OS Lab 1

System Calls

王世炟

PB20151796

October 26, 2022

Part 1: 实验要求

本次实验要求我们完成 System Calls 的某些功能的添加，了解用户态与内核态间的联系。
简要介绍一下系统调用：
系统调用的功能就是用户态的程序需要执行等级更高的指令时，需要先切换到内核态，在内核态执行完后，再切换到用户态。过程如下图所示：

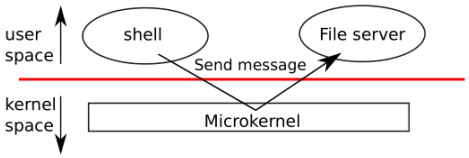

图 1: A microkernel with a file-system server

切换时，使用的是 cpu 特定的中断符号。所有的系统调用都是通过该中断进入到内核态中，并将所需要的执行的指令码传入到中断内核态中。内核态的程序取出指令码，执行相应的指令，然后返回到用户态。
主要涉及到的文件有：trace.c user.h usys.pl syscall.c sysproc.c proc.h proc.c kalloc.c defs.h

Part 2: System call tracing

2.1 问题简述

该部分需要我们实现一个功能 trace ，其作用是根据参数 mask 追踪相应的系统调用。用户态中的代码 user/trace.c 已经给出，需要我们实现的是内核态中的 trace 功能。

2.2 实验步骤

(1)Makefile

首先根据实验提示，首先在 Makefile 文件中的 UPROGS 部分中添加 $U/_trace：

```
UPROGS=\
	$U/_cat\
	$U/_echo\
	$U/_forktest\
	$U/_grep\
	$U/_init\
	$U/_kill\
	$U/_ln\
	$U/_ls\
	$U/_mkdir\
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_trace \
```

(2) **添加声明**

可以观察到 user/trace.c 中调用了 trace() 函数，但是它并未被定义，所以在 user/user.h 中加入声明：int trace(int);

同时根据给出的提示，为了生成进入中断的汇编文件，需要在 user/usys.pl 添加进入内核态的入口函数的声明，以便使用中断指令进入内核态；entry("trace");

```
sub entry {
    my $name = shift;
    print ".global␣$name\n";
    print "${name}:\n";
    print "␣li␣a7,␣SYS_${name}\n";
    print "␣ecall\n";
    print "␣ret\n";
}
```

根据该文件的内容可以判断，汇编语言将指令码放到了 a7 寄存器中。在内核态 kernel/syscall.c 的 syscall 函数中，取出寄存器中的指令码，然后调用对应的函数。

所以，在 kernel/syscall.h 中添加相应指令码。

```
// System call numbers
```

```
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup     10
#define SYS_getpid  11
#define SYS_sbrk    12
#define SYS_sleep   13
#define SYS_uptime  14
#define SYS_open    15
#define SYS_write   16
#define SYS_mknod   17
#define SYS_unlink  18
#define SYS_link    19
#define SYS_mkdir   20
#define SYS_close   21
#define SYS_trace   22
```

(3) **实现** sys_trace() **函数**

根据提示，在 kernel/proc.h struct proc 中 (即 PCB) 加入一个变量 mask, 用于储存用户态传进来的 mask 。

```
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

  // 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)
  int mask;
};
```

然后在 kernel/sysproc.c 中添加 sys_trace() 函数为 mask 赋值为该字段进行赋值，赋值的 mask 为系统调用传过来的参数，放在了 a0 寄存器中：

```
uint64
sys_trace(void)
{
  int mask;
  if(argint(0, &mask) < 0)
    return -1;
  myproc()->mask = mask;
  return 0;
}
```

其中 argint(int n, int *ip) 函数位于 kernel/syscall.c 中，作用是将 trapframe 中 a_n 寄存器中的值取出来。myproc() 函数的作用是获取当前进程的 PCB。

(4) **追踪子进程**

需要跟踪所有 trace 进程下的子进程，在 kernel/proc.c 的 fork() 代码中，添加子进程的 mask。

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
```

```
  np->sz = p->sz;

  np->parent = p;

  //copy the mask
  np->mask = p->mask;

  .
  .
  .

  release(&np->lock);

  return pid;
}
```

(5) **打印**

先叙述一下代码运行的流程：user/trace.c 中调用 trace() 函数，引发中断，user/usys,pl 生成汇编文件 usys.S，是用户态系统调用接口，将 SYS_trace 的值放入 a7 寄存器，然后使用 ecall 命令进入内核，跳转到 kernel/syscall.c 中的 syscall() 函数处运行，于是可以这样修改：

```
void syscall(void)
{
  int num;
  char *syscalls_name[] = {
       [SYS_fork] "fork",
       [SYS_exit] "exit",
       [SYS_wait] "wait",
       [SYS_pipe] "pipe",
       [SYS_read] "read",
       [SYS_kill] "kill",
       [SYS_exec] "exec",
       [SYS_fstat] "fstat",
       [SYS_chdir] "chdir",
       [SYS_dup] "dup",
       [SYS_getpid] "getpid",
       [SYS_sbrk] "sbrk",
       [SYS_sleep] "sleep",
       [SYS_uptime] "uptime",
       [SYS_open] "open",
       [SYS_write] "write",
       [SYS_mknod] "mknod",
```

```
22          [SYS_unlink] "unlink",
23          [SYS_link] "link",
24          [SYS_mkdir] "mkdir",
25          [SYS_close] "close",
26          [SYS_trace] "trace",
27      };
28      struct proc *p = myproc();
29
30      num = p->trapframe->a7;
31      if (num > 0 && num < NELEM(syscalls) && syscalls[num])
32      {
33        p->trapframe->a0 = syscalls[num]();
34        if ((1 << num) & p->mask)
35        {
36          printf("%d: syscall %s -> %d\n", p->pid, syscalls_name[num],
37          p->trapframe->a0);
38        }
39      }
40      else
41      {
42        printf("%d %s: unknown sys call %d\n",
43                p->pid, p->name, num);
44        p->trapframe->a0 = -1;
45      }
46  }
```

并且在该函数上面的 static uint64 (*syscalls[])(void) 中添加 [SYS_trace] sys_trace, , 在上面加入 extern uint64 sys_trace(void); 用于将 sys_trace(void) 函数扩展进来。这样，即完成了第一部分，实现了 trace 的功能。

Part 2: Sysinfo

2.1 问题简述

该部分让我们实现一个系统调用 sysinfo ，它收集有关正在运行的系统信息，记录空闲内存的字节数与状态不是 UNUSED 的进程数。

2.2 实验步骤

(1)Makefile

同上，添加 $U/_sysinfotest 到 Makefile 文件中

(2) **添加声明**

同样需要添加一些声明才能进行编译，启动 qemu。需要以下几步：

在 user/user.h 文件中加入函数声明，同时添加结构体声明；

在 user/usys.pl 添加进入内核态的入口函数的声明；

同时在 kernel/syscall.h 中添加系统调用的指令码。

(3) **获取内存信息**

可以在 kernel/sysinfo.h 中查看结构体 struct sysinfo，其中只有两个字段，一个是保存空闲内存信息，一个是保存正在运行的进程数目。

两个字段的信息都需要自己写函数调用来获取，先来获取内存信息。内存信息的处理都写在 kernel/kalloc.c 文件中了，内存信息以链表的形式存储，每个节点存储一个物理内存页。

从 kfree 函数中可以发现，每次创建一个页时，将其内容初始化为 1，然后将它的下一个节点指向当前节点的 freelist，更新 freelist 为这个新创建的页。也就是说，freelist 指向最后一个可以使用的内存页，它的 next 指向上一个可用的内存页。

因此，我们可以通过遍历所有的 freelist 来获取可用内存页数，然后乘上页大小即可。

```
uint64 free_mem(void)
{
  struct run *r = kmem.freelist;
  uint64 n = 0;
  while (r) {
    n++;
    r = r->next;
  }
  return n * PGSIZE;
}
```

(4) **获取进程数目**

所有的进程有关的操作都保存在 /kernel/proc.c 文件中，其中的 proc 数组保存了所有进程。进程有五种状态，我们只需要遍历 proc 数组，计算不为 UNUSED 状态的进程数目即可。

```
int proc_size(void)
{
  int i;
  int n = 0;
  for (i = 0; i < NPROC; i++)
  {
    if (proc[i].state != UNUSED)
    {
      n++;
```

```
    }
  }
  return n;
}
```

(4) **声明和调用**

在 kernel/defs.h 中添加上面这两个函数的声明。

然后在 kernel/sysproc.c 中的 sys_sysinfo 函数进行调用。

需要注意的是，这里使用 copyout 方法将内核空间中，相应的地址内容复制到用户空间中。这里就是将 info 的内容复制到进程的虚拟地址内，具体是哪个虚拟地址，由函数传入的参数决定（addr 读取第一个参数并转成地址的形式）。

```
uint64
sys_sysinfo(void)
{
  struct sysinfo info;
  uint64 addr;
  struct proc* p = myproc();
  if(argaddr(0, &addr) < 0) {
    return -1;
  }
  info.freemem = free_mem();
  info.nproc = proc_size();
  if (copyout(p->pagetable, addr, (char*)&info, sizeof(info)) < 0) {
    return -1;
  }
  return 0;
}
```

Part 3: 实验结果

运行 make grade，检验结果：

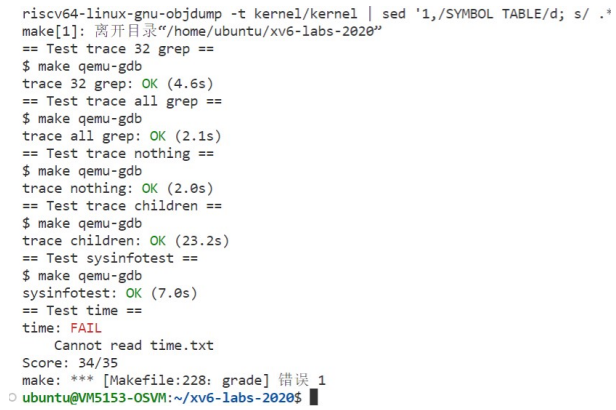

图 2: Result

本次实验完成了系统调用某些功能的实现。在实验过程中遇到了以下问题:

- 对于每个文件都不是很理解，无法弄懂其中的原理
- 各种函数调用有些复杂，调试十分不方便
- 对于用户态和内核态的理解十分不深刻，亟待加强