

【开发日志】操作系统大赛原理赛道开发日志

一、基本介绍

这是2024年全国大学生计算机系统能力大赛-操作系统设计赛(华东区域赛)-OS原理比赛的开发日志，本项目基于xv6-riscv开发

二、开发日志

2024/11/20

小组成员均已结束前置学习，克隆项目，配置工作区以及一些相关操作系统知识的学习

2024/11/23

基本了解riscv-xv6架构，进行初步更改。并且尝试添加第一个系统调用print_hello，基本了解添加系统调用的相关步骤以及原理，开始详细了解各个部分的设计细节并且结合linux的相关调用的思想来获取灵感。

2024/11/24

小组成员开会确定相关规范，并且配置好相关的代码拉取提交配置以及代码修改注释等规范。

2024/11/27

各个成员根据这几天的详细学习讲一下大致的目标以及想要进行开发的模块。

2024/11/28

添加系统调用 `sys_time`，返回系统启动以来的时钟滴答数

2024/11/30

添加功能 `ctrl+c` 用于杀死正在运行的进程

尝试实现进程优先级调度，修改调度器函数，初步判断成功实现了优先级调度，但是xv6的开机变得非常缓慢.....平均时间在十二秒左右，单个任务调度也比原先的调度任务慢。

2024/12/7

添加 `sys_printprocstate` 系统调用用于打印指定进程ID的详细信息或在未找到进程时报告错误。

添加 `sys_getppid` 系统调用用于获取指定进程的父进程ID

添加 `sys_printcpu` 系统调用用于打印指定进程的CPU寄存器状态信息。

添加 `sys_printmem` 系统调用用于打印指定进程的内存信息，包括虚拟内存大小和页表地址。如果找不到指定的进程，它会返回错误。

添加相关readme更新信息

2024/12/11

添加 `sys_ps` 系统调用用于显示系统中所有活动的进程及其状态和名称。

添加 `sys_setpriority` 允许改变指定进程的优先级。

添加 `sys_getpriority` 打印出所有进程的优先级信息。

添加 `sys_print_pcb_io` 显示特定进程的I/O操作计数。

2024/12/13

添加 `sys_printsiginfo` 系统调用用于打印指定进程的信号和中断信息，包括待处理的信号、已处理的信号以及各个中断的计数。如果找不到指定的进程，它会返回错误。

开会再次讨论一下进程和相关任务与困难，以及安排相关文档和展示方面的内容。

2024/12/14

添加 `sys_sem_p` 用于请求资源进入临界区。

添加 `sys_sem_v` 用于释放资源离开临界区。

添加 `sys_sh_var_read` 系统调用用于读取 `sh_var_for_sem_demo` 的共享变量。

添加 `sys_sh_var_write` 系统调用用于写入 `sh_var_for_sem_demo` 的共享变量。

2024/12/19

成功解决了之前出现的因为加入优先级调度机制而导致的系统卡顿问题。

添加 `sys_create_shared_memory` 系统调用用于创建共享内存段。

添加 `sys_attach_shared_memory` 系统调用给进程附加共享内存段。

添加 `sys_detach_shared_memory` 系统调用将进程从共享内存段分离。

添加 `sys_delete_shared_memory` 系统调用用于删除共享内存段。

2024/12/21

开始编写文档以及相关规范内容

添加 `sys_chmod` 对指定文件权限进行修改

添加 `sys_adduser` 和 `sys_login` 和 `sys_us` 进行用户相关操作

添加 `sys_getlogin` 用于用户态获取登陆信息

2024/12/24

完成编写文档

三、项目功能阐述

1.简单系统调用

(1) 获取系统时间： `SYS_time`

通过返回系统的 `ticks` 使得用户可以获取系统时间

(2) 打印进程状态： `SYS_printprocstate`

通过接受一个传入的参数——进程的 `pid`，返回这个进程的名字，状态，父进程pid，进程占用内存大小等信息

(3) 获取父进程的 `pid`： `SYS_getppid`

通过接受一个传入的参数——进程的 `pid`，返回父进程的 `pid` 和 `name`

(4) 获取进程CPU信息： `SYS_printcpu`

通过接受一个传入的参数——进程的 `pid`，返回内核模式下的页表地址、栈指针、内核陷阱/中断向量、异常发生时的程序计数器值、硬件线程ID、地址寄存器、栈指针、全局指针、线程指针、临时寄存器、保存寄存器、函数参数和返回值寄存器、额外的保存寄存器、额外的临时寄存器等信息

(5) 获取进程内存信息： `SYS_printmem`

通过接受一个传入的参数——进程的 `pid`，返回进程ID和名称，虚拟内存大小（以字节为单位），页表的地址等信息

(6) 打印当前系统进程信息： `SYS_ps`

显示系统中存在的进程名和pid及其状态

(7) 获取进程I/O信息： `SYS_print_pcb_io`

通过接受一个传入的参数——进程的 `pid`，返回进程ID，读取操作的计数（`io_read_count`），写入操作的计数（`io_write_count`）等信息

（8）获取进程信号和中断信息： `SYS_printsiginfo`

通过接受一个传入的参数——进程的 `pid`，返回待处理的信号掩码（`pending_signals`），已处理信号掩码（`handled_signals`），中断计数信息，对于每个非零的中断计数，打印中断号和发生次数

（9）更改文件名： `SYS_rename`

接收传递两个字符串参数：`oldpath`：要重命名的文件或目录的当前路径。`newpath`：要重命名为的新路径。然后分别解析并将路径分离为目录部分和文件名部分。确定旧文件所在的目录 `olddir` 并锁定避免修改，查找并锁定目标文件的inode。如果 `olddir` 和 `newdir` 不同且新路径并不存在，则锁定新文件所在的目录 `newdir`

删除旧的目录项 `oldname`，然后在新目录 `newdir` 中添加新的目录项 `newname`，指向相同的inode `ip`。解锁并释放所有涉及的inode并结束文件系统操作事务

2.小功能

在控制台中添加了 Ctrl+C 操作，以便杀死当前正在运行的进程。

原调度器在获取进程信息时存在问题，溯源后发现是因为 CPU 在调度进程前会将其加入到 CPU 结构体的 `proc` 字段下，调度结束后重新将 `proc` 置为 0。因此，如果不是在对应进程下运行，使用 `myproc` 函数很难获取运行中的进程号。

为此，我在调度器中引入了 `runproc` 变量，用于记录正在运行的进程。即使该进程的时间片结束，只要未到下一次调度，`runproc` 仍会记录该进程。通过这一改动，我们成功获取到了正在运行的进程，并对其调用 `kill` 函数以实现杀死进程的功能。

这个功能的实用性在于，相较于通过指定进程PID进行终止操作，它能够更加便捷地实现即使误将一个循环进程置于前台的情况下，仍然能够有效终止该进程，而无需重启系统。

3.功能系统调用

（1）增加信号量功能： `SYS_sem_v` `SYS_sem_p` `SYS_sem_create` `SYS_sem_free`

定义信号量结构体，

```
1 struct sem {
2     struct spinlock lock; // 内核自旋锁
3     int resource_count; // 资源计数
4     int resource; // 剩余资源数
5     int allocated; // 是否被分配使用：1 已分配，0 未分配
```

```
6  };
```

实例化为struct sem sems[SEM_MAX_NUM]信号量数组，并在系统启动时初始化信号量数组，设置为未分配状态。

`SYS_sem_create`：进程申请信号量，从数组中分配一个未分配的信号量。并打印分配的日志信息，函数接收参数n，n代表可用的资源计数

`SYS_sem_free`：进程释放一个指定 ID 的信号量。判断剩余资源数是否等于总的资源计数，如果是则设置该信号量为未分配状态，并打印释放的日志信息

`SYS_sem_p` `SYS_sem_v`：信号量的p/v操作，分别对信号量包含的资源数减一和加一。p操作需判断当前是否还有可用的资源，无则进入睡眠状态等待其他进程释放资源。v操作唤醒一个因为缺少资源而睡眠的进程（使用新定义的wakeup1p实现）

上面所有的操作都在自旋锁保护下进行，以保证并发安全

(2) 增加进程优先级： `SYS_setpriority` `SYS_getpriority`

在进程结构体中引入静态优先级 `staticpriority` 和动态优先级 `dynamicpriority`

创建进程时默认 `staticpriority` 和 `dynamicpriority` 为1

`SYS_setpriority`：接受两个传入的参数，分别为进程的 `pid` 以及要设置的进程优先级大小 `priority`，调用后将进程的 `staticpriority` 和 `dynamicpriority` 都设置为 `priority`

`SYS_getpriority`：调用后输出所有进程的 `pid` `name` `staticpriority` `dynamicpriority`

更改调度器使得其能根据优先级进行调度，具体代码如下：

旧代码（存在问题）：

```
1 void
2 scheduler(void)
3 {
4     struct proc *p;
5     struct cpu *c = mycpu();
6
7     c->proc = 0;
8     for(;;){
9         // The most recent process to run may have had interrupts
10        // turned off; enable them to avoid a deadlock if all
11        // processes are waiting.
12        intr_on();
13
14        int found = 0;
```

```

15     for(p = proc; p < &proc[NPROC]; p++) {
16         acquire(&p->lock);
17         if(p->state == RUNNABLE) {
18             // Switch to chosen process. It is the process's job
19             // to release its lock and then reacquire it
20             // before jumping back to us.
21             p->state = RUNNING;
22             p->dynamicpriority--;
23             if (p->dynamicpriority == 0) {
24                 p->dynamicpriority = p->staticpriority;
25                 finished = 1;
26             }
27             c->proc = p;
28             runproc = p;
29             swtch(&c->context, &p->context);
30
31             // Process is done running for now.
32             // It should have changed its p->state before coming back.
33             c->proc = 0;
34             found = 1;
35         }
36         release(&p->lock);
37     }
38     if(found == 0) {
39         // nothing to run; stop running on this core until an interrupt.
40         intr_on();
41         asm volatile("wfi");
42     }
43 }
44 }

```

实现方法是进程调度前获取其优先级，如果优先级不为0，则CPU运行完一个时间片后执行调度仍然运行这个进程，使得高优先级进程占据CPU时间长

上述代码在实际运行过程中使CPU出现了运行速率大幅降低的情况，通过分析可知：该调度运行过程仍遵循顺序调度，只是在高优先级下继续调度同一个进程而不调度下一个进程，进而导致当高优先级进程出现时，多个CPU在同时等待调度同一个进程，直至该进程动态优先级为0，造成了CPU的忙等现象。而且该调度没有使得优先级高的进程先执行，只是让优先级高的进程占据更长时间的CPU，不能算是一个基于优先级优先的调度，而是基于类时间片的调度

为此，我重新更改了调度逻辑，使得其能找到高优先级进程先运行，以及先判断进程是否被其他CPU调度，若被调度则忽略该进程，即使他是最高优先级的进程，转而去找次高优先级的进程，以下是具体代码的实现：

新代码：

```

1 void scheduler(void) {
2     struct proc *p;
3     struct cpu *c = mycpu();
4     c->proc = 0;
5     // 记录上次调度的位置
6     static struct proc *last_scheduled = 0;
7
8     for(;;) {
9         intr_on();
10        int found = 0;
11        struct proc *highest_priority = 0;
12        int max_priority = -1;
13
14        // 从上次调度的下一个位置开始查找
15        struct proc *start = last_scheduled ? last_scheduled + 1 : proc;
16        if(start >= &proc[NPROC]) start = proc;
17
18        for(p = start; p < &proc[NPROC]; p++) {
19            if(p->state == RUNNABLE && p->dynamicpriority > max_priority) {
20                highest_priority = p;
21                max_priority = p->dynamicpriority;
22            }
23        }
24
25        for(p = proc; p < start; p++) {
26            if(p->state == RUNNABLE && p->dynamicpriority > max_priority) {
27                highest_priority = p;
28                max_priority = p->dynamicpriority;
29            }
30        }
31
32        if(highest_priority) {
33            acquire(&highest_priority->lock);
34            if(highest_priority->state == RUNNABLE) {
35                highest_priority->state = RUNNING;
36                highest_priority->dynamicpriority--;
37                if(highest_priority->dynamicpriority <= 0)
38                    highest_priority->dynamicpriority = highest_priority-
>staticpriority;
39
40                c->proc = highest_priority;
41                runproc = highest_priority;
42                last_scheduled = highest_priority; // 更新上次调度位置
43                swtch(&c->context, &highest_priority->context);
44                c->proc = 0;
45                found = 1;
46        }

```

```

47         release(&highest_priority->lock);
48     }
49
50     if(found == 0) {
51         intr_on();
52         asm volatile("wfi");
53     }
54 }
55 }

```

引入 `highest_priority` 获取当前优先级最高的可运行进程，通过判断 `dynamicpriority` 的值和进程的 `state` 来查找优先级最高的进程，查找后运行该进程并让 `dynamicpriority` 的值减一，使得优先级较小的进程也有机会进入调度，当 `dynamicpriority` 小于等于0时，将 `dynamicpriority` 重新置为原优先级 `staticpriority`

引入 `last_scheduled` 记录上次进程调度的位置，解决因优先级相同导致 `pid` 较大的进程一直得不到运行而产生的饥饿现象

(3) 实现进程间通信： `SYS_create_shared_memory`

`SYS_attach_shared_memory` `SYS_detach_shared_memory`

`SYS_delete_shared_memory`

声明共享内存区结构体：

```

1 struct shm_segment {
2     uint64 *addr;           // 共享内存段地址
3     uint64 size;           // 大小
4     int ref_count;         // 引用计数
5     int valid;             // 是否有效
6 };

```

创建共享内存区结构体数组用于存放共享内存段信息： `struct shm_segment shm_table[MAX_SHM_COUNT]`

创建共享内存锁使得进程互斥访问： `struct spinlock shm_lock`

在进程结构体中引入结构体记录其拿到的共享内存段：

```

1 struct {
2     int shm_id;             // 共享内存ID
3     uint64 va;             // 映射的虚拟地址
4     int valid;             // 是否有效
5 } shm_mapping;

```

`sys_create_shared_memory` 代码如下：

```
1 int
2 sys_create_shared_memory(void) {
3
4     acquire(&shm_lock);
5
6     int i;
7     for(i = 0; i < MAX_SHM_COUNT; i++) {
8         if(!shm_table[i].valid)
9             break;
10    }
11    if(i == MAX_SHM_COUNT) {
12        release(&shm_lock);
13        return -1;
14    }
15
16    // 分配内存并初始化为0
17    uint64 *mem = kalloc();
18    if(mem == 0) {
19        release(&shm_lock);
20        return -1;
21    }
22
23    shm_table[i].addr = mem;
24    shm_table[i].size = PGSIZE;
25    shm_table[i].ref_count = 0;
26    shm_table[i].valid = 1;
27
28    release(&shm_lock);
29    return i;
30 }
```

函数实现过程：

- 查找 `shm_table` 是否存在空闲的共享内存区
- 若存在则分配一页内存
- 将信息记录在该共享内存区中
- 返回其共享内存区编号

sys_attach_shared_memory 代码如下:

```
1  int
2  sys_attach_shared_memory(void) {
3      int id;
4      argint(0, &id);
5      if(id < 0 || id >= MAX_SHM_COUNT)
6          return -1;
7
8      struct proc *p = myproc();
9
10     // 检查是否已有映射
11     if(p->shm_mapping.valid) {
12         return -1;
13     }
14
15     acquire(&shm_lock);
16
17     if(!shm_table[id].valid) {
18         release(&shm_lock);
19         return -1;
20     }
21
22     uint64 va = PGROUNDUP(p->sz);
23     if(mappages(p->pagetable, va, PGSIZE,
24                 (uint64)shm_table[id].addr,
25                 PTE_R|PTE_W|PTE_U) < 0) {
26         release(&shm_lock);
27         return -1;
28     }
29
30     // 记录映射
31     p->shm_mapping.shm_id = id;
32     p->shm_mapping.va = va;
33     p->shm_mapping.valid = 1;
34
35     shm_table[id].ref_count++;
36     p->sz = va + PGSIZE;
37
38     release(&shm_lock);
39     return va;
40 }
```

函数实现过程:

- 接受一个传入的参数为共享内存区编号
- 检查该编号是否已创建共享内存
- 若有则获取当前进程的内存段
- 调用 `PGROUNDUP` 函数使得其页面对齐
- 使用 `mmap` 函数在当前进程内存段的下一页建立虚拟映射与共享内存进行连接
- 在进程结构体中记录映射，共享内存区的引用计数加 1
- 返回映射地址

`sys_detach_shared_memory` 代码如下：

```
1 int
2 sys_detach_shared_memory(void) {
3     int id;
4     argint(0, &id);
5     if(id < 0 || id >= MAX_SHM_COUNT)
6         return -1;
7
8     struct proc *p = myproc();
9
10    acquire(&shm_lock); // 提前获取锁
11
12    // 检查是否是当前映射的内存
13    if(!p->shm_mapping.valid || p->shm_mapping.shm_id != id) {
14        release(&shm_lock);
15        return -1;
16    }
17
18    uint64 va = p->shm_mapping.va;
19
20    // 检查映射是否有效
21    pte_t *pte = walk(p->pagetable, va, 0);
22    if(pte == 0 || (*pte & PTE_V) == 0) {
23        release(&shm_lock);
24        return -1;
25    }
26
27    // 解除一页的映射
28    uvmunmap(p->pagetable, va, 1, 0);
29
30    // 更新状态
31    p->shm_mapping.valid = 0;
32    shm_table[id].ref_count--;
```

```
33
34     release(&shm_lock);
35     return 0;
36 }
```

函数实现过程：

- 接受一个传入的参数为共享内存区编号
- 检查编号是否合理
- 调用 `uvmunmap` 解除当前进程的虚拟映射
- 将进程的共享内存段有效值置为0，共享内存区的引用计数减 1

`sys_delete_shared_memory` 代码如下：

```
1  int
2  sys_delete_shared_memory(void) {
3      int id;
4      argint(0, &id);
5      if(id < 0 || id >= MAX_SHM_COUNT)
6          return -1;
7
8      acquire(&shm_lock);
9
10     if(!shm_table[id].valid) {
11         release(&shm_lock);
12         return -1;
13     }
14
15     // 只有当没有进程在使用这块内存时才能删除
16     if(shm_table[id].ref_count > 0) {
17         release(&shm_lock);
18         return -1;
19     }
20
21     // 释放内存
22     kfree(shm_table[id].addr);
23     shm_table[id].valid = 0;
24
25     release(&shm_lock);
26     return 0;
27 }
```

函数实现过程：

- 接受一个传入的参数为共享内存区编号
- 检查编号是否合理
- 调用 `kfree` 删除共享内存段
- 将该共享内存区的有效值置为0

(4) 用户管理功能： `SYS_adduser` `SYS_login` `SYS_us` `SYS_getlogin`

定义用户结构体

```
1 struct user {
2     char username[MAX_USERNAME]; // 用户名
3     char password[MAX_PASSWORD]; // 密码
4     int uid;                      // 用户ID
5     int gid;                      // 组ID
6 };
7 struct user users[MAXUSERNUM] = {
8     {"admin", "admin", 0, 0} // admin用户
9 };
```

建立用户表，并初始化一个管理员用户，拥有添加用户的权限。定义 `loginstate` 作为一个全局变量，表示当前登录用户的状态。如果其值为 `0`，则表示当前用户是管理员可以使用管理员权限。自行定义了 `kstrcmp` 和 `kstrncpy` 用于字符串比较和复制的内核函数，实现密码和用户名的正确匹配。添加用户时会检查用户名是否已存在，避免重复创建同一用户名的用户。

用户登陆使用login用户函数实现登陆操作，登陆成功则将当前用户状态设置为登陆用户。

```
1 int sys_login(void) {
2     char username[MAX_USERNAME];
3     char password[MAX_PASSWORD];
4     int user_index;
5
6     // 从用户栈中获取用户名和密码
7     if (argstr(0, username, sizeof(username)) < 0 || argstr(1, password,
8         sizeof(password)) < 0) {
9         return -1; // 参数错误
10    }
11
12    // 验证用户名和密码
13    user_index = authenticate_user(username, password);
14    if (user_index != -1) {
15        // 登录成功，设置登录状态并返回用户的 ID
16    }
```

```

15     printf("login success\n");
16     //loginstate = user_index;
17     kstrncpy(loginstate.username, username, MAX_USERNAME);
18     loginstate.uid=user_index;
19     //printf("%c", loginstate[0]);
20     return user_index;
21 } else {
22     // 登录失败
23     return -1;
24 }
25 }

```

使用 `sys_us` 打印当前系统中存在的用户表信息。 `sys_getlogin` 返回当前登陆信息给用户态，让 sh 能在 \$ 前打印出当前的用户名。

(5) 文件权限

在 `dinode` 结构体中添加文件权限字段和所属用户字段，为了确保满足 xv6 文件系统要求，即块大小 `BSIZE` 能被 `dinode` 结构体的大小整除，需要设置填充字段，为后续可能添加的参数保留空间。

```

1 struct dinode {
2     .....
3     short mode; //文件权限;
4     short user; //用户
5     char pad[58]; // 填充字段，让结构体大小对齐到原先合适的倍数
6 };
7

```

在 `mkfs` 中修改 `ialloc` 函数，对 xv6 创建的文件映像的相关字段进行赋值

```
din.mode = xshort(0755);  din.user = xshort(-1);
```

在 `kernal` 中，也需要修改 `fs.c` 中的 `ialloc` 函数，对用户生成的文件进行赋值，权限均设置为默认的 0755，并使用自定义函数 `getlogin` 得到当前登陆用户的信息，对 `dip->user` 赋值

```
dip->user = getlogin();
```

同时修改 `iupdate` 函数，`ilock` 函数，确保在将内存中的 `inode` 结构体更新到磁盘中和将磁盘中的 `inode` 数据拷贝到内存中时文件的权限信息和所属用户信息不变。在 `stat` 结构体中也要添加相关字段，并修改 `stati` 函数，确保用户空间与内核空间之间的数据传输。

为了实现文件权限的合理控制，修改了 `fileread`，在读文件前添加了如下的判断，`checkuser` 函数判断当前登陆用户是否是文件所属用户或者是否是同组用户，`check_permission` 判断是否有权限访问

```
1     int j=checkuser(f->ip->user);
```

```

2     int o =check_permission(f->ip->mode,0,j);
3     if(f->ip->user==-1) goto a; //如果该文件是xv6自动生成的文件映像文件，则所有用户都
    有读取权限
4     if(o==0)
5     {
6
7         iunlock(f->ip);
8         return -1;
9     }
10    a:

```

在针对写操作的控制中，修改了sys_open，在文件进行写或者截断时判断是否是有权合法操作，避免虽然无权限修改，但因为已经启动文件的截断而导致文件被置空

```

1     int j=checkuser(ip->user);
2     int o =check_permission(ip->mode,1,j);
3     // 权限检查：如果需要写权限，检查是否有写权限
4     if((omode & O_WRONLY) || (omode & O_RDWR)){
5         if(o== 0){ // 检查写权限
6             iunlockput(ip);
7             end_op();
8             return -1;
9         }
10    }
11
12    // 检查是否有权限进行截断
13    if((omode & O_TRUNC) && ip->type == T_FILE){
14        if(o == 0){ // 如果要截断文件，也需要写权限
15            iunlockput(ip);
16            end_op();
17            return -1;
18        }
19    }

```

添加系统调用 chmod对指定文件的权限进行修改，并编写调用它的用户态函数，

```

1 uint64
2 sys_chmod(void)
3 {
4     char pathname[MAXPATH];
5     int mode;
6     struct inode *ip; argint(1,&mode);
7     if(argstr(0, pathname, MAXPATH) < 0 || mode<0)

```

```

8  {
9      return -1;
10 }
11 begin_op();
12 if((ip=namei(pathname))==0)
13 {
14     end_op();
15     return -1;
16 }
17
18 ilock(ip);
19 ip->mode =mode;
20 iupdate(ip);
21 iunlock(ip);
22
23 end_op();
24
25 return 0;
26
27 }

```

参考linux改写ls代码，让它能输出文件的所属用户和权限信息

```

1  void mode_to_string(int mode, char *str) {
2      str[0] = (mode & (1 << 8)) ? 'r' : '-'; // 用户读取权限
3      str[1] = (mode & (1 << 7)) ? 'w' : '-'; // 用户写入权限
4      str[2] = (mode & (1 << 6)) ? 'x' : '-'; // 用户执行权限
5      str[3] = (mode & (1 << 5)) ? 'r' : '-'; // 组读取权限
6      str[4] = (mode & (1 << 4)) ? 'w' : '-'; // 组写入权限
7      str[5] = (mode & (1 << 3)) ? 'x' : '-'; // 组执行权限
8      str[6] = (mode & (1 << 2)) ? 'r' : '-'; // 其他用户读取权限
9      str[7] = (mode & (1 << 1)) ? 'w' : '-'; // 其他用户写入权限
10     str[8] = (mode & (1 << 0)) ? 'x' : '-'; // 其他用户执行权限
11     str[9] = '\\0'; // 结束符
12 }
13
14 void
15 ls(char *path)
16 {
17     .....
18     mode_to_string(st.mode, mode_str);
19     printf("%d %s %s %d %d %d\\n",st.user,mode_str,fmtname(path), st.type,
20         st.ino, (int) st.size);
21     .....
22 }

```