Lab 4 Traps

目录

[1. RISC-V assembly (easy) 1](#_Toc174491131)

[1.1 实验目的 1](#_Toc174491132)

[1.2 实验步骤 1](#_Toc174491133)

[1.3 问题解答 4](#_Toc174491134)

[1.4 遇到的问题 6](#_Toc174491135)

[1.5 实验心得 6](#_Toc174491136)

[2. Backtrace (moderate) 7](#_Toc174491137)

[2.1 实验目的 7](#_Toc174491138)

[2.2 实验步骤 7](#_Toc174491139)

[2.3 实验结果 9](#_Toc174491140)

[2.4 实验中遇到的问题和解决方法 9](#_Toc174491141)

[2.5 实验心得 9](#_Toc174491142)

[3. Alarm (hard) 10](#_Toc174491143)

[3.1 实验目的 10](#_Toc174491144)

[3.2 实验步骤 10](#_Toc174491145)

[3.3 实验结果 14](#_Toc174491146)

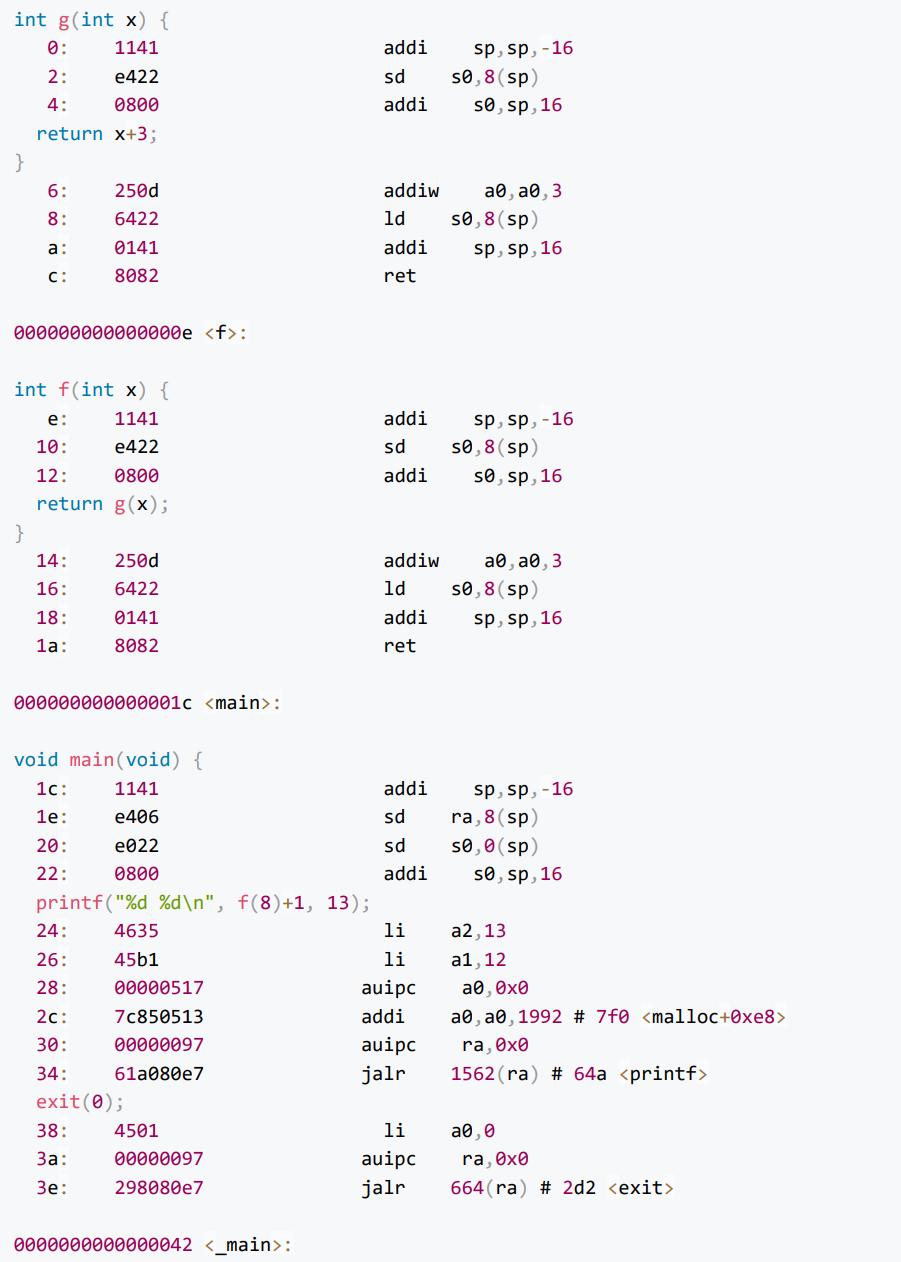
[3.4 遇到的问题 14](#_Toc174491147)

[3.5 实验心得 15](#_Toc174491148)

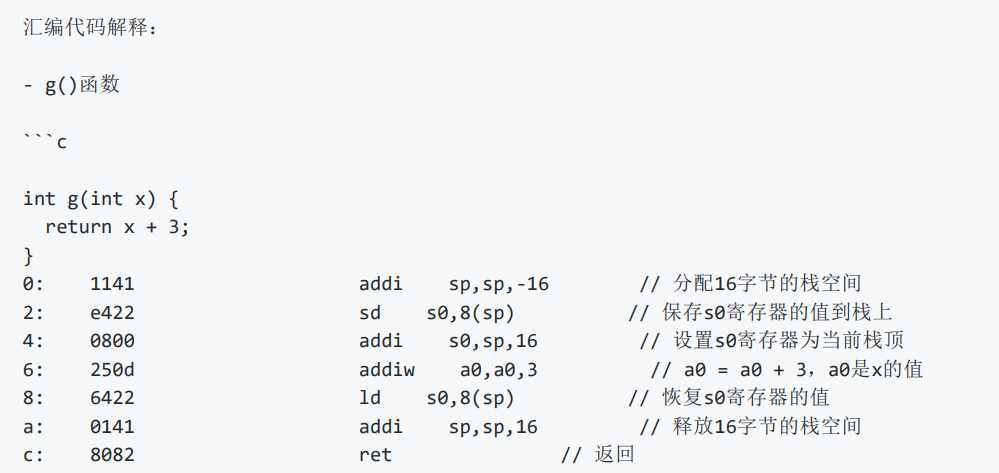
[4. 测试结果 16](#_Toc174491149)

RISC-V assembly (easy)

* 1. 实验目的
* 理解使用陷阱实现系统调用的基本概念和原理
* 熟悉RISC-V汇编语言，特别是与函数调用和参数传递相关的寄存器使用
* 掌握在汇编代码中定位函数调用和了解函数地址的方法
* 理解RISC-V的字节序（小端序）对程序输出的影响，并能够根据字节序进行数值调整
* 理解格式化输出函数printf的用法和参数传递机制，并能够解析printf函数的输出结果
* 理解缺少参数传递时printf函数的行为，并能解释产生的输出结果
  1. 实验步骤
* make fs.img 对文件 user/call.c进行编译，在 user/call.asm 中生成程序的可读汇编版本
* 阅读 user/call.asm中的main函数、g、f函数



* g ( )函数



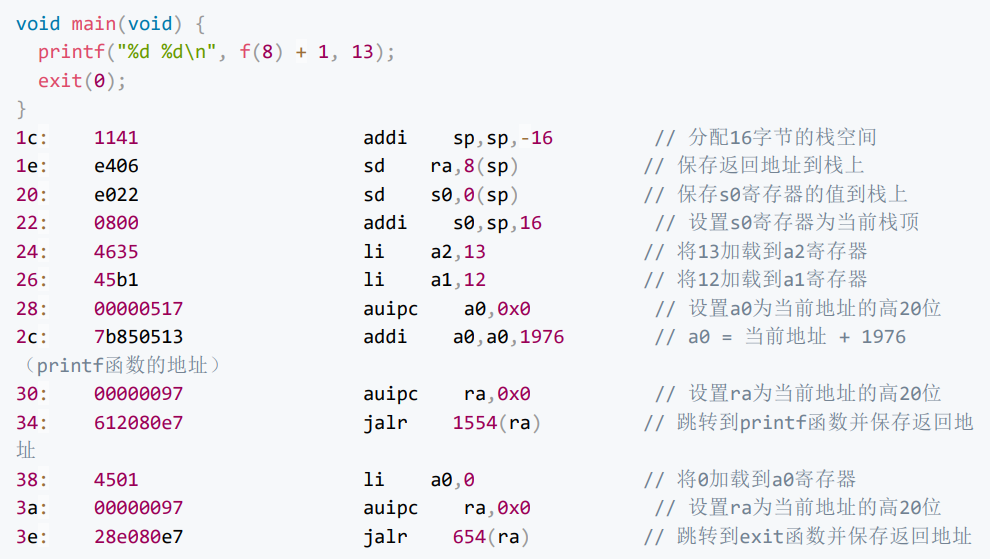
通过 addi 和 sd 指令，g 函数首先为新的栈帧分配空间，并保存调用者的 s0 寄存器值。然后通过 addiw 指令计算 x + 3 并将结果存储在 a0 寄存器中（RISC-V中 a0 用来传递返回值）。最后恢复 s0 的值，释放栈空间，并返回调用者。

* f ( )函数



f 函数通过相同的方式分配栈空间并保存寄存器。由于编译器内联了 g 函数的代码，直接在 f 函数中进行了 x + 3 的计算。最后恢复栈空间和寄存器并返回。

* main ( )函数



main 函数首先分配栈空间并保存 ra 和 s0 寄存器的值。 li 指令加载常数值到寄存器（这里是13和12）。 auipc 和 addi 指令计算 printf 函数的地址并加载到 a0 寄存器。 jalr 指令跳转到printf 函数，并在返回时继续执行后续代码。最后通过 li 和 jalr 指令调用 exit 函数以结束程序。

* 1. 问题解答
* 问题：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？
* 解答：前八个函数参数依次存放在 a0 到 a7 寄存器中。在 main 函数对 printf 的调用中，参数13保存在 a2 寄存器中，参数12（即 f(8) + 1 的结果）保存在 a1 寄存器中。
* 问题：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?(提示：编译器可能会将函数内联）
* 回答：在 main 函数中，并没有直接调用 f 和 g 的指令，而是直接对寄存器 a1 进行了加载指令：这意味着 main 函数中对 f 和 g 的调用已被内联，因此没有显式的调用指令。 f 和 g 的代码被直接插入到了 main 函数中，进行计算。
* 问题：printf 函数位于哪个地址？
* 回答：34: 612080e7 jalr 1554(ra) // 跳转到printf函数并保存返回地址因此地址为ra+1554
* 问题：在 main 中 printf 的 jalr 之后的寄存器 ra 中有什么值？
* 回答：jalr 指令位于地址 0x34 。 jalr 执行后， ra 将保存 0x34 + 4 的值，即 0x38 ，因为RISC-V指令长度固定为4字节。所以，在 main 函数中调用 printf 函数后， ra 寄存器中的值是0x38 。
* 问题：运行以下代码，指出程序的输出，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？



* 回答：小端的输出结果为 He110 World ,这个代码中的 i 在小端存储下表示为72 6C 64 00 （从低字节到高字节）,在大端存储下表示为：00 64 6C 72 （从高字节到低字节）,为了让输出结果保持不变，我们需要将 i 重新设置为在大端存储下与原来在小端存储相同的数值。因此，在大端存储下，需要将 i 设置为 0x726C6400 ,57616不需要变化。
* 问题：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？



* 回答：printf() 会从 a2 寄存器中读取第二个参数作为 y 的值，寄存器 a2 中存储的值是无法预估的。
  1. 遇到的问题
* 问题：

理解RISC-V汇编指令的含义和操作数格式：RISC-V指令集有许多不同的指令和操作数格式，我作为初学者在一开始感到很困惑。

* 解决方法：

仔细阅读RISC-V指令集手册，参考相关的教学资料和示例代码，以便更好地理解每个指令的作用和使用方法。

* 问题：

解析printf函数的输出结果：printf函数是一个格式化输出函数，输出的结果可能包含不同的格式符和参数。我一开始理解输出结果很困难，后来查阅资料后才明白需要解析格式符并将参数正确地插入到格式字符串中。

* 解决方法：

仔细阅读printf函数的文档和格式化字符串的规则，以及了解不同格式符对应的参数类型和打印方式。

* 1. 实验心得

在这次实验中，我深入理解了RISC-V架构的系统调用机制，熟悉了汇编语言特别是函数调用和寄存器使用。通过分析字节序的影响，我学会了如何在不同字节序下保持程序输出的一致性。此外，我还掌握了 printf 函数的参数传递机制，理解了编译器的内联优化。总体而言，这次实验加深了我对底层操作和汇编语言的理解，提升了调试和解决复杂问题的能力。

Backtrace (moderate)

* 1. 实验目的
* 理解回溯（backtrace）的概念：回溯是指获取函数调用堆栈的信息，即了解函数被调用的顺序和层次关系。通过回溯，可以追踪到错误发生的位置，对调试和错误定位非常有用。
* 学习帧指针和堆栈布局：在编译器生成的执行代码中，每个函数调用都会创建一个新的栈帧，其中包含有关函数调用的信息。通过理解帧指针和堆栈布局，可以定位到每个栈帧的关键信息，例如返回地址和保存的帧指针。
* 实现回溯函数：在操作系统内核中实现回溯函数，需要使用汇编和内联汇编语言，以及了解编译器对栈帧的布局和访问规则。
* 加深对操作系统内核的理解：通过实现回溯函数，需要对操作系统内核的结构和执行过程有更深入的理解。了解内核中的函数调用关系和堆栈布局有助于理解操作系统的工作原理和内部机制。
  1. 实验步骤
* 在 kernel/defs.h 中添加 backtrace() 的声明

// printf.c

void printf(char\*, ...);

void panic(char\*) \_\_attribute\_\_((noreturn));

void printfinit(void);

void backtrace(void);

* 在 kernel/sysproc.c 中的 sys\_sleep 函数中添加对 backtrace() 的调用

uint64

sys\_sleep(void)

{

int n;

uint ticks0;

backtrace();

argint(0, &n);

if (n < 0)

n = 0;

acquire(&tickslock);

ticks0 = ticks;

while (ticks - ticks0 < n) {

if (killed(myproc())) {

release(&tickslock);

return -1;

}

sleep(&ticks, &tickslock);

}

release(&tickslock);

return 0;

}

* 将提供的 r\_fp() 函数添加到 kernel/riscv.h 中

static inline uint64

r\_fp()

{

uint64 x;

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

return x;

}

* 在kernel/printf.c中实现名为backtrace()的函数

void backtrace(void)

{

uint64 fp = r\_fp();

uint64 boundary = PGROUNDUP(fp);

printf("backtrace:\n");

while (fp < boundary) {

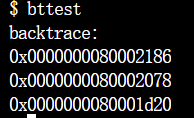
printf("%p\n", \*((uint64\*)(fp - 8)));

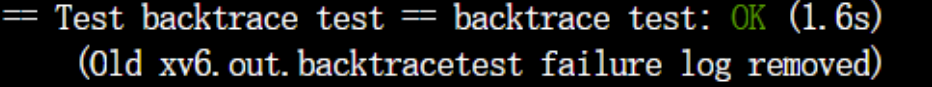
fp = \*((uint64\*)(fp - 16));

}

}

* 1. 实验结果





* 1. 实验中遇到的问题和解决方法
* 问题：

无法获取正确的回溯输出

* 解决方法：

仔细检查代码以后我发现这个是由于计算帧指针、返回地址错误导致的。所以在计算的过程中，我们应确保计算正确并且确保正确地访问和打印返回地址

* 1. 实验心得

通过此次实验，我深入理解了回溯（backtrace）的概念，掌握了通过栈帧获取函数调用堆栈信息的方法。学习了帧指针和堆栈布局后，我能够准确定位每个栈帧中的关键信息。在实现回溯函数的过程中，我加深了对操作系统内核结构和函数调用关系的理解，这对错误定位和调试大有帮助。

Alarm (hard)

* 1. 实验目的
* 理解信号处理，理解操作系统如何通过信号与应用程序进行通信，以及如何响应特定事件或条件
* 理解信号处理，了解到中断是如何触发、处理和分发的，以及如何在中断处理程序中保存和恢复寄存器状态
* 熟悉进程上下文切换，学习如何保存和恢复进程的寄存器状态，以确保在处理完中断或信号后，进程能够正确恢复执行
  1. 实验步骤
* 修改Makefile以编译 alarmtest.c 为xv6用户程序
* 在 user/user.h 中添加以下声明：

int sigalarm(int ticks, void (\*handler)());

int sigreturn(void);

* 更新 user/usys.pl （生成 user/usys.S ）， kernel/syscall.h ，和 kernel/syscall.c 以允许 alarmtest 调用 sigalarm 和 sigreturn 系统调用

// user/usys.pl

entry("sigalarm");

entry("sigreturn");

// kernel/syscall.h

entry("sigalarm");

entry("sigreturn");

// kernel/syscall.h

extern uint64 sys\_sigalarm(void);

extern uint64 sys\_sigreturn(void);

static uint64(\*syscalls[])(void) = {

[SYS\_fork] sys\_fork,

[SYS\_exit] sys\_exit,

[SYS\_wait] sys\_wait,

[SYS\_pipe] sys\_pipe,

[SYS\_read] sys\_read,

[SYS\_kill] sys\_kill,

[SYS\_exec] sys\_exec,

[SYS\_fstat] sys\_fstat,

[SYS\_chdir] sys\_chdir,

[SYS\_dup] sys\_dup,

[SYS\_getpid] sys\_getpid,

[SYS\_sbrk] sys\_sbrk,

[SYS\_sleep] sys\_sleep,

[SYS\_uptime] sys\_uptime,

[SYS\_open] sys\_open,

[SYS\_write] sys\_write,

[SYS\_mknod] sys\_mknod,

[SYS\_unlink] sys\_unlink,

[SYS\_link] sys\_link,

[SYS\_mkdir] sys\_mkdir,

[SYS\_close] sys\_close,

// added

[SYS\_sigalarm] sys\_sigalarm,

[SYS\_sigreturn] sys\_sigreturn,

};

* 在 kernel/proc.h 中为 proc 结构体添加新的字段以存储报警间隔和处理函数的指针

struct proc {

...

int interval; // 间隔

int ticks; // Tick数

uint64 handler; // 处理函数

struct trapframe\* trapframe\_saved; // 用来保存和还原寄存器状态

};

* 在 proc.c 的 allocproc() 中初始化这些新的字段

static struct proc\* allocproc(void)

{

...

found:

...

p->interval = 0;

p->ticks = 0;

p->handler = 0;

return p;

}

* 在 kernel/sysproc.c 中添加 sys\_sigalarm() 系统调用

uint64 sys\_sigalarm(void)

{

int interval;

uint64 handler;

struct proc\* p = myproc();

argint(0, &interval);

argaddr(1, &handler);

p->interval = interval;

p->handler = handler;

return 0;

}

* 修改 kernel/trap.c 的 usertrap() 函数，以在每次时钟中断时检查是否需要调用报警处理函数

void

usertrap(void)

{

int which\_dev = 0;

if ((r\_sstatus() & SSTATUS\_SPP) != 0)

panic("usertrap: not from user mode");

// send interrupts and exceptions to kerneltrap(),

// since we're now in the kernel.

w\_stvec((uint64)kernelvec);

struct proc\* p = myproc();

// save user program counter.

p->trapframe->epc = r\_sepc();

if (r\_scause() == 8) {

// system call

if (killed(p))

exit(-1);

// sepc points to the ecall instruction,

// but we want to return to the next instruction.

p->trapframe->epc += 4;

// an interrupt will change sepc, scause, and sstatus,

// so enable only now that we're done with those registers.

intr\_on();

syscall();

}

else if ((which\_dev = devintr()) != 0) {

// ok

}

else {

printf("usertrap(): unexpected scause %p pid=%d\n", r\_scause(), p->pid);

printf(" sepc=%p stval=%p\n", r\_sepc(), r\_stval());

setkilled(p);

}

if (killed(p))

exit(-1);

// give up the CPU if this is a timer interrupt.

if (which\_dev == 2)

yield();

// give up the CPU if this is a timer interrupt.

if (which\_dev == 2) {

if (p->interval != 0) {

p->ticks += 1;

if (p->ticks == p->interval) {

p->ticks = 0;

if (p->trapframe\_saved == 0) {

p->trapframe\_saved = (struct trapframe\*)kalloc();

memmove(p->trapframe\_saved, p->trapframe, sizeof(\*p ->trapframe\_saved));

p->trapframe->epc = p->handler;

}

}

}

yield();

}

usertrapret();

}

usertrapret();

}

* 在 kernel/sysproc.c 中添加 sys\_sigreturn() 系统调用，用于恢复寄存器状态并释放空间

uint64 sys\_sigreturn(void)

{

struct proc\* p = myproc();

if (p->trapframe\_saved) {

memmove(p->trapframe, p->trapframe\_saved, sizeof(\*p->trapframe\_saved));

kfree((void\*)p->trapframe\_saved);

p->trapframe\_saved = 0;

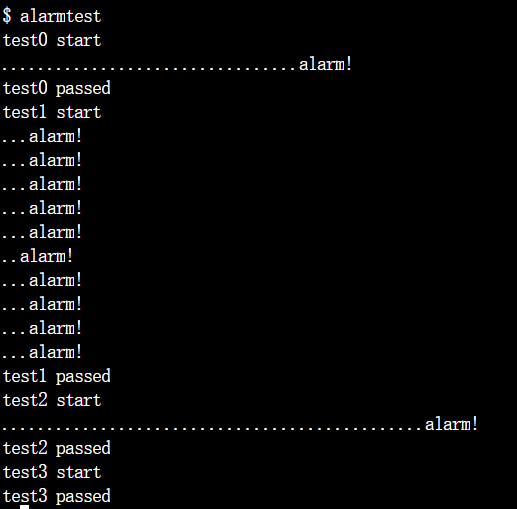
}

p->ticks = 0;

return p->trapframe->a0;

}

* 1. 实验结果



* 1. 遇到的问题
* 问题：

在修改 usertrap 函数以支持中断恢复时，遇到寄存器状态恢复错误的情况。

* 解决方法：

在处理中断时，要确保正确保存中断发生前的寄存器状态，以便在中断处理程序完成后能够正确恢复。在保存和恢复寄存器状态时，需要注意所有需要保存和恢复的寄存器，以确保中断发生前后的代码执行环境保持一致。初次之外，为了避免中断处理程序重入，可以使用标志或互斥锁来保护中断处理程序的执行，用来确保同一时间只有一个中断处理程序在执行。

* 1. 实验心得

我深入理解了操作系统中的信号处理机制，包括如何通过信号与应用程序进行通信以及如何处理中断。通过实现 sigalarm 和 sigreturn 系统调用，我学会了如何在中断处理中保存和恢复寄存器状态，从而确保进程在处理完中断后能够正确恢复执行。此外，实验让我熟悉了进程上下文切换的细节，增强了对操作系统底层机制的理解

测试结果

