哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2021 秋季

课程名称：操作系统

实验名称：系统调用

实验性质：课内实验

实验时间：2021/10/21 地点：T2210

学生班级：计科 2 班

学生学号：1190303311

学生姓名：王志军

评阅教师：

报告成绩：

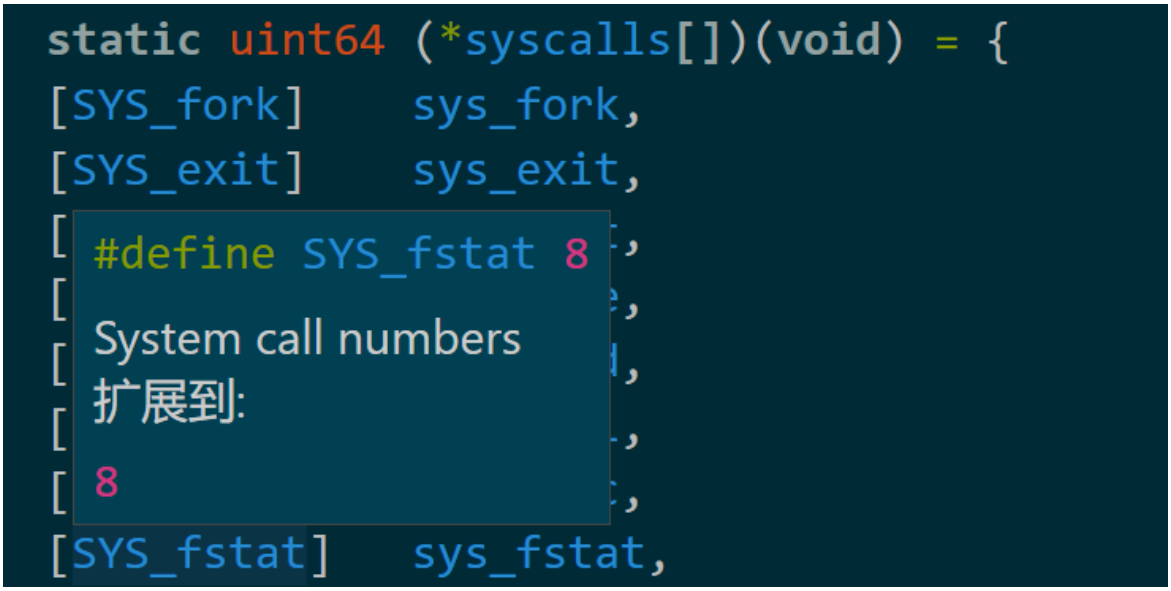
实验与创新实践教育中心印制

2018 年 12 月

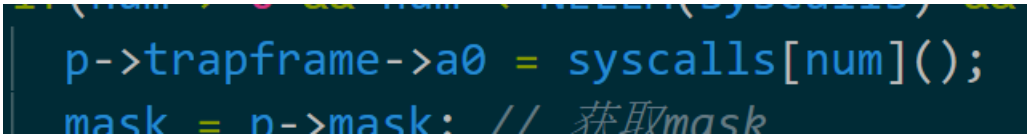
一、 回答问题

1. 阅读 kernel/syscall.c，试解释函数 syscall()`如何根据系统调用号调用对应的系统调用处理函数（例如 sys_fork）？syscall()`将具体系统调用的返回值存放在哪里？

   Syscall()定义了一个数组，通过数组来获取相应调用号的函数，数组中，如[SYS_fork]是 fork 系统调用号的宏定义，后面的 sys_fork 就是内核中对应的函数，这些函数的实现都在 sysproc 中。



   Syscall()将系统调用的返回值存在寄存器 a0，一般情况下函数返回值都会放到 a0 或者 eax,rax 寄存器中。



2. 阅读 kernel/syscall.c，哪些函数用于传递系统调用参数？试解释 argraw()函数的含义。

   用于传递系统调用参数的函数有 argraw(int n), argint(int n, int *p),argaddr(int n, uint64 *ip), argstr(int n ,char *buf, int max)

   Argraw(int n)函数，函数体是一个 switch 语句，先获取当前进程的 proc 结构体，其中包含了参数的值，根据参数 n 返回相应第 n 个参数，比如 n=2 就

返回寄存器 a2 的值：

```
case 2:
  return p->trapframe->a2;
```

3. 阅读 kernel/proc.c 和 proc.h，进程控制块存储在哪个数组中？进程控制块中哪个成员指示了进程的状态？一共有哪些状态？

进程控制块存储在 proc 数组中，NPROC 是最大进程个数；

```
struct proc proc[NPROC];
```

PCB 结构体定义在 proc.h 中，proc 中 state 指示了进程的状态，一共有 5 个状态，unused 空闲，sleeping 休眠，runnable 可运行 ,running 正在运行,zombie 僵死进程

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

```
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 wai
  int pid;                     // Process ID
  int mask;                    // mask

  // 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)
};
```

4. 阅读 kernel/kalloc.c，哪个结构体中的哪个成员可以指示空闲的内存页？Xv6 中的一个页有多少字节？

结构体 mem 的成员 freelist 可以指示空闲的内存页，实际上也是空闲内存页链表，一个页有 4096 个字节，4K.

```
struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;
```

5. 阅读 kernel/vm.c，试解释 copyout()函数各个参数的含义。

下面是函数定义，pagetable 是用户进程 PCB 中的页表；

dstva 是用户虚拟地址空间的一个地址，是数据要从内核地址空间复制到用户地址空间时，用户地址空间接收的地址，传递时要转换成 uint64；

src 是要复制的数据对象在内核地址空间的起始地址，按照字节为单位复制的；

len 是数据的字节数，按字节传递；

```
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
```

二、 实验详细设计

1、 trace 系统调用设计

a) 首先明确系统调用的过程，即进程代码中有系统调用，就去 user.h 找对应的函数定义，如果有就通过脚本 usys.pl 将这个系统调用的号写入 a7 寄存器，然后 ecall 陷入内核，内核在 syscall（）函数中判定系统调用的函数，然后跳转到函数中执行，系统调用函数的实现都在 sysproc.c 中，myproc()可以获取当前进程的 PCB。

b) 看过指导书后确定了思路：trace 在命令中第一个执行，参数是 mask，指定了需要追踪的调用号，因此 trace 系统调用的作用就是将 mask 写入当前内核上下文的 PCB 当中，然后后续的程序使用系统调用时就可以获取这个 mask,并依此判断自己是否需要输出。

c) 首先做准备工作，在 user.h，usys.pl，syscall.h 等文件写入 trace 相关的定义；

d) 设计函数，sys_trace()，先通过 argint 获取参数 mask，然后写入 PCB；

```
sys_trace(void)
{
  int mask;
  if(argint(0, &mask) <0 ) // 获取mask参数
    return -1;
  struct proc *p = myproc(); // 将当前进程的proc结构
  p->mask = mask; // 因此trace后面的程序根据mask进行
  return 0;
}
```

e) 现在后面的程序就可以通过 PCB 获取 mask 了,因为所有系统调用都需要经过 syscall()，因此在这个地方写相应的输出逻辑，就不用一个个地修改系统调用定义了，在 syscall()中获取 mask，将调用号移位处理后和 mask 按位于并判断，然后输出，这个地方需要输出系统调用地名字，定义了 sys_call_num，一个字符串数组，保存了名字。

```
syscall(void)
{
  int num;
  int a;
  int mask;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  argint(0,&a);//在if外面获取，防止第一个参数就会被返回值覆盖
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
    mask = p->mask; // 获取mask
    if(((1 << num) & mask) !=0) // 判断这一位是否是1
    {
      printf("%d: %s(%d) -> %d\n",p->pid,sys_call_num[num-1],(int)a,p->trapframe->a0);
    }
```

Fork 还需要修改，fork 子进程会复制父进程的一部分 PCB，由于 fork 是一个属性一个属性地 copy 地，而我们在 PCB 结构体中加了一个 mask 属性，所以这里还要 copy 父进程地 mask。

```
np->mask = p->mask;
```

2、 sysinfo 系统调用设计

a) sysinfo 要求获取系统的三个属性，根据指导，大概思路是三个属性各写一个函数实现（内核中），然后系统调用是 sysinfo()，它的参数是一个用户地址空间指向 sysinfo 结构体的指针，也就是需要 sysinfo 获取这个地址，然后通过三个函数得到自己内核地址空间下的 sysinfo 结构体对象（已有属性值），然后将其复制到用户地址空间即可。

b) 三个函数设计

首先明确，三个函数的参数只有一个，就是系统调用 sysinfo 传递来的内核地址空间的结构体指针 info，函数内计算好值之后直接赋值;
首先是计算剩余内存空间，由上面的回答，kmem.freelist 是空闲内存页链表，因此直接统计其长度即可：

```
int
calsize(struct sysinfo *info)
{
  struct run *r;
  r = kmem.freelist; //kmem.freelist 是内存空闲链表
  uint64 count = 0;
  while(r)
  {
    count++;
    r = r->next;
  }
  info->freemem = count*PGSIZE;
  return 0;
}
```

接着是空闲进程数量，proc 保存了所有进程，遍历获取每个进程状态，统计 unused 个数：

```
unused_pro(struct sysinfo *info)
{
  struct proc *p;
  uint64 count = 0;
  for(p = proc; p < &proc[NPROC]; p++) // p
  {
    if(p->state == UNUSED)
      count++;
  }
  info->nproc = count;
  return 0;
```

最后是可用文件描述符个数，NOFILE 宏定义为最大的文件描述符个数，PCB 中的 ofile 是一个指针数组，元素是文件描述符指针，因此遍历这个数组，统计为空的指针个数即可：

```
usable_file(struct sysinfo *info)
{
  struct proc *p;
  int count = 0;
  p = myproc(); //获取当前进程的proc结构体
  for(int i=0; i<NOFILE; i++)// NOFILE是文件描
  {
    if(!p->ofile[i]) //p->ofile是一个file指针数
      count++;
  }
  info->freefd = count;
  return 0;
```

c) 最后编写系统调用 sysinfo()，
如图，pin 是内核地址空间的结构体，用来存储上面计算出的三个值，pout 是参数，也就是用户地址空间的地址，也就是 pin 要复制的地址，获取值之后直接 copyout 即可。

```
sys_sysinfo(void)
{
  uint64 pout; //用来获取参数，也就是copyout到用户内存地址空间的地址
  struct sysinfo pin; //用来获取结构体三个属性的值
  struct proc *p = myproc(); // 生成copyout的参数，pagetable
  if(argaddr(0, &pout)<0) // 获取参数，一个在用户地址空间指向sysinfo结构体的
    return -1;
  calsize(&pin); // 获取内存剩余字节数
  unused_pro(&pin); // 获取空闲进程数
  usable_file(&pin); // 获取可用文件描述符数
  if(copyout(p->pagetable, (uint64)pout, (char *)&pin, sizeof(pin)) < 0)
    return -1; // 将内核地址空间的pin复制到用户地址空间
  return 0;
```

三、 实验结果截图

学号 1190303311

```
make[1]: Leaving directory '/home/guests/1190303311/xv6-labs-2020'
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (5.4s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (1.1s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (14.0s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (2.9s)
== Test time ==
time: OK
Score: 35/35
```