Traps 实验报告

专　业：软　件　工　程

指　导　老　师：王　冬　青

年　级：大　学　二　年　级

学　号：2153061

姓　名：谢　嘉　麒

1. RISC-V assembly (easy)

1.1.实验目的

本次实验要求理解Xv6实验提供的汇编代码，从而了解一些关于RISC-V汇编的代码意义和行为。

1.2.实验步骤

1.2.1.生成观察文件

实验首先需要使用make fs.img将user/call.c源文件编译为一个可读的目标版本user/call.asm。

1.2.2.哪些寄存器包含函数的参数？例如，在main对printf的调用中，哪个寄存器保存13?

阅读RISC-V的官方文档，可以得知寄存器a0-a7（即x10-x17）用于存放函数调用的参数；

阅读user/call.c中的printf相关内容，可以看到汇编代码中指令li a2,13将参数13存放在寄存器a2中，即a2用于保存13。

1.2.3.main的汇编代码中对函数f的调用在哪里？调用g在哪里？（提示：编译器可以内联函数。）

在user/call.c中可以得到，printf("%d %d\n",f(8)+1,13)调用函数f，因此在汇编代码中查看相关内容。

在printf相关部分，有语句li a1,12将立即数12存入寄存器a1。然而立即数12是f(8)+1的结果，因此推断编译器的常量优化方式将立即数计算出来填入printf，没有执行f。

而代码段int f(int x)最后在返回值中调用g(x)，但是编译器将函数内联到f中，减少了压栈和跳转开销，在14: 250d addiw a0,a0,3中调用了g。

1.2.4.函数printf位于哪个地址?

在文件user/call.asm中，代码void printf()明确指出640: 711d，即printf的地址为0x640。

1.2.5.“main”中的“jalr”到“printf”之后的寄存器“ra”中的值是多少?

在文件user/call.asm中，代码34: 610080e7 jalr 1552(ra) #640 可以看出ra寄存器中存储的地址是0x38。

1.2.6.运行实验代码并观察结果

将给出代码添加到user/call.c中进行编译运行，输出为：HE110 World。

因为57616的十六进制为E110，同时0x72，0x6c，0x64转换为ASCII码形式为r，l，d。

如果是big-endian的话则将i改成0x726x6400。

1.2.7.观察代码病会带'y='后将要打印的内容

将语句加入user/call.c中进行编译运行，得到'y='后的打印内容为1。

1.3.实验中遇到的问题和解决方法

本次实验主要考验对于汇编代码的阅读和理解，由于笔者此前没有汇编学习的经验，因此利用额外时间进行了初步的汇编语言的学习，对理解问题有了很大帮助。同时也对RISC-V下汇编语言有了进一步的认识。

1.4.实验心得

通过本次实验，笔者对于传统汇编语言和RISC-V下汇编语言都有了一定的了解和学习，同时也对于很多函数和功能的底层操作逻辑有了进一步的认识，以栈，寄存器等硬件视角来全新地观察这些计算机功能，对构建知识体系大有裨益。

2. Backtrace (moderate)

2.1.实验目的

Backtrace对于debug有很大的用处。backtrace是在发生错误的点之上的堆栈上的函数调用列表。实验要求在文件kernel/printf.c中实现函数backtrace()，并且在sys_sleep的调用中插入backtrace()来打印栈。

2.2.实验步骤

2.2.1.阅读汇编代码并确定函数功能

实现backtrace()函数首先需要通过汇编文件来了解函数调用过程。在user/call.asm中阅读main函数，了解到函数调用将返回地址ra和栈基地址s0存入栈。则backtrace()只需要打印每次保存的ra，然后根据s0寻找上一个栈，进一步打印保存的ra，直到完全打印。

2.2.2.读取当前寄存器

首先需要再kernel/riscv.h中加入内联汇编函数来读取当前的寄存器s0，即调用语句：

```
asm volatile("mv %0, s0" : "=r" (x) );
```

函数返回x即可。

2.2.3.实现backtrace()

之后则在printf.c文件中实现函数backtrace()：

（1）首先用myproc()函数来获取当前进程的指针p；

（2）用r_fp()函数（2.2.2.中实现）来获取当前帧指针fp；

（3）循环遍历函数调用栈，直到栈帧指针fp指向当前进程的内核栈的底部，即PGROUNDUP(p->kstack)；

（4）再循环内打印当前栈帧指针fp所指向的返回地址，通过解引用(uint64*)(fp-8)实现；

（5）将栈帧指针fp更新为前一个栈帧的指针，通过解引用(uint64*)(fp-16)来获取。

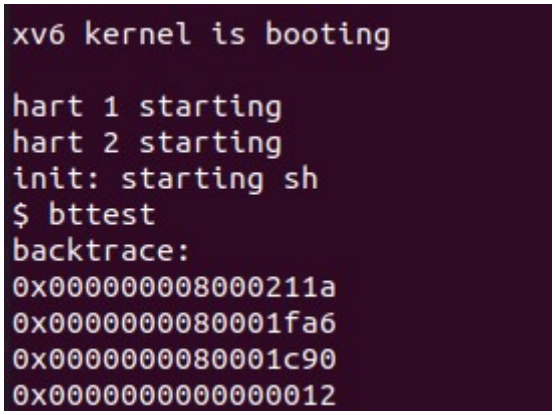
2.2.4.调用和函数声明

在sysproc.c的sys_sleep()函数中调用backtrace()；

注意将backtrace()函数声明在defs.h中；

2.2.5.编译与检测

在完成以上步骤后，进入Xv6目录终端并运行make qemu编译，编译完成后与实验提供的准确结果对比发现无误，因此试验成功，实验结果如下：



2.3.实验中遇到的问题和解决方法

2.3.1.理解函数调用过程

实现backtrace()要求笔者理解函数调用过程栈并确定要打印的关键内容。通过阅读汇编代码并加以理解可以找到关键量ra和栈s0。同时也需要关注内联汇编函数的使用。

2.3.2.循环读栈

在backtrace()函数中进行循环读栈，利用解引用来不断获取fp的位置，并打印其值。

2.4.实验心得

通过本次实验，笔者在backtrace函数的实现过程中进一步学习了汇编相关的内容，也对函数调用的底层实现有了更深层次的认识和理解。

3. Alarm (hard)

3.1.实验目的

本实验要求为Xv6添加一个功能，这一功能会在进程使用CPU时间时定期发出警报。这种设计对于限制占用CPU时间的计算绑定进程，或计算但希望采取定期操作的进程很有用。更广泛地，本实验要求实现用户级中断/错误处理程序的原始形式。

实验需要实现sigalarm(n,fn)系统调用，这一系统调用被用户程序执行后，将会在每消耗n个tick的CPU时间后被中断并且运行给定函数fn。

3.2.实验步骤

3.2.1.系统调用入口

需要在user/user.h中加入两个系统调用的入口sigalarm()和sigreturn()，分别用于设置定时器和从定时器中断处理过程中返回。

3.2.2.定义alarm参数

需要在每个进程控制块中加入sigalarm参数项，即在kernel/proc.h的struct proc中加入int alarminterval; int alarmticks; void (*alarmhandler)(); int sigreturned;四个项。

在添加完项之后，需要在kernel/proc.c中的allocproc()函数中对这些项进行初始化。

3.2.3.实现sigalarm()系统调用

需要在kernel/sysproc.c中实现sys_sigalarm()函数：

（1）定义变量ticks和handler分别保存定时器周期和定时器信号处理函数的地址；

（2）利用myproc()函数来获取当前进程的指针p；

（3）通过系统调用参数获取用户传递的定时器周期ticks和定时器信号处理函数地址handler；

（4）使用argint和argaddr函数从用户空间将参数读取到内核空间；

（5）将定时器周期保存到当前进程alarminterval字段中，定时器将在指定的周期触发；

（6）将定时器信号处理函数地址保存到当前进程的alarmhandler字段中，定时器触发时将调用该函数；

（7）将alarmticks字段设置为0，表示当前定时器未触发；

3.2.4.更新计时器并判断alarm

在kernel/trap.c中的usertrap()中的if(which_dev==2)中加入p->alarmticks+=1；这样即可更新每个已经消耗的tick。

同时可以判断已经使用的CPU时间是否可以触发alarm：

即满足(p->alarmticks >= p->alarminterval) && (p->alarminterval > 0)，则触发alarm，将p->alarmticks重新置0。

3.2.5.备份上下文并调用定时器

在tick到达预定值后，需要将计数器清零并将用户进程跳转到预设的过程fn处。因此需要在struct proc中备份进程当前的上下文，加入alarmtrapframe变量即可；

之后再usertrap()中备份上下文，并将用户进程设置到fn并返回用户态，使用以下语句即可：

```
if (p->sigreturned == 1)
{
p->alarmtrapframe = *(p->trapframe);
p->trapframe->epc = (uint64)p->alarmhandler;
p->sigreturned = 0;
usertrapret();
}
```

3.2.6.fn执行完毕后返回正常执行过程

在kernel/sysproc.c中实现sys_sigreturn()函数，利用之前保存的上下文来实现这一功能。需要在内核态中恢复上下文，然后返回用户态。

（1）先利用myproc()函数获取当前进程并赋予进程指针p；

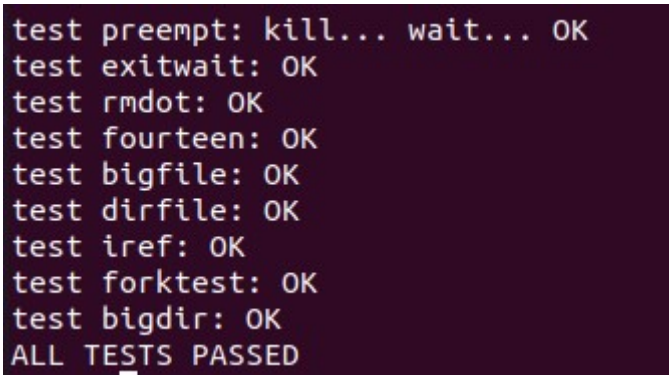
（2）置p->sigreturned=1表示已经返回；

（3）将*(p->trapframe)设置为p->alarmtrapframe，实现进程返回；

（4）利用usertrapret()返回用户态；

3.2.7.编译与检测

在完成以上步骤后哦，进入Xv6目录终端并且执行编译指令make qemu，在终端中运行usertests即可看到预期输出：



3.3.实验中遇到的问题和解决方法

3.3.1.设置合理数据结构存储alarm相关变量

实验需要在struct proc中添加合适的项来记录alarm相关内容，包括定时器周期，定时器信号处理函数和上下文恢复等项。首先要根据alarm的功能来判断整体的需求，再根据需求来完成任务。

3.3.2.并发访问冲突

考虑到存在多个进程同时调用该函数设置定时器，可能会导致竞争条件和并发访问冲突，需要采取合适的同步措施来解决这个问题。

3.4.实验心得

本次实验大体上介绍了RISC-V的中断机制，但是具体的详细内容需要自主进行补充和学习。通过阅读手册和官方文档可以看出，RISC-V主要依靠定时器中断，内核中断和PLIC设备中断等方式来实现中断机制。

通过本次实验和实验之外的学习内容，笔者对于RISC-V的中断机制有了进一步的了解，也在实现定时器中断的过程中对相关的代码与数据结构有了更深刻的认识。