同济大学操作系统课程设计 - Xv6 - Lab: traps

【实验目的】

【实验环境】

- **虚拟机：**VMare Workstation 17
- **操作系统：**Ubuntu-20.04.6
- **实验系统：**xv6-labs-2021

【实验内容】

要启动实验室，需要先切换到 traps 分支：

```
$ git fetch
$ git checkout traps
$ make clean
```

1 RISC-V assembly

xv6仓库中有一个文件`user/call.c`。执行`make fs.img`编译它，可在`user/call.asm`中生成可读的汇编版本。阅读`call.asm`中函数`g`、`f`和`main`的代码。RISC-V的使用手册在参考页上。回答下面的问题：

1. **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？**

   在 RISC-V 架构中，函数的前 8 个参数通过寄存器 a0 到 a7 传递。对于 `main` 函数中的 `printf` 调用，如果 `printf` 的第三个参数是 `13`，它会被存放在寄存器 `a2` 中。寄存器 `a0` 和 `a1` 分别存放前两个参数（例如，第一个是格式化字符串，第二个是另一个参数）。

2. **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？**

   在 RISC-V 汇编中，如果编译器进行了优化，`f` 和 `g` 函数可能会被内联，这意味着它们的代码直接被嵌入到 `main` 函数中，而不是通过函数调用的方式。这样，汇编代码中不会有对 `f` 和 `g` 的明确调用，而是会看到直接计算的指令。例如，如果 `f(8)` 的值被计算为 `9`，这个结果会直接在 `main` 中使用，而不通过 `f` 的调用。

3. **printf 函数位于哪个地址？**

   在这个例子中，`printf` 函数的地址为 `0x630`。RISC-V 使用 `auipc` 和 `jalr` 指令组合来计算跳转地址。`auipc` 指令将一个偏移量加到当前的 PC 上，而 `jalr` 则完成跳转。具体计算是通过将 `auipc` 结果加上一个立即数偏移来确定 `printf` 的地址。例如，如果 `auipc` 生成的基地址是 `0x600`，然后通过 `jalr` 加上 `0x30`，最终地址为 `0x630`。

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

   在 `printf` 调用之后，寄存器 `ra` 保存的是返回地址，即在 `printf` 执行完毕后，程序返回到 `main` 函数继续执行的位置。假设 `jalr` 指令使 PC 跳转到了 `printf` 的地址 `0x630`，返回地址会是 `jalr` 指令之后的一条指令的地址。在这个例子中，`ra` 中的值是 `0x38`，这是 `main` 函数中的下一条指令地址。

5. **运行以下代码。`unsigned int i = 0x00646c72; printf("H%x Wo%s", 57616, &i);` 程序的输出是什么?**

   程序输出为 `He110 World`。这是由于 RISC-V 是小端存储的架构，即数据在内存中以反向字节序排列。`i` 的值 `0x00646c72` 在内存中的存储顺序为 `72 6c 64 00`，被作为字符串处理时就是 `"rl"` 之后紧跟一个空字符，因此 `Wo%s` 部分输出 `"World"`。

2 Backtrace

1. **在 `kernel/defs.h` 中添加 `backtrace` 函数的原型**
   为了在 `sys_sleep` 和 `panic` 等函数中调用 `backtrace`，需要在 `kernel/defs.h` 中声明该函数的原型。

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

2. **添加内联汇编函数 `r_fp`**
   GCC 编译器将当前正在执行的函数的帧指针（frame pointer）存储在寄存器 `s0` 中。为了读取这个值，需要在 `kernel/riscv.h` 中添加一个内联函数 `r_fp`：

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

3. **在 `kernel/printf.c` 中实现 `backtrace` 函数**
   `backtrace` 函数的目的是遍历调用堆栈中的帧指针，输出每个栈帧中的返回地址。首先获取当前函数的栈帧指针，然后设置栈的上界和下界，避免遍历超出用户栈的范围。在循环中，逐步读取并打印栈帧中的返回地址。

```
void backtrace(void)
{
  printf("backtrace:\n");
  uint64 fp = r_fp();//读取当前帧指针的值
  while (fp < PGROUNDUP(fp)) {
    printf("%p\n", *(uint64*)(fp-8));  //输出当前栈帧
    fp = *(uint64*)(fp - 16);//更新帧指针 fp_address
  }
}
```

4. **在 `sys_sleep` 函数中调用 `backtrace`**
   为了在 `sys_sleep` 中打印出调用栈，在函数末尾添加对 `backtrace` 的调用。

```
uint64 sys_sleep(void) {
    // ...
    release(&tickslock);
    backtrace();  // 添加对 backtrace 的调用
    return 0;
}
```

5. **在 `kernel/printf.c` 的 `panic` 函数中调用 `backtrace`**
   当发生 panic 时，可以使用 `backtrace` 打印出当前的调用栈，以便调试和分析问题。

```
void panic(char *s) {
    // ...
    backtrace();  // 在 panic 时调用 backtrace
    panicked = 1; // 停止其他 CPU 的输出
    for(;;);
}
```

6. **编译并运行 xv6**
   保存所有修改后，运行 `make qemu` 以编译并启动 xv6。

7. **在命令行中输入 `bttest`**
   在 xv6 的命令行中输入 `bttest`，这会触发 backtrace 并输出当前的调用栈地址：

```
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab4$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-de
vice,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
0x0000000080002a7c
0x0000000080002a20
0x0000000080001d74
$
```

8. **通过 `addr2line` 工具解析调用栈**
   退出 xv6 后，运行 `addr2line -e kernel/kernel`，将 `bttest` 的输出作为输入，以解析出调用栈中每个地址对应的源码文件和行号。发现这些地址对应的正是 `backtrace` 函数在 `sysproc.c`、`syscall.c` 和 `trap.c` 中的调用栈。

3 Alarm

您应该添加新的 `sigalarm（interval， handler）` 系统调用。 如果应用程序调用 `sigalarm（n， fn）`， 则在程序消耗的每 `n`“tick”的 CPU 时间之后，内核 应调用应用程序函数 `fn`。当 `fn` 返回时，应用程序 应该从上次中断的地方继续。刻度是 xv6 中的时间，由硬件计时器生成的频率决定 中断。 如果应用程序调用 `sigalarm（0， 0）`， 则内核应 停止生成定期警报呼叫。

您将在 xv6 中找到一个文件 `user/alarmtest.c` 存储 库。将其添加到 Makefile 中。它无法正确编译直到您添加了 `sigalarm` 和 `sigreturn` 系统调用（见下文）。

`alarmtest` 调用 `test0` 中的 `sigalarm（2， periodic）` 以请求内核强制调用 到 `periodic（）` 每 2 个时钟周期，然后旋转一段时间。你 可以在 user/alarmtest.asm 中看到 alarmTest 的汇编代码，其中 可能便于调试。当 `alarmtest` 生成这样的输出并且 usertests 也运行时，您的解决方案是正确的

1. **在 `Makefile` 中添加 `alarmtest`**
   在 `Makefile` 中添加 `alarmtest`，以便将 `alarmtest.c` 作为 xv6 用户程序进行编译。确保 `Makefile` 包含如下内容：

```
UPROGS=\
    $U/_alarmtest\
```

2. **在 `user/user.h` 中声明系统调用入口**
   在 `user/user.h` 文件中添加两个系统调用的声明：`sigalarm` 用于设置定时器，`sigreturn` 用于从定时器中断处理过程中返回：

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

3. **更新 `user/usys.pl` 生成 `user/usys.S`**
   在 `user/usys.pl` 文件中添加用户态库函数的入口，以生成 `user/usys.S`。添加相应的系统调用条目

4. **在 `syscall.h` 中声明 `sigalarm` 和 `sigreturn` 的用户态库函数**
   在 `syscall.h` 中添加 `sigalarm` 和 `sigreturn` 的系统调用声明：

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

5. **在 `syscall.c` 中添加系统调用处理函数**
   实现 `sigalarm` 和 `sigreturn` 的系统调用处理逻辑。在 `syscall.c` 中添加相应的系统调用处理函数：

```
uint64 sys_sigalarm(void) {
    // 实现 sigalarm 处理逻辑
}

uint64 sys_sigreturn(void) {
    return 0;  // 返回零
}
```

6. **在 `kernel/proc.h` 中更新 `proc` 结构体**
   在 `proc` 结构体中添加用于存储警报间隔和处理函数指针的新字段：

```
struct proc {
    // ...
    int alarm_interval;  // 警报间隔
    void (*alarm_handler)(void);  // 处理函数指针
    int ticks;  // 从上次处理函数调用以来经过的 ticks
};
```

   同时在 `sys_sigreturn` 中返回零

7. **在 `proc.c` 的 `allocproc` 函数中初始化 `proc` 字段**
   在 `allocproc()` 函数中初始化 `proc` 结构体中的新字段：

```
static struct proc* allocproc(void) {
    // ...
    p->alarm_interval = 0;
    p->alarm_handler = 0;
    p->alarm_ticks = 0;
    // ...
}
```

8. **在 `sysproc.c` 中实现 `sys_sigalarm` 和 `sys_sigreturn`**
   实现 `sys_sigalarm` 系统调用逻辑，设置警报间隔和处理函数；实现 `sys_sigreturn` 恢复现场：

```
uint64 sys_sigalarm(void) {
    int interval;
    void (*handler)(void);

    if (argint(0, &interval) < 0 || argaddr(1, (uint64*)&handler) < 0)
        return -1;

    struct proc *p = myproc();
    p->alarm_interval = interval;
    p->alarm_handler = handler;
    p->alarm_ticks = 0;

    return 0;
}

uint64 sys_sigreturn(void) {
    return 0;  // 恢复现场
}
```

9. **在 `kernel/trap.c` 的 `usertrap` 函数中处理定时器中断**
   在 `usertrap()` 中处理定时器中断，更新进程的 `alarm_ticks`，并检查是否达到定时器间隔：

```
void usertrap(void) {
    // ...
    if (r_scause() == 5) {  // 检查是否为定时器中断
        struct proc *p = myproc();
        p->alarm_ticks++;
        if (p->alarm_ticks >= p->alarm_interval && p->alarm_handler) {
            p->alarm_ticks = 0;
            // 保存上下文并调用处理函数
            p->alarm_handler();
            // 恢复现场
            sys_sigreturn();
        }
    }
    // ...
}
```

10. **在 `sysproc.c` 中增加 `cpytrapframe` 函数声明**
    在 `sysproc.c` 中添加 `cpytrapframe` 函数的声明：

```
void cpytrapframe(struct trapframe * , struct trapframe *);
```

11. **编译并运行 xv6**
    保存所有修改后，在终端中运行 `make qemu` 编译并启动 xv6。
12. **在命令行中输入 `alarmtest`**
    在 xv6 的命令行中输入 `alarmtest` 以测试定时器和处理函数的功能：

```
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
$
```

13. 在Makefile 中添加`$U/_usertests\`，运行`usertests`：

```
hart 2 starting
hart 1 starting
init: starting sh
$ usertests
usertests starting
test MAXVAplus: usertrap(): unexpected scause 0x000000000000000f pid=6
            sepc=0x0000000000002208 stval=0x0000004000000000
usertrap(): unexpected scause 0x000000000000000f pid=7
            sepc=0x0000000000002208 stval=0x0000008000000000
usertrap(): unexpected scause 0x000000000000000f pid=8
            sepc=0x0000000000002208 stval=0x0000010000000000
usertrap(): unexpected scause 0x000000000000000f pid=9
            sepc=0x0000000000002208 stval=0x0000020000000000
usertrap(): unexpected scause 0x000000000000000f pid=10
            sepc=0x0000000000002208 stval=0x0000040000000000
usertrap(): unexpected scause 0x000000000000000f pid=11
            sepc=0x0000000000002208 stval=0x0000080000000000
usertrap(): unexpected scause 0x000000000000000f pid=12
            sepc=0x0000000000002208 stval=0x0000100000000000
usertrap(): unexpected scause 0x000000000000000f pid=13
            sepc=0x0000000000002208 stval=0x0000200000000000
usertrap(): unexpected scause 0x000000000000000f pid=14
            sepc=0x0000000000002208 stval=0x0000400000000000
usertrap(): unexpected scause 0x000000000000000f pid=15
```

【分析讨论】

通过本次实验，我对定时中断的处理有了更深入的理解。在实现定时中断处理函数的过程中，我涉及到了操作硬件定时器以及设置中断处理程序的技术细节。这不仅让我掌握了如何在操作系统中处理周期性事件，还加深了我对中断处理机制的整体了解，包括中断的触发、处理和恢复机制。定时中断的处理涉及到从硬件定时器获取中断信号，并在操作系统内核中实现相应的中断处理逻辑，这对于理解操作系统的实时性能和调度算法是至关重要的。

此外，这次实验还涉及到系统调用的设计与实现。通过设计和实现`sigalarm`和`sigreturn`系统调用，我学习到了如何在用户程序与内核之间进行数据传递和共享。这包括了如何在系统调用中处理用户态请求、如何在内核态中正确设置和恢复上下文，以及如何确保用户程序能够正确响应定时器事件。这一过程巩固了我对系统调用接口的理解，并提升了我在用户空间和内核空间之间进行有效交互的能力。

总的来说，本次实验不仅提高了我对定时中断处理机制的认识，也加强了我对系统调用设计和实现的理解。这些知识对于深入学习操作系统的内部机制和实现复杂的系统功能都是非常宝贵的经验。

【实验验证】

新建`time.txt`，输入自己做实验的用时，运行`make grade`进行评分：

```
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (2.8s)
== Test running alarmtest ==
$ make qemu-gdb
(3.3s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
Timeout! usertests: FAIL (300.2s)
    ...
        OK
        test opentest: OK
        test writetest: OK
        test writebig: panic: balloc: out of blocks
        qemu-system-riscv64: terminating on signal 15 from pid 129277 (make)
    MISSING '^ALL TESTS PASSED$'
    QEMU output saved to xv6.out.usertests
== Test time ==
time: OK
Score: 66/85
make: *** [Makefile:338: grade] Error 1
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab4$
```