**Lab 2**

**Part A: System call tracing**

首先在相关文件中添加宏定义、用户态函数定义等，让qemu可以成功编译trace.c。在proc.h中proc结构定义部分添加uint64型变量trace\_mask。

在sysproc.c中的sys\_trace部分模仿该文件其他函数，通过myproc()函数获得指向当前进程的结构体指针后，将存储在寄存器a0中的参数存储至proc结构体中的trace\_mask，并返回0表示函数调用成功。

**uint64**

**sys\_trace**(void)

{

int no;

struct **proc** \*p = **myproc**();

**argint**(0, &no);

p->trace\_mask = no;

return 0;

}

在proc.c中fork()函数部分np lock上锁期间按照如下方式修改，以使子进程获取trace\_mask同状态更改一并成为原子操作，防止因竞争产生数据错误。

**acquire**(&np->lock);

np->state = RUNNABLE;

np->trace\_mask = p->trace\_mask;

**release**(&np->lock);

syscall部分，先按照提示定义从宏定义数值到系统调用函数的反响映射，之后通过num变量和syscall函数及宏定义进行系统函数调用，并将返回值存储在当前进程的a0寄存器中。随后按照提示以标准输出的方式进行系统信息打印，相关代码如下。

static char \*syscall\_names[] = {

"fork", "exit", "wait", "pipe", "read", "kill", "exec", "fstat", "chdir", "dup", "getpid", "sbrk",

"sleep", "uptime", "open", "write", "mknod", "unlink", "link", "mkdir", "close", "trace"

};

if(num > 0 && num < **NELEM**(syscalls) && syscalls[num]) {

*// Use num to lookup the system call function for num, call it,*

*// and store its return value in p->trapframe->a0*

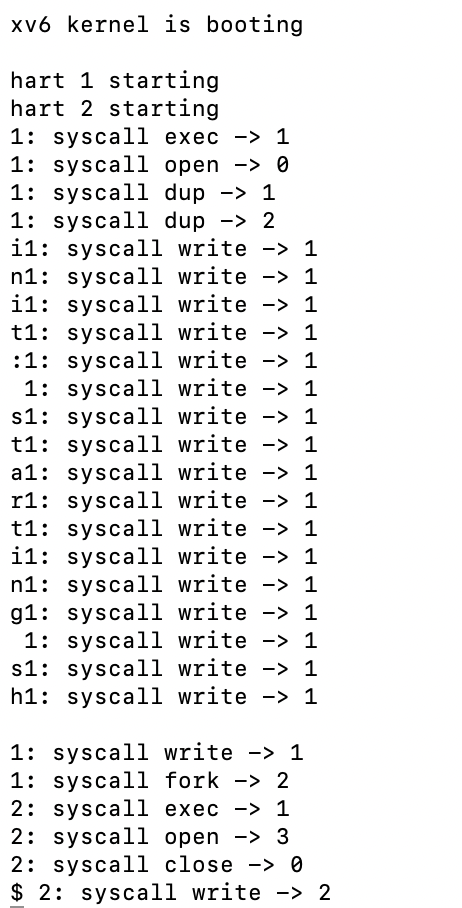
p->trapframe->a0 = syscalls[num]();

**printf**("%d: syscall %s -> %d\n", p->pid, syscall\_names[num-1],p->trapframe->a0);

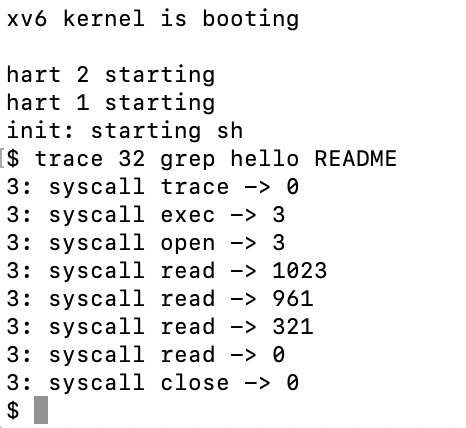
}

}

完成上述工作后，运行make qemu后出现如下输出：



可以看出，内核态在没有用户态文件请求的情况下，将所有进程信息一并输出（从每次输出一个字符都会紧接着打印syscall write中可以看出这点）。因此对打印新增条件判断。添加条件p->trace\_mask > 0，运行输出结果如下。



添加条件p->trace\_mask & (1<<num)。以下是最终代码。

void

**syscall**(void)

{

int num;

struct **proc** \*p = **myproc**();

num = p->trapframe->a7;

if(num > 0 && num < **NELEM**(syscalls) && syscalls[num]) {

*// Use num to lookup the system call function for num, call it,*

*// and store its return value in p->trapframe->a0*

p->trapframe->a0 = syscalls[num]();

if (p->trace\_mask > 0 && (p->trace\_mask & (1<<num))){

**printf**("%d: syscall %s -> %d\n", p->pid, syscall\_names[num-1], p->trapframe->a0);

}

}

else {

**printf**("%d %s: unknown sys call %d\n",

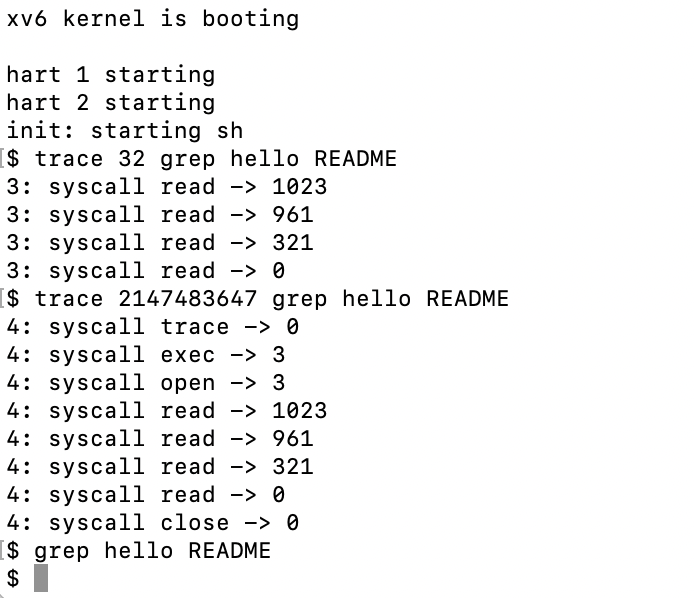
p->pid, p->name, num);

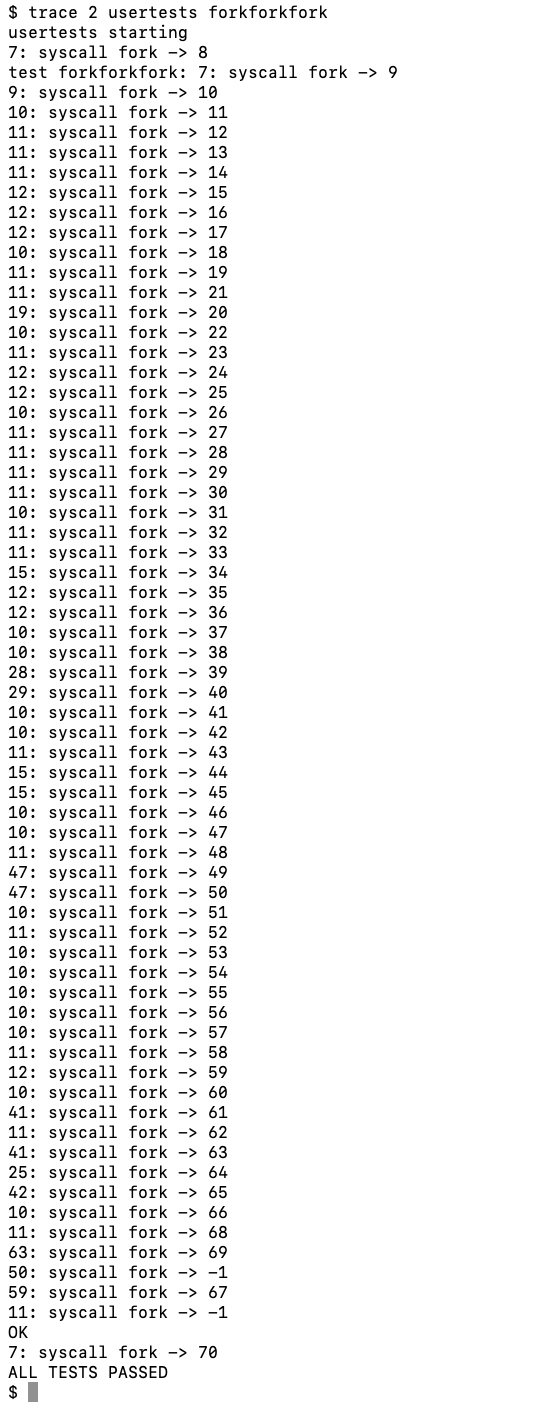
p->trapframe->a0 = -1;

}

}

重新运行项目，得到正确结果。





**Part B: Sysinfo**

根据提示，与Part A一样在user.h等相关文件中添加定义。在proc.c末尾定义函数get\_proc\_num()，利用指向proc结构体的工作指针p遍历当前进程列表，每遍历一项判断当前进程状态是否为UNUSED，若不是则将返回值增加1。以上过程应当通过操作进程内置的锁变量构成原子操作，否则无法保证p指针的移动和对p状态判断的执行先后顺序，可能导致对某些进程状态的重复统计。代码如下。

**uint64**

**get\_proc\_num**(void){

struct **proc** \*p;

int num\_proc = 0;

for (p = proc; p < &proc[**NPROC**]; p++){

**acquire**(&p->lock);

if (p->state != UNUSED) num\_proc++;

**release**(&p->lock);

}

return num\_proc;

}

在kalloc.c末尾定义函数free\_memory()，只需在给kmem.lock上锁的情况下遍历freelist链表即可。若出现多进程并行的情况，其他进程可能会更改freelist的内容（增、删等），因此执行顺序将影响最终统计结果正确与否。值得一提的是，上锁部分内部不需要再额外上锁，因为此处循环体展开后的执行顺序不影响最终结果，只要没有其他进程对freelist做改变。

**uint64**

**free\_memory**(void){

struct **run** \*r;

int num\_page = 0;

**acquire**(&kmem.lock);

r = kmem.freelist;

while(r){

num\_page += 1;

r = r->next;

}

**release**(&kmem.lock);

return num\_page \* **PGSIZE**;

}

完成以上工作后，在sysproc.c中添加函数sys\_sysinfo()。从a0寄存器获取用户态传入的sysinfo结构体指针，并通过新定义的sysinfo结构变量si及先前完成的两个函数获取需要的信息。之后，模仿file.c使用copyout函数将si的信息存至当前进程的页表及用户态使用的结构体中，完成信息转移。另外，根据题目要求在sysinfo.h中定义结构stu\_num，在其中存储字符串变量ID，并在调用sys\_sysinfo函数最后在标准输出中打印学号及提示信息。

**uint64**

**sys\_sysinfo**(void)

{

**uint64** addr;

**argaddr**(0, &addr);

struct **sysinfo** si;

si.freemem = **free\_memory**();

si.nproc = **get\_proc\_num**();

struct **proc** \*p = **myproc**();

if (**copyout**(p->pagetable, addr, (char \*)&si, sizeof(si)) < 0) return -1;

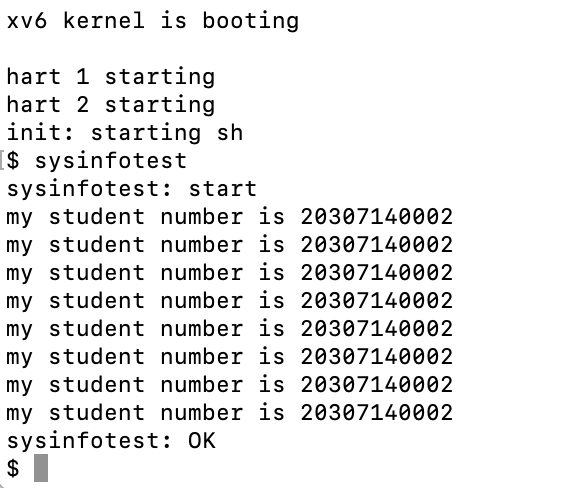
struct **stu\_num** sn;

sn.ID = "20307140002";

**printf**("my student number is %s\n", sn.ID);

return 0;

}



测试结果见上。

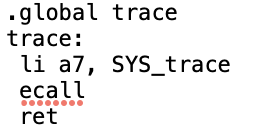
**Questions**

1. **System calls Part A 部分，简述一下 trace 全流程。**

进入qemu后运行trace脚本，trace.c将先在用户态下运行，直到脚本最后调用exec()函数进行相应的系统调用。之后，系统进入内核态（将当前工作路径更改为xv6-labs-2022/kernel），并执行syscall()函数。之后，syscall按照Part A中介绍的方式，先执行被调用的系统函数，将返回值输出到当前进程的a0寄存器中，然后经过判断输出系统调用信息。

1. **kernel/syscall.h是干什么的，如何起作用的？**

syscall.h将一系列系统函数名通过宏定义的方式定义为表征正整数的关键字，从而在应当执行的系统函数与正整数之间建立一一映射的函数关系，因此syscall.c可以通过用户态下传入寄存器a7的操作数来决定需要执行的系统函数，并将结果存储与寄存器a0。以trace为例，可参考usys.S中相应汇编代码：



1. **命令“trace 32 grep hello README”中的trace字段是用户态下的还是实现的系统调用函数trace？**

用户态trace.c文件。Xv6的Makefile中的可执行文件均在user路径下。用户态启动trace.c编译生成的可执行文件后，通过exec()系统调用系统函数trace进入kernel模式，继而完成trace工作。

**Thoughts**

该实验完成了对系统调用函数的追踪以及进程信息查询，加深了对操作系统如何在用户态与内核态状态下交替运行的理解。