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

实验设计报告

开课学期：大三上

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：2021.10.22 地点：T2210

学生班级：1901105

学生学号：190110509

学生姓名：王铭

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？

   每次进行系统调用时，要调用的系统调用号存在陷入内核进程前的 a7 寄存器中，只需获取 a7 寄存器的值即可知道要调用的系统调用号，syscall.c 中有存储函数指针的数组，而各个系统调用号就是找到该函数指针的下标，故 syscall()函数根据系统调用号查询该数组即可调用相应的函数。通过 syscall()的代码可以看出，syscall()将系统调用的返回值存放到了 a0 这个寄存器中

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。

   argraw(),argint(),argstr(),argaddr()都可以获取系统调用参数。系统调用的参数一般存储在寄存器当中，argraw()可以根据传入的寄存器号返回寄存器内存储的值，从而起到获取参数的作用。

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。进程控制块中的 state 成员指示了当前进程的状态。一共有五种状态，UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE。对应未使用，休眠，就绪态，运行态和僵尸进程。

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？

   kmem 结构体中的 freelist 指向当前空闲的内存页的链表头节点，通过调试可以知道 xv6 中的 PGSIZE 为 4096 个字节。

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

   copyout 传入第一个参数为页表，第二个参数即为虚拟地址，第三个为要开始复制的内存首地址，第四个为要复制的字节数。

   copyout 中利用 walkaddr 函数可以用传入的虚拟地址在传入的页表中进行查询，从而获得虚拟地址对应的物理页，利用给出的虚拟地址计算出的偏移量与物理页的基地址相加可以获取正确的物理地址，从而正确地将信息复制给用户态。

二、 实验详细设计

1.trace

按照要求，定义 trace 的系统调用号，并加入 user/usys.pl 中加入 entry 项。在 PCB 中添加一个成员变量 mask，表示需要追踪的程序，在进行系统调用 syscall()时，通过 a7 寄存器可知要调用的系统调用函数号 n，mask 右移 n 位后判断是否为 0。若为 0，则表示不需要追踪（即对应 mask 位上的二进制数为 0），否则根据 n 可以获取其系统调用的函数名称，获取 a0 寄存器作为传入参数，然后进行系统调用。调用返回后，将返回参数存储的寄存器 a0 打印出来即可。在 fork 出子进程时，子进程继承父进程的 mask 即可。

系统调用号以及 entry 项：

```
23  #define SYS_trace  22
```

```
39  entry("trace");
```

PCB 定义：

```
86  struct proc {
87    struct spinlock lock;
88
89    // p->lock must be held when using these:
90    enum procstate state;        // Process state
91    struct proc *parent;         // Parent process
92    void *chan;                  // If non-zero, sleeping on chan
93    int killed;                  // If non-zero, have been killed
94    int xstate;                  // Exit status to be returned to parent's wait
95    int pid;                     // Process ID
96    // these are private to the process, so p->lock need not be held.
97    uint64 kstack;               // Virtual address of kernel stack
98    uint64 sz;                   // Size of process memory (bytes)
99    pagetable_t pagetable;       // User page table
100   struct trapframe *trapframe; // data page for trampoline.S
101   struct context context;      // swtch() here to run process
102   struct file *ofile[NOFILE];  // Open files
103   struct inode *cwd;           // Current directory
104   char name[16];               // Process name (debugging)
105   int mask;
106 };
107
```

fork 时继承父进程的 mask：

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

trace 函数定义：

```
100 uint64
101 sys_trace(void){
102   int n;
103   struct proc* p = myproc();
104   if(argint(0,&n) < -1){
105     return -1;
106   }
107   p->mask = n;
108   return 0;
109 }
```

根据系统调用号获取调用函数名称的数组：

```
135   static char*sysName[] = {
136   [SYS_fork]    "sys_fork",
137   [SYS_exit]    "sys_exit",
138   [SYS_wait]    "sys_wait",
139   [SYS_pipe]    "sys_pipe",
140   [SYS_read]    "sys_read",
141   [SYS_kill]    "sys_kill",
142   [SYS_exec]    "sys_exec",
143   [SYS_fstat]   "sys_fstat",
144   [SYS_chdir]   "sys_chdir",
145   [SYS_dup]     "sys_dup",
146   [SYS_getpid]  "sys_getpid",
147   [SYS_sbrk]    "sys_sbrk",
148   [SYS_sleep]   "sys_sleep",
149   [SYS_uptime]  "sys_uptime",
150   [SYS_open]    "sys_open",
151   [SYS_write]   "sys_write",
152   [SYS_mknod]   "sys_mknod",
153   [SYS_unlink]  "sys_unlink",
154   [SYS_link]    "sys_link",
155   [SYS_mkdir]   "sys_mkdir",
156   [SYS_close]   "sys_close",
157   [SYS_trace]   "sys_trace",
158   [SYS_sysinfo]  "sys_sysinfo"
159   };
```

Syscall()更改：

```
160   void
161 syscall(void)
162   {
163     int num;
164     struct proc *p = myproc();
165     num = p->trapframe->a7;
166     int x = p->trapframe->a0;
167     int pid = sys_getpid();
168     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
169       p->trapframe->a0 = syscalls[num]();
170       if( ((p->mask >> num) & (1)) == 1){ // 说明需要trace
171           printf("%d: %s(%d) -> %d\n",pid,sysName[num],x,p->trapframe->a0);
172       }
173     } else {
174       printf("%d %s: unknown sys call %d\n",
175               p->pid, p->name, num);
176       p->trapframe->a0 = -1;
177     }
178   }
```

2.sysinfo

①获取当前未使用的进程数目。

遍历所有进程，判断每一个进程的状态是否为 UNUSED 即可。

```c
int getUProc(){
  int number = 0;
  for(struct proc *p = proc;p < &proc[NPROC];p++){
    if(p->state == UNUSED){
      number++;
    }
  }
  return number;
}
```

②获取当前空闲页的字节。

获取当前空闲页链表的首字节，遍历链表后可得到空闲页的数目，空闲页的数目乘上每个页的大小 PGSIZE 即可获取所有空闲的字节数。

```c
84  int getFreeMem(){
85    struct run *r;
86    int number = 0;
87    acquire(&kmem.lock);
88    r = kmem.freelist;
89    while(r){
90      number++;
91      r = r->next;
92    }
93    release(&kmem.lock);
94    return number * PGSIZE;
95  }
```

③获取当前进程未使用的文件描述符。

遍历当前进程所有文件指针，若未使用计数器加一即可。

```c
int getFreeFd(){
  struct proc *p = myproc();
  int i,counter = 0;
  for(i = 0;i < 16;i++){
    if(p->ofile[i] == 0){
      counter++;
    }
  }

  return counter;
}
```

④将获取的信息保存到用户态的结构体

调用 copyout 函数，传入用户态进程的页表和虚拟地址，以及当前信息存储的首地址和字节数即可。

```
uint64
sys_sysinfo(void){
    uint64 s;// 指向struct sysinfo
    struct proc* p = myproc();
    struct sysinfo info;
    info.freemem = getFreeMem();
    info.nproc = getUProc();
    info.freefd = getFreeFd();
    if(argaddr(0,&s) < 0){
      return -1;
    }
    if(copyout(p->pagetable,s,(char*)&info,sizeof(info)) < 0 ){
      return -1;
    }
    return 0;
}
```

三、 实验结果截图

运行 make grade 测试，全部通过：

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (3.9s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.8s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (0.7s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (17.0s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (2.3s)
== Test time ==
time: OK
Score: 35/35
```