Lab2 SYSTEM CALL

- 姓名：吴欣怡
- 学号：PB21051111
- 虚拟机用户名：OS-PB21051111

System call tracing

实验分析

分析实验文档中的要求：
1.向 user.h 中添加声明
2.向 usys.pl 中添加存根
3.向 syscall.h 中添加宏定义
4.向 sysproc.c 中添加 sys-trace()函数，实现对于新的系统调用，能够记住输入的指令
5.修改 proc.c 中的 frok()来把掩码从父进程复制到子进程
6.修改 syscall.c 中的 syscall()来打印除 trace

实验过程

第一步（包含实验分析中的 1、2、3）

在 user/user.h 中添加 trace 这个系统调用的声明，首先需要理解 trace.c 的类型，接收和输出的参数类型。trace 函数接收至少两个命令行参数，且第二个命令行参数要为大于等于 0 的整数。接着传递第三个命令行参数作为要执行的命令名称，而后面的参数作为该命令的参数。交给 exec 执行。执行结束后正常退出程序。所以推测出这个 trace 函数应该需要接收一个整数。所以在 user.h 中添加：

```
void trace(int);
```

向 usys.pl 中添加存根:
其实不太清楚是要做什么，但是仿照着写了

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

在 kernel/syscall.h 中添加宏定义：
参考前面的格式，新增一行：

```
#define SYS_trace  22
```

第二步（包含实验分析中的 4）

向 sysproc.c 中添加 sys_trace()函数，实现对于新的系统调用，能够记住输入的指令：
首先查看了 sysproc.c 文件中其他函数的定义，应该都是给出系统 wait(),exit()等函数时的执行规则。那么这里 sys_trace()同理，也要做类似的操作。但是还不知道要怎么写函数的内容，所以按照实验说明中的提示，查看 syscall.c 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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

其中 struct proc *p = myproc() 语句获取了当前正在运行的进程（myproc() 定义在 proc.c 中，用于返回当前进程的指针），然后从该进程的 trapframe 中提取了系统调用号。把系统调用号存储在寄存器 a7 中。
后面的部分首先检查系统调用号是否有效，必须大于 0、小于系统调用数组的大小 NELEM(syscalls)，并且相应的系统调用函数指针必须存在于 syscalls[num]。如果这些条件都满足，它会调用相应的系统调用函数，并将返回值存储在寄存器 a0 中，否则报错。

给 proc.h 中 porc 结构体的定义中增加一个整数型变量 mask，表示各个进程的掩码，便于后面 sys_trace 的追踪后把掩码信息保存在结构体中。
kernel/sysproc.c 中的 set_trace 定义为：

```
uint64
sys_trace(void)
{
  int n;
  //检验是否取到了a0寄存器的参数
  if(argint(0, &n) < 0)
    return -1;
  //获取指针
  myproc()->mask=n;
  return 0;

}
```

第三步（包含实验分析中的 5）

修改 proc.c 中的 fork 函数，增加把掩码从父进程复制到子进程的一步。

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
```

```
  if((np = allocproc()) == 0){
    return -1;
  }

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

  np->parent = p;

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

  safestrcpy(np->name, p->name, sizeof(p->name));

  np->mask=p->mask;//这一步把掩码复制到子进程

  pid = np->pid;

  np->state = RUNNABLE;

  release(&np->lock);

  return pid;
}
```

第四步（包含实验分析中的 6）

修改 kernel/syscall.c:
参考其他声明，添加 sys_trace 声明

```
extern uint64 sys_trace(void);
```

查看 static uint64 (*syscalls[])(void)，其中定义了一系列系统调用函数指针，那么 sys_trace 也需要加入函数指针数组 *syscalls[]中，格式为：

```
[SYS_trace]   sys_trace,
```

基本内容补上了，下面来看我们需要打印的内容：
结合之前对 syscall 函数的分析，这个打印输出的部分应该加在

```
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
    \\补充trace部分
}
```

打印内容类似于 407: syscall fork -> 408
即&pid: syscall 系统调用名 -> &trapframe a0 寄存器的内容
因为 proc 结构体里的 name 是整个线程的名字，不是函数调用的函数名称，所以我们不能直接使用 p->name ，而要自己定义一个数组，把系统调用名按顺序保存起来，以 num 作为索引找到对应名字。在判定输出条件时，若相应的位在 p->mask 中被设置，表示当前进程允许记录这个系统调用，就可以打印。具体程序如下：

```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();
  char* syscalls_name[23] = {"", "fork", "exit", "wait", "pipe", "read", "kill",
"exec","fstat", "chdir", "dup", "getpid", "sbrk", "sleep", "uptime","open", "write",
"mknod", "unlink", "link", "mkdir", "close", "trace"};
  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
    if(p->mask &(1<<num))
    {
        //参考num = p->trapframe->a7以及后面错误信息的打印仿照着写。
        printf("%d: syscall %s -> %d\n",p->pid,syscalls_name[num],p->trapframe-
>a0);}
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

sysinfo

实验分析

1.前面的步骤和 trace 相同（向 user.h 中添加声明，向 usys.pl 中添加存根，向 syscall.h 中添加宏定义）
2.在 kernel/proc.c 中添加函数获取正在使用的进程(非 UNUSED 状态下的)
3.kernel/kalloc.c 中添加函数可用的内存数，将添加的函数声明在 defs.h

实验过程

第一步

user.h 声明：

```
struct sysinfo;
 int sysinfo(struct sysinfo *);
```

usys.pl:

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

syscall.h:

```
#define SYS_sysinfo  23
```

在之前 tracing 相关的部分也要把 sysinfo 添加到系统调用名的数组中。

第二步

首先阅读 kernel/proc.c 和 proc.h 中的内容:
proc.h 中给出了结构体 proc 的定义，
enum procstate state; 定义了 state,可以用来筛选出 UNUSED 状态下的进程。
proc.c 中的 struct proc proc[NPROC];定义了一个包含多个进程的数组 proc[NPROC]。
根据实验说明，NPROC 是所有进程的总数量。仿照 proc.c 中的 allocproc()函数，可以在 proc.c 中添加函数 size()为：

```
int size()
{
    struct proc* p;

    int count=0;
    for (p=proc;p<&proc[NPROC];p++)
    {
        acquire(&p->lock);
        if (p->state!=UNUSED)
            count++;
        release(&p->lock);
    }
    return count;
}
```

第三步

kernel/kalloc.c 中添加函数可用的内存数，将添加的函数声明在 defs.h
查看 kalloc.c 文件，理解内容：
run 定义了一个链表，维护当前可用的内存。当分配内存（kalloc）时，从链表中移除一个页面。当释放内存（kfree）时，将页面重新添加到链表中。
所以要获取所有可用的内存时，只需要顺着 run 链表往后走，找出所有可用的即可。PGSIZE 是单个内存页的大小。
在 kalloc.c 中添加函数 freememory():

```
int freememory()
{
    struct run *r;
    acquire(&kmem.lock);
    r = kmem.freelist;
    int num=0;
    while(r)
    {
        num++;
        r=r->next;
    }
    release(&kmem.lock);
    return num*PGSIZE;
}
```

把这两个函数加入到 defs.h 文件中

```
int size(void);
int freememory(void);
```

第四步

编写 kernel/sysproc.c 文件中的 sysinfo()函数

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

其中 copyout()函数的定义在 kernel/vm.c 中查看：
参数：进程页表，用户态目标地址，数据源地址，数据大小
返回值：数据大小
具体使用方法参考了 kernel/file.c 中的 filestat() 函数。
实际的 sysinfo()的效果是，先用 argaddr 函数读进来要保存的在用户态的数据 sysinfo 的指针地址，然后再把从内核里得到的 sysinfo 形式的内容以 sizeof(info) 大小的的数据复制到指针上。

实验评分

```
ubuntu@VM7782-OS-PB21051111:/home/ubuntu/桌面/xv6-labs-2020$ python3 grade-lab-s
yscall
make: “kernel/kernel”已是最新。
== Test trace 32 grep == trace 32 grep: OK (2.2s)
== Test trace all grep == trace all grep: OK (1.3s)
== Test trace nothing == trace nothing: OK (1.3s)
== Test trace children == trace children: OK (19.2s)
== Test sysinfotest == sysinfotest: OK (3.2s)
== Test time ==
time: OK
Score: 35/35
```

实验总结

在实验过程中，在一些细节的地方犯了错。比如在 trace 部分，仿照报错信息写的输出信息，在系统调用名处写的是 p->name 导致了奇怪的输出。还有在 freememory 函数中，最后的返回值没有用 num*PGSIZE 而是单纯地只考虑了内存页的个数 num，导致输出答案与正确答案差距很大。写好了 freememory()、size()函数后，没有在 defs.h 文件中声明，导致没法在 sysproc.c 中正常使用。还有与锁有关的问题，最开始先试了一下不写锁相关的 acquire 和 release 操作，发现评分文件还是会给出满分的成绩。但是这样做可能会造成竞争或者数据的破坏和冲突。
感觉到去按照指示把相关的文件中的函数还有一些内置的功能都理解下来很重要。例如对 copyout 这一函数使用方法的学习。