OS拾贝-5 页写入异常的处理流程

关键词：三个指针（内核态的异常处理指针 `KERNEL_SP - TF_SIZE`，用户态的被 `page_fault_handler` 设置在[UXSTACKTOP - BY2PG, UXSTACKTOP - 1]区域的指针，以及**真正发生页写入异常**的用户态栈指针）

两个EPC：一个是 `page_fault_handler` 函数给 `KERNEL_SP - TF_SIZE` 设置的epc，存放 `__asm_pgfault_handler` 的地址，用于在回到用户态的时候，系统处于这个函数的地址。

一个则是在 设置在[UXSTACKTOP - BY2PG, UXSTACKTOP - 1]区域的指针 指向的环境中，存放的**真正发生页写入异常**的epc值。

*Step*0 触发页写入异常，进入内核态

用户态下，当对带有PTE_COW的页面进行写入时，就会触发写入页异常，系统进入内核态。

> **中断异常入口**
>
> 根据R3000手册, R3000的中断异常入口有5个, 此处列出其中在MOS中最重要的两个入口:
>
> | 入口地址 | 所在内存区 | 描述 |
> | --- | --- | --- |
> | `0x80000000` | kseg0 | TLB缺失时, PC转至此处 |
> | `0x80000080` | kseg0 | 对于除了TLB缺失之外的其他异常, PC转至此处 |

> 注意，在MOS中，仅仅实现了 `0x80000080` 处的中断异常处理程序。这样也能工作的原因是：当TLB缺失时，PC转至 `0x80000000` 后，**空转32周期(执行了32条 `nop` 指令)**，到达 `0x80000080` ，随后也就与其他的中断异常一同处理了。也就是说我们只需要实现 `0x80000080` 处的程序即可。

因此，**一旦 CPU 发生除了TLB缺失以外的异常，就会自动跳转到地址 0x80000080 处**，开始执行异常分发程序。

*Step*1 执行异常分发程序 except_vec3

.text.exc_vec3 段需要被链接器放到特定的位置，在 R3000 中这一段是要求放到地址 0x80000080 处，这个地址处存放的是异常处理程序的入口地址。一旦 CPU 发生异常，就会自动跳转到地址 0x80000080 处，开始执行。因此，我们在lab3中需要在 tools/scse0_3.lds 中的开头位置增加如下代码，即将.text.exc_vec3 放到 0x80000080 处, 来为操作系统增加异常分发功能。

```
. = 0x80000080;
.except_vec3 : {
    *(.text.exc_vec3)
}
```

异常分发程序的具体代码位于start.S的开头。

概括来说，这个函数的作用是：

1. 将Cause寄存器中的值取出，存到 k1 寄存器
2. 将 exception_handlers 这个数组的首地址存到了 k0 寄存器
3. 将 k1 中 Cause寄存器的值 和0x7c做按位与，使得 k1 中仅保留Cause寄存器中的 ExcCode段 乘4的值

> 因为ExcCode是 Cause寄存器 的2-6位，当按位与后，低2位置0，就相当于ExcCode表示的异常号左移了两位，即乘以4；而异常向量组中每一个成员的大小都是4字节，这样就相当于直接计算出了对于成员相对于数组基地址的偏移量。

4. addu k0,k1　这条指令做偏移, 使得 k0 的值为 exception_handlers 数组的第ExcCode项
5. jr k0 跳转到该中断处理子函数

```
.section .text.exc_vec3
NESTED(except_vec3, 0, sp)
    .set noat
    .set noreorder
1:
    mfc0 k1,CP0_CAUSE           # 将Cause寄存器中的值取出，存到 k1 寄存器
    la k0,exception_handlers    # la 为扩展指令，意为load address。
                                # 该指令将 exception_handlers 这个数组的首地址存到了
k0 寄存器。
    andi k1,0x7c                # 此处和0x7c做按位与，使得 k1 中仅保留Cause寄存器中的
ExcCode段                                   # 0x7c=01111100，说明 k1 中存的是ExcCode
乘4的值
    addu k0,k1
    # 在前面的语句中，k0 寄存器保存了 exception_handlers 的首地址，k1 保存了ExcCode乘4
的值，
    # 此处做偏移，使得 k0 的值为 exception_handlers 数组的第ExcCode项
    lw k0,(k0)
    # 将此时 k0 指向的数组项取出，仍存到 k0 中去。这里 lw k0 (k0) 就是 lw k0 (k0)
    # 此时，k0 的值就是数组第ExcCode项的值了，也就是异常码ExcCode对应的中断异常处理子函数的
入口地址.
    nop
    jr k0                       # 跳转到该中断处理子函数
    nop
END(except_vec3)
.set at
```

值得注意的是, 这里完全使用 `k0` ，`k1` 两个内核保留的寄存器，**可以保证用户态下其他通用寄存器的值不被改变**，也就保持了现场不变。

这里再回想起TF开头的宏中（即TrapFrame相关），有这样的注释

```
/*
 * $26 (k0) and $27 (k1) not saved
 */
```

联系这里，就不难理解上面这句注释的含义了。

> 此外，这里涉及到了两个 `.set` 指令：
>
> `noat` ：意味着接下来的代码中**不允许汇编器使用 `at` 寄存器(即 1 号寄存器)**。这是因为此时刚刚陷入内核，还未保存现场，用户态下除了 `k0` ，`k1` 之外都不能够被改变。
>
> `noreorder` : 意味着接下来的代码中不允许汇编器重排指令顺序。

*Step*1.5 通过 except_vec3异常向量组，进入异常处理子函数

异常分发程序通过 exception_handlers 数组定位中断处理程序，而 exception_handlers 就称作异常向量组。

lib/traps.c 中的 trap_init() 函数说明了异常向量组里存放了什么。

trap_init() 和 set_except_vector()

```
extern void handle_int();
extern void handle_reserved();
extern void handle_tlb();
extern void handle_sys();
extern void handle_mod();
unsigned long exception_handlers[32];
void trap_init()
{
    int i;
    for (i = 0; i < 32; i++) {
        set_except_vector(i, handle_reserved);
    }
    set_except_vector(0, handle_int);
    set_except_vector(1, handle_mod);
    set_except_vector(2, handle_tlb);
    set_except_vector(3, handle_tlb);
    set_except_vector(8, handle_sys);
}
// 向异常向量组 exception_handlers 中数组下标为 n 的地方存入异常处理函数地址 addr
void *set_except_vector(int n, void *addr)
{
    unsigned long handler = (unsigned long)addr;
    unsigned long old_handler = exception_handlers[n];
    exception_handlers[n] = handler;
    return (void *)old_handler;
}
```

实际上，trap_init()函数实现了对全局变量 exception_handlers[32] 数组初始化的工作，即通过把相应处理函数的地址填到对应数组项中，初始化了如下异常：

0 号异常的处理函数为 `handle_int`，表示中断，由时钟中断、控制台中断等中断造成

1 号异常的处理函数为 `handle_mod`，表示存储异常，进行存储操作时该页被标记为只读

2 号异常的处理函数为 `handle_tlb`，TLB 异常，TLB 中没有和程序地址匹配的有效入口

3 号异常的处理函数为 `handle_tlb`，TLB 异常，TLB 失效，且未处于异常模式（用于提高处理效率）

8 号异常的处理函数为 `handle_sys`，系统调用，陷入内核，执行了 syscall 指令

注意：异常不等于中断，x号中断和x号异常不一样。

那么这里，页写入异常，对应着1号异常。现在，我们由异常分发程序，其实就已经进入了异常向量组中注册过的的异常处理子函数。

Step2 执行对应异常处理子函数handle_mod

handle_mod函数实际上是由BUILD_HANDLER汇编宏函数构造出来的。

BUILD_HANDLER

```
BUILD_HANDLER mod   page_fault_handler cli

.macro  __build_clear_sti
    STI
.endm

.macro  __build_clear_cli
    CLI
.endm
```

```
.macro  BUILD_HANDLER exception handler clear
    .align  5
    NESTED(handle_\exception, TF_SIZE, sp)
    .set    noat
nop
    SAVE_ALL                    # 保存现场到异常处理栈
    __build_clear_\clear        // 对于写入页异常来说，这里是允许用户态下使用cp0寄存器，
且禁用中断
    .set    at
    move    a0, sp      # 将压栈以后的异常处理栈存入a0
    // 在页写入异常中，这就是KERNEL_SP - struct Trapframe
    #（4号中断是TIMESTACK - struct Trapframe处，其他异常则是KERNEL_SP - struct
Trapframe处）
    // 此时 a0 作为下面的 \handle（即 page_fault_handler ）对应的函数 的 参数

    jal \handler        # 在ra处存入这里下一部分的地址，跳转到 \handle （即
page_fault_handler ）参数对应的函数的位置
    nop                 # tlb异常的时候执行 do_refill，存储异常的时候调用
page_fault_handler
    j   ret_from_exception      # 调转到 ret_from_exception 恢复现场与回滚
    nop
    END(handle_\exception)
.endm
```

保存现场到异常处理栈

这里就是调用SAVE_ALL函数，将发生写入页异常时的环境存储到 异常处理栈 `KERNEL_SP - TF_SIZE` 往上一个结构体的空间。具体来说是：

1.将用户栈指针存入 k0 寄存器

2.调用get_sp 将sp置为异常处理栈指针，并将处理栈指针下压一个 struct Trapframe 的大小

3.将 k0 寄存器中存储的用户栈指针存入 `TF_REG29(sp)`

4.将用户态下的 v0 寄存器存入 `TF_REG2(sp)`

5.将CP0寄存器 和 其他通用寄存器(0-31，除了2和29)存入异常处理栈下压出来的栈空间

sti cli

```
#define STATUS_CU0 0x10000000
// 表示允许用户态下使用cp0寄存器，且 启用中断
.macro STI
    mfc0    t0, CP0_STATUS
    li      t1, (STATUS_CU0 | 0x1)  # 0x1000 0001 28位0位为1
    or      t0, t1
    mtc0    t0, CP0_STATUS

.endm
// 表示允许用户态下使用cp0寄存器，且 禁用中断
.macro CLI
    mfc0    t0, CP0_STATUS
    li      t1, (STATUS_CU0 | 0x1)
    or      t0, t1
    xor     t0, 0x1
    mtc0    t0, CP0_STATUS
.endm
```

page_fault_handler

具体代码解析上面说过了，这里作概括。

传入的参数 `struct Trapframe *tf` 实际上就是 `KERNEL_SP - struct Trapframe` 。

1.将当前环境中用户栈指针 `tf->regs[29]` 设置为进程异常处理栈区域 [curenv->env_xstacktop - BY2PG, curenv->env_xstacktop - 1]。而在函数 `set_pgfault_handler` 中，env_xstacktop 域被设置为了 UXSTACKTOP，所以这里就是把用户栈指针设置在了[UXSTACKTOP - BY2PG, UXSTACKTOP - 1]的区域。

2.将 `KERNEL_SP - struct Trapframe` 存放的进程上下文环境复制进入**设置以后的的用户栈指针** `tf->regs[29]` 指向的区域，因此这里：

如果说记 `tf->regs[29]` 存储的用户栈指针为 `user_sp` ，那么 `TF_EPC(user_sp)` 才是发生页写入异常时的PC值，这点很重要！！！因为下面会改变 `KERNEL_SP - struct Trapframe` 区域中的EPC，用于开始执行__asm_pgfault_handler。

2.设置当前环境中的epc寄存器为 `env_pgfault_handler` 域的值，也就是__asm_pgfault_handler的地址。（ `tf->cp0_epc = curenv->env_pgfault_handler` ）

由此可见，其实内核态下，页写入异常的处理函数只是设置了用户态栈指针位于异常处理栈区域，并且设置了epc寄存器，存放异常处理函数__asm_pgfault_handler的地址。

恢复现场与回滚 RESTORE_SOME

恢复现场的主要逻辑由汇编宏 `RESTORE_SOME` (include/stackframe.h)支持。此宏将现场中除了 `sp` `k1` `k0` 寄存器之外的所有寄存器都恢复到CPU中：

```
.macro RESTORE_SOME
    .set mips1
    lw      v0, TF_STATUS(sp)
    mtc0    v0, CP0_STATUS
    lw      v1, TF_LO(sp)
    mtlo    v1
    lw      v0, TF_HI(sp)
    mtlo    v0
    lw      v1, TF_EPC(sp)
    mtc0    v1, CP0_EPC
    lw      $31, TF_REG31(sp)
    lw      $30, TF_REG30(sp)
    //lw      $29, TF_REG29(sp) <~~ $29 <=> sp
    lw      $28, TF_REG28(sp)
    //lw      $27, TF_REG27(sp) <~~ $27 <=> k1
    //lw      $26, TF_REG26(sp) <~~ $26 <=> k0
    lw      $25, TF_REG25(sp)
    /* $24 - $2 都有，这里就不写了*/
    lw      $1, TF_REF1(sp)
.endm
```

此处并未恢复 `k0/k1` 这两个内核态寄存器, 这是**由于在 MIPS 规范下, 用户程序不会使用这两个寄存器. 而用户程序的编写者也应该遵循这个规范.**

- 若需要直接编写汇编代码, 应当时刻遵循此规范.
- 若通过编译器得到汇编代码, 则应当选择符合规范的编译器, 如: mips-4KC.

恢复现场与回滚 ret_from_exception

通过 `RESTORE_SOME` ，可以使得用户现场中，除了 `sp` 栈指针寄存器外的信息都被恢复，而 `sp` 寄存器将在回滚现场前的最后关头才恢复。**MOS中定义了汇编函数 `ret_from_exception` (lib/genex.S)用于恢复现场并回滚用户态。**

在页写入异常中，这里是把**内核态指针 `KERNEL_SP - TF_SIZE` 记录的环境**中的epc中存放的 `__asm_pgfault_handler` 地址存入了k0寄存器，并由 `jr  k0` 跳转到这个函数的入口，并执行 rfe 回到用户态（SR寄存器低六位的二重栈压栈）

新申请的进程的cp0_status被写入0001 0000 0000 0000 0001 0000 0000 1100，在出栈后，最后两位 KUc，IEc 为 [1,1]，表示CPU目前**在用户态下运行，并且开启了中断**。

至此，sp寄存器恢复成了用户态指针，这个值位于用户态的异常处理区域[UXSTACKTOP - BY2PG, UXSTACKTOP - 1]，并且指向的环境结构体就是发生页写入异常时的上下文环境。

存在k0中的，是内核态指针 `KERNEL_SP` 指向的epc位置，这里存放的是 __asm_pgfault_handler；

而在恢复sp为用户栈指针后，sp指向的epc位置，是存放着发生页写入异常的pc值。是处理完页写入异常以后函数真正应该跳转回去的值。

```
FEXPORT(ret_from_exception)
    .set noat
    .set noreorder
    RESTORE_SOME
    .set at
    lw  k0,TF_EPC(sp)           # 将异常处理栈中EPC对应的位置存入k0，
    // 那么这里就是相当于把epc中存放的__asm_pgfault_handler存入了k0
    # 此时sp的值依然是0x82000000-TF_SIZE (TF_SIZE 为 sizeof(struct Trapframe)
    lw  sp,TF_REG29(sp) /* Deallocate stack */
    # 在SAVE_ALL中，将用户栈指针由sp 存入 k0 再存入 Trapframe 中的 TF_REG29 处，这里就是将sp重置为用户栈指针
//1:    j   1b
    nop
    jr  k0                      # 异常处理完毕，跳转到中断时的位置继续执行
    rfe
```

Step3 在用户态下完成页写入异常的处理

在回到用户态以后，系统现在位于__asm_pgfault_handler函数的入口。

__asm_pgfault_handler

从内核返回后，此时的栈指针是由内核设置的，处于异常处理栈中，且指向一个由内核复制好的 Trapframe 结构体的底部。

`__asm_pgfault_handler` 函数通过宏定义的偏移量 TF_BADVADDR，用 lw 指令读取了 Trapframe 中的 cp0_badvaddr 字段的值，这个值 也正是 CPU 设置的发生页写入异常的地址。

该函数将这个地址作为参数去调用了 __pgfault_handler 这个变量内存储的函数指针，其指向的函数就是**“真正”进行处理的函数pgfault()**（这个赋值过程发生在上面的 `set_pgfault_handler()` 函数）。

```
lw  a0, TF_BADVADDR(sp)     # 读取了 Trapframe 中的 cp0_badvaddr 字段的值，这个值 也正是 CPU 设置的发生页写入异常的地址，存入a0 作为pgfault参数
lw  t1, __pgfault_handler
jalr    t1
```

随后就是一段用于恢复现场的汇编，最后使用 MIPS 的延时槽特性，在跳转的同时恢复了正常的栈指针。

由于**设置在[UXSTACKTOP - BY2PG, UXSTACKTOP - 1]区域以后**的用户态指针指向的epc值是发生页写入异常时真正的epc值，所以这里的跳转才没有问题，由于延迟槽特性，`lw sp,TF_REG29(sp)` 会在跳转发生前指向，而被设置在区域的用户态指针sp的 `TF_REG29(sp)` 中存放的，才是 **发生页写入异常的用户栈指针。因此需要再次还原。**

```
lw  k0,TF_EPC(sp)   //atomic operation needed
jr  k0
lw  sp,TF_REG29(sp)  /* Deallocate stack */
```

pgfault() 在用户态中真正处理写入页异常

`pgfault` 需要完成这些任务：

1. 判断参数虚拟地址va是否为 COW 的页面，是则进行下一步，否则报错
2. 分配一个新的临时物理页到临时位置，将要复制的内容拷贝到刚刚分配的页中（临时页面位置可以自定义，观察mmu.h的地址分配查看哪个地址没有被用到，思考这个临时位置可以定在哪）

   这里的临时位置就是USTACKTOP地址往上的一页，在mmu.h中是Invalid memory。

   注意是USTACKTOP不是UXSTACKTOP！
3. 将发生页写入异常的地址映射到临时页面上，注意设定好对应的页面权限位，然后解除临时位置的内存映射