# Lab2：system calls

## System call tracing

### 实验目的：

In this assignment you will add a system call tracing feature that may help you when debugging later labs. You'll create a new trace system call that will control tracing. It should take one argument, an integer "mask", whose bits specify which system calls to trace. For example, to trace the fork system call, a program calls trace(1 << SYS\_fork), where SYS\_fork is a syscall number from kernel/syscall.h. You have to modify the xv6 kernel to print out a line when each system call is about to return, if the system call's number is set in the mask. The line should contain the process id, the name of the system call and the return value; you don't need to print the system call arguments. The trace system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.

添加一个系统调用跟踪功能，该功能可以在以后的实验中为你提供帮助。你将创建一个新的 `trace` 系统调用来控制跟踪。

它应该有一个参数，一个整数`mask(掩码)`，其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出一行。

该行应包含 进程 ID 、 系统调用名称 和 返回值 ；您不需要打印系统调用参数。 trace 系统调用应该为调用它的进程和它随后派生的任何子进程启用跟踪，但不应影响其他进程。

简言之就是实现一个trace系统调用，把握整个流程

### 实验步骤：

根据hints：

1.在makefile中加上

$U/\_trace\

此时运行make qemu，编译器会报错，显示找不到trace函数，因为trace是一个系统调用，我们此时还没有实现它

1. 在user.h中加上系统调用的声明

int trace(int);

1. 在usys.pl中加上entry

entry("trace");

1. 在syscall.h中加上trace的编号

#define SYS\_trace  22

此时运行make qemu，是可以运行起来x86系统的，但是trace函数还是没有实现

1. 在sysproc.c中也加上sys\_trace()函数
2. 在syscall.c中加上sys\_trace的声明

extern uint64 sys\_trace(void);

[SYS\_trace]   sys\_trace,

1. 在proc.h中声明trace\_mask，要利用此结构体存下mask的信息

int trace\_mask;              // for trace

1. 完成sys\_trace()函数的编写

//add sys\_trace() in kernel/sysproc.c

uint64

sys\_trace(void)

{

  int mask;

  if(argint(0, &mask) < 0)

    return -1;

  struct proc \*p = myproc();//获取当前进程

  p->trace\_mask = mask;

  // printf("hi!");

  return 0;

}

在终端运行

trace 32 grep hello README

trace 2147483647 grep hello README

可以看到已经可以追踪到具体的系统调用

但是grep hello README仍然会追踪，仍会把系统调用的信息给存下来

问题出在了proc.c中的freeproc函数里，当我们释放进程时，里面变量的值要重置，所以要在这个函数里加上

  p->trace\_mask = 0;

1. 在fork()函数中加上trace\_mask的父子进程传递

  //copy trace\_mask in the child

  np->trace\_mask = p->trace\_mask;

最后重新编译，在终端写上测试脚本验证

trace 2 usertests forkforkfork

得到的测试通过结果

### 实验中遇到的问题和解决办法：

算是第一次完整地实现一次系统调用，主要是要注意细节。

实验中的报错主要都是漏了某一步没有做，比如没有声明序列号，SYS写成sys等。

实验心得：

从代码上接触了系统调用的实现流程，可以说是更深入内核去理解操作系统，再次感慨一个操作系统中那么多系统调用实现的精妙。

## sysinfo实验

### 实验目的：

In this assignment you will add a system call, sysinfo, that collects information about the running system. The system call takes one argument: a pointer to a struct sysinfo (see kernel/sysinfo.h). The kernel should fill out the fields of this struct: the freemem field should be set to the number of bytes of free memory, and the nproc field should be set to the number of processes whose state is not UNUSED. We provide a test program sysinfotest; you pass this assignment if it prints "sysinfotest: OK".

在本实验中，您将添加一个系统调用 `sysinfo` ，它收集有关正在运行的系统信息。系统调用接受一个参数：一个指向` struct sysinfo `的指针(参见` kernel/sysinfo.h` )。内核应该填写这个结构体的字段：` freemem` 字段应该设置为空闲内存的字节数，` nproc `字段应该设置为状态不是` UNUSED `的进程数。我们提供了一个测试程序` sysinfotest` ；如果它打印 `sysinfotest：OK` ，则实验结果通过测试。

实现一个名为sysinfo的系统调用函数

### 实验步骤：

根据hints：

1. 在makefile中加上以下代码

$U/\_sysinfotest\

此时运行make qemu，编译器会报错，显示找不到sys\_info函数，因为sys\_info是一个系统调用，我们此时还没有实现它

2.在usys.pl中加上entry

entry("sysinfo");

3.在syscall.h中加上trace的编号

#define SYS\_sysinfo  23

4.在user.h中加上sysinfo函数和sysinfo结构体的声明

struct sysinfo;

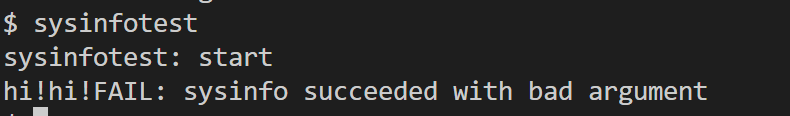
int sysinfo(struct sysinfo\*);

5.在syscall.c中加上

[SYS\_sysinfo]   sys\_sysinfo,

此时运行make qemu，是可以运行起来x86系统的，但是sys\_info函数还是没有实现具体功能

利用sysinfotest进行测试：



接下来实现sys\_sys\_info()函数的具体功能：

在sysproc.c中加上以下代码：

#include "sysinfo.h"

uint64 acquire\_freemem();

uint64 acquire\_nproc();

//add sys\_sysinfo() in kernel/sysproc.c

uint64

sys\_sysinfo(void)

{

  struct sysinfo info;

  uint64 addr;

  struct proc \*p = myproc();

  info.nproc = acquire\_nproc();

  info.freemem = acquire\_freemem();

  if(argaddr(0, &addr) < 0)

    return -1;

  if(copyout(p->pagetable, addr, (char\*)&info, sizeof(info)) <0)

    return -1;

  return 0;

}

在kalloc.c中加上以下代码：

uint64

acquire\_freemem(void)

{

  struct run \*r;

  uint64 cnt = 0;

  acquire(&kmem.lock);

  r = kmem.freelist;

  while(r){

    r = r->next;

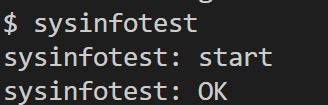
    cnt++;

  }

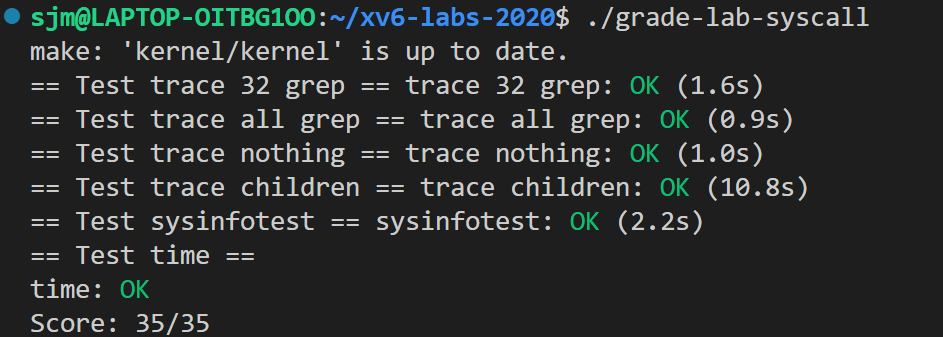
  release(&kmem.lock);

  return cnt \* PGSIZE;

}

此时sysinfo系统调用实现完成，在终端输入make qemu编译并启动xv6后输入sysinfotest指令，可以看到预期输出：  


退出后进行本实验的测试：



可见测试全部通过

### 实验中遇到的困难和解决办法：

注意在实现共有变量的通信时要加一个锁来控制；不同文件使用相同函数要先声明

### 实验心得：

本实验让我掌握了一个新系统调用的添加过程，首先在user/user.h中加入系统调用在用户态的入口，其次在同级目录下的usys.pl中加入在用户态执行的汇编代码，然后在kernel/syscall.h中为系统调用分配一个系统调用的编号，最后深入内核代码，实现系统调用具体的逻辑，包括设计数据结构及其依赖的函数等。

通过这个实验我对xv6的内核进行了深入的剖析，对于系统调用的理解也更加透彻。