

开发日志

小组成员：刘昊鹏 陈旭 王江涛

任务0： 前导

2022-06-22

配置环境

- 首先在 GitHub fork [xv6-k210](#) 项目；
- 安装环境依赖

```
sudo apt update && sudo apt install gcc-riscv64-unknown-elf 安装64位 RISC-V 的编译器。
```

```
sudo apt install qemu-system-misc , 安装 RISC-V 的 QEMU 模拟器
```

```
sudo apt install python3 , 安装python3, 因为我们的测试脚本是用 Python 写的;
```

```
sudo apt install dosfstools , 安装 mkfs.vfat 工具;
```

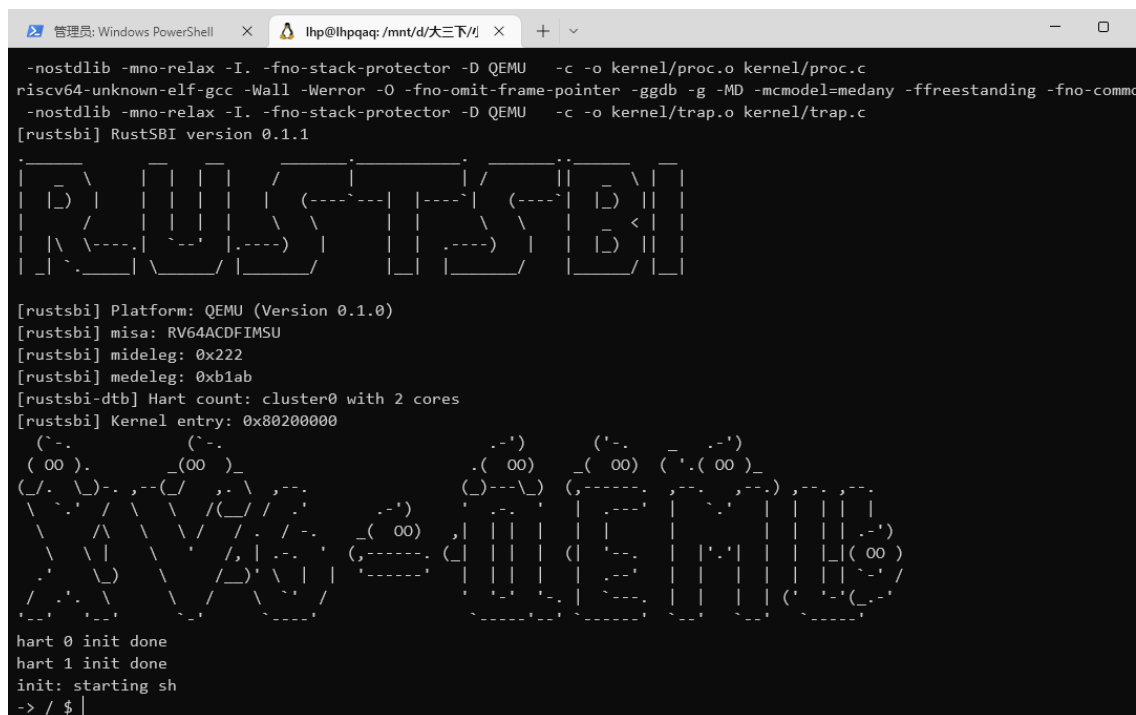
- 运行

```
make fs 生成一个 FAT32 的文件系统镜像, 并将它保存在 fs.img ;
```

修改makefile第一行, 运行平台为qemu

```
#platform      := k210
platform      := qemu
```

`make run` , 在 QEMU 上运行 xv6-k210:



```
管理员: Windows PowerShell  x  lhp@lhpqag: /mnt/d/大三下/  x  +  v
-nostdlib -mno-relax -I. -fno-stack-protector -D QEMU -c -o kernel/proc.o kernel/proc.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -g -MD -mmodel=medany -ffreestanding -fno-comm
-nostdlib -mno-relax -I. -fno-stack-protector -D QEMU -c -o kernel/trap.o kernel/trap.c
[rustsbi] RustSBI version 0.1.1

[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 2 cores
[rustsbi] Kernel entry: 0x80200000

hart 0 init done
hart 1 init done
init: starting sh
-> / $ |
```

任务1： 实现进程相关的系统调用

2022-06-22 上午

阅读代码和xv6的相关资料

1. xv6 手册: <https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf> , 以及其中文译本: <https://th0ar.gitbooks.io/xv6-chinese/content/>
2. build a OS (关于 xv6 的笔记) : https://xiayingp.gitbook.io/build_a_os/

搜索在xv6中添加新的系统调用的方法

2022-06-22 下午

o 在xv6-user目录下

- 在user.h文件中, 添加新系统调用封装后的函数声明, 假设其函数名为 `getppid`。

- ```
int getppid(void);
```

- 在usys.pl文件末尾, 添加如下行:

- ```
entry("getppid");
```

o 在kernel目录下

- 在include/sysnum.h文件中, 添加新系统调用号的宏定义:

- ```
#define SYS_getppid 27
```

- 其中, 27为新的合法系统调用号

- 在syscall.c文件中, 添加功能函数的声明, 并更新系统调用表:

- ```
extern uint64 sys_getppid(void);

static uint64 (*syscalls[])(void) = {
    .....
    [SYS_getppid]    sys_getppid,
};

static char *sysnames[] = {
    .....
    [SYS_getppid]    "getppid",
};
```

- 在sysproc.c文件中, 实现 `sys_getppid` 函数的功能如下:

- ```
uint64 sys_getppid(void)
{
 return myproc()->parent->pid;
}
```

o 在xv6-user目录下

- 创建getppid.c文件, 编写系统调用 `getppid` 的测试函数 `test_getppid` 如下:

```
int test_getppid()
{
 //TEST_START(__func__);
 int pid = getppid();
 //assert(pid >= 0);
 printf("getppid success.\nppid = %d\n", pid);
 //TEST_END(__func__);
 return 0;
}
```

- 在根目录下
  - 在Makefile文件中加一行如下：

```
UPROGS=\
 $U/_init\
 $U/_sh\
 $U/_cat\
 ...
 $U/_getppid\
```

运行结果

```
init: starting sh
-> / $ getppid
hello
getppid success.
ppid = 2
```

2022-06-23

## 任务2：添加信号

系统调用：alarm

2022-06-24

首先要在进程结构体里增加条目，alarm\_flag用以代指该进程被alarm信号标记，alarm\_tick和alarm\_para分别代表当前运行时间与限定运行时间



在sysproc.c中完成sys\_alarm，在用户态调用alarm函数时，使用argint读取参数，修改当前进程的各个变量

```

184 + uint64
185 + sys_alarm(void){
186 + int second;
187 + if(argint(0, &second) < 0) {
188 + return -1;
189 + }
190 +
191 + myproc()->signal=SIGALARM;
192 +
193 + myproc()->alarm_flag=1;
194 + myproc()->alarm_para=second*5;
195 + myproc()->alarm_tick=0;
196 + return 0;
197 + }

```

增加signal.c文件，signal\_handle函数用以处理当前进程捕获的信号。如检测到当前信号为SIGALARM(在signal.h中定义的宏)，则将alarm标记位赋值为1

```

... ... @@ -0,0 +1,9 @@
1 + #define __module_name__ "signal"
2 +
3 + #include "include/signal.h"
4 +
5 + void signal_handle(){
6 + if(myproc()->signal==SIGALARM){
7 + myproc()->alarm_flag=1;
8 + }
9 + }

```

在timer.c的timer\_tick函数中增加以下内容，代表当前进程如果被标记为alarm\_flag，那么就累加alarm\_tick，直到和alarm\_para相等，kill此进程

```

... ... @@ -41,6 +41,14 @@ set_next_timeout() {
41 41 void timer_tick() {
42 42 acquire(&tickslock);
43 43 ticks++;
44 44 if(myproc()){
45 45 if(myproc()->alarm_flag){
46 46 if(myproc()->alarm_tick==myproc()->alarm_para){
47 47 kill(myproc()->pid);
48 48 }
49 49 myproc()->alarm_tick++;
50 50 }
51 51 }
44 52 + wakeup(&ticks);
45 53 release(&tickslock);
46 54 set_next_timeout();

```

## 系统调用：pause

2022-06-24

当用户态调用pause函数时，直接使该进程sleep

```

... ... @@ -195,3 +195,13 @@ sys_alarm(void){
195 195 myproc()->alarm_tick=0;
196 196 return 0;
197 197 }
198 +
199 + uint64
200 + sys_pause(void){
201 201 acquire(&tickslock);
202 202 while(myproc()->killed == 0){
203 203 sleep(&ticks, &tickslock);
204 204 }
205 205 release(&tickslock);
206 206 return 0;
207 207 }

```

## 系统调用：signal 第一步

2022-06-27

首先需要在signal.h中扩展信号，增加SIG\_IGN和SIG\_DFL

```
kernel/include/signal.h
...
1 1 #ifndef __SIGNAL_H
2 2 #define __SIGNAL_H
3 3
4 4 - #include "types.h"
5 5 - #include "trap.h"
6 6 - #include "proc.h"
7 7 - #include "signal.h"
8 8 - #include "types.h"
9 9 - #include "param.h"
10 10 - #include "memlayout.h"
11 11 - #include "riscv.h"
12 12 - #include "spinlock.h"
13 13 -
14 14 -
15 15 4 #define SIGALRM 0
16 16 5 + #define SIG_IGN 1
17 17 6 + #define SIG_DFL 2
18 18 7
19 19 8 void signal_handle();
20 20 9
```

proc.h中增加新条目，表示当前进程接收到的信号之后的操作

```
kernel/include/proc.h
...
8 8 #include "file.h"
9 9 #include "fat32.h"
10 10 #include "trap.h"
11 11 + #include "signal.h"
12 12
13 13 // Saved registers for kernel context switches.
14 14 struct context {
15 15
16 16 @@ -83,6 +84,8 @@ struct proc {
17 17
18 18 83 84 uint64 alarm_tick; //当前alarm信号标记后运行了多少个tick
19 19 84 85 uint64 alarm_para; //alarm函数参数，表示alarm信号需要运行多少个tick后kill
20 20 85 86
21 21 87 + int signal_action; //signal函数的第二个参数
22 22 88 +
23 23 86 89 };
24 24 87 90
25 25 88 91 void reg_info(void);
```

与此同时扩展signal\_handle函数，如果当前行为为SIG\_DFL就立即kill(相当于调用alarm(0))，如果为SIG\_IGN就忽略，否则视为调用alarm

```
void sighandle(void)
{
 struct proc *p = myproc();
 int signum = p->killed;
 int i = 0;
 for(i=0; i<2; i++){
 if(signum == p->sigact[i].sig){
 if(p->sigact[i].handler == SIG_DEF){
 exit(-1);
 }else if(p->sigact[i].handler == SIG_IGN){
 p->killed = 0;
 break;
 }else{ //step2
 }
 }
 }
}
```

```
}
```

2022-06-28

在proc.h文件的proc结构体中添加成员变量如下，并在proc.c中进程初始化时进行初始化。

```
//signal, 本实验只要求实现两种信号及其处理，因此声明数组大小为2
struct sigaction sigact[2];
```

```
void procinit(){
 ...
 p->sigact[0].sig = SIGALRM;
 p->sigact[0].handler = SIG_DEF;
 p->sigact[1].sig = SIGINT;
 p->sigact[1].handler = SIG_DEF;
 ...
}
```

在sysproc.c文件中，实现 sys\_signal 函数的功能如下

```
uint64 sys_signal(void)
{
 uint64 sig;
 func handler;
 if (argaddr(0, &sig) < 0 || argaddr(1, (uint64*)&handler) < 0) {
 return -1;
 }
 //printf("signal:%d,%d\n",sig,handler);
 struct proc* p = myproc();
 int i = 0;
 for(i=0;i<2;i++){
 if(sig == p->sigact[i].sig){
 p->sigact[i].handler = handler;
 break;
 }
 }
 return 0;
}
```

在proc.c文件的usertrap函数中修改对信号的处理如下

```
void usertrap(void)
{
 ...
 if(r_scause() == 8){
 if(p->killed == SIGTERM)
 exit(-1);
 ...
 }
 else if((which_dev = devintr()) != 0){
 ...
 }
 else {
 ...
 p->killed = SIGTERM;
 }
}
```

```

 }
 if(p->killed){
 if(p->killed == SIGTERM)
 exit(-1);
 sighandle();
 }
 ...
}

```

在xv6-user目录下

创建alarmtest2.c文件，编写系统调用 alarm 的测试函数 test\_alarm 如下：

```

int test_alarm()
{
 printf("Alarm testing!\n");
 alarm (5);
 printf("waiting for alarm to go off\n");
 (void) signal (SIGALRM, SIG_DEF); //test1
 //(void) signal (SIGALRM, SIG_IGN); //test2
 pause(); //process suspended, waiting for signals to wake up
 printf("now reachable!\n");
 return 0;
}

```

在根目录下

在Makefile文件中如下编译目标：

```

UPROGS=\
 $U/_init\
 $U/_sh\
 $U/_cat\
 ...
 $U/_alarmtest2\

```

## 系统调用：signal 第二步

2022-06-29

在kernel目录下

在signal.c中修改 sighandle 函数的定义，修改 p->trapframe->epc，当进程返回用户态时首先执行信号处理函数，下一次陷入内核时通过 r\_sepc() 恢复到原来的epc。

```

void sighandle(void)
{
 struct proc *p = myproc();
 int signum = p->killed;
 int i = 0;
 for(i=0;i<2;i++){
 if(signum == p->sigact[i].sig){
 if(p->sigact[i].handler == SIG_DEF){
 exit(-1);
 }else if(p->sigact[i].handler == SIG_IGN){
 p->killed = 0;
 break;
 }
 }
 }
 p->trapframe->epc = r_sepc();
}

```

```

 }else{ //step2
 //当进程返回用户态时，从epc所指向的地址处开始执行，并通过a0传递处理函数的参
 数

 p->trapframe->epc = (uint64)p->sigact[i].handler;
 p->trapframe->a0 = p->killed;
 p->killed = 0;
 }
 }
}

```

在xv6-user目录下

创建alarmtest3.c文件，编写系统调用 alarm 的测试函数 test\_alarm 如下：

```

void ding (int sig)
{
 printf("[%d] Alarm has gone off\n",sig);
}
int test_alarm()
{
 printf("Alarm testing!\n");
 alarm (5);
 printf("waiting for alarm to go off\n");
 (void) signal (SIGALRM, ding);
 pause(); //process suspended, waiting for signals to wake up
 printf("now reachable!\n");
 return 0;
}

```

在根目录下

在Makefile文件中如下编译目标：

```

UPROGS=\
 $U/_init\
 $U/_sh\
 $U/_cat\
 ...
 $U/_alarmtest3\

```

## 系统调用：kill

2022-06-30

在kernel目录下

在proc.h文件中修改 kill(int) 函数声明，并在proc.c文件中修改该函数的定义。



```
//proc.h
int kill(int,int);
//proc.c
int kill(int pid, int sig)
{
 ...
 p->killed = sig;
 ...
}
```

在sysproc.c文件中修改系统调用 `sys_kill()` 函数的定义

```
uint64 sys_kill(void)
{
 int pid,sig;
 if(argint(0, &pid) < 0 || argint(0, &sig) < 0)
 return -1;
 return kill(pid,sig);
}
```

修改该目录下所有文件中的 `kill()` 调用

在xv6-user目录下

在user.h文件中，修改kill系统调用封装后的函数声明

```
int kill(int pid,int sig);
```

修改kill的测试函数以及该目录下其他文件中的 `kill()` 调用

```
#include "kernel/include/signal.h"
...
int main(int argc, char **argv)
{
 ...
 kill(atoi(argv[i]),SIGTERM);
 ...
}
```

## 按下CTRL-C 向前台进程发送 SIGINT 信号

2022-06-30

在kernel目录下

在proc.h文件中增加 `procint()` 函数声明，并在proc.c文件中增加该函数的定义。

如果是后台进程，sh程序在调用exec执行程序时会调用两次fork，通过子进程的子进程来执行程序，同时第一次的fork的子进程再fork第二个子进程后会直接退出，这是执行程序子进程交由操作系统来管理，因此当前子进程的父进程是操作系统，其ppid值是1。而前台进程由于是通过第一次fork的子进程，其ppid值为sh的pid值。据此可以在按下Ctrl-C时杀掉前台进程。

```
void procint(void)
{
 int flag = 0;
 struct proc *p;
```

```

for(p = proc; p < &proc[NPROC]; p++){
 if(p->pid > 2 && p->parent->pid == 2){
 if(p->state == RUNNING || p->state == RUNNABLE || p->state == SLEEPING){
 kill(p->pid,SIGINT);
 flag = 1;
 }
 }
}
if(!flag){
 printf("\n-> / $ ");
}
}

```

在console.c文件中修改 `consoleintr()` 函数，添加对硬件中断 Ctrl-C 的处理。

```

void consoleintr(int c)
{
 ...
 switch(c){
 case C('C'):
 procint();
 break;
 ...
 }
}

```

## 任务3：实现虚拟文件系统 `/proc`

2022-08-26

阅读任务三描述：

在 Linux 中，`/proc` 目录下有一系列不需要存储在磁盘上的文件，这些“虚拟”文件描述了系统的状态、配置以及进程等等。事实上，`/proc` 目录也确实不在磁盘存储，而是在每次被读取时动态生成其中内容。

在这项任务中，我们将基于 xv6 实现 `/proc` 文件系统。

### 虚拟目录列表

利用 `mkdir` 创建目录 `/proc`，当执行 `ls /proc` 时，将会出现一系列虚拟目录（并不真正存储在磁盘上）。每一个正在运行的进程都拥有自己的目录。

在 xv6 中，一个文件即为一个 `inode`，`inode` 中含有读写文件内容以及构成 `inode` 数据的函数。这些 `inode` 函数将会调用硬件接口，但事实上 `/proc` 在磁盘上并不存在。

所以我们需要修改文件系统相关的代码。更具体地说，现在 `struct inode` 中包含一个叫做 `struct inode_functions` 的指针，其中包含指向读写 `inode` 的函数指针。这里将提供一个 `/proc` `inode` 的实现。

```

struct inode_functions {
 void (*ipopulate)(struct inode*); // fill in inode details
 void (*iupdate)(struct inode*); // write inode details
back to disk
 int (*readi)(struct inode*, char*, uint, uint); // read from file contents
 int (*writei)(struct inode*, char*, uint, uint); // write to file contents
};

struct inode {
 /// ... other fields ...
 struct inode_functions *i_func; // NEW!
};

```

首先修改 inode 中的 `i_func` 指针，以便读取 `/proc` 时函数能够被调用。从 `ls.c` 可以观察到如何获取目录列表，然后便可以写一个 `procfs_readi` 函数。`namei` 函数根据给定的文件路径（利用参数进行传递）返回对应的 `struct inode*`。

需要确保 `ls /proc` 显示正确的文件类型和大小。因此需要实现 `procfs_ipopulate`，注意此处的 `iget()`：对于“proc”文件使用不同的设备号来避免重复获取错误的 inode。

2022-08-27

查阅关于虚拟文件系统的资料，阅读理解xv6文件系统的代码。

2022-08-31

追溯mkdir的流程，定位到sys\_mkdir，添加新建/proc时的处理的，既 `createproc()` 函数

```

uint64
sys_mkdir(void)
{
 char path[FAT32_MAX_PATH];
 struct dirent *ep;
 char* proc = "/proc";
 char* proc1 = "proc";
 if(argstr(0, path, FAT32_MAX_PATH) < 0 || (ep = create(path, T_DIR, 0)) == 0)
 {
 return -1;
 }
 if (strncmp(proc, path, 5) == 0 || strncmp(proc1, path, 4)==0)
 {
 eunlock(ep);
 eput(ep);
 createproc(); //创建进程信息
 return 0;
 }
 eunlock(ep);
 eput(ep);
 return 0;
}

```

添加 `createproc()` 函数：

```

void createproc()
{
 struct dirent* ep = ename("/proc");
 elock(ep);
}

```

```

ep->e_func = &procfs_e_func;
struct proc* p;
struct dirent* tep;
char pdir[20];
for (p = proc; p < &proc[NPROC]; p++)
{
 itoa(p->pid, pdir);
 if (p->state != UNUSED)
 {
 tep = ealloc_inmemory(ep, pdir, ATTR_DIRECTORY); //pid dir
 ealloc_inmemory(tep, "stat", ATTR_ARCHIVE); // stat
 }
}
eunlock(ep);
}

```

添加 `ealloc_inmemory` 函数，既在内存上分配进程信息文件：

```

struct dirent *ealloc_inmemory(struct dirent *dp, char *name, int attr)
{
 if (!(dp->attribute & ATTR_DIRECTORY))
 {
 panic("ealloc not dir");
 }
 if (dp->valid != 1 || !(name = formatname(name)))
 {
 return NULL;
 }
 struct dirent *ep;
 uint off = 0;
 if ((ep = dirlookup(dp, name, &off)) != 0)
 {
 return ep;
 }
 ep = eget(dp, name);
 elock(ep);
 ep->attribute = attr;
 ep->file_size = 0;
 ep->first_clus = 0;
 ep->parent = edup(dp);
 ep->off = off;
 ep->clus_cnt = 0;
 ep->cur_clus = 0;
 ep->dirty = 0;
 strncpy(ep->filename, name, FAT32_MAX_FILENAME);
 ep->filename[FAT32_MAX_FILENAME] = '\0';
 if (attr == ATTR_DIRECTORY)
 {
 ep->attribute |= ATTR_DIRECTORY;
 }
 else
 {
 ep->attribute |= ATTR_ARCHIVE;
 }
 ep->valid = 1;
 eunlock(ep);
 return ep;
}

```

```
}
```

2022-08-31

## ls /proc

修改dirnext函数，在执行ls /proc或ls /proc/[pid]时，实时打印出当前的进程信息

```
int dirnext(struct file *f, uint64 addr)
{
 if(f->readable == 0 || !(f->ep->attribute & ATTR_DIRECTORY))
 return -1;
 struct dirent de;
 struct stat st;
 int count = 0;
 int ret;
 elock(f->ep);
 while ((ret = enext(f->ep, &de, f->off, &count)) == 0) { // skip empty entry
 f->off += count * 32;
 }
 eunlock(f->ep);
 if (ret != -1)
 {
 f->off += count * 32;
 estat(&de, &st);
 }
 else
 {
 if(strncmp(f->ep->filename, "proc", 4) == 0 || strncmp(f->ep->parent-
>filename, "proc", 4) == 0)
 {
 struct dirent* en;
 elock(f->ep);
 if((en = procfs_enext(f->ep)) != NULL)
 {
 f->ep->child_index++;
 }
 else
 {
 f->ep->child_index = 0;
 eunlock(f->ep);
 return 0;
 }
 eunlock(f->ep);
 estat(en,&st);
 }
 else
 {
 return 0;
 }
 }
 if(copyout2(addr, (char *)&st, sizeof(st)) < 0)
 return -1;
 return 1;
}
```

## cat /proc

在fat32.h中新增定义结构体dirent\_function，并在结构体dirent中增加成员变量e\_func，然后在file.c中修改系统中对eread的调用。

fat32.c中在文件系统初始化或分配dirent时初始化dirent的e\_func。使用 tmp->e\_func = &procfs\_e\_func; 语句在为/proc目录下的子目录或文件分配dirent时重定向e\_func。

```
73
74 // Check ELF header
75 if(eread(ep, 0, (uint64) &elf, 0, sizeof(elf)) != sizeof(elf))
76 if(ep->e_func->eread(ep, 0, (uint64) &elf, 0, sizeof(elf)) != sizeof(elf))
77 goto bad;
78 if(elf.magic != ELF_MAGIC)
79 goto bad;
80 if((pagetable = proc_pagetable(p)) == NULL)
81 goto bad;
82 // Load program into memory.
83 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
84 if(eread(ep, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
85 if(ep->e_func->eread(ep, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
86 goto bad;
87 if(ph.type != ELF_PROG_LOAD)
88 continue;
```

在fat32.c中实现procfs\_eread函数

```
int procfs_eread(struct dirent *entry, int user_dst, uint64 dst, uint off, uint
n)
{
 if (off > entry->file_size || off + n < off || (entry->attribute &
ATTR_DIRECTORY)) {
 return 0;
 }
 int pid = 0;
 int len = 0;
 pid = atoi(entry->parent->filename);
 if(pid > 0)
 {
 char buf[512];
 proc_read(pid,buf);
 len = strlen(buf);
 either_copyout(user_dst,dst,buf,len);
 }
 return len;
}
```

在proc.c中实现辅助函数proc\_read，获取到所读取的进程的信息，处理成特定的字符串，作为stat的内容

```
void proc_read(int pid, char* s)
{
 struct proc *p;
 for(p = proc; p < &proc[NPROC]; p++) {
 if(pid == p->pid)
 {
 break;
 }
 }
```

```

}
s[0] = '\0';
strcat(s,"pid\tcommand\t\tstate\t\tppid\tutime\tstime\tcutime\tcstime\tvsz\n");
;
char tmp[128];
itoa(p->pid,tmp);
strcat(s,tmp);
strcat(s,"\t");
strcat(s,p->name);
strcat(s,"\t\t");
if(p->state == UNUSED)
{
 strcat(s,"UNUSED\t\t");
}
else if(p->state == SLEEPING)
{
 strcat(s,"SLEEPING\t");
}
else if(p->state == RUNNABLE)
{
 strcat(s,"RUNNABLE\t");
}
else if(p->state == RUNNING)
{
 strcat(s,"RUNNING \t");
}
else
{
 strcat(s,"ZOMBIE\t\t");
}
if(p->pid == 1)
 p->parent->pid = 0;
itoa(p->parent->pid,tmp);
strcat(s,tmp);
strcat(s,"\t");
itoa(p->times.utime,tmp);
strcat(s,tmp);
strcat(s,"\t");
itoa(p->times.stime,tmp);
strcat(s,tmp);
strcat(s,"\t");
itoa(p->times.cutime,tmp);
strcat(s,tmp);
strcat(s,"\t");
itoa(p->times.cstime,tmp);
strcat(s,tmp);
strcat(s,"\t");
itoa(p->sz,tmp);
strcat(s,tmp);
strcat(s,"\n");
}

```

## 任务4：实现 `ps` 命令

通过增加系统调用，返回ps命令执行时系统中正在运行的进程信息。

在进程控制结构体中增加成员变量starttime，用于记录进程被创建并开始运行的系统时间

```
//proc.h -> struct proc
uint64 starttime;
```

进程被创建时，记录当前时间

```
//proc.c->forc
np->starttime = retime();
```

调用ps时，计算当前运行的时间

```
//acquire(&tickslock);
pi->etime = retime() - p->starttime;
//release(&tickslock);
```

实现proc\_ps用于读取指定pid的进程的信息到procinfo结构体

```
void proc_ps(int pid, struct procinfo* pi)
{
 struct proc *p;
 for(p = proc; p < &proc[NPROC]; p++)
 {
 if(pid == p->pid)
 {
 break;
 }
 }
 pi->pid = p->pid;
 pi->ppid = p->parent->pid;
 if(p->pid == 1)
 pi->ppid = 0;
 pi->command[0] = '\0';
 strcat(pi->command, p->name);
 if(p->state == SLEEPING)
 {
 pi->state = 'S';
 }
 else
 {
 pi->state = 'R';
 }
 uint64 maxt = p->u2stime;
 if(p->s2utime > maxt)
 maxt = p->s2utime;
 pi->times = p->times.stime + p->times.utime;
 pi->etime = retime() - p->starttime;
 pi->vsz = p->sz;
}
```

在sysproc.c文件中，实现sys\_procps函数系统调用，把procinfo中的信息给用户态

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



```

int pids[NPROC];
struct procinfo pinfo[NPROC];
int i;
uint64 len = 0;
int cnt = getPids(pids);
for(i = 0; i < cnt; i++)
{
 proc_ps(pids[i], &pinfo[i]);
 len += sizeof(pinfo[i]);
}
copyout2(addr, (char*)pinfo, len);
return cnt;
}

```

在xv6\_user目录下新建ps.c，用print\_usage函数打印ps命令的用法提示信息

使用set\_flag解析输入的命令行参数设置标记位，如果检测到参数有误，直接打印出错误参数提示信息并退出程序

实现parse\_arg函数解析命令行参数

2022-09-01 下午

基本完成所有任务，开始做PPT

2022-09-02

## Bonus：自动创建 /proc 目录并挂载 proc

Xv6执行的第一个进程为init，在init中添加创建mkdir的命令，即可实现自动挂载

```

char *proc = "/proc";
if(mkdir(proc)<0)
{
 printf("mkdir /proc failed\n");
}

```