@[TOC](Mit6.S081-实验4-Traps)

# 一、RISC-V assembly

## 1，实验准备

1）阅读xv6 book章节4

2）从user space过渡到kernel space，kernel space返回到user space的汇编代码：kernel/trampoline.S

3）解决所有中断的代码：kernel/trap.c

## 2，实验要求

理解一些RISC-V汇编是很重要的，你已经在6.004中学过。

有个文件user/call.c在xv6代码库中。执行make fs.img，编译call.c文件，并生成一个可读的汇编版本程序。

阅读call.asm中函数g，f 和main的代码。risc-v的指令参考手册在[参考页](https://pdos.csail.mit.edu/6.828/2020/reference.html)。

回答下面问题。

## 3，相关问题

1、哪些寄存器包含函数参数？例如哪个寄存器持有13（在main函数中的printf中）？

a0-a7包含函数参数；a2存储13。

2、在main汇编代码中哪里调用了函数f？哪里调用了函数g？

编译器优化后，无函数调用

3、函数printf位于哪个地址？

0x630

4、在main中jalr到printf后，ra寄存器的值多少？

0x30，执行auipc ra,0x0，将PC寄存器值放到ra中

运行下面代码：

unsigned int i = 0x00646c72;

printf("H%x Wo%s", 57616, &i);

输出：HE110 WORLD

如果risc-v是大端序，i需要设置为0x726c64，57616不需要变

# 二、Backtrace

## 1，实验要求

对于调试来说，有一个backtrace通常是有用的：error发生点之前，栈上的一系列函数调用!

在kernel/printf.c中实现backtrace()函数。在sys\_sleep()中插入对这个函数的调用，然后运行bttest，它调用sys\_sleep。你的输出应该是：

backtrace:

0x0000000080002cda

0x0000000080002bb6

0x0000000080002898

在bttest之后退出qemu。在你的terminal中：地址可能是不同的，但如果你运行addr2line -e kernel/kernel（riscv64-unknown-elf-addr2line -e kernel/kernel）并且把上面地址复制粘贴如下所示：

$ addr2line -e kernel/kernel

0x0000000080002de2

0x0000000080002f4a

0x0000000080002bfc

Ctrl-D

你应该可以看到像下面所示：

kernel/sysproc.c:74

kernel/syscall.c:224

kernel/trap.c:85

编译器在每个栈帧中放入一个帧指针，该指针保留调用方帧指针的地址。你的backtrace应该使用这些帧指针，来遍历stack并且打印每个栈帧中保存的返回地址。

一些提示：

1.添加backtrace原型到kernel/defs.h，以便于你可以在sys\_sleep中调用。

2.GCC编译器存储当前执行函数的帧指针在寄存器S0，添加下面函数到kernel/risc.h：

static inline uint64 r\_fp(){

uint64 x;

asm volatile(“mv %0, s0” : “=r” (x) );

}

3.在backtrace中调用这个函数来读取当前帧指针。这个函数使用内联汇编来读取s0。

4.这些讲义有一个栈帧布局的图片。注意返回地址位于一个固定偏移量（对栈帧的帧指针偏移-8），保存的帧指针位于固定偏移量（对帧指针偏移-16）

5.xv6为每个kernel stack分配一页于页对齐地址。你可以计算栈页的顶部和底部地址，通过使用PGROUNDDOWN(fp)和PGROUNDUP(fp)（看kernel/riscv.h，这些数字对backtrace终止循环是有帮助的）。

一旦你的backtrace起作用，panic（kernel/printf.c）调用它，以便于在panic时可以看到kernel的backtrace。

## 2，具体实现

1）在kernel/defs.h添加定义

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201214165422768.png)

2）修改kernel/riscv.h中增加r\_fp()的实现，用来读取寄存器s0

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201214165519829.png)

3）在kernel/printf.c中增加backtrace()的定义

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201214165633833.png)

4）在kernel/sysproc.c的sys\_sleep()函数中调用backtrace()

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201214165812257.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

## 3，执行效果

在xv6代码库中执行make qemu，xv6中执行bttest，bttest调用sleep(system call)。依次调用trap.c、syscall.c、sysproc.c。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201214170004367.png)

地址换算行号。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201214170621889.png)

# 三、Alarm-test0()：invoke handler

## 1，实验要求

在这个练习中，你将为xv6添加一个特性：

在进程使用cpu时间时，定期发出警报。对于compute-bound进程（想限制它们使用多少cpu时间），或对于既想计算也想采取一些周期性动作的进程，这可能是有用的。

更普遍一些，你将能实现一个原始形式的用户级中断/fault handlers；你可以使用某些相似的东西来处理在应用中的page faults。

如果通过alarmtest和usertests，你的解决方案就是正确的。

你应该添加一个新的sigalarm(interval, handler)system call。如果一个应用调用sigalarm(n, fn)，那么在每n ticks个cpu time（程序花费）后，kernel应该导致应用函数fn被调用。

当fn返回时，应用应该从它离开的地方重新恢复。在xv6中一个trick是一个相当任意的时间单位，取决于硬件定时器多久生成一个中断。

如果一个应用调用sigalarm(0,0)，kernel应该停止生成周期alarm call。

你将找到一个文件user/alarmtest.c在xv6代码库。把它添加到Makefile。它将不会编译成功，直到你添加sigalarm和sigreturn system calls。

alarmtest在test0中调用sigalarm(2, periodic)，来告知kernel每两个tick强制调用periodic()，然后自旋一会。

你可以在user/alarmtest.asm中看到allarmtest的汇编代码，可以用来调试。当alarmtest生成下面输出并且usertests正确运行，你的方案就是正确的。

![在这里插入图片描述](https://img-blog.csdnimg.cn/2020121713432634.png)

当你完成时，你的方案将会只有几行代码，但要把它做对可能很棘手。

我们将用原始版本的alarmtest.c来测试你的代码。你可以更改alarmtest.c来帮你debug，但要确保原始alarmtest让所有test通过。

invoke handler：

修改kernel来跳到user space的alarm handler，将导致test打印”alarm”。现在不必担心输出后发生了什么，如果程序在打印“alarm”后崩溃，现在就可以了。这是一些提示

1.你将需要更改Makefile，来让alarmtest.c被当作一个xv6用户程序被编译。

2.user/user.h中的定义：

![在这里插入图片描述](https://img-blog.csdnimg.cn/2020121713503854.png)

3.更新user/usys.pl（可以生成user/usys.S），kernel/syscall.h，和kernel/syscall.c来允许alarmtest调用调用sigalarm和sigreturn system calls。

4.现在你的sys\_return应该只返回0

5.你的sys\_sigalarm()应该存储alarm interval和handler function的指针，在proc结构的新field中（kernel/proc.h）。

6.在上一次进程的alarm handler调用后（或在下次调用前），你将需要保持跟踪度过了多少个ticks；你将也需要一个新field在struct proc。你可以初始化proc fields在proc.c中的allocproc()中。

7.每个tick，硬件时钟强制一个中断，被在kernel/trap.c的usertrap()中处理

8.你仅想在定时器中断时操纵进程的tick，您想要某些如下所示：

if(which\_dev == 2) ...

9.仅当进程有明显的定时器时，才调用alarm函数。注意user alarm函数地址可能是0（例如：在user/alarmtest.asm，periodic在地址0）.

10.你将需要更改usertrap()，以便于当一个进程的alarm interval到期时，用户进程执行handler函数。当一个RISC-V的trap返回到用户空间时，什么决定指令地址（用户空间代码恢复执行）？

11.如果你告诉qemu仅使用一个cpu，用gdb看traps会更简单，用以下方式： make CPUS=1 qemu-gdb

12.如果alarmtest打印“alarm”那么你就成功了

## 2，具体实现

1）修改Makefile

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217135142621.png)

2）修改user/usys.pl

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217135247392.png)

3）修改user/user.h

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217135403664.png)

4）修改kernel/syscall.h

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217140141367.png)

5）修改kernel/syscall.c

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217140318278.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

6）在kernel/proc.h

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217135658470.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

7）在kernel/sysproc.c

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217135751923.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

8）在kernel/trap.c

![在这里插入图片描述](https://img-blog.csdnimg.cn/2020121714045734.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

## 3，测试效果

启动xv6，执行alarmtest，test0通过

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217140723616.png)

# 三、Alarm-test1/test2(): resume interrupted code

## 1，实验要求

风险是：alarmtest在打印”alarm” 后，或 打印“test1 failed”后，或alarmtest退出（却没打印”test1 passed”）之后，crashes in test0或test1。

为了修复这个，你必须确保：当alarm handler完成后，控制返回到用户程序被timer interrupte打断的指令处。

你必须确保：寄存器内容恢复为中断时持有的值，因此用户程序可以在alarm之后继续保持无干扰。最后，你应该让alarm counter在每次执行完置0，因此handler可以周期性的调用。

一开始时，我们已经为你做了一个设计决定：在alarm函数调用完后，user alarm handlers需要去调用sigreturn system call。看一下alarmtest.c中的periodic作为一个例子。这意味着你可以在usertrap和sys\_sigreturn添加代码（联合起来让用户进程在处理完alarm之后正确恢复）。

一些提示：

1. 你的方案将需要你存储并恢复寄存器---什么寄存器需要你保存和恢复，来正确地重新恢复中断代码？

2. usertrap保存足够状态在struct proc，sigreturn可以正确地返回到中断的用户代码

3. 阻止重复调用handler---如果一个handler仍未返回，kernel不该再次调用它。test2测试这个

一旦你通过 test0、test1和test2，执行usertests确认你没有影响其他kernel部分

## 2，具体实现

1）修改kernel/proc.h

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217161822394.png)

2）修改kernel/proc.c

allocproc()：

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217161923697.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

freeproc():

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217164224583.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

3）修改kernel/trap.c

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217163154530.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217163213280.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

4）修改kernel/defs.h

![在这里插入图片描述](https://img-blog.csdnimg.cn/2020121716335166.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217163416526.png)

5）在kernel/sysproc.c

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217163524565.png)

## 3，测试效果

启动xv6，执行alarmtest，test0、test1、test2通过

![在这里插入图片描述](https://img-blog.csdnimg.cn/20201217164337973.png)