Syscall

实验内容

1. trace

目标是实现具有系统调用跟踪功能的trace系统调用，它可以打印系统调用信息。

接口如下：

```
trace mask cmd ..
```

其中第一个参数mask是一个无符号整数，以状压形式传入需要追踪的系统调用集合（比如要追踪编号为i的系统调用可以令`mask |= (1 << i)`），第二个参数是需要运行的命令，trace会打印命令运行过程中的系统调用信息。

实现分为如下几个步骤：

- 在`kernel/proc.h`的进程结构体`struct proc`定义中加入mask的声明`int tracemask;`
- 在`kernel/proc.c`的`fork()`实现中加上子进程继承父进程mask的机制：`np->tracemask = p->tracemask;`
- 在`kernel/syscall.h`中为trace编号：`#define SYS_trace  23`
- 在`kernel/syscall.c`中加上trace的相关声明，并加入一个存放系统调用名称的字符串数组（为了打印名称）：`static char *syscallnames[];`
- 在`kernel/syscall.c`中的`syscall`里实现打印逻辑：

```
arg0 = p->trapframe->a0;
p->trapframe->a0 = syscalls[num]();
if ((p->tracemask >> num) & 1) { // check if the num-th bit is set
  printf("%d: sys_%s(%d) -> %d\n", p->pid, syscallnames[num], arg0, p->trapframe->a0);
}
```

- 在`kernel/sysproc.c`里实现trace系统调用，解析传入的mask置入当前进程的`tracemask`：

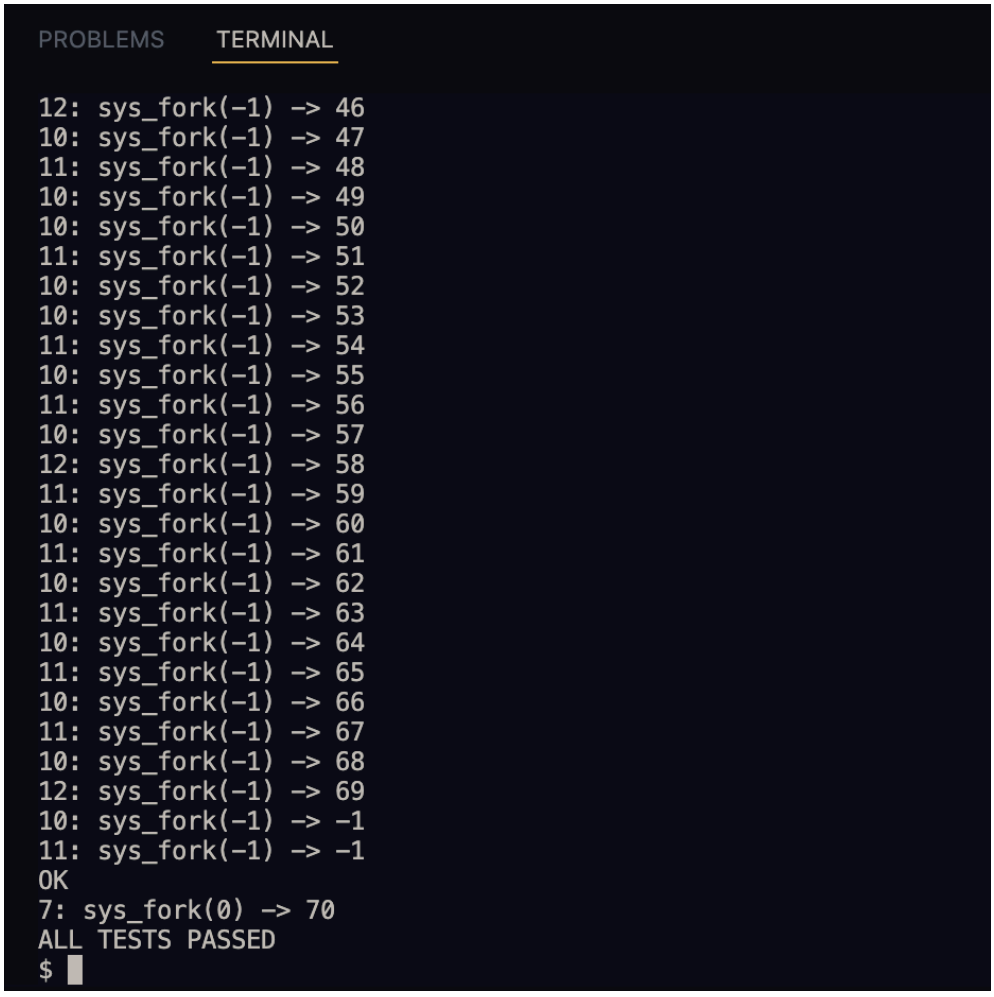
```
uint64 sys_trace(void) {
  if(argint(0, &(myproc()->tracemask)) < 0) // use argument to set mask
    return -1;
  return 0;
}
```

- 在`Makefile`、`user/user.h`和`user/user.pl`中加入trace的相关声明。

运行结果：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
3: sys_read(3) -> 1023
3: sys_read(3) -> 966
3: sys_read(3) -> 70
3: sys_read(3) -> 0
$ trace 2147483647 grep hello README
4: sys_trace(2147483647) -> 0
4: sys_exec(12240) -> 3
4: sys_open(12240) -> 3
4: sys_read(3) -> 1023
4: sys_read(3) -> 966
4: sys_read(3) -> 70
4: sys_read(3) -> 0
4: sys_close(3) -> 0
$ grep hello README
$
```



2. sysinfo

目标是实现具有收集xv6运行信息功能的sysinfo系统调用。sysinfo接受一个`struct sysinfo *`作为参数，它会给结构体填上对应的值：当前剩余的内存字节数freemem和状态为UNUSED的进程个数nproc。

实现上我用了两个定义在kernel内的辅助函数`memamt`和`proccnt`分别实现freemem和nproc的计算。分为如下几个步骤：

- 在`kernel/def.h`内加入两个辅助函数的声明：

```
uint64          memamt(void);
uint64          proccnt(void);
```

- 在`kernel/kalloc.c`中实现`memamt`：遍历`kmem.freelist`（记录空页的链表）

```
uint64 memamt(void) {
  struct run *r;
  uint64 amt;
  amt = 0;
  for (r = kmem.freelist; r; r = r->next) { // traverse the linked-list
    amt += PGSIZE;
  }
  return amt;
}
```

- 在`kernel/proc.c`中实现`proccnt`：遍历`proc`（记录进程的数组）

```
uint64 proccnt(void) {
  struct proc *p;
  int cnt;
  cnt = 0;
  for (p = proc; p < &proc[NPROC]; p++) {
    if(p->state != UNUSED)
      cnt++;
  }
  return cnt;
}
```

- 在`kernel/syscall.h`中为sysinfo编号：`#define SYS_sysinfo 24`
- 在`kernel/syscall.c`中加上sysinfo的相关声明
- 在`kernel/sysinfo.c`内实现`sys_sysinfo`，其中`copyout`用于把填好的结构体复制到用户地址空间：

```
uint64 sys_sysinfo(void) {
  uint64 addr;
  struct proc *p = myproc();
  struct sysinfo info;
  if (argaddr(0, &addr) < 0) // read address from trapframe
    return -1;
  info.freemem = memamt();
  info.nproc = proccnt();
  if (copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
      return -1;
  return 0;
}
```

- 在`Makefile`、`user/user.h`和`user/user.pl`中加入sysinfo的相关声明。

运行结果：

问答题

1. 简述`trace`全流程

用户在shell执行`trace`后，系统会trap into内核态，转换为一个`sys_trace`调用，它会把当前进程的`tracemask`设置成`trapframe`中传入的参数。在接下来命令的执行过程中，一旦`syscall`内判断条件达成，就会输出系统调用信息。

2. 解释`kernel/syscall.h`作用

该文件中定义了各个系统调用的编号，也就是它们的内部表示。trap时系统会把系统调用编号放到某个寄存器上（这里是`a7`），在`kernel/syscall.c`里的`syscall`内读出系统调用编号并以`syscalls[num]()`执行调用。这种方式避免了直接对字符串传递和比较，更加干净。`#include<kernel/syscall.h>`即可访问这些编号定义。

3. 解释`syscall`如何根据系统调用号调用对应的系统调用处理函数

如 2 中所述，它会从`trapframe`的`a7`寄存器中读出系统调用号，然后以`syscalls[num]()`执行调用：系统调用号调用对应的系统调用处理函数已经在函数指针数组`syscalls`数组中定义好了。`syscall`把返回值放在`trapframe`的寄存器`a0`。

4. 传递系统调用参数

`kernel/syscall.c`中`argint`、`argaddr`和`argstr`用于传递参数。`argraw`函数用于读当前进程的`trapframe`指定寄存器的原始值（用unsigned i64表示），只能读`a0`到`a5`，用于参数传递的六个寄存器。