OS Lab 2

Traps

王世炟

PB20151796

October 30, 2022

Part 1: 实验要求

本次实验要求我们完成 Traps **中断陷阱**的某些功能的添加，了解用户态与内核态间的联系以及中断。完成本实验之前首先需要阅读xv6 book的第四章, 以了解在中断过程中各个寄存器的作用（具体使用时会在之后的内容解释）。
主要涉及到的文件有：riscv.h def.h sysproc.c printf.c Makefile user.h usys.pl syscall.h syscall.c proc.h proc.c trap.c

Part 2: Backtrace

2.1 问题简述

在 kernel/printf.c 中实现一个函数 backtrace() , 并在 sys_sleep() 中调用它。backtrace() 会打印当前栈上所有函数调用。

2.2 知识基础

在做本实验的过程中遇到了一些困难，发现是因为不懂堆栈造成的，于是先观看了一个小时click here to the video, 学习了有关堆栈的知识：
x86 使用的函数参数压栈的方式来保存函数参数，xv6 使用寄存器的方式保存参数。
无论是 x86 还是 xv6，函数调用时，都需要将返回地址和调用函数（父函数）的栈帧起始地址压入栈中。即被调用函数的栈帧中保存着这两个值。在 xv6 中，fp 为当前函数的栈顶指针，sp 为栈指针。fp-8 存放返回地址，fp-16 存放原栈帧（调用函数的 fp）。

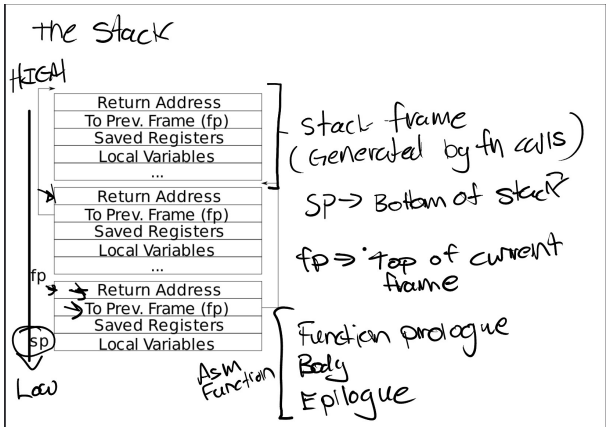

图 1: the stack

2.3 实验步骤

(1) **添加声明**

首先根据实验提示，首先在 kernel/defs.h 中添加 backtrace 的声明：void backtrace(void);

(2) **添加** r_fp()

按照提示添加 r_fp()

```
static inline uint64
r_fp()
{
  uint64 x;
  asm volatile("mv %0, s0" : "=r" (x) );
  return x;
}
```

先获取到当前函数的栈帧 fp 的值，该值存放在 s0 寄存器中，r+fp() 即是一个能够读取 s0 寄存器值的函数。

(3) **实现** backtrace()

根据前面给出的理论，循环跳出的条件就是栈底地址大于栈顶，于是有了以下代码：

```
void
backtrace(void)
{
  printf("backtrace:\n");

  uint64 addr = r_fp();
  uint64 return_addr;
  uint64 bottom = PGROUNDUP(addr);
  while (addr < bottom)
  {
```

```
11      return_addr = *((uint64*)(addr-8));
12      addr = *((uint64*)(addr-16));
13      printf("%p\n", return_addr);
14    }
15    return;
16  }
```

最后在 printf.c 中的 panic() 函数中加入调用 backtrace(); 在 sysproc.c 中的 sys_sleep() 函数中加入调用 backtrace(); 即可。

(4) **结果**

按照实验流程得到以下结果：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ bttest
backtrace:
0x0000000080002d4c
0x0000000080002c26
0x00000000800028c2
$ QEMU: Terminated
```

图 2: bttest

```
ubuntu@VM5153-OSVM:~/桌面/xv6-labs-2020$ addr2line -e kernel/kernel
0x0000000080002d4c
/home/ubuntu/桌面/xv6-labs-2020/kernel/sysproc.c:62
0x0000000080002c26
/home/ubuntu/桌面/xv6-labs-2020/kernel/syscall.c:144
0x00000000800028c2
/home/ubuntu/桌面/xv6-labs-2020/kernel/trap.c:76
```

图 3: addr2line

Part 3: Alarm

3.1 问题简述

实现系统调用 sigalarm(int ticks, void (*handler)()); 和 sigreturn(void); 使得每过一定数目的 cpu 的切片时间，调用一个用户函数，同时，在调用完成后，需要恢复到之前没调用时的状态。

这里需要注意的是：

- 在当前进程，已经有一个要调用的函数正在运行时，不能再运行第二个；
- 注意寄存器值的保存方式，在返回时需要保存寄存器的值；
- 系统调用的声明和书写方式。

3.2 知识基础

首先，补充一下关于指令 ecall 的有关内容：

- 首先，当用户调用系统调用的函数时，在进入函数前，会执行 user/usys.S 中相应的汇编指令，指令首先将系统调用的函数码放到 a7 寄存器内，然后执行 ecall 指令进入内核态。
- ecall 指令是 cpu 指令，该指令只做三件事情。
  - 首先将 cpu 的状态由用户态（user mode）切换为内核态（supervisor mode）；
  - 然后将程序计数器的值保存在了 SEPC 寄存器；
  - 最后跳转到 STVEC 寄存器指向的指令。
- 在 kernel/trap.c 中，需要检查触发 trap 的原因，以确定相应的处理方式。产生中断的原因有很多，比如系统调用、运算时除以 0、使用了一个未被映射的虚拟地址、或者是设备中断等等。这里是因为系统调用，所以以系统调用的方式进行处理。
- 接下来开始在内核态执行系统调用函数，在 kernel/syscall.c 中取出 a7 寄存器中的函数码，根据该函数码，调用 kernel/sysproc.c 中对应的系统调用函数。
- 最后，在系统调用函数执行完成后，将保存在 trapframe 中的 SEPC 寄存器的值取出来，从该地址存储的指令处开始执行（保存的值为 ecall 指令处的 PC 值加上 4，即为 ecall 指令的下一条指令）。随后执行 ret 恢复进入内核态之前的状态，转为用户态。

3.3 实验步骤

3.3.1 Makefile

在 Makefile 中添加 $U/_alarmtest 让 make 能识别它

3.3.2 添加声明

同实验 1，实现系统调用：
在 user.h 中添加声明：

```
1 // system calls
2 int sigalarm(int ticks, void (*handler)());
3 int sigreturn(void);
```

在 usys.pl 中添加如下内容：

```
entry("sigalarm");
entry("sigreturn");
```

在 kernel/syscall.h 中添加如下内容：

```
#define SYS_sigalarm  22
#define SYS_sigreturn  23
```

在 kernel/syscall.c 中添加如下内容：

```
extern uint64 sys_sigalarm(void);
extern uint64 sys_sigreturn(void);

static uint64 (*syscalls[])(void) = {
...
[SYS_sigalarm]   sys_sigalarm,
[SYS_sigreturn]   sys_sigreturn,
};
```

3.3.3 **完成** test0

按照提示完成 test0

可以查看 alarmtest.c 的代码，能够发现 test0 只需要进入内核，并执行至少一次即可。不需要正确返回也可以通过测试。

首先，写一个 sys_sigreturn 的代码，直接返回 0 即可（后面再添加）：

```
uint64
sys_sigreturn(void)
{
  return 0;
}
```

然后，在 kernel/proc.h 中的 proc 结构体添加字段，用于记录时间间隔，经过的时钟数和调用的函数信息：

```
uint64
sys_sigreturn(void)
struct proc {
  ...
  void(*handler)();             // function pointer
  int alarm_interval;           // interval
  int total_ticks;              // totol ticks
};
```

编写 sys_sigalarm() 函数，获取相应 proc 结构体中的字段的值, 由实验 1 的知识，函数存进来的两个参数分别储存在 a0 和 a1 里：

```
uint64
sys_sigalarm(void)
{
  int interval;
  uint64 pointer;
  struct proc *p;
  if (argint(0, &interval)<0 || argaddr(1, &pointer) < 0 || interval < 0)
  {
    return -1;
  }
  p=myproc();
  p->alarm_interval = interval;
  p->handler = (void*)pointer;
  return 0;
}
```

按照提示进行进程初始化，在 proc.c 中 allocproc() 加入如下内容：

```
found:
    p->alarm_interval = 0;
    p->total_ticks = 0;
    p->handler = (void*)0;
```

最后就到了改写中断的步骤，每经历一个 tick ，系统发生时钟中断，执行 trap.c, 根据提示修改 usertrap() 函数 if(which_dev == 2) 部分：

```
if(which_dev == 2)
  {
    if(p->alarm_interval) {
          if(p->total_ticks == p->alarm_interval) {
            p->total_ticks = 0;
                p->trapframe->epc = (uint64)p->handler;
          }
          p->total_ticks++;
        }
    yield();
  }
```

第 6 行将 epc 赋值成 handler 的理由是：根据上面补充的知识以及xv6 book的第四章可以了解到，在用户态引起中断时，系统将程序计数器的值保存在了 SEPC 寄存器，以便回到用户态时可以恢复到中断之前的状态，此处将 epc 赋值成 handler，既做到了回到用户态时执行 handler。

如上改动过后即可通过 test0：

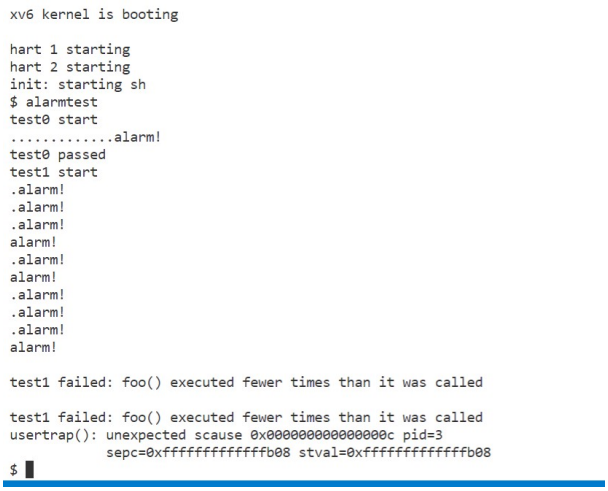

图 4: test0 passed

3.3.4 **完成** test1、test2

在这里需要实现正确返回到调用前的状态。为了实现此功能，我们首先要对中断有以下理解：

- 在执行好 handler 后，我们希望的是回到用户调用 handler 前的状态。但那时的状态已经被用来调用 handler 函数了，现在的 trapframe 中存放的是执行 sys_sigreturn 前的 trapframe，如果直接返回到用户态，则找不到之前的状态，无法实现我们的预期。

- 在 alarmtest 代码中可以看到，每个 handler 函数最后都会调用 sigreturn 函数，用于恢复之前的状态。由于每次使用 ecall 进入中断处理前，都会使用 trapframe 存储当时的寄存器信息，包括时钟中断。因此 trapframe 在每次中断前后都会产生变换，如果要恢复状态，需要额外存储 handler 执行前的 trapframe（即更改返回值为 handler 前的 trapframe），这样，无论中间发生多少次时钟中断或是其他中断，保存的值都不会变。

- 因此，在 sigreturn 只需要使用存储的状态覆盖调用 sigreturn 时的 trapframe，就可以在 sigreturn 系统调用后恢复到调用 handler 之前的状态。再使用 ret 返回时，就可以返回到执行 handler 之前的用户代码部分。

所以，其实只需要增加一个字段，用于保存调用 handler 之前的 trapframe 即可。
实验还要求我们在一个 handler 执行期间另一个 handler 不能执行，实现这个功能比较简单，添加一个进程字段用于识别是否在执行 handler 即可
在 proc.h 中加入如下一个指向 trapframe 结构体的指针和一个标志位：

```
struct proc {
  ...
  void(*handler)();             // function pointer
  int alarm_interval;           // interval
  int total_ticks;              // totol ticks
  struct trapframe *copytrapframe; // store the trapframe before the handler
                                   //is run
  int is_handler_in;            // judge whether the handler is still running

};
```

在进程初始化时为指针和标志位赋值:

```
found:
    p->alarm_interval = 0;
    p->total_ticks = 0;
    p->handler = (void*)0;
    p->is_handler_in = 1;

    // Allocate a copytrapframe page.
    if((p->copytrapframe = (struct trapframe *)kalloc()) == 0){
        release(&p->lock);
        return 0;
    }
```

之后按照上述内容对于 trap.c 中的 usertrap() 和 kernel/sysproc.c 中的 sys_sigreturn(void) 函数进行修改:

```
if(which_dev == 2)
  {
    if(p->alarm_interval) {
      if(p->total_ticks == p->alarm_interval) {
        if (p->is_handler_in)
        {
          p->is_handler_in = 0;
          *p->copytrapframe = *p->trapframe;
          p->trapframe->epc = (uint64)p->handler;
          p->total_ticks = 0;
        }
      }
            p->total_ticks++;
          }
    yield();
  }
```

```
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  *p->trapframe = *p->copytrapframe;
  p->is_handler_in = 1;
  return 0;
}
```

本以为到这里就已经实现了，运行 test0,1,2 也确实均通过了：

```
$ alarmtest
test0 start
.........alarm!
test0 passed
test1 start
..alarm!
.alarm!
.alarm!
.alarm!
..alarm!
...alarm!
.alarm!
.alarm!
..alarm!
.alarm!
test1 passed
test2 start
............alarm!
test2 passed
$ QEMU: Terminated
ubuntu@VM5153-OSVM:~/桌面/xv6-labs-2020$
```

图 5: test0、1、2 passed

但是当我运行 make grade 时却在 usertests 报错了，第一个错误是 timeout 这个错误还可以理解, 毕竟 vlab 有很多人使用，性能一般可以理解，我去详细检查了一遍 grade-lab-traps 函数，发现在 63 行最大时间设的是 300s, 我将其改为 500s.

```
@test(19, "usertests")
def test_usertests():
    r.run_qemu(shell_script([
        'usertests'
    ]), timeout=500)
    r.match('^ALL TESTS PASSED$')
```

但是我改完之后重新运行 make grade , 仍然报错，如下：

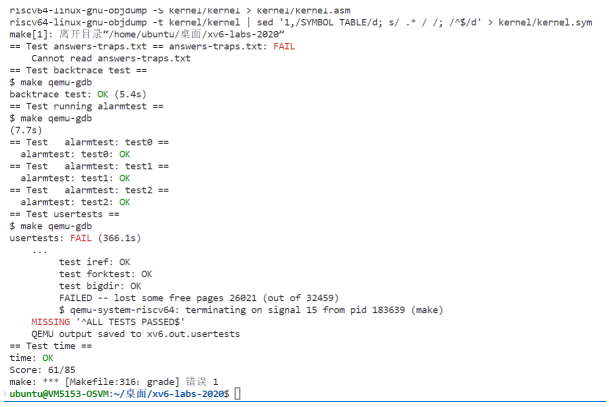

图 6: error

当时人比较慌乱，以为是不小心修改了系统的其他部分。冷静下来之后查看报错信息，lost some free pages ，我意识到这不一定是修改了其他部分，于是查看 usertests 代码，找到如下内容：

```
if(fail){
    printf("SOME TESTS FAILED\n");
    exit(1);
  } else if((free1 = countfree()) < free0){
    printf("FAILED -- lost some free pages %d (out of %d)\n", free1, free0);
    exit(1);
  } else {
    printf("ALL TESTS PASSED\n");
    exit(0);
  }
```

其中最关键的部分就是这个 countfree() 函数，根据程序介绍这个函数是使用 sbrk() 来计算有多少空闲的物理内存页。因为懒惰分配（lazy allocation 不知道怎么翻译）的内存耗尽会导致进程出现故障并被杀死，所以要分叉并报告。我明白了是因为我申请分配内存没有释放，于是修改代码：

在 ‘proc.c‘ 中 ‘freeproc()‘加入如下内容，用于进程结束释放内存：

```
p->alarm_interval = 0;
p->handler = 0;
p->total_ticks = 0;
if(p->copytrapframe)
    kfree((void*)p->copytrapframe);
```

修改之后，运行成功！

Part 4: 实验结果

运行 make grade，检验结果：

```
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TA
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
== Test answers-traps.txt == answers-traps.txt: FAIL
    Cannot read answers-traps.txt
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (15.9s)
== Test running alarmtest ==
$ make qemu-gdb
(8.9s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (395.6s)
== Test time ==
time: OK
Score: 80/85
make: *** [Makefile:316: grade] 错误 1
ubuntu@VM5153-OSVM:~/桌面/xv6-labs-2020$
```

图 7: Result

本次实验完成了 Traps 某些功能的实现。在实验过程中遇到了以下问题：

- 堆栈及中断的基础知识不是很了解，有关寄存器的知识看了 xv6book 之后也未完全搞懂
- 不理解报错信息，无法及时找到错误原因
- 对于用户态和内核态的理解十分不深刻，亟待加强

小小吐槽

感觉实验难度比较高，本次实验 + 实验报告大概花费我两天晚上 + 一天下午，上手比较困难，文件有些多，互相之间的调用也有些复杂，不知道如何才能调试 (gdb)，知识储备（比如堆栈）也不够（大数据人没上过计组），make 和 qemu 也是现学现卖，能不能麻烦助教开个线上答疑课稍微介绍或者演示一下这 (make, qemu,gdb) 方面的知识？（看 mit 的视频会讲解有关知识之后才做实验，我们是不是难度有些过高了）。