实现U模式的trap机制 | TimerのBlog

yanglianoo.github.io/2023/08/04/实现U模式的trap机制

2023年8月4日

> 好久没更新了，最近在摆烂，天天打瓦洛兰特……瓦洛兰特是真好玩啊hhhhh

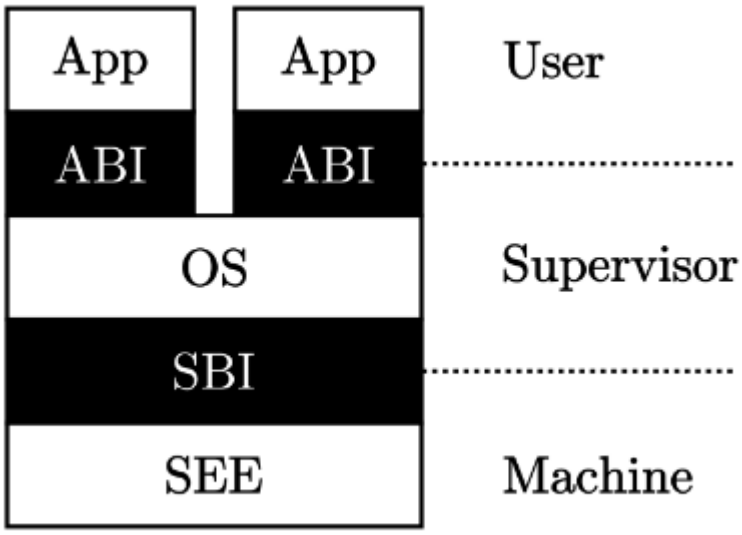
1. 测试用户态的syscall

1.1 riscv特权级切换

RISC-V 架构中一共定义了 4 种特权级：

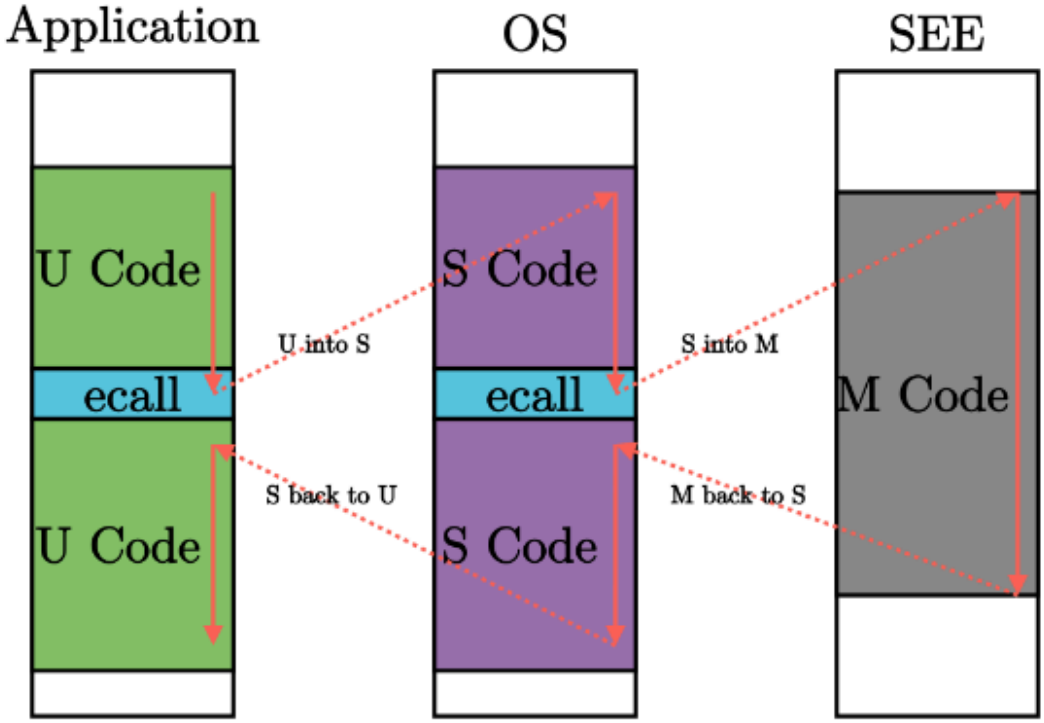
级别	编码	名称
0	00	用户/应用模式 (U, User/Application)
1	01	监督模式 (S, Supervisor)
2	10	虚拟监督模式 (H, Hypervisor)
3	11	机器模式 (M, Machine)

其中级别数值越大，特权级越高，对硬件的控制能力越强。之前移植的Opensbi运行在M模式下，S模式下的程序通过ecall指令去调用Opensbi提供的服务，U模式下的程序同样也可通过ecall指令来获取S模式下提供的服务。



我们编写的os程序就是运行在S态的，向用户态提供的接口标准被称为ABI。用户态应用直接触发从用户态到内核态的异常的原因总体上可以分为两种：其一是用户态软件为获得内核态操作系统的服务功能而执行特殊指令；其二是在执行某条指令期间产生了错误（如执

行了用户态不允许执行的指令或者其他错误）并被 CPU 检测到。特权切换的机制如下图：



1.2 riscv 系统调用简介

我们知道我们在linux系统下编写的应用程序是去调用C库的函数去实现对应的功能，而C库呢会去使用内核提供的一组接口去访问硬件设备和操作系统资源，这组接口就被称为系统调用。

在X86平台上，Linux在用int 0x80进行系统调用时，调用号存在于EAX中，第一个参数存在于EBX，第二个参数存在于ECX，第三个参数存在于EDX。而在riscv平台下，系统调用是通过ecall指令来触发的，ecall 指令规范中没有其他的参数，Syscall 的调用参数和返回值传递通过遵循如下约定实现：

- 调用参数
  - a7 寄存器存放系统调用号，区分是哪个 Syscall
  - a0-a5 寄存器依次用来表示 Syscall 编程接口中定义的参数
- 返回值
  a0 寄存器存放 Syscall 的返回值

ecall 指令会根据当前所处模式触发不同的执行环境切换异常：

- in U-mode: environment-call-from-U-mode exception
- in S-mode: environment-call-from-S-mode exception
- in M-mode: environment-call-from-M-mode exception

Syscall 场景下是在 U-mode（用户模式）下执行 ecall 指令，主要会触发如下变更：

- 处理器特权级别由 User-mode（用户模式）提升为 Supervisor-mode（内核模式）
- 当前指令地址保存到 sepc 特权寄存器
- 设置 scause 特权寄存器
- 跳转到 stvec 特权寄存器指向的指令地址

1.3 riscv 系统调用测试

首先我在test文件夹下新建了一个syscall目录，里面新建了三个文件：

```
timer@DESKTOP-JI9EVEH:~/quard-star/test/syscall$ ls
Makefile  syscall.c  test_write.c
```

首先看Makefile:

makefile

```
CC=riscv64-unknown-elf-
gcc

write:test_write.c
syscall.c
	$(CC) $^  -o
write.out
```

很简单就是编译两个源文件，生成write.out

然后是syscall.c

c

```
#include "stddef.h"
#include "stdint.h"
#include "stdio.h"
size_t syscall(size_t id, uintptr_t arg1, uintptr_t arg2, uintptr_t
arg3) {
    long ret;
    asm volatile (
        "mv a7, %1\n\t"   // Move syscall id to a0 register
        "mv a0, %2\n\t"   // Move args[0] to a1 register
        "mv a1, %3\n\t"   // Move args[1] to a2 register
        "mv a2, %4\n\t"   // Move args[2] to a3 register
        "ecall\n\t"       // Perform syscall
        "mv %0, a0"       // Move return value to 'ret' variable
        : "=r" (ret)
        : "r" (id), "r" (arg1), "r" (arg2), "r" (arg3)
        : "a7", "a0", "a1", "a2", "memory"
    );
    return ret;
}
```

可以看见在这里定义了一个syscall函数，传入的参数为系统调用号以及三个参数，通过内联汇编的形式将系统调用号写入了a7寄存器，然后将传入的三个参数分别写入了a0,a1,a2寄存器，然后调用ecall指令进入内核的异常处理程序。再调用完成后内核会将返回值放在a0寄存器中。

> 在这段代码中，”memory” 是一种内联汇编（inline assembly）中的约束（constraint）。内联汇编是一种在C或C++代码中嵌入汇编指令的技术，它允许直接在高级语言中嵌入底层的汇编代码。
>
> 在这里，”memory” 约束告诉编译器该内联汇编代码可能会读取或修改内存中的数据，因此编译器不能对与内存访问相关的操作进行优化或重排。
>
> 为什么需要这个约束呢？因为系统调用（syscall）可能会对内存中的数据进行读取或修改，而编译器在进行代码优化时通常会假设汇编代码不会影响内存中的数据。如果没有加上 “memory” 约束，编译器可能会错误地优化掉对内存的读写操作，导致系统调用出现问题。

`test_write.c`

c

```c
#include "stddef.h"
#include "stdint.h"
#include "stdio.h"
extern size_t syscall(size_t id, uintptr_t arg1, uintptr_t arg2, uintptr_t
arg3);
int main() {
    const char *message = "Hello, RISC-V!\n";
    int len = strlen(message);
    int ret = syscall(0x40,1,message, len);
    printf("ret:%d\n",ret);
    return 0;
}
```

这里就是去调用syscall函数去执行系统调用，它传递了四个参数：`0x40`代表系统调用号64，它是`write`系统调用的号码，在RISC-V下是用于输出信息到标准输出的；`1`是标准输出的文件描述符，`message`是要输出的字符串的地址，`len`是要输出的字符串的长度。

我一直想找在`RV64`的`linux`系统下的系统调用号是多少的文档，找了一圈找不到，最后没办法只有去看`linux`源码中的定义，在内核源码的`arch/riscv/include/uapi/asm/unistd.h`中，如下：

```
#ifdef __LP64__
#define __ARCH_WANT_NEW_STAT
#define __ARCH_WANT_SET_GET_RLIMIT
#endif /* __LP64__ */

#define __ARCH_WANT_SYS_CLONE3

#include <asm-generic/unistd.h>

/*
 * Allows the instruction cache to be flushed from userspace.  Despite RISC-V
 * having a direct 'fence.i' instruction available to userspace (which we
 * can't trap!), that's not actually viable when running on Linux because the
 * kernel might schedule a process on another hart.  There is no way for
 * userspace to handle this without invoking the kernel (as it doesn't know the
 * thread->hart mappings), so we've defined a RISC-V specific system call to
 * flush the instruction cache.
 *
 * __NR_riscv_flush_icache is defined to flush the instruction cache over an
 * address range, with the flush applying to either all threads or just the
 * caller.  We don't currently do anything with the address range, that's just
 * in there for forwards compatibility.
 */
#ifndef __NR_riscv_flush_icache
#define __NR_riscv_flush_icache (__NR_arch_specific_syscall + 15)
#endif
__SYSCALL(__NR_riscv_flush_icache, sys_riscv_flush_icache)

```

打开上面红色箭头这个头文件就能找到系统调用号的定义：

```
/* fs/read_write.c */
#define __NR3264_lseek 62
__SC_3264(__NR3264_lseek, sys_llseek, sys_lseek)
#define __NR_read 63
__SYSCALL(__NR_read, sys_read)
#define __NR_write 64
__SYSCALL(__NR_write, sys_write)
#define __NR_readv 65
__SC_COMP(__NR_readv, sys_readv, compat_sys_readv)
#define __NR_writev 66
__SC_COMP(__NR_writev, sys_writev, compat_sys_writev)
#define __NR_pread64 67
__SC_COMP(__NR_pread64, sys_pread64, compat_sys_pread64)
#define __NR_pwrite64 68
__SC_COMP(__NR_pwrite64, sys_pwrite64, compat_sys_pwrite64)
#define __NR_preadv 69
__SC_COMP(__NR_preadv, sys_preadv, compat_sys_preadv)
#define __NR_pwritev 70
__SC_COMP(__NR_pwritev, sys_pwritev, compat_sys_pwritev)

```

可以看见`write`的系统调用号是`64`即`0x40`。

编译，然后用qemu运行：

shell

```
timer@DESKTOP-JI9EVEH:~/quard-star/test/syscall$ make
timer@DESKTOP-JI9EVEH:~/quard-star/test/syscall$ qemu-riscv64
wirte.out
Hello, RISC-V!
ret:15
```

可以看见输出了`Hello, RISC-V!`，系统调用成功。`qemu-riscv64`模拟了一个`64`位的`linux`系统，所以可以加载`elf`格式的可执行文件运行。

2. 内核trap机制简介

首先先明确一下我们的目标是用户态的程序通过ecall指令陷入S态即我们的os，os需要对此次ecall进行处理，处理完毕后返回到用户态继续执行。应用程序被切换回来之后需要从发出系统调用请求的执行位置恢复应用程序上下文并继续执行，这需要在切换前后维持应用程序的上下文保持不变。应用程序的上下文包括通用寄存器和栈两个主要部分。由于 CPU 在不同特权级下共享一套通用寄存器，所以在运行操作系统的 Trap 处理过程中，操作系统也会用到这些寄存器，这会改变应用程序的上下文。因此，与函数调用需要保存函数调用上下文/活动记录一样，在执行操作系统的 Trap 处理过程（会修改通用寄存器）之前，我们需要在某个地方（某内存块或内核的栈）保存这些寄存器并在 Trap 处理结束后恢复这些寄存器。这里显而易见我们会使用栈来保存相关的寄存器。

2.1 与S模式相关的异常寄存器

与特权级无关的一般的指令和通用寄存器 `x0` ~ `x31` 在任何特权级都可以执行。而每个特权级都对应一些特殊指令和 **控制状态寄存器** (CSR, Control and Status Register)，来控制该特权级的某些行为并描述其状态。当然特权指令不仅具有读写 CSR 的指令，还有其他功能的特权指令。

如果处于低特权级状态的处理器执行了高特权级的指令，会产生非法指令错误的异常。这样，位于高特权级的执行环境能够得知低特权级的软件出现了错误，这个错误一般是不可恢复的，此时执行环境会将低特权级的软件终止。

在RV64架构下，寄存器的长度是64位。

2.1.1 Supervisor Status Register (sstatus)

63	62 … 34	33 … 32	31 … 20	19	18	17
SD	WPRI	UXL[1:0]	WPRI	MXR	SUM	WPRI
1	29	2	12	1	1	1

16 … 15	14 … 13	12 … 11	10 … 9	8	7	6	5	4 … 2	1	0
XS[1:0]	FS[1:0]	WPRI	VS[1:0]	SPP	WPRI	UBE	SPIE	WPRI	SIE	WPRI
2	2	2	2	1	1	1	1	3	1	1

Supervisor-mode status register (sstatus) when SXLEN=64.

这个寄存器我们主要关注的是`bit[8]:SPP`，该位表示cpu在进入S模式之前正在执行的特权级别。当接收到trap时，如果该trap来自用户模式，则SPP设置为0，否则设置为1。当执行一条SRET指令从trap处理程序返回时，如果SPP位为0，则特权级别被设置为U模式，如果SPP位为1，则特权级别被设置为S模式;SPP设置为0。

2.1.2 Supervisor Trap Vector Base Address Register (`stvec`)

SXLEN-1 … 2	1 … 0
BASE[SXLEN-1:2] (**WARL**)	MODE (**WARL**)
SXLEN-2	2

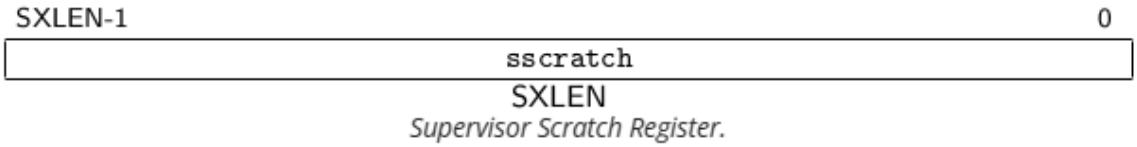
Supervisor trap vector base address register (stvec).

stvec寄存器用于设置发生trap时，异常处理程序的地址。

- MODE 位于 [1:0]，长度为 2 bits；
- BASE 位于 [63:2]，长度为 62 bits。

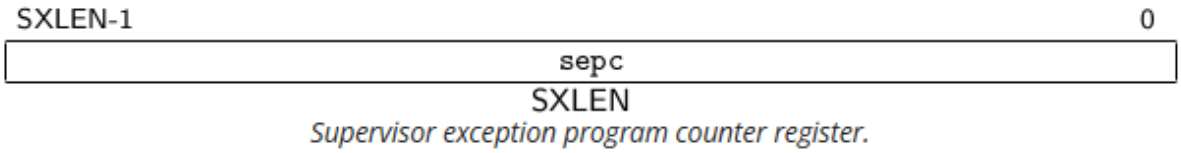
当 MODE 字段为 0 的时候， `stvec` 被设置为 Direct 模式，此时进入 S 模式的 Trap 无论原因如何，处理 Trap 的入口地址都是 `BASE<<2` ，CPU 会跳转到这个地方进行异常处理。当MODE 字段为 1 的时候，异常触发后会跳转到以BASE字段对应的异常向量表钟，每个向量占4个字节。

2.1.3 Supervisor Scratch Register (sscratch)



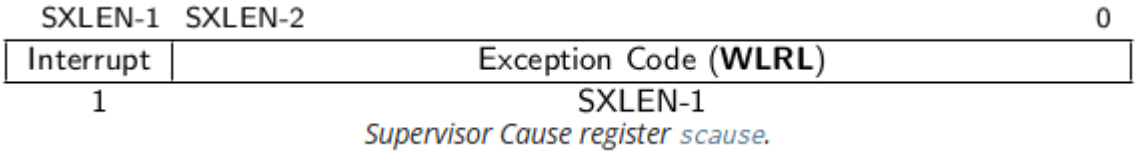
sscratch寄存器是一个可读/写的辅助寄存器，通常，在hart执行用户代码时，sscratch用于切换上下文的栈。

2.1.4 Supervisor Exception Program Counter (`sepc`)



sepc记录了 Trap 发生之前执行的最后一条指令的地址

2.1.5 Supervisor Cause Register (`scause`)

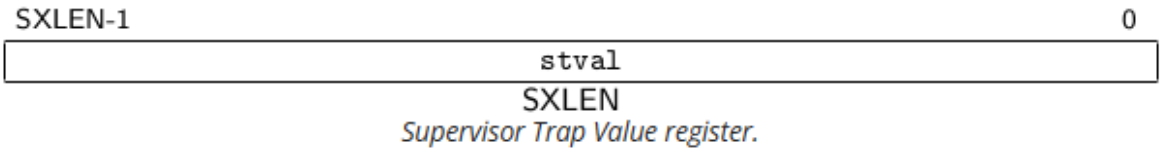


scause寄存器记录了S模式下异常发生的原因，最高位为`interrupt`字段，如下表所示，当`interrupt`字段为1时，代表触发的异常类型为中断类型。否则为同步类型异常。

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt

Interrupt	Exception Code	Description
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

2.1.6 Supervisor Trap Value (stval) Register



Supervisor Trap Value register.

当处理器陷入S模式时，stval寄存器记录了发生异常的虚拟地址。

> 更详细的寄存器解释可在这里看见：
>
> The RISC-V Instruction Set Manual, Volume II: Privileged Architecture | Five EmbedDev (five-embeddev.com)

2.2 特权级切换的软硬件控制机制

当 CPU 执行完一条指令（如 ecall ）并准备从用户特权级 陷入（ Trap ）到 S 特权级的时候，硬件会自动完成如下这些事情：

- `sstatus` 的 `SPP` 字段会被修改为 CPU 当前的特权级（U/S）。
- `sepc` 会被修改为 Trap 处理完成后默认会执行的下一条指令的地址。
- `scause/stval` 分别会被修改成这次 Trap 的原因以及相关的附加信息。
- CPU 会跳转到 `stvec` 所设置的 Trap 处理入口地址，并将当前特权级设置为 S ，然后从Trap 处理入口地址处开始执行。这里会根据`scause`中保存的异常原因进行分发处理

而当 CPU 完成 Trap 处理准备返回的时候，需要通过一条 S 特权级的特权指令 `sret` 来完成，这一条指令具体完成以下功能：

- CPU 会将当前的特权级按照 `sstatus` 的 `SPP` 字段设置为 U 或者 S ；
- CPU 会跳转到 `sepc` 寄存器指向的那条指令，然后继续执行。

在具体执行trap处理程序时，由于执行完毕后我们需要恢复到原来的地址继续执行所以我们需要保存寄存器的值，需要恢复知情trap前后的上下文信息，因此需要定义一个栈段来保存用户态的寄存器的值。所以os需要做的软件工作如下：

- 应用程序通过 `ecall` 进入到内核状态时，操作系统保存被打断的应用程序的 Trap 上下文；
- 操作系统根据Trap相关的CSR寄存器内容，完成系统调用服务的分发与处理；
- 操作系统完成系统调用服务后，需要恢复被打断的应用程序的Trap 上下文，并通 `sret` 让应用程序继续执行。

3. 为timer_os实现trap机制

我在os目录下新增了一个types.h的文件，里面声明了一些数据定义类型：

C

```
#ifndef __TYPES_H__
#define __TYPES_H__
// 定义无符号整型
typedef unsigned char uint8_t;
typedef unsigned short
uint16_t;
typedef unsigned int
uint32_t;
typedef unsigned long long
uint64_t;
/*
 * RISCV64: 寄存器的大小是64位
的
 */
typedef uint64_t reg_t;
#endif
```

rv64的寄存器是64位的，用typedef定义了一个reg_t的类型用于定义使用的寄存器

3.1 寄存器读写

我在os的目录下新建了一个riscv.h的文件，此文件中定义了一些获取寄存器值的函数。

c

```
#ifndef __RISCV_H__
#define __RISCV_H__

#include "os.h"
/* 读取 sepc 寄存器的值 */
static inline reg_t r_sepc()
{
  reg_t x;
  asm volatile("csrr %0, sepc" : "=r" (x)
);
  return x;
}
/* scause 记录了异常原因 */
static inline reg_t r_scause()
{
  reg_t x;
  asm volatile("csrr %0, scause" : "=r"
(x) );
  return x;
}
// stval 记录了trap发生时的地址
static inline reg_t r_stval()
{
  reg_t x;
  asm volatile("csrr %0, stval" : "=r"
(x) );
  return x;
}
/* sstatus记录S模式下处理器内核的运行状态
*/
static inline reg_t r_sstatus()
{
  reg_t x;
  asm volatile("csrr %0, sstatus" : "=r"
(x) );
  return x;
}
static inline void  w_sstatus(reg_t x)
{
  asm volatile("csrw sstatus, %0" : : "r"
(x));
}

/* stvec寄存器 */
static inline void  w_stvec(reg_t x)
{
  asm volatile("csrw stvec, %0" : : "r"
(x));
}
static inline reg_t r_stvec()
{
  reg_t x;
  asm volatile("csrr %0, stvec" : "=r"
(x) );
  return x;
}
#endif
```

可以看见用内联汇编的方式来读写与S态相关的控制寄存器的值。

3.2 用户栈和内核栈定义

当应用程序在用户态执行ecall指令陷入内核时，内核需要保存应用程序的各个寄存器的值，在内核中我们可以定义一个栈段来进行保存，同时为了为了安全机制，让用户程序不会干扰到内核栈，我们当用户程序在执行时专门为用户程序分配一段栈。由此需要定义一个内核栈专门给S态的内核使用，专门定义一个用户栈给用户程序使用。我在os目录下新建了一个`batch.c`的文件，在此文件中定义了`KernelStack`和`UserStack`

c

```
#define USER_STACK_SIZE (4096 *
2)
#define KERNEL_STACK_SIZE (4096 *
2)
uint8_t
KernelStack[KERNEL_STACK_SIZE];
uint8_t
UserStack[USER_STACK_SIZE];
```

`KernelStack`和`UserStack`的大小被定义为8kb。

3.3 Trap上下文执行流定义

Trap上下文执行流的数据就是寄存器中的数据，有`x0~x31`总共`32`个通用寄存器以及`sstatus`和`sepc`等控制寄存器需要保存。在os目录下新建了一个`context.h`

```
#ifndef __CONTEXT_H__
#define __CONTEXT_H__
#include "os.h"
/*S模式的trap上下文*/
typedef struct pt_regs
{
        reg_t x0;
        reg_t ra;
        reg_t sp;
        reg_t gp;
        reg_t tp;
        reg_t t0;
        reg_t t1;
        reg_t t2;
        reg_t s0;
        reg_t s1;
        reg_t a0;
        reg_t a1;
        reg_t a2;
        reg_t a3;
        reg_t a4;
        reg_t a5;
        reg_t a6;
        reg_t a7;
        reg_t s2;
        reg_t s3;
        reg_t s4;
        reg_t s5;
        reg_t s6;
        reg_t s7;
        reg_t s8;
        reg_t s9;
        reg_t s10;
        reg_t s11;
        reg_t t3;
        reg_t t4;
        reg_t t5;
        reg_t t6;
        /* S模式下的寄
存器 */
        reg_t sstatus;
        reg_t sepc;
}pt_regs;
#endif
```

3.4 Trap上下文的保存和恢复

在os目录下新建了一个`kerneltrap.S`的文件，此汇编文件中定义了两个函数：`__alltraps`、`__restore`。

首先来看`__alltraps`：

plaintext

```
.globl __alltraps
.align 4
__alltraps:
    # 从sscratch获取S模式下的SP，把U模式下的SP保存到sscratch寄存器
中
    csrrw sp, sscratch, sp
    # now sp->kernel stack, sscratch->user stack
    # allocate a TrapContext on kernel stack
    addi sp, sp, -34*8
    # save general-purpose registers
    sd x1, 1*8(sp)
    # skip sp(x2), we will save it later
    sd x3, 3*8(sp)
    # skip tp(x4), application does not use it
    # save x5~x31
    sd x4, 4*8(sp)
    sd x5, 5*8(sp)
    sd x6, 6*8(sp)
    sd x7, 7*8(sp)
    sd x8, 8*8(sp)
    sd x9, 9*8(sp)
    sd x10,10*8(sp)
    sd x11, 11*8(sp)
    sd x12, 12*8(sp)
    sd x13, 13*8(sp)
    sd x14, 14*8(sp)
    sd x15, 15*8(sp)
    sd x16, 16*8(sp)
    sd x17, 17*8(sp)
    sd x18, 18*8(sp)
    sd x19, 19*8(sp)
    sd x20, 20*8(sp)
    sd x21, 21*8(sp)
    sd x22, 22*8(sp)
    sd x23, 23*8(sp)
    sd x24, 24*8(sp)
    sd x25, 25*8(sp)
    sd x26, 26*8(sp)
    sd x27, 27*8(sp)
    sd x28, 28*8(sp)
    sd x29, 29*8(sp)
    sd x30, 30*8(sp)
    sd x31, 31*8(sp)

    # we can use t0/t1/t2 freely, because they were saved on kernel
stack
    csrr t0, sstatus
    csrr t1, sepc
    sd t0, 32*8(sp)
    sd t1, 33*8(sp)
    # read user stack from sscratch and save it on the kernel stack
    csrr t2, sscratch
    sd t2, 2*8(sp)
    # set input argument of trap_handler(TrapContext)
    mv a0, sp
    call trap_handler
```

`__alltraps` 函数就是发生异常时的处理函数，在此函数中：

- 第五行中将sscratch和sp的值进行了交换，在进入此函数时sp指向的是用户栈，sscratch中的值保存的是内核栈的栈顶。进行交换后，由于此时进入了S态，所以需要切换栈，由此就切换到了内核栈。
- 然后就是将寄存器的值保存进内核栈中，在上面上下文的定义可以看见pt_regs中定义了34个寄存器，所以通过`addi sp, sp, -34*8`指令来压栈，然后依次保存寄存器的值
- 最后两行将内核栈的sp保存进a0寄存器用于传参，所以将用户态寄存器保存进内核栈后，调用了`trap_handler`函数，在此函数中可通过a0传入的参数访问内核栈中储存的寄存器的值。

然后是`__restore`函数，此函数需要将内核栈中的存储的寄存器的值恢复，然后通过`sret`指令返回从S态到用户态继续执行。

plaintext

```
.globl __restore
.align 4
__restore:
    # case1: start running app by __restore
    # case2: back to U after handling trap
    mv sp, a0
    # now sp->kernel stack(after allocated), sscratch->user
stack
    # restore sstatus/sepc
    ld t0, 32*8(sp)
    ld t1, 33*8(sp)
    ld t2, 2*8(sp)
    csrw sstatus, t0
    csrw sepc, t1
    csrw sscratch, t2
    # restore general-purpuse registers except sp/tp
    ld x1, 1*8(sp)
    ld x3, 3*8(sp)
    ld x4, 4*8(sp)
    ld x5, 5*8(sp)
    ld x6, 6*8(sp)
    ld x7, 7*8(sp)
    ld x8, 8*8(sp)
    ld x9, 9*8(sp)
    ld x10,10*8(sp)
    ld x11, 11*8(sp)
    ld x12, 12*8(sp)
    ld x13, 13*8(sp)
    ld x14, 14*8(sp)
    ld x15, 15*8(sp)
    ld x16, 16*8(sp)
    ld x17, 17*8(sp)
    ld x18, 18*8(sp)
    ld x19, 19*8(sp)
    ld x20, 20*8(sp)
    ld x21, 21*8(sp)
    ld x22, 22*8(sp)
    ld x23, 23*8(sp)
    ld x24, 24*8(sp)
    ld x25, 25*8(sp)
    ld x26, 26*8(sp)
    ld x27, 27*8(sp)
    ld x28, 28*8(sp)
    ld x29, 29*8(sp)
    ld x30, 30*8(sp)
    ld x31, 31*8(sp)

    # release TrapContext on kernel stack
    addi sp, sp, 34*8
    # now sp->kernel stack, sscratch->user stack
    csrrw sp, sscratch, sp
    # now sp->user stack, sscratch->kernel stack
    sret
```

- `__restore`函数的定义为`__restore(pt_regs *next)`，所以在第一行传入内核栈地址，然后将内核栈中存放的寄存器的值恢复，然后切换sp，最后通过sret返回用户态继续执行
- 在最后两行会将`sp`指向用户栈，`sscratch`指向内核栈

3.5 编写应用程序测试

3.5.1 编写应用程序

我在`batch.c`中新增了一段用户代码的程序：

c

```
size_t syscall(size_t id, reg_t arg1, reg_t arg2, reg_t
arg3) {
    long ret;
    asm volatile (
        "mv a7, %1\n\t"   // Move syscall id to a7
register
        "mv a0, %2\n\t"   // Move args[0] to a1 register
        "mv a1, %3\n\t"   // Move args[1] to a2 register
        "mv a2, %4\n\t"   // Move args[2] to a3 register
        "ecall\n\t"       // Perform syscall
        "mv %0, a0"       // Move return value to 'ret'
variable
        : "=r" (ret)
        : "r" (id), "r" (arg1), "r" (arg2), "r" (arg3)
        : "a7", "a0", "a1", "a2", "memory"
    );
    return ret;
}

void testsys() {

    syscall(2,3,4,5);
    syscall(1,1,1,1);
    syscall(1,2,3,4);
    while (1)
    {

    }
}
```

可以看见在`testsys()`函数中，调用了三次syscall函数进行测试，每一次传入的参数都不同，仅用于测试。

3.5.2 trap.c

在上面的`__alltraps`函数中，调用了`trap_handler`函数对异常进行处理，因此我们需要实现此函数，我在os目录下定义了一个`trap.c`的文件：

c

```
extern void __alltraps(void);
extern void __restore(pt_regs *next);
pt_regs* trap_handler(pt_regs* cx)
{
    reg_t scause = r_scause() ;
    printf("cause:%x\n",scause);
        printf("a0:%x\n",cx->a0);
        printf("a1:%x\n",cx->a1);
        printf("a2:%x\n",cx->a2);
        printf("a7:%x\n",cx->a7);
        printf("sepc:%x\n",cx->sepc);
        printf("sstatus:%x\n",cx-
>sstatus);
        printf("sp:%x\n",cx->sp);

        cx->sepc += 8;
        __restore(cx);
        return cx;
}
void trap_init()
{
        /*
         * 设置 trap 时调用函数的基地
址
         */
        w_stvec((reg_t)__alltraps);
}
```

这里定义了两个函数，其中trap_init用于设置stvec寄存器的值，这里是告诉cpu发生异常时处理函数的地址，将其设置为 __alltraps的地址。

trap_handler函数中打印了内核栈中储存的寄存器的值，在syscall中对a0,a1,a2,a7寄存器的值进行了修改，这些寄存器的值通过__alltraps函数会保存在内核栈中，然后将内核栈的地址放入a0寄存器中作为函数参数传了出来，因为我们可以在此来进行异常的分发，这里只是打印传入的系统调用参数来验证。在系统调用的逻辑处理完后，需要将sepc 的值+8，然后调用

__restore函数来恢复寄存器的值，同时切换栈指针到用户栈，并通过sret返回到sepc+8的地址处继续执行代码。

3.5.3 测试代码

我们直接看代码，再分析逻辑，在batch.c中新增了如下代码：

代码测试逻辑是：伪造一个内核栈，然后通过__restore函数从S态返回U态进行函数执行，返回的地址设置为testsys()函数的地址

c

```
extern void __restore(pt_regs *next);

struct pt_regs tasks;
void app_init_context()
{

    reg_t user_sp = &UserStack + USER_STACK_SIZE;

    trap_init();

    reg_t sstatus = r_sstatus();
    // 设置 sstatus 寄存器第8位即SPP位为0 表示为U模式
    sstatus &= (0U << 8);
    w_sstatus(sstatus);

    tasks.sepc = (reg_t)testsys;
    tasks.sstatus = sstatus;
    tasks.sp = user_sp;


    pt_regs* cx_ptr = &KernelStack[0] + KERNEL_STACK_SIZE -
sizeof(pt_regs);
    cx_ptr->sepc = tasks.sepc;
    cx_ptr->sstatus = tasks.sstatus;
    cx_ptr->sp = tasks.sp;

    __restore(cx_ptr);
}
```

- 首先得到了用户栈的地址，因为栈是从高地址往低地址向下增长的，所以用户栈的地址为`&UserStack + USER_STACK_SIZE`
- 然后调用`trap_init`函数来设置`stvec`寄存器的值为`__alltraps`，这里告诉cpu发生trap时去哪里执行
- 然后设置sstatus寄存器的SPP位为0。这是为啥呢？在上面对寄存器的介绍中提到“当执行一条SRET指令从trap处理程序返回时，如果SPP位为0，则特权级别被设置为U模式，如果SPP位为1，则特权级别被设置为S模式;”所以我们为了从S模式返回用户模式去执行testsys()中的代码，我们需要将SPP位设置为0。
- 然后就是事先构造一段内核栈，设置`sstatus`、`sepc`、`sp`的值，这里由于下一阶段为用户模式，所以sepc会设置成用户态程序的地址，sp设置为用户栈的地址。
- 设置完成后调用`__restore`函数，让其返回用户态执行程序。

3.5.4 编译测试

在`main.c`中调用`app_init_context();`函数：

c

```
#include "os.h"
void os_main()
{
   printf("hello timer
os!\n");
   app_init_context();
}
```

修改Makefile，添加新增的源文件

c

```
SRCS_ASM = \
        entry.S
\

kerneltrap.S \

SRCS_C = \
        sbi.c \
        main.c \
        printf.c
\
        batch.c
\
        trap.c \
```

Makefile还修改了一个地方

makefile

```
os.elf: ${OBJS}
        ${CC} ${CFLAGS} -T os.ld -Wl,-Map=os.map -o
os.elf $^
        ${OBJCOPY} -O binary os.elf os.bin
```

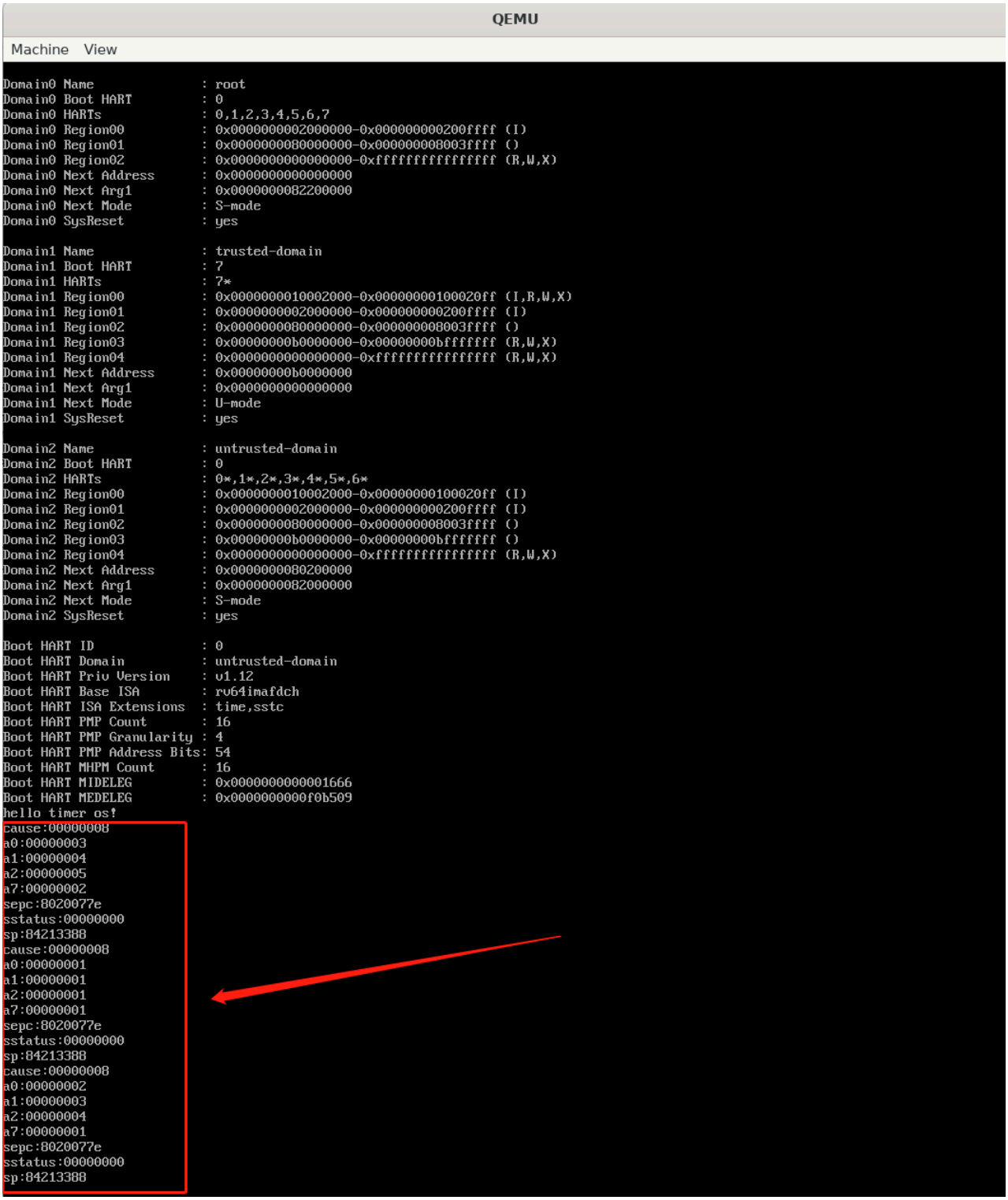
这里新增了-Wl,-Map=os.map选项，会在编译时生成一个os.map的符号表用于调试。

回到timer@DESKTOP-JI9EVEH:~/quard-star$目录，构建执行：

shell

```
timer@DESKTOP-JI9EVEH:~/quard-star$
./build.sh
timer@DESKTOP-JI9EVEH:~/quard-star$
./run.sh
```

结果如下：



可以看见进行系统调用的参数都成功打印，验证成功。cause的值为8，对应上面的异常原因表可以看见是U模式的系统调用。

> 但是这里有个很奇怪的点就是，每次返回的sepc都是同一个地址，我很奇怪，按道理来说每次syscall都会去调用一次ecall指令，所以sepc应该会被设置成每次syscall的ecall的地址，由于我进行了多次syscall调用sepc会不同，但是实际上每次都sepc都被设置成了同一个ecall的地址。转念一想，编译器确实在处理syscall函数时，其中ecall这条指令的地址始终是不会变的，但是我疑惑为什么将此地址+8后，就能跳到下一条正确的地址执行.....

参考链接

说明一下，我的`timer_os`其实是在复现并修改`rCore`，`rCore`是`rust`编写的`os`，我想把它改成c语言的用于个人学习。此篇文章对应的是`rCore`第二章-批处理系统的内容

相关推荐