Simple RISC-V OS

Porterlu
SUSTech

2022 年 10 月 2 日

目录

1 启动 Simple RISC-V OS

1.1 Makefile

```
include ../common.mk

SYSCALL = y
ifeq (${SYSCALL}, y)
CFLAGS += -D CONFIG_SYSCALL
endif

SRCS_ASM = \
        start.S \
        mem.S \
        entry.S \
        usys.S

SRCS_C = \
        kernel.c \
        uart.c \
        printf.c \
        page.c \
        sched.c \
        user.c \
        trap.c \
        plic.c \
        timer.c \
        lock.c \
        syscall.c

OBJS = ${SRCS_ASM: .S=.o}
OBJS += ${SRCS_C: .c=.o}

.DEFAULT_GOAL := all
all:os.elf

os.elf:${OBJS}
        ${CC} ${CFLAGS} -T os.ld -o os.elf $^
        ${OBJCOPY} -O binary os.elf os.bin

%.o : %.c
        ${CC} ${CFLAGS} -c -o $@ $<

%.o : %.S
        ${CC} ${CFLAGS} -c -o $@ $<

run: all
        @${QEMU} -M ? || grep virt >/dev/null || exit
        @echo "Press Ctrl-A and then X to exit QEMU"
        @echo "------------------------------------"
        @${QEMU} ${QFLAGS} -kernel os.elf

.PHONY: debug
debug: all
        @echo "Press Ctrl-C and then input 'quit' to
                exit GDB and QEMU"
        @echo "-------------------------------------
                ----------------"
        @${QEMU} ${QFLAGS} -kernel os.elf -s -S &
        $${GDB} os.elf -q -x ../gdbinit

.PHONY: code
code: all
        @${OBJDUMP} -S os.elf | less

.PHNOY: clean
clean:
        rm -rf *.o *.bin *.elf
```

如上的 **Makefile** 用于 **OS** 的编译、运行、清理 ***Simple RISCV OS***, 其中 ***common.mk*** 定义了所使用的编译器、模拟器、调试器及其参数。

接下来定义了是否使用 **SYSCALL**, 用于条件编译, ***SRCS_ASM*** 和 ***SRCS_C*** 定义所使用的源文件，这些文件将用于编译生成我们的目标代码。所以目标文件 ***OBJS*** 就可以替换为同名的.*o* 文件

接下来就是依赖规则：

%.o : %.S 所有的.*S* 文件都可以生成目标文件。

%.o : %.c 所有的.*c* 文件都可以生成目标文件。

os.elf : $OBJS 根据目标文件和链接脚本 *os.ld* 链接生成可执行文件

run, ***debug***, ***code*** 分别是对 ***qemu***, ***gdb***, ***objdump*** 的封装, ***qemu*** 使用的是 ***virt*** 平台，在系统模式下进行执行,***gdb*** 用于调试 qemu，最后 ***objdump*** 用于查询反汇编代码。

1.2 链接文件

链接脚本定义了我们的内存布局，在 ***Simple RISC-V OS*** 中，它的内容定义在 ***os.ld*** 中：

```
OUTPUT_ARCH( "riscv" )

ENTRY( _start_ )

MEMORY
{
        ram(wxa!ri): ORIGIN = 0x80000000,
                     LENGTH = 128M
}

SECTIONS
{
        .text : {
                PROVIDE(_text_start = .);
                *(.text .text.*)
                PROVIDE(_text_end = .);
        } >ram

        .rodata : {
                PROVIDE(_rodata_start = .);
                *(.rodata .rodata.*)
                PROVIDE{_rodata_end = .};
        } >ram

        .data : {
                . = ALIGN(4096);
                PROVIDE(_data_start = .);
                *(.sdata .sdata.*)
                *(.data .data.*)
                PROVIDE(_data_end = .);
        } >ram

        .bss : {
                PROVIDE(_bss_start = .);
                *(.sbss .sbss.*)
                *(.bss .bss.*)
                *(COMMON)
                PROVIDE(_bss_end = .);
        }

        PROVIDE(_memory_start = ORIGIN(ram));
        PROVIDE(_memory_end = ORIGIN(ram) +
                LENGTH(ram));

        PROVIDE(_heap_start = _bss_end);
        PROVIDE(_heap_size = _memory_end -
                _heap_start);
}
```

链接脚本用于定义了 **128MB** 的物理地址空间，在这个物理地址空间内分布了代码段、只读数据段、数据段、静态数据段，然后是堆区的起始地址。

在 ***mem.S*** 中根据链接脚本中定义的了一系列的全局变量，并且将这些变量都放在了***.rodata*** 中，如：***HEAP_START*** 和 ***HEAP_SIZE***。

1.3 start.S

在 ***start.S*** 中做了基础的初始化，就跳转至内核：

```
#include "platform.h"

    .equ STACK_SIZE, 1024

    .global _start

    .text
```

```
_start:
    csrr   t0, mhartid
    mv     tp, t0
    bnez   t0, park

    la     a0, _bss_start
    la     a0, _bss_end
    bgeu   a0, a1, 2f
1:
    sd     zero, (a0)
    add    a0, a0, 8
    bltu   a0, a1, 1b
2:
    slli   t0, t0, 10
    la     sp, stacks + STACK_SIZE

    add    sp, sp, t0

#ifdef CONFIG_SYSCALL
    li     t0, 0xffffffff
    csrw   pmpaddr0, t0
    li     t0, 0xf
    csrw   pmpcfg0, t0
#endif

#ifdef CONFIG_SYSCALL
    li     t0, 1 << 7
#else
    li     t0, 3 << 11 | 1 << 7
#endif

    csrr   a1, mstatus
    or     t0, t0, a1
    csrw   mstatus, t0

    j start_kernel

park:
    j park

.align 3
stacks:
    .skip STACK_SIZE * MAXNUM_CPU
    .end
```

跳转至内核前有已下的几步：

1. 如果 hartid 不为 0, 则进行空转。
2. 将静态数据区域内的初始值设为 0。
3. 设置栈空间，所有 hart 的栈都在 stacks 之后，每个 hart 有 1024Bytes。
4. mstatus 需要初始化为需要的值，如果使用 syscall 我们除了开启全区中断还要设置 mpp 在 USER MODE。
5. 跳转至内核代码

2 内核初始化

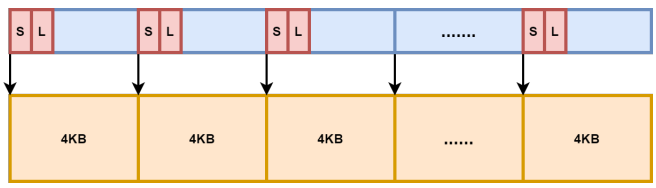
2.1 串口初始化

```
#define UART_REG(reg) ((volatile uint8_t *) \
        (UART0 + reg))
#define RHR 0   //接受状态寄存器(read mode)
#define THR 0   //传输状态寄存器(write mode)
#define DLL 0   //低位分频寄存器(write mode)
#define IER 1   //中断使能寄存器(write mode)
#define DLM 1   //高位分频寄存器(write mode)
#define FCR 2   //先进先出控制器(write mode)
#define ISR 2   //中断状态寄存器(read mode)
#define LCR 3   //行状态控制寄存器(read mode)
#define MCR 4   //调制控制寄存器
#define LSR 5   //行状态寄存器
#define MSR 6   //调制状态寄存器
#define SPR 7   //暂存寄存器

#define LSR_RX_READY (1 << 0)
#define LSR_TX_IDLE (1 << 5)

#define uart_read_reg(reg) (*(UART_REG(reg)))
#define uart_write_reg(reg, v) (*(UART_REG(reg)) = \
                                (v))
```

```
void uart_init(){
        //禁用中断
        uart_write_reg(IER, 0x00);

        //设置波特率
        uint8_t lcr = uart_read_reg(LCR);
        uart_write_reg(LCR, lcr | (1 << 7));
        uart_write_reg(DLL, 0x03);
        uart_write_reg(DLM, 0x00);

        //设置奇偶校验位，停止位，数据长度
        lcr = 0;
        uart_write_reg(LCR, lcr | (3 << 0))
}

int putc(char ch){
        while(uart_read_reg(LSR) & LSR_TX_IDLE == 0);
        return uart_write_reg(THR, ch);
}

void uart_puts(char *s){
        while(*s){
                uart_putc(*s);
                s++;
        }
}
```

重复的寄存器意味复用，在读写或者不同的顺序下作为不同的寄存器使用。我们这里因为不使用 ***uart*** 输入，所以关闭了中断。输出时，我们这里使用轮询模式，查询 **LSR** 中是否空闲，之后想传输寄存器中输出一个字节即可。

2.2 堆区初始化



这里使用数组的形式来管理以页为粒度的堆区内存，这里使用每一个页都有一个 **Byte** 进行管理我们这里只使用了其中两位，分别用于判断改 **Byte** 对应的页是否已经被分配和判断该页是否是这一组被分配的页中的最后一位。

在分配若干个页时，首先找到一个空闲的页，之后判断这个页后是否有满足数量要求的若干个连续的页，存在则在对应的标志上置位，同时标志最后一个位分配的最后一个页，最后返回分配的起始地址。

而对于释放申请的空间，只需将指针指向需要处理的堆区管理 **Byte**，将直到最后一个页的管理 **Byte** 的有效位置 **0** 即可。

2.3 Trap 初始化

陷入的地址即 ***trap_vector***，在这里中断和异常都将做执行具体服务前的处理。

```
csrrw   t6, mscratch, t6
reg_save  t6

mv      t5, t6
csrr    t6, mscratch
sw      t6, 120(t5)

csrr    a0, mepc
sw      a0, 124(t5)

csrw    mscratch, t5

csrr    a0, mepc
csrr    a1, mcause
csrr    a2, mscratch
call    trap_handler

csrw    mepc, a0

csrr    t6, mscratch
reg_restore t6
mret
```

陷入 ***trap_vector*** 首先要保存上下文，在 ***Simple RISC-V OS*** 中上下指的是除了 **0** 号寄存器外的 **31** 个通用寄存器还 ***mepc*** 寄存器。保存信息的过程分为两步：

1. 使用 *CSR* 指令交换 *t6* 和 *mscratch* 中的值，这时就可通 *t6* 寄存器将除了 t6 寄存器外的所有值存入 *mscratch* 所指向的上下文中。
2. 在将 *mscratch* 的值从 *t6* 存储到 *t5*，这时就可以将 *t6* 中原本的值返回到寄存器中，并存储到上下文中。

这时就可以正式调用处理函数，调用处理函数时要传入三个参数，分别是 ***mepc,mcause,mcause***，最后调用 ***handler***。返回时，就可根据返回值修改 ***mepc***，根据上下文恢复寄存器中的值，最后使用 ***mret*** 返回。

3 格式化输出

3.1 printf

格式化输出中 ***printf*** 调用了 ***__vprintf***, ***__vprintf*** 调用了 ***__vsnprintf***, 在 ***__vsnprintf*** 做真正的格式处理，支持的格式有：*"%d"*、*%p*、*%x*、*%s*、*%c*, 输入的参数为输出字符、个数、格式化字符、参数列表。格式化的过程将遍历整个输入的格式字符串：

```
static int _vsnprintf(char *out, size_t n,
                const char *s, va_list vl)
{
        int format = 0;
        int longarg = 0;
        size_t pos = 0;
        for(;*s;s++){
            if(format){
                判断当前字符
                如果是 'l':
                        设置longarg为1
                如果是 'p':
                        输出'0x'
                        之后根据'x'的情况进行输出
                如果是 'x':
                        从va_list中取出一个long,
                        将这个数翻译为16进制进行输出
                如果是 'd':
                        从va_list取出一个long,
                        将这个数翻译为10进制进行输出
                如果是 's':
                        从va_list取出一个long,
                        取出这个指针，输入直至'\0'
                如果是 'c':
                        取出一个long，根据ascii码进
                        行输出
            }else if(*s == '%'){
                format = 1;
            }else{
                if(out && pos < n){
                        out[pos] = *s;
                }
                pos++;
            }
        }
}
```

这里由于运行 **RV64** 所以都是 **64** 位的参数，同时有一个就是*%p* 的输出方法，它采取和*%x* 的方法进行输出。

4 中断

4.1 时间中断

4.1.1 硬件时钟

硬件上，***RISC-V*** 上有 ***Clint*** 单元，每过一段固定的时间，一旦 ***mtime*** 中的值大于等于 ***mtimecmp***，硬件会自动触发一个时钟中断，这时向 ***mtimecmp*** 中重新写入一个值就可以清理这次时钟，为下一次的时钟中断到来做准备。

```
void timer_load(int interval){
    int id = r_mhartid();
    *(uint64_t *)CLINT_MTIMECMP(id) = *(uint64_t *)
                                CLINT_MTIME + interval;
```

```
}

void timer_init(){
        timer_load(TIMER_INTERVAL);
        w_mie(r_mie() | MIE_MTIE);
}

void timer_handler(){
        _tick ++;
        printf("tick: %d\n", _tick);
        timer_check();
        timer_load(TIMER_INTERVAL);
        schedule();
}
```

初始化过程中首先为 ***MTIMECMP*** 载入一个初始值，之后就可以打开全局中断的中时间中断。在时钟中断的 **handler** 函数中，将全局维护的 ***__tick*** 加 **1**，同时加载下一次触发时钟中断的时间点，最后由于 ***Simple RISC-V OS*** 是分时操作系统，所以时钟中断到来时，会进行线程调度。

4.1.2 软件时钟与锁

软件时钟是基于硬件时钟用软件实现的，所以它有更多的灵活性但是精度也会比较差，在 ***Simple RISC-V OS*** 过多的软件时钟会非常影响性能。

锁是在进入关键区之前，用于维护并发执行正确性的一种手段，由于并没使用原子指令，所以这里使用开关中断进行实现。

```
struct timer * timer_create(
                void (*handler)(void *arg),
                void *arg,
                uint32_t timeout){

        if(NULL == handler || 0 == timeout){
                return NULL;
        }

        spin_lock();

        struct timer *t = &(timer_list[0]);
        for(int i = 0; i < MAX_TIMER; i++){
                if(NULL == t->func)
                        break;
                t++;
        }

        if(t == &timer_list[0] + 10){
                spin_unlock();
                return NULL;
        }

        t->func = handler;
        t->arg = arg;
        t->timeout = _tick + timeout;

        spin_unlock();
        return t;
}

void timer_delete(struct timer* timer){
        spin_lock();

        struct timer *t = &(timer_list[0]);
        for(int i = 0; i < MAX_TIMER; i++){
                if(t == timer){
                        t->func = NULL;
                        t->arg = NULL;
                        break;
                }
                t++;
        }

        spin_unlock();
}
```

这里由于创建软件时钟访问的是一个公共的数据结构，如果并发执行可能存在错误，在这里进入公共的数据

结构 ***timer_list*** 前关闭时钟中断，相当于加上了一层锁。

注册软件时钟的过程是一个填充 ***timer_list*** 的过程，在软件时钟的结构体中输入触发时钟的全局时间，还有要执行的函数和函数的参数，那么由于每次在触发硬件时钟都要进行 ***timer_check()***。这个函数会遍历所有软件时钟，依次检查，由于执行软件时钟的处理是在中断上下文中，所以这里在没打开全局中断的情况下是不会进行中断嵌套的。

4.2 软件中断

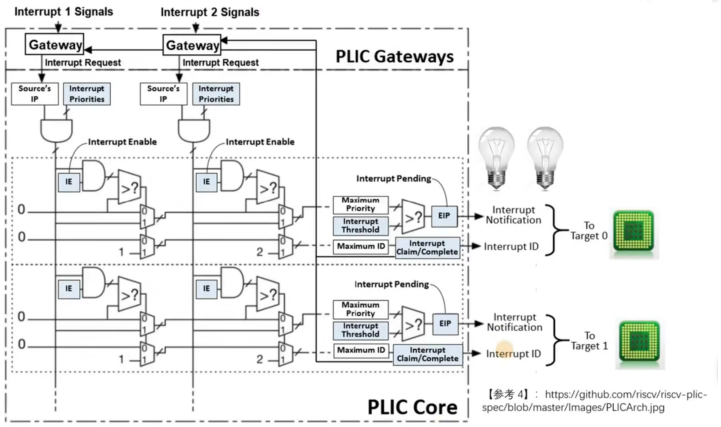
在 ***Clint*** 中有一个 ***MSIP*** 用控制软件中断，向其写一个值，那么对应的 ***hart*** 就会触发一个软件中断，之后进入 **handler** 函数根据软件中断进行处理。

```
if(cause & 0x8000000000000000LL){
        switch(cause_code){
                case 3:
                uart_puts("software interruption\n");
                int id = r_mhartid();
                *(uint64*) CLINT_MSIP(id) = 0;
                break;
        }       ...
}
```

4.3 外部中断

本仓库中的代码并没有实现外部中断，但是 **PLCT** 实现室的源码中是有实现外部中断的。

```
#include "os.h"

void plic_init(void){
        int hart = r_tp();

        //设置UART中断优先级
        *(uint64_t *) PLIC_PRIORITY(UART0_IRQ) = 1;

        //使能UART中断
        *(uint64_t *) PLIC_MENABLE(hart) =
                (1 << UART0_IRQ);

        //设置中断阈值，所有优先级小于或者等于阈值的中
        *(uint64_t *) PLIC_MTHRESHOLD(hart) = 0;

        //打败hart的外部中断
        w_mie(r_mie() | MIE_MEIE);
}

int plic_claim(void){
        int hart = r_tp();
        //将获取外部中断号
        int irq = *(uint64_t *)PLIC_MCLAIM(hart);
        return irq;
}

void plic_complete(int irq){
        int hart = r_tp();
        //通知plic这个中断已经被处理完毕
        *(uint64_t *)PLIC_MCOMPLETE(hart) = irq;
}
```



【参考 4】：https://github.com/riscv/riscv-plic-spec/blob/master/Images/PLICArch.jpg

如下是 **uart** 的中断处理函数：

```
int uart_getc(void){
        if(uart_read_reg(LSR) & LSR_RX_READY){
                return uart_read_reg(RHR);
        }else{
                return -1;
```

```
        }
}
void uart_isr(void){
        whlie(1){
                int c = uart_getc();
                if(c == -1){
                        break;
                }else{
                        uart_putc((char) c);
                        uart_putc('\n');
                }
        }
}
```

uart_get() 将检查中断到来的信号，并获取 **uart** 中的值，最后我们将获取到的值进行输出。

5 线程切换

5.1 协作式线程切换

协作式线程切换，即线程自己主动放弃 **CPU**，将执行的权利让给其他线程，各个线程通过这种方式协作并发地运行，下面将介绍如何进行上下文的切换，上下文的保存和恢复其实在 *trap_vector* 中已经进行了介绍。

线程调用 *yield* 函数，将自己的上下文保存到一个结构体中，同时在上下文队列中按顺序取出下一个线程的上下文之后用新的上下文将寄存器恢复，之后一个 *ret* 或者 *mret* 指令就可以将执行让给下一个线程。这是一种上古时期的线程协作方式，可能存在特殊利用的可能，但是这里不做考虑。

5.2 抢占式线程切换

抢占式，意味着内核有权力夺取线程的执行，这里是通过时钟中断实现的，一旦一个时钟中断到来，内核中会对线程进行依次调度。

```
.global switch_to
.align 4
switch _to:
        csrw    mscratch, a0
        ld      a1, 248(a0)
        csrw    mepc, a1

        mv      t6, a0
        reg_restore t6

        mret
```

这个函数将 **a0** 所指向的上下文恢复到寄存器中，于是 *mret* 后 *mepc* 中的值将恢复到 *pc* 中，于是执行流便发生了切换。线程队列也是一个公共数据结构，但是由于中断上下文这里是关中断的，所以不用进行上锁。

6 系统调用

系统调用就涉及到权限的概念，这里我们要将线程的运行权限降到用户权限：

```
li      t0, 1 << 7
csrr    a1, mstatus
or      t0, t0, a0
csrw    mstatus, t0
```

前面这一段 *start.S* 中在开启系统调用的条件编译选项后将只设置 **mstatus** 中的中断使能位，但是 *mpp* 位仍然为 **0**, 所以第一次 *mret* 后将进入用户模式。这时如果还想要执行一些高权限的操作必须通过系统调用，我们这里的例子是获取 *hartid*。

```
if(中断){
        ......
}else{
        switch(cause_code){
                case 8:
                        do_syscall(cxt);
                        return_pc += 4;
                        break;
        }
}
```

```
void do_syscall(struct context *cxt){
        uint64_t syscall_num = cxt->a7;
        switch(syscall_num){
                case SYS_gethid:
                        cxt->a0 = sys_gethid(
                        (uint64_t*)(cxt->a0));
                        break;
                default:
                        ...
        }
}

int sys_gethid(uint64_t *ptr_hid){
        if(ptr_hid == NULL){
                return -1;
        }else{
                *ptr_hid = r_mhartid();
                return 0;
        }
}
```

这一个过程中必须通过 $a7$ 寄存器传递系统调用号，因为系统调用会将上下文作为参数传入到系统调用的处理函数，所以这时就可以分辨具体是哪个系统调用。同时 $a0$ 仍旧是函数的参数，$sys\_\_gethid$ 会把它作为参数参数，该函数默认将它作为一个长整型的指针，将 $hartid$ 放入指针所执行的区域，最后 $a0$ 中放入返回值，这时指针就已经不存在于 $a0$ 中了，通过返回值可以得知是否执行成功，同时 $SYS\_\_gethid$ 是将返回的地址设置为当前 $mepc+4$ 异常返回后将执行下一条指令，最后在用户只用调用封装号 $ecall$ 的库函数即可实现 $hartid$ 的获取。