# ARM64中断处理过程

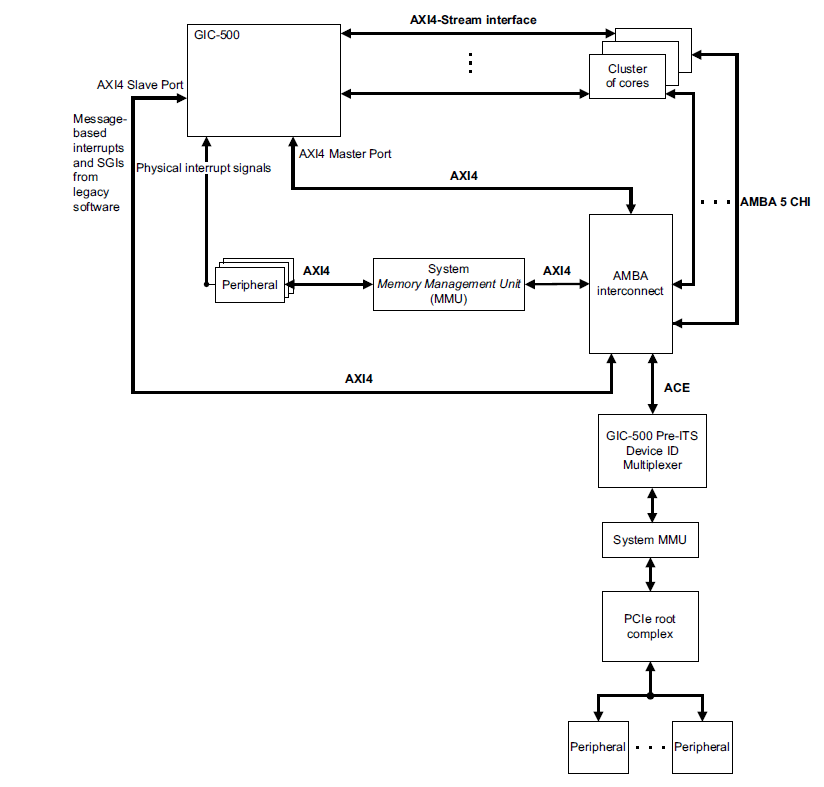
Linhai Zhou（David） ver 0.1 Draft

中断控制器基本上都是定义在device tree文件里面，比如K3的中断处理器定义：

gic500: interrupt-controller@1800000

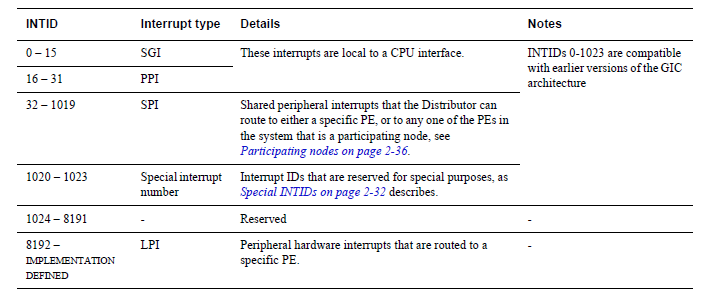
从中得知K3用的中断控制器是GIC500。

GIC-500最高支持128cores，仅支持ARMv8架构和通过标准GIC协议流到GIC CPU接口的核，比如A53/A57。



GIC-500通过AXI4 Slave port或者Physical interrupt singals口来接收中断。支持的中断：

1. SGI 这是处理器间中断，由一个core产生并发送到其他核。在一个core上激活SGI并不会影响其他核上相同的ID的SGI，这意味着发送SGI给所有的core，每个core处理是独立的。可以在core上利用系统寄存器，或软件写SGCD\_SGIR来产生SGI中断
2. PPI 这是外设与core紧密耦合中断，意味着PPI中断只会发送给跟这个外设耦合的core上。PPI默认是低电平触发，当然可以编程为上升沿触发（edge-trigger）
3. SPI 与PPI相反，这意味着SGI在每个core上是共享的。SPI可以通过线路中断输入或者写AXI4 slave programming interface来产生。GIC-500支持960个SPI，从32-991。可配置上升沿触发（同PPI）或高电平触发（level-trigger）
4. LPI 通常用于基于消息的外设中断，比如PICe中断。不同于SPI，在给定时间内LPI只会给一个core处理。ITS可以提供中断ID转换，这样允许外设（这些外设可以使用System MMU）直接给虚拟机使用（EL2）。GIC-500 LPI只能通过AXI4 Slave programming interface写入ITS来产生LPI。GIC-500支持57344个LPI数目，从8192开始。并有一个保存最频繁中断的缓存。



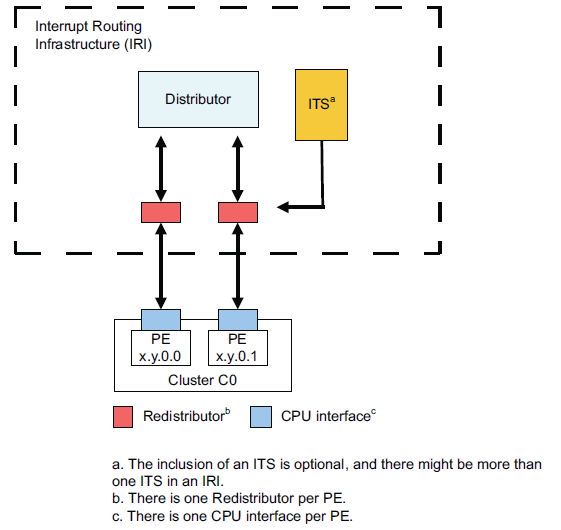
通过这几个终端类型我们可以知道，SPI和LPI是我们所需要特别关注的中断类型，那么我们是选择SPI还是LPI呢？

我们可以把LPI和SPI用于基于消息的中断。但：

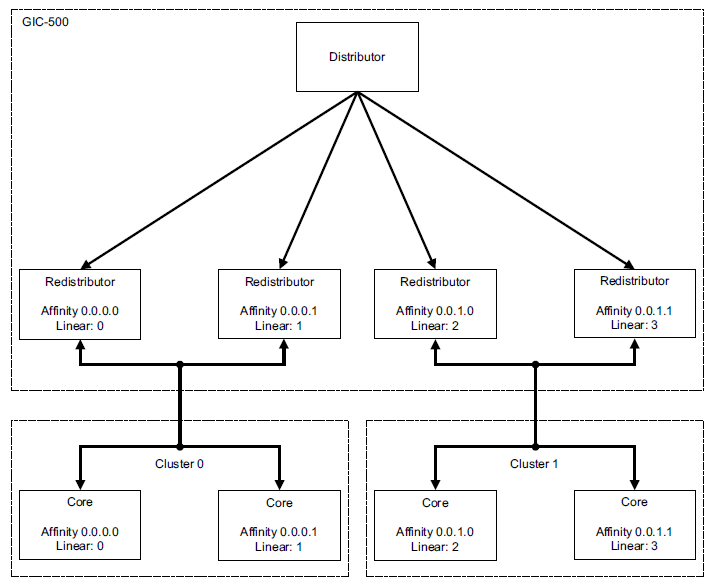
1. 只有ITS提供中断ID的服务，所以针对虚拟机外设，LPI更适合
2. LPI通常在Interrupt Group1 Non-secure。因此对于安全软件显然只能用SPI
3. 只有SPI才能提供把中断送到所有核，这意味着GIC-500可以做中断自动均衡。
4. LPI的中断数目比SPI多
5. 一些小型系统是没有ITS支持的，所以只能用SPI
6. 延时敏感类中断最好使用SPI。因为SPI是在GIC-500内部，而不是缓存。

GIC-500可分成3个部分：

1. ITS 负责将基于消息的中断转成LPI
2. Distributor 分发器接收来自线路中断或编程接口的中断，他需要负责中断的优先级，并通过标准GIC协议流接口送到CPU接口上（PE）
3. Redistributor 每一个core都对应一个重分发器，他保存着一个特定core的状态，如PPI、SGI的设置。他还要存储使用ITS生成该core的LPI。



每部分都有特定的寄存器，如GICD\_\*, GICR\_\*, GITS\_\*。



GIC-500的IP是GICv3，GICv3有3个interrupt groups。

GROUP0：为EL3层级处理的中断。

GROUP1(Secure)：为secure EL1处理的中断

GROUP1(Non-secure)：为Non-secure EL2（虚拟机）或者Non-secure EL1处理的中断。

Linux Kernel是运行在EL1（Non-secure）上，所以用的是GROUP1（Non-secure）

接下来我们来观察一下kernel的实现，首先看一下device tree的描述：

gic500: interrupt-controller@1800000 {

compatible = "arm,gic-v3";

#address-cells = <2>;

#size-cells = <2>;

ranges;

#interrupt-cells = <3>;

interrupt-controller;

reg = <0x00 0x01800000 0x00 0x10000>, /\* GICD \*/

<0x00 0x01880000 0x00 0x90000>; /\* GICR \*/

interrupts = <GIC\_PPI 9 IRQ\_TYPE\_LEVEL\_HIGH>;

gic\_its: msi-controller@1820000 {

compatible = "arm,gic-v3-its";

reg = <0x00 0x01820000 0x00 0x10000>;

socionext,synquacer-pre-its = <0x1000000 0x400000>;

msi-controller;

#msi-cells = <1>;

};

};

从device tree可以明确Ti-K3用的中断控制器是GIC-500，控制器地址是0x1800000（物理地址还是虚拟地址？）

GICD寄存器组地址0x1800000， size=0x10000.

GICR寄存器组地址0x1880000， size=0x90000.

GITS寄存器组地址0x1820000， size=0x10000. PCIe用的是LPI，需要ITS来转换interrupt ID，所以msi-controller在gic-its下面。

当系统发生中断后，我们都知道需要有中断处理程序去处理。那么Kernel是怎么做到的呢？

首先ARM64有4个异常级别，从EL0-EL3

EL0：无特权级别，应用程序运行于此

EL1：特权级别，kernel就运行于此级别

EL2：Hypervisor，虚拟化监控程序运行于此级别。Linux下常见就是KVM，会运行在这个级别（KVM也可以运行在EL1）

EL3：Secure mode，这个级别我们是碰不到的，可以不去管它。

本文也不准备将虚拟化，所以我们着重看EL1和EL0。

ARM64下面，异常分为同步和异步两种模式：

**同步异常：**如果一个异常是由于执行或试图执行指令而产生的，并且返回地址提供了引起异常的指令细节。

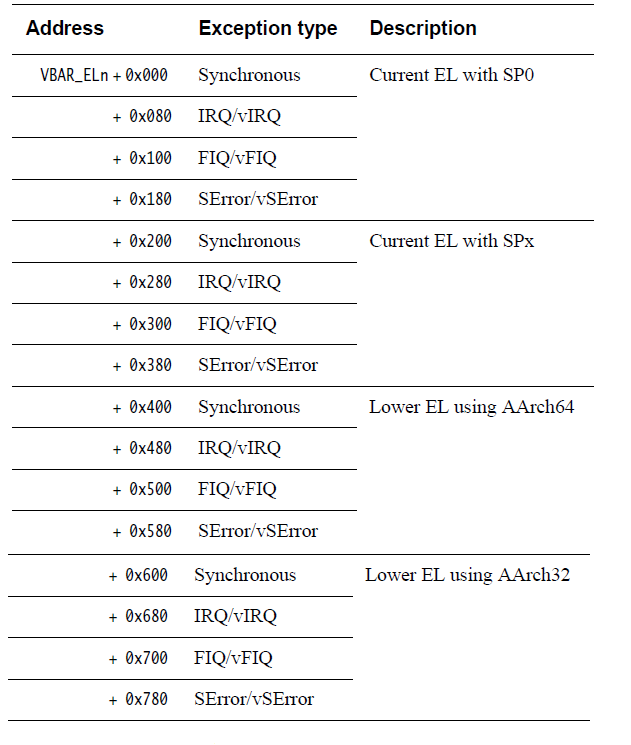
**异步异常：**如果一个异常并不是由执行指令产生的，而返回地址可能并不能提供导致异常的细节。

根据定义很容易得出：

1. MMU指令引起的异常（缺页）
2. MMU数据abort引起的异常（对齐）
3. SP和PC对齐检查
4. 未分配指令
5. 异常调式（gdb）
6. 系统调用

以上都是同步异常。除了这些都是异步异常

每个异常级别都有一个异常向量表，表地址保存在VBAR\_ELn下面。该表分成4组，每组4个异常入口，总共16个异常向量。



第一部分Current EL with SP0：这部分表示当异常发生但是没有EL切换，且使用的栈指针SP\_EL是SP\_EL0

第二部分Current EL with SPx:这部分表示当异常发生但是没有EL切换，且使用的栈指针SP\_EL是SP\_EL1/2/3

第三部分Lower EL using AARCH64：这部分表示当异常发生但是EL切换了，并且EL比目的EL低一级运行，只在64位下

第四部分Lower EL using AARCH32: 同第三部分，但是只在32位下。

我们可以分析出：

第一部分：在EL0下发生的任何异常都会在EL1下处理，所以这部分是无效的。

第二部分：在Kernel或者Hypervisor或者Secure mode下的中断发生，且不切换EL时处理的。

第三部分：64位下，用户态发生异常来处理的。

第四部分：32位下，用户态发生异常来处理的。

有了前面的铺垫，接下来就是看内核怎么处理这些工作。

Let’s go

从一个系统调用看起：int fd = socket(AF\_INET, SOCK\_DGRAM,0);

GLIBC调用宏INTERNAL\_SYSCALL，这个宏把name，我们这里的socket转化成\_\_NR\_socket宏，在aarch64下面为198，这就是我们所知的系统调用号。

接下来把参数0-3，压入x0,x1,x2寄存器。把系统调用号压入x8寄存器。通过svc触发异常。上面得知，EL0的异常会路由到EL1去处理，所以会执行向量表的第三部分。从异常的分类上可以得知的是这是一个同步异常。执行0x400处的异常入口。

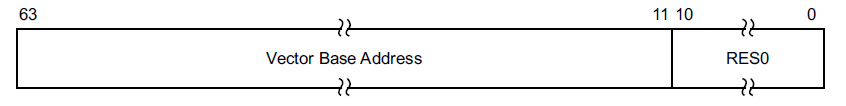
那么0x400处的异常是怎么定义的呢：

当uboot把控制权交给kernel以后，kernel在\_\_primary\_switch：

adr\_l x8, vectors

msr vbar\_el1, x8

这两句话表示把vectors的地址赋值给x8，然后通过msr把x8保存的值送入vbar\_el1里面。vbar\_el1就是EL1下面的一场向量表地址寄存器。



为什么是11：63有效？从前面可以看出每个vectors表是0x800对齐，故低11位是无效的。从vectors的定义也能看出：

/\*

\* Exception vectors.

\*/

.pushsection ".entry.text", "ax"

.align 11

SYM\_CODE\_START(vectors)

kernel\_ventry 1, sync\_invalid // Synchronous EL1t

kernel\_ventry 1, irq\_invalid // IRQ EL1t

kernel\_ventry 1, fiq\_invalid // FIQ EL1t

kernel\_ventry 1, error\_invalid // Error EL1t

kernel\_ventry 1, sync // Synchronous EL1h

kernel\_ventry 1, irq // IRQ EL1h

kernel\_ventry 1, fiq\_invalid // FIQ EL1h

kernel\_ventry 1, error // Error EL1h

kernel\_ventry 0, sync // Synchronous 64-bit EL0

kernel\_ventry 0, irq // IRQ 64-bit EL0

kernel\_ventry 0, fiq\_invalid // FIQ 64-bit EL0

kernel\_ventry 0, error // Error 64-bit EL0

#ifdef CONFIG\_COMPAT

kernel\_ventry 0, sync\_compat, 32 // Synchronous 32-bit EL0

kernel\_ventry 0, irq\_compat, 32 // IRQ 32-bit EL0

kernel\_ventry 0, fiq\_invalid\_compat, 32 // FIQ 32-bit EL0

kernel\_ventry 0, error\_compat, 32 // Error 32-bit EL0

#else

kernel\_ventry 0, sync\_invalid, 32 // Synchronous 32-bit EL0

kernel\_ventry 0, irq\_invalid, 32 // IRQ 32-bit EL0

kernel\_ventry 0, fiq\_invalid, 32 // FIQ 32-bit EL0

kernel\_ventry 0, error\_invalid, 32 // Error 32-bit EL0

#endif

SYM\_CODE\_END(vectors)

这段代码是2048位对齐的。

kernel\_ventry是一个宏：

.macro kernel\_ventry, el, label, regsize = 64

.align 7

sub sp, sp, #S\_FRAME\_SIZE

b el\()\el\()\_\label

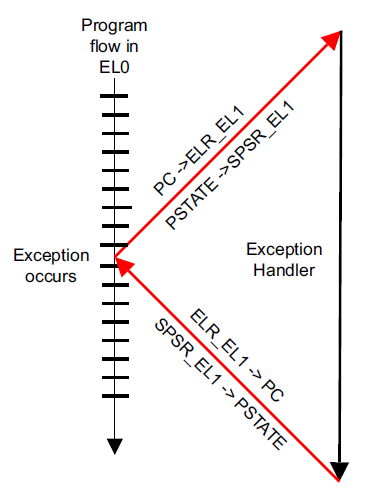
.endm

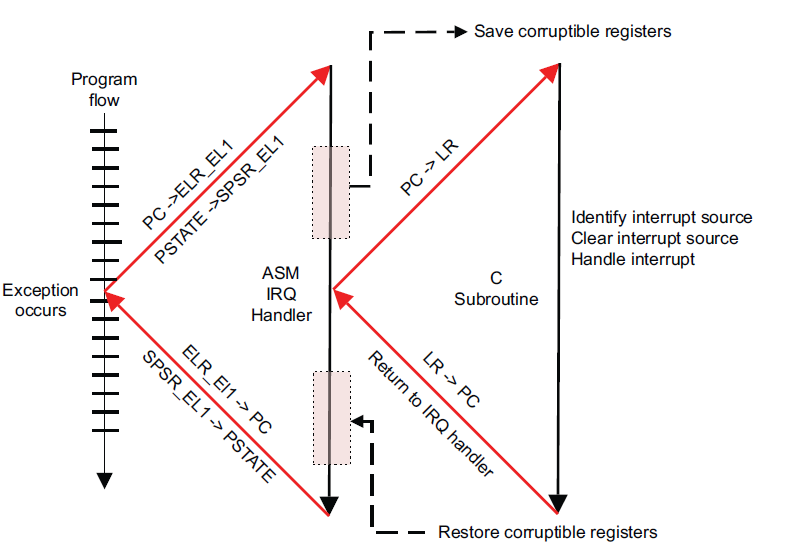
这个宏是128位对齐，vectors表里面正好对应每个向量入口是0xFF大小。

这里值得注意的是：

当处理器在AArch64执行状态下发生异常时，所有的PSTATE中断掩码被自动设置。这意味着进一步的异常被禁止。如果软件要支持嵌套异常，例如，允许一个较高优先级的中断中断处理一个较低优先级的源，那么软件需要明确地重新启用中断。

我们回到之前，svc产生一个同步异常，aarch64是怎么处理的？

<pic exp\_procedure>

 <pic：intr\_procedure>

PC是程序计数器，表示下一条要执行的指令。

ELR\_EL1 当EL1进行异常处理时，保存要返回的地址，也就是异常前要执行的下一条指令

SPSR\_EL1 当EL进行异常处理时，保留保存的进程状态

接下来ARM64调用kernel\_ventry 0, sync。

kernel\_ventry：

sub sp, sp, #S\_FRAME\_SIZE

分配中断栈，栈大小位pt\_regs结构体大小。接下来测试栈是否溢出：

add sp, sp, x0 // sp' = sp + x0

sub x0, sp, x0 // x0' = sp' - x0 = (sp + x0) - x0 = sp

tbnz x0, #THREAD\_SHIFT, 0f

sub x0, sp, x0 // x0'' = sp' - x0' = (sp + x0) - sp = x0

sub sp, sp, x0 // sp'' = sp' - x0 = (sp + x0) - x0 = sp

b el\()\el\()\_\label

这里用了一个小技巧，sp栈指针先加上x0的值。比较栈是否溢出，只要跟1<<THREAD\_SHIFT相于一下，不是0就表明溢出，需要到标号0这段去处理。否则跳到el0\_sync。那sp+x0相当于保存x0的值，因为我们不能直接拿sp来做任何运算。前面也说了x0是系统调用的第一个参数值（AF\_INET），并不能丢。

我们假设一切正常，于是来到：

SYM\_CODE\_START\_LOCAL\_NOALIGN(el0\_sync)

kernel\_entry 0

mov x0, sp

bl el0\_sync\_handler

b ret\_to\_user

SYM\_CODE\_END(el0\_sync)

据图intr\_procedure，进入exception Handle后先要保存register。

Kernel\_entry 0

stp x0, x1, [sp, #16 \* 0]

stp x2, x3, [sp, #16 \* 1]

stp x4, x5, [sp, #16 \* 2]

stp x6, x7, [sp, #16 \* 3]

stp x8, x9, [sp, #16 \* 4]

stp x10, x11, [sp, #16 \* 5]

stp x12, x13, [sp, #16 \* 6]

stp x14, x15, [sp, #16 \* 7]

stp x16, x17, [sp, #16 \* 8]

stp x18, x19, [sp, #16 \* 9]

stp x20, x21, [sp, #16 \* 10]

stp x22, x23, [sp, #16 \* 11]

stp x24, x25, [sp, #16 \* 12]

stp x26, x27, [sp, #16 \* 13]

stp x28, x29, [sp, #16 \* 14]

这里保存所有的寄存器到栈（内核进程栈,进行EL切换时SP\_ELn会自动切换）上。

mrs x21, sp\_el0

ldr\_this\_cpu tsk, \_\_entry\_task, x20

msr sp\_el0, tsk

把sp\_el0（用户栈）读入x21。

读取当前cpu中当前进程地址到tsk，然后保存到sp\_el0中，因为现在使用的时sp\_el1，sp\_el0暂时不使用。

ldr x19, [tsk, #TSK\_TI\_FLAGS]

disable\_step\_tsk x19, x20

这里清mdscr\_el1的SS位。mdscr\_el1调试实现的寄存器，其中ss[bit0]是Software step control bit

0 -> disabled

1 -> enabled

mrs x22, elr\_el1

mrs x23, spsr\_el1

stp lr, x21, [sp, #S\_LR]

elr\_el1 这里保存的是返回地址，存入x22。Spsr\_el1保存的进程状态，存入x23。

add x29, sp, #S\_STACKFRAME

stp x22, x23, [sp, #S\_PC]

mov w21, #NO\_SYSCALL

str w21, [sp, #S\_SYSCALLNO]

x29保存栈顶地址，设置NO\_SYSCALL。SYSCALL nr会在el0\_svc里面读取。

执行完kernel\_entry后：

mov x0, sp

bl el0\_sync\_handler

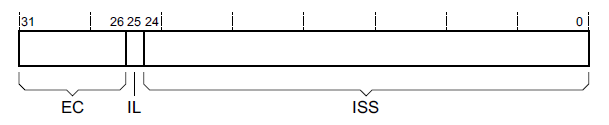
b ret\_to\_user

把sp栈底地址赋值给x0，x0之前是保存第一个参数的，那么现在被重新赋值了，参数去哪里了？

调用el0\_sync\_handler处理函数。

unsigned long esr = read\_sysreg(esr\_el1);

esr\_el1保存异常的信息。

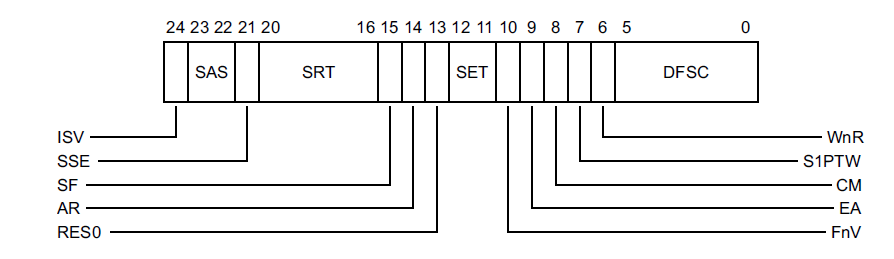


EC, bits[31:26] The Exception class field, that indicates the cause of the exception.

IL, bit[25] The Instruction length bit, for synchronous exceptions, that indicates whether a trapped instruction was a 16-bit or a 32-bit instruction.

ISS, bits[24:0] The Instruction specific syndrome field. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

ISS在kernel用的比较多的是



这个结构，DFSC指明了数据故障状态码。定义在fault\_info结构体里面。

读出esr\_el1后，从EC可以得到异常。这里是svc调用，所以EC是010101 SVC instruction execution in AArch64 state转入el0\_svc(regs);

这里的regs就是x0寄存器，我们这道这是SP栈底指针，被转为pt\_regs结构体。

最终调用el0\_svc\_common(regs, regs->regs[8], \_\_NR\_syscalls, sys\_call\_table);

GLIBC已经把x8设置成系统调用号，那么这里的regs[8]就是stp x8, x9, [sp, #16 \* 4]这条指令保存的。

el0\_svc\_common这里需要注意的是regs[0]既是参数，也是返回值。

static void el0\_svc\_common(struct pt\_regs \*regs, int scno, int sc\_nr,

const syscall\_fn\_t syscall\_table[])

{

unsigned long flags = current\_thread\_info()->flags;

regs->orig\_x0 = regs->regs[0];

regs->syscallno = scno;

cortex\_a76\_erratum\_1463225\_svc\_handler();

local\_daif\_restore(DAIF\_PROCCTX);

if (system\_supports\_mte() && (flags & \_TIF\_MTE\_ASYNC\_FAULT)) {

regs->regs[0] = -ERESTARTNOINTR;

return;

}

if (has\_syscall\_work(flags)) {

if (scno == NO\_SYSCALL)

regs->regs[0] = -ENOSYS;

scno = syscall\_trace\_enter(regs);

if (scno == NO\_SYSCALL)

goto trace\_exit;

}

invoke\_syscall(regs, scno, sc\_nr, syscall\_table);

if (!has\_syscall\_work(flags) && !IS\_ENABLED(CONFIG\_DEBUG\_RSEQ)) {

local\_daif\_mask();

flags = current\_thread\_info()->flags;

if (!has\_syscall\_work(flags) && !(flags & \_TIF\_SINGLESTEP))

return;

local\_daif\_restore(DAIF\_PROCCTX);

}

trace\_exit:

syscall\_trace\_exit(regs);

}

local\_daif\_restore(DAIF\_PROCCTX);这个函数的作用就是重置DAIF。使得后续的异常能够正常到达。

接下来就比较简单了，拿出系统调用号198，在unistd.h里面定义号的系统调用号对比，执行socket函数即可。

SYM\_CODE\_START\_LOCAL(ret\_to\_user)

disable\_daif

gic\_prio\_kentry\_setup tmp=x3

ldr x19, [tsk, #TSK\_TI\_FLAGS]

and x2, x19, #\_TIF\_WORK\_MASK

cbnz x2, work\_pending

finish\_ret\_to\_user:

user\_enter\_irqoff

clear\_mte\_async\_tcf

enable\_step\_tsk x19, x2

kernel\_exit 0

work\_pending:

mov x0, sp

mov x1, x19

bl do\_notify\_resume

ldr x19, [tsk, #TSK\_TI\_FLAGS] // re-check for single-step

b finish\_ret\_to\_user

SYM\_CODE\_END(ret\_to\_user)

这个函数第一步就是关闭DAIF。从内核返回到用户态有很多事情要处理，比如是否有调度需求？有没有未决信号要处理？

ARM64使用TSK\_TI\_FLAGS标志和\_TIF\_WORK\_MASK来比较，如果没有任何事情需要处理，那么调用kernel\_exit返回，kernel\_exit就是把保存在栈上的寄存器值恢复到相应的寄存器，特别是SP\_EL0，用户栈的恢复，以及PC。

有悬而未决的事务，就要跳到work\_pending这个慢处理机制上。主要执行do\_notify\_resume。

do\_notify\_resume先检查\_TIF\_NEED\_RESCHED，标记了就调用schedule（这部分在Kernel调度里讲诉）注意这里会把中断禁用。

还会检查

if (thread\_flags & \_TIF\_SIGPENDING)

do\_signal(regs);

这里就是信号处理函数执行的地方。

处理号所有的TI\_FLAGS后，执行kernel\_exit返回。

以上就是用户态产生同步异常后的处理情况。

内核态产生同步异常：

因为kernel是跑在EL1上，在这里产生的异常时不会更改EL的，所以处理的程序是el1\_sync。

缺页异常：

这在Kernel里面是最常见的一种同步异常，接下来我们来看看缺页是怎么处理的：

中断流程：

中断有2中情况：

1. 在用户态收到中断，执行中断处理
2. 在内核态收到中断，执行中断处理

第一种情况：