Lab2: system calls 系统调用实验

2351882 王小萌

Tongji University,2025 Summer

代码仓库链接：https://github.com/wxmxmw/Tongji-University-xv6-labs-2021.git

本实验旨在帮助了解系统调用跟踪的实现，并演示如何修改 xv6 操作系统以添加新功能。通过本实验，将熟悉内核级编程，包括修改进程结构、处理系统调用以及管理跟踪掩码。

1 System call tracing

1.1 实验目的

本实验旨在添加一个系统调用追踪功能，以便在后续实验中进行调试。

1.2 实验步骤

1. 作为一个系统调用，先要定义一个系统调用的序号。系统调用序号的宏定义在 kernel/syscall.h 文件中。我们在 kernel/syscall.h 添加宏定义，模仿已经存在的系统调用序号的宏定义：

```
#define SYS_trace 22
```

syscall.h

kernel > C syscall.h

```
// 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
```

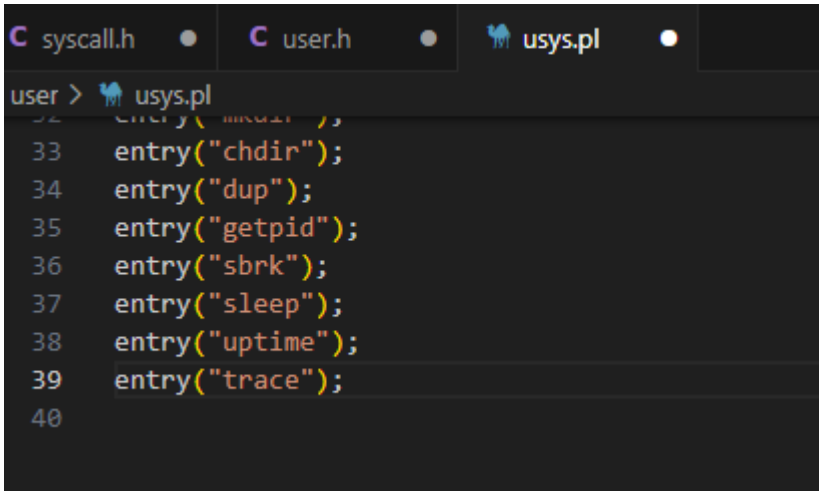
2. 在 user/user.h 文件中声明用户态可以调用 trace 系统调用。

syscall.h | user.h

user > C user.h

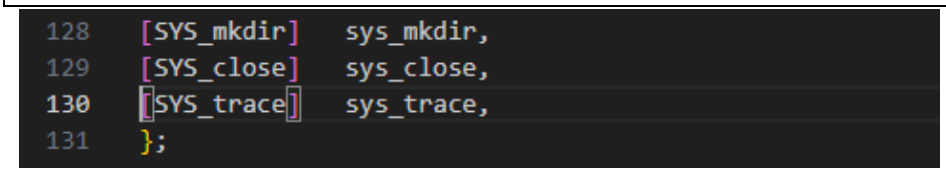
```
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int trace(int);
```

3. 查看 user/usys.pl 文件。该文件中使用 Perl 语言自动生成用户态系统调用接口的汇编语言文件 usys.S。因此，我们需要在 user/usys.pl 文件中加入以下语句：entry("trace");

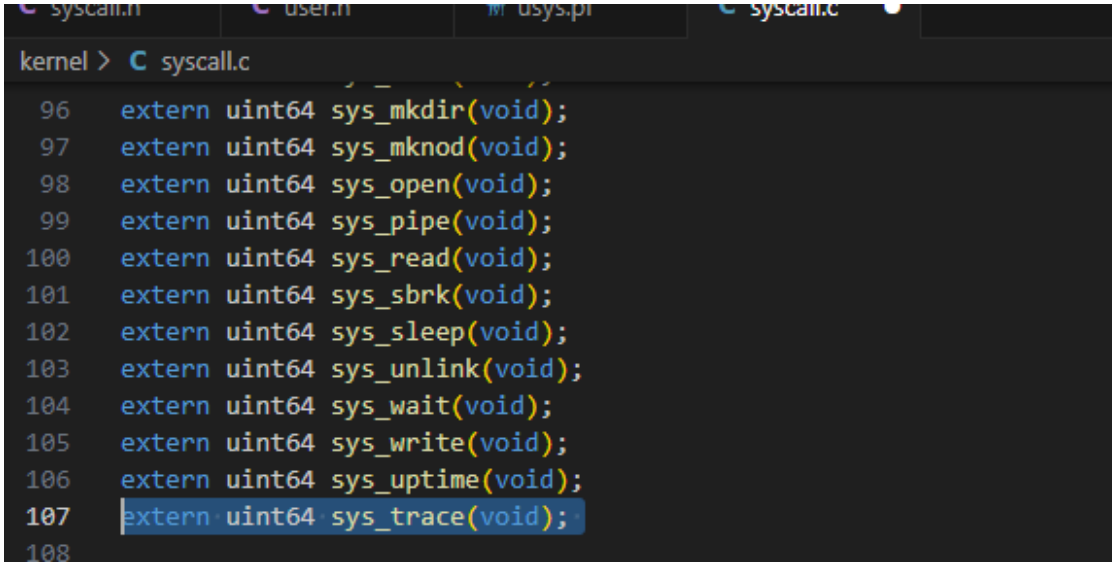


4. 在 kernel/syscall.c 中，把新增的 trace 系统调用添加到函数指针数组 *syscalls[]上：

```
static uint64 (*syscalls[])(void) = {
...
 [SYS_trace] sys_trace, };,
```



5. 在 kernel/syscall.c 文件开头，给内核态的系统调用 trace 加上声明：



6. 根据提示， trace 系统调用应该有一个参数，一个整数“mask(掩码)”，其指定要跟踪的系统调 用。所以，在 kernel/proc.h 文件的 proc 结构体中，新添加一个变量 mask ，使得每一个进程都有自己的 mask ，即要跟踪的系统调用。

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

7. 在 kernel/sysproc.c 给出 sys_trace 函数的具体实现，只要把传进来的参数给到现有进程的 mask 即可：

```
98
99  uint64
100 sys_trace(void)
101 {
102     int mask;
103     // 取 a0 寄存器中的值返回给 mask
104     if(argint(0, &mask) < 0)
105       return -1;
106     // 把 mask 传给现有进程的 mask
107     myproc()->tracemask = mask;
108     return 0;
109 }
110
```

8. 需要在 kernel/syscall.c 中定义一个数组来存储系统调用的名字。这里需要注意，系 统调用的名字必须按顺序排列，第一个为空字符串。当然，也可以去掉第一个空字符串，但在取值 时需要将索引减一，因为系统调用号是从 1 开始的。

kernel > C syscall.c

```
9
10  static char *syscall_names[] = {
11  "", "fork", "exit", "wait", "pipe",
12  "read", "kill", "exec", "fstat", "chdir",
13  "dup", "getpid", "sbrk", "sleep", "uptime",
14  "open", "write", "mknod", "unlink", "link",
15  "mkdir", "close", "trace"};
16
```

9. 在 kernel/proc.c 中 fork 函数调用时，添加子进程复制父进程的 mask 的代码：。

```
  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);
  // copy tracemask
  np->tracemask = p->tracemask;
  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;
```

10. 最后在 Makefile 的 UPROGS 中添加 $U/_trace\ 。

```
198     $U/_primes\
199     $U/_find\
200     $U/_xargs\
201     $U/_trace\
202
```

11. 在终端编译测试。结果如下：

```
$ trace 2147483647 grep hello README
5: syscall trace -> 0
5: syscall exec -> 3
5: syscall open -> 3
5: syscall read -> 1023
5: syscall read -> 968
5: syscall read -> 235
5: syscall read -> 0
5: syscall close -> 0
```

```
$ trace 2 usertests forkforkfork
usertests starting
6: syscall fork -> 7
test forkforkfork: 6: syscall fork -> 8
8: syscall fork -> 9
9: syscall fork -> 10
10: syscall fork -> 11
9: syscall fork -> 12
10: syscall fork -> 13
11: syscall fork -> 14
9: syscall fork -> 15
12: syscall fork -> 16
11: syscall fork -> 17
9: syscall fork -> 18
10: syscall fork -> 19
11: syscall fork -> 20
12: syscall fork -> 21
10: syscall fork -> 22
11: syscall fork -> 23
12: syscall fork -> 24
10: syscall fork -> 25
11: syscall fork -> 26
12: syscall fork -> 27
9: syscall fork -> 28
10: syscall fork -> 29
```

```
10: syscall fork -> 67
11: syscall fork -> 68
12: syscall fork -> -1
9: syscall fork -> -1
10: syscall fork -> -1
OK
6: syscall fork -> 69
ALL TESTS PASSED
$
```

12. 单项检测得分:

```
$ QEMU: Terminated
vale@Puppyyoo:~/xv6-labs-2021$ ./grade-lab-syscall trace
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (0.7s)
== Test trace all grep == trace all grep: OK (0.9s)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (8.0s)
vale@Puppyyoo:~/xv6-labs-2021$
```

1.3 实验中遇到的问题和解决办法

问题 ： mask 命名冲突导致寄存器读取错误。

解决办法 ： 最初将 tracemask 命名为 mask，在实现过程中遇到了一些寄存器读取错误。由于在函数实现和调用过程中，mask 这个名字可能会与其他变量名冲突，导致读取和赋值操作出现错误。通过更改变量名为 tracemask，避免了这种命名冲突问题。

1.4 实验心得

本次实验成功实现了系统调用追踪功能，通过添加新的 trace 系统调用，能够按需控制 追踪特定的系统调用。在实现过程中，通过解决变量命名冲突、确保系统调用编号的正确定义、生 成用户态系统调用接口、正确继承 tracemask 以及完善输出信息，最终达到了预期的实验目标。此 功能在后续实验和开发中将大大提升调试效率和系统理解深度。

2 Sysinfo

2.1 实验目的

1. 添加一个名为 sysinfo 的系统调用，用于收集系统运行信息。
2. 该系统调用需要一个参数：指向 struct sysinfo 的指针（参见 kernel/sysinfo.h）。内核需要填写该结构体的以下字段： freemem 字段应设置为可用内存的字节数，nproc 字段应设置为状态不是 UNUSED 的进程数。

2.2 实验步骤

1. 定义一个系统调用的序号。系统调用序号的宏定义在 kernel/syscall.h 文件中。在 kernel/syscall.h 添加宏定义 SYS_sysinfo 如下：

```
22  #define SYS_close  21
23  #define SYS_trace  22
24  #define SYS_sysinfo  23
```

2. 在 user/usys.pl 文件加入下面的语句：

```
37  entry("sleep");
38  entry("uptime");
39  entry("trace");
40  entry("sysinfo");
```

3. 在 user/user.h 中添加 sysinfo 结构体以及 sysinfo 函数的声明：

```
ser > C user.h
1   struct stat;
2   struct rtcdate;
3   struct sysinfo;
4
```

```
26  int uptime(void);
27  int trace(int);
28  int sysinfo(struct sysinfo *);
29
```

4. 在 kernel/syscall.c 中新增 sys_sysinfo 函数的定义：

```
106  extern uint64 sys_uptime(void);
107  extern uint64 sys_trace(void);
108  extern uint64 sys_sysinfo(void);
109
```

5. 在 kernel/syscall.c 中函数指针数组新增 sysinfo ：

```
130  [SYS_mkdir]   sys_mkdir,
131  [SYS_close]   sys_close,
132  [SYS_trace]   sys_trace,
133  [SYS_sysinfo]    sys_sysinfo,
134  };
135
136
137  static char *syscall_names[] = {
138  "", "fork", "exit", "wait", "pipe",
139  "read", "kill", "exec", "fstat", "chdir",
140  "dup", "getpid", "sbrk", "sleep", "uptime",
141  "open", "write", "mknod", "unlink", "link",
142  "mkdir", "close", "trace","sysinfo"};
```

6. 在 kernel/proc.c 中新增了一个名为 nproc 的函数，用于获取可用进程的数量，代码如下：

```
uint64
nproc(void)
{
    struct proc *p;
    uint64 num = 0;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      // add lock
      acquire(&p->lock);
      // if the processes's state is not UNUSED
      if (p->state != UNUSED)
      {
      // the num add one
        num++;
      }
      // release lock
      release(&p->lock);
    }
    return num;
```

7. 在 kernel/kalloc.c 中添加一个 free_mem 函数，用于收集可用内存的数量。计算未使用内存的字节数 freemem 需遍历该链表，得到链表节点数，并与页表大小（4KB）相乘即可。

```
// Return the number of bytes of free memory
uint64
free_mem(void)
{
    struct run *r;
    uint64 num = 0;
    acquire(&kmem.lock);
    r = kmem.freelist;
    while (r){
        num++;
        r = r->next;
    }
    release(&kmem.lock);
    return num * PGSIZE;
}
```

8. 在 kernel/defs.h 中添加上述两个新增函数的声明：

```
// kalloc.c
void*           kalloc(void);
void            kfree(void *);
void            kinit(void);
unit64          free_mem(void);
```

```
void            yield(void);
int             either_copyout(int user_dst, uint64 dst, void *src, uint6
int             either_copyin(void *dst, int user_src, uint64 src, uint64
void            procdump(void);
unit64          nproc(void);
```

9. 在 kernel/sysproc.c 文件中添加的 sys_sysinfo 函数的具体实现：

首先使用 argaddr 函数读取用户态数据 sysinfo 的指针地址。然 后，将内核中获取的 sysinfo 数据，按 sizeof(info) 大小复制到该指针指向的内存位置。

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

10. 最后在 user 目录下添加一个 sysinfo.c 用户程序：

user > C sysinfo.c

```
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/sysinfo.h"
#include "user/user.h"
int main(int argc, char *argv[])
{
    // param error
    if (argc != 1){
    fprintf(2, "Usage: %s need not param\n", argv[0]);
    exit(1);
        }
    struct sysinfo info;
        sysinfo(&info);
    // print the sysinfo
    printf("free space: %d\nused process: %d\n", info.freemem,
    info.nproc);
    exit(0);
}
```

11. 在 Makefile 的 UPROGS 中添加:

```
195     $U/_zombie\
196     $U/_trace\
197     $U/_sysinfo\
198     $U/_sysinfotest\
```

12. 终端编译，分别运行 sysinfo 与 sysinfotest

结果如下:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ sysinfo
free space: 133386240
used process: 3
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$
```

13. 单项测试得分:

```
vale@Puppyyoo:~/xv6-labs-2021$ ./grade-lab-syscall sysinfo
make: 'kernel/kernel' is up to date.
== Test sysinfotest == sysinfotest: OK (1.8s)
vale@Puppyyoo:~/xv6-labs-2021$
```

2.3 实验中遇到的问题和解决办法

问题：在获取可用内存大小时，访问链表节点可能导致竞争条件。

解决办法：在遍历 kmem.freelist 链表时，加锁以确保线程安全。

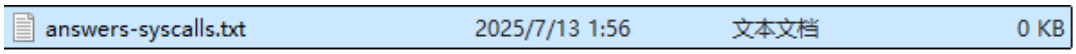
```
acquire(&kmem.lock);
    r = kmem.freelist;
    while (r){
        num++;
        r = r->next;
    }
release(&kmem.lock);
```

2.4 实验心得

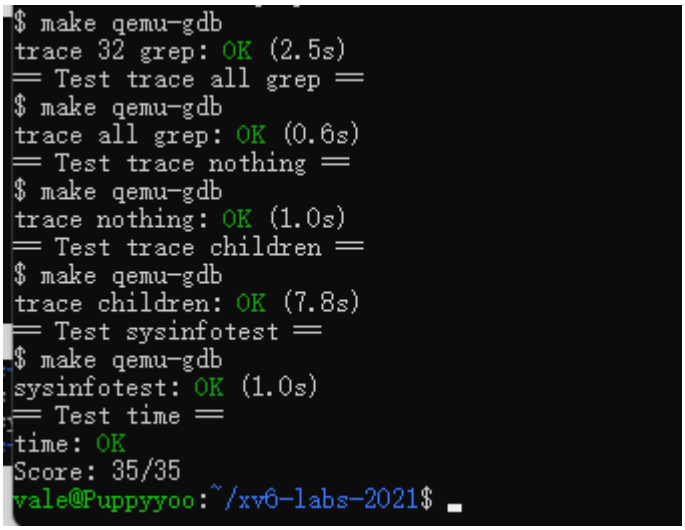
通过本次实验，我学会为系统添加了一个新的系统调用 sysinfo ，实现了收集运行系统的信息， 如可用内存数、进程数等。 在完成实验的过程中，我还了解了实现所需的几个基础数据结构，比如 kmem 链表，以及表示进程 状态的字段 UNUSED 等。要实现这些能够反应系统运行信息功能，首先需要了解这些信息分别由什 么记录，这样才能更有针对性地做出追踪和检测。

3 实验检验得分

1. 在实验目录下创建 time.txt，填写完成实验时间数。
2. 创建 answers-syscalls 文件将程序运行结果填入。



3. 在终端中执行 make grade,得到 lab2 总分：



4. 代码提交