

实验二 traps

PB20050987 梁兆懿

在实验二中，需要尝试对堆栈进行操作，并且通过陷阱实现实验功能，完成实验BACKTRACL 和 ALARM。

Backtrace

回溯在debug中有着较多作用。在实验中通过bttest实现对sys_sleep进行回溯操作。在回溯后应该得到如下结果：

```
backtrace:
0x0000000080002cda
0x0000000080002bb6
0x0000000080002898
```

在离开qemu之后，我们运行：

```
$ addr2line -e kernel/kernel
0x0000000080002de2
0x0000000080002f4a
0x0000000080002bfc//用自己得到的代码代替
Ctrl-D
```

得到：

```
kernel/sysproc.c:74
kernel/syscall.c:224
kernel/trap.c:85
```

实验代码

首先，我们在 kernel/defs.h 中增加头文件：

```
// printf.c
void      printf(char*, ...);
void      panic(char*) __attribute__((noreturn));
void      printfinit(void);
void      backtrace(void);
```

随后，在 printf.c 中添加函数：

```

void backtrace(void){
    uint64 fpad = r_fp();
    uint64 max = PGROUNDUP(fpad);
    while (fpaddr < max) {
        printf("%p\n", *((uint64*)(fpaddr - 8)));
        fpaddr = *((uint64*)(fpaddr - 16));
    }
}

```

其中，由提示得知，`r_fp`函数应在 `kernel/riscv.h` 中给出，该函数用于获取栈首地址。函数 `PGROUNDUP (fapd)` 用于计算堆栈页面的顶部地址。在读取顶部指针后，我们在`backtrack`函数中输出。

由xv6实验讲义第四章知，返回地址应位于距堆栈帧的帧指针固定偏移（-8）的位置，并且帧指针应位于距帧指针固定偏移（-16）的位置。

我们在 `kernel/riscv.h` 文件添加函数 `r_fp` 用于读取 `s0` 寄存器的值。

```

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

```

在 `kernel/syspro.c` 文件中，我们对`sys_sleep`函数添加对`backtrace`的调用：

```

uint64 sys_sleep(void){
    int n;
    uint ticks0;

    backtrace();//调用处

    if(argint(0, &n) < 0)
        return -1;
    acquire(&tickslock);
    ticks0 = ticks;
    while(ticks - ticks0 < n){
        if(myproc()->killed){
            release(&tickslock);
            return -1;
        }
        sleep(&ticks, &tickslock);
    }
    release(&tickslock);
    return 0;
}

```

此外，为了在系统崩溃时添加回溯功能（`panickernel`），我们在 `panic`函数中添加回溯地址显示。在 `kernel/printf.c`文件中添加`backtrack`：

```

void
panic(char *s)
{

    backtrace();//添加处

    pr.locking = 0;
    printf("panic: ");
    printf(s);
    printf("\n");
    panicked = 1;
    for(;;)
        ;
}

```

运行结果

```

ubuntu@VM5878-LZY: /home/ubuntu/桌面/xv6-labs-2020
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ sys_sleep
exec sys_sleep failed
$ bttest
0x0000000080002dc6
0x0000000080002ca0
0x0000000080002898
$ QEMU: Terminated
ubuntu@VM5878-LZY:/home/ubuntu/桌面/xv6-labs-2020$ addr2line -e kernel/kernel^C
ubuntu@VM5878-LZY:/home/ubuntu/桌面/xv6-labs-2020$ addr2line -e kernel/kernel 0x0000000080002dc6 0x0000000080002ca0
/home/ubuntu/桌面/xv6-labs-2020/kernel/sysproc.c:62
/home/ubuntu/桌面/xv6-labs-2020/kernel/syscall.c:144
ubuntu@VM5878-LZY:/home/ubuntu/桌面/xv6-labs-2020$ addr2line -e kernel/kernel 0x0000000080002898
/home/ubuntu/桌面/xv6-labs-2020/kernel/trap.c:76
ubuntu@VM5878-LZY:/home/ubuntu/桌面/xv6-labs-2020$ █

```

实验感受

在实验一中由于粗心大意，两个实验做了整整一天。在实验二中，按照提示以及XV6课本，理解系统相关栈以及指针的结构（其实在计嵌已经学了一遍，提示也给了许多）后，第一个问题进展比较顺利。

Alarm

该题的主要目的是让我们增加一个系统调用，该系统调用在使用CPU时间的情况下定期向进程发出警报会在指定时间间隔后调用用户态的函数，因此需要我们对内核态的相关状态进行保存。我们需要正确添加 `sigalarm` 和 `sigreturn` 系统调用，使得 `alarmtest` 能够正确通过。

最后得到的正确输出应该是：

```

$ 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
$ usertests
...
ALL TESTS PASSED
$

```

实验代码

首先 test0 要求实现处理程序的调用，即正确实现打印“alarm”以及调用alarmtest。

正如实验一，我们首先要在文件Makefile中添加alarmtest的调用：

```
$U/_alarmtest\
```

并且更新user / usys.pl、kernel/syscall.h和kernel / syscall.c，以允许alarmtest调用sigalarm和sigreturn系统调用。

```

//kernel/syscall.h
#define SYS_sigalarm 22
#define SYS_sigreturn 23

//kernel/syscall.c
extern uint64 sys_sigalarm(void);
extern uint64 sys_sigreturn(void);
SYS_sigalarm sys_sigalarm,
[SYS_sigreturn] sys_sigreturn,

//user/usys.pl
entry("sigalarm");
entry("sigreturn");

//user/user.h
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);

```

在 kernel/sysproc.c 中，我们增加要用到的两个调用函数，return 0在test0中不会受影响。

```

uint64 sys_sigalarm(void){
    int ticks;
    uint64 handler;
    if( argint(0,&ticks)< 0 || argaddr(1,&handler)<0 ){

```

```

        return -1;
    }
    struct proc *p = myproc();
    p->alarm_interval = ticks;
    p->alarm_handler = (void *) handler;
    return 0;
}

uint64 sys_sigreturn(){
    return 0;
}

```

并且在 `kernel/proc.c` 中做好变量的初始化：

```

static struct proc*

//allocproc(void)
    p->alarm_interval = 0;
    p->ticks_since_last_call = 0;

//freeproc
    p->alarm_interval = 0;
    p->ticks_since_last_call = 0;

```

在// `kernel/trap.c`中，我们把 handler 的地址放入 epc中：

```

if(which_dev == 2){
    struct proc *p = myproc();
    if(p->alarm_interval != 0){
        p->ticks_since_last_call++;
        if(p->ticks_since_last_call >= p->alarm_interval){
            p->trapframe->epc = (uint64) p->alarm_handler;
            p->ticks_since_last_call = 0;
        }
    }
    yield();
}

```

写完上述操作后，运行，发现终端疯狂崩溃（panic: panickernel）。由于实验一backtrack回溯，上面还附带了一长串地址。折腾了一个多小时后，几经崩溃之余，终于发现是没有声明变量~

```

// kernel/proc.h
int alarm_interval;
void (*alarm_handler);
int ticks_since_last_call;

```

一通操作过后，我们可以成功将alarmtest.c编译为xv6用户程序并且运用。进程会一直产生alarm，直到进程的计时器溢出，如图所示：


```
uint64 re_a2;
uint64 re_a3;
uint64 re_a4;
uint64 re_a5;
uint64 re_a6;
uint64 re_a7;
uint64 re_t3;
uint64 re_t4;
uint64 re_t5;
uint64 re_t6;
```

```
// kernel/sysproc.c
uint64 sys_sigreturn(){
    struct proc *p = myproc();
    p->trapframe->epc = p->re_epc;
    p->trapframe->ra = p->re_ra;
    p->trapframe->sp = p->re_sp;
    p->trapframe->gp = p->re_gp;
    p->trapframe->tp = p->re_tp;
    p->trapframe->t0 = p->re_t0;
    p->trapframe->t1 = p->re_t1;
    p->trapframe->t2 = p->re_t2;
    p->trapframe->t3 = p->re_t3;
    p->trapframe->t4 = p->re_t4;
    p->trapframe->t5 = p->re_t5;
    p->trapframe->t6 = p->re_t6;
    p->trapframe->s0 = p->re_s0;
    p->trapframe->s1 = p->re_s1;
    p->trapframe->s2 = p->re_s2;
    p->trapframe->s3 = p->re_s3;
    p->trapframe->s4 = p->re_s4;
    p->trapframe->s5 = p->re_s5;
    p->trapframe->s6 = p->re_s6;
    p->trapframe->s7 = p->re_s7;
    p->trapframe->s8 = p->re_s8;
    p->trapframe->s9 = p->re_s9;
    p->trapframe->s10 = p->re_s10;
    p->trapframe->s11 = p->re_s11;
    p->trapframe->a0 = p->re_a0;
    p->trapframe->a1 = p->re_a1;
    p->trapframe->a2 = p->re_a2;
    p->trapframe->a3 = p->re_a3;
    p->trapframe->a4 = p->re_a4;
    p->trapframe->a5 = p->re_a5;
    p->trapframe->a6 = p->re_a6;
    p->trapframe->a7 = p->re_a7;
    p->re_flag = 0;
    return 0;
}
```

```
// kernel/trap.c 替换前文的修改
if(which_dev == 2){

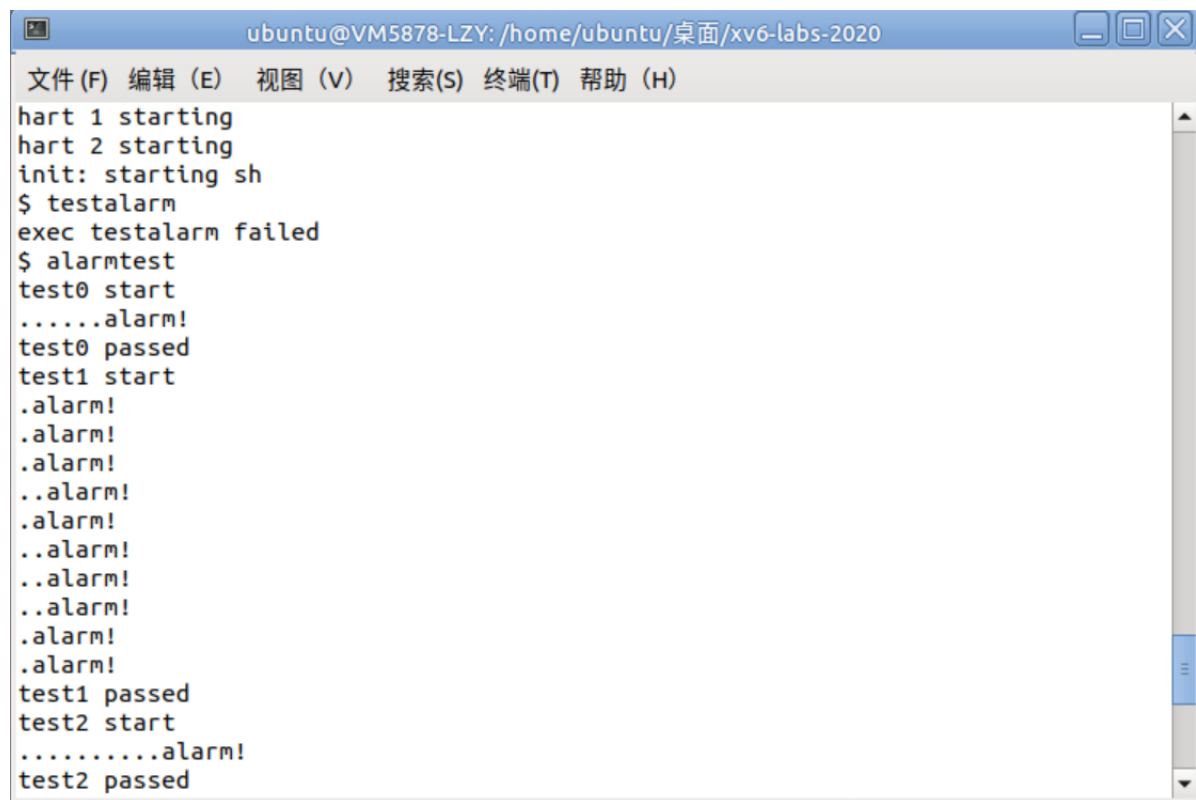
    if(p->alarm_interval != 0){
        p->ticks_since_last_call++;
        if(p->ticks_since_last_call >= p->alarm_interval && p->alarmre_flag == 0){
```

```

    p->re_epc = p->trapframe->epc;
    p->re_ra = p->trapframe->ra;
    p->re_sp = p->trapframe->sp;
    p->re_gp = p->trapframe->gp;
    p->re_tp = p->trapframe->tp;
    p->re_t0 = p->trapframe->t0;
    p->re_t1 = p->trapframe->t1;
    p->re_t2 = p->trapframe->t2;
    p->re_t3 = p->trapframe->t3;
    p->re_t4 = p->trapframe->t4;
    p->re_t5 = p->trapframe->t5;
    p->re_t6 = p->trapframe->t6;
    p->re_s0 = p->trapframe->s0;
    p->re_s1 = p->trapframe->s1;
    p->re_s2 = p->trapframe->s2;
    p->re_s3 = p->trapframe->s3;
    p->re_s4 = p->trapframe->s4;
    p->re_s5 = p->trapframe->s5;
    p->re_s6 = p->trapframe->s6;
    p->re_s7 = p->trapframe->s7;
    p->re_s8 = p->trapframe->s8;
    p->re_s9 = p->trapframe->s9;
    p->re_s10 = p->trapframe->s10;
    p->re_s11 = p->trapframe->s11;
    p->re_a0 = p->trapframe->a0;
    p->re_a1 = p->trapframe->a1;
    p->re_a2 = p->trapframe->a2;
    p->re_a3 = p->trapframe->a3;
    p->re_a4 = p->trapframe->a4;
    p->re_a5 = p->trapframe->a5;
    p->re_a6 = p->trapframe->a6;
    p->re_a7 = p->trapframe->a7;
    p->trapframe->epc = (uint64) p->alarm_handler;
    p->ticks_since_last_call = 0;
    p->re_flag = 1;
}
}
yield();

```

最后，我们通过了test2.

A terminal window titled 'ubuntu@VM5878-LZY: /home/ubuntu/桌面/xv6-labs-2020'. The window contains the following text:

```
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
hart 1 starting
hart 2 starting
init: starting sh
$ testalarm
exec testalarm failed
$ 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
```

实验感受

我刚刚开始写第二问的时候有点急于求成，想一步到位，因此浪费了几个小时。后面在大佬的指点下理清了思路——从test0到test1，一步步实现打印alarm以及中断现场的还原，课本上的知识要搬运到代码上，还是需要相当大的努力。该实验难度极大，不过做完之后满满都是收获。