同濟大學
TONGJI UNIVERSITY

MIT 6.S081 实验报告

Lab 4

2151767 薛树建

日期：2023 年 7 月 20 日

目录

Lab4: traps

1. RISC-V assembly (easy)

1) 实验目的

本实验让我们取阅读一下汇编相关的内容，提供了一个 call.c 函数，通过 make fs.img 指令编译它，并在 call.asm 中生成可读的汇编版本，我们需要根据此来回答一些问题。

2) 实验步骤

a) 哪些寄存器保存函数的参数？例如，在 main 对 printf 的调用中，哪个寄存器保存 13？

```
 1c: 1141                addi  sp,sp,-16
 1e: e406                sd ra,8(sp)
 20: e022                sd s0,0(sp)
 22: 0800                addi  s0,sp,16
printf("%d %d\n", f(8)+1, 13);
 24: 4635                li a2,13
 26: 45b1                li a1,12
```

在 xv6 中 a0-a7 寄存器用来保存参数。在本题中，阅读汇编代码不难发现 a2 寄存器保存了 13

b) `main` 的汇编代码中对函数 `f` 的调用在哪里？对 `g` 的调用在哪里？

实际上 main 的汇编代码中并没有调用这两个函数，而是通过编译优化将它们内联到了 main 函数中（如上图所示直接将 12 保存在了 a1 中）。

c) printf 函数位于哪个地址？

```
28: 00000517            auipc a0,0x0
2c: 7c050513            addi  a0,a0,1984 # 7e8 <malloc+0xea>
30: 00000097            auipc ra,0x0
34: 610080e7            jalr  1552(ra) # 640 <printf>
```

首先 ra 先加上此时的 pc 值，接着与 1552 做运算，其结果为 1552+0x30=1600=0x640,即 printf 函数的地址为 0x640。

d) 在 main 中 printf 的 jalr 之后的寄存器 ra 中有什么值？

Jalr 在计算出跳转地址之后会将返回地址存放在 ra 中，此时 ra 中的值为 pc+4

e) 运行以下代码。

unsigned int i = 0x00646c72;
printf ("H%x Wo%s", 57616, &i);
程序的输出是什么？
输出取决于 RISC-V 小端存储的事实。如果 RISC-V 是大端存储，为了得到相同的输出，你会把 i 设置成什么？是否需要将 57616 更改为其他值？

程序输出是 HE110 World。
57616 的十六进制表示为 e110，而 0x00646c72，从低为到高位以此为 0x72(r)，0x6c(l)，0x64(d)。
小端存储是指低地址对应低字节位，大端则正好相反，高字节位对应低地址位，如果 RISCV 是大端存储，则需要要 i 进行修改位 0x726c6400，57616 不需要改变。

f) 在下面的代码中，“y=”之后将打印什么(注：答案不是一个特定的值）？为什么会发生这种情况？
printf("x=%d y=%d", 3);

输出结果为 1，y 输出的结果其实应该是从 a2 寄存器中获取的值，不固定。

3) 实验中遇到的问题和解决方法

本实验的内容主要为考察一些汇编内容，难度不大，在完成该实验的过程中有一些知识点（比如大小端）有些遗忘，但是基本上可以通过查询资料来完成。

4) 实验心得

通过本实验对一个简单的 c 程序的汇编版本的阅读，我理解了一些函数执行过程中的操作，比如内联优化，对于 call.c 中的 f，g 这类简单函数，直接将其内联到 main 函数中，从而减少了调用开销。

其次是 ra 寄存器，jalr 指令在跳转到相关函数时会通过计算出需要跳转的函数的地址，同时将 pc 值加 4 存在 ra 中，作为函数的返回地址。

还有就是大端小端部分，这时一种编码风格，二者在效率上并没有什么大的差异。

2. Backtrace (moderate)

1) 实验目的

为了方便对内核进行调试，我们可以让内核在发生 panic 时调用一个 backtrace 函数来打印当前的函数调用列表。

本实验的目的便是让我们去实现一个 backtrace 函数来打印当前的函数调用列表，打印出执行函数的地址，从而方便去调试

2) 实验步骤

a) 首先我们需要 kernel/defs.h 下添加 backtrace 函数的声明，并在 sys_sleep 中去调用它。

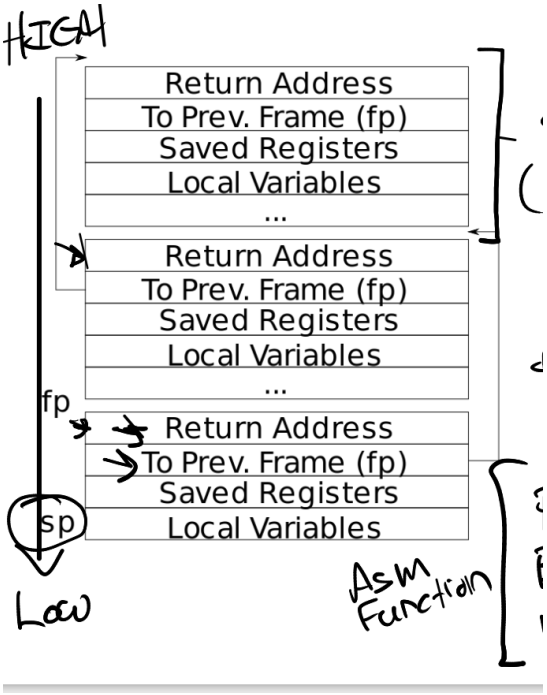
```
void            backtrace(void);//backtrace函数的声明

// proc.c
```

```
  }
  release(&tickslock);
  backtrace();
  return 0;
}
```

b) 随后我们将提示中给出的 r_fp()函数添加到 kernel/riscv.h。此函数的作用为读取当前函数的帧指针。

```
static inline uint64//读取当前函数的帧指针
r_fp()
{
  uint64 x;
  asm volatile("mv %0, s0" : "=r" (x) );
  // 从寄存器 s0（Saved register/frame pointer） 中读取值，并存储到变量 x 中
  return x;
}
```

c) 根据这张课堂笔记的图片，我们可以得出 fp 向下偏移一个地址位置存放的为该函数的返回地址，偏移两个地址位置村存放的是上一级函数帧的地址。



d) 我们可以通过这种方式对其进行遍历，同时通过 PGROUNDUP 与 PGROUNDDOWN 来确定循环遍历的结束条件。函数实现如下：

```
void backtrace(void) {
  printf("backtrace:\n"); // 打印回溯信息的提示
  uint64 fp = r_fp(); // 获取当前函数的帧指针
  uint64* frame = (uint64*)fp;//转为指针类型
  // 使用帧指针来进行遍历
  uint64 up=PGROUNDUP(fp);
  uint64 down=PGROUNDDOWN(fp);
  while (fp < up&&fp>down) {
    printf("%p\n", frame[-1]); // 打印返回地址,偏移-8
    fp=frame[-2];//偏移16获得上一帧指针地址
    frame=(uint64*)fp;
  }
}
```

e) 测试结果如下：

```
$ bttest
backtrace:
0x00000000800021d0
0x0000000080002032
0x0000000080001cd8
```

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ addr2line -e kernel/kernel
0x00000000800021d0
/home/tjxsj/xv6-labs-2021/kernel/sysproc.c:74
0x0000000080002032
/home/tjxsj/xv6-labs-2021/kernel/syscall.c:144
0x0000000080001cd8
/home/tjxsj/xv6-labs-2021/kernel/trap.c:76
```

f) 测试通过以后我们将该函数加入到 panic 函数中。

```
printf(s);
printf("\n");
panicked = 1; // freeze uart output from other CPUs
backtrace();
for(;;)
```

3) 实验中遇到的问题和解决方法

对于该实验，基本上需要用到的东西在提示中都已经给出，跟着提示做的话还是可以的，部分不理解的部分基本上可以通过网络或是别人的笔记来查询，相对而言本实验还是较为容易的。

4) 实验心得

通过本实验我对于 xv6 系统中的函数调用栈有了一定的了解，结合第一个实验的内容，我们发现每次函数在调用之前，都会对 sp 寄存器进行修改，实际上 sp 寄存器存放的便是栈的栈顶地址，每一个调用函数的过程都会入栈。

关于为什么每次减去 16，是因为 xv6 的栈自上向下生长的，它的栈底地址位于高地址部分，每个函数调用的返回地址就存放在该函数帧指针的向下偏移 8 的位置。

当然也不一定每一个函数调用时都 sp 减去 16，有些函数可能需要占用更多的内存。

本实验评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-traps backtrace
make: 'kernel/kernel' is up to date.
== Test backtrace test == backtrace test: OK (1.1s)
```

3. Alarm (hard)

1) 实验目的

本实验的目的便是让我们去实现一个系统调用 sigalarm(interval, handler)，它的功能是当调用它的应用程序消耗的 cpu 时间片达到 interval 之后，便会向内核发送中断请求并调用 handler 指向的函数。

这个功能对于那些不需要占用过多 cpu 时间或者是需要周期性执行的程序时很有必要的。

在开始本实验之前需要先稍微了解一下 xv6 中断的流程。在 lab 时添加系统调用时，在汇编代码中是通过 ecall 指令进入内核，ecall 指令大概只是进行了几件事，将 mode 修改为 supervisor mode(RISCV 为 xv6 提供了三种模式，机器模式，监管者模式与用户模式，xv6 只有在启动时为机器模式，随后会快速跳到监管者模式)，将 pc 值存放在 spec 寄存器中，之后该指令会修改 scause 寄存器的内容(产生中断的原因)，并修改 pc 值为 stvec 寄存器中的内容，stvec 寄存器存放的便是 trampoline 页的首地址，接下来将跳转到此处执行。

在这里执行主要作用便是保存上下文(将用户态下寄存器中的相关内容存放在虚拟地址空间的 TRAPFRAME 中)，以便于在返回用户态时可以恢复现场，同时将内核的页表，栈装入寄存器，并跳转到 usertrap。

Usertarp 函数根据中断类型进行对于处理，我们本次实验需要修改 usertrap 函数。Usertrap 函数结束之后便会调用 usertrapret 与 userret 来进行返回用户态的相关操作。

2) 实验步骤

a) 首先我们 user.h 中声明这两个系统调用的函数，分别用来调用定时器中断处理与从定时器中断处理过程中返回。

```
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);
```

并在 usys.pl 文件中进行相应设置，在 syscall.h，syscall.c 中添加系统调用号与相应函数声明。

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

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

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

```
[SYS_sigalarm] sys_sigalarm,
[SYS_sigreturn] sys_sigreturn
```

b) 随后我们应该在 proc 进程结构体中添加相应的变量去记录有关 alarm 的信息，比如指针地址，已占用 cpu 周期数与 alarm 所需 cpu 周期数等。后两个字段与 test1 和 test2 相关，前者用于记录是否在执行 handler 函数，后者则用来保存上下文。

```
//储存参数
int ticks;
int tickscount;
uint64 handler;
int handler_exec;
struct trapframe* alarmf;
```

c) 随后我们在 allocproc 与 freeproc 中做好相关的释放与初始化。

```
p->tickscount = 0;
p->handler = 0;
p->ticks = 0;
p->handler_exec=0;
if((p->alarmf = (struct trapframe *)kalloc()) == 0){
  release(&p->lock);
  return 0;
}
```

```
  p->tickscount = 0;
  p->handler = 0;
  p->ticks = 0;
  if(p->alarmf)
    kfree((void*)p->alarmf);
}
```

d) 接着我们需要去实现 sys_sigalarm,逻辑还是较为简单的，当某进程调用该函数时，则就对其进程块中的相应字段进行赋值。

```
uint64
sys_sigalarm(void)
{
  int ticks;
  uint64 handler;
  struct proc *p=myproc();
  argint(0,&ticks);
  argaddr(1,&handler);
  p->ticks=ticks;
  p->handler=handler;
  p->tickscount=0;
  return 0;
}
```

e) 由于我们需要每隔一段时间来执行该 handler 函数，接下来我们需要在 usertrap 中进行相应的修改，每次发生时钟中断时，修改该进程的 tickcount 的值，一旦该值大于 ticks，则我们需要保存此时的内核态的上下文（在 proc 中添加新的 trapframe

字段来储存)，同时将寄存器 epc 的值修改为 handler，保证在返回用户态后直接会调用此函数。同时要记得将 tockcount 值置为 0。
handler_exec 也是 proc 中的成员，在这里的作用是起到一个判断 handler 函数是否已被调用的作用，如果它还未返回则不能再被内核调用。(test2 的测试的部分)。

```
if(which_dev == 2){

  if(p->ticks>0){
    p->tickscount++;//统计时钟数
    if(p->tickscount>p->ticks&&p->handler_exec==0){
      p->tickscount=0;
      p->handler_exec=1;
      memmove(p->alarmf,p->trapframe,PGSIZE);
      p->trapframe->epc=p->handler;

    }
  }
  yield();//将控制权转会调度器
}
```

这时示例中的调用的 periodic 函数

```
void
periodic()
{
  count = count + 1;
  printf("alarm!\n");
  sigreturn();
}
```

我们可以看到在该函数结束时调用了系统调用函数 sigreturn，该调用目的是恢复内核的上下文。
实现也较为简单

```
uint64
sys_sigreturn(void){
  struct proc* p=myproc();
  p->handler_exec=0;
  memmove(p->trapframe,p->alarmf,PGSIZE);
  return 0;
}
```

f) 测试如下：

```
$ 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
```

3) 实验中遇到的问题和解决方法

对于该实验，对于时钟中断这部分不是太过了解，但是实际实验过程中其实也并未涉及，只需要在时钟中断的 if 语句，对该进程内核状态进行一些修改就行了。

做实验时，其实我对于中断的处理逻辑还不是很明白，不过这一方面可以通过查看源码于别人的学习笔记来解决。

4) 实验心得

通过该实验，我个人理解 epc 寄存器在中断过程中的作用，同时也了解了 xv6 操作系统对于中断处理的基本逻辑，大致为保存上下文，根据相应中断原因执行逻辑，恢复上下文这一系列流程。

在平时的操作系统理论课上，感觉中断还是挺好理解的，实际上在代码中却异常复杂。

本实验评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-traps alarm
make: 'kernel/kernel' is up to date.
== Test running alarmtest == (5.3s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
```

总结

通过本次实验，我较好地理解了 xv6 操作系统的中断流程，同时对于 trapframe 中储存的一些重要寄存器的值也有了一定的了解。

本实验整体评分如下：

```
tjxsj@LAPTOP-ONAKFDL3:~/xv6-labs-2021$ ./grade-lab-traps
make: 'kernel/kernel' is up to date.
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test == backtrace test: OK (1.5s)
== Test running alarmtest == (4.9s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests == usertests: OK (93.7s)
== Test time ==
time: OK
Score: 85/85
```