Homework 4

钟赟

2016K8009915009

1. **在观看网络视频时，网卡将接收到的网络包放到内存并通知处理器对视频数据流进行解码后发送给显示单元。此时，处理器对该网络包的中断处理是否应该分上下部分？若是，应该如何分？**

处理器对该网络包的中断处理应该分上下部分。

上半部分为硬中断，处理器读网卡寄存器，确定需要处理的时间，关闭网卡中断使能，避免大量网络包引起频繁中断，通知 NAPI 有新任务，但不立即执行；

下半部分为软中断，在开中断状态下运行，处理硬中断期间所设置的处理任务，即将接收到的网络包写到内存并对视频数据流进行解码后发送给显示单元。

2. **(1) 用 MIPS 汇编程序来举例并分析未同步的线程之间进行共享数据访问出错的情况。**

```
#假设现在又两个线程同时运行，它们的c代码如下：
#int i1 = 0, i2 = 0;
#void thread1(){
#    while(1)
#        i2 = ++ i1;
#}
#void thread2(){
#    while(1)
#        i2 = ++ i1;
#}
#变量i即为thread1和thread2的共享数据，未同步时两个线程对其的访问可能会出错
#开始时在thread1中：
lw      a0, i1      # i1 = 0
sw      a0, i2      # i2 = 0
#切换到thread2
lw      a0, i1      # i1 = 0
sw      a0, i2      # i2 = 0
addi    a0, a0, 0x1
sw      a0, i1      #i1 = 1
#切换到thread1
addi    a0, a0, 0x1
sw      a0, i1      #i1 = 2
#在thread1中，经过一次while循环，i1和i2的值本应为1，却由于未同步状态下thread2
#同时对i访问，使得结果为i1 = 2, i2 = 0，产生错误
```

(2) 用 LL/SC 指令改写你的程序，使它们共享数据正确。

```
#下面为利用LL/SC指令修改后的程序
#开始时在thread1中：
ll      a0, i1      # i1 = 0
sw      a0, i2      # i2 = 0
#切换到thread2
ll      a0, i1      # i1 = 0
sw      a0, i2      # i2 = 0
addi    a0, a0, 0x1
sc      a0, i1      #i1 = 1
#切换到thread1
addi    a0, a0, 0x1
sw      a0, i1      #i1 = 2
```

3. (1) 在你的机器上安装 MIPS 交叉编译器，通过编译-反汇编的方式提取函数调用的核心片段。

C 程序如下：

```c
#include<stdio.h>

//func: func(1) = 2, func(2) = 3, func(n) = func(n-1) * func(n-2)
int func(int n)
{
        int r;
        r = (n==1) ? 2 :
            (n==2) ? 3 :
            func(n-1) * func(n-2);
        return r;
}

int main()
{
        int a;
        scanf("%d", &a);
        printf("%d\n", func(a));
}
```

反汇编得到的 MIPS 汇编如下：

```
    .text
    .globl  func
    .type   func, @function
func:
    .set    noreorder
    addiu   $sp,$sp,-56
    sw  $31,52($sp)
    sw  $fp,48($sp)
    sw  $16,44($sp)
    move    $fp,$sp
    sw  $4,56($fp)
    lw  $3,56($fp)
    li  $2,1            # 0x1
    beq $3,$2,$L2
    nop
    lw  $3,56($fp)
    li  $2,2            # 0x2
    beq $3,$2,$L3
    nop
    lw  $2,56($fp)
    nop
    addiu   $2,$2,-1
    move    $4,$2
    .option pic0
    jal func
    nop
    .option pic2
    lw  $28,16($fp)
    move    $16,$2
    lw  $2,56($fp)
    nop
    addiu   $2,$2,-2
    move    $4,$2
    .option pic0
    jal func
    nop

    .option pic2
    lw  $28,16($fp)
    mult    $16,$2
    mflo    $3
    sw  $3,32($fp)
    b   $L4
    nop
$L3:
    li  $2,3            # 0x3
    sw  $2,32($fp)
$L4:
    lw  $3,32($fp)
    nop
    sw  $3,36($fp)
    b   $L5
    nop
$L2:
    li  $2,2            # 0x2
    sw  $2,36($fp)
$L5:
    lw  $3,36($fp)
    nop
    sw  $3,24($fp)
    lw  $2,24($fp)
    move    $sp,$fp
    lw  $31,52($sp)
    lw  $fp,48($sp)
    lw  $16,44($sp)
    addiu   $sp,$sp,56
    j   $31
    nop
    .end    func
    .rdata
$LC0:
    .ascii  "%d\000"
$LC1:
    .ascii  "%d\012\000"
    .text
    .globl  main
    .type   main, @function
main:
```

```
    addiu   $sp,$sp,-48
    sw  $31,44($sp)
    sw  $fp,40($sp)
    move    $fp,$sp
    lui $28,%hi(__gnu_local_gp)
    addiu   $28,$28,%lo(__gnu_local_gp)
    .cprestore  16
    lui $2,%hi($LC0)
    addiu   $4,$2,%lo($LC0)
    addiu   $2,$fp,24
    move    $5,$2
    lw  $25,%call16(scanf)($28)
    nop
    jalr    $25
    nop
    lw  $28,16($fp)
    lw  $2,24($fp)
    nop
    move    $4,$2
    jal func
    nop
    .option pic2
    lw  $28,16($fp)
    move    $3,$2
    lui $2,%hi($LC1)
    addiu   $4,$2,%lo($LC1)
    move    $5,$3
    lw  $25,%call16(printf)($28)
    nop
    jalr    $25
    nop
    lw  $28,16($fp)
    move    $sp,$fp
    lw  $31,44($sp)
    lw  $fp,40($sp)
    addiu   $sp,$sp,48
    j   $31
    nop
```

(2) 改变编译的优化选项，记录函数调用核心片段的变化，并分析不同优化选项的效果。

采用 02 优化，得到的反汇编如下：

```
    .text
    .globl  func
    .type   func, @function
func:
    addiu   $sp,$sp,-48
    sw  $31,44($sp)
    sw  $19,40($sp)
    sw  $18,36($sp)
    sw  $17,32($sp)
    sw  $16,28($sp)
    li  $2,1            # 0x1
    beq $4,$2,$L10
    move    $16,$4
    li  $2,2            # 0x2
    beq $4,$2,$L11
    li  $17,1           # 0x1
    li  $19,1           # 0x1
    b   $L5
    li  $18,2           # 0x2
$L6:
    beq $16,$18,$L12
    addu    $2,$2,$17
$L5:
    .option pic0
    jal func
    .option pic2
    addiu   $4,$16,-1
    mult    $17,$2
    addiu   $16,$16,-2
    lw  $28,16($sp)
    mflo    $17
    bne $16,$19,$L6
    sll $2,$17,1
$L3:
    lw  $31,44($sp)
    lw  $19,40($sp)
    lw  $18,36($sp)
    lw  $17,32($sp)
    lw  $16,28($sp)
    j   $31
    addiu   $sp,$sp,48

$L12:
    lw  $31,44($sp)
    lw  $19,40($sp)
    lw  $18,36($sp)
    lw  $17,32($sp)
    lw  $16,28($sp)
    j   $31
    addiu   $sp,$sp,48
$L10:
    b   $L3
    li  $2,2            # 0x2
$L11:
    b   $L3
    li  $2,3            # 0x3
    .end    func
$LC0:
    .ascii  "%d\000"
$LC1:
    .ascii  "%d\012\000"
    .text
    .globl  main
    .type   main, @function
```

```
main:
    lui $28,%hi(__gnu_local_gp)
    addiu   $sp,$sp,-40
    addiu   $28,$28,%lo(__gnu_local_gp)
    sw  $31,36($sp)
    .cprestore  16
    lw  $25,%call16(scanf)($28)
    lui $4,%hi($LC0)
    addiu   $5,$sp,24
    jalr    $25
    addiu   $4,$4,%lo($LC0)
    lw  $4,24($sp)
    .option pic0
    jal func
    nop
    .option pic2
    lw  $28,16($sp)
    lui $4,%hi($LC1)
    lw  $25,%call16(printf)($28)
    addiu   $4,$4,%lo($LC1)
    jalr    $25
    move    $5,$2
    lw  $31,36($sp)
    lw  $28,16($sp)
    j   $31
    addiu   $sp,$sp,40
    .end    main
```

可见，采用 O2 编译优化后，main 函数调用 func 函数的差别不是很大，但是在函数 func 递归调用的过程中，将 func(n-1)和 func(n-2)直接存入寄存器中，减少了计算 func(n)时对 func 的递归调用，大大节省了消耗。

4. **ABI 中会包含对结构体中各元素的对齐和摆放方式的定义。**

(1) **在你的机器上用 C 语言编写一段包含不同语言类型的结构体，并获得结构体总空间占用情况。**

C 程序如下：

```c
#include<stdio.h>

struct example1{
        int     i1;
        long double i2;
        short   i3;
        char    i4;
};

struct example2{
        long double i2;
        char    i4;
        int     i1;
        short   i3;
};

int main()
{
        printf("size of char: %d\n", sizeof(char));
        printf("size of short: %d\n", sizeof(short));
        printf("size of int: %d\n", sizeof(int));
        printf("size of double: %d\n", sizeof(double));
        printf("size of long double: %d\n", sizeof(long double));
        printf("size of struct1: %d\n", sizeof(struct example1));
        return 0;
}
```

程序执行情况如下：

```
stu@stu-VirtualBox:~/Desktop$ ./1
size of char: 1
size of short: 2
size of int: 4
size of double: 8
size of long double: 12
size of struct1: 20
```

(2) **调整结构体元素顺序，推测并分析结构体对齐的方式。**

调整后的结构体为 example2，程序执行情况如下：

```
stu@stu-VirtualBox:~/Desktop$ ./1
size of char: 1
size of short: 2
size of int: 4
size of double: 8
size of long double: 12
size of struct2: 24
```

结构体 exmple1 所占字节数为 4+12+4 = 20，其中 short 型变量占两字节，char 型占一字节，但是为了与 int 型变量的 4 字节对齐，必须凑够 4 字节，因此一共 20 字节。

结构体 exmple2 所占字节数为 12+12 = 24，其中 char, int, short 型变量加在一起不够 12 字节，但是必须与 long double 型变量的 12 字节对齐，所以共占用 24 字节。

因此结构体成员字节对齐的方式为：如果只有一个变量，结构体大小即为该变量大小；如果多于两个元素，则以结构体中所占字节数最大的变量的字节数为每一行所能盛放的最大字节数，然后每一个元素放在上一个元素之后，如果该行放不下，则另起一行。

5. **函数调用、系统调用和中断处理都需要上下文切换，请结合 MIPS O32 ABI 说明上述三种上下文切换过程时保留现场有什么不同（内容、位置）。**

函数调用

根据 MIPS O32 ABI 规定，在上下文切换时，调用者保存 s0~s7，ra，sp，fp 寄存器，被调用者保存 at， v0， v1， a0~a3, t0-t7, t8, t9, gp 寄存器，操作系统内核保存 k0, k1 寄存器。

系统调用

上下文切换时，调用者需要保存除 v0~v1, a0~a3, 之外的寄存器。

中断处理

上下文切换时，需要保存 32 个通用寄存器，CP0 寄存器的 status，cause 寄存器，被异常打断的指令寄存器 epc，以及高位低位寄存器 hi,lo。