Lab2 system calls

1. System call tracing (moderate)

1）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.
> 在本作业中，您将添加一个系统调用跟踪功能，该功能可能会在以后 trace 实验时对您有所帮助。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，要跟踪fork系统调用，程序调用 trace (1<<SYS_fork) ，其中 SYS_fork 是 kernel/syscall.h 中的系统调用编号。如果系统调用的编号在掩码中设置，则必须修改xv6内核，以便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、系统调用的名称和返回值；您不需要打印系统调用参数。 trace 系统调用应启用对调用它的进程及其随后分支的任何子进程的跟踪，但不应影响其他进程。

2）实验步骤

编写代码

在 kernel/syscall.h 文件中定义系统调用号

```
...
#define SYS_close  21
#define SYS_trace  22
```

user/trace.c 文件中调用 trace() 函数
我们需要在 user/user.h 对此声明，可以看出它的参数和返回值均为 int 类型

```
...
int trace(int);
...
```

在 user/usys.pl 文件中加入 trace 系统调用的入口

```
entry("trace");
```

在 kernel/syscall.c 文件中添加系统调用声明

```
extern uint64 sys_trace(void);

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

修改 Makefile 文件，在 UPROGS 项中最后一行添加 $U/_trace\ 使得编译器可以对该文件进行链接编译

```
    ...
        $U/_trace\
```

为了实现 trace() 函数，我们还需要在 struct proc 结构体中添加数据成员才能让当前进程有数据可以传入 trace() 函数中

在 kernel/proc.h 文件中的 struct proc 结构体中添加数据成员 mask

```
struct proc {
  ...
  int mask;
};
```

并且在 kernel/proc.c 文件中对 fork() 函数进行修改，创建子进程的时候也需要将父进程的 mask 继承

```
int
fork(void)
{
  ...
  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
  // 继承父进程的mask
  np->trace_mask = p->trace_mask;

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  ...
}
```

在 kernel/sysproc.c 文件中实现 sys_trace() 函数

```
uint64
sys_trace(void)
{
        int n;
        // 获取追踪的mask 取a0寄存器的值给mask
        if (argint(0, &n) < 0)
                return -1;
        // 将mask保存在本地进程的proc中
        myproc()->trace_mask = n;
        return 0;
}
```

kernel/syscall.c 文件中定义系统调用名称数组

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

修改 kernel/syscall.c 文件中的 syscall() 函数

```
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]();
        // 判断mask是否覆盖当前调用号
        if ((1 << num) & p->trace_mask)
                printf("%d: syscall %s -> %d\n", p->pid, syscall_list[num], p->trapframe->a0);
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

测试程序

进入QEMU模拟器

```
$ make qemu
```

进行 trace 程序的测试

```
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
$ 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
```

键入 Ctrl+a ， 松开，然后键入 x ， 退出xv6系统，并进行单元测试

```
root@LAPTOP-UER420HO:~/lab2_syscall# ./grade-lab-syscall trace
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.1s)
== Test trace all grep == trace all grep: OK (0.9s)
== Test trace nothing == trace nothing: OK (1.1s)
== Test trace children == trace children: OK (9.1s)
```

3）实验中遇到的问题和解决方法

跟着提示做没什么问题，就是在理解上需要下功夫，见实验心得

4）实验心得

本次实验对添加系统调用有了一定的认识，基于以下流程

- 在用户程序中也就是 user 目录下添加的程序中调用系统调用
- 在 user/user.h 添加系统调用的函数原型声明
- 在 user/usys.pl 文件中添加入口函数，这是perl脚本，可以生成 usys.S 文件
- usys.S 文件中设置了a7寄存器存储系统调用号，ecall指令进行系统调用
- 然后转到 kernel/syscall.c 文件中的 syscall() 函数通过添加的系统调用号找到 kernel/sysproc.c 文件中的对应添加的函数并执行

我们可以看到 user/usys.pl 文件中的内容，perl语言可以生成汇编语言 usys.S 的文件，即用户态进行系统调用的接口

以下为 user/usys.pl 入口函数

```
sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}
```

其中

ecall 将权限提升到内核模式，即进入内核态，并将程序跳转到指定的地址，

li 将立即数加载到寄存器，上面代码即为将系统调用号放入 a7 寄存器

可以看到 kernel/syscall.c 文件中的 syscall() 函数中，从当前进程中的 trapframe 中取出 a7 寄存器中的系统调用号，并且执行

```
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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

我们进一步看看 trapframe 这个数据结构中存储了什么东西

根据在 kernel/proc.h 文件中的定义以及网上查询的资料

trapframe 用于在操作系统内核中保存处理中断、异常或系统调用时的CPU上下文。具体包含了一些重要的寄存器和状态信息，以便在中断或异常发生时能够恢复到正确的执行状态

```
struct trapframe {
  /*   0 */ uint64 kernel_satp;   // kernel page table
  /*   8 */ uint64 kernel_sp;     // top of process's kernel stack
  /*  16 */ uint64 kernel_trap;   // usertrap()
  /*  24 */ uint64 epc;           // saved user program counter
  /*  32 */ uint64 kernel_hartid; // saved kernel tp
  /*  40 */ uint64 ra;
  /*  48 */ uint64 sp;
  /*  56 */ uint64 gp;
  /*  64 */ uint64 tp;
  /*  72 */ uint64 t0;
  /*  80 */ uint64 t1;
  /*  88 */ uint64 t2;
  /*  96 */ uint64 s0;
  /* 104 */ uint64 s1;
  /* 112 */ uint64 a0;
  /* 120 */ uint64 a1;
  /* 128 */ uint64 a2;
  /* 136 */ uint64 a3;
  /* 144 */ uint64 a4;
  /* 152 */ uint64 a5;
  /* 160 */ uint64 a6;
  /* 168 */ uint64 a7;
  /* 176 */ uint64 s2;
  /* 184 */ uint64 s3;
  /* 192 */ uint64 s4;
  /* 200 */ uint64 s5;
  /* 208 */ uint64 s6;
  /* 216 */ uint64 s7;
  /* 224 */ uint64 s8;
  /* 232 */ uint64 s9;
  /* 240 */ uint64 s10;
  /* 248 */ uint64 s11;
  /* 256 */ uint64 t3;
  /* 264 */ uint64 t4;
  /* 272 */ uint64 t5;
  /* 280 */ uint64 t6;
};
```

同时，我在做实验的时候还有一个问题，这些系统调用都没有显式的传递参数的方法，那么参数是怎么获得的呢

我们可以看到在 `kernel/sysproc.c` 文件中对系统调用的实现，会使用 `argint()` 函数获得参数，它可以读取在 `a0-a5` 的寄存器中传递的系统调用的参数

2. Sysinfo (moderate)

1）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".
> 在这个任务中，您将添加一个系统调用 `sysinfo`，用于收集有关正在运行的系统的信息。系统调用采用一个参数：指 `struct sysinfo` 的指针（请参阅 `kernel/sysinfo.h`）。内核应该填写此结构的字段：`freemem` 字段应该设置为可用内存的字节数，`nproc` 字段应该设为 `state` 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”，那么您就通过了这个任务。

2）实验步骤

编写代码

修改 `Makefile` 文件，在 `UPROGS` 项中最后一行添加 `$U/_sysinfotest\` 和 `$U/_sysinfo\` 使得编译器可以对该文件进行链接编译

```
...
$U/_sysinfotest\
$U/_sysinfo\
```

在 `user/user.h` 文件中声明 `struct sysinfo` 和 `sysinfo()` 函数

```
struct stat;
struct rtcdate;
struct sysinfo;
...
int uptime(void);
int trace(int);
int sysinfo(struct sysinfo*);
...
```

在 `kernel/syscall.h` 文件中定义系统调用号

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

在 user/usys.pl 文件中加入 sysinfo 系统调用的入口

```
entry("sysinfo");
```

在 kernel/syscall.c 文件中添加系统调用声明

```
extern uint64 sys_sysinfo(void);

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

在 kernel/syscall.c 文件中系统调用名称数组中添加 sysinfo

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

在 kernel/proc.c 文件中添加获得可用进程数目的函数 nproc()

```
// 获取当前可用的进程数目
uint64
nproc(void)
{
        struct proc* p;
        uint64 ans = 0;
        for (p = proc; p < &proc[NPROC]; ++p) {
                // 加锁
                acquire(&p->lock);
                if (p->state != UNUSED)
                        ++ans;
                release(&p->lock);
        }
        return ans;
}
```

在 kernel/defs.h 文件中添加上述函数声明

```
...
uint64                  nproc(void);
...
```

在 kernel/kalloc.c 文件中添加当前可用内存空间大小函数 free_mem()

```
// 返回当前可用内存空间大小
uint64
free_mem(void)
{
        struct run* p;
        uint64 ans = 0;
        acquire(&kmem.lock);
        p = kmem.freelist;
        while (p) {
                ++ans;
                p = p->next;
        }
        release(&kmem.lock);
        return ans * PGSIZE;
}
```

在 kernel/defs.h 文件中添加上述函数声明

```
// kalloc.c
...
void            kinit(void);
uint64                  free_mem(void);
```

在 kernel/sysproc.c 文件中实现 sys_sysinfo() 函数

```
...
#include "sysinfo.h"
...
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;
}
```

在 `user/sysinfo.c` 中调用上述系统调用

```
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/sysinfo.h"
#include "user/user.h"

int main(int argc, char* argv[]) {
        if (argc != 1) {
                fprintf(2, "Usage: %s do not need arg", argv[0]);
                exit(1);
        }
        struct sysinfo info;
        sysinfo(&info);
        printf("free memory: %d\nused processes: %d\n", info.freemem, info.nproc);
        exit(0);
}
```

测试程序

进入QEMU模拟器

```
$ make qemu
```

进行 `sysinfo` 程序的测试

```
$ sysinfo
free memory: 133386240
used processes: 3
$ sysinfotest
sysinfotest: start
sysinfotest: OK
```

键入 Ctrl+a ， 松开，然后键入 x ， 退出xv6系统，并进行单元测试

```
root@LAPTOP-UER420HO:~/lab2_syscall# ./grade-lab-syscall
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.1s)
== Test trace all grep == trace all grep: OK (0.9s)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (9.2s)
== Test sysinfotest == sysinfotest: OK (2.0s)
== Test time ==
time: OK
Score: 35/35
```

3）实验中遇到的问题和解决方法

没什么问题

4）实验心得

本次实验和上一个实验差不多，但是更加复杂，需要自定义结构体并且了解一些关于内存块和进程数量的概念

其中xv6系统中的所有进程存放在一个数组里面 proc[NPROC] 我们可以遍历整个数组，从而找到所有可用的进程，并统计。在这里需要注意的一点是在查看进程的状态即 p->state 的时候，我们需要给进程加锁，保证执行该程序的进程访问这个变量的时候不被其他进程可能的修改影响

同理，xv6系统的内存块以统计在一个叫做 freelist 的链表中，我们也可以遍历该链表统计空闲页面的个数，在返回空间大小的同时注意乘上页面大小 PGSIZE

在实现 sys_sysinfo() 函数的时候，我了解到我们需要使用 argaddr() 函数用户程序的命令行参数的地址，从而可以调用 copyout() 函数从内核态将数据传输回用户态

总结下来，更加熟悉了用户态和内核态进入和返回的过程和调用的函数