# Lab04: Trap

## RISC-V assembly (easy)

### 1) 实验目的

本实验的目的是通过分析 RISC-V 汇编代码，理解函数调用和参数传递机制，并掌握如何从汇编代码中提取信息及调试程序。

### 2) 实验步骤

1. **阅读汇编代码**：
   - 查看 `user/call.c` 的源代码，并通过 `make fs.img` 编译生成汇编代码 `user/call.asm`。
2. **回答问题**：
   - 分析 `call.asm` 文件中的汇编代码，回答以下问题。

### 3) 问题及解答

1. **函数的参数存在哪些寄存器中？例如，main 中调用 printf 时，哪个寄存器保存了 13？**

   在 RISC-V 中，函数参数存储在寄存器 `a0` 到 `a7` 中。例如，在 main 函数中调用 printf 时，寄存器 `a0` 保存了值 13。

2. **在 main 的汇编代码中，函数 f 的调用在哪里？g 的调用在哪里？**

   可以通过查看 `call.asm` 文件中的 main 函数来找到这些函数调用的位置。需要注意，编译器可能会将函数内联，因此函数调用可能不会以显式的 `jal` 指令出现。

3. **printf 函数位于什么地址？**

   printf 函数的地址可以通过在 `call.asm` 文件中找到 printf 函数定义的位置来确定。

4. **在 main 中调用 printf 之后，寄存器 ra 中的值是什么？**

   在执行 `jalr` 指令跳转到 printf 之后，寄存器 `ra` 中保存了返回地址，即从 printf 返回到 main 的地址。

5. **运行以下代码：**

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

   **输出是什么？**

   由于 RISC-V 是小端存储，因此输出将是：

   ```
   HE WoRd
   ```

如果 RISC-V 是大端存储，`i` 应该设置为 `0x726c6400`，而 `57616` 的值不需要改变。

6. **在以下代码中，`y=` 之后将打印什么?**

```
printf("x=%d y=%d", 3);
```

**为什么会这样?**

由于 `printf` 的格式字符串中只有两个 `%d` 占位符，但实际参数提供了三个值（3、未定义的值），因此第三个值（未定义的值）将导致未定义的行为。`y=` 后的值将是随机的，取决于栈中残留的数据。

### 4) 实验心得

通过本次实验，我加深了对 RISC-V 汇编语言的理解，特别是在函数调用、参数传递及内存表示方面。实验过程中通过解析汇编代码，我学会了如何从汇编代码中提取有用的信息，并理解了不同的存储模式（如小端和大端）对输出结果的影响。通过这些实践，我提升了调试程序的能力，也对系统编程有了更深入的认识。

## Backtrace (Moderate)

### 1) 实验目的

本实验的目的是实现一个 `backtrace()` 函数，用于在错误发生时打印函数调用栈。这对于调试非常有用，因为它可以帮助我们确定错误发生的上下文。

### 2) 实验步骤

1. **实现 `backtrace()` 函数**:
   - 在 `kernel/printf.c` 中实现 `backtrace()` 函数。该函数将使用帧指针（frame pointer）来遍历调用栈，并打印每个栈帧中的返回地址。
2. **在 `sys_sleep` 中调用 `backtrace()`**:
   - 在 `sys_sleep` 函数中插入对 `backtrace()` 函数的调用。这将允许我们在执行 `sys_sleep` 时输出调用栈。
3. **测试和验证**:
   - 编译并运行 `bttest`，该测试会调用 `sys_sleep`。检查输出是否符合预期。
   - 使用 `addr2line` 工具将返回地址转换为源代码中的文件名和行号，以验证 `backtrace()` 输出的准确性。

### 3) 实验代码

1. **在 `kernel/defs.h` 中添加 `backtrace()` 的原型**:

```
void backtrace(void);
```

2. **在 `kernel/printf.c` 中实现 `backtrace()` 函数**：

```
#include "riscv.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"

static inline uint64 r_fp() {
  uint64 x;
  asm volatile("mv %0, s0" : "=r" (x));
  return x;
}

void backtrace(void) {
  uint64 fp = r_fp();
  uint64 top_of_stack = PGROUNDDOWN(fp);
  uint64 bottom_of_stack = PGROUNDUP(fp);

  cprintf("backtrace:\n");
  while (fp >= top_of_stack && fp < bottom_of_stack) {
    uint64 ra = *(uint64*)(fp - 8);
    cprintf("%p\n", ra);
    fp = *(uint64*)(fp - 16);
  }
}
```

3. **在 `sys_sleep` 中调用 `backtrace()`**：

```
int sys_sleep(void) {
  int n;
  if (argint(0, &n) < 0)
    return -1;
  backtrace();  // 添加此行以输出调用栈
  sleep(n);
  return 0;
}
```

4. **编译并运行测试**：
   - 编译 xv6，并运行 `bttest` 测试程序：

     ```
     make fs.img
     qemu-system-riscv64 -drive file=fs.img,format=raw
     ```

   - 观察输出并记录返回地址。

5. **使用 `addr2line` 工具验证返回地址**：
    - 退出 QEMU 后，在终端中运行以下命令来将返回地址转换为文件名和行号：

```
addr2line -e kernel/kernel
```

### 4) 实验中遇到的困难和解决办法

1. **帧指针读取问题**：
    - **困难描述**：读取帧指针寄存器 `s0` 时，可能遇到寄存器值不正确或编译错误。
    - **解决办法**：确保 `r_fp()` 函数中使用的内联汇编代码正确读取寄存器，并确保头文件包含正确。
2. **栈范围计算问题**：
    - **困难描述**：计算栈的顶部和底部时，可能由于对 `PGROUNDDOWN` 和 `PGROUNDUP` 宏的误解而导致错误。
    - **解决办法**：仔细检查宏的定义，确保计算出的栈范围正确覆盖了所有栈帧。
3. **输出验证问题**：
    - **困难描述**：返回地址可能不匹配源代码中的行号。
    - **解决办法**：确保在 `addr2line` 中使用了正确的二进制文件，并且 `backtrace()` 函数正确地读取和输出地址。

### 5) 实验心得

通过本次实验，我深入理解了操作系统中的栈帧结构和函数调用的工作原理。实现 `backtrace()` 函数让我学会了如何遍历调用栈并打印栈帧中的返回地址，这对于调试复杂的系统问题非常有帮助。在调试过程中，使用 `addr2line` 工具验证返回地址的准确性，使我更好地理解了如何将低级的地址映射到源代码中的具体位置。总体而言，这次实验提升了我对系统底层工作原理的理解，也增强了我在实际调试中的能力。

## Alarm (Hard)

### 1) 实验目的

本实验的目的是在 xv6 中实现一个定时器报警功能，使得用户进程能够在每隔一定的 CPU 时间片（tick）后调用一个用户定义的处理函数。通过实现 `sigalarm` 和 `sigreturn` 系统调用，用户进程可以在每次处理函数调用后恢复到原来的执行位置，从而实现周期性操作或其他功能。

### 2) 实验步骤

1. **添加 `sigalarm` 和 `sigreturn` 系统调用**：
    - **在 `user/user.h` 中声明系统调用**：

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

- **更新** `user/usys.pl`，生成系统调用相关的汇编代码：

```
SYSCALL(sigalarm)
SYSCALL(sigreturn)
```

- **在** `kernel/syscall.h` **和** `kernel/syscall.c` **中添加系统调用处理函数**：

```
// syscall.h
int sys_sigalarm(void);
int sys_sigreturn(void);

// syscall.c
extern int sys_sigalarm(void);
extern int sys_sigreturn(void);

static int (*syscalls[])(void) = {
  // other syscalls...
  [SYS_sigalarm]    sys_sigalarm,
  [SYS_sigreturn]   sys_sigreturn,
};
```

2. **实现** `sys_sigalarm` **和** `sys_sigreturn` **系统调用**：

- **在** `kernel/proc.h` **中添加新的进程字段**：

```
struct proc {
  // existing fields...
  uint alarm_ticks;     // Number of ticks until next alarm
  void (*alarm_handler)();  // Pointer to alarm handler function
  uint alarm_interval;  // Interval in ticks for the alarm
  int alarm_active;     // Flag to indicate if alarm is active
};
```

- **在** `kernel/proc.c` **中初始化新的字段**：

```
void proc_init(struct proc *p) {
  // existing initialization...
  p->alarm_ticks = 0;
  p->alarm_handler = 0;
  p->alarm_interval = 0;
  p->alarm_active = 0;
}
```

- **实现** `sys_sigalarm` **和** `sys_sigreturn` **函数**：

```
int sys_sigalarm(void) {
  int ticks;
  void (*handler)();

  if (argint(0, &ticks) < 0 || argptr(1, (char**)&handler,
sizeof(handler)) < 0)
    return -1;

  struct proc *p = myproc();
  p->alarm_ticks = ticks;
  p->alarm_handler = handler;
  p->alarm_interval = ticks;
  p->alarm_active = 1;
  return 0;
}

int sys_sigreturn(void) {
  return 0;  // For now, just return 0.
}
```

3. **在 `usertrap` 中处理定时器中断**:
   - **修改 `usertrap` 函数来处理定时器中断**:
4. **更新 `sigreturn` 函数以恢复进程状态**:
   - **在 `kernel/printf.c` 中实现 `sigreturn`**:

```
int sys_sigreturn(void) {
  struct proc *p = myproc();
  struct trapframe *tf = (struct trapframe*)p->trapframe->sp;
  memmove(p->trapframe, (void*)tf, sizeof(*tf));
  p->trapframe->sp += sizeof(*tf);  // Adjust stack pointer
  return 0;
}
```

5. **测试和验证**:
   - **在 `user/alarmtest.c` 中测试 `sigalarm` 和 `sigreturn`**:

```
#include "user.h"
#include "types.h"

void periodic(void) {
  printf("alarm!\n");
  sigreturn();
}

int main(void) {
```

```
  printf("test0 start\n");
  sigalarm(2, periodic);
  while(1);
  printf("test0 passed\n");
  exit(0);
}
```

- **更新 `Makefile` 以编译 `alarmtest.c`**：
- **运行测试**：

```
make fs.img
qemu-system-riscv64 -drive file=fs.img,format=raw
```

- **检查输出是否符合预期**：

```
$ alarmtest
test0 start
........alarm!
test0 passed
```

### 3) 实验中遇到的困难和解决办法

1. **定时器中断处理**：
   - **困难描述**：在处理定时器中断时，确保正确保存和恢复进程状态。
   - **解决办法**：仔细检查 `usertrap` 函数，确保在处理中断时正确保存和恢复寄存器状态。
2. **返回地址和栈帧处理**：
   - **困难描述**：确保在处理完定时器中断后正确恢复用户空间的执行位置。
   - **解决办法**：在 `usertrap` 和 `sys_sigreturn` 中正确保存和恢复栈帧状态，以保证用户程序的正确执行。
3. **测试和调试**：
   - **困难描述**：测试 `sigalarm` 和 `sigreturn` 时可能出现程序崩溃或不按预期运行。
   - **解决办法**：使用调试工具（如 `gdb`）来逐步检查和验证系统调用的实现，确保所有边界情况都得到处理。

### 4) 实验心得

通过本次实验，我深入理解了用户进程中断处理的机制以及如何实现定时器报警功能。实现 `sigalarm` 和 `sigreturn` 系统调用，让我学会了如何在操作系统中管理和恢复用户进程的状态。在处理定时器中断时，正确地保存和恢复寄存器状态是确保系统稳定运行的关键。通过这次实验，我提高了对操作系统底层机制的理解，也增强了在实际开发中的调试和问题解决能力。