PA 3

201300035 方盛俊

1. 阶段一: 穿越时空的旅程

1.1 设置异常入口地址

要想正确实现 `csrrw` 指令, 就需要先加入 `mtvec`, `mcause`, `mstatus`, `mepc` 这四个寄存器, 然后再使用这些寄存器实现第一条 CSR 指令.

1. 加入 CSR 寄存器
    1. 在 `isa-def.h` 加入 CSR 的四个寄存器 `mtvec`, `mcause`, `mstatus`, `mepc`
    2. 在 `reg.h` 中加入 `csr(idx)` 全局宏
    3. 在 `reg.c` 中修改寄存器有关的函数, 如 `isa_reg_display()`
2. 加入 `csrrw` 指令
    1. 在 `isa-all-instr.h` 之类的添加就不过多赘述
    2. 在 `decode.c` 里加入了 `def_DopHelper(csr)` 辅助函数
        1. 用于将 12 位的 csr 地址映射到只有 4 个的 csr 寄存器中 (暂时)
        2. 这步踩了很多坑...
    3. 在 `csr.h` 加入 `def_EHelper(csrrs)` 实现对应指令

做完这些之后, 终于能开始写自陷操作了...

1.2 触发自陷操作

1. 完成 `ecall` 指令
    1. `rtl_j(s, isa_raise_intr(11, cpu.pc + 4)); // 异常号 11, 代表 Environment call from M-mode`
    2. 注意不要与 `mret` 重合.
2. 完成 `csrrs` 指令
3. 完成 `mret` 指令, 返回到 `mepc` 寄存器所保存的地址

这里踩了很多坑, difftest 还出问题了...

1.3 重新组织结构体

通过观察 `trap.S` 的内容, 对 `Context` 重新整理如下:

```
struct Context {
  // fix the order of these members to match trap.S
  uintptr_t gpr[32], mcause, mstatus, mepc;
  void *pdir;
};
```

1.4 实现正确的事件分发

1. 在 `__am_irq_handle()` 加入了 `case -1: ev.event = EVENT_YIELD; break;`, 识别异常号 `-1`, 并打包为 `EVENT_YIELD` 事件.

2. 在 `do_event()` 加入了 `case EVENT_YIELD: printf("Event: Yield\n"); break;`, 识别出自陷事件 `EVENT_YIELD`, 然后输出 `Event: Yield`.

1.5 理解上下文结构体的前世今生 & 理解穿越时空的旅程

从 Nanos-lite 调用 `yield()` 开始, 到从 `yield()` 返回的期间, 这一趟旅程具体经历了什么?

前置工作: 初始化 `init_irq(void)`:

1. `init_irq(void)` 调用了 `cte_init(do_event)`, 将 `do_event()` 这个函数传入;
2. `cte_init()` 调用 `asm volatile("csrw mtvec, %0" : : "r"(__am_asm_trap))` 将 `__am_asm_trap()` 函数地址保存在 `mtvec` 中;
3. `cte_init()` 调用 `user_handler = handler` 将 `do_event` 保存在全局变量中, 以便后续回调;

正式工作: 调用 `yield()`:

1. `yield()` 调用了 `asm volatile("li a7, -1; ecall")`, 使用 `ecall` (设置异常号 `11`) 跳转到 `mtvec` 寄存器的 `__am_asm_trap()` 函数中;
2. `__am_asm_trap()` 使用 `addi sp, sp, -CONTEXT_SIZE` 在堆栈区初始化了 `CONTEXT_SIZE` 大小的上下文结构体 `c`, **这是上下文结构体生命周期的开端**;
3. `__am_asm_trap()` 使用 `MAP(REGS, PUSH)` 的宏展开式函数映射编程法, 类似于 `PUSH(REGS)` 这样, 将 `32` 个通用寄存器保存到了 `c` 中相应位置;
4. `__am_asm_trap()` 使用类似于 `csrr t0, mcause; STORE t0, OFFSET_CAUSE(sp)` 的汇编语句将 `mcause`, `mstatus` 和 `mepc` 三个寄存器保存到了 `c` 中相应位置;
5. `__am_asm_trap()` 将 `mstatus.MPRV` 置位, 以便通过 difftest;
6. `__am_asm_trap()` 使用 `mv a0, sp; jal __am_irq_handle` 将位于堆栈区的 `c` 上下文结构体保存到函数传参寄存器 `a0` 中, 作为函数参数调用并传给 `__am_irq_handle()` 函数;
7. `__am_irq_handle()` 通过 `c->mcause` 判别异常号, 并创建对应 `Event ev`, 调用 `user_handler(ev, c)`, 即调用上文提到的 `do_event(e, c)`;
8. `do_event()` 对异常或中断做完相应处理后, 返回到 `__am_irq_handle()` 中;
9. `__am_irq_handle()` 也做完了相应处理, 返回到 `__am_asm_trap()` 中;
10. `__am_asm_trap()` 使用类似于 `LOAD t1, OFFSET_STATUS(sp); csrw mstatus, t1` 的汇编语句将 `c` 中相应位置保存到 `mstatus` 和 `mepc` 两个寄存器中;
11. `__am_asm_trap()` 使用 `MAP(REGS, POP)` 将 `c` 中相应位置数据复原回 `32` 个通用寄存器中;
12. `__am_asm_trap()` 使用 `addi sp, sp, CONTEXT_SIZE` 将堆栈区复原, 相当于将 `c` 释放, **这是上下文结构体生命周期的结束**;
13. `__am_asm_trap()` 使用 `mret`, 将 `mtvec` 寄存器内保存的数据取出, 并跳转到该位置, 即回到了调用中断代码的 `yield()` 函数中;
14. `yield()` 处理完所有事情, 便返回了, 进而调用了 `panic("Should not reach here")`.

此外, 我还在 `trap.S` 的 `csrw mepc, t2` 指令前加入了 `addi t2, t2, 4`, 来实现自陷指令 `ecall` PC 加 4 的效果.

1.6 异常处理的踪迹 - etrace

修改 Kconfig, 并在 `intr.c` 的 `isa_raise_intr(word_t NO, vaddr_t epc)` 中加入

```
#ifdef CONFIG_ETRACE
log_write("[etrace] mcause: %d, mstatus: %x, mepc: %x\n", cpu.csr[1]._32, cpu.csr[2]._32, cpu.csr[
#endif
```

即可.

2. 阶段二: 穿越时空的旅程

2.1 简化操作: 自动化脚本

我们需要在 `riscv32-nemu` 和 `native` 之间不断切换, 要做如下三件事情:

1. 使用 `ISA=xxx` 编译 dummy
2. 把编译出的 dummy ELF 文件作为 nanos-lite 的 ramdisk, 复制过去
3. 使用 `ARCH=xxx` 编译并运行 nanos-lite

为了简化操作, 写了一个脚本, 脚本如下所示:

```
#!/bin/bash
program=dummy

function init() {

    make -C ../navy-apps/tests/dummy ISA=$1
    cp ../navy-apps/tests/dummy/build/dummy-$1 ./build/ramdisk.img
    make ARCH=$2 run
}

case $1 in
nemu)
    init riscv32 riscv32-nemu
    ;;
native)
    init am_native native
    ;;
*)
    echo "Invalid input..."
    exit
    ;;
esac
```

2.2 堆和栈在哪里?

> **Question:** 我们提到了代码和数据都在可执行文件里面, 但却没有提到堆 (heap) 和栈 (stack). 为什么堆和栈的内容没有放入可执行文件里面? 那程序运行时刻用到的堆和栈又是怎么来的? AM 的代码是否能给你带来一些启发?
>
> **Answer:** 堆和栈是进程才有的概念, 程序只是一个静态的可执行文件, 包含着一个进程运行所需的代码信息, 本身并不是运行着的. 只有程序被加载为进程, 才会出现堆和栈.

2.3 如何识别不同格式的可执行文件?

> **Question:** 如果你在GNU/Linux下执行一个从Windows拷过来的可执行文件, 将会报告"格式错误". 思考一下, GNU/Linux是如何知道"格式错误"的?
>
> **Answer:** 通过文件头的 "魔数", 对于 ELF 文件来说, 这个魔数为
>
> `Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 .`

2.4 冗余的属性?

> **Question:** 使用 readelf 查看一个 ELF 文件的信息, 你会看到一个 segment 包含两个大小的属性, 分别是 FileSiz 和 MemSiz, 这是为什么? 再仔细观察一下, 你会发现 FileSiz 通常不会大于相应的 MemSiz, 这又是为什么?
>
> **Answer:** 因为程序中 `.bss` 节对应的是未初始化的全局变量, 在程序中也就不需要占用空间, 即长度为 0; 但是对于进程来说, `.bss` 节加载到内存中, 仍然是需要占用空间的, 此时长度就不为 0, 因此 `MemSiz` 总长度也会大于 `FileSiz`.

2.5 实现 loader

我实现的 `loader` 如下:

```
static uintptr_t loader(PCB *pcb, const char *filename) {

    Elf_Ehdr elf = {};
    ramdisk_read(&elf, 0, sizeof(Elf_Ehdr));

    // Make sure that the file is an elf file
    assert(*(uint32_t *)elf.e_ident == ELF_MAGIC);

    Elf_Phdr ph = {};
    for (int i = 0; i < elf.e_phnum; ++i) {
        ramdisk_read(&ph, elf.e_phoff + i * sizeof(Elf_Phdr), sizeof(Elf_Phdr));
        if (ph.p_type == PT_LOAD) {
            // Copy to [VirtAddr, VirtAddr + FileSiz)
            memcpy((void *)ph.p_vaddr, &ramdisk_start + ph.p_offset, ph.p_filesz);
            // Set [VirtAddr + FileSiz, VirtAddr + MenSiz) with zero
            memset((void *)(ph.p_vaddr + ph.p_filesz), 0, ph.p_memsz - ph.p_filesz);
        }
    }

    return elf.e_entry;
}
```

此外, 我还修改了 `cte.c` 中 `event` 分发的方式, 这里我们认为 `a7 == -1` 时是 `EVENT_YIELD`, 否则是 `EVENT_SYSCALL`

```
switch (c->mcause) {
  case 11: {
    if (c->GPR1 == -1) {
      ev.event = EVENT_YIELD;
    } else {
      ev.event = EVENT_SYSCALL;
    }
    break;
  }
  default: ev.event = EVENT_ERROR; break;
}
```

2.6 识别系统调用

目前 dummy 已经通过 `_syscall_()` 直接触发系统调用, 我需要让 Nanos-lite 识别出系统调用事件 EVENT_SYSCALL.

于是我在 `irq.c` 中将其修改为

```
switch (e.event) {
 case EVENT_YIELD: printf("EVENT_YIELD, GPR1: %d\n", c->GPR1); break;
 case EVENT_SYSCALL: {
     do_syscall(c);
     break;
 }
 default: panic("Unhandled event ID = %d, GPR1: %d\n", e.event, c->GPR1);
}
```

2.7 实现 SYS_yield 系统调用

我在补充完整 `syscall.h` 的基础上, 在 `syscall.c` 中加入了:

```
void do_syscall(Context *c) {
  uintptr_t a[4];
  a[0] = c->GPR1;
  a[1] = c->GPR2;
  a[2] = c->GPR3;
  a[3] = c->GPRx;

  switch (a[0]) {
    case SYS_yield: {
        yield();
        c->GPRx = 0;
        break;
    }
    default: panic("Unhandled syscall ID = %d", a[0]);
  }
}
```

并且 `riscv32-nemu.h` 要改为:

```
#define GPR1 gpr[17] // a7
#define GPR2 gpr[10] // a0
#define GPR3 gpr[11] // a1
#define GPR4 gpr[12] // a2
#define GPRx gpr[10] // a0
```

2.8 实现 SYS_exit 系统调用

在 `syscall.c` 中加入

```
case SYS_exit: halt(0); break;
```

即可.

2.9 系统调用的痕迹 - strace

```
#ifdef ENABLE_STRACE
 printf("[strace] %s(%d, %d, %d) = %d\n", syscall_names[a[0]], a[1], a[2], a[3], c->GPRx);
#endif

 switch (a[0]) {
     case SYS_exit:
         printf("[strace] SYS_exit(%d)\n", a[1]);
         halt(0);
```

```
        break;
}
```

其中开关位于 `common.h`.

具体输出大致如下:

```
[strace] SYS_yield(0, 0, 0) = 0
EVENT_YIELD, GPR1: -1
[strace] SYS_exit(0, 0, 0) = 0
[strace] SYS_exit(0)
```

2.10 标准输出

```
case SYS_write: {
   // int _write(int fd, void *buf, size_t count)
   if (a[1] == 1 || a[1] == 2) {
       for (size_t i = 0; i < a[3]; ++i) putch(((char *) a[2])[i]);
       c->GPRx = a[3];
   }
   break;
}
```

2.11 堆区管理

在 Navy 的 `syscall.c` 中加入

```
extern char end;
void *program_break = &end;

void *_sbrk(intptr_t increment) {
  if (increment == 0) {
      return program_break;
  }
  if (_syscall_(SYS_brk, increment, 0, 0)) {
    return -1;
  } else {
    program_break += increment;
    return program_break - increment;
  }
}
```

2.12 hello 程序是什么, 它从而何来, 要到哪里去

> 我们知道 `navy-apps/tests/hello/hello.c` 只是一个 C 源文件, 它会被编译链接成一个ELF文件. 那么, hello程序一开始在哪里? 它是怎么出现内存中的? 为什么会出现在目前的内存位置? 它的第一条指令在哪里? 究竟是怎么执行到它的第一条指令的? hello 程序在不断地打印字符串, 每一个字符又是经历了什么才会最终出现在终端上?

1. 在 `navy-apps/tests/hello` 目录下经过 `make ISA=riscv32` 编译之后, 在 `build` 目录下生成了 `hello-riscv32` 文件, 这是一个 ELF 格式的可执行文件.
2. 我们将 `hello-riscv32` 文件复制到 `nanos-lite/build/ramdisk.img` 文件, 作为给 Nanos 使用的 "内存虚拟盘" `ramdisk` 加载.
3. Nanos 的 `resources.S` 内部通过 `ramdisk_start:; .incbin "build/ramdisk.img"; ramdisk_end:` 语法将 `ramdisk.img` 文件加载进内存里.

4. Nanos 在 `ramdisk.c` 文件中使用 `&ramdisk_start` 来获取已经加载进内存的 `ramdisk.img` 对应的内存地址.
5. `hello-riscv32` 即 `ramdisk` 被我们自己编写的 `loader()` 函数识别为一个 ELF 文件, 并按照约定加载到地址 `0x83000000` 中.
6. 第一条指令的地址通过 ELF 文件中 `elf.e_entry` 给出.
7. 为了调用它的第一条指令, 我们只需要将 `elf.e_entry` 作为一个函数地址进行调用即可, 就像 `((void (*)())entry)()` 这样调用.
8. hello 程序打印字符串的经历如下:
    1. `hello.c` 调用 `printf("Hello World from Navy-apps for the %dth time!\n", i ++);`
    2. libc 库中的 `printf()` 会将字符暂时放置到 `wbuf.c` 的缓冲区中, 当达到一定条件一会调用一次 `_write()` 函数进行输出.
    3. `_write()` 函数会触发中断 `_syscall_(SYS_write, fd, buf, count)`, 后者调用了 `ecall` 指令.
    4. `ecall` 指令实际上跳转到了 `__am_asm_trap()` 函数, 封装好 `Context c` 后进一步调用 `__am_irq_handle(c)` 函数, 其封装为事件 `EVENT_SYSCALL` 后调用 `do_event(Event e, Context* c)`
    5. `do_event(Event e, Context* c)` 确认是事件 `EVENT_SYSCALL` 后, 调用 `do_syscall(Context *c)`, 其中 `c->GPR1` 存储了系统调用号, `c->GPR2~4` 是三个参数, `c->GPRx` 用来存放返回值.
    6. 对于 `SYS_write` 系统调用, 我们通过 `for (size_t i = 0; i < a[3]; ++i) putch(((char *) a[2])[i]);` 输出每一个字符.
    7. AM 的 `putch()` 又调用了 NEMU 里的串口设备, 进行输出.
    8. 最后由 NEMU 把字符输出到控制台.