

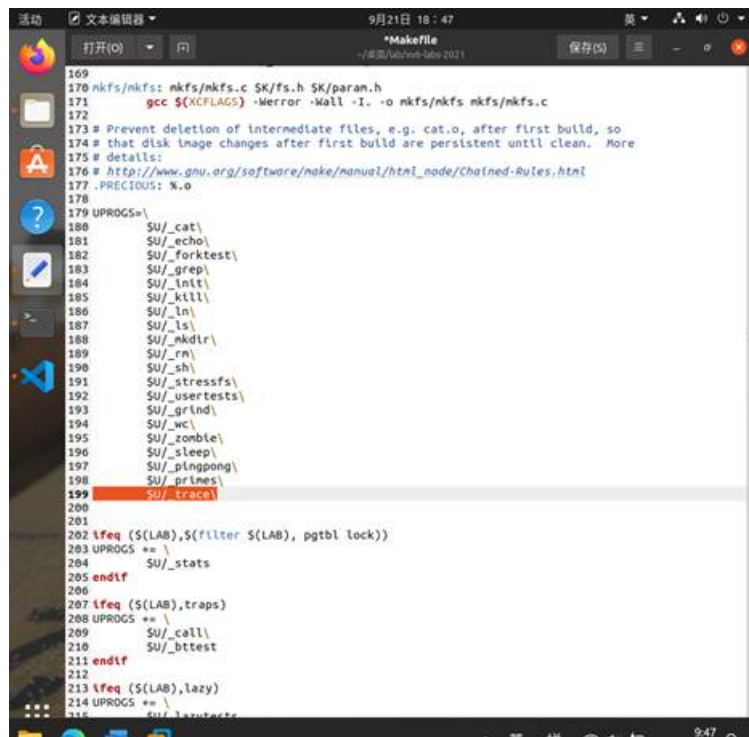
## 一. 实验前准备工作

- 切换git仓库分支为lab2

```
vajackye@ubuntu:~/桌面/lab/xv6-labs-2021$ git checkout syscall
M
Makefile
分支 'syscall' 设置为跟踪来自 'origin' 的远程分支 'syscall'。
切换到一个新分支 'syscall'
vajackye@ubuntu:~/桌面/lab/xv6-labs-2021$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out kernel/kernel fs.img \
mkfs/mkfs .gdbinit \
user/usys.S \
user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln us
er/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind
user/_wc user/_zombie user/_sleep user/_pingpong user/_primes \
ph barrier
vajackye@ubuntu:~/桌面/lab/xv6-labs-2021$
```

## 二. 实验过程

- 实验1: `trace()` 的新系统调用
  - ① 先在Makefile添加 `$U/_trace\` 表明要编译该文件。



- ② 在xv6根目录下运行 `make qemu` , 得到目前还无系统调用的trace接口的定

```
vajackye@ubuntu: ~/桌面/lab/xv6-labs-2021
user/pingpong.o user/ulib.o user/usys.o user/printf.o user/umalloc.o
riscv64-linux-gnu-objdump -S user/_pingpong > user/pingpong.asm
riscv64-linux-gnu-objdump -t user/_pingpong | sed '1,/SYMBOL TABLE/d; s/ .* / /; /
/^$/d' > user/pingpong.sym
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_SYSCA
LL -DLAB_SYSCALL -MD -mmodel=medany -ffreestanding -fno-common -nostdlib -mno-r
elax -I. -fno-stack-protector -fno-pie -no-pie -c -o user/primes.o user/primes
.c
riscv64-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_primes u
ser/primes.o user/ulib.o user/usys.o user/printf.o user/umalloc.o
riscv64-linux-gnu-objdump -S user/_primes > user/primes.asm
riscv64-linux-gnu-objdump -t user/_primes | sed '1,/SYMBOL TABLE/d; s/ .* / /; /
/^$/d' > user/primes.sym
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_SYSCA
LL -DLAB_SYSCALL -MD -mmodel=medany -ffreestanding -fno-common -nostdlib -mno-r
elax -I. -fno-stack-protector -fno-pie -no-pie -c -o user/trace.o user/trace.c
user/trace.c: In function 'main':
user/trace.c:17:7: error: implicit declaration of function 'trace' [-Werror=impli
citt-function-declaration]
   17 |     if (trace(atol(argv[1])) < 0) {
       |         ^
cc1: all warnings being treated as errors
make: *** [<内置>: user/trace.o] 错误 1
vajackye@ubuntu: ~/桌面/lab/xv6-labs-2021$
```

- ③ 根据xv6官方实验文档的指引，修改 user/user.h， user/usys.pl, kernel/syscall.h 这三个文件以添加系统调用trace的声明 (prototype)，存根 (stub)，调用号 (syscall number)。其中，添加声明为 int trace(int) 可通过 user/trace.c 的比较判断 trace(atol(argv[1]))<0

```
1 struct stat;
2 struct rtcdate;
3
4 // system calls
5 int fork(void);
6 int exit(int) __attribute__((noreturn));
7 int wait(int*);
8 int pipe(int*);
9 int write(int, const void*, int);
10 int read(int, void*, int);
11 int close(int);
12 int kill(int);
13 int exec(char*, char**);
14 int open(const char*, int);
15 int mknod(const char*, short, short);
16 int unlink(const char*);
17 int fstat(int fd, struct stat*);
18 int link(const char*, const char*);
19 int mkdir(const char*);
20 int chdir(const char*);
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int trace(int);
27
28 // ulib.c
29 int stat(const char*, struct stat*);
30 char* strcpy(char*, const char*);
31 void *memmove(void*, const void*, int);
32 char* strchr(const char*, char c);
33 int strcmp(const char*, const char*);
34 void fprintf(int, const char*, ...);
35 void printf(const char*, ...);
36 char* gets(char*, int max);
37 uint strlen(const char*);
38 void* memset(void*, int, uint);
39 void* malloc(uint);
40 void free(void*);
41 int atoi(const char*);
42 int memcmp(const void *, const void *, uint);
43 void *memcpy(void *, const void *, uint);
44
45 #!/usr/bin/perl -w
46
47 # Generate usys.S, the stubs for syscalls.
48
49 print "Generated by usys.pl - do not edit\n";
50
51 print "include \"kernel/syscall.h\"\n";
52
53 sub entry {
54     my $name = shift;
55     print "global $name\n";
56     print "$name:\n";
57     print "    li a7, SYS_$name\n";
58     print "    ecall\n";
59     print "    ret\n";
60 }
61
62 entry("fork");
63 entry("exit");
64 entry("wait");
65 entry("pipe");
66 entry("read");
67 entry("write");
68 entry("close");
69 entry("kill");
70 entry("exec");
71 entry("open");
72 entry("mknod");
73 entry("unlink");
74 entry("fstat");
75 entry("link");
76 entry("mkdir");
77 entry("chdir");
78 entry("dup");
79 entry("getpid");
80 entry("sbrk");
81 entry("sleep");
82 entry("uptime");
83 entry("trace");
```

```
1 // System call numbers
2 #define SYS_fork 1
3 #define SYS_exit 2
4 #define SYS_wait 3
5 #define SYS_pipe 4
6 #define SYS_read 5
7 #define SYS_kill 6
8 #define SYS_exec 7
9 #define SYS_fstat 8
10 #define SYS_chdir 9
11 #define SYS_dup 10
12 #define SYS_getpid 11
13 #define SYS_sbrk 12
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_trace 22
```

- ④ 添加完系统调用trace的原型，存根，调用号之后，再次执行make qemu，发现xv6正常启动，但是仍旧是无法执行trace.c的执行文件，因为上述过程并没有进行内部代码实现。

```
vajackye@ubuntu: ~/桌面/lab/xv6-labs-2021
lax -I. -fno-stack-protector -fno-pie -no-pie -c -o user/trace.o user/trace.c
iscv64-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_trace us
r/trace.o user/ulib.o user/usys.o user/printf.o user/umalloc.o
iscv64-linux-gnu-objdump -S user/_trace > user/trace.asm
iscv64-linux-gnu-objdump -t user/_trace | sed '1,/SYMBOL TABLE/d; s/ / /; / ^
/d' > user/trace.sym
kfs/nkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_ln
t user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs use
r/_ustests user/_grind user/_wc user/_zombie user/_sleep user/_pingpong user/_
rimes user/_trace
meta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 95
total 1000
alloc: first 696 blocks have been allocated
alloc: write bitmap block at sector 45
emu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
-nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-devl
e,drive=x0,bus=virtio-mmio-bus.0

v6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh

hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
3 trace: unknown sys call 22
trace: trace failed
$
```

- ⑤ 因此，下一步，我们需要在 kernel/sysproc.c，kernel/proc.h，kernel/syscall.c 内添加实现代码，并修改 kernel/proc.c 的 fork() 源码，进而可跟踪子进程，修改 kernel/syscall.c 的 syscall() 源码，输出跟踪信息。

- kernel/syscall.c:
  - 我们通过跟踪usys.S生成的汇编语言，找到ecall指令跳转执行的地方 syscall.c 的 syscall(void) 函数，跟踪发现，从a7寄存器取出值，并调用静态函数 syscall[num]()，把返回值存在a0，根据文档提示，这里的返回值就是跟踪信息。查看 syscall[num]() 的内容，猜测是需要补充trace的内容，仿照 syscall[num]()的格式补充trace内容。

```

91 extern uint64 sys_fork(void);
92 extern uint64 sys_fstat(void);
93 extern uint64 sys_getpid(void);
94 extern uint64 sys_kill(void);
95 extern uint64 sys_link(void);
96 extern uint64 sys_mkdir(void);
97 extern uint64 sys_mknod(void);
98 extern uint64 sys_open(void);
99 extern uint64 sys_pipe(void);
100 extern uint64 sys_read(void);
101 extern uint64 sys_sbrk(void);
102 extern uint64 sys_sleep(void);
103 extern uint64 sys_unlink(void);
104 extern uint64 sys_wait(void);
105 extern uint64 sys_write(void);
106 extern uint64 sys_uptime(void);
107 extern uint64 sys_trace(void);
108
109 static uint64 (*syscalls[])(void) = {
110 [SYS_fork] sys_fork,
111 [SYS_exit] sys_exit,
112 [SYS_wait] sys_wait,
113 [SYS_pipe] sys_pipe,
114 [SYS_read] sys_read,
115 [SYS_kill] sys_kill,
116 [SYS_exec] sys_exec,
117 [SYS_fstat] sys_fstat,
118 [SYS_chdir] sys_chdir,
119 [SYS_dup] sys_dup,
120 [SYS_getpid] sys_getpid,
121 [SYS_sbrk] sys_sbrk,
122 [SYS_sleep] sys_sleep,
123 [SYS_uptime] sys_uptime,
124 [SYS_open] sys_open,
125 [SYS_write] sys_write,
126 [SYS_mknod] sys_mknod,
127 [SYS_unlink] sys_unlink,
128 [SYS_link] sys_link,
129 [SYS_mkdir] sys_mkdir,
130 [SYS_close] sys_close,
131 [SYS_trace] sys_trace,
132 };
133

```

#### ■ kernel/proc.c:

```

60 /* 128 */ uint64 a1;
61 /* 128 */ uint64 a2;
62 /* 136 */ uint64 a3;
63 /* 144 */ uint64 a4;
64 /* 152 */ uint64 a5;
65 /* 160 */ uint64 a6;
66 /* 168 */ uint64 a7;
67 /* 176 */ uint64 s2;
68 /* 184 */ uint64 s3;
69 /* 192 */ uint64 s4;
70 /* 200 */ uint64 s5;
71 /* 208 */ uint64 s6;
72 /* 216 */ uint64 s7;
73 /* 224 */ uint64 s8;
74 /* 232 */ uint64 s9;
75 /* 240 */ uint64 s10;
76 /* 248 */ uint64 s11;
77 /* 256 */ uint64 t3;
78 /* 264 */ uint64 t4;
79 /* 272 */ uint64 t5;
80 /* 280 */ uint64 t6;
81 };
82
83 enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
84
85 // Per-process state
86 struct proc {
87     struct spinlock lock;
88
89     // p->lock must be held when using these:
90     enum procstate state; // Process state
91     void *chan; // If non-zero, sleeping on chan
92     int killed; // If non-zero, have been killed
93     int xstate; // Exit status to be returned to parent's wait
94     int pid; // Process ID
95     int mask; // Specify the syscall
96
97     // wait_lock must be held when using this:
98     struct proc *parent; // Parent process
99
100     // these are private to the process. so n->lock need not be held.

```

#### ■ kernel/sysproc.c:

```

//trace code
uint64
sys_trace(void)
{
    int mask;
    //取得a0寄存器的值返回给mask
    if(argint(0, &mask) < 0)
        return -1;
    //将mask传给目前进程的mask
    myproc()->mask = mask;
    return 0;
}

```

- `kernel/syscall.c`: 字符数组的添加, 并打印

```
//定义数组判断输出功能是不是mask规定的输出函数
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"
};

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]();
        //修改部分
        if((1 << num) & p->mask) {
            printf("%d: syscall %s -> %d\n", p->pid,
                syscall_names[num], p->trapframe->a0);
        }
    } else {
        printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}
```

- `kernel/proc.c` 的 `fork()` 修改:

```
pid = np->pid;
//子进程复制父进程的mask
np->mask = p->mask;
release(&np->lock);
```

- ⑥ 最后, 执行: `trace 32 grep hello README` 和 `trace 2147483647 grep hello README` 和 `grep hello README` 和 `trace 2 usertests forkforkfork` (最后出现: `ALL TESTS PASSED`)



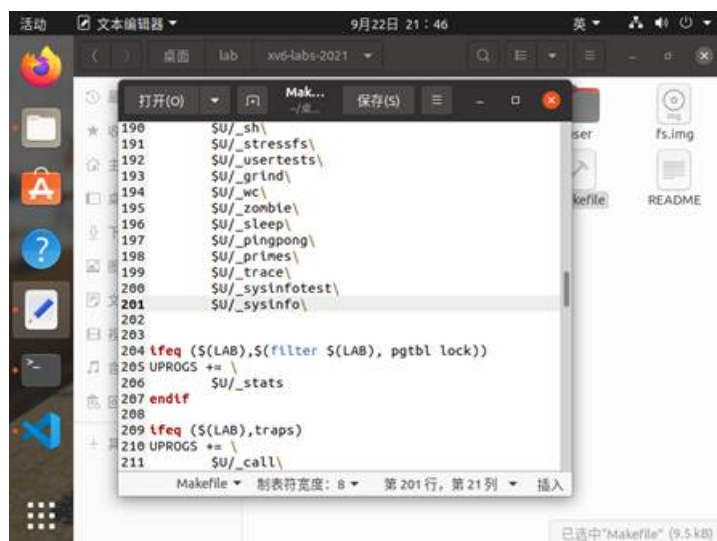
```

hart 1 starting
hart 2 starting
init: starting sh
5 trace 32 grep hello README
8: syscall read -> 1023
8: syscall read -> 968
8: syscall read -> 235
8: syscall read -> 0
5 trace 2147483647 grep hello README
exec trace failed
5 trace 2147483647 grep hello README
5: syscall trace -> 0
5: syscall exec -> 3
5: syscall open -> 3
5: syscall read -> 1023
5: syscall read -> 968
5: syscall read -> 235
5: syscall read -> 0
5: syscall close -> 0
5 grep hello README
5 trace 2 usertests forkforkfork
usertests starting
7: syscall fork -> 8
test forkforkfork: 7: syscall fork -> 9
9: syscall fork -> 10
11: syscall fork -> 12
11: syscall fork -> 13
10: syscall fork -> 11
12: syscall fork -> 14
11: syscall fork -> 15
12: syscall fork -> 16
13: syscall fork -> 17
12: syscall fork -> 18
13: syscall fork -> 19
12: syscall fork -> 20
12: syscall fork -> 21
10: syscall fork -> 22
20: syscall fork -> 24
10: syscall fork -> 25
12: syscall fork -> 23
12: syscall fork -> 27
10: syscall fork -> 28
11: syscall fork -> 26

```

## • 实验2: `sysinfo()` 的新系统调用

- ①查看实验要求, 了解 `sysinfo()` 系统调用的功能: 是收集有关正在运行的系统信息。接受一个参数: 指向 `struct sysinfo` 的指针。内核应填写该结构体的字段: `freemem` 字段设置为空闲内存字节数, `nproc` 字段设置为状态非 `UNUSED` 的进程数目。当 `sysinfotest` 测试程序打印: `sysinfotest:OK` 则实验成功。
- ②第一步, 先将编译配置写入 `Makefile`: `$U/_sysinfotest`, `$U/_sysinfo`:



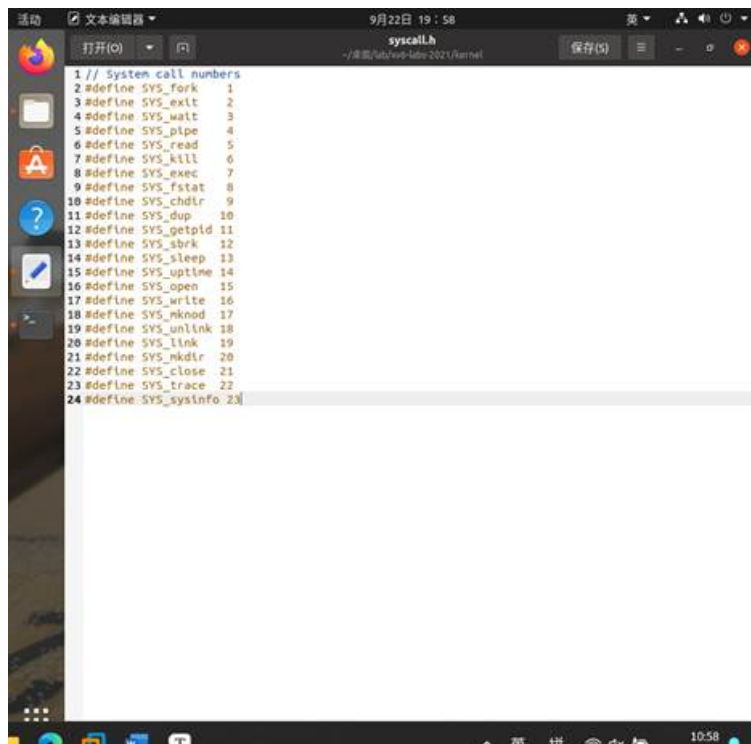
- ③第二步, 和 `trace()` 相同, 运行 `make qemu` 发现编译器无法编译 `user/sysinfotest.c`, 需要在 `user/user.h` 声明 `sysinfo()` 原型, 在 `user/usys.p` 添加存根, 在 `kernel/syscall.h` 添加系统调用号。

```
活动 文本编辑器 9月22日 19:56 user.h
~/桌面/lab/week-lab01-2021/user
保存(S)

1 struct stat;
2 struct rtdat;
3 struct sysinfo;
4
5 // system calls
6 int fork(void);
7 int exit(int) __attribute__((noreturn));
8 int wait(int*);
9 int pipe(int*);
10 int write(int, const void*, int);
11 int read(int, void*, int);
12 int close(int);
13 int kill(int);
14 int exec(char*, char**);
15 int open(const char*, int);
16 int mknod(const char*, short, short);
17 int unlink(const char*);
18 int fstat(int fd, struct stat*);
19 int link(const char*, const char*);
20 int mkdir(const char*);
21 int chdir(const char*);
22 int dup(int);
23 int getpid(void);
24 char* sbrk(int);
25 int sleep(int);
26 int uptime(void);
27 int trace(int);
28 int sysinfo(struct sysinfo*);
29
30 // ulib.c
31 int stat(const char*, struct stat*);
32 char* strcpy(char*, const char*);
33 void *memcpy(void*, const void*, int);
34 char* strchr(const char*, char c);
35 int strcmp(const char*, const char*);
36 void fprintf(int, const char*, ...);
37 void printf(const char*, ...);
38 char* gets(char*, int max);
39 uint strlen(const char*);
40 void* memset(void*, int, uint);
41 void* malloc(uint);
42 void free(void*);
43 int atoi(const char*);
44 int memcpy(const void*, const void*, uint);
45 void *memcpy(void*, const void*, uint);
```

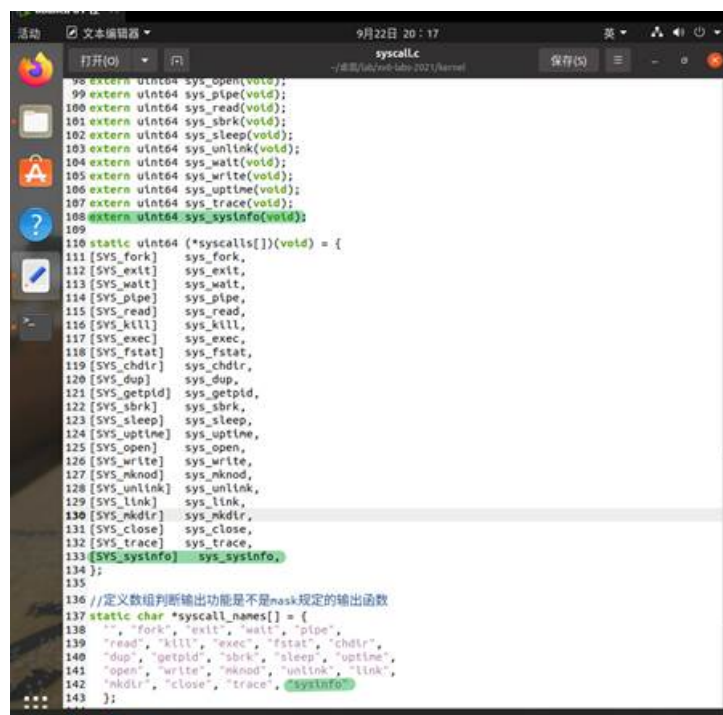
```
活动 文本编辑器 9月22日 19:57 usys.pl
~/桌面/lab/week-lab01-2021/user
保存(S)

1 #!/usr/bin/perl -w
2
3 # Generate usys.S, the stubs for syscalls.
4
5 print "# generated by usys.pl - do not edit!\n";
6
7 print "#include \"kernel/syscall.h\"\n";
8
9 sub entry {
10     my $name = shift;
11     print ".global $name\n";
12     print "$name:\n";
13     print "    .i 0, SYS_$name\n";
14     print "    .ecall\n";
15     print "    ret\n";
16 }
17
18 entry("fork");
19 entry("exit");
20 entry("wait");
21 entry("pipe");
22 entry("read");
23 entry("write");
24 entry("close");
25 entry("kill");
26 entry("exec");
27 entry("open");
28 entry("mknod");
29 entry("unlink");
30 entry("fstat");
31 entry("link");
32 entry("mkdir");
33 entry("chdir");
34 entry("dup");
35 entry("getpid");
36 entry("sbrk");
37 entry("sleep");
38 entry("uptime");
39 entry("trace");
40 entry("sysinfo");
```



```
// System call numbers
1 #define SYS_fork 1
2 #define SYS_exit 2
3 #define SYS_wait 3
4 #define SYS_pipe 4
5 #define SYS_read 5
6 #define SYS_kill 6
7 #define SYS_exec 7
8 #define SYS_fstat 8
9 #define SYS_chdir 9
10 #define SYS_dup 10
11 #define SYS_getpid 11
12 #define SYS_sbrk 12
13 #define SYS_sleep 13
14 #define SYS_uptime 14
15 #define SYS_open 15
16 #define SYS_write 16
17 #define SYS_mknod 17
18 #define SYS_unlink 18
19 #define SYS_link 19
20 #define SYS_mkdir 20
21 #define SYS_close 21
22 #define SYS_trace 22
23 #define SYS_sysinfo 23
```

- ④第三步，仿照 trace() 在 kernel/syscall.c 增加定义



```
99 extern uint64 sys_open(void);
100 extern uint64 sys_pipe(void);
101 extern uint64 sys_read(void);
102 extern uint64 sys_sbrk(void);
103 extern uint64 sys_sleep(void);
104 extern uint64 sys_unlink(void);
105 extern uint64 sys_wait(void);
106 extern uint64 sys_write(void);
107 extern uint64 sys_uptime(void);
108 extern uint64 sys_trace(void);
109 extern uint64 sys_sysinfo(void);
110
111 static uint64 (*syscalls[])(void) = {
112 [SYS_fork] sys_fork,
113 [SYS_exit] sys_exit,
114 [SYS_wait] sys_wait,
115 [SYS_pipe] sys_pipe,
116 [SYS_read] sys_read,
117 [SYS_kill] sys_kill,
118 [SYS_exec] sys_exec,
119 [SYS_fstat] sys_fstat,
120 [SYS_chdir] sys_chdir,
121 [SYS_dup] sys_dup,
122 [SYS_getpid] sys_getpid,
123 [SYS_sbrk] sys_sbrk,
124 [SYS_sleep] sys_sleep,
125 [SYS_uptime] sys_uptime,
126 [SYS_open] sys_open,
127 [SYS_write] sys_write,
128 [SYS_mknod] sys_mknod,
129 [SYS_unlink] sys_unlink,
130 [SYS_link] sys_link,
131 [SYS_mkdir] sys_mkdir,
132 [SYS_close] sys_close,
133 [SYS_trace] sys_trace,
134 [SYS_sysinfo] sys_sysinfo,
135 };
136
137 // 定义数组判断输出功能是不是mask规定的输出函数
138 static char *syscall_names[] = {
139 "fork", "exit", "wait", "pipe",
140 "read", "kill", "exec", "fstat", "chdir",
141 "dup", "getpid", "sbrk", "sleep", "uptime",
142 "open", "write", "mknod", "unlink", "link",
143 "mkdir", "close", "trace", "sysinfo",
144 };
145
```

- ⑤第四步，在 kernel/sysproc.c , kernel/ proc.c , kernel/kalloc.c 添加实现代码
  - kernel/proc.c :根据①实验要求，我们先写获取可用进程数目的函数实现（从 kernel/proc.c 的语句 struct proc proc[NPROC];知道，这里保存了所有进程），所以新增一个 nproc() 函数，获取进程数目。

```
// Return numbers of process that state is USED
uint64
nproc(void)
{
    struct proc *p;
```



```

//统计进程数目
uint64 num = 0;
//遍历所有进程
for(p = proc; p < &proc[NPROC]; p++)
{
    //加锁
    acquire(&p->lock);
    //进程是USED
    if(p->state != UNUSED)
    {
        num++;
    }
    //释放锁
    release(&p->lock);
}
return num;
}

```

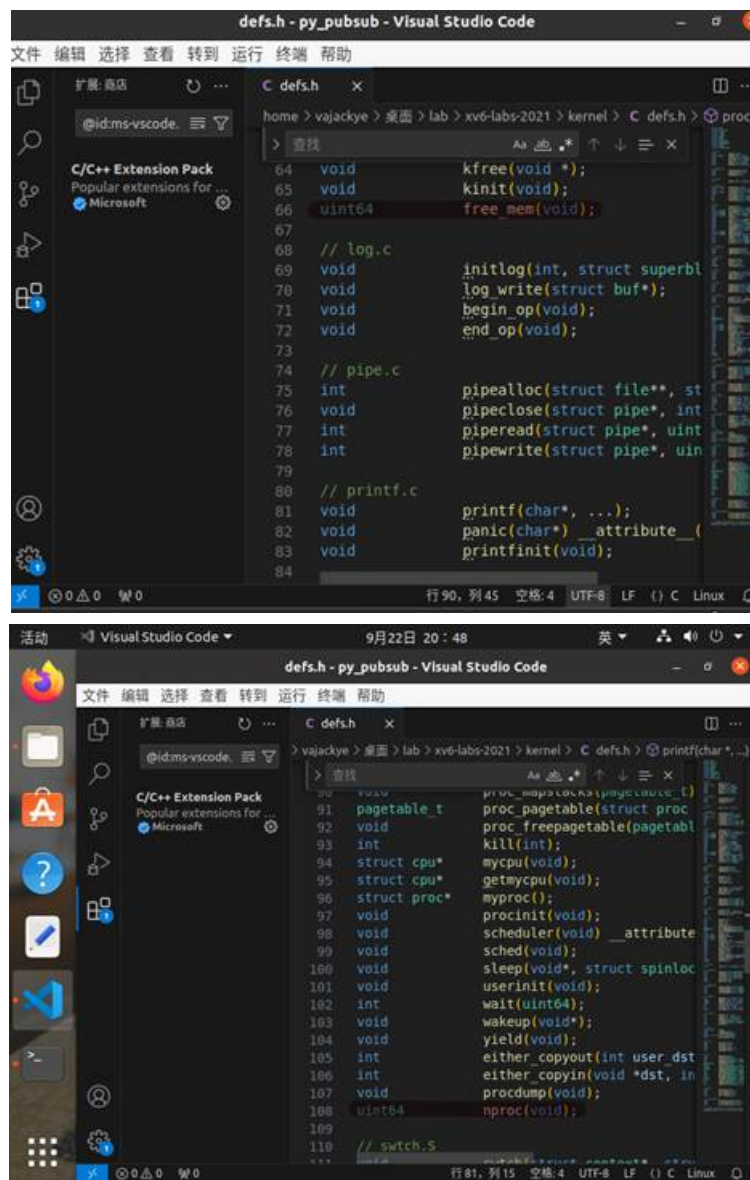
- `kernel/kalloc.c`: 同理, 根据①实验要求, 知道我们还需要获取空闲内存数目, 故新增函数 `free_mem` 获取空闲内存数量。

```

// Return the number of bytes of free memory
uint64
free_mem(void)
{
    struct run *r;
    //统计空闲页
    uint64 num = 0;
    //加锁
    acquire(&kmem.lock);
    //r 指向 空闲页表
    r = kmem.freelist;
    //如果页表有位置
    while(r)
    {
        num++;
        //r 指向下一页
        r = r->next;
    }
    //释放锁
    release(&kmem.lock);
    //每页4096字节
    return num * PGSIZE;
}

```

- 添加了代码实现后, 要有相应的函数声明, 上网查询后, 知道这两函数声明应该放在 `defs.h` 头文件中, 故在 `kernel/defs.h` 添加第i和第ii步的函数声明。



- `kernel/sysproc.c`：，最后，在 `sysproc.c` 内部添加 `sys_sysinfo` 获取当前进程数目和空闲内存数目的系统函数 `sysinfo()` 的具体实现

```
//sys_info实现
uint64
sys_sysinfo(void)
{
    //添加一个虚拟地址，指向struct sysinfo
    uint64 addr;
    struct sysinfo info;
    struct proc *p = myproc();

    if(argaddr(0, &addr) < 0)
    {
        return -1;
    }
    //获取空闲内存的字节数
    info.freemem = free_mem();
    //获取USED进程数目
    info.nproc = nproc();
    if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
    {
```

```

    return -1;
}
return 0;
}

```

- `user/sysinfo.c`: 之后, 还要再在 `user` 下添加 `sysinfo.c` 文件, 用于实现传入参数, 调用系统函数 `sys_sysinfo()` 获取当前进程数目和空闲内存数目, 并打印。

```

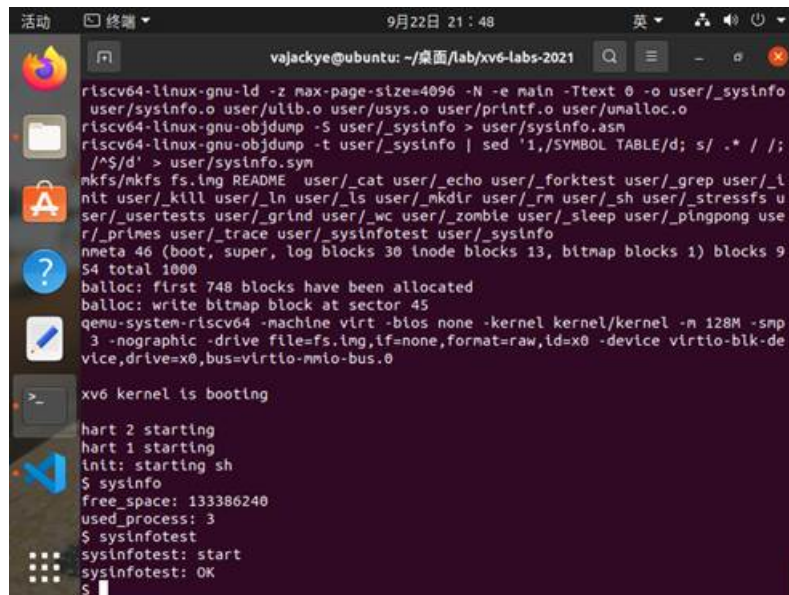
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/sysinfo.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
    //数目错误
    if(argc != 1)
    {
        printf("Usage: %s number argc fail\n", argv[0]);
        exit(-1);
    }

    struct sysinfo info;
    sysinfo(&info);
    //打印
    printf("free_space: %d\nused_process: %d\n", info.freemem,
info.nproc);
    exit(0);
}

```

- ⑥第五步, 配置完后, 编译并运行xv6进行样例测试。



```

vjackye@ubuntu: ~/桌面/lab/xv6-labs-2021
riscv64-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_sysinfo
user/sysinfo.o user/_lib.o user/_sys.o user/_printf.o user/_umalloc.o
riscv64-linux-gnu-objdump -S user/_sysinfo > user/sysinfo.asm
riscv64-linux-gnu-objdump -t user/_sysinfo | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > user/sysinfo.syn
mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_l
nit user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs u
ser/_usertests user/_grind user/_wc user/_zombie user/_sleep user/_pingpong use
r/_primes user/_trace user/_sysinfotest user/_sysinfo
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 9
54 total 1000
ballocc: first 748 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -n 128M -smp
3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-de
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ sysinfo
free_space: 133386240
used_process: 3
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$

```

### 三. 实验心得

---

- 在本次实验中，完成了两个任务：trace(), sysinfo(), 实验过程中通过阅读 xv6 的相关文档，以及上网查阅资料，了解函数实现时，应该在哪里添加声明，添加存根等，原因分别是什么，系统是如何调用这些函数的。
- 本次实验，个人感觉难点主要在于，要修改的文件很多，而且一开始还不明白为何要在某处修改添加内容。然后第二个实验又和第一个实验添加的东西略微不同，添加的地方也不同，导致需要不断地上网查资料，查攻略，查原理，问AI，不断地梳理流程。以及查阅了一些参考代码，了解代码如何书写，含义分别是什么，哪个是核心代码等。
- 总之，这次实验，相比起第一次实验，极大的拓宽了我对底层系统部分执行逻辑的认识。