Lab 实验报告 IV

Lab: Traps

丁睿

dromniscience@gmail.com

更新：2020 年 11 月 13 日

目录

1 任务完成清单

Subtask	Done?	Time
`RISC-V assembly`	Y	10min
`backtrace()`	Y	30min
`alarm()`	Y	3h

Grade: 85/85

2 详细情况 & 困难和收获

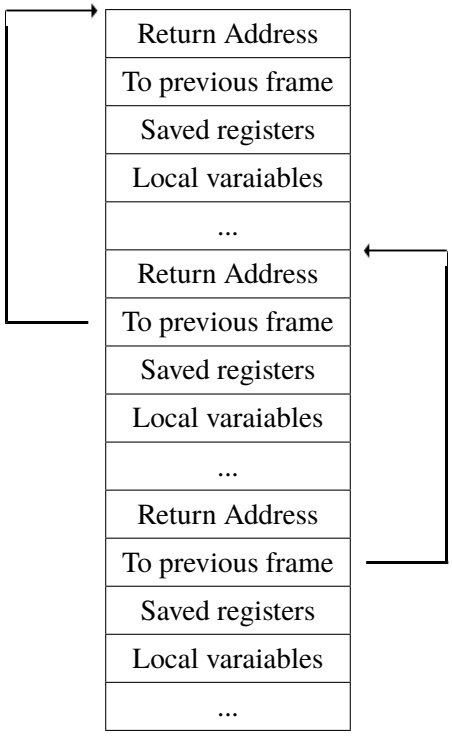
我的作业已开放在这个网页上。

这一次的练习综合了之前 Lab2: System Calls 的内容，并且必须依赖于清晰的对 RISC-V xv6 的 trap 处理流程的理解 (不论是硬件还是软件的)。总而言之，这是一次兼具挑战性和趣味性的实验作业。

我们不再重复具体的 traps 处理流程，而是直接分析我们设计的思路。所有处理中涉及到的细节在参考文献 [1.] 和 [2.] 中均有详细的说明。

2.1 Backtrace

唯一的问题在于理解栈帧的结构以及 xv6 给函数栈分配的大小。它的结构如下图所示。



与 x86-64 相比，我们实现的栈帧有这样几点不同：

1. 总是需要当前栈的基指针 (s0寄存器，它的使用类似于 IA32 的%ebp) 和栈顶指针 (sp寄存器，它的使用类似于 x86-64 的%rsp)。
2. 一个函数的运行时栈总是从返回地址开始，而不像 x86-64 里返回地址将被算做调用者栈帧的一部分。
3. 运行时栈维护一个栈帧的链表。由于 xv6 给每个进程的栈分配一个页做用户栈，因此链表的末尾可由一个指向用户栈虚拟页起始的指针标识。

这个惯例很容易被我们翻译成如下的代码实现。

```
/* kernel/printf.c */
void backtrace(void)
{
    uint64 fp = r_fp(); // Frame start
    uint64 prev;

    /* traverse the linked list */
    while(1){
        if(PGROUNDDOWN(fp) == fp) break;
        prev = *(uint64 *)(fp - 16);
        printptr(*(uint64 *)(fp - 8));
        printf("\n");
        fp = prev;
    }
}
```

2.2 Alarm

理解清楚 Handler 的实现原理是关键的。sys_sigalarm将alarm的间隔 (以下称为ticks) 和一个位于用户空间的handler设置到对应进程的 PCB 中。同时，为了记录剩余描述，必须增加一个计算剩余时钟中断数的tickseleft变量。另外，为了防止handler重入 (即前一次尚未执行完又因为tickseleft用完而再启动一个过程)，必须加上一个in_handler标记。注意，这些变量都是进程私有的，没有必要在访问时加锁互斥。

此后，一旦时钟中断到来[1]，它将在被调度做一些更多的工作——检查是否有一个非空的ticks，如果有并且不在handler中，就对tickseleft减一。如果操作后tickseleft为零，那么就应该调用handler。

请注意，现在该核正工作在内核态下！为了用户和内核的分离 (isolation)，我们必须回到用户空间[2]执行。这里我们做实现了这样一个 trick——修改trapframe的返回地址 (将在usertrapret的sret指令中使用到)epc为handler的地址。由于它没有任何参数，且随后由于usertrapret切

[1]这里我们只考虑从用户态到来的时钟。如果是内核态下到来的时钟，在我们的当前设计下则无法立刻返回用户态执行 handler，因此我们的修改只发生在 usertrap 中。

[2]即特权级设为 user mode，同时satp装入用户页表而非当前的内核页表、sp寄存器从当前的内核栈切回用户栈。

换了页表和栈，sret指令从sepc弹出该地址到pc中，handler就能正确执行，而且不会对内核造成任何威胁。问题是我们维护原来被中断的主进程的运行时状态[3]并且使得handler执行完后能够正常返回。

我们设计了一个惯例——要求handler必须以调用sigreturn实现返回。这样，通过提前保存这些运行时状态到某个结构体中，然后sys_sigreturn设置它们回到原来中断时的状态，一切就能融洽地工作起来。

具体的实现不必多言。我想展示的是代码中微妙的细节。至少有以下几种非常值得讨论的细节：

- 由于执行系统调用时中断是开放的，因此在sys_sigalarm设置ticks，必须满足如下约束：
  - p->ticks必须晚于p->handler被赋值。
    否则如果一个中断在p->ticks设为非零后p->handler还没安装前到来，并且p->ticks恰好是 1，这意味着当前中断后就该执行handler，这在sret返回后会产生不可预料的问题！更糟糕的是，可能内核运行 10000 次都没有遇到这种情况，却在第 10001 次时崩溃了！
  - p->ticks必须晚于p->tickslefte被赋值。
    理由同上。不论使用的是减法器 (即统计据下次调用handler还有几个 ticks) 还是累加器 (即统计据上次调用handler过了几个 ticks) 的系统，都需要注意该问题。
- 类似的理由，当sys_sigalarm由于一个(0,0)的参数列表取消ticks时，必须满足如下约束：
  - p->ticks必须早于p->handler被置零。
  - p->ticks必须早于p->tickslefte被置零。
- 运行时状态的可选性[4]如下表所示。

项目	属性
kernel_sp, kernel_hartid	绝对不能保存！
kernel_satp, kernel_trap	没必要保存
epc	必须保存！
caller-saved registers	必须保存！
callee-saved registers	必须保存！

你会注意到 32 个通用寄存器里我们故意漏掉了x0，原因很显然——它恒零，所以甚至trapframe中都没有它的位置。

我们依次解释各行的理由如下：

- kernel_sp, kernel_hartid 绝对不能保存
  当执行handler时，进程完全可能被中断，然后因为调度被挂起。如果其后上了一个新的核，那么就意味着此时内核栈 (回想一下，RISC-V xv6 中每个核在启动时就分配了从物理地址零开始的内核栈，所以一个核对应一个内核栈) 和hartid必须设置为新的核所对应的那个。如果此时还原为旧的hartid和旧的内核栈，那么将导致两个核产生

[3]简单地系统级分析可以发现，这里仅仅包括寄存器的状态。
[4]我们需要保存的内容全部在 trapframe 中了。所以我们只需要筛选其中必要的内容。

数据冒险！

- kernel_satp，kernel_trap 没必要保存

  在运行过程中，内核页表和 trap 处理的入口地址一般不会发生变化。当然保存一下也没问题。但是如果考虑usertrap的地址可以在系统运行中发生变化，那么kernel_trap将变成不得保存的了，原因是该项必须时刻保持系统的最新值。当然，这里我们不考虑这么复杂的问题，姑且认为它是可选的。

- epc 必须保存！

  这个极为显然。调用handler时，我们就有意地修改掉它的值进入它。现在sys_sigreturn里，我们要还原到原来被中断的下一条指令。

- caller-saved registers 必须保存！

  同样很显然。因为它们的值可以被handler任意覆盖。

- callee-saved registers 必须保存！

  这一点乍看之下令人意外。被调用者保存不是能够确保被中断程序的寄存器返回时被handler还原到进入它时的值吗？

  实际上，调用sigreturn的handler过程并不能知道自己永不返回。编译器会按照正常的顺序写好调用它的ecall指令，然后再写上从栈帧中加载被调用者保存寄存器的指令。但实际上这最后的过程永不能被执行。因此，被调用者寄存器的值在调用sys_sigreturn[5]时是被破坏的。

  如果你想亲自确认这个事实，那我建议去读 user/usertests.asm 中的periodic函数返回之前做的事情。我摘录了最重要的内容如下。

```
/* user/usertests.asm */
void
periodic()
{
   0:   1141                addi    sp,sp,-16
   2:   e406                sd      ra,8(sp)
   4:   e022                sd      s0,0(sp)
   6:   0800                addi    s0,sp,16

     ... # Some codes

  sigreturn();
  2a:   00000097        auipc ra,0x0
  2e:   6ba080e7        jalr  1722(ra) # 6e4 <sigreturn>
}
  32:   60a2              ld      ra,8(sp)
  34:   6402              ld      s0,0(sp)
  36:   0141              addi    sp,sp,16
  38:   8082              ret
```

[5]这是个合适的地方确认sigreturn和sys_sigreturn的区别。请不要混淆两者！前者是系统调用的用户接口，后者是系统调用的实现实体。

3 对课程和 lab 的建议

此 lab 非常值得一做。

4 参考资料

1. *xv6: a simple, Unix-like teaching operating system*
   Russ Cox, Frans Kaashoek, Robert Morris *August 31, 2020*
2. *The RISC-V Reader: An Open Architecture Atlas*
   David Patterson, Andrew Waterman *1st Edition*
3. 现代操作系统 [M]
   A.S.Tanenbaum, H. Bos 著, 陈向群 马洪兵 译, 北京: 机械工业出版社, 2011: 47-95