

System Calls 实验报告



Tongji University

专 业：软 件 工 程

指 导 老 师：王 冬 青

年 级：大 学 二 年 级

学 号：2153061

姓 名：谢 嘉 麒

- 1. Before Start
- 2. System call tracing (moderate)
 - 2.1.实验目的
 - 2.2.实验步骤
 - 2.2.1.实现trace系统调用的前提
 - 2.2.2.实现trace系统调用
 - 2.2.3.编译与检测
 - 2.3.实验中遇到的问题和解决方法
 - 2.3.1.阅读内核代码并确定修改位置
 - 2.3.2.学习并理解Xv6的系统调用过程
 - 2.4.实验心得
- 3. Sysinfo
 - 3.1.实验目的
 - 3.2.实验步骤
 - 3.2.1.实现sysinfo系统调用的前提
 - 3.2.2.实现sysinfo系统调用
 - 3.2.3.编写sysinfo.c用户程序
 - 3.2.4.编译与检测
 - 3.3.实验中遇到的问题和解决方法
 - 3.3.1.确定内核修改范围和修改规范
 - 3.3.2.学习调用进程相关函数
 - 3.4.实验心得
- 4.实验总结

1. Before Start

在实验之前，应阅读xv6 book的第二章，以及第四章的4.3，4.4节，并且浏览以下文件：

- (1) 系统调用的用户空间代码在user/user.h和user/usys.pl中；
- (2) 内核空间代码在kernel/syscall.h和kernel/syscall.c中；
- (3) 进程相关代码在kernel/proc.h和kernel/proc.c中；

开启实验需要将git分支切换到syscall，需要在Xv6文件目录下启动终端，执行以下指令：

```
$ git fetch
$ git checkout syscall
$ make clean
```

2. System call tracing (moderate)

2.1.实验目的

本次实验要求实现一个追踪系统调用的功能，这一功能在后续实验的debug过程中会发挥作用。要求笔者创建一个新的trace系统调用来控制追踪。这个系统调用接受一个成为mask的integer型变量，来设置被追踪的系统调用。例如，为了追踪fork系统调用，则调用trace(1<<SYS_fork),其中SYS_fork是来自kernel/syscall.h的系统调用编号。

需要完成在进程调用trace后，若后续调用了fork系统调用，内核会打印: syscall fork -> <ret_valye>的信息，但trace并不应该影响其他进程。

实验提供了一个用户级程序trace（详见user/trace.c）

2.2.实验步骤

2.2.1.实现trace系统调用的前提

加入trace系统调用所需要修改的文件较多，主要需要添加用户空间和内核两个部分的代码，在实现系统调用之前，需要先修改用户空间和内核中的几个文件，为系统调用添加入口和编号，具体方法如下：

- (1) 在Makefile的UPROGS环境变量中加入\$U/_trace\;
- (2) 在user/user.h中加入int trace(int)声明，为trace系统调用提供用户空间的入口；
- (3) 在user/usys.pl中加入entry("trace")，来生成调用入口时在用户空间中执行的汇编代码；
- (4) 在kernel/syscall.h中加入#define SYS_trace 22，为trace规定一个系统调用编号。
- (5) 在kernel/syscall.c中新增sys_trace函数定义；

此时trace系统调用实现的前提基础已经完成，能够通过make qemu编译。

2.2.2.实现trace系统调用

实验核心在于实现trace的系统调用，此时需要对内核代码进行修改，具体步骤如下：

- (1) 根据实验指导，在kernel/proc.h中添加一个新的进程控制块变量，需要在表示进程的数据结构struct proc中，加入int tracemask变量，用来记录被追踪的系统调用；
- (2) 同时浏览kernel/sysproc.c可以知道，需要设置新的函数来讲trace系统调用的参数保存在struct proc中，因此设置sys_trace(void)函数来完成这个任务：

```
uint64
sys_trace(void)
{
    int sys_trace_mask;
    argint(0, &sys_trace_mask);
    myproc()->tracemask = sys_trace_mask;
    return 0;
}
```

函数使用argint函数来获取用户传入系统调用的第一个参数，而利用myproc()来获取当前进程控制块的指针，将sys_trace_mask赋予这一进程控制块的tracemask。

- (3) 接下来实现具体的打印内容。在kernel/syscall.c中添加一个长字符串类型常数组，保存各个系统调用名称：

```
const char *syscallnames[25] = {
    "",
    "fork",
    "exit",
    "wait",
    "pipe",
    "read",
    ...
}
```

```

    "kill",
    "exec",
    "fstat",
    "chdir",
    "dup",
    "getpid",
    "sbrk",
    "sleep",
    "uptime",
    "open",
    "write",
    "mknod",
    "unlink",
    "link",
    "mkdir",
    "close",
    "trace",
    "sysinfo",
};

```

(4) 观察到其上方static uint64 (*syscalls[])(void)也存放了各个系统调用，因此也需要将[SYS_trace] sys_trace加入其中。

(5) 为了打印系统调用内容，不难看出syscall为系统调用函数，因此在其中发生系统调用的部分加入打印代码即可：

```

if (p->tracemask >> num & 1)
{
    printf("%d: syscall %s -> %d\n", p->pid, syscallnames[num], p->trapframe-
>a0);
}

```

(6) 实验要求子进程也能够继承trace的掩码，因此需要定位到内核代码的进程部分的fork系统调用，在其中加入能够让子进程也获取tracemask的语句，语句如下：

```

np->tracemask = p->tracemask;

```

2.2.3.编译与检测

在完成以上步骤后，在Xv6目录下启动终端，执行make qemu指令进行编译，运行 trace 32 grep hello README，可以得到预期输出如下：

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 968
3: syscall read -> 235
3: syscall read -> 0

```

利用实验提供的测评工具，在终端中执行指令./grade-lab-syscall trace，得到测评结果如下：

```
cinderlord@ubuntu:~/xv6-labs-2021$ ./grade-lab-syscall sysinfo
make: 'kernel/kernel' is up to date.
== Test sysinfotest == sysinfotest: OK (2.4s)
(Old xv6.out.sysinfotest failure log removed)
```

2.3.实验中遇到的问题和解决方法

2.3.1.阅读内核代码并确定修改位置

本次实验难点之一在于阅读并理解xv6系统调用的内核代码，并能够据此判断添加trace系统调用的代码修改位置。所幸实验设计者提示可以阅读哪些代码来获取信息（见1 Before start），通过阅读这些代码能够基本推断出实现一个xv6系统调用所需要的代码内容，仿照已有代码进行相同格式的添加即可。

值得注意的是，还需要合理判断trace系统调用涉及哪些模块，如需要了解子进程也能够继承tracemask这一特点，则需要在fork中进行tracemask的复制，否则系统调用会出错。

2.3.2.学习并理解Xv6的系统调用过程

实验目的在于简单引入Xv6的系统调用过程，因此需要笔者额外搜集资料进行进一步的了解和学习。不同系统调用之间可能存在差异，但基本工作原理都接近一致。了解系统调用过程后再进行实验能够使代码编写更加规范，有目的性。

2.4.实验心得

通过本次实验，笔者在浏览，学习Xv6内核和用户态的系统调用相关代码的过程中收获颇丰。基本上了解了系统调用代码的书写逻辑和编码规范，同时也通过实际编写代码来加深印象。

同时笔者也自主学习并了解了Xv6的系统调用工作原理，对于系统有了进一步的认识和理解。

3. Sysinfo

3.1.实验目的

本次实验要求添加一个系统调用sysinfo，这一系统调用能够手机当前运行系统的信息。这一系统调用接受一个参数，是指向struct sysinfo(详见kernel/sysinfo.h)的指针。所实现的内核代码需要填充这个结构体：

- (1) 用空闲内存的大小填充freemem;
- (2) 用状态为UNUSED的进程数量来填充nproc;

实验提供测试程序sysinfotest，实验通过时这一程序会打印“sysinfotest:OK”;

3.2.实验步骤

3.2.1.实现sysinfo系统调用的前提

实现sysinfo系统调用需要先配置基本代码，包括以下几个部分：

- (1) 在Makefile中的UPROGS环境变量中加入 \$U/_sysinfotest ;
- (2) 在user/user.h中加入int sysinfo(struct sysinfo *);声明，为sysinfo系统调用提供用户空间的入口;
- (3) 在user/usys.pl中加入entry("sysinfo"), 来生成调用入口时在用户空间中执行的汇编代码;

(4) 在kernel/syscall.h中加入#define SYS_sysinfo 23, 为sysinfo规定一个系统调用编号。

(5) 在user/user.h中加入struct sysinfo的声明;

(6) 在kernel/syscall.c中新增sys_sysinfo函数定义;

此时trace系统调用实现的前提基础已经完成, 能够通过make qemu编译。

3.2.2.实现sysinfo系统调用

在实现系统调用前提的基础上, 实现具体的系统调用相关代码, 具体操作步骤如下:

(1) 在kernel/syscall.c的static uint64 (*syscalls[])(void)中加入[SYS_trace] sys_trace。

(2) 在2.2.2.3的数组中加入"sysinfo";

(3) 在kernel/kalloc.c中新添加函数freemem_size来获取空闲内存数量, 具体做法为遍历空闲内存链, 实现代码如下:

```
int
freemem_size(void)
{
    struct run *r;
    int num = 0;
    for(r=kmem.freelist;r=r->next)
    {
        num++;
    }
    return num*PGSIZE;
}
```

函数中使用run结构体定义循环变量r, 利用for循环来统计空闲内存块数量, 乘以PGSIZE即得到空闲内存数量;

(4) 在kernel/proc.c中新增函数proc_num来获取可用进程数目, 实现代码如下:

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

函数利用proc结构体定义指针p, 并利用p作为循环变量遍历进程, 如果当前进程的state是UNUSED, 则总数加一。最终函数就可以返回可用进程数目。

(5) 由于新增了两个基础函数, 则需要kernel/defs.h中做出函数声明: 在kernel/defs.h文件中加入int freemem_size(void);和int proc_num(void);

(6) 在kernel/sysproc.c中加入#include "sysinfo.h", 注意此时共有23个系统调用, 同时在此文件下新增sysinfo系统调用的实现代码:

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

系统调用利用sysinfo结构体来存储info信息, 定义addr来存储用户传递的参数地址。利用myproc()函数来获取当前进程的信息, 存储在proc类型的指针变量中。如果argaddr函数返回值小于0, 则表示获取参数地址失败。成功则利用之前实现的函数来获取空闲内存和进程数量, 将其存储在info内容中。利用copyout将info中的内容从内核空间复制到用户空间的addr中, 若返回值小于0, 则复制失败。否则程序成功结束。

3.2.3.编写sysinfo.c用户程序

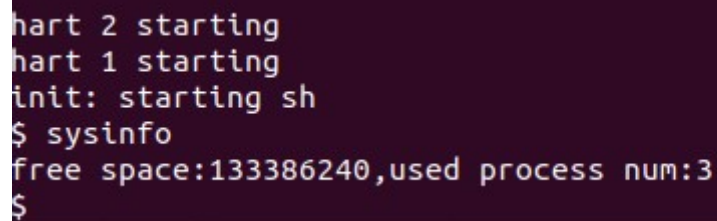
由于这个实验并没有提供用户态程序, 因此需要自己编写, 但这一部分内容并不困难。在user目录下添加sysinfo.c文件, 在main函数中实现用户程序即可:

(1) 首先判断参数数量是否合法, 如果argc不等于1, 则表示不合法, 打印违法信息并退出程序;

(2) 合法则定义sysinfo类型结构体info, 并执行系统调用sysinfo, 将系统信息填充在info中后打印当前空闲空间数量和进程数量即可。

3.2.4.编译与检测

保存代码后进入Xv6目录, 打开终端执行make qemu命令, 编译成功后执行sysinfo, 得到预期输出:



```
hart 2 starting
hart 1 starting
init: starting sh
$ sysinfo
free space:133386240,used process num:3
$
```

执行自动检测程序, 输入命令./grade-lab-syscall sysinfo执行, 得到结果如下:


```
cinderlord@ubuntu:~/xv6-labs-2021$ ./grade-lab-syscall sysinfo
make: 'kernel/kernel' is up to date.
== Test sysinfotest == sysinfotest: OK (2.4s)
(Old xv6.out.sysinfotest failure log removed)
```

3.3.实验中遇到的问题和解决方法

3.3.1.确定内核修改范围和修改规范

涉及到系统内核的部分仍需要谨慎修改，有了前一个实验的基础，这一实验的实现并不复杂，需要修改的部分基本相同。值得注意的是增加函数后记得添加相关的函数声明到头文件，同时在引用结构体和函数时也要注意引入头文件。

3.3.2.学习调用进程相关函数

难点在于学习和调用进程相关函数，并且合理划分内核态和用户态之间的界限。应该明确哪些需要再内核态实现，哪些需要在用户态实现。

3.4.实验心得

通过本次实验，笔者对于内核态和用户态的代码理解和实现有了更进一步的认识，同时也学习了进程，系统调用追踪，内存追踪的相关知识。在阅读了源代码，了解具体的数据结构类型和函数实现后，这些抽象概念能够更加具象化地储存在知识结构中。

4.实验总结

通过以上的两个实验，笔者大体上把握了一个系统调用发生的全部过程。从系统调用的追踪入手，进一步引入内存和进程模块，从内核态和用户态两个视角来理解系统调用的发生。不仅如此，在具体的代码实现过程中，通过阅读源代码和实验参考书，系统调用的代码实现也逐渐明了，使知识体系更加完善具体。