第三章作业

钟赟
2016K8009915009

1. 简述 MIPS 与 X86 在异常处理过程中的区别。
   a) 异常处理准备过程不同
      MIPS 中，被异常打断的指令 EPTR 存储在 EPC 寄存器，而在 X86 中，EPTR 用栈存放 CS 和 EIP 组合。
   b) 确定异常来源的过程不同
      X86 由硬件进行异常和中断号查询，根据编号查询预设好的中断描述符表（IDT），得到不同异常处理的入口地址，并将 CS/EIP 等压栈。
      MIPS 将异常相关状态存于 Cause 寄存器，由异常处理程序做进一步的查询和区别处理，但对存储管理使用的地址转换异常等频繁发生的异常设置了专用的异常处理入口地址。
   c) 恢复执行状态并返回过程不同
      MIPS 中，异常返回的指令是 ERET，可同时清除特权等级并返回 EPC 保存的地址继续执行。而在 X86 中利用的是 IRET 指令。

2. MIPS 的 LL bit 在发生异常后会怎样处理，为什么？
   LL 指令从内存载入一个字后，将处理器内部的链接状态位 LL bit 置 1，将链接载入地址保存到 LLAddr 中。如果 LL 和 SC 指令之间产生异常，则处理异常，处理器将 LL bit 置 0。
   原因：当运行到 SC 指令时，从 LL 指令的 RMW（Read-Modify-Write）序列进行检查，判断 LL bit 的值，如果为 0 表示发生了异常，则不进行写回操作，并设置一个通用寄存器，将其值置为 0，表示失败。

3. 简述精确异常与非精确的区别。找一个使用非精确异常的处理器。
   a) 非精确异常
      在多发射乱序执行的流水线处理器上，从指令进入流水线到异常事件的发生，期间经过若干流水级，此时 PC 的值已指向其后的某条指令，实现非精确异常的处理器就把此时的PC值作为引起异常指令的所在,即EPTR的指向并非真正引起异常的指令，而是其后面的某条指令。非精确异常就是不需要返回的。
   b) 精确异常
      实现精确异常的处理器在最后指令提交时按指令流的顺序提交，可以精确计算出引起异常的指令相对于当前 PC 的偏移，从而保证精确异常，即 EPTR 指向是真正引起异常的指令。精确异常需要保存返回地址到 EPC。
   c) 非精确异常的处理器示例
      MIPS 74K Core

4. 在一台 MIPS-linux 机器（页大小为 4KB）上执行下列程序片段，会发生多少次异常？说明其过程。

```
void cycle(double *a){
        int i;
        double b[65536];
        for(i=0;i<3;i++)
                memcpy(a,b,sizeof(b));
}
```

For 循环中 memcpy 函数的功能为：以 b 所指向的内存地址位起始地址，拷贝 sizeof(b)个字节到 a 中，共需要拷贝 65535*8B = 512KB，即 128 页。
在进行拷贝的过程中，按以下顺序发生和处理异常：
用户程序尝试读取 b 所在内存地址的数据，在 TLB 中查找失败，触发 TLB refill 异常；
TLB refill 试图读取页表中的相应内容，写入 TLB；
发现页表未初始化，TLB refill 返回到应用程序，访问 TLB；
TLB 页表无效，发生 TLB Invalid 例外；
操作系统查找 b，判断其状态位可读，为其分配物理空间，将物理地址写入页表，更新 TLB，例外返回。
读操作重试，成功。
用户程序尝试在 a 所在内存地址写数据，在 TLB 中查找失败，触发 TLB refill 异常；
TLB refill 试图读取页表中的相应内容，写入 TLB；
发现页表未初始化，TLB refill 返回到应用程序，访问 TLB；
TLB 页表无效，发生 TLB Invalid 例外；
操作系统查找 b，判断其状态位可读，为其分配物理空间，将物理地址写入页表，更新 TLB，例外返回。
写操作重试，成功。
TLB 读写 2*4KB 的数据之后，再次发生 b 和 a 的 TLB refill 例外。之后不再发生异常。

综上，一共发生了$\frac{512KB}{2*4KB} * 2 * 3 + 2 = 386$次异常。

5. 用 C 语言描述包含 TLB 的页式存储管理过程（包含 TLB 操作）。
   代码如下：

```
//TLB struct
typedef struct TLB
{
    unsigned int VPN;
    unsigned int PFN_1;
    unsigned int PEN_2;
    unsigned int G;
    unsigned int ASID;
    unsigned int V_1;
    unsigned int V_2;
    unsigned int mask;
    unsigned int C_1;
    unsigned int C_2;
    unsigned int D_1;
    unsigned int D_2;
}TLB;

//virtual address
typedef struct
{
```

```
    int ASID;
    int VPN;
    int offset;
}Virtual_addr;

//search tlb table in virtual address
TLB search(virtual_addr, TLB tlb[]);

//TLB Refill
void tlb_refill(Virtual_addr vaddr,TLB tlb[]);

//TLB Invalid
void tlb_invalid(Virtual_addr vaddr,TLB tlb_i);

//get physical address
phy_addr get_addr(Virtual_addr vaddr,TLB tlb_i);

TLB tlb[128];

void main(){

    TLB tlb_i;
    Virtual_addr vaddr;
    phy_address paddr;
    while(TRUE){
        if(!(tlb_i=search(vaddr,tlb))){
            tlb_refill(vaddr,tlb);
        }
        else if(!tlb_i.V_1){
            tlb_invalid(vaddr,tlb_i);
        }
        else{
            paddr = get_addr(vaddr,tlb_i);
            break;
        }
    }
    return 0;
}

TLB search(Virtual_addr vaddr,TLB tlb[]){
    for(i=0;i<128;i++){
        if(tlb[i].ASID == vaddr.ASID && tlb[i].VPN == vaddr.VPN)
            return tlb[i];
    }
```

```
    return NULL;
}

void tlb_refill(Virtual_addr vaddr,TLB tlb[]){
    for(i=0;i<128;i++){
        if(tlb[i]==NULL){
            tlb[i].ASID = vaddr.ASID;
            tlb[i].VPN  = vaddr.VPN;
            break;
        }
    }
}

void tlb_invalid(Virtual_addr vaddr,TLB tlb_i){
    if(check_vaddr(vaddr)){
        tlb_i.V_1 = 1;
    }
    else{
        return ERROR;
    }
}

int check_vaddr(Virtual_addr vaadr){
    return alloc(vaddr)?1:0;
}

phy_address get_addr(Virtual_addr vaddr,TLB tlb_i){
    return (int *)(tbl_hit.PFN_1<<5+v_addr.offset);
}
```