第一周

第一天

根据实验指导任务0，完成实验环境配置 下载wsl，输入命令：

```
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

sudo apt install make git #安装一些实验相关软件

sudo apt install dosfstools #安装 mkfs.vfat 工具

使用github fork仓库地址https://github.com/abrasumente233/xv6-k210.git (https://github.com/abrasumente233/xv6-k210.git)

git clone https://github.com/[your_github_username]/xv6-k210.git (https://github.com/%5Byour_github_username%5D/xv6-k210.git) #方括号以及其中 内容为你自己的GitHub用户名

cd xv6-k210 #进入xv6-k210文件夹

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

make run platform=qemu #在 QEMU 上运行 xv6-k210
```

运行结果：

第二天

实现简单系统调用

1. 首先在ueys.pl文件中添加entry项(以下是本实验添加的所有entry项，以及所有要添加的函数等)

```
entry("fork");
entry("exit");
entry("wait");
entry("pipe");
entry("read");
entry("write");
entry("close");
entry("kill");
entry("exec");
entry("open");
entry("fstat");
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");
entry("test_proc");
entry("dev");
entry("readdir");
entry("getcwd");
entry("remove");
entry("trace");
entry("sysinfo");
entry("rename");
entry("getppid");
entry("times");
entry("getmem");
entry("alarm");
entry("signal");
entry("pause");
entry("ps");
```

2. 在user.h文件中添加相应的用户声明，来使得用户可以调用相应接口

```
// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int fd, const void *buf, int len);
int read(int fd, void *buf, int len);
int close(int fd);
int kill(int pid);
int exec(char*, char**);
int open(const char *filename, int mode);
int fstat(int fd, struct stat*);
int mkdir(const char *dirname);
int chdir(const char *dirname);
int dup(int fd);
int getpid(void);
char* sbrk(int size);
int sleep(int ticks);
int uptime(void);
int test_proc(int);
int dev(int, short, short);
int readdir(int fd, struct stat*);
int getcwd(char *buf);
int remove(char *filename);
int trace(int mask);
int sysinfo(struct sysinfo *);
int rename(char *old, char *new);
int getppid(void);
int times(struct tms * tm);
int getmem(void);
int alarm(int second);
void (*signal(int sig, void (*func)(int)))(int);
// ulib.c
int stat(const char*, struct stat*);
char* strcpy(char*, const char*);
char* strcat(char*, const char*);
void *memmove(void*, const void*, int);
char* strchr(const char*, char c);
int strcmp(const char*, const char*);
void fprintf(int, const char*, ...);
void printf(const char*, ...);
char* gets(char*, int max);
uint strlen(const char*);
void* memset(void*, int, uint);
void* malloc(uint);
void free(void*);
int atoi(const char*);
int memcmp(const void *, const void *, uint);
void *memcpy(void *, const void *, uint);
void pause(void);
void ps();
```

3. 在include/sysnum.h文件中，添加新系统调用号的宏定义：

```
// We predefined some system call numbers for you!
// You don't need to change this part.
#define SYS_getppid     27
#define SYS_times       28
#define SYS_getmem      29
#define SYS_alarm       30
#define SYS_signal      31
#define SYS_pause       32
#define SYS_ps          33
```

4. 在syscall.c文件中，添加功能函数的声明，并更新系统调用表

```
//声明部分
extern uint64 sys_getppid(void);
uint64 sys_times(void);
extern uint64 sys_getmem(void);
extern uint64 sys_alarm(void);
extern uint64 sys_signal(void);
extern uint64 sys_pause(void);
extern uint64 sys_ps(void);
static uint64 (*syscalls[])(void) = {
......
[SYS_getppid]      sys_getppid,
[SYS_times]      sys_times,
[SYS_getmem]      sys_getmem,
[SYS_alarm]      sys_alarm,
[SYS_signal]      sys_signal,
[SYS_pause]      sys_pause,
[SYS_ps]         sys_ps,
};
static char *sysnames[] = {
......
[SYS_getppid]      "getppid",
[SYS_times]      "times",
[SYS_getmem]      "getmem",
[SYS_alarm]      "alarm",
[SYS_signal]      "signal",
[SYS_pause]      "pause",
[SYS_ps]          "ps",
};
```

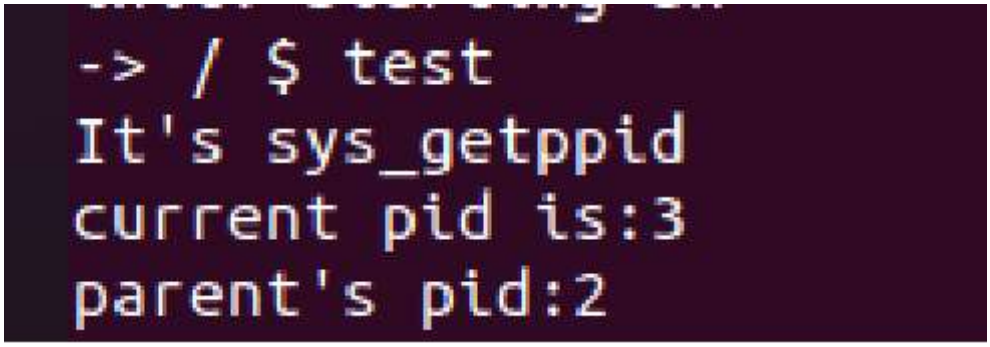
1. getppid

思路：从进程中获取父进程的pid，每个进程结构体都有一个pid成员变量，同时存在父进程指针

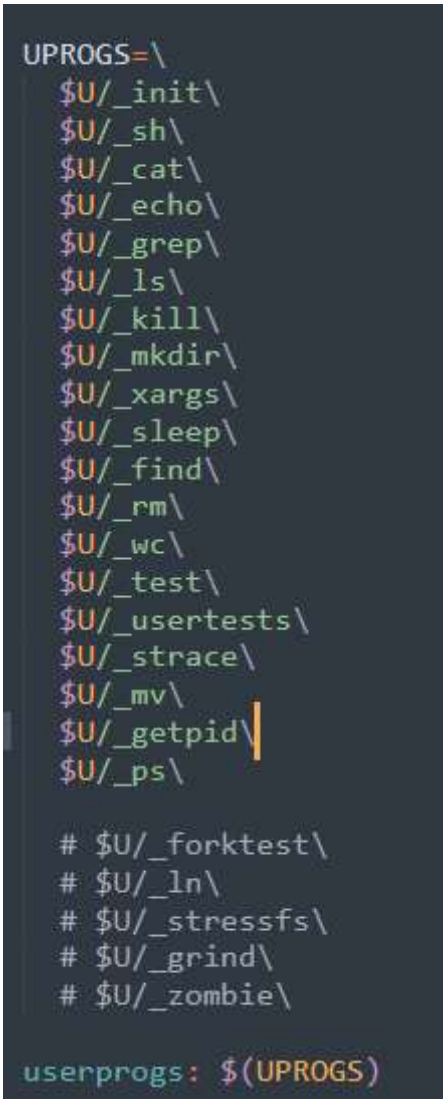
```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  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
```

```
uint64 sys_getppid(void)
{
  printf("It's sys_getppid\n");
  struct proc *p = myproc();  //获取当前进程
  struct proc *parents = p->parent;  //获取父进程
  printf("current pid is:%d\n",p->pid); //输出当前进程pid
  printf("parent's pid:%d\n",parents->pid); //输出父进程pid
  return parents->pid; //返回父进程pid
}
```

注释如图，运行结果：



用户态添加文件需要在Makefile中更改



注意最后 $U/_ps\ 和注释中间的一个空行必须保留，不然报错

第三天

2.times

第一步：proc.h添加相应成员变量（注释如图）,同时在procinit中初始化各成员变量

```
uint64 ikstmp;      // the last moment when entering kernel
uint64 okstmp;      // the last moment when leaving kernel
struct tms proc_tms;
```

```
struct tms {
  uint64 utime;   // user time
  uint64 stime;   // system time
  uint64 cutime;    // user time of children
  uint64 cstime;    // system time of children
};
```

```
// initialize the proc table at boot time.
void
procinit(void)
{
  struct proc *p;

  initlock(&pid_lock, "nextpid");
  for(p = proc; p < &proc[NPROC]; p++) {
      initlock(&p->lock, "proc");
      p->proc_tms.utime = 0;
      p->proc_tms.stime = 0;
      p->proc_tms.cutime = 0;
      p->proc_tms.cstime = 0;
```

第二步：记录用户进入内核的时间，和出内核的时间（在trap.c中的usertrap函数修改）

```
struct proc *p = myproc(); //获取当前进程
uint64 timestamp = r_time();//记录当前时间
p->ikstmp = timestamp; //赋值给那个进程相应成员变量
p->proc_tms.utime += timestamp - p->okstmp;
//当前时间减去上次出内核的时间就是进入内核前的用户时间
```

第三步：记录用户从内核返回用户态的时间，同时赋值给stime（在trap.c中的usertrapret)

```
void
usertrapret(void)
{
  struct proc *p = myproc();
  uint64 timestamp = r_time();
  p->okstmp = timestamp;//出内核的时间
  p->proc_tms.stime += timestamp - p->ikstmp;
  //当前时间减去之前进入内核的时间就是在内核的时间
```

在sche函数中进程会进行上下文切换，会放弃一次cpu，所以这个时候stime也需要处理

```
  p->proc_tms.stime += r_time() - p->ikstmp;


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

第四步：在proc.c中记录wait函数记录子进程的所有时间

```
//在wait中等待所有子进程，计算子进程的相应时间
p->proc_tms.cstime += np->proc_tms.stime + np->proc_tms.cstime;
p->proc_tms.cutime += np->proc_tms.utime + np->proc_tms.cutime;
```

第五步：在系统调用sys_times中进行赋值

```
uint64 sys_times(void) {
  uint64 tms;
  if (argaddr(0, &tms) < 0) {
    return -1;
  }
  struct tms* t = (struct tms*)tms;
  struct proc *p = myproc();
//  if (copyout2(tms, (char*)&(p->proc_tms), sizeof(p->proc_tms)) < 0) {
//    return -14;
//  }
  t->utime = p->proc_tms.utime;
  t->stime = p->proc_tms.stime;
  t->cutime = p->proc_tms.cutime;
  t->cstime = p->proc_tms.cstime;
  return readtime();
}
```

3.getmem

直接获取当前进程结构体成员变量sz，除1024得KB大小

```
uint64 sys_getmem(void)
{
  struct proc *p = myproc();
  return p->sz/1024;
}
```

第四天

添加信号

4.alarm

同样的操作先添加sys_alarm的系统调用

```
uint64
sys_alarm(void)
{
  int second;
  if(argint(0, &second) < 0) {
    return -1;
  }
 // myproc()->signum=SIGALARM;
  myproc()->alarm_tick=second*5;

  return 0;
}
```

通过多次实验，大概1s和tick之间的关系是5倍 首先内核初始化会初始化中断

```
void
main(unsigned long hartid, unsigned long dtb_pa)
{
  inithartid(hartid);

  if (hartid == 0) {
    consoleinit();
    printfinit();   // init a lock for printf
    print_logo();
    #ifdef DEBUG
    printf("hart %d enter main()...\n", hartid);
    #endif
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    timerinit();     // init a lock for timer
    trapinithart();  // install kernel trap vector, including interrupt handler
```

中断函数里面会包括设置一个时钟定时的中断

```
void
trapinithart(void)
{
  w_stvec((uint64)kernelvec);
  w_sstatus(r_sstatus() | SSTATUS_SIE);
  // enable supervisor-mode timer interrupts.
  w_sie(r_sie() | SIE_SEIE | SIE_SSIE | SIE_STIE);
  set_next_timeout();
  #ifdef DEBUG
  printf("trapinithart\n");
  #endif
}
```

每一次都会进行一个时钟计时，每次进入内核会进行一个判断是哪个类型，如果是时钟就会进入timer_tick函数

```
  }
  else if (0x8000000000000005L == scause) {
    timer_tick();
    return 2;
  }
```

我们在sys_alarm中已经对alarm进行计时，当计时为0的时候发送一个SIGALRM信号（还需要保证在此之前不会有意外情况，比如键盘的外部中断）

```
void timer_tick() {
    //printf("timer_tick------\n");
    acquire(&tickslock);
    ticks++;
    //printf("timer_tick\n");
    struct proc *p;

    for(p = proc; p < &proc[NPROC]; p++) {

      if(p->alarm_tick>0)
      {
        p->alarm_tick--;
       // printf("alarm_timer_tick减减\n");
        if(p->alarm_tick==0)
        {
         //   printf("alarm_timer_tick time out    proc pid:%d\n",p->pid);
            if(p->sigaction.sig_flags==0)//没有被signal处理过
                kill(p->pid,SIGALARM);
            else
                kill(p->pid,p->sigaction.sig_type);

        }
      }
    }


    wakeup(&ticks);
    release(&tickslock);
    set_next_timeout();
}
```

此外在每一次进入内核时候都需要进行一个信号处理（如果有信号）

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

信号处理函数中，需要判断对该信号是默认执行还是忽略

```
void sighandle(void)
{
    struct proc *p = myproc();
    printf("sighandle\n");
    printf("current process killed:%d\n",p->killed);
    printf("current process alarm_ticks:%d\n",p->alarm_tick);
    if(p->killed==SIGALARM&&p->alarm_tick==0)
    {
        printf("SIGALARM\n");
        if(p->sigaction.sig_action==SIG_DFL)
        {
            printf("SIG_DFL\n");
            kill(p->pid,SIGTERM );
        }
        else if(p->sigaction.sig_action==SIG_IGN)
        {
            printf("SIG_IGN\n");
            myproc()->killed=0;
        }
        else
        {
            printf("SIG_FUNC\n");
            myproc()->trapframe->a0=myproc()->killed;
            myproc()->trapframe->epc=myproc()->sigaction.p;
            myproc()->killed=0;
        }
```

设置好相应的处理即可完成alarm信号

第五天

5.添加pause

实现系统调用 void pause()。pause 暂停当前进程, 直到接收到一个信号。 当前进程的killed标志即进程收到信号的类型，不为0即是收到信号，如果为0则一直循环，直到收到信号。

```
171 uint64 sys_pause(void)
172 {
173   struct proc* p = myproc();
174   printf("pause: waiting for signal to wake up!\n");
175   // p->state = SLEEPING;
176   // scheduler();
177   acquire(&tickslock);
178   while(p->killed == 0){
179     sleep(&ticks, &tickslock);
180   }
181   release(&tickslock);
182   return 0;
183 }
```

第六天

6.signal

- signal()接受两个参数：sig和func，func指定接收信号sig时的处理函数，这个函数必须以一个int作为参数并且其返回类型为void。signal() 函数返回一个同类型的函数，这是sig信号的旧处理函数，或者以下两个特殊值中的其中一个：
- SIG_IGN：忽略此信号
- SIG_DFL：为默认操作，即终止进程。
- 如果第二个参数为一函数(非SIG_DFL或SIG_ING)，那么收到信号时就要利用此函数来处理。

```
uint64 sys_signal(void)
{
  int sig;
  uint64 addr;
  if (argint(0, &sig) < 0) {
    return -1;
  }
  if(argaddr(1,&addr)<0){
    return -1;
  }
  myproc()->sigaction.sig_flags=1;
  myproc()->sigaction.sig_action=(func)addr;
  if((func)addr==SIG_FUNC)
    myproc()->sigaction.p=addr;
  myproc()->sigaction.sig_type=sig;          //我应该等待alarm到达之后再更改
killed信号
  return 0;
}
```

7.kill

将 kill(int pid)扩展为kill(int pid, int sig)，使得kill()能够：向由 pid 指定的进程发送信号；发送任意信号，例如 SIGINT 和 SIGALARM

①在进程队列中找到指定进程。

②将进程的killed标志设置为传入的sig参数，进程处在睡眠状态应唤醒进程，使得进程可以对我们的信号做出反应。

```
int
kill(int pid)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->pid == pid){
      p->killed = 1;
      if(p->state == SLEEPING){
        // Wake process from sleep().
        p->state = RUNNABLE;
      }
      release(&p->lock);
      return 0;
    }
    release(&p->lock);
  }
  return -1;
}
```

第七天

8.CTRL+C

修改 console.c 文件中的 consoleintr() 函数，使得不管何时按下 CTRL-C ， 内核将向 sh 发送信号 SIGINT 。接着 sh 将执行其信号处理函数，具体为检查正在前台运行的进程并向其发送 SIGINT 。如果无任何进程在前台运行，那么只需要重新打印 shell 的命令提示符。

第一步：修改console.c 文件中的 consoleintr()函数，使得不管何时按下CTRL-C ， 内核将向sh发送号 SIGINT 。修改方法为：在consoleintr()函数的switch条件结构中增加case('C'),即按下Ctrl+C执行，执行inthandle函数。

```
case C('C'):inthandle();//或者用kill(myproc()->pid,SIGIN);
```

第二步：修改inthandle函数。接着sh将执行其信号处理函数，具体为检查正在前台运行的进程并向其发送 SIGINT。如果无任何进程在前台运行，那么只需要重新打印 shell 的命令提示符。

```
void inthandle(void)
{
    // printf("handle Ctrl C\n");
     struct proc* p;
     int runflag=0;
     for (p = proc; p < &proc[NPROC]; p++) {
         if (p->pid >2 && p->parent->pid==2) {
             if (p->state == RUNNING || p->state == SLEEPING || p->state == RUNNABLE)
             {
                 kill(p->pid, SIGINT);
                 runflag = 1;
             }
         }
     }
     if (!runflag) {
         printf("\n-> / $ ");
     }
```

第八天

实现虚拟文件系统

在mkdir.c文件中查看main函数，发现调用了sys_mkdir()，在mkdir中调用了linkproc()（建立函数指针的指向和连接，同时分配文件空间（虚拟的））

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

  if(argc < 2){
    fprintf(2, "Usage: mkdir files...\n");
    exit(1);
  }

  for(i = 1; i < argc; i++){
    if(mkdir(argv[i]) < 0){
      fprintf(2, "mkdir: %s failed to create\n", argv[i]);
      break;
    }
  }

  exit(0);
}
```

在mkdir中调用了linkproc()

```c
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);
      //printf("mkdir success\n");
      linkproc();
      return 0;
  }
  eunlock(ep);
  eput(ep);
  return 0;
```

linkproc给dirent的函数指针建立指向。 给proc文件夹下的文件建立联系。遍历当前所有进程，给所有进程通过ealloc_memory函数分配空间，但不是真正在磁盘中分配空间。

```
void linkproc()
{
    //printf("linkproc\n");
    struct dirent* ep = ename("/proc");
    elock(ep);
    ep->e_func = &procfs_func;
    struct proc* p;
    struct dirent* tep;
    char pdir[20];
    for (p = proc; p < &proc[NPROC]; p++) {
        itoa(p->pid, pdir);
        //printf("pid:%d state:%d\n", p->pid, p->state);
        if (p->state != UNUSED) {
            tep = ealloc_memory(ep, pdir, ATTR_DIRECTORY);
            ealloc_memory(tep, "stat", ATTR_ARCHIVE);
            //printf("%s DIR\t0\n", fmtname(tep->filename), tep->file_size);
        }
    }
    eunlock(ep);
}
```

再给proc下的文件创建stat文件。

```
//changed
struct dirent_functions fs_func = {eread};
struct dirent_functions procfs_func = {proc_eread};
```

创建好proc文件夹之后我们可以通过ls /proc命令查看当前文件夹下的文件。

```
-> / $ ls /proc
.                               DIR     0
..                              DIR     0
1                               DIR     0
2                               DIR     0
19                              DIR     0
-> / $
```

用户态直接调用接口，open是打开文件路径下的文件，通过fstat函数去读取文件信息，然后判断如果文件类型是T_DIR最后通过readdir函数去不断读取文件夹下的所有文件（通过返回值判断是否还有文件或者文件夹）

```
if((fd = open(path, 0)) < 0){
  fprintf(2, "ls: cannot open %s\n", path);
  return;
}

if(fstat(fd, &st) < 0){
  fprintf(2, "ls: cannot stat %s\n", path);
  close(fd);
  return;
}

if (st.type == T_DIR){
  while(readdir(fd, &st) == 1){
    printf("%s %s\t%d\n", fmtname(st.name), types[st.type], st.size);
  }
} else {
  printf("%s %s\t%l\n", fmtname(st.name), types[st.type], st.size);
}
close(fd);
}
```

```
 * @param    dp       the directory
 * @param    ep       the struct to be written with info
 * @param    off      offset off the directory
 * @param    count    to write the count of entries
 * @return   -1       meet the end of dir
 *           0        find empty slots
 *           1        find a file with all its entries
```

用户态调用cat接口，cat通过逐层调用函数最后调用fileread函数，在其中判断文件类型，最后调用在linkproc函数中指向的建立联系的eread_func函数。

```
case FD_ENTRY:
    elock(f->ep);
    //printf("FD_ENTRY\n");
    //changed
      if((r = f->ep->e_func->eread_func(f->ep, 1, addr, f->off, n)) > 0){
        //printf("if success\n");
        f->off += r;
      }
    eunlock(f->ep);
```

```
int proc_eread(struct dirent* entry, int user_dst, uint64 dst, uint off, uint n) {
    // printf("file:%s\n", entry->parent->filename);
    // printf("size:%d\n", entry->file_size);
    // printf("off:%d\n", off);
    char* states[] = {
    [RUNNABLE] "R",
    [RUNNING]  "R",
    [SLEEPING]  "S",
    [ZOMBIE]    "Z",
    };
    if (off > entry->file_size || off + n < off || (entry->attribute & ATTR_DIRECTORY)) {
        return 0;
    }
    char buf[128] = {"\0"};
    char tmp[10];
    int pid = atoi(entry->parent->filename);
    struct proc* p = getproc(pid);
    if (p == NULL) {
        return 0;
    }

    //printf("11111111111\n");
    printf("pid\t(command)\tstate\tppid\tutime\tstime\tcutime\tcstime\tvsz\n");
    strcat(buf, entry->parent->filename);
    strcat(buf, "\t(");
    strcat(buf, p->name);
    strcat(buf, ")\t\t");
    strcat(buf, states[p->state]);
    strcat(buf, "\t");
    if (p->pid == 1) {
        strcat(buf, "0");
    }
    else {
        itoa(p->parent->pid, tmp);
        strcat(buf, tmp);
    }
    strcat(buf, "\t");
```

运行结果：

```
-> / $ mkdir /proc
-> / $ cat /proc/1/stat
pid     (command)       state   ppid    utime   stime   cutime  cstime  vsz
1       (init)          S       0       0       0       0       0       12
-> / $
```

第九天

实现PS命令

第一步：添加PS系统调用

①和之前步骤一样添加相应的系统调用sys_ps

②根据前面已经做好的搜集的进程信息，通过系统调用sys_ps,调用内核的ps函数，对所有信息进行输出

```
uint64 sys_ps(void){
  //printf("sys_ps\n");
  ps();
  return 0;
}
```

```
int                    ps();
#endif
```

第二步：PS遍历所有进程并输出所有信息

```
int ps()
{
struct proc* p;
    char time[20] = {"\0"};
    uint64 t;
    char* states[] = {
    [RUNNABLE] "R",
    [RUNNING]  "R",
    [SLEEPING]  "S",
    [ZOMBIE]    "Z",
    };
    int flag = 0;
    printf("PID\tCOMMAND\tSTATE\tPPID\tTIME\t\tELAPSED\tVSZ\n");
    for (p = proc; p < &proc[NPROC]; p++) {
        if (p->state != UNUSED) {
            strncpy(time, "\0", 20);
            flag = 1;
            printf("%d\t%s\t%s\t", p->pid,p->name, states[p->state]);
            if (p->pid == 1) {
                printf("0\t");
            }
            else {
                printf("%d\t", p->parent->pid);
            }
            proctime((p->proc_tms.stime+p->proc_tms.utime)/10, time);
            printf("%s\t", time);
            strncpy(time, "\0", 20);
            acquire(&tickslock);
            t = ticks;
            release(&tickslock);
            proctime((t - p->pstmp) / 10, time);
            printf("%s\t%d\n", time,p->sz/1024);
        }
    }
    if (flag==0)
        return -1;
    return 0;

}
```

运行结果：

```
-> / $ ps
PID     COMMAND STATE   PPID    TIME            ELAPSED VSZ
1       init    S       0       00:00:00        00:47:36        12
2       sh      S       1       00:00:00        00:47:36        84
18      ps      R       2       00:00:00        00:00:00        12
```