同濟大學
TONGJI UNIVERSITY

MIT 6.S081 实验报告

Lab 2

2151767 薛树建

日期：2023 年 7 月 20 日

目录

Lab2: system calls

1. System call tracing (moderate)

1) 实验目的

本实验目的是让我们实现一个新的系统调用 trace，从而用来追踪打印用户程序使用系统调用的情况。在该实验中已经提供了一个 trace.c 文件来进行相关系统调用，我们所需要做到工作便是在内核代码中去添加相关的打印函数。

2) 实验步骤

在做完第一个实验之后，应该对于其中的一些操作比较熟悉，在第一个实验中，通过 pl 脚本文件来生成对应的 usyscall.S 汇编文件，通过该文件中的代码来进入内核态。因而想要添加新的系统调用，我们需要在 usys.pl 文件中添加对应代码。

a) 首先通过阅读实验提示可得知本次实验是通过 trace 程序来进行系统调用，我们的目的主要便是对其进行实现（内核层次上），因而首先还是因为在 makefile 文件的 UPROGS 下添加 trace 的声明（trace.c 文件已经被提供好了，我们无需去实现）。

b) 接着我们需要在 usys.pl 中添加 trace 系统调用的入口。仿照其他例子：

```
entry("sleep");
entry("uptime");
entry("trace");
```

其次我们需要在 user.h 中声明该函数，具体返回类型和参数可以通过实验网站的提示或者是 trace.c 中对 trace 函数的具体调用格式来获取。

```
int trace(int);
```

这几件事完成之后便可以转到内核层面了。

c) 我们需要先搞清楚系统调用的具体实现流程，在从用户态进入内核态时，会先将参数保存到 a0 寄存器中（多个参数则会向后保存到 a1,a2…a7）

x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller

在这之后将跳转并运行 usys.S 中的汇编代码，该代码会将系统调用的调用好保存到 a7 寄存器中，随后在进入中断时会根据 a7 寄存器的值来选择对应的内核函数来执行。保存调用号之后，会调用 ecall 函数来进入内核产生中断（traps）.ret 指令则是用来恢复现场。

```
.global trace
trace:
 li a7, SYS_trace
 ecall
 ret
```

随后将会进行一些保存现场的操作，此时不再详述，其接着便跳转到 usertrap 来处理中断。

```
if(r_scause() == 8){
  // system call

  if(p->killed)
    exit(-1);

  // sepc points to the ecall inst
  // but we want to return to the
  p->trapframe->epc += 4;

  // an interrupt will change ssta
  // so don't enable until done wi
  intr_on();

  syscall();
```

r_scause 记录的时产生该中断的原因，8 表示是系统调用，进行了一些操作，我们主要关心最后一个函数 syscall，该函数如下：

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

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
    if(p->mask > 0 && (p->mask&(1<<num)))
    {
      printf("%d: syscall %s -> %d\n", p->pid,syscallname[num-1],p->trapframe->a0);
    }

  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

它通过 trapframe 来获取 a7 的值（在 ecall 指令中 a7 寄存器的值已经被保存到了进程的 trapframe 中），a7 记录的便是系统调用号，以此来调用相应的内核系统调用函数。完成之后便调用 usertrapret()从而返回用户态。

d) 那么我们需要去实现 trace 相关的系统调用，我们需要在 kernel/syscall.h 中去定义系统调用号

```
#define SYS_close  21
#define SYS_trace  22
```

接着在 syscall.c 中添加相关的映射与系统调用函数 sys_trace 函数的声明

```
[SYS_trace]   sys_trace,
[SYS_sysinfo]   sys_sysinfo,
```

```
extern uint64 sys_uptime(void);
extern uint64 sys_trace(void);
```

e) 接下来我们需要考虑 sys_trace 函数该如何实现，实际上我们去查看 trace,c 文件的相关逻辑不难发现，在 trace.c 文件在调用了 trace 之后，直接就进行了 exec 函数来执行后续操作，并没有进行 fork，也就是 trace.c 完全成为了一条新的程序。我们的目的是在该程序中追踪某个系统调用。因而我们在进行 trace 调用时需要保存一下追踪的调用号，保存调用号位置其实是在 proc 结构体中，虽然 exec 函数会完全替换掉原来的程序，但是原来程序的进程 id 不变，proc 中的某些内容也不会改变，因而我们可以在进行 trace 调用时将调用好保存在 proc 中，如下：

```
  char name[16];               // Process name (debugging)
  int mask;                    //记录追踪哪些种类的系统调用
};
```

该值的获取可以用 sys_trace 来执行，通过读取参数值获取

```
uint64
sys_trace(void)
{
  int mask;
  argint(0, &mask); //从内核中获取参数int参数

  myproc()->mask = mask; //对进程的掩码进行设置

  return 0;
}
```

实际上，我们在此处还需要对 fork 函数进行修改，保证子进程能够继承父进程的 mask，不然 children 测试无法通过。

```
  release(&np->lock);

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

f) 最后只需要在 syscall 函数中执行系统调用时，将该调用号与我们所想要追踪的调用进行对比，然后 printf 相关格式化内容就行了。

```
num = p->trapframe->a7;
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
  p->trapframe->a0 = syscalls[num]();
  if(p->mask > 0 && (p->mask&(1<<num)))
  {
    printf("%d: syscall %s -> %d\n", p->pid,syscallname[num-1],p->trapframe->a0);
  }

} else {
  printf("%d %s: unknown sys call %d\n",
          p->pid, p->name, num);
  p->trapframe->a0 = -1;
}
```

3) 实验中遇到的问题和解决方法

本实验主要涉及到去添加一个系统调用，这其中需要你对 xv6 的 traps 实现有一定了解，如何进入内核，保存现场，存放参数的寄存器等等，实际上了解了之后，再来写本实验的代码还是相当轻松的。

关于这部分的内容，解决方法主要还阅读 xv6 的实验指导书以及网站上的一些总结与经验。

4) 实验心得

在完成 xv6 实验 2 的 systemcalltracing 过程中，我深刻体会到了系统调用的重要性和它在操作系统中的作用。同时通过追踪系统调用的执行过程，我也是比较清晰地了解了操作系统与用户程序之间的交互方式。

在本次实验中，我了解了如何通过修改 xv6 的源代码来实现系统调用的追踪与系统调用的执行过程，这些内容为我后续的实验内容帮助还是挺大的

同时，通过本次实验我也了解到了系统调用的重要性，它一方面保护了系统的内核部分，一方面给用户提供了相关的调用接口从而来执行一些任务。

总的来说，通过完成 xv6 实验 2，我对系统调用有了更深入的理解，并且对操作系统的工作原理也有了更清晰的认识。

评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-syscall trace
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.6s)
== Test trace all grep == trace all grep: OK (0.8s)
== Test trace nothing == trace nothing: OK (1.1s)
== Test trace children == trace children: OK (9.6s)
```

2. Sysinfo (moderate)

1) 实验目的

在这个实验中，我们将添加一个系统调用 sysinfo，它收集有关正在运行的系统的信息。系统调用采用一个参数：一个指向 struct sysinfo 的指针（参见 kernel/sysinfo.h）。内核应该填写这个结构的字段：freemem 字段应该设置为空闲内存的字节数，nproc 字段应该设置为 state 字段不为 UNUSED 的进程数。

2) 实验步骤

在追踪调用实验的基础上，本实验应该可以比较容易的解决。步骤如下:

a) 根据 hints，首先在 makefile 的 UPROGS 中添加$U/_sysinfotest，保证 xv6 命令行可以运行该指令，其次在 user/user.h 中添加 sysinfo 函数的声明以及其参数 struct sysinfo 的定义，不要忘记在 usys.pl 中也添加相应的内容。

b) 接着在 kernel/syscall.h 中添加该系统调用好，并在 syscall.c 中添加相应的声明与相应，如下:

```
extern uint64 sys_sysinfo(void);
```

```
[SYS_sysinfo]   sys_sysinfo,
```

```
#define SYS_sysinfo  23
```

c) 根据 hints，接着我们在 kernel/kalloc.c 中添加对空闲空间统计的函数，记录空闲块的结构如下:

```
struct run {
  struct run *next;
};

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

因而可以直接通过遍历 keme 中的 freelist 来获取其数量。需要注意的一点是在访问 freelist 时需要先获取 keme 的锁。函数实现如下:

```
uint64
getfreemem(void){//遍历kmem其本身的freelist即可
  struct run *r;
  uint64 freemem=0;
  acquire(&kmem.lock);//避免并发读写,需要先获取其锁的所有权
  r=kmem.freelist;
  while(r){
    freemem++;
    r=r->next;
  }
  release(&kmem.lock);//释放
  return freemem*PGSIZE;

}
```

d) 接着我们需要在 kernel/proc.c 中添加函数来获取统计状态不为 unused 的进程数量。可以通过遍历 proc 来实现：

```
uint64
getnproc(void){//注意到11行的位置处有进程数组的定义，直接对其遍历即
  struct proc* p=proc;
  uint64 count_proc=0;
  for(;p<proc+NPROC;p++){
    acquire(&p->lock);//获取锁
    if(p->state!=UNUSED)
      count_proc++;
    release(&p->lock);
  }
  return count_proc;
}
```

e) 最后在 sys_sysinfo 中调用这两个函数，并调用 copyout 函数将结果复制到用户指令位置，代码如下：

```
uint64
sys_sysinfo(void){
  struct sysinfo info;//储存内容的结构体
  uint64 addr;//首先从寄存器a0中获取参数指针
  argaddr(0,&addr);
  info.freemem=getfreemem();
  info.nproc=getnproc();
  if(copyout(myproc()->pagetable, addr, (char *)&info, sizeof(info)) < 0)//
      return -1;

  return 0;
}
```

3) 实验中遇到的问题和解决方法

在第一个实验的基础上，本实验遇到的主要困难时如何实现获取进程数量与空闲空间的函数，其实这两个函数在实验网站给定提示中已经暗示了，你在相应文件中添加函数时会注意到相关的结构体变量。

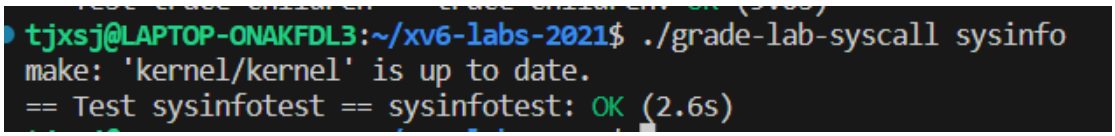
另外在如何将数据拷贝到用户空间，其实也在实验网站的暗示中给出了，可以采用 copyout 函数来实现。

总的来说本实验在第一个实验的基础上，还是很好解决的。

4) 实验心得

通过完成本实验，我对于 xv6 操作系统内部的一些进程与空闲块管理有了一些了解，同时也了解了 copyout 函数的用法。

评分如下：



总结

通过完成本次实验，我对于 xv6 操作系统内部的系统调用有了更加深入的理解，同时也对其 trap 有所学习与理解。

本实验总体评分如下：