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：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
1: syscall exec -> 1
1: syscall open -> 0
1: syscall dup -> 1
1: syscall dup -> 2
i1: syscall write -> 1
n1: syscall write -> 1
i1: syscall write -> 1
t1: syscall write -> 1
:1: syscall write -> 1
 1: syscall write -> 1
s1: syscall write -> 1
t1: syscall write -> 1
a1: syscall write -> 1
r1: syscall write -> 1
t1: syscall write -> 1
i1: syscall write -> 1
n1: syscall write -> 1
g1: syscall write -> 1
 1: syscall write -> 1
s1: syscall write -> 1
h1: syscall write -> 1

1: syscall write -> 1
1: syscall fork -> 2
2: syscall exec -> 1
2: syscall open -> 3
2: syscall close -> 0
$ 2: syscall write -> 2
```

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。

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
3: syscall trace -> 0
3: syscall exec -> 3
3: syscall open -> 3
3: syscall read -> 1023
3: syscall read -> 961
3: syscall read -> 321
3: syscall read -> 0
3: syscall close -> 0
$
```

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。

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 961
3: syscall read -> 321
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 -> 961
4: syscall read -> 321
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$
```

```
$ trace 2 usertests forkforkfork
usertests starting
7: syscall fork -> 8
test forkforkfork: 7: syscall fork -> 9
9: syscall fork -> 10
10: syscall fork -> 11
11: syscall fork -> 12
11: syscall fork -> 13
11: syscall fork -> 14
12: syscall fork -> 15
12: syscall fork -> 16
12: syscall fork -> 17
10: syscall fork -> 18
11: syscall fork -> 19
11: syscall fork -> 21
19: syscall fork -> 20
10: syscall fork -> 22
11: syscall fork -> 23
12: syscall fork -> 24
12: syscall fork -> 25
10: syscall fork -> 26
11: syscall fork -> 27
11: syscall fork -> 28
11: syscall fork -> 29
11: syscall fork -> 30
10: syscall fork -> 31
11: syscall fork -> 32
11: syscall fork -> 33
15: syscall fork -> 34
12: syscall fork -> 35
12: syscall fork -> 36
10: syscall fork -> 37
10: syscall fork -> 38
28: syscall fork -> 39
29: syscall fork -> 40
10: syscall fork -> 41
10: syscall fork -> 42
11: syscall fork -> 43
15: syscall fork -> 44
15: syscall fork -> 45
10: syscall fork -> 46
10: syscall fork -> 47
11: syscall fork -> 48
47: syscall fork -> 49
47: syscall fork -> 50
10: syscall fork -> 51
11: syscall fork -> 52
10: syscall fork -> 53
10: syscall fork -> 54
10: syscall fork -> 55
10: syscall fork -> 56
10: syscall fork -> 57
11: syscall fork -> 58
12: syscall fork -> 59
10: syscall fork -> 60
41: syscall fork -> 61
11: syscall fork -> 62
41: syscall fork -> 63
25: syscall fork -> 64
42: syscall fork -> 65
10: syscall fork -> 66
11: syscall fork -> 68
63: syscall fork -> 69
50: syscall fork -> -1
59: syscall fork -> 67
11: syscall fork -> -1
OK
7: syscall fork -> 70
ALL TESTS PASSED
$
```

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;
}
```

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
my student number is 20307140002
my student number is 20307140002
my student number is 20307140002
my student number is 20307140002
my student number is 20307140002
my student number is 20307140002
my student number is 20307140002
my student number is 20307140002
sysinfotest: OK
$
```

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。

2. **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：

```
.global trace
trace:
 li a7, SYS_trace
 ecall
 ret
```

3. **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。