Syscall 实验报告

20307130340 杨孟颖 计算机

一、实验内容

实验一：添加系统调用函数 sys_trace 用于记录系统调用的 syscall number，系统调用的名称及返回值。

实验二：添加系统调用 sysinfo，查看系统调用的信息，若满足有足够空闲空间、记录数据等于真实情况则测试通过。

二、实现过程

首先按照实验提示中的添加要求在相应头文件中声明函数及结构体。需要在 proc 结构体中添加 int trace_mask，以 int 型数据记录调用的过程的返回值。通过阅读 xv6 book 得知系统调用过程返回值储存在 a0 中，而 syscall.c 中显示应该用 argint(0, &n)获取 a0 的数据。在 int sys_trace(void)中将 mask 存入 int trace_mask 中。SYS_${name}的含义是在 kernel/syscall.h 中#define SYS_${name} xxx 的系统调用号，这样就可以告诉 ecall 要调用几号系统调用。因此在 void syscall(void)中通过 syscall_name = syscall_names[num]获取系统调用的名称，从 xv6 book 中查到 syscall records its return value in p->trapframe->a0，p->trapframe->a0 = syscalls[num]()使系统调用的返回值储存在 a0 中。需要注意的是在 static void freeproc(struct proc *p)函数中因为将 proc 结构体中储存的数据都清零了，所以应该将添加的 trace_mask 同样清零，否则在下次使用 trace 时可能出现问题。

```
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  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

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // 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 trace_mask;
```

```
};

int sys_trace(void){
  int n;
  argint(0, &n);
  myproc()->trace_mask = n;
  return 0;
}
void syscall(void)
{
  int num;
  struct proc *p = myproc();
  char* syscall_name;

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
    if ((p->trace_mask & (1 << num)) != 0) {
        syscall_name = syscall_names[num];
        printf("%d:   syscall   %s   ->   %d\n",   p->pid,   syscall_name,
p->trapframe->a0);
    }
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  if(p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
```

```
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
  p->trace_mask = 0;               //添加记录
}
```

实验二：添加函数声明的过程与实验一一致。Sysinfo.h 中原有定义可以得知需要获取的数据的定义：

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

在添加 char*类型变量学号。

于是在相应文件中定义函数：

```
uint64 sys_sysinfo(void) {
  struct sysinfo info;
  uint64 addr;

  info.nproc = proc_num(); //需要添加
  info.id = "20307130340";
  info.freemem = freemem(); //需要添加
  argaddr(0, &addr);
  if(copyout(myproc()->pagetable, addr, (char *)&info, sizeof(info))<0)
      return -1;
  printf("my student number is %s\n", info.id);
  return 0;
}
```

在 proc.c 中添加 proc_num()，用于计算状态为正在使用的 proc 的数量：

```
uint64 proc_num() {
  struct proc *p;
  uint64 count = 0;
  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state != UNUSED) {
      count++;
    }
    release(&p->lock);
  }
  return count;
}
```

Proc 的总数储存在&proc[NPROC]中，通过移动指针遍历 proc，并且检查 proc 的状态，若状态不为 UNUSED，则计数器+1.

在 kalloc.c 中添加 freemem(void)，用于计数处于空闲状态的空间：

```
uint64 freemem(void) {
  struct run *r;
```

```
  uint64 count = 0;
  acquire(&kmem.lock);
  r = kmem.freelist;
  while (r) {
    r = r->next;
    count++;
  }
  release(&kmem.lock);
  return count * PGSIZE;
}
```

处于空闲状态的空间位于列表 kmem.freelist 中。通过指针遍历 kmem.freelist，计数器增加。

需要注意的是这两个内核态函数都需要上锁以保证原子性。

三、问题

（1） System calls Part A 部分，简述一下 trace 全流程.

请求系统调用函数 exec 时，用户参数按顺序分别被放置到寄存器 a0 和 a1，而系统调用号被放置在 a7，然后执行 ecall 指令，一个用户空间下的 trap 正式发生，如 user/usys.S 中所示。接着，经过 uservec 的准备工作后，开始处理 trap。在 usertrap 中，发现 trap 的起因是系统调用，因此 usertrap 会执行 syscall 函数。syscall 根据传入的系统调用号，索引 syscalls 数组，找到对应的那个系统调用，通过函数指针执行相应的系统调用。即 kernel/syscall.c 中的 static uint64 (*syscalls[])(void)部分，在实验中添加了[SYS_trace] sys_trace。内核执行系统调用 sys_exec（kernel/sysfile.c），完成一些检查和相关用户参数的复制之后，就执行 kernel/exec.c 中的内容。系统调用有一个返回值，在 syscall 中，内核将该返回值放置到 trapframe 中的 a0，稍后在 userret 中会将该返回值取出，从而作为最终返回的用户 a0 值。RISC-V 的调用管理就是将返回值放置在 a0 寄存器中，并且非负值代表调用成功，负值代表调用失败。

（2） kernel/syscall.h 是干什么的，如何起作用的？

定义系统调用的编号。除了定义编号之外还有函数 argraw，argint，argaddr，argstr，fetchstr，fetchaddr。他们的作用分别为：

Argraw：获取用户线程 trap 内核之前的寄存器的信息；

Argint：获取第 n 个寄存器的信息，并用 int 型指针指向它；

Argaddr：获取第 n 个寄存器的信息，并用无符号 int64 型指针指向它(这个内容应该是地址)；

Argstr：获取第 n 个字大小的系统调用参数作为以 null 结尾的字符串。复制到 buf。如果 OK（包括 nul），则返回字符串长度；如果出错，则返回-1。

Fetchstr:从当前进程获取 addr 处以 nul 结尾的字符串。返回字符串的长度,不包括 nul，错误返回-1。

Fetchaddr：从当前进程获取 addr 处的 uint64。

通过这几个函数可以获取进程的参数。

（3） 命令 “trace 32 grep hello README”中的 trace 字段是用户态下的还是实现的系统调用函数 trace？

是用户态下的 trace，由于命令中的 trace 代表的是调用的 trace 文件，而文件 trace.c 中调用的才是系统调用函数 trace：trace(atoi(argv[1])。

实验结果如下：

```
$ trace 32 grep hello README
6: syscall read -> 1023
6: syscall read -> 961
6: syscall read -> 321
6: syscall read -> 0
$ trace 2147483647 grep hello README
7: syscall trace -> 0
7: syscall exec -> 3
7: syscall open -> 3
7: syscall read -> 1023
7: syscall read -> 961
7: syscall read -> 321
7: syscall read -> 0
7: syscall close -> 0
$ grep heool README
$ trace 2 usertests forkforkfork
usertests starting
9: syscall fork -> 10
test forkforkfork: 9: syscall fork -> 11
11: syscall fork -> 12
12: syscall fork -> 13
12: syscall fork -> 14
13: syscall fork -> 15
12: syscall fork -> 16
12: syscall fork -> 17
13: syscall fork -> 18
12: syscall fork -> 19
13: syscall fork -> 20
14: syscall fork -> 21
14: syscall fork -> 22
12: syscall fork -> 23
12: syscall fork -> 24
13: syscall fork -> 25
12: syscall fork -> 26
14: syscall fork -> 27
13: syscall fork -> 28
14: syscall fork -> 29
15: syscall fork -> 30
12: syscall fork -> 31
13: syscall fork -> 32
14: syscall fork -> 33
12: syscall fork -> 34
18: syscall fork -> 35
13: syscall fork -> 36
28: syscall fork -> 37
32: syscall fork -> 38
16: syscall fork -> 39
17: syscall fork -> 40
16: syscall fork -> 41
28: syscall fork -> 42
12: syscall fork -> 43
40: syscall fork -> 44
30: syscall fork -> 45
12: syscall fork -> 46
23: syscall fork -> 47
12: syscall fork -> 48
29: syscall fork -> 49
17: syscall fork -> 50
29: syscall fork -> 51
12: syscall fork -> 52
13: syscall fork -> 53
12: syscall fork -> 54
```

```
29: syscall fork -> 49
17: syscall fork -> 50
29: syscall fork -> 51
12: syscall fork -> 52
13: syscall fork -> 53
12: syscall fork -> 54
26: syscall fork -> 55
41: syscall fork -> 56
41: syscall fork -> 57
12: syscall fork -> 58
13: syscall fork -> 59
12: syscall fork -> 60
OK
9: syscall fork -> 61
ALL TESTS PASSED
```

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ sysinfotest
sysinfotest: start
my student number is 20307130340
my student number is 20307130340
my student number is 20307130340
my student number is 20307130340
my student number is 20307130340
my student number is 20307130340
my student number is 20307130340
my student number is 20307130340
sysinfotest: OK
```

四、实验中遇到的问题及解决

再打印学号时刚开始将这个过程写在 `sysinfotest.c` 文件中，会出现如下错误。将其改写到 `sysinfo` 函数中结果正确。

```
hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
my student number is usertrap(): unexpected scause 0x000000000000000d pid=3
            sepc=0x00000000000009e4 stval=0x0000000080008620
```

五、实验感想

在实验过程中，我对系统调用有了更深刻的理解。用户态下建立系统调用函数需要使用 `system call` 的方法。