lab2令人心碎，尽管这个lab用时最短。

(实际上运行结果是正确的，但是测试的话超时了，我目前仍未找到合适的解决办法)

```
ubuntu@VM-16-9-ubuntu:~/xv6-labs-2021$ grade-lab-syscall
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (4.1s)
== Test trace all grep == trace all grep: OK (3.2s)
== Test trace nothing == trace nothing: OK (3.1s)
== Test trace children == Timeout! trace children: FAIL (30.2s)
    ...
         init: starting sh
         $ trace 2 usertests forkforkfork
         usertests starting
         3: syscall fork -> 4
         qemu-system-riscv64: terminating on signal 15 from pid 1993899 (make)
    MISSING '^5: syscall fork -> \d+'
    QEMU output saved to xv6.out.trace_children
== Test sysinfotest == sysinfotest: OK (14.5s)
== Test time ==
time: OK
Score: 30/35
```

系统调用的通用注册流程。

1. 在 `user/usys.pl` 中添加一行 `entry("trace")` ,当在用户程序中使用系统调用时，会通过这个perl脚本的预设操作提高特权等级(`ecall`)。
2. 在 `user/user.h` 中声明 `int trace(int);`
3. 在 `kernel/syscall.h` 中添加一行 `#define SYS_trace 22`

> 到这里,要停一下。在最开始，我们设置了一个 `perl` 脚本， `Makefile` 会调用 `perl` 脚本 (`user/usys.pl`)，生成一个 `user/usys.S` ，这个文件就像一个switch语句一样，当你使用一个系统调用的时候，比如read，
>
> 操作系统会查询 `user/usys.S` 中对应的标签(`target`)来执行相应的语句。

```
.global read
read:
 li a7, SYS_read
 ecall
 ret
```

语句大同小异，即将SYS_read这个名称放入a7，然后使用 `ecall` ， 请求提升硬件权限。

此时，进入内核，内核会执行syscall()，这个函数会检验 `a7` 中的参数是否合法。如果合法，就执行对应的系统调用。(`kernel/syscall.c:145`),执行完毕之后，将返回值放在 `a0` 中。

内核是如何判断系统调用是否合法呢?

实际上所有的系统调用都被放在一个数组syscall[]中，只要 `num < len(syscall)` 那么内核就会认为这是个合法的系统调用，我们在 `kernel/syscall.h` 中添加一行 `#define SYS_trace 22` ,实际上相当于在数组中添加一个新元素。

4. 在 `kernel/sysproc.c` 中写下 `sys_trace` 的函数实现。
5. 在 `kernel/syscall.c` 中添加 `extern uint64 sys_trace(void);`
6. 在 `kernel/syscall.c` 的数组中

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

以上便是系统调用的通用注册流程。

sys_trace(void)

由于 `kernel` 与普通进程的页表并不相同。所以内核要获得进程中的数据就需要通过类似 `argint` 这样的操作。

想要将系统调用的结果提供给普通进程，也需要通过像 `copyout` 这样的操作

```
uint64 sys_trace(void){
  int n;
  if (argint(0, &n) < 0) {
      return -1;
  }
  myproc()->mask = n;
  return 0;
}
```

为了完成第一个实验，

1. 在 kernel/proc.h 中的 struct proc 中添加一个跟踪变量mark
2. 编写 sys_trace 这个实际很简单，参考其他程序，填充 struct proc 中的mask变量的值就可以了。
3. 修改 kernel/syscall.c 中的函数 void syscall()，使系统调用在执行时可以打印出对应信息。注意，在此之前，要确保完成**通用注册流程**。

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

void syscall(){
    ...
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        p->trapframe->a0 = syscalls[num]();
        if (p->mask & (1 << num)) {
                printf("%d: syscall %s -> %d\n",p->pid,syscallsname[num],p->trapframe->a0);
    }
  }
    ...
}
```

4. 最后，添加 $U/_trace 到 Makefile 中的 UPROGS 中

sysinfo

1. 如通用流程所示，按步骤完成。
2. 在 `kernel/kalloc.c` 中编写程序 `freemem()`
3. 在 `kernel/proc.c` 中编写程序 `nproc()`
4. 将这两个函数在 `kernel/defs.h` 中声明
5. 添加 `$U/_sysinfotest` 到 `Makefile` 中的 `UPROGS` 中

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


uint64 nproc(void){
  int cnt = 0;
  int i;
  for (i = 0; i < NPROC; i++){
    if (proc[i].state ==UNUSED)
        cnt++;
  }
  return (NPROC - cnt);
}


uint64 freemem(void) {
  uint64 num = 0;
  struct run *p = kmem.freelist;
  while (p) {
    num++;
    p = p->next;
  }
  return num * 4096;
}
```