Lab 2: system calls

1950787 杨鑫

1. 实验目的：

- 熟悉系统调用，了解其执行过程
- 编写几个系统调用以实现相应功能
- 通过编写系统调用，对内核的组织结构加深认知和了解

2. 实验步骤：

首先切换分支至 syscall branch，以获取本次实验的内容

同时清理文件，以获得纯净的初始文件系统

```
git checkout syscall
make clean
```

a. System call tracing

实验目的：

在本次实验中，您将添加一个系统调用跟踪功能，该功能可以帮助您在以后的实验中进行调试。 您将创建一个新的 trace 系统调用来控制跟踪。 它应该有一个参数，一个整数“掩码”，其位指定要跟踪的系统调用。 例如，为了跟踪 fork 系统调用，程序调用 trace(1 << SYS_fork)，其中 SYS_fork 是来自 kernel/syscall.h 的系统调用号。 如果掩码中设置了系统调用的编号，则必须修改 xv6 内核以在每个系统调用即将返回时打印一行。 该行应包含进程ID、系统调用的名称和返回值； 您不需要打印系统调用参数。 跟踪系统调用应该启用对调用它的进程以及它随后派生的任何子进程的跟踪，但不应影响其他进程。

实验分析：

为了构建这样一个系统调用，我们需要在内核去写一个sys_trace()函数，它将取出参数然后分析它包含哪些系统调用的编号然后存入一个数据结构中（如数组），然后在之后执行 syscall() 函数执行系统调用时，先判断该系统调用的编号在不在该数组中，如果在则打印其信息即可。同时由于需要在 proc.h 中新加数据结构，所以需要修改 fork() 函数以确保父子进程的该数据结构相同。

然后为了让用户区能调用新加的 trace，需要在 user.h 中增加系统调用声明，在syscall.h 增加新系统调用的编号。同时增加函数指针等一些其他必要的东西。

实验核心代码：

sysproc.c 新加函数 systrace

```
uint64
sys_trace(void)
{
  int n;

  if(argint(0, &n) < 0)
    return -1;
  struct proc *p = myproc();
  char *mask = p->mask;
  int i = 0;
  while (n > 0) {
    if (n % 2 == 0) {
      mask[i++] = '0';
    } else {
      mask[i++] = '1';
    }
    n >>= 1;
  }
  return 0;
}
```

syscall.c 中新加系统调用名字数组（方便打印信息），并且修改 syscall 函数

```
const char* syscallName[22] = {"fork", "exit", "wait", "pipe", "read",
"kill", "exec", "fstat", "chdir",
"dup", "getpid", "sbrk", "sleep", "uptime", "open", "write", "mknod",
"unlink", "link", "mkdir", "close",
"trace"};

void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
    if (strlen(p->mask) > 0 && p->mask[num] == '1') {
      printf("%d: syscall %s -> %d\n", p->pid, syscallName[num - 1], p-
>trapframe->a0);
    }
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

在 proc.c 的 fork() 函数中新增复制 mask 数组代码

```
// copy mask to the child.
safestrcpy(np->mask, p->mask, sizeof p->mask);
```

测试：

```
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 -> 966
3: syscall read -> 70
3: syscall read -> 0
$
```

跟踪 read 系统调用并打印信息

```
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
$
```

跟踪所有系统调用并打印信息

评分：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-syscall trace
make: “kernel/kernel”已是最新。
== Test trace 32 grep == trace 32 grep: OK (2.0s)
== Test trace all grep == trace all grep: OK (0.8s)
== Test trace nothing == trace nothing: OK (0.8s)
== Test trace children == trace children: OK (12.3s)
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

可以通过所有测试

b. Sysinfo

实验目的：

在这个作业中，您将添加一个系统调用 sysinfo，它收集有关正在运行的系统的信息。 系统调用有一个参数：指向 struct sysinfo 的指针（参见 kernel/sysinfo.h）。 内核应填写此结构的字段：freemem 字段应设置为空闲内存的字节数，nproc 字段应设置为状态不是 UNUSED 的进程数。

实验分析：

在 sysproc.c 中编写一个 sys_sysinfo 函数来执行该功能，它在获取空闲内存和进程数后将其放入 sysinfo 结构体中，并通过 copyout 函数返回给用户区。

在 kalloc.c 中编写 freemem 函数来返回空闲内存数，在 proc.c 中编写 proc_num 函数来返回进程数。其中 freemem 用到了 freelist 链表以获取空闲内存数，proc_num 用到了proc 结构体来统计进程数。

添加一些声明和引用（同 **a**）

此外，还可以在用户区写一个 sysinfo 程序来使用该系统调用，这可以方便调试和观察系统调用运行，添加声明同 a 类似，然后新建 sysinfo.c 来调用该系统调用即可。

实验核心代码：

proc.c 中添加 proc_num 函数：

```
int
proc_num()
{
  struct proc *p;
  int count = 0;
  for (p = proc; p < (proc + NPROC); p++) {
    if (p->state != UNUSED) count++;
  }
  return count;
}
```

kalloc.c 中添加 freemem 函数：

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

sysproc.c 中添加 sys_sysinfo 函数：

```
uint64
sys_sysinfo(void)
{
  uint64 st;
  struct proc* p = myproc();

  if(argaddr(0, &st) < 0)
    return -1;

  struct sysinfo si;
  si.freemem = freemem();
  si.nproc = proc_num();

  if (copyout(p->pagetable, st, (char*)&si, sizeof si) < 0) return -1;
  return 0;
}
```

sysinfo.c

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/sysinfo.h"

int
main(int argc, char *argv[])
{
  if(argc != 1){
    fprintf(2, "sysinfo.c wants 0 parameter, but %d parameters provided\n",
argc - 1);
    exit(1);
  }

  struct sysinfo si;
  if (sysinfo(&si) < 0) {
    exit(1);
  }
  printf("free memory nums: %d, process nums: %d\n", si.freemem, si.nproc);
  exit(0);
}
```

测试：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sysinfo
free memory nums: 133382144, process nums: 3
$
```

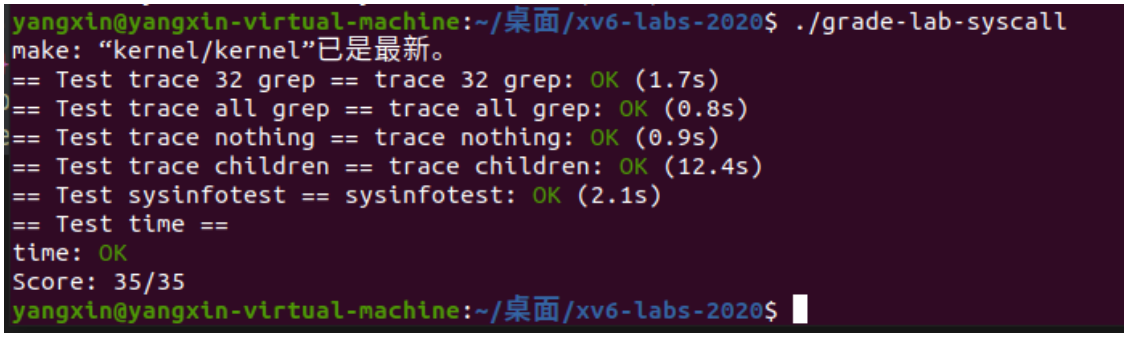
评分：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-syscall sysinf
o
make: “kernel/kernel”已是最新。
== Test sysinfotest == sysinfotest: OK (2.1s)
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

可以通过所有测试

3. 实验评分：

按照实验要求对本次实验的所有小实验进行评分，结果如图：



通过了所有测试

4. 问题以及解决办法：

a.获取内核信息

本次实验的两个小实验都需要获取内核的一些信息：trace 需要获取内核的 mask 数组以知道那些系统调用需要被跟踪；sysinfo 需要获取空闲内存和进程数量，但是在之前还不知道如何从内核中获取这些信息。经过阅读内核中其他函数的源码，发现可以使用 myproc 函数获取当前的 proc 结构体，即表征了当前的内核信息，里面的各种成员对应于一系列信息。所以要获取内核信息时，只需要调用 myproc 函数即可。

b.如何将信息从内核态返回给用户态

从内核获取到所需要的信息后，可以进行一些处理分析得到一些结果。要将结果信息返回，可以通过调用 printf 将信息打印出来（trace 就是这样的方式，在 syscall 函数里面打印出所需要的信息）；或者可以将信息返回给用户态，这就需要使用 copyout 函数：首先获取内核的页表信息 p->pagetable，然后获取虚拟地址，再将他们以及待传送的信息（这里是 sysinfo 结构体)作为参数给 copyout，即可完成内核态到用户态的数据的传输。

5. 实验心得：

1. 本次实验加深了对系统调用的理解和体会，在前一个实验我进行的深入探究实际上就是这次实验的一部分内容，即用户态究竟是如何调用系统调用的，以及如何向内核函数传递参数的
2. 通过手动实现了两个系统调用，以及在内核区增加，修改函数，加深了对内核区文件组织结构的理解
3. 了解了如何在内核区获取相应信息，以及如何在内核态向用户态传送信息