

Lab2 报告

刘鸣霄 自 12 2021010584

5 月 10 日

xv6 Lab2 系统调用带领我们初步认识并熟悉了 xv6 的内核，并为其添加新的系统调用。

1. Using gdb

我在报告里详细描述了完成这道题的过程。按照 handout 的指示，首先输入

```
$ make qemu-gdb
```

然后打开另一个终端，并输入（如果失败，按照终端的指示在某个文件内添加一句话）

```
$ gdb-multiarch
```

就打开了 gdb，下面进行第一个任务

1) 哪个函数调用 syscall

按照 handout 的指示依次：在函数 syscall 处打断点，运行至断点，查看 backtrace

```
(gdb) b syscall
Breakpoint 1 at 0x8000203e: file kernel/syscall.c, line 164.
(gdb) c
Continuing.
Thread 1 hit Breakpoint 1, syscall () at kernel/syscall.c:164
164 {
(gdb) backtrace
#0  syscall () at kernel/syscall.c:164
#1  0x0000000080001d72 in usertrap () at kernel/trap.c:67
#2  0x0505050505050505 in ?? ()
```

答案：usertrap 调用了 syscall

2) p->trapframe->a7 的值是什么，那个值代表什么？

继续输入 n 指令，直到运行到指令 `struct proc* p=myproc();` 运行结束，此时使用 p 指令查看 p->trapframe->a7 的值。

```
(gdb) n
```

```
166 struct proc *p = myproc();  
  
(gdb) n  
  
168 num = p->trapframe->a7;  
  
(gdb) p/x p->trapframe->a7  
$2 = 0x7
```

可见 a7 内的值是 0x7。

(下面是一张运行 gdb 的图片，没有 layout src)

```

th@ubuntu: ~/Desktop/Labs/Lab2_system calls/xv6_ForLab2
th@ubuntu: ~/Desktop/Labs/Lab2_s...  th@ubuntu: ~/Desktop/Labs/Lab2_s...
Thread 1 hit Breakpoint 1, syscall () at kernel/syscall.c:164
164 {
(gdb) bttrace
#0  syscall () at kernel/syscall.c:164
#1  0x00000000000001d72 in usertrap () at kernel/trap.c:67
#2  0x0505050505050505 in ?? ()
(gdb) n
166     struct proc *p = myproc();
(gdb) n
168     num = p->trapframe->a7;
(gdb) p/x *p
s1 = {lock = 0x0, killed = 0x0, vstate = 0x0, pid = 0x1, parent = 0x0,
kstack = 0x3fffffff000, sz = 0x1000, pageable = 0x87f73000,
trapframe = 0x87ff74000, context = {ra = 0x80001496, sp = 0x3fffffffde00,
s0 = 0x8000fde0b0, s1 = 0x80000bec0, s2 = 0x8000ba90, s3 = 0x1, s4 = 0x0,
s5 = 0x3, s6 = 0x80019d60, s7 = 0x8, s8 = 0x800019e88, s9 = 0x4, s10 = 0x1,
s11 = 0x0}, ofile = {0x0 <repeats 16 times>}, cwd = 0x800171d0, name = {
0x69, 0x6e, 0x69, 0x73, 0x63, 0x6f, 0x64, 0x65, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0}, tracemask = 0x0}
(gdb) q
(gdb) p/x p->trapframe->a7
s2 = 0x7
(gdb)

```

xv6 手册的第二章中讲到，a7 寄存器保存了即将调用的系统调用的代号，查看 kernel/syscall.h 可知 7 号系统调用是 SYS_exec。exec 是 xv6 开机调用的第一个系统调用。

答案: 0x7; SYS_exec

3) 现在处理器正在运行于内核模式，处理器的前一个模式是什么？

这里参考 handout 提供的帮助文档。

The SPP bit indicates the privilege level at which a hart was executing before entering supervisor mode. When a trap is taken, SPP is set to 0 if the trap originated from user mode, or 1 otherwise. When an SRET instruction (see Section 3.3.2) is executed to return from the trap handler, the privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0.

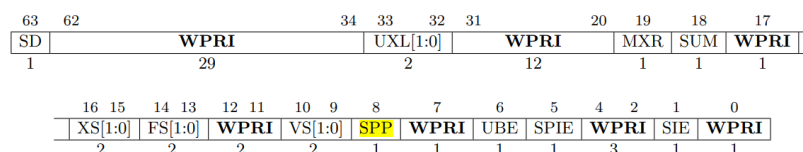


Figure 4.2: Supervisor-mode status register (**sstatus**) when SXLEN=64.

使用 gdb 查看 mstatus 寄存器中的值

```
(gdb) p/t $mstatus
```

```
$1 = 10100010
```

按照图示，SPP 应为第 8 位，为 0（最低位是第 0 位）。因此先前的模式是用户模式。

答案：用户模式(user mode)

4) 哪条汇编指令导致内核崩溃(panic)，哪个寄存器对应变量 num?

将 syscall 函数中的一条指令 num = p->trapframe->a7; 改为 num = *(int*)0;，

之后运行 make qemu 发现 xv6 开机时内核崩溃，崩溃的信息如下。

```
xv6 kernel is booting
```

```
hart 1 starting
```

```
hart 2 starting
```

```
scause 0x000000000000000d
```

```
sepc=0x0000000080002056 stval=0x0000000000000000
```

```
panic: kerneltrap
```

在此，sepc 寄存器的值就是导致内核崩溃的代码地址。为了验证是否真的是这条指令导致了内核崩溃，我们使用 gdb 进行调试，在 sepc 指示的地址打断点

```
(gdb) b *0x0000000080002056
```

```
Breakpoint 1 at 0x80002056: file kernel/syscall.c, line 169.
```

```
(gdb) layout asm
```

The screenshot shows a GDB terminal window with the following content:

```
thu@ubuntu: ~/Desktop/Labs/Lab2_system calls/xv6_for_Lab2
thu@ubuntu: ~/Desktop/Labs/Lab2_s... x  thu@ubuntu: ~/Desktop/Labs/Lab2_s... x
0x80002048 <syscall+10> sd      s3,8(sp)
0x8000204a <syscall+12> addi    s0,sp,48
0x8000204c <syscall+14> auipc   ra,0xfffff
0x80002050 <syscall+18> jalr    -464(ra)
0x80002054 <syscall+22> mv      s1,a0
B+>0x80002056 <syscall+24> lw      s2,0(zero) # 0x0
0x8000205a <syscall+28> addiw   a4,s2,-1
0x8000205e <syscall+32> li      a5,22
0x80002060 <syscall+34> bltu    a5,a4,0x800020b4 <syscall+118>
0x80002064 <syscall+38> slli    a4,s2,0x3
0x80002068 <syscall+42> auipc   a5,0x6
0x8000206c <syscall+46> addi    a5,a5,1072
0x80002070 <syscall+50> add     a5,a5,a4
remote Thread 1.2 In: syscall L169 PC: 0x80002056
(gdb) c
Continuing.
[Switching to Thread 1.2]
Thread 2 hit Breakpoint 1, syscall () at kernel/syscall.c:169
(gdb)
```

可见导致内核崩溃的指令是 `lw s2,0(zero)`。在 RISC-V 指令集中, `lw` 是 `load word`, 即将一个 `word` 大小的数据从内存加载到寄存器。

在 `kernel/kernel.asm` 中搜索这句汇编代码:

```
num = *(int*)0;

80002056: 00002903          lw s2,0(zero) # 0 <_entry-0x80000000>
```

对应的正是那句修改了的代码, `s2` 是变量 `num` 对应的寄存器 (我记得第一遍做这个题的时候好像不是这个寄存器, 变了)

答案: `lw s2,0(zero); s2`

5) 内核崩溃的原因是什么? `0` 被映射到内核的地址空间吗?

`scause` 寄存器保存了内核崩溃的原因。在 RISC-V 手册中查询

0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10-11	Reserved
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved
0	15	Store/AMO page fault
0	16-23	Reserved
0	24-31	Designated for custom use
0	32-47	Reserved
0	48-63	Designated for custom use
0	≥64	Reserved

`13(0xd)` 表示 `load page fault`, 表示内核因为加载了 `0` 地址而出错。

根据 `xv6` 手册 2.6 节, `xv6` 开机时内核加载到 `0x80000000` (开机时虚拟地址直接映射到物理地址), 所以 `0` 地址并不映射到内核空间, `0-0x80000000` 有其他功能, 如留给 I/O 设备。

答案: `Load page fault`; 不映射

6) 内核崩溃时运行的二进制程序是什么, 它的 `pid` 是什么?

直接在 `gdb` 中使用 `p` 指令打印。

```
(gdb) p p->name
$1 = "initcode\000\000\000\000\000\000\000"
(gdb) p p->pid
$2 = 1
```

`initcode` 是 `xv6` 开机时运行的第一个用户程序。

答案: `initcode`; `1`

2. System call tracing

实现一个系统调用 `trace`，其参数为一个掩码 `mask`，其中若 `mask` 的第 `n` 位为 `1`，则在调用 `n` 号系统调用的时候输出一条提示信息。

根据提示一步步完成。

1) 在 `Makefile` 中添加用户程序 `$U/_trace`

2) 在 `user/user.h` 中添加 `trace` 的声明

```
int trace(int);
```

在 `usys.pl` 中添加一个 `entry`

```
entry("trace");
```

3)

在 `sysproc.c` 中实现该调用。由于内核态和用户态是隔离的，所以需要采用特殊的方式从用户态获得参数。与其他的系统调用传参类似，内核的 `sys_trace` 函数需要调用 `argint` 函数，从进程的 `trapframe->a0` 中获得参数。

`trace` 的实现需要在 `struct proc` 结构体中再添加一个数据成员 `tracemask`，记录需要被追踪的系统调用。在调用 `trace` 时，其功能是设置好进程的 `tracemask`。

```
// proc.h
struct proc{
    //...
    int tracemask;           // 新增：追踪的系统调用
};

// sysproc.c
uint64
sys_trace(void)
{
    int n;

    argint(0, &n); // 获取参数
    myproc()->tracemask = n; // 设置 tracemask
    return 0;
}
```

```
}
```

4)

修改 `fork` 函数，使得父进程的 `tracemask` 被子进程继承。修改 `allocproc` 函数，使得 `tracemask` 的初始值设置为 0。

```
// proc.c
static struct proc*
allocproc(void)
{
    //...
    p->tracemask = 0;
    return p;
}

int
fork(void)
{
    //...
    safestrcpy(np->name, p->name, sizeof(p->name));
    np->tracemask = p->tracemask; //新增: 复制 tracemask
    //...
}
```

5) 修改 `syscall.h` 和 `syscall.c`

在 `syscall.h` 中添加

```
#define SYS_trace 22
```

在 `syscall.c` 中添加: `sys_trace` 的声明, `sys_trace` 的函数指针, 和一个记录所有系统调用名称的数组

```
...
extern uint64 sys_trace(void);

static uint64 (*syscalls[])(void) = {...,
```

```

[SYS_trace]    sys_trace
};

// 新增：记录各个系统调用的名称
char* syscall_name[]={
[SYS_fork]     "fork",
[SYS_exit]     "exit",
...
};

```

最后一步是在 `syscall.c` 中修改函数 `syscall`。

```

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]();
        if(p->tracemask & (1 << num)){
            printf("%d: syscall %s -> %d\n", p->pid,
                syscall_name[num], p->trapframe->a0);
        }
    } else {
        printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);

        p->trapframe->a0 = -1;
    }
}

```

红字是新增的代码。`num` 就是系统调用号。在系统调用返回之后，如果 `num` 是被追踪的

系统调用，则输出提示信息。p->pid 是进程的进程号，p->trapframe->a0 是返回值。

备注：我在测试 forkforkfork 程序的时候会超时，可能是我的虚拟机配置不够高。我将提交的源代码中的测评脚本的超时时间从 30s 改成了 50s，然后是可以通过的（大概用 30 多秒）。

3. Sysinfo

实现一个系统调用 sysinfo，返回一个结构体 struct sysinfo，包含目前的可用内存和进程数。

前几步与 trace 一样：

- 1) 在 Makefile 中添加 \$U/_sysinfo
- 2) 在 user/user.h 中添加系统调用函数声明，在 usys.pl 中添加 entry
- 3) 在 kernel/syscall.h 中添加系统调用号，在 syscall.c 中添加一些相应条目

然后在 sysproc.c 中实现 sys_sysinfo，暂且把这个系统调用归为进程类。首先通过 argaddr 来获取地址参数（此处的参数是一个指针 struct sysinfo*，用于输出），之后通过两个函数（稍后实现）count_freemem 和 count_nproc 来获取空闲内存和进程数，然后通过 copyout 函数将结果拷贝到用户空间（copyout 的用法可以参考 fstat）。

```
uint64
sys_sysinfo(void)
{
    struct sysinfo sys_info;
    uint64 addr;
    argaddr(0, &addr);
    struct proc* p = myproc();

    // 获取空闲内存和进程数
    sys_info.freemem = count_freemem();
    sys_info.nproc = count_nproc();

    // 调用 copyout 将信息拷贝到用户空间
    if(copyout(p->pagetable, addr,
```



```

    (char *)&sys_info, sizeof(sys_info)) < 0){
        return -1;
    }

    return 0;
}

```

下面来实现这两个函数 `count_freemem` 和 `count_nproc`。先在 `kernel/def.h` 中添加对应的函数声明。

在 `kernel/kalloc.c` 中实现 `count_freemem`。根据 `xv6` 的手册，`xv6` 将所有空闲页面组织成一个链表，而空闲页面本身作为链表的节点，`kmem.freelist` 指向这个链表的首节点（`struct run` 是节点类型）。

```

// kalloc.c
struct run {
    struct run *next;
};

struct {
    struct spinlock lock;
    struct run *freelist;
} kmem;

```

统计空闲内存就是计算这个链表的长度，再将结果乘以页大小 `PGSIZE`。

```

// kalloc.c
//计算可用内存数量
uint64
count_freemem(void){
    struct run* p = kmem.freelist;
    uint64 cnt = 0;
    //可用的内存数量就是 freelist 的长度乘以页的大小
    while(p){

```

```
    ++cnt;

    p = p->next;
}

return cnt * PGSIZE;
}
```

在 kernel/proc.c 中实现 count_nproc。proc.c 开头定义了一个 struct proc 数组用于存储所有的进程。

```
struct proc proc[NPROC];
```

我们需要遍历这个数组，计数所有 state 为 UNUSED 的进程的数量。

```
// 统计进程数量
uint64
count_nproc(void)
{
    struct proc* p;
    uint64 cnt = 0;

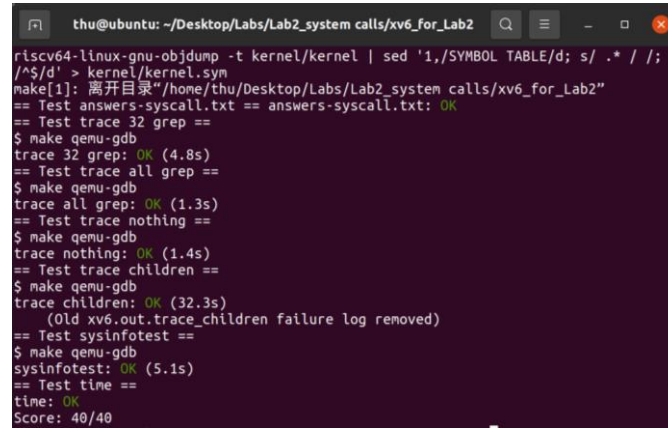
    // 遍历 proc 数组，对状态不为 UNUSED 的进程计数
    for(p = proc; p < &proc[NPROC]; p++) {
        if(p->state != UNUSED) {
            ++cnt;
        }
    }

    return cnt;
}
```

4. Time

该实验用时约 6 小时。难点不在实验，而是在阅读并理解 xv6 手册和内核代码。

5. Make Grade

A terminal window titled 'thu@ubuntu: ~/Desktop/Labs/Lab2_system calls/xv6_for_Lab2' showing the process of building and testing the xv6 kernel. The user runs 'riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /<\$/d' > kernel/kernel.sym', followed by 'make[1]: 离开目录"/home/thu/Desktop/Labs/Lab2_system calls/xv6_for_Lab2"'. The output shows a series of tests: 'Test answers-syscall.txt == answers-syscall.txt: OK', 'Test trace 32 grep ==', '\$ make qemu-gdb', 'trace 32 grep: OK (4.8s)', 'Test trace all grep ==', '\$ make qemu-gdb', 'trace all grep: OK (1.3s)', 'Test trace nothing ==', '\$ make qemu-gdb', 'trace nothing: OK (1.4s)', 'Test trace children ==', '\$ make qemu-gdb', 'trace children: OK (32.3s)', '(Old xv6.out.trace_children failure log removed)', 'Test sysinfotest ==', '\$ make qemu-gdb', 'sysinfotest: OK (5.1s)', 'Test time ==', 'time: OK', and finally 'Score: 40/40'.

```
thu@ubuntu: ~/Desktop/Labs/Lab2_system calls/xv6_for_Lab2
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^\$/d' > kernel/kernel.sym
make[1]: 离开目录"/home/thu/Desktop/Labs/Lab2_system calls/xv6_for_Lab2"
== Test answers-syscall.txt == answers-syscall.txt: OK
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (4.8s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (1.3s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.4s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (32.3s)
(Old xv6.out.trace_children failure log removed)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (5.1s)
== Test time ==
time: OK
Score: 40/40
```