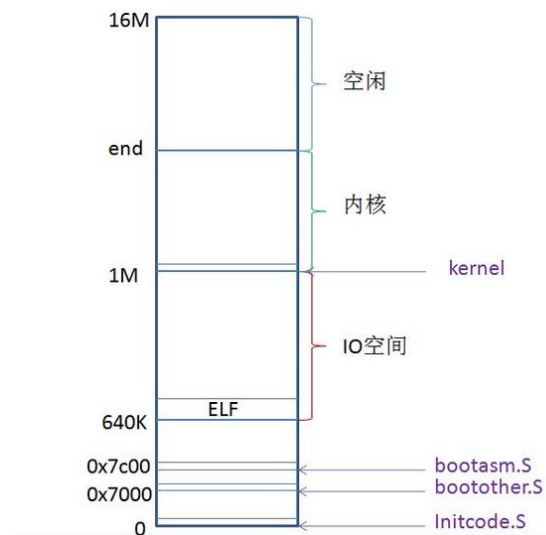
内存管理的核心是维护内存池，利用位图结构来实现物理内存池、虚拟内存池对内存进行管理。其中物理内存池以及虚拟内存池中又分别可以划分为用户内存池和内核内存池，总共 4 种内存池。内存管理的本质其实是地址的提供以及映射的建立和删除。内存池提供物理地址和虚拟地址，并在中间使用二级页表机制建立或删除映射，实现内存的分配。

3.5. 文件系统模块



4. 内存管理

内存管理代码主要设计在 kernel 文件夹下的 kalloc.c 文件中。
系统中的内存布局如下：



4.1. kmem

kmem 结构体:

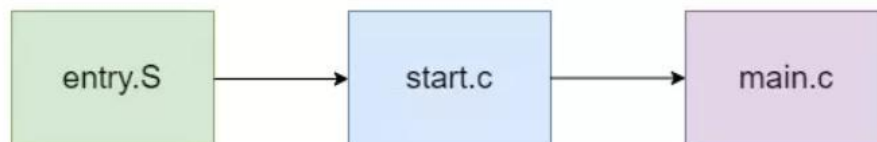
```
struct run {
    struct run *next;
};

// 这是一个临界资源，修改时候要获取锁
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem;
```

kmem 是一个临界资源，其中定义了一个锁，还有一个空闲页的链表，在这个空闲页的链表中存储着当前所有的空闲页。

4.2. 内存的初始化

启动的流程为 entry.S --> start.c --> main.c。在 main.c 中会初始化各个部件，比如内存初始化、页表初始化等等。



初始化内存的函数是 kinit(), 这里的锁就不特别说明了，因为 kmem 结构是个临界资源，所以修改的时候要获取锁，防止在当前进程修改的过程中被其他进程修改了。

```
void
kinit()
{
    // 初始化锁
    initlock(&kmem.lock, "kmem");
    /* 将第一个可用的内存到最后一个可用的内存分成一页一页的
       * 并将这些页添加到空闲页链表中
       */
    freerange(end, (void*)PHYSTOP);
}
```

kinit()中调用了 freerange(), 该函数传入两个指针，从 pa_start 到 pa_end 的物理地址分成一页一页的，然后加入到空闲链表中。

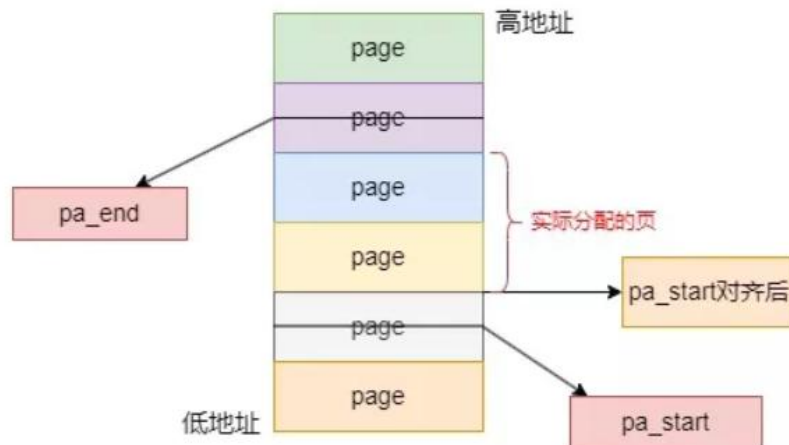
```
void
freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start); // p 是比 pa_start 地址高，且最近的一个页
```

```
// 从 pa_start 内存对齐后的页，到 pa_end 的每个页都释放掉，并加入到空闲页链表中
```

```
for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
```

```
}
```

freerange()中用到了一个 PGROUNDUP，这个的作用是页对齐，也就是获取大于等于 pa_start 的下一个页面对齐的地址，因为 pa_start 到 pa_end 不一定能分成整数个的页，所以找到下一个是 PGSIZE 整数倍的地址（假设将所有物理内存分成一页一页的，将 pa_start 放到 PGSIZE 整数倍的地址处，能保证将该物理区域分成整数个 page）。



freerange()用到了 kfree (void *pa)，它的功能是将 pa 页释放掉，并且将 pa 页面加入到空闲页链表中。

```
void
```

```
kfree(void *pa)
```

```
{
```

```
    struct run *r;
```

```
    // 如果该页不是 PGSIZE 的整数倍，或者小于第一个可用内存的地址，或者大于最大可用内存地址 -- 就陷入恐慌
```

```
    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;
```

```
    acquire(&kmem.lock);
```

```
    // 将 pa 页加入到空闲页链表中
```

```
    r->next = kmem.freelist;
```

```
kmem.freelist = r;
release(&kmem.lock);
}
```

kfree 就是先判断页是否有效，然后 fill with junk to catch dangling refs., 最后将 pa 页添加到空闲页链表中。

在 kinit() 中 freerange(void *pa_start, void *pa_end) 传入了两个参数，一个是 end()、一个是 ****(void*)PHYSTOP****。end() 表示是内核区域后第一个可用的地址，(void*)PHYSTOP 表示的是物理地址的结束地址。所以 kinit() 表示将当前可用的物理地址初始化为整数个页，并将这些页加入到空闲页链表中。

4.3. 分配内存

使用 kalloc(void) 函数分配一个 4096-byte 的物理页，然后返回指向该页起始地址的指针。

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *
kalloc(void)
{
    struct run *r;

    acquire(&kmem.lock);
    // 获取空闲页链表中的第一个空闲页面
    r = kmem.freelist;
    if(r)
        // 将空闲页链表中的第一个空闲页面丢弃
        kmem.freelist = r->next;
    release(&kmem.lock);

    if(r)
        memset((char*)r, 5, PGSIZE); // fill with junk

    // 返回一个指向该地址的指针
    return (void*)r;
}
```

可以看到，上锁就不用多说了（因为要修改临界资源），先获取空闲链表中的第一个空闲页面，如果存在这个页面，就将该页面从空闲页面中去掉，然后把该页面 fill with junk，最后返回指向该页起始地址的指针。

5. 文件系统

文件系统涉及的代码主要在 kernel 文件夹下的 fat32.c 中。

我们主要实现的是基于 FAT32 的文件系统，主要功能包括文件和目录管理、路径查找、簇的分配与释放、文件的读取与写入、目录的条目管理等。此文件系统主要涉及的操作包括文件条目的分配（`ealloc`）、文件读取（`eread`）、文件写入（`ewrite`）、目录操作、路径解析、文件状态查询等。

5.1. 系统结构与数据结构

5.1.1. 文件系统结构

文件系统通过结构体 `fs` 来表示，每个文件系统实例包含以下信息：

- 设备号 (`devno`): 每个文件系统实例的唯一标识。
- 有效标志 (`valid`): 文件系统是否已初始化。
- 磁盘读写函数(`disk_init`, `disk_read`, `disk_write`): 用于操作磁盘的函数指针。
- 根目录 (`root`): 根目录的 `dirent` 结构，表示文件系统的根目录。
- 文件分配表 (`fat`): 存储关于 FAT（文件分配表）的信息，包括 BPB（启动参数块）等。
- 缓存管理 (`ecache`): 用于管理目录项缓存。

```
struct fs {  
    int valid;                // 文件系统是否已初始化（有效）  
    int devno;               // 文件的设备号，每个文件的唯一标识  
  
    struct dirent root;      // 根目录的 dirent 结构  
    struct fat32 fat;        // 存储 FAT 表和启动参数块 (BPB) 等  
    struct entry_cache ecache; // 目录项缓存，用于提高查找效率  
    void *image;             // 文件系统映像  
    void (*disk_init)(void); // 磁盘初始化函数  
    void (*disk_read)(struct buf *b); // 磁盘读取函数  
    void (*disk_write)(struct buf *b); // 磁盘写入函数  
};
```

5.1.2. 目录项

`dirent` 结构用于表示文件系统中的文件或目录的条目，每个目录项包含如下信息：

- 文件名 (`filename`): 文件或目录的名称。
- 属性 (`attribute`): 文件或目录的属性，如是否为目录、是否为只读文件等。
- 簇号 (`first_clus`, `cur_clus`): 文件数据所在簇的位置。
- 文件大小 (`file_size`): 文件的大小。
- 引用计数 (`ref`): 用于管理文件或目录的引用计数。

```
struct dirent {  
    char filename[FAT32_MAX_FILENAME + 1]; // 文件或目录的名称  
    uint32 first_clus;                       // 文件数据的起始簇号  
    uint32 cur_clus;                         // 当前访问的簇号  
    uint32 file_size;                       // 文件大小  
    uint16 attribute;                      // 文件属性（如只读、目录等）  
    struct dirent *prev, *next;             // 用于在目录项缓存中连接条目的链表指针  
    int ref;                               // 引用计数，用于文件或目录的引用管理  
};
```

```

int valid;           // 标记该目录项是否有效
int dirty;           // 标记该目录项是否被修改
struct dirent *parent; // 父目录
int off;             // 在父目录中的偏移量
int clus_cnt;        // 簇计数（文件跨越的簇的数量）
};

```

5.1.3. 目录项缓存

`ecache` 是文件系统中用于存储目录项缓存的结构，它包含一个目录项缓存数组。该缓存用于提高对目录项的访问速度，减少频繁的磁盘读写操作。`ecache` 中存储的目录项通过 `prev` 和 `next` 链接在一起，形成一个双向链表。

`ecache` 缓存有一个最大容量（`ENTRY_CACHE_NUM`），当缓存满时，旧的条目会被替换出去。

为了提高目录项的访问效率，每个文件系统实例通过 `ecache` 维护一个目录项缓存。文件系统会尽量将最近访问的目录项保留在缓存中，当缓存满时，按照某种策略（如 LRU）淘汰旧的目录项。

在进行修改操作时，文件系统会使用锁（如 `elock` 和 `eunlock`）来确保对目录项的操作是线程安全的。只有当文件或目录的引用计数为 1 且没有其他线程在使用时，才能修改目录项。

5.2. 关键程序实现

5.2.1. 路径查找（`env_lookup_path`）

该函数用于查找给定路径上的文件或目录。路径是由多个目录名和文件名组成的，函数会逐步解析路径，直到找到目标文件或目录。

实现过程：

通过 `skipelem` 函数解析路径中的每个元素（文件夹或文件）。

遇到目录时，通过 `dirlookup` 函数查找目录中的对应条目。

如果目录项是目录，继续递归查找；如果是文件，则返回文件条目。

```

static struct dirent *env_lookup_path(struct dirent* env, char *path, int parent, char *name) {
    struct dirent *entry, *next;
    entry = edup(env); // 获取目录的副本
    while ((path = skipelem(path, name)) != 0) { // 遍历路径元素
        elock(entry); // 加锁
        if (!(entry->attribute & ATTR_DIRECTORY)) { // 如果不是目录，返回 NULL
            return NULL;
        }
        eunlock(entry);
        eput(entry);
        return NULL;
    }
    if (parent && *path == '\0') { // 如果是父目录且路径已解析完
        eunlock(entry);
        return entry;
    }
    // 查找子目录或文件
}

```

```

        if ((next = dirlookup(entry, name, 0)) == 0) {
            eunlock(entry);
            eput(entry);
            return NULL;
        }
        eunlock(entry);
        eput(entry);
        entry = next;
    }
    return entry;
}

```

5.2.2. ealloc - 分配文件/目录条目

ealloc 函数用于在目录中分配一个新的文件或目录条目。它首先检查条目是否已存在，如果已存在则返回现有条目。否则，分配一个新的条目并初始化。

实现过程：

检查父目录 dp 是否有效且是一个目录。

检查文件名是否合法。

如果条目已存在，则返回已有的目录项。

如果条目不存在，创建新条目并初始化相关字段（如 first_clus、file_size）。

对于目录条目，还需要创建特殊的“.”和“..”目录项。

```

struct dirent *ealloc(struct dirent *dp, char *name, int attr) {
    // 省略部分代码
    if ((ep = dirlookup(dp, name, &off)) != 0) { // 如果条目已存在
        return ep;
    }
    ep = eget(dp, name);
    elock(ep); // 获取锁
    ep->attribute = attr;
    ep->file_size = 0;
    ep->first_clus = 0;
    ep->parent = edup(dp); // 设置父目录
    ep->off = off;
    ep->clus_cnt = 0;
    ep->cur_clus = 0;
    ep->dirty = 0;
    // 初始化目录条目
    if (attr == ATTR_DIRECTORY) {
        ep->cur_clus = ep->first_clus = alloc_clus(self_fs, dp->dev);
        emake(ep, ep, 0);
        emake(ep, dp, 32);
    } else {
        ep->attribute |= ATTR_ARCHIVE;
    }
    emake(dp, ep, off);
}

```

```

    ep->valid = 1;
    eunlock(ep);
    return ep;
}

```

5.2.3. ewrite 和 eread - 文件读写

这两个函数分别实现文件的读写操作。它们通过 `rw_clus` 函数对簇进行读写。

对于读操作 (`eread`): `eread` 函数会查找文件的每个簇, 并逐簇读取数据, 直到读取完指定的字节数。

写操作 (`ewrite`): `ewrite` 函数与读操作类似, 它通过 `rw_clus` 函数将数据写入文件的各个簇中。

```

int eread(struct dirent *entry, int user_dst, uint64 dst, uint off, uint n) {
    uint tot, m;
    for (tot = 0; entry->cur_clus < FAT32_EOC && tot < n; tot += m, off += m, dst += m) {
        reloc_clus(self_fs, entry, off, 0);
        m = self_fs->fat.byts_per_clus - off % self_fs->fat.byts_per_clus;
        if (n - tot < m) {
            m = n - tot;
        }
        if (rw_clus(self_fs, entry->cur_clus, 0, user_dst, dst, off % self_fs->fat.byts_per_clus, m) != m) {
            break;
        }
    }
    return tot;
}

```

5.2.4. isdirempty - 判断目录是否为空

函数检查一个目录是否为空, 仅包含 “.” 和 “..” 两个目录项。

通过 `enext` 函数跳过 “.” 和 “..”, 然后检查目录是否还有其他条目。

```

int isdirempty(struct dirent *dp) {
    struct dirent ep;
    int count;
    int ret;
    ep.valid = 0;
    ret = enext(dp, &ep, 2 * 32, &count); // 跳过 “.” 和 “..”
    return ret == -1;
}

```

5.2.5. emount 和 eumount - 文件系统挂载与卸载

`emount` 用于挂载文件系统, `eumount` 用于卸载文件系统。

挂载 (`emount`): 通过 `ename` 函数查找挂载点, 设置其 `mnt` 标志为 1, 并将文件系统的根目录设置为挂载点的父目录。

卸载 (`eumount`): 通过 `ename` 查找挂载点, 重置其挂载标志, 释放文件系

统的相关资源。

```
int emount(struct fs* fatfs, char* mnt) {
    struct dirent* mntpoint = ename(mnt);
    if (mntpoint == NULL) return -1;
    mntpoint = edup(mntpoint);
    mntpoint->mnt = 1;
    mntpoint->dev = fatfs->devno;
    fatfs->root.parent = mntpoint;
    return 0;
}

int eumount(char* mnt) {
    struct dirent* mntpoint = ename(mnt);
    if (mntpoint == NULL) return -1;
    if (mntpoint->mnt) mntpoint->mnt = 0;
    if (FatFs[mntpoint->dev].image) eput(FatFs[mntpoint->dev].image);
    if (mntpoint->parent) mntpoint->dev = mntpoint->parent->dev;
    eput(mntpoint);
    return 0;
}
```

5.3. 算法与系统实现思路

簇的分配与释放：通过 `alloc_clus` 和 `free_clus` 函数管理 FAT32 文件系统中的簇。簇是文件存储的最小单位，文件的内容通过簇链接在一起。文件系统通过 FAT 表来跟踪每个簇的状态。

目录项的查找与分配：通过 `dirlookup` 函数查找目录中的文件或子目录。如果目录项不存在，则使用 `ealloc` 分配一个新的目录项。对于长文件名，使用多个 `long_name_entry` 来存储。

文件操作的同步：所有对目录项的操作都使用锁（`elock` 和 `eunlock`）来确保线程安全，避免并发访问时出现竞争条件。

挂载与卸载：文件系统的挂载与卸载由 `emount` 和 `eumount` 实现，通过设置文件系统的根目录的父目录来实现挂载操作。

6. 进程管理

进程管理模块主要设计在 `kernel` 文件夹下的 `proc.c` 文件中。

6.1. 相关原理

6.1.1. 进程

操作系统为了管理各种各样程序的运行，提出了进程的抽象，每个进程都对应一个程序，有了进程的抽象，仿佛应用程序占用了整个 CPU，而不需要考虑何时需要将 CPU 让给其他程序；进程的管理、CPU 的资源分配是交给操作系统来处理的。

6.1.2. 进程 API

现代操作系统都以某种形式提供下面几种 API:

创建: 操作系统必须包含一些创建新进程的方法。在 shell 中键入命令或双击应用程序图标时, 会调用操作系统来创建新进程, 运行指定的程序。

销毁: 由于存在创建进程的接口, 因此系统还提供了一个强制销毁进程的接口。当然, 很多进程会在运行完成后自行退出。但是, 如果它们不退出, 用户可能希望终止它们, 因此停止失控进程的接口非常有用。

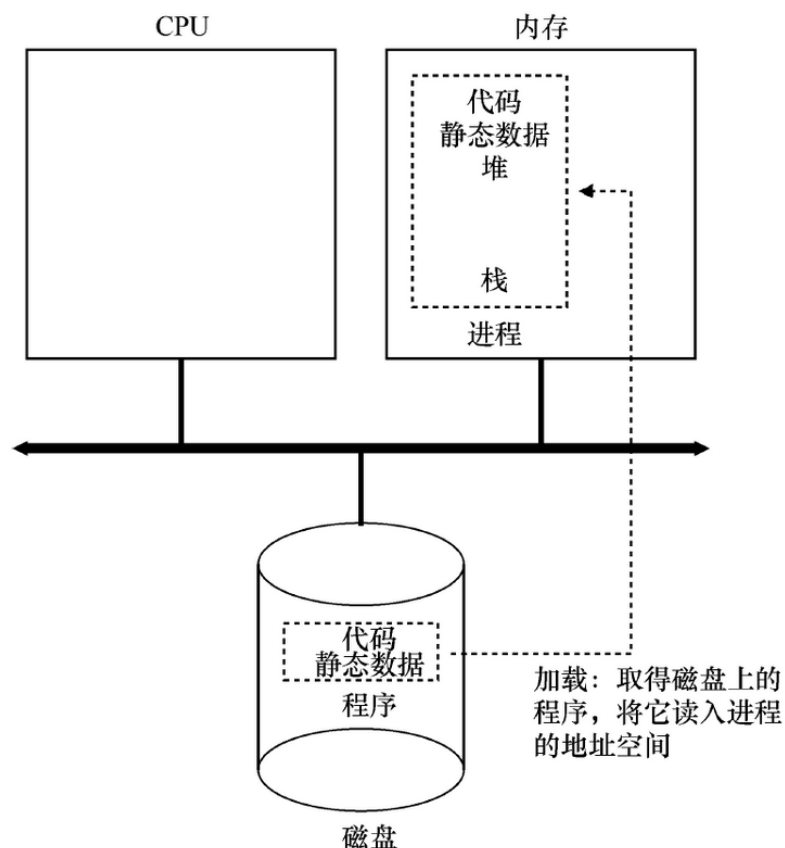
等待: 有时等待进程停止运行是有用的, 因此经常提供某种等待接口。

其他控制: 除了杀死或等待进程外, 有时还可能有其他控制。例如, 大多数操作系统提供某种方法来暂停进程(停止运行一段时间), 然后恢复(继续运行)。

状态: 通常也有一些接口可以获得有关进程的状态信息, 例如运行了多长时间, 或者处于什么状态

6.1.3. 进程创建

操作系统运行程序必须做的第一件事是将代码和所有静态数据(例如初始化变量)加载到内存中, 加载到进程的地址空间中。程序最初以某种可执行格式驻留在磁盘上(或 SSD 上)。因此, 将程序和静态数据加载到内存中的过程, 需要操作系统从磁盘读取这些字节, 并将它们放在内存中的某处。



操作系统运行程序需要进程以下几个步骤:

- 1、将代码和静态数据加载的内存中;
- 2、为程序的运行时栈分配一些内容, C 语言中栈用来存放局部变量、函数参数和返回地址;

3、为程序的堆分配一些空间，C 语言中堆用于显示请求动态分配数据，一般通过 `malloc()` 调用；

4、执行输入输出相关的任务，默认情况下每个进程都打开 3 个文件描述符，用于标准输入，输出，错误；

5、启动程序，在入口处运行，即 `main()` 函数。通过跳转到 `main()` 例程，OS 将 CPU 的控制权转移到新创建的进程中，从而程序开始执行。

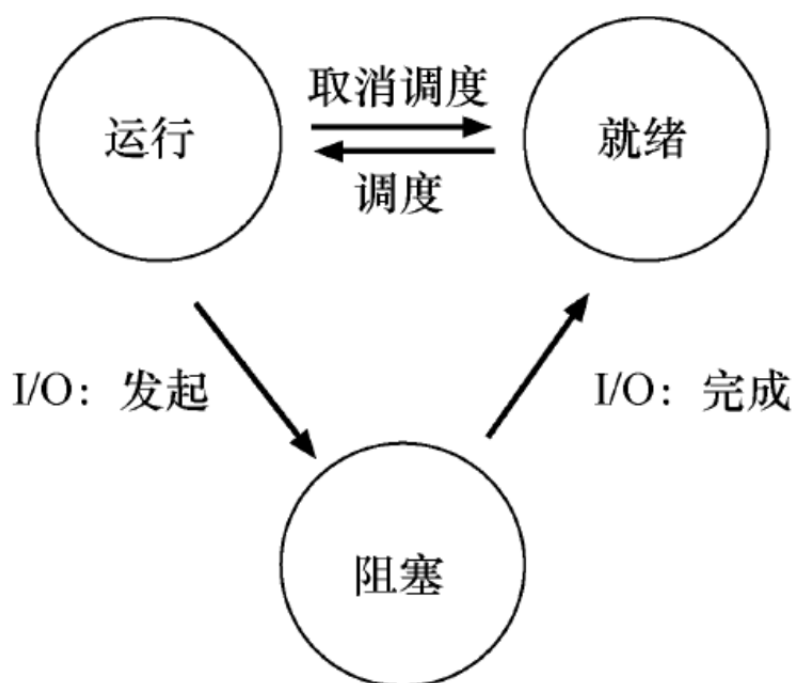
6.1.4. 进程状态

运行： 该状态下，进程正在处理器中运行；

就绪： 就绪态下，进程已准备好运行，但由于某种情况，操作系统此时不运行该进程；

阻塞/等待： 阻塞状态下，进程执行了某种操作，直到发生其他事件时才会准备运行。 例如：当进程向磁盘发起 I/O 请求时， 它会被阻塞，因此其他进程可以使用处理器。

除了上述三个状态，一般还有 创建 状态，和 终止 状态。

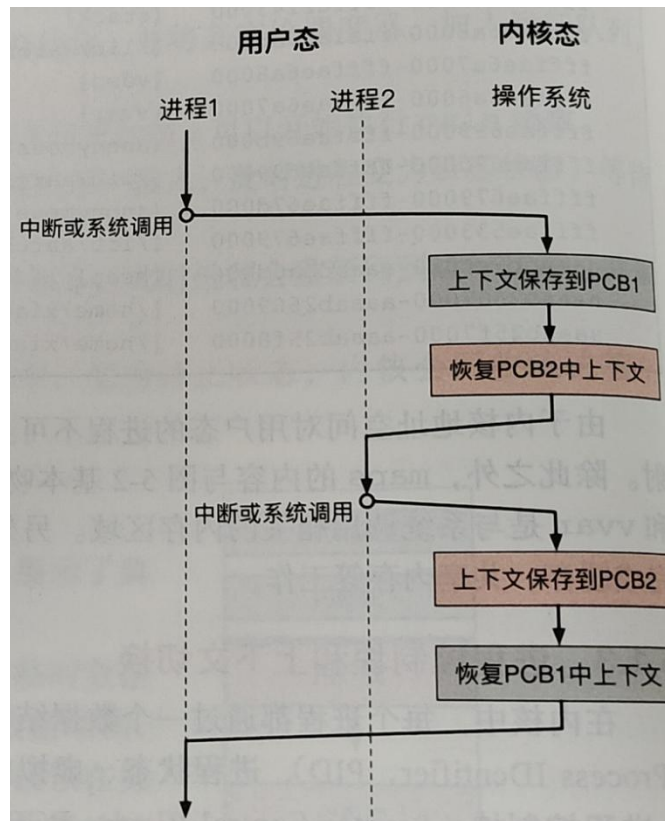


6.1.5. 数据结构

每个进程都有一个数据结构来保存它的相关状态，例如进程标识符 (PID)、进程状态、虚拟内存地址、打开的文件描述符等。这个数据结构称为进程控制块 (PCB)。

操作系统可以通过 PCB 来跟踪进程的相关信息，比如跟踪被阻塞的进程，或当 IO 事件完成时，操作系统能够正确的唤醒进程，使其继续运行。

当操作系统需要切换当前运行的进程时，会利用上下文切换机制，该机制会将一个进程的寄存器保存到 PCB 中，然后将下一个进程先前保存的状态写入寄存器中，从而切换到这个进程运行。



6.2. 相关函数实现

6.2.1. 进程控制块 PCB

进程控制块定义：

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

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

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

    // these are private to the process, so p->lock need not be held.
    uint64 kstack;                  // Virtual address of kernel stack
    uint64 sz;                      // Size of process memory (bytes)
    pagetable_t pagetable;          // User page table
    struct trapframe *trapframe;    // data page for trampoline.S
    struct context context;         // swtch() here to run process
};
```

```

struct file *ofile[NOFILE]; // Open files
struct inode *cwd;          // Current directory
char name[16];              // Process name (debugging)
};

```

包含了进程的 PID，进程状态，内核栈地址，进程页表等基本内容。

初始化的时候创建了一个进程池，当需要进程的时候会从池中找到一个未使用的进程，然后分配出去。进程池的大小为 64，也就是说系统中最多支持 64 个进程同时运行，超过这个数量就不支持了，相关定义如下：

```

// kernel/proc.c
struct proc proc[NPROC];

// kernel/param.h
#define NPROC      64 // maximum number of processes

```

进程的状态，定义在 kernel/proc.h 的 procstate 枚举类型中：

```

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

```

SLEEPING、RUNNABLE 和 RUNNING 分别表示睡眠、阻塞和正在运行三个状态，这个比较好理解；

UNUSED：在进程池中，尚未使用，还没有被分配出去；

USED：刚从进程池中分配出去，但是此时进程相关资源还没分配；

ZOMBIE：僵尸状态，当一个进程执行完成的时候，会调用 exit 退出，此时进程会进入这个状态，需要等待父进程调用 wait 来回收子进程所有剩下的资源；

当一个子进程执行 exit 退出的时候，但是父进程一直没有调用 wait 来回收子进程的资源，这个时候这个子进程就被称为僵尸进程；

如果子进程退出之前，父进程就已经终止了，此时子进程还在运行，这个进程就称为孤儿进程，孤儿进程会被 init 进程所接管，最后会由 init 进程调用 wait 来释放资源；

除了每个进程有一个控制块外，每个 CPU 也有一个状态，结构体如下，可以看到对于一个 CPU 来说，它保存了当前运行在 CPU 上的进程状态，以及 context 上下文（下面会提到），另外两个变量与中断相关（以后有机会在分析一下时钟中断和设备中断）；

```

// Per-CPU state.
struct cpu {
    struct proc *proc;          // The process running on this cpu, or null.
    struct context context;     // swtch() here to enter scheduler().
    int noff;                   // Depth of push_off() nesting.
    int intena;                 // Were interrupts enabled before push_off()?
};

```

CPU 和进程一样也有一个结构体数组，说明 xv6 最多可支持 8 核：

```

// kernel/proc.c
struct cpu cpus[NCPU];

```

```
// kernel/param.h
#define NCPU 8 // maximum number of CPUs
```

6.2.2. 进程初始化

进程的初始化, 准确来说应该是进程池初始化, 因为当进程池初始化完成后, 需要进程的时候从池子中取出, 释放进程的时候再放回进程池中(这里的取和放, 对应的其实就是将进程的状态设为 USED 和 UNUSED), 定义在 kernel/proc.c 中:

```
// initialize the proc table.
void
procinit(void)
{
    struct proc *p;
    // 初始化相关的锁
    initlock(&pid_lock, "nextpid");
    initlock(&wait_lock, "wait_lock");
    // 枚举进程池, 设置进程状态以及内核栈
    for(p = proc; p < &proc[NPROC]; p++) {
        initlock(&p->lock, "proc");
        p->state = UNUSED;
        p->kstack = KSTACK((int) (p - proc));
    }
}
```

这样需要注意一下 kstack 这个成员变量, 它的含义是内核栈, 它是在内核物理地址空间中最上面的一块区域, 每个栈下面还有一页 PageGuard, 为了防止栈溢出; 内核栈进程池初始化之前就映射到物理地址上了, 在 kernel/proc.c 中可以看到 proc_mapstacks 函数, 它是在内核页表初始化的时候被调用的:

```
void
proc_mapstacks(pagetable_t kpgtbl)
{
    struct proc *p;
    // 枚举进程池
    for(p = proc; p < &proc[NPROC]; p++) {
        // 分配一页物理地址作为内核栈
        char *pa = kalloc();
        if(pa == 0)
            panic("kalloc");
        // 计算得到进程内核栈的虚拟地址
        uint64 va = KSTACK((int) (p - proc));
        // 在内核页表中建立映射关系
        kvmmap(kpgtbl, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
    }
}
```

所以对于一个进程来说, 可以认为它有一个对应的用户线程, 以及一个对应的内核线程, 一个进程最多只能运行一个线程, 要不运行在用户模式下, 要不运

行在内核模式下。当发现 `trap` 的时候会从用户线程切换到内核线程。

6.2.3. 进程分配

当需要创建一个新的进程的时候，会调用 `allocproc` 函数从进程池中分配一个进程以供使用。

```
// Look in the process table for an UNUSED proc.
// If found, initialize state required to run in the kernel,
// and return with p->lock held.
// If there are no free procs, or a memory allocation fails, return 0.
static struct proc*
allocproc(void)
{
    struct proc *p;
    // 遍历进程池，找到未使用的进程
    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state == UNUSED) {
            goto found;
        } else {
            release(&p->lock);
        }
    }
    return 0;

found: // 找到一个未使用的进程
    p->pid = allocpid(); // 分配一个 PID
    p->state = USED;      // 设置进程状态，此时已经被分配出去了

    // Allocate a trapframe page.
    // 分配一页物理内存作为进程的 trap 上下文切换页面
    if((p->trapframe = (struct trapframe *)kalloc()) == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    }

    // An empty user page table.
    // 初始化进程页表，只对 trampoline 和 trapframe 页面做了映射
    // 并没有实际任何的物理内存大小，即此时的进程可以使用的空间大小为 0
    p->pagetable = proc_pagetable(p);
    if(p->pagetable == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    }
}
```

```

// Set up new context to start executing at forkret,
// which returns to user space.
// 设置进程切换上下文
memset(&p->context, 0, sizeof(p->context));
// forkret 作为进程切换的返回地址
p->context.ra = (uint64)forkret;
// 设置进程的内核栈指针（栈是向上生长的，栈大小为 PGSIZE）
p->context.sp = p->kstack + PGSIZE;

return p;
}

```

allocproc 函数找到一个未使用的进程，然后设置进程的 PID、状态、进程页表等内容。

allocpid() 函数分配一个 PID 号，相关定义如下，可以用一个全局变量 nextpid，每分配一个 PID 号，nextpid 就加 1，pid_lock 是自选锁避免多线程写入 nextpid 导致数据竞争：

```

// kernel/proc.c
int nextpid = 1;
struct spinlock pid_lock;
...

int
allocpid()
{
    int pid;

    acquire(&pid_lock);
    pid = nextpid;
    nextpid = nextpid + 1;
    release(&pid_lock);

    return pid;
}

```

proc_pagetable() 这个函数创建了一个空的页表，并映射了 TRAMPOLINE 以及 TRAPFRAME 这两个虚拟地址，这两个地址都是固定的，TRAMPOLINE 在内核页表中也是映射在同样的物理内存上，TRAPFRAME 映射到进程的 p->trapframe 上，这个变量前面已经提到了。

```

pagetable_t
proc_pagetable(struct proc *p)
{
    pagetable_t pagetable;

    // An empty page table.

```

```

// 新创建一个空的页表
pagetable = uvmcreate();
if(pagetable == 0)
    return 0;

// map the trampoline code (for system call return)
// at the highest user virtual address.
// only the supervisor uses it, on the way
// to/from user space, so not PTE_U.
// 将虚拟地址 TRAMPOLINE 映射到物理地址 trampoline（全局符号）上
// TRAMPOLINE 页面只有 S 模式下才能使用
if(mappages(pagetable, TRAMPOLINE, PGSIZE,
            (uint64)trampoline, PTE_R | PTE_X) < 0){
    uvmfree(pagetable, 0);
    return 0;
}

// map the trapframe page just below the trampoline page, for
// trampoline.S.
// 将虚拟地址 TRAPFRAME 映射到进程控制块的成员 p->trapframe 上
if(mappages(pagetable, TRAPFRAME, PGSIZE,
            (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
}

return pagetable;
}

```

6.2.4. 进程的上下文切换

进程分配的最后一个地方可能不是很好理解，这里来详细介绍一下进程的上下文切换机制。首先看下 `struct context` 结构体，进程控制块中也有一个 `context`，`p->context` 是进程上下文切换需要保存的一些寄存器。

```

struct context {
    uint64 ra;
    uint64 sp;

    // callee-saved
    uint64 s0;
    uint64 s1;
    uint64 s2;
    uint64 s3;
    uint64 s4;
    uint64 s5;
}

```



```
uint64 s6;
uint64 s7;
uint64 s8;
uint64 s9;
uint64 s10;
uint64 s11;
};
```

可以看到 context 保存了 ra(当前函数的返回地址)、sp(栈指针) 以及所有的被调用者寄存器。

这里插入一下 RISC-V 中寄存器的调用规范(来自 为内核支持函数调用 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn))：

RISC-V 架构上的 C 语言调用规范对通用寄存器的使用做出了如下约定：

寄存器组	保存者	功能
a0~a7 (x10~x17)	调用者保存	用来传递输入参数。其中的 a0 和 a1 还用来保存返回值。
t0~t6(x5~x7,x28~x31)	调用者保存	作为临时寄存器使用，在被调函数中可以随意使用无需保存。
s0~s11(x8~x9,x18~x27)	被调用者保存	作为临时寄存器使用，被调函数保存后才能在被调函数中使用。

剩下的 5 个通用寄存器情况如下：

zero(x0) 之前提到过，它恒为零，函数调用不会对它产生影响；

ra(x1) 是被调用者保存的。被调用者函数可能也会调用函数，在调用之前就需要修改 ra 使得这次调用能正确返回。因此，每个函数都需要在开头保存 ra 到自己的栈帧中，并在结尾使用 ret 返回之前将其恢复。栈帧是当前执行函数用于存储局部变量和函数返回信息的内存结构。

sp(x2) 是被调用者保存的。这个是之后就会提到的栈指针 (Stack Pointer) 寄存器，它指向下一个将要被存储的栈顶位置。

fp(s0)，它既可作为 s0 临时寄存器，也可作为栈帧指针 (Frame Pointer) 寄存器，表示当前栈帧的起始位置，是一个被调用者保存寄存器。fp 指向的栈帧起始位置 和 sp 指向的栈帧的当前栈顶位置形成了所对应函数栈帧的空间范围。

gp(x3) 和 tp(x4) 在一个程序运行期间都不会变化，因此不必放在函数调用上下文中。它们的用途在后面的章节会提到。

然后是 swtch 函数，它定义在汇编代码 kernel/swtch.S 中：

```
.globl swtch
swtch:
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
```

```

sd s3, 40(a0)
sd s4, 48(a0)
sd s5, 56(a0)
sd s6, 64(a0)
sd s7, 72(a0)
sd s8, 80(a0)
sd s9, 88(a0)
sd s10, 96(a0)
sd s11, 104(a0)

```

```

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

```

```
ret
```

`swtch` 函数有两个参数，第一个参数是当前进程的上下文，第二个参数是待切换的新进程的上下文。`swtch` 函数将当前相关寄存器保存到第一个参数中，然后从第二个参数恢复这些寄存器的内容，最后的 `sret` 指令会将程序寄存器 `PC` 设置为 `ra` 寄存器的内容，也就是第二个参数的函数返回地址。

为什么只保存了被调用者保存寄存器，是因为 `swtch` 是按照普通函数调用的，在调用 `swtch` 之前 `C` 编译器会将调用者保存寄存器保存到栈上，而剩下的被调用者保存寄存器需要由 `swtch` 函数自行保存。

调度函数 `scheduler()`，对于每个 `CPU` 来说都有一个 `scheduler` 程序，`CPU` 也有和进程控制块相同的上下文 `context` 变量，下面是 `scheduler` 函数定义：

```

void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();

    c->proc = 0;
    for(;;){
        // Avoid deadlock by ensuring that devices can interrupt.

```

```

intr_on();
// 从进程池中找到一个可运行的进程
for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == RUNNABLE) {
        // Switch to chosen process. It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        // 调用 swtch, 进行上下文切换
        // scheduler -> process
        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
    }
    release(&p->lock);
}
}
}

```

sheduler() 函数是个死循环，它一直不断的从进程池中找到一个可运行的进程，然后调用 swtch 进行上下文切换。切换到新的进程后，执行控制流就不在这里了。

下面是 sched() 函数，这个函数相当于从一个进程执行控制流切换到 scheduler 调度函数中，切回到 scheduler 函数后，调度程序会继续从进程池找找到可运行的其他进程，然后再发现上下文切换。

```

void
sched(void)
{
    int intena;
    struct proc *p = myproc();

    if(!holding(&p->lock))
        panic("sched p->lock");
    if(mycpu()->noff != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");
    if(intr_get())
        panic("sched interruptible");

    intena = mycpu()->intena;
}

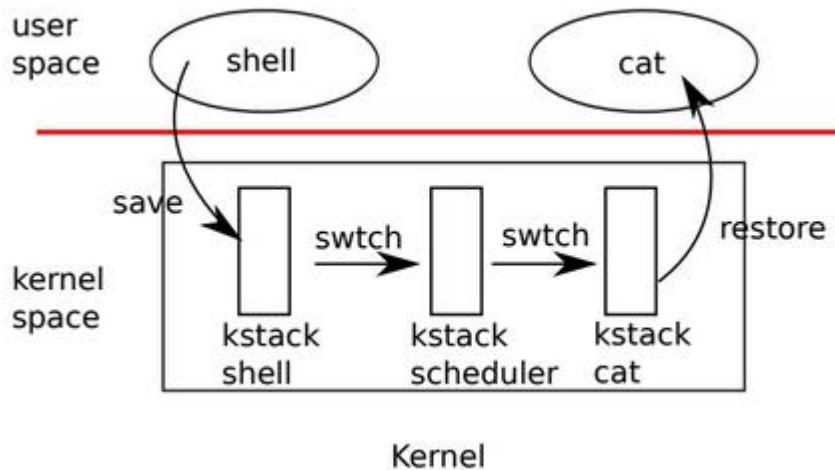
```

```

swtch(&p->context, &mycpu()->context);
mycpu()->intena = intena;
}

```

xv6 book 中有一张图更加详细的解释了两个用户进程之间是如何发生上下文切换的：



对于任意的用户进程来说，想切换到其他的进程都是要进程 scheduler 调度函数的。

6.2.5. init 进程

在 kernel/main.c 中调用了 userinit() 函数来初始化第一个进程。

```

void
userinit(void)
{
    struct proc *p;
    // 从进程池中分配一个进程
    p = allocproc();
    // 设置全局的 initproc
    initproc = p;

    // allocate one user page and copy initcode's instructions
    // and data into it.
    // 创建新的进程页表，并将 initcode 的内容写入到物理地址中
    // 这里写入的地址就是从进程内核地址空间起始地址 0 开始的
    uvmfirst(p->pagetable, initcode, sizeof(initcode));
    p->sz = PGSIZE;

    // prepare for the very first "return" from kernel to user.
    // 设置 epc 寄存器，返回到用户空间的时候，用户程序计数器会设置为 0
    p->trapframe->epc = 0;          // user program counter
    // 用户栈，从 PGSIZE 开始，这里因为 initcode 大小不会超过 PGSIZE
    // 所以设置为 PGSIZE 没有关系
}

```

```
p->trapframe->sp = PGSIZE; // user stack pointer

// 设置进程的名字、cwd 以及 state 变量
safestrcpy(p->name, "initcode", sizeof(p->name));
p->cwd = namei("/");

p->state = RUNNABLE;

release(&p->lock);
}
```

userinit 函数最主要的工作是，创建了用户进程的页表，并将 initcode 的内容写入到虚拟地址 0 开始的位置上。

init 进程是一个无限循环，先创建一个子进程，子进程用来执行 sh 进程（即 Shell 命令行），然后父进程一直等待，如果发现是 shell 进程退出，需要重启一下。或者有孤儿进程退出了，这里不需要做任何处理，因为 wait 系统调用已经回收孤儿进程相关资源了。

7. 系统调用

7.1. 系统调用内核的原理

系统调用内核的代码主要封装在 kernel 文件夹下的 syscall.c 文件中。

系统调用（以下简称 syscall）用于在用户程序中执行一些特权模式下才能完成的操作（如 I/O 操作）。对于处理器而言，syscall 是一种同步发生的事件，在基于 RISC-V 的操作系统中由 ecall 指令（Environment Call）产生。当系统调用事件结束后，处理器将返回到 ecall 指令的下一条指令继续执行。

原版 xv6 的文档将“systemcall”、“exception”和“interrupt”统称为“trap”。这与我们平常使用的术语有一定出入。在 RISC-V 指令集手册的 1.6 节中，对“exception”、“interrupt”和“trap”的定义是这样的：

We use the term exception to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart. We use the term interrupt to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control. We use the term trap to refer to the transfer of control to a trap handler caused by either an exception or an interrupt.

从这个角度来看，trap 在 RISC-V 的语境下指的是控制权的转换，也就是特权级的转换。

7.1.1. 在 RISC-V 中请求系统调用

在 RISC-V 的用户程序中，使用 ecall 指令请求 syscall。操作系统提供不同功能的 syscall，对应不同的请求调用号。因此请求前，需要向 a7 寄存器中写入所请求的 syscall 的调用号。以下为一个请求 8 号调用的例子：

```
li a7, 8
ecall
```

7.1.2. 系统调用的处理过程

内核态和用户态

对于用户程序而言，一般是从用户模式（U 模式）陷入到监管模式（S 模式）。中断向量表，保存着各类异常与中断的处理例程的入口，`syscall` 的处理例程也在其中。对于不同类型的调用请求，`syscall` 的处理例程也有相应的系统调用表，保存不同类型的系统调用功能函数的入口。根据用户事先写入寄存器的系统调用号，`syscall` 的处理例程调用相应的功能函数，完成系统调用，随后返回至原先的特权态。

那么，执行 `ecall` 指令后，系统中发生了什么呢？其实不仅是系统调用，异常与各类中断发生后，首先都会经过这一过程——特权态的转换。

在 RISC-V 中，各个特权级都有一组状态与控制寄存器（CSRs），当在 U 模式下中断发生时（除了时钟中断），RISC-V 硬件系统会自动完成下列操作：

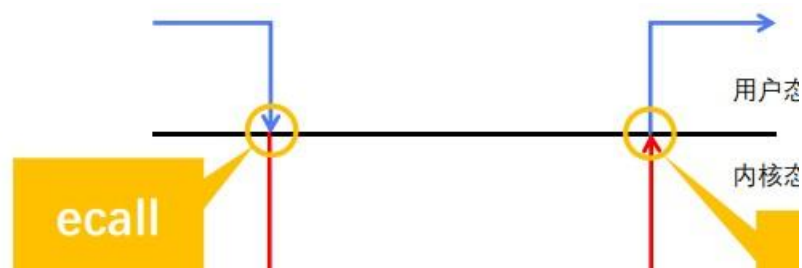
- 1 如果发生的是设备中断，且 `sstatus` 寄存器的 SIE 位为 0（即中断关闭），则不继续下列的操作，也就是暂时忽略这个中断；
- 2 关闭外部设备中断，即将 `sstatus` 寄存器的 SIE 位设为 0；
- 3 拷贝 `pc` 寄存器的值至 `sepc` 寄存器；
- 4 保存当前的特权态信息至 `sstatus` 寄存器的 SPP 字段；
- 5 设置 `scause` 寄存器，其值表示 `trap` 产生的原因；
- 6 将特权态设置为 S 模式；
- 7 将 `stvec` 寄存器的值存入 `pc` 寄存器（`stvec` 寄存器存放 S 模式的中断处理函数的入口地址）；
- 8 从新的 `pc` 值指向的指令继续执行。

执行上述步骤后，处理机就进入了 S 模式。此时，操作系统可能还需要进行一些额外操作，以应对 S 模式下再次发生 `trap` 事件。

之后，根据 `scause` 寄存器，判断发生的是异常、中断或是系统调用，并进行相应的处理。处理完毕后，就执行 `sret` 指令，返回 U 模式。

`sret` 指令会执行若干相反的操作，例如将 `sepc` 的值写回 `pc`、更新 `sstatus` 寄存器的 SIE 字段等，使处理机恢复到原本 U 模式的执行状态。

系统模式的切换



ECALL 命令用于主动触发异常，根据调用 ECALL 的权限级别产生不同的 exception code，异常产生时 `epc` 寄存器的值存放的是 ECALL 指令本身的地址。

系统调用的传参

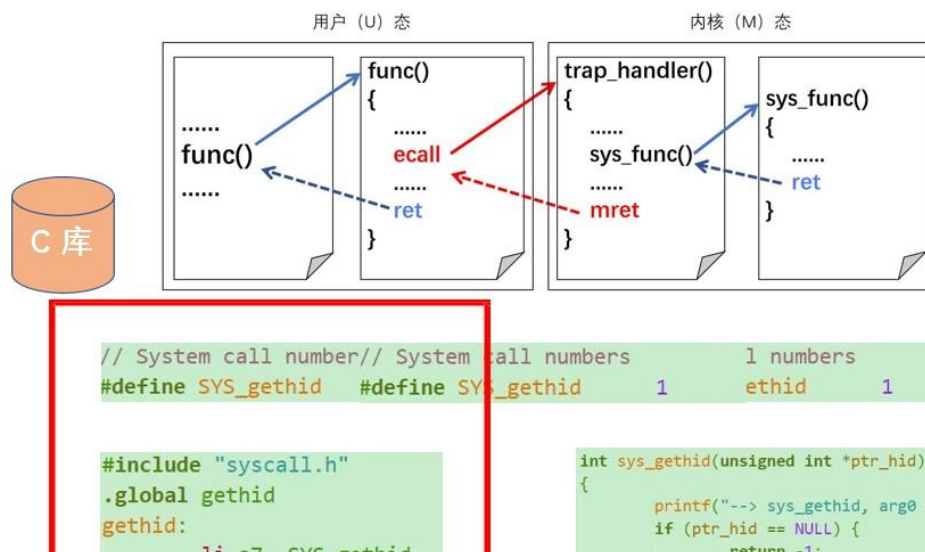
参考 Linux 的系统调用，RVOS 定义系统调用的传参规则如下：

系统调用号放在 `a7` 中

系统调用参数使用 `a0 ~ a5`

返回值使用 `a0`

系统调用的封装



7.2. 相关系统调用的实现

7.2.1. 进程管理系统调用

进程管理的系统调用实现模块主要实现在 kernel 文件夹下的 sysproc.c 文件中。主要的实现函数有：

序号	系统函数名	功能
1	sys_clone	创建新进程
2	sys_wait4	等待子进程退出
3	sys_sleep	让进程睡眠指定的秒数
4	sys_exit	进程退出
5	sys_getpid	获取当前进程的进程ID
6	sys_getppid	获取当前进程的父进程ID
7	sys_fork	创建一个新的子进程
8	sys_rt_sigtimedwait	等待并处理指定的信号，当前未实现
9	execve	执行程序

sys_clone

作用：创建一个新的子进程或线程。

实现：此函数首先获取相关的参数，如栈地址、父线程 ID、TLS 描述符等，最后调用 do_clone2 完成进程或线程的创建。

```

uint64 sys_clone(void)
{
    uint64 flags, stack, ptid, tls, ctid;

    // 获取用户传入的参数，包括标志、栈地址、父线程 ID (ptid)、线程本地存储 (tls) 描述符、子线程 ID (ctid)
    if (argaddr(0, &flags) < 0 || argaddr(1, &stack) < 0 || argaddr(1, &ptid) < 0
    || argaddr(1, &tls) < 0 || argaddr(1, &ctid) < 0)

```

```

return -1; // 如果获取参数失败，返回 -1

// 再次获取栈地址（此处为 stack 的地址）
argaddr(1, &stack);

// 如果栈地址为 0，表示没有提供栈地址，直接创建一个子进程
if (stack == 0) {
    return fork(); // 调用 fork 创建子进程
}

// 调用 do_clone2 函数创建新进程或线程，并传递相关参数
return do_clone2(flags, stack, ptid, tls, ctid);
}

```

sys_wait4

作用：等待一个指定进程的状态变化，并获取进程的退出状态。

实现：这个系统调用扩展了 `wait`，可以等待特定的进程，并提供更多的选项，如 `WNOHANG`（不挂起）等。

```

uint64 sys_wait4()
{
    uint64 addr;
    int pid, options;

    // 获取系统调用参数，pid 为等待的子进程 ID
    if (argint(0, &pid) < 0) {
        return -1; // 获取 pid 参数失败，返回 -1
    }

    // 获取系统调用参数，addr 为存储子进程状态的地址
    if (argaddr(1, &addr) < 0) {
        return -1; // 获取 addr 参数失败，返回 -1
    }

    // 获取系统调用参数，options 为等待选项，如 WNOHANG 等
    if (argint(2, &options) < 0) {
        return -1; // 获取 options 参数失败，返回 -1
    }

    // 调用 wait4pid 函数处理进程等待并返回结果
    return wait4pid(pid, addr, options);
}

```

sys_sleep

作用：使当前进程进入休眠状态，休眠一定的时间（tick 数）。

实现：通过 `ticks` 变量记录系统时钟的滴答数，当经过指定的时间后，恢复进程的执行。


```

uint64 sys_sleep(void)
{
    int n;
    uint ticks0;

    // 获取用户传入的休眠时间（单位为滴答数）
    if (argint(0, &n) < 0)
        return -1; // 如果获取失败，返回 -1

    // 获取系统时钟的锁，确保操作的原子性
    acquire(&tickslock);

    // 获取当前的系统时钟滴答数
    ticks0 = ticks;

    // 进入循环，检查当前的系统时钟滴答数与初始时刻的差值是否大于等于 n
    while (ticks - ticks0 < n) {
        // 如果当前进程被标记为“杀死”，则释放锁并返回 -1
        if (myproc()->killed) {
            release(&tickslock);
            return -1;
        }

        // 当前进程休眠，直到 ticks 变量发生变化
        sleep(&ticks, &tickslock);
    }

    // 释放时钟锁
    release(&tickslock);

    // 成功休眠指定时间后，返回 0
    return 0;
}

```

sys_exit

作用：该系统调用用于终止当前进程，传递给父进程终止状态。

实现：通过 `exit` 函数实现进程的退出，并将退出状态返回给父进程。

```

uint64 sys_exit(void)
{
    int n;

    // 获取用户传入的退出状态值
    if (argint(0, &n) < 0)
        return -1; // 如果获取失败，返回 -1
}

```

```

// 调用 exit 函数终止当前进程，并传递退出状态
exit(n);

// 如果 exit 执行成功，这一行代码将不会被执行，因为进程已终止
return 0; // not reached
}

```

sys_getpid

作用：返回当前进程的进程 ID（PID）。

实现：直接从当前进程结构体 `myproc()` 中获取并返回 PID。

```

sys_getpid(void)
{
    return myproc()->pid;
}

```

进程 ID 获取：通过 `myproc()` 获取当前进程的内核数据结构 `proc`，从中提取进程 ID（pid）。该实现假设当前进程结构体是通过 `myproc` 函数访问的。

sys_getppid

作用：返回当前进程的父进程 ID（PPID）。每个进程在创建时都会由另一个进程（即父进程）创建，`getppid` 提供了一种方式来查询父进程的信息。

函数原型

```

int getppid(void);

```

实现：获取当前进程的父进程，并返回其 PID。如果当前进程没有父进程（例如在系统初始化时），则返回 -1。

```

uint64 sys_getppid(void)
{
    // 获取当前进程的进程结构体
    struct proc* p = myproc();
    // 获取当前进程的锁，以确保访问进程信息时的同步
    acquire(&p->lock);
    // 获取当前进程的父进程 PID
    uint64 ppid = p->parent->pid;
    // 释放当前进程的锁
    release(&p->lock);
    // 返回父进程 ID
    return ppid;
}

```

sys_fork

作用：创建一个新的进程（子进程）。新创建的进程是当前进程的副本，其内存空间、文件描述符等资源会被复制给新进程。父进程和子进程从 `fork` 返回的地方继续执行，父进程获取子进程的 PID，子进程则获得 0。

函数原型

```

int fork(void);

```

实现：通过调用 `fork` 函数来创建子进程。子进程会从调用 `fork` 的地方开始执行。在父进程中返回子进程的 PID，在子进程中返回 0。

```

uint64 sys_fork(void) {
    struct proc *p = myproc();
    struct proc *np; // 子进程

    // 分配内存并创建新进程
    if ((np = allocproc()) == 0)
        return -1;

    // 复制父进程的内存、寄存器、文件描述符等信息
    np->parent = p;
    *np = *p; // 复制父进程的所有信息

    // 为新进程分配堆栈等资源
    np->pid = nextpid++;

    // 调用调度器分配 CPU 资源
    sched();

    return np->pid;
}

```

sys_rt_sigtimedwait

作用：等待指定的信号。

目前尚未完成，可能的实现方法：sys_rt_sigtimedwait 是一个用于等待信号的系统调用，通常用于进程等待并处理实时信号。在 Linux 和类 Unix 操作系统中，sigtimedwait 用于在给定时间内等待特定的信号。它在指定的信号集上等待信号，并可以处理阻塞信号集，返回已接收到的信号。如果超时发生或没有信号被接收，sigtimedwait 会返回。

execve-执行程序

execve 是一个非常重要的系统调用，它允许当前进程加载并执行一个新的程序映像。调用 execve 后，原进程的地址空间将被新的程序所替代，当前进程将开始执行新的程序。该调用的实现要求内核具备灵活的内存管理能力，以便在不同程序之间进行切换，并确保新程序的环境变量、命令行参数等正确传递。

函数原型：

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

核心实现：

```

uint64 sys_execve(void) {
    char path[FAT32_MAX_PATH], *argv[MAXARG], *environ[MAXARG];
    uint64 uargv, uenviron, uarg;
    int i;

    // 获取路径、命令行参数和环境变量
    if (argstr(0, path, sizeof(path)) < 0 ||
        argaddr(1, &uargv) < 0 ||

```

```

        argaddr(2, &uenvIRON) < 0) {
            return -1;
        }

// 获取命令行参数 argv
if (parse_argv(uargv, argv, MAXARG) < 0) {
    return -1;
}

// 获取环境变量 environ
if (parse_argv(uenvIRON, environ, MAXARG) < 0) {
    cleanup_argv(argv);
    return -1;
}

// 调用 exec 函数执行目标程序
int ret = exec(path, argv, environ);

// 清理内存
cleanup_argv(argv);
cleanup_argv(environ);

return ret;
}

// 解析参数数组
int parse_argv(uint64 uargv, char *argv[], int maxargs) {
    uint64 uarg;
    for (int i = 0; i < maxargs; i++) {
        if (fetchaddr(uargv + sizeof(uint64) * i, &uarg) < 0) {
            return -1;
        }
        if (uarg == 0) {
            argv[i] = 0;
            break;
        }
        argv[i] = kalloc();
        if (!argv[i] || fetchstr(uarg, argv[i], PGSIZE) < 0) {
            cleanup_argv(argv);
            return -1;
        }
    }
    return 0;
}
}

```

```
// 清理内存
void cleanup_argv(char *argv[]) {
    for (int i = 0; argv[i] != 0; i++) {
        kfree(argv[i]);
    }
}
```

7.2.2. 内存管理系统调用

内存管理的系统调用实现模块主要实现在 `kernel` 文件夹下的 `sysmem.c` 文件中。主要实现函数如下：

序号	系统函数名	功能
1	<code>sys_brk</code>	调整当前进程的数据段大小
2	<code>sys_mmap</code>	将文件或设备映射到内存
3	<code>sys_munmap</code>	将文件或设备取消映射到内存中
4	<code>munmap</code>	取消指定内存区域的映射
5	<code>free_page</code>	释放物理内存页

`sys_brk`

`sys_brk` 系统调用的作用是调整进程的数据段大小。进程的堆内存是动态分配的，`brk` 系统调用允许进程增加或减少其堆的大小。

实现方法：

1. 获取参数：通过 `argint(0, &new_addr)` 获取传入的新地址（`new_addr`），表示数据段的大小。
2. 检查地址变化：获取当前进程的数据段结束地址 `myproc()->sz`，与新地址 `new_addr` 比较。
3. 调整内存大小：如果新的地址非零，则调用 `growproc(new_addr - addr)` 来调整内存空间。`growproc` 函数会根据给定的大小调整进程的堆内存。
4. 返回值：成功时返回新的数据段大小，失败时返回 `-1`。

```
uint64 sys_brk(void)
{
    int addr;
    int new_addr;

    // 获取用户传入的新的数据段大小地址
    if (argint(0, &new_addr) < 0)
        return -1; // 获取失败，返回 -1

    // 获取当前进程的数据段大小（进程的虚拟内存大小）
    addr = myproc()->sz;

    // 如果 new_addr 为 0，表示查询当前数据段大小，直接返回
    if (new_addr == 0)
```

```

{
    return addr;
}

// 如果 new_addr 不为 0，调用 growproc 函数调整进程的内存大小
// 如果调整失败，返回 -1
if (growproc(new_addr - addr) < 0)
    return -1;

// 返回新的数据段大小
return new_addr;
}

```

sys_mmap

sys_mmap 是一个将文件或设备映射到进程的虚拟内存中的系统调用。通常用于共享内存或将文件内容映射到内存中，这样可以让应用程序直接访问文件中的数据。映射区域具有指定的保护方式（如可读、可写、可执行）和映射标志（如是否共享）。

实现方法：

1. 获取用户传入的参数：通过 **argaddr** 和 **argint** 获取用户传入的起始地址、映射长度、保护方式、标志、文件描述符和偏移量等。
2. 校验文件描述符：检查文件描述符是否有效，文件描述符超出范围则返回 -1。
3. 调用 **mmap** 函数：传递获取的参数给 **mmap** 函数，执行内存映射操作。

```

uint64 sys_mmap()
{
    uint64 start_addr;
    int prot, flags, fd, off, len;

    // 获取传入的参数：映射起始地址、长度、保护方式、标志、文件句柄和偏移量
    if (argaddr(0, &start_addr) < 0 || argint(1, &len) < 0 || argint(2, &prot) < 0)
        return -1; // 如果获取失败，返回 -1

    if (argint(3, &flags) < 0 || (argint(4, &fd) < 0) || argint(5, &off) < 0)
        return -1; // 如果获取失败，返回 -1

    // 检查文件描述符是否有效
    if (fd < 0 || fd >= NOFILE)
        return -1; // 文件描述符无效，返回 -1

    // 调用 mmap 函数执行内存映射操作
    return mmap(start_addr, len, prot, flags, fd, off);
}

```

sys_munmap

`sys_munmap` 系统调用的作用是取消之前通过 `mmap` 映射到进程虚拟内存中的区域。该系统调用通过给定的地址和长度，解除映射并释放占用的内存。

实现方法：

1. 获取用户传入的地址和长度：通过 `argaddr` 获取映射的起始地址和长度。
2. 验证地址和长度：如果地址为 0 或长度小于等于 0，则返回 -1。
3. 调用 `munmap` 函数：通过 `munmap` 函数执行具体的取消映射操作。
4. 返回值：如果 `munmap` 成功，返回 0；失败则返回 -1。

```
uint64 sys_munmap()
{
    uint64 start, len;

    // 获取用户传入的起始地址和长度
    if (argaddr(0, &start) < 0 || argaddr(1, &len) < 0) {
        return -1; // 如果获取参数失败，返回 -1
    }

    // 验证传入的地址和长度是否合法
    if (start == 0 || len <= 0) {
        return -1; // 无效的地址或长度，返回 -1
    }

    // 调用函数取消映射
    int result = munmap(start, len);

    // 如果取消映射成功，返回 0，否则返回 -1
    if (result == 0) {
        return 0;
    } else {
        return -1;
    }
}
```

munmap

`munmap` 函数的作用是解除内存映射，将映射区域从进程的虚拟内存中移除，并释放相关的物理内存。它遍历给定的地址区间，对于每一页，清除页表项并释放对应的物理内存。

```
/*
    取消指定内存区域的映射；

    输入：映射的起始地址和长度；

    返回值：成功返回 0，失败返回 -1；
*/
int munmap(uint64 start, uint64 len)
{
```

```

// 获取当前进程的页表
struct proc *p = myproc();
pte_t *pte;

// 确保地址范围内的每一页都有效
for (uint64 addr = start; addr < start + len; addr += PGSIZE) {
    // 获取当前地址对应的页表项
    pte = walkpgdir(p->pagetable, addr, 0);

    // 如果页表项无效，表示该地址没有映射，直接返回 -1
    if (*pte == 0) {
        return -1; // 页表项无效，表示未映射
    }

    // 解除映射，将页表项清空
    *pte = 0;

    // 释放该页占用的物理内存
    uint64 pa = PTE2PA(*pte);
    if (pa != 0) {
        free_page(pa); // 释放物理页
    }
}

return 0; // 成功取消映射
}

```

free_page

free_page 函数用于释放给定物理内存页。该函数将指定的物理内存页返回给系统的内存池，使其能够被重新分配给其他进程或用于其他目的。

实现方法：

1. 释放物理页：调用内核中的 **kfree** 函数来释放物理内存页。在这里，**pa** 表示物理地址，**kfree** 函数将物理内存页标记为可用，通常是将其返回到空闲内存池中。
2. 物理地址转换：**kfree** 函数通常是基于虚拟地址操作的，因此需要将物理地址 **pa** 转换为相应的虚拟地址，以便内存管理系统能够处理。

```

/*
    释放指定的物理页；

    输入：物理页的物理地址；

    返回值：无返回值；
*/
void free_page(uint64 pa)
{

```



```
// 释放物理内存页
kfree((void*)pa); // 调用内核的内存释放函数
}
```

7.2.3. 文件系统系统调用

这一部分着重介绍文件系统的系统调用的底层实现。

getcwd — 获取当前工作目录

getcwd 系统调用提供了当前进程工作目录的获取功能。通过递归遍历目录树，最终将完整的工作目录路径以字符串形式返回到用户空间。

函数原型：

```
char* getcwd(char *buf, int size);
```

代码实现：

```
uint64 sys_getcwd(void) {
    uint64 addr;
    if (argaddr(0, &addr) < 0) return -1;

    int n;
    if (argint(1, &n) < 0) return -1;

    struct dirent *de = myproc()->cwd;
    char path[FAT32_MAX_PATH];
    char *s = path + FAT32_MAX_PATH - 1;
    *s = '\0';

    if (de->parent == NULL) {
        *--s = '/';
    } else {
        while (de->parent) {
            int len = strlen(de->filename);
            if (s - len <= path) return -1;
            s -= len;
            memcpy(s, de->filename, len);
            *--s = '/';
            de = de->parent;
        }
    }

    if (copyout2(addr, s, n) < 0) return -1;

    return addr;
}
```

pipe — 创建管道

pipe 系统调用在内核中创建了一个数据通道，并返回两个文件描述符，分别对应管道的读端和写端。此调用广泛应用于进程间通信（IPC）。

```
int pipe(int *fdarray);
```

代码实现:

```
uint64 sys_pipe(void) {
    uint64 fdarray;
    if (argaddr(0, &fdarray) < 0) return -1;

    struct file *rf, *wf;
    if (pipealloc(&rf, &wf) < 0) return -1;

    int fd0 = fdalloc(rf);
    int fd1 = fdalloc(wf);
    if (fd0 < 0 || fd1 < 0) {
        if (fd0 >= 0) myproc()->ofile[fd0] = 0;
        fileclose(rf);
        fileclose(wf);
        return -1;
    }

    if (copyout2(fdarray, (char*)&fd0, sizeof(fd0)) < 0 ||
        copyout2(fdarray + sizeof(fd0), (char*)&fd1, sizeof(fd1)) < 0) {
        myproc()->ofile[fd0] = 0;
        myproc()->ofile[fd1] = 0;
        fileclose(rf);
        fileclose(wf);
        return -1;
    }

    return 0;
}
```

dup — 复制文件描述符

dup 系统调用为现有的文件描述符创建一个新的副本。新旧文件描述符共享相同的文件对象，并且操作互相影响。

函数原型:

```
int dup(int fd);
```

代码实现:

```
uint64 sys_dup(void) {
    struct file *f;
    int fd;
    if (argfd(0, 0, &f) < 0) return -1;

    if ((fd = fdalloc(f)) < 0) return -1;

    filedup(f);
    return fd;
}
```

dup2 — 将文件描述符复制到指定描述符

dup2 的核心功能在于允许用户将文件描述符复制到指定位置。如果目标描述符已被占用，系统会先关闭该描述符。

函数原型：

```
int dup2(int fd, int n);
```

代码实现：

```
uint64 sys_dup2(void) {
    struct file *f;
    int fd, n;
    if (argfd(0, 0, &f) < 0 || argint(1, &n) < 0) return -1;

    if (n != fd) close(n);

    if ((fd = fdalloc(f)) < 0) return -1;

    filedup(f);
    return fd;
}
```

chdir — 改变当前工作目录

通过 **chdir** 系统调用，进程可以动态调整其工作目录，从而影响后续相对路径的解析。

函数原型：

```
int chdir(const char *path);
```

代码实现：

```
uint64 sys_chdir(void) {
    char path[512];
    if (argstr(0, path, sizeof(path)) < 0) return -1;

    struct inode *ip = namei(path);
    if (ip == NULL || ip->type != T_DIR) return -1;

    myproc()->cwd = ip;
    return 0;
}
```

close — 关闭文件描述符

函数原型：

```
int close(int fd);
```

代码实现：

```
uint64 sys_close(void) {
    int fd;
    struct file *f;
    if (argfd(0, &fd, &f) < 0) return -1;
```

```
myproc()->ofile[fd] = 0;
fclose(f);
return 0;
}
```

read — 从文件读取数据

read 系统调用允许从文件描述符指定的文件中读取数据，并将其存入用户提供的缓冲区中。此调用实现了文件输入操作。

函数原型：

```
ssize_t read(int fd, void *buf, size_t count);
```

实现分析

输入参数：

fd：文件描述符，标识读取操作的目标文件。

buf：指向用户空间的缓冲区，用于存储读取到的数据。

count：请求读取的字节数。

返回值：返回成功读取的字节数；如果到达文件末尾，返回 0；失败返回 -1。

代码实现：

```
uint64 sys_read(void) {
    struct file *f;
    int n;
    char *buf;

    if (argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argaddr(1, &buf) < 0) return
-1;

    return fileread(f, buf, n);
}
```

write — 向文件写入数据

write 系统调用实现了文件输出操作。通过此调用，可以将用户缓冲区的数据写入到文件或其他支持写操作的目标中。

函数原型：

```
ssize_t write(int fd, const void *buf, size_t count);
```

实现分析

输入参数：

fd：文件描述符，标识写入目标。

buf：用户空间的缓冲区，包含需要写入的数据。

count：要写入的字节数。

返回值：成功写入的字节数；失败返回 -1。

代码实现：

```
uint64 sys_write(void) {
    struct file *f;
    int n;
    char *buf;
```

```

    if (argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argaddr(1, &buf) < 0) return
    -1;

    return filewrite(f, buf, n);
}

```

unlink — 删除文件

unlink 系统调用用于从文件系统中删除指定的文件或目录链接。如果删除的是目录，其必须为空。

函数原型：

```
int unlink(const char *pathname);
```

代码实现：

```

uint64 sys_unlink(void) {
    char path[512];
    struct inode *ip;

    if (argstr(0, path, sizeof(path)) < 0) return -1;
    if ((ip = namei(path)) == 0) return -1;

    if (ip->type == T_DIR) return -1; // 禁止直接删除非空目录

    ilock(ip);
    itrunc(ip); // 清空文件内容
    iput(ip);   // 释放 inode
    return 0;
}

```

mkdir — 创建目录

mkdir 系统调用为文件系统提供了目录创建功能。目录是文件系统中管理文件层次结构的重要组成部分。

函数原型：

```
int mkdir(const char *pathname, mode_t mode);
```

实现分析

输入参数：

- **pathname**: 要创建的目录路径。
- **mode**: 目录的权限模式。
- 返回值: 成功返回 0; 失败返回 -1。

代码实现：

```

uint64 sys_mkdir(void) {
    char path[512];
    struct inode *ip;

    if (argstr(0, path, sizeof(path)) < 0) return -1;

    begin_op();
}

```

```
ip = create(path, T_DIR, 0, 0); // 创建目录类型的 inode
if (ip == NULL) {
    end_op();
    return -1;
}
iunlockput(ip);
end_op();
return 0;
}
```

fstat — 获取文件状态

fstat 系统调用用于检索文件描述符对应文件的元数据信息，例如文件大小、权限和类型。

函数原型：

```
int fstat(int fd, struct stat *buf);
```

实现分析

输入参数：

fd：文件描述符。

buf：指向用户空间的 **stat** 结构，用于存储文件状态信息。

返回值：成功返回 0；失败返回 -1。

代码实现：

```
uint64 sys_fstat(void) {
    struct file *f;
    struct stat st;

    if (argfd(0, &f) < 0) return -1;

    filestat(f, &st);
    if (copyout2((uint64)&st, &st, sizeof(st)) < 0) return -1;

    return 0;
}
```

7.2.4. 其它系统调用

这一部分详细介绍 **uname** 和 **gettimeofday** 两个系统调用，这两个系统调用为程序提供了获取操作系统信息和系统时间的基础功能。

uname — 获取系统信息

uname 系统调用是操作系统中常见的功能之一，主要用于提供关于操作系统的各种信息，包括操作系统的名称、版本、机器架构以及主机名等。其主要应用场景包括硬件抽象层的识别、操作系统版本的检测等，这些信息对运行在不同操作系统平台上的程序具有重要意义。

函数原型：

```
int uname(struct utsname *buf);
```

uname 系统调用接收一个指向 **utsname** 结构体的指针，该结构体用于存储操作系统的各项信息。该结构体包含以下几个字段：

- **sysname**: 操作系统的名称。

- nodename: 网络节点名称，通常是主机名。
- release: 操作系统的发布版本。
- version: 操作系统的具体版本号。
- machine: 系统的硬件架构。
- domainname: 操作系统的域名。

utsname 结构体的定义如下：

```
struct utsname {  
    char sysname[65];        // 操作系统名称  
    char nodename[65];       // 主机名  
    char release[65];        // 操作系统发布版本  
    char version[65];        // 操作系统版本号  
    char machine[65];        // 机器硬件架构  
    char domainname[65];     // 域名  
};
```

在 `sys_uname` 系统调用的实现中，首先通过 `argaddr` 函数获取用户传递的结构体指针，然后填充该结构体的各个字段以返回操作系统的相关信息。信息的来源可以是硬编码值或从内核数据结构中提取。在填充完结构体后，使用 `copy out2` 将内核空间的数据拷贝到用户空间。

```
uint64 sys_uname(void)  
{  
    uint64 addr;  
  
    // 获取用户传递的结构体地址  
    if (argaddr(0, &addr) < 0) {  
        return -1; // 如果地址无效，返回错误  
    }  
  
    struct utsname info;  
  
    // 动态获取操作系统信息，减少硬编码（如果有相关内核 API 或配置可以获取）  
    const char *sysname = "K210 OS"; // 操作系统名称  
    const char *nodename = "YouDaoLi"; // 主机名  
    const char *release = "1.0"; // 操作系统发布版本  
    const char *version = "0.1"; // 操作系统版本号  
    const char *machine = "arm64"; // 机器架构  
    const char *domainname = "local"; // 域名  
  
    // 使用安全的字符串复制函数 strncpy，并确保每个字段都有合理的大小  
    strncpy(info.sysname, sysname, sizeof(info.sysname) - 1);  
    info.sysname[sizeof(info.sysname) - 1] = '\0';  
  
    strncpy(info.nodename, nodename, sizeof(info.nodename) - 1);  
    info.nodename[sizeof(info.nodename) - 1] = '\0';
```

```
strncpy(info.release, release, sizeof(info.release) - 1);
info.release[sizeof(info.release) - 1] = '\0';

strncpy(info.version, version, sizeof(info.version) - 1);
info.version[sizeof(info.version) - 1] = '\0';

strncpy(info.machine, machine, sizeof(info.machine) - 1);
info.machine[sizeof(info.machine) - 1] = '\0';

strncpy(info.domainname, domainname, sizeof(info.domainname) - 1);
info.domainname[sizeof(info.domainname) - 1] = '\0';

// 将结构体数据从内核空间复制到用户空间
if (copyout2(addr, (char *)&info, sizeof(info)) < 0) {
    return -1; // 如果拷贝失败，返回错误
}

return 0; // 成功返回 0
}
```

gettimeofday — 获取系统时间

`gettimeofday` 系统调用是用于获取系统当前时间的常用接口。它返回自 Unix 纪元以来的时间戳，可以精确到微秒。这一系统调用广泛用于测量时间、计算延迟、生成时间戳等场景。其实现需要依赖内核的时钟机制，通过硬件计时器来提供高精度的时间数据。

函数原型：

```
int gettimeofday(struct timespec *tv, void *tz);
```

`gettimeofday` 系统调用通过 `tv` 参数返回当前的时间信息，`tv` 是一个指向 `timespec` 结构体的指针。该结构体包含两部分数据：

- `sec`: 自 Unix 纪元以来的秒数。
- `usec`: 微秒数，表示秒数的精确小数部分。

`timespec` 结构体的定义如下：

```
struct timespec {
    long sec; // 秒数
    long usec; // 微秒数
};
```

在 `sys_gettimeofday` 的实现中，首先通过 `argaddr` 获取用户传递的 `timespec` 结构体指针。然后，使用硬件时钟的计数器读取当前时间，假设系统的时钟频率为 32768 Hz（每秒 32768 次计数），从而通过计数器值计算出当前的秒数和微秒数。

```
uint64 sys_gettimeofday(void)
{
    timespec *ptr;
```



```

// 获取用户传递的 timespec 结构体地址
if (argaddr(0, (uint64*)&ptr) < 0) {
    return -1; // 如果地址无效, 返回错误
}

// 获取硬件计时器的当前值, 假设时钟频率为 32768 Hz
uint64 clock_frequency = get_clock_frequency(); // 例如从硬件寄存器或
配置获取频率
if (clock_frequency == 0) {
    return -1; // 如果时钟频率无效, 返回错误
}

// 获取当前系统的时钟计数
uint64 time_count = r_time(); // 读取当前时钟计数

// 计算秒和微秒
ptr->sec = time_count / clock_frequency;
ptr->usec = (time_count % clock_frequency) * 1000000 / clock_frequency;

return 0; // 成功返回 0
}

```

8. 运行界面示例

8.1. 单个命令测试

- 在 VScode 中, 连接好虚拟机以后, 进入项目文件夹, 打开命令窗口, 输入如下命令:

```
make fs
```

得到如下结果:

```

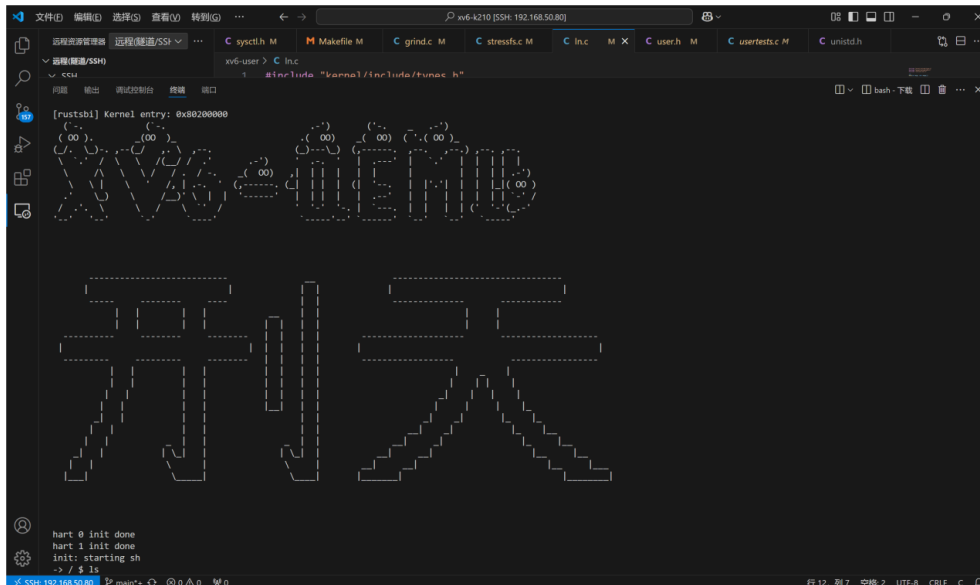
cong@cong-virtual-machine:~/xv6-k210-chage$ make fs
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -g -
l=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-pro
QEMU -c -o xv6-user/ulib.o xv6-user/ulib.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -g -
l=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-pro
QEMU -c -o xv6-user/usys.o xv6-user/usys.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -g -

```

- 然后再输入如下命令:

```
make run platform=qemu
```

运行会得到如下界面:



说明系统运行成功。

- 使用如下命令可以查看当前系统中的进程情况：
- 输入如下命令可以查看当前目录下的文件：

```
ls
```

执行结果：

```
hart 0 init done
hart 1 init done
init: starting sh
-> / $ ls
bin                               DIR      0
README                           FILE     2104
cat                               FILE     25288
echo                             FILE     24120
find                             FILE     26296
grep                             FILE     28592
init                             FILE     24656
kill                             FILE     24064
ls                               FILE     26408
mkdir                            FILE     24192
mv                               FILE     26000
rm                               FILE     24136
sh                               FILE     50712
sleep                           FILE     24120
strace                          FILE     24488
test                            FILE     23824
usertests                       FILE     135240
wc                              FILE     26488
xargs                           FILE     25936
forktest                        FILE     14304
zombie                          FILE     23448
```

- 输入如下命令,可以测试基本的文件读写功能。

```
echo hello,刑天! > hello
cat hello
```

执行结果:

```
hart 0 init done
hart 1 init done
init: starting sh
-> / $ echo hello,刑天! > hello
-> / $ cat hello
hello,刑天!
-> / $
```

使用 `echo` 命令创建了一个包含 `hello,刑天!` 的文件 `hello`。
使用 `cat` 命令读取并显示了该文件的内容。

- 输入如下命令,可以验证 `fork` 创建新进程的系统调用是否被执行成功。

```
forktest
```

执行结果:

```
-> / $ forktest
fork test
fork test OK
```

`fork test OK`: 表示测试通过,说明 `fork` 系统调用在此系统中的实现是正确的。

- 输入如下命令,可以查看当前系统中的资源使用情况,具体包括剩余内存和当前进程的数量

```
test
```

执行结果:

```
-> / $ test
memory left: 3648 KB
process amount: 3
```

表示系统当前剩余的内存容量为 3648 KB (约 3.56 MB)。系统中运行的进程数量为 3: 就是前面测试得到的三个进程 `init`、`sh`、`zrzps`。

8.2. 多命令同时测试

编写测试文件,是 `xv6-user` 文件夹下的 `usertests.c` 文件。如下:

stresfs.c	2024/12/20 21:48	C 源文件	2 KB
test.c	2024/12/20 21:48	C 源文件	1 KB
text.txt	2024/12/20 21:48	文本文档	1 KB
ulib.c	2024/12/20 21:48	C 源文件	2 KB
umalloc.c	2024/12/20 21:48	C 源文件	2 KB
user.h	2024/12/20 21:48	C Header File	3 KB
usertests.c	2024/12/20 21:48	C 源文件	56 KB
usys.pl	2024/12/20 21:48	Perl 源文件	1 KB
wc.c	2024/12/20 21:48	C 源文件	1 KB
xargs.c	2024/12/20 21:48	C 源文件	2 KB
xargtest.sh	2024/12/20 21:48	Shell Script	1 KB
xv6-riscv-license	2024/12/20 21:48	文件	2 KB
YzY.c	2024/12/20 21:48	C 源文件	1 KB
zombie.c	2024/12/20 21:48	C 源文件	1 KB

```

2627 }
2628
2629 int
2630 main(int argc, char *argv[])
2631 {
2632     int continuous = 0;
2633     char *justone = 0;
2634
2635     if(argc == 2 && strcmp(argv[1], "-c") == 0){
2636         continuous = 1;
2637     } else if(argc == 2 && strcmp(argv[1], "-c") == 0){
2638         continuous = 2;
2639     } else if(argc == 2 && argv[1][0] != '-'){
2640         justone = argv[1];
2641     } else if(argc > 1){
2642         printf("Usage: usertests [-c] [testname]\n");
2643         exit(1);
2644     }
2645
2646     struct test {
2647         void (*f)(char *);
2648         char *s;
2649     } tests[] = {
2650         {execout, "execout"},
2651         {copyin, "copyin"},
2652         {copyout, "copyout"},
2653         {copyinstr, "copyinstr"},
2654         {copyinstr2, "copyinstr2"},
2655         {copyinstr3, "copyinstr3"},
2656         {truncate1, "truncate1"},
2657         {truncate2, "truncate2"},
2658         {truncate3, "truncate3"},
2659         {reparent2, "reparent2"},
2660         {pgbug, "pgbug"},
2661         {sbrkbugs, "sbrkbugs"},
2662         {badwrite, "badwrite"},

```

不带参数的 `usertests` 会运行所有测试，而 `usertests <name>` 会运行 `<name>` 测试。测试运行程序会为每个测试创建一个进程，并根据进程的退出状态报告“OK”或“FAILED”。某些测试会导致内核打印 `usertrap` 消息，如果测试会打印“OK”，则可忽略这些信息。

OK 表示功能正常。

`usertrap()` 异常是内核问题的信号，需要进一步分析 `scause` 和 `sepc`。

测试存储在一个结构体中：

```

struct test {
    void (*f)(char *); // 测试函数的指针
    char *s;           // 测试名称
} tests[] = {
    {execout, "execout"},
    {copyin, "copyin"},
    ...
    {forktest, "forktest"},
    { 0, 0},
};

```

➤ 输入如下命令

usertests

执行结果:

```
-> / $ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs:
usertrap(): unexpected scause 0x000000000000000c pid=3288 usertests
sepc=0x0000000000004924 stval=0x0000000000004924

usertrap(): unexpected scause 0x000000000000000c pid=3289 usertests
sepc=0x0000000000004924 stval=0x0000000000004924
OK
test badwrite: OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test removeread: OK
test subdir: OK
test fourfiles: OK
test sharedfd: OK
test exectest: OK
test bigargtest: OK
test bigwrite: OK
```

```
sepc=0x0000000000001a2c stval=0x00000000803b7740

usertrap(): unexpected scause 0x00000000000000d pid=6196 usertests
sepc=0x0000000000001a2c stval=0x00000000803c3a90

usertrap(): unexpected scause 0x00000000000000d pid=6197 usertests
sepc=0x0000000000001a2c stval=0x00000000803cfd0

usertrap(): unexpected scause 0x00000000000000d pid=6198 usertests
sepc=0x0000000000001a2c stval=0x00000000803dc130
OK
test sbrkfail:
usertrap(): unexpected scause 0x00000000000000d pid=6210 usertests
sepc=0x00000000000033b2 stval=0x000000000000f000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest:
usertrap(): unexpected scause 0x00000000000000d pid=6214 usertests
sepc=0x0000000000001baa stval=0x000000000000cc00
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openinput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
ALL TESTS PASSED
```

9. 总结体会及未来设计方向

9.1. 总结体会

参加这次操作系统内核设计比赛总体感受是然人爱恨交加，一方面通过是环境配置过程中的各种报错，修改过代码之后编译报错、运行报错、运行不出预期的结果不断打击着我们团队中的每一个人的自信心，我们不禁怀疑：这个比赛是学生参加的吗？又不由得对操作系统设计人员生起由衷的敬佩甚至一度有过退赛的念头。但是另一方面，当我们配置环境第一次运行成功，屏幕中出现 xv6 字样时，修改代码第一次没有报错时，修改代码第一次看到预想的结果时，一次次成功的时刻无不激励着我们继续比下去。无疑参加此次操作系统设计比赛，提升了我们对于操作系统内核的理解，和交叉编程能力，但是相比于这些，于我们而言，更大的收获是在比赛过程中培养锻炼出的坚韧不拔的意志力，不同于书本上的理论环境，实际环境下的编程与内核实现比理想的实验环境下要难得多得多。

在刚开始设计时，首先面临的问题就是选择做什么操作系统。平心而论，我们团队没有能力独立完成一个操作系统内核的编写，我们所取得的成果是建立在在往年学长学姐的肩膀上的。至于为什么选择 xv6 操作系统作为基础的操作系统，首要原因就是 xv6 操作系统的结构比较清晰，代码逻辑易于理解，当然，很重要的一点就是 xv6 操作系统有学长学姐的基础，对于相关的问题，我们能够请教学

长学姐。

大致方向开始落地以后，在刚开始配置环境的时候，我们自己在网上找资料配置，但是效果并不好，后来，我们参考了往年“赤道会师”编写的环境配置手册，即使这样，由于我们对虚拟机的使用并不是很熟练，因此也遇到了很多的困难，最终也是经过几天的团队沟通才完成环境的配置，并在虚拟机中成功运行了 xv6-k210 项目。并且为了节省时间，我们队伍的三个人，有两个人配环境，剩下一个人着手开始理解代码和尝试对 xv6 代码进行修改。

配置环境成功后，由于我们在配置环境之前就已经在网络上找到了 xv6 的相关资料进行学习，已经对 xv6 操作系统内核有了基础了解，于是我们可以直接正式开始对内核的编写。

由于有往年的内核资料进行参考，我们队伍进行的还是比较顺利的。在编写内核代码的同时，我们也对编程语言有了更加深刻的认识，如果说 Python 里一切皆对象的话，那么 C 语言中一切皆是地址。无论是变量、函数、结构体，还是其他类型的数据。这些标号就是内存中的地址，它们可以用不同的方式来解释和使用。不同的类型只不过代表着不同的偏移量的单位罢了，利用基址+偏移量这一简单组合，就可以自由访问或转换 C 中的一切标号。例如，一个 int 类型的变量，它的地址可以用一个 char 类型的指针来访问，从而得到它的每个字节的值；一个函数的地址，它可以用一个函数指针来调用，从而执行它的代码；一个结构体的地址，它可以用一个结构体指针来访问，从而得到它的每个成员的值。这些地址的转换，虽然看似违背了类型的规则，但却是 C 语言的强大之处，它让我们可以自由地控制内存，实现各种功能。当然，这种自由也带来了危险，如果我们不小心，就可能造成内存泄漏、越界访问、非法操作等错误，导致程序崩溃或者系统崩溃。我们在编写内核代码时，也遇到过这种非法操作问题，导致编写的代码运行不出预期的结果。

目前，我们已经实现了二十个左右的简单的系统调用，内核的基本功能差不多已经基本能够实现，虽然有时在一些极端情况下，执行这些功能可能会报错。我们目前已经停止了对系统调用的增加，转而进一步对这些偶尔会出现的问题进行排查和纠正代码，然而成效进展非常缓慢。我们认为，正如同人学习一样，只有基础打好了，才能有更高的成就，一个好的操作系统，对基础功能的实现不能出现差错。我们团队希望能够尽可能做到这些基础功能在任何情况下都不会出问题，尽管这可能会很难实现。但是“世上无难事，只怕有心人”，只要我们不放弃，不断深入学习，就一定会成功。

总而言之，通过参加这次操作系统内核的设计比赛，我们团队的每一个人都有了巨大的进步和成长。比赛过程中，一次又一次的挑战，不断磨炼我们的意志，增强我们的本领，使我们受益匪浅。

9.2. 未来设计方向

未来，我们的设计方向将集中在增强现有功能的稳定性、提高系统性能、扩展操作系统功能以及加强安全性等方面。首先，文件系统的稳定性和性能是我们未来工作的重点。当前，我们的文件系统已经能够处理文件的创建、删除、读写等基本操作，但在极端情况下仍然可能出现错误，因此，使文件系统在所有情况下都能稳定运行、减少错误和异常是我们的首要任务。

同时，我们还计划优化文件系统的性能，特别是在磁盘访问和目录项缓存管理方面。通过改进缓存管理策略，减少不必要的磁盘操作，提升文件系统的响应

速度和效率。

总的来说，我们团队将着重于优化现有功能、提高系统性能，希望能够实现一个更高效、更稳定的操作系统。这些工作不仅能提升操作系统的实际应用能力，也将极大地提高我们对操作系统设计与实现的理解和实践水平。

参考文献:

- [1] 合肥工业大学. oskernel2024-ywd[EB/OL]. <https://gitlab.eduxiji.net/T202410359992654/oskernel2024-ywd/-/tree/main>. (访问日期: 2024-12-22).
- [2] 能 run 就行-2958[EB/OL]. <https://gitlab.eduxiji.net/educg-group-14238-914330/853476998-2958>. (访问日期: 2024-12-22).
- [3] HUST-OS. xv6-k210: Port XV6 to K210 board[EB/OL]. <https://github.com/HUST-OS/xv6-k210>. (访问日期: 2024-12-22).
- [4] xv6-中文手册[EB/OL]. https://gitcode.com/gh_mirrors/xv/xv6-book-2020-Chinese/overview?utm_source=replace_article_gitcode&index=top&type=card&&isLogin=1. (访问日期: 2024-12-22).
- [5] 专栏文章. 知乎专栏[EB/OL]. <https://zhuanlan.zhihu.com/p/646876948>. (访问日期: 2024-12-22).
- [6] Roger_Spencer. CSDN 博客[EB/OL]. https://blog.csdn.net/Roger_Spencer/article/details/135417466. (访问日期: 2024-12-22).
- [7] 张雷. CSDN 博客[EB/OL]. https://blog.csdn.net/zhanglei8893/article/details/6100815?spm=1001.2101.3001.6650.4&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-4-6100815-blog-135417466.235%5Ev43%5Epc_blog_bottom_relevance_base1&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-4-6100815-blog-135417466.235%5Ev43%5Epc_blog_bottom_relevance_base1&utm_relevant_index=8. (访问日期: 2024-12-22).
- [8] MIT. xv6 Operating System Source Code [EB/OL]. <https://github.com/mit-pdos/xv6>. (访问日期: 2024-12-22).
- [9] 邹宇. xv6: An introduction to the UNIX-like operating system (for learning purposes) [J]. 计算机科学与工程, 2020, 43(3): 127-136.
- [10] 张宇, 王丽. xv6 操作系统介绍与源代码分析 [J]. 操作系统技术, 2019, 27(1): 23-30.
- [11] 赵勇. xv6 操作系统实现与实践 [M]. 北京: 高等教育出版社, 2020.
- [12] 冯·特伦. xv6: A Simple, High-Level Operating System [M]. 北京: 电子工业出版社, 2015.