Lab4-Traps

RISC-V assembly

实验目的

- 理解RISC-V汇编语言
- 掌握函数参数传递机制，分析函数调用过程，理解函数返回地址寄存器的使用
- 理解字节序的影响，分析格式化输出的行为

实验步骤

1. 执行`make fs.img`编译`user/call.c`，阅读生成的`user/call.asm`文件中函数f,g,mian的汇编代码：

```
int g(int x) {
   0:    1141                    addi    sp,sp,-16
   2:    e422                    sd    s0,8(sp)
   4:    0800                    addi    s0,sp,16
  return x+3;
}
   6:    250d                    addiw    a0,a0,3
   8:    6422                    ld    s0,8(sp)
   a:    0141                    addi    sp,sp,16
   c:    8082                    ret

000000000000000e <f>:

int f(int x) {
   e:    1141                    addi    sp,sp,-16
  10:    e422                    sd    s0,8(sp)
  12:    0800                    addi    s0,sp,16
  return g(x);
}
  14:    250d                    addiw    a0,a0,3
  16:    6422                    ld    s0,8(sp)
  18:    0141                    addi    sp,sp,16
  1a:    8082                    ret

000000000000001c <main>:

void main(void) {
  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
  28:    00000517              auipc    a0,0x0
  2c:    7c850513              addi    a0,a0,1992 # 7f0 <malloc+0xe8>
  30:    00000097              auipc    ra,0x0
  34:    61a080e7              jalr    1562(ra) # 64a <printf>
  exit(0);
  38:    4501                    li    a0,0
  3a:    00000097              auipc    ra,0x0
  3e:    298080e7              jalr    664(ra) # 2d2 <exit>

0000000000000042 <_main>:
```

汇编代码解释：

- g()函数

```c

int g(int x) {
  return x + 3;
}
0:    1141                    addi    sp,sp,-16        // 分配16字节的栈空间
2:    e422                    sd    s0,8(sp)          // 保存s0寄存器的值到栈上
4:    0800                    addi    s0,sp,16         // 设置s0寄存器为当前栈顶
6:    250d                    addiw    a0,a0,3          // a0 = a0 + 3，a0是x的值
8:    6422                    ld    s0,8(sp)          // 恢复s0寄存器的值
a:    0141                    addi    sp,sp,16         // 释放16字节的栈空间
c:    8082                    ret                // 返回
```

通过`addi`和`sd`指令，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

```c
int f(int x) {
  return g(x);
}
e:    1141                    addi    sp,sp,-16        // 分配16字节的栈空间
10:    e422                    sd    s0,8(sp)          // 保存s0寄存器的值到栈上
12:    0800                    addi    s0,sp,16         // 设置s0寄存器为当前栈顶
14:    250d                    addiw    a0,a0,3          // 内联了g函数的代码：a0 = a0
+ 3
16:    6422                    ld    s0,8(sp)          // 恢复s0寄存器的值
18:    0141                    addi    sp,sp,16         // 释放16字节的栈空间
1a:    8082                    ret                // 返回
```

`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

```
void main(void) {
  printf("%d %d\n", f(8) + 1, 13);
  exit(0);
}
1c:    1141                    addi    sp,sp,-16          // 分配16字节的栈空间
1e:    e406                    sd    ra,8(sp)           // 保存返回地址到栈上
20:    e022                    sd    s0,0(sp)           // 保存s0寄存器的值到栈上
22:    0800                    addi    s0,sp,16          // 设置s0寄存器为当前栈顶
24:    4635                    li    a2,13              // 将13加载到a2寄存器
26:    45b1                    li    a1,12              // 将12加载到a1寄存器
28:    00000517                auipc    a0,0x0          // 设置a0为当前地址的高20位
2c:    7b850513                addi    a0,a0,1976       // a0 = 当前地址 + 1976
（printf函数的地址）
30:    00000097                auipc    ra,0x0          // 设置ra为当前地址的高20位
34:    612080e7                jalr    1554(ra)         // 跳转到printf函数并保存返回地
址
38:    4501                    li    a0,0               // 将0加载到a0寄存器
3a:    00000097                auipc    ra,0x0          // 设置ra为当前地址的高20位
3e:    28e080e7                jalr    654(ra)          // 跳转到exit函数并保存返回地址
```

`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。

2. 问题解答

- **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？**

  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。

- **`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)**

  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：

  ```
  26:    45b1                    li    a1,12              // 将12加载到a1寄存器
  ```

  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。

- **`printf` 函数位于哪个地址?**

  `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

- **在 `main` 中 `printf` 的 `jalr` 之后的寄存器 `ra` 中有什么值?**

  `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` 。

- **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?**

```
unsigned int i = 0x00646c72;
printf("H%x Wo%s", 57616, &i);
```

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

- **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);
```

`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。

实验中遇到的困难和解决办法

汇编代码接触的比较少，RISC-V汇编语言和课内看到的汇编语言有些区别，通过查询资料和询问大语言模型获得解答

实验心得

在这次实验中，我学习了RISC-V汇编语言中的关键概念，包括函数参数传递、函数调用和内存访问。通过详细阅读user/call.asm文件，我理解了汇编代码如何与C代码交互，认识到RISC-V采用小

端字节序，这对输出结果有重要影响。我学会了阅读和解释汇编代码的基本方法，理解了函数调用的细节，以及内联函数对代码执行的影响。

Backtrace

实验目的

- 实现 backtrace 函数，在特定函数中调用 backtrace
- 确保正确识别堆栈帧边界
- 在内核 panic 时调用 backtrace

实验步骤

1. 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);
```

2. 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;
}
```

3. 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;
}
```

4. 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));
    }
}
```

5. 测试

运行make qemu后，执行bttest

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
0x0000000080002132
0x000000008000201e
0x0000000080001d14
```

运行make grade:

```
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (4.0s)
```

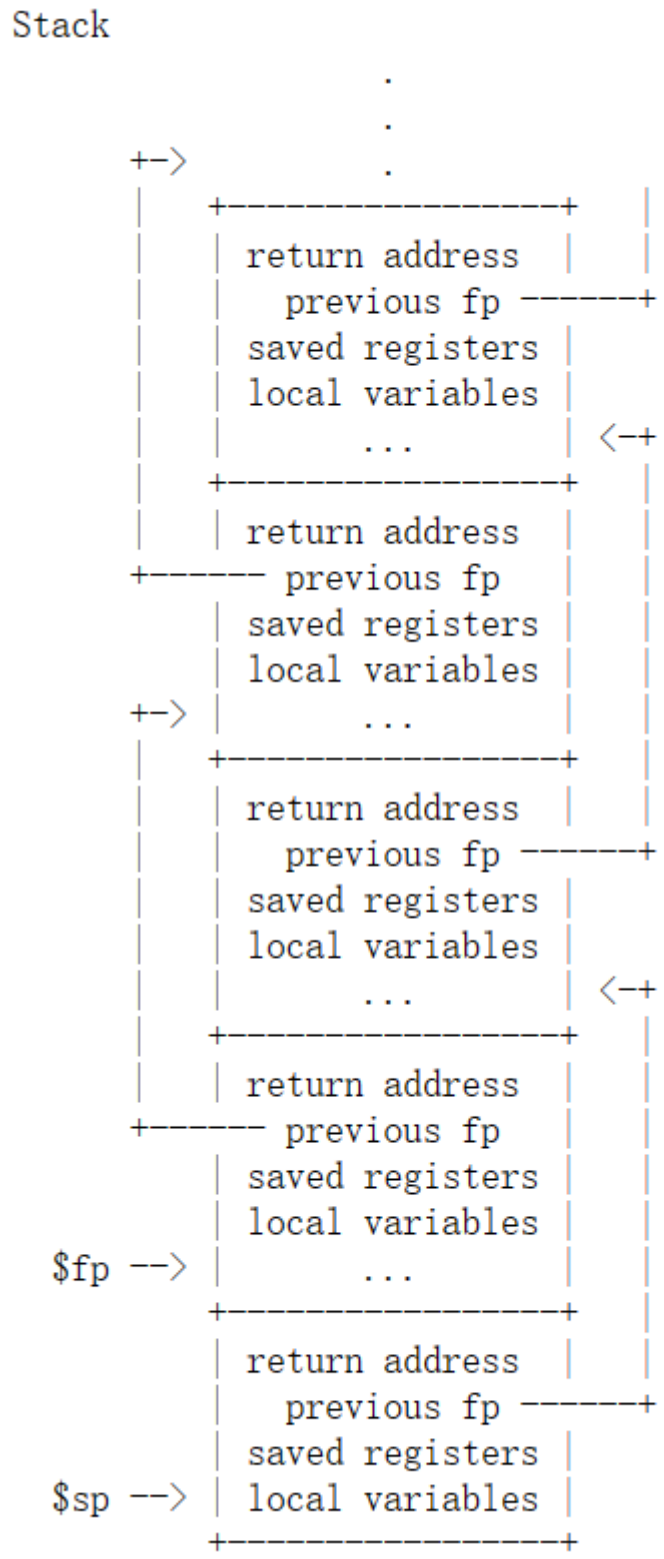
运行`addr2line -e kernel/kernel`，并输入上面回溯的三个地址：

```
0x0000000080002132
0x000000008000201e
0x0000000080001d14/root/workspace/personal_data/zwc/Xv6/Lab4-Traps/kernel/sysproc.c:58
/root/workspace/personal_data/zwc/Xv6/Lab4-Traps/kernel/syscall.c:141

/root/workspace/personal_data/zwc/Xv6/Lab4-Traps/kernel/trap.c:76
```

实验中遇到的问题和解决办法

实现backtrace()函数时刚开始比较没有思路，不知道如何向前回溯。解决办法是阅读教程，学习提供的栈结构图



若 `FP` 寄存器的值是地址值，那么 `FP-8` 就是返回地址的值， `FP-16` 就是上一个栈指针的地址值

实验心得

在此次实验中，通过实现 `backtrace()` 函数并调试内核调用栈，我深刻体会到了对底层计算机系统结构和工作原理的理解的重要性。核心的收获在于如何利用帧指针（frame pointer）和栈帧结构进行堆栈回溯，以调试内核代码。在实验初期，面对如何向前回溯调用栈的挑战时，我感到十

分困惑。然而，通过仔细研究教程和提供的栈结构图，我逐步理解了栈帧的布局和帧指针的作用。每个栈帧包含返回地址和前一个帧指针，这两个关键值使得我们能够从当前帧指针开始，逐步回溯到前一个栈帧，并提取每个函数调用的返回地址。

通过这个实验，我不仅实现了 `backtrace()` 函数，还加深了对栈结构、帧指针和内联汇编的理解。

Alarm

实验目的

- 实现一个定期报警（alarm）功能
- 添加一个新的系统调用 `sigalarm(interval, handler)`
- 修改 `alarmtest.c` 使其能够验证新系统调用的正确性，并确保其通过所有的测试用例

实验步骤

1. 修改Makefile以编译 `alarmtest.c` 为xv6用户程序。

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_alarmtest\
```

2. 在 `user/user.h` 中添加以下声明：

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

3. 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,
};
```

4. 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
};
```

5. 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;
}
```

6. 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;
}
```

7. 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();
}
```

8. 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;
}
```

9. 测试

   运行`make qemu`后执行`alarmtest`：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ 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
test3 start
test3 passed
```

运行`make grade`：

```
== Test running alarmtest ==
$ make qemu-gdb
(6.1s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test   alarmtest: test3 ==
  alarmtest: test3: OK
```

实验中遇到的问题和解决办法

本次实验中需要添加的声明和定义比较多，容易忽略，并且难以排查，主要是有关系统调用的声明，解决办法是重新回顾了一下第二个大实验，梳理清楚添加调用的流程。

实验心得

在本次实验中，我实现一个定期报警（alarm）功能深入理解了用户级中断处理机制的实现，尤其是进程状态的保存与恢复这一关键环节。在实现`sigalarm`系统调用时，我遇到了如何保存和恢复进程寄存器、程序计数器和堆栈指针等状态的挑战。通过本次实验，我深刻体会到细节和工具在系统编程中的重要性。

总结

本实验旨在通过实现和调试中断和陷阱机制，加深对操作系统底层工作原理的理解。具体目标包括理解RISC-V汇编语言、实现函数调用堆栈回溯、实现定期报警功能，并确保这些功能在Xv6操作系统中正确工作。

- RISC-V assembly
  - **目的**:
    - 理解RISC-V汇编语言，掌握函数参数传递机制，分析函数调用过程。
    - 理解字节序的影响，分析格式化输出的行为。
  - **心得**： 通过阅读和分析RISC-V汇编代码，理解了函数调用和参数传递机制，学会了如何解析汇编代码，理解了小端字节序对输出结果的影响。
- Backtrace

- **目的**:
  - 实现 `backtrace` 函数，在特定函数中调用 `backtrace` ，确保正确识别堆栈帧边界。
  - 在内核 `panic` 时调用 `backtrace` 。
- **心得**： 通过实现 `backtrace()` 函数，理解了栈帧的布局和帧指针的作用，掌握了如何进行堆栈回溯和调试内核调用栈，提升了对底层计算机系统结构和工作原理的理解。

- Alarm
  - **目的**:
    - 实现一个定期报警（alarm）功能，添加 `sigalarm` 和 `sigreturn` 系统调用。
    - 修改 `alarmtest.c` 验证新系统调用的正确性，并确保通过所有测试用例。
  -
  - **心得**： 通过实现定期报警功能，深入理解了用户级中断处理机制的实现，掌握了进程状态的保存与恢复。遇到系统调用声明和定义的相关问题，通过重新梳理添加调用的流程，解决了这些问题，提升了系统编程的能力。