Q1.处理器对网络包的中断处理是否应分为上下半部?

将中断处理分为上下半部的思路如下:

上半部的功能是响应中断。上半部是完全屏蔽中断的，其作用为处理发生的中断，分配中断处理程序，控制 PC 跳转到服务程序入口地址。当中断发生时，它就把设备驱动程序中中断处理例程的下半部挂到设备的下半部执行队列中去，然后继续等待新的中断到来。

下半部的功能是中断服务。下半部所负责的工作一般是查看设备以获得产生中断的事件信息，并根据这些信息（一般通过读设备上的寄存器得来）进行相应的处理。其最大的特点是可中断。

这样分配的好处是将中断信息分析和中断处理分开。一个中断到来时，需要第一时间分析中断的信息（如优先级等），这样将分析这部分信息的部分确定为上半部，使其按中断顺序执行信息分析，可以保证中断信息分析速度。随后具体的中断服务由下半部进行。下半部在进行的时候同时也可以由其他的高优先级中断打断，这样分配既可以保证中断处理速度，又能保证优先级的实现。

对于处理网络包，有如下划分:

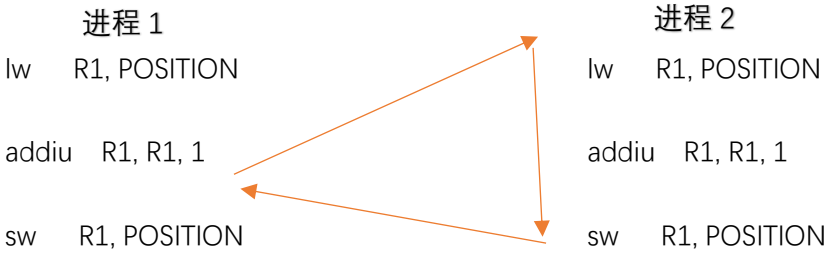
上半部: 网卡发出中断，PC 跳转至 0x80000180（通用例外入口），取出中断型，直接跳转到外部中断入口。外部中断程序保存进程上下文，关 IE,清 EXL，进入内核态，设置返回函数，跳到平台相关中断分发入口。中断分发读中断状态，得到中断号，调用 do_IRQ，随后退出硬中断。转入下半部。

下半部: 网卡的中断处理。读网卡寄存器，确认事件，关闭网卡中断使能，解码网卡视频流，发送给显示单元。恢复上下文，退出内核态。

Q2:

（1） 未同步的进程共享数据访问出错:

POSITION 处存放的值为 0




在这个例子中，起初 POSITION 的值为 0，在进程 1 中被取出（0），addiu 后为（1），在将要写回 POSITION 时被中断，切换到进程 2；进程 2 从 POSITION 中取值依旧为 0，+1，写回，此时 POSITION 为 1。进程 2 执行结束后，回到进程 1，虽然恢复了上下文，但是写入 POSITION 的值仍为 1。这样，原本应该为 2 的 POSITION 现在却是 1，发生错误。

（2） 用 LL 和 SC 修改程序。

```
ll      R1, POSITION
addiu   R1, R1, 1
sc      R1, POSITION
```

Q3.

（1） 函数调用的核心片段

mips 编译-反汇编：mipsel-linux-gcc test.c -S -o test.S

test.c

```
#include<stdio.h>

void hello()
{
    printf("Hello world");
}
int main()
{
    hello();
    return 0;
}
```

test.S

```
    .file  1 "test.c"
    .section .mdebug.abi32
    .previous
    .gnu_attribute 4, 1
    .abicalls
    .rdata
    .align    2
$LC0:
    .ascii    "Hello world\000"
    .text
    .align    2
    .globl    hello
    .ent hello
    .type    hello, @function
hello:
    .set  nomips16
    .frame    $fp,32,$31        # vars= 0,
regs= 2/0, args= 16, gp= 8
    .mask    0xc0000000,-4
    .fmask    0x00000000,0
    .set  noreorder
    .set  nomacro

    addiu    $sp,$sp,-32
    sw   $31,28($sp)
    sw   $fp,24($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)
    lw   $25,%call16(printf)($28)
    nop
    jalr  $25
    nop

    lw   $28,16($fp)
    move    $sp,$fp
    lw   $31,28($sp)
    lw   $fp,24($sp)
    addiu    $sp,$sp,32
    j    $31
    nop

    .set  macro
    .set  reorder
    .end hello
    .align    2
    .globl    main
    .ent main
    .type    main, @function
main:
    .set  nomips16
    .frame    $fp,32,$31        # vars= 0,
regs= 2/0, args= 16, gp= 8
    .mask    0xc0000000,-4
    .fmask    0x00000000,0
    .set  noreorder
    .set  nomacro

    addiu    $sp,$sp,-32
    sw   $31,28($sp)
    sw   $fp,24($sp)
    move    $fp,$sp
    .cprestore    16
    .option   pic0
    jal   hello
```

```
    nop                                         addiu   $sp,$sp,32
                                                j   $31
    .option pic2                                nop
    lw  $28,16($fp)
    move    $2,$0                               .set macro
    move    $sp,$fp                             .set reorder
    lw  $31,28($sp)                             .end main
    lw  $fp,24($sp)                             .ident  "GCC: (GNU) 4.3.0"
```

代码中加灰部分为函数调用的部分。首先 sp-32，分配子函数栈空间。随后 sw 保存返回地址和栈指针。然后 jal，跳转到子函数部分。

（2） 优化选项的影响

-O1 无变化，其目的为使代码尽可能变快

-O2 -O3 将简单的子函数直接内联到 main 函数中，代替了调用

```
    lui     $28,%hi(__gnu_local_gp)
    addiu$sp,$sp,-32
    addiu$28,$28,%lo(__gnu_local_gp)
     sw     $31,28($sp)
    .cprestore 16
    lw      $25,%call16(printf)($28)
    lui     $4,%hi($LC0)
    jalr    $25
    addiu$4,$4,%lo($LC0)
```

Q4. 分析结构体的对齐方式

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

struct A
{char a;
 int b;};
struct B
{char a;
 int b;
 double c;};
struct C
{char a;
 double c;
 int b;};
void main()
{
    printf("%d\n",sizeof(A));
    printf("%d\n",sizeof(B));
    printf("%d\n",sizeof(C));

}
```

```
C:\WINDOWS\syste
8
16
24
请按任意键继续. . .
```

char 1

int 4

double 8

对齐方式应该是先找到 struct 中长度最大的格式（A 中是 int，B 和 C 中为 double）作为对其的基准（A 的大小就为 4*2，B 的大小为 8*2， C 的大小为 8*3）。对齐方式从结构体的第一个组分开始，以一个基准长度为准，如果放得下就将其放入（如 B 中的 char a 和 int b 就放在了同一个块中，占用了 8 个字节），如果放不下就新增加一个基准长度（如 C 中 double c 不能与 char a 放入同一个块，就新增加了一个；int b 不能与 double c 放在一起，就又新增加了一个）。

Q5.

表 4.1 MIPS 通用寄存器约定

| 寄存器编号 | O32 助记符 | N32 助记符 | 使用约定 |
|---|---|---|---|
| 0 | zero | | 总是返回 0 |
| 1 | at | | 汇编暂存寄存器 |
| 2 ~ 3 | v0，v1 | | 子程序返回值 |
| 4 ~ 7 | a0 ~ a3 | | 子程序的前几个参数 |
| 8 ~ 11 | t0 ~ t3 | a4 ~ a7 | N32 作为参数，O32 作为不需保存的暂存器 |
| 12 ~ 15 | t4 ~ t7 | t0 ~ t3 | 不需保存的暂存器，但 N32 和 O32 命名不同 |
| 16 ~ 23 | s0 ~ s7 | | 寄存器变量，过程调用时需要存储和恢复 |
| 24 ~ 25 | t8，t9 | | 暂存器 |
| 26 ~ 27 | k0，k1 | | 为异常处理保留 |
| 28 | gp | | 全局指针 |
| 29 | sp | | 栈指针 |
| 30 | s8/fp | | 寄存器变量，或作为帧指针 |
| 31 | ra | | 子程序返回地址 |

函数调用：是预料范围内的代码执行，当前执行的函数调用另外一个函数时，是从当前代码段通过跳转指令主动跳转到另外一个代码段，只需保存跳转之前的栈顶指针(fp)，栈底指针(sp)到栈空间，保存跳转指令的下一条函数的地址到 ra 寄存器，保存 s0-s7 寄存器到内存或者栈空间，其他寄存器无须保存。使用寄存器 r0-r3 传递函数参数，多的参数使用栈空间传递函数参数。实际上，函数调用的保存不能说保存现场，它只完成了一部分的工作。

中断处理：比较严重的任务切换。由于将要执行程序的一切均未知，需要保存除了 k0，k1 之外的所有寄存器。保存的位置保存到该线程对应的 PCB 块下。另外，还需要保存 SR,EPC 和 CAUSE 等协寄存器的值。

系统调用：是中断处理的一种。有趣的是，系统调用过程中，触发系统调用的操作是中断，故对被中断打断的用户态而言，保存现场与中断处理中的保存现场相同；在中断处理内部，由于需要调用系统调用程序，其现场的保存与函数调用更为类似。