Lab 实验报告 II

Lab: System Calls

丁睿

dromniscience@gmail.com

更新：2020 年 10 月 28 日

目录

1 任务完成清单

Subtask	Done?	Time
`sys_trace()`	Y	1h
`sys_sysinfo()`	Y	1h

Grade: 35/35

2 详细情况 & 困难和收获

我的作业已开放在这个网页上。

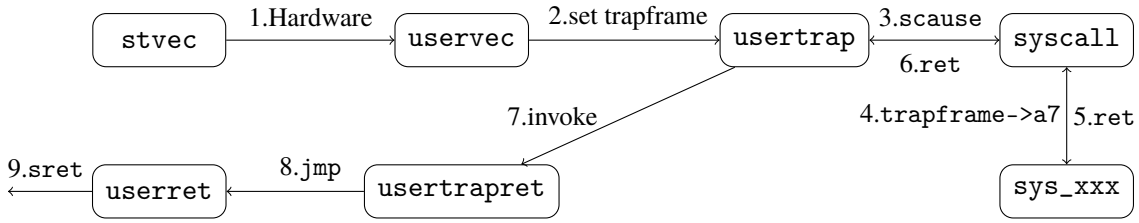
这一次的练习围绕着注册新的系统调用展开，但是涉及到不少内核数据结构的使用，例如空闲物理页链表、进程表等 (如下所示)。由于对这些细节尚不够清楚，我在使用它们的过程中伴随着不少经验猜测，或者通过阅读上下文中的代码来获得启发。事实证明，这些猜测以及启发都是正确的。这也是本次 lab 作业我完成得比较顺利的原因。

```
/* kernel/kalloc.c */
struct run {
  struct run *next;
};

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

/* kernel/proc.c */
struct proc proc[NPROC];
```

我们简单地归纳一下系统调用的过程，以指导我们注册新的系统调用 (这在 lab 主页上有详细的指导，但我以为理清这背后的逻辑既不难且很重要)。



注册系统调用的另一部分是在用户文件中完成的，即要提供对应的函数原型 (`user/user.h`) 以及提供一个汇编例程标签以供链接使用 (`user/usys.pl`)。另外，观察`syscalls[]`系统调用表的构造和使用方式，可知我们新加入的系统调用未必需要在号码上连续。

2.1 Trace

是否跟踪系统调用是一个进程状态，因此我们应该扩展`struct proc`的成员，即追加一个`int traced`来表明哪些系统调用被追踪了。为了使子进程继承该状态，需要显式地在`kernel/proc.c`中的`fork()`函数中追加`np->traced = p->traced`。其中`np`和`p`分别指向父子进程的 PCB。它的添加位置相对灵活。最后，由于每个系统调用都以`syscall`作为统一的出入口，我们需要在此检查当前调用号是否被追踪。于是增加以下代码：

```
/* kernel/syscall.c:syscall() */
if(p->traced & (1 << num))
        printf("%d: syscall %s -> %d\n",
                    p->pid, sysname[num], p->trapframe->a0);
```

其中数组`const char *sysname[]`将调用号转换成调用名。

敏感的同学可能会有如下两个疑问：

1. **如果系统调用是一个`fork()`，那么`p`不需要使用`p = myproc()`更新吗？**
   实际上，这根本不需要，也不应该这样做。请注意，此时只有内核返回到`syscall()`余下的部分，子进程是不会回到这里的 (它可能正在等待调度)！因此根本不存在子进程在`fork()`后出来后被追踪打印的可能！当然，确实`p`仍然指向父进程的进程表表项。

2. **如果在`syscall()`里使用`printf()`，它本身是要调用`write()`这样的系统调用的。那么如果`write()`被同时追踪，那么岂不是会继续调用`write()`，从而产生一个无尽循环？**
   这是很有意思的思考。但前提是我们的追踪发生在用户级别陷入内核的过程中，上面设想的情景从`write()`开始就不满足这个要求了。请注意为什么这里我们的函数都以`user`作为前缀。某种意义上就是为了区别这一点。

2.2 Sysinfo

这一次我们的核心工作是完成那两个函数的编写，即填充以下这个数据字段的函数。

```
/* kernel/sysinfo.h */
struct sysinfo {
  uint64 freemem;   // amount of free memory (bytes)
  uint64 nproc;     // number of process
};
```

我将直接给出自己的解答并说明为什么会这样编写。

首先是统计物理内存的空余字节数的函数`kfreemem()`。

```
/* kernel/kalloc.h */
uint64
kfreemem(void)
{
    uint64 frpg = 0;

    acquire(&kmem.lock);
    struct run *r = kmem.freelist;
    while(r) {
        frpg += 1;
        r = r->next;
    }
    release(&kmem.lock);

    return frpg * PGSIZE;
}
```

很大程度上，我受到了kfree()的启发：

```
/* kernel/kalloc.c:kfree() */
r->next = kmem.freelist;
kmem.freelist = r;
```

明显看出kmem维护了一个空闲页的链表，因此只需计数该链表的节点，并乘以物理页的大小，我们就得到了答案。**加锁**则是为了互斥地访问共享数据。这是每个内核进程都可以看到的。例如，如果分布在两个核上的进程同时申请释放一个磁盘页，那么内核就有可能并发地访问这个数据结构。

我们直接给出后一个答案。它遍历进程表统计不是UNUSED的表项数目。由于进程表也可能被并发地修改和访问，因此必须对它加一个互斥锁。

```
uint64
nproc(void)
{
    uint64 cnt = 0;
    struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state != UNUSED) cnt++;
    release(&p->lock);
  }

  return cnt;
}
```

3 对课程和 lab 的建议

这一次暂时没有新的想法。如果愿意，您可以回顾第一次 lab 报告中我提到的两点内容。

4 参考资料

1. *xv6: a simple, Unix-like teaching operating system*
   Russ Cox, Frans Kaashoek, Robert Morris *August 31, 2020*
2. 现代操作系统 [M]
   A.S.Tanenbaum, H. Bos 著, 陈向群 马洪兵 译, 北京: 机械工业出版社, 2011: 47-95
3. Git 教程 by 廖雪峰