考试科目名称　计算机系统基础　（A 卷）

2016—2017 学年第 1 学期　教师 袁春风 路通 苏丰 唐杰 汪亮　　考试方式：开卷

系（专业）　计算机科学与技术　　年级　2015　　班级＿＿＿＿

学号＿＿＿＿＿＿＿＿＿＿　　姓名＿＿＿＿＿＿　　成绩＿＿＿＿＿

题号	一	二	三	四	五	六	七	八	九	十	十一	十二	十三
分数													

某生写了一个C语言程序，用于计算$f(n)=2^n+2^{n-1}+\cdots+2^0=11\cdots1B$（n+1个1）。该程序有两个源文件：**main.c**和**test.c**，它们的内容如下图所示。

```
/* main.c */
1   #include <stdio.h>
2   typedef int ret_type;
3   extern ret_type i_max( unsigned int );
4   int inval;
5
6     void main( ) {
7          ret_type result;
8          scanf( "%d", &inval );
9          result = i_max( inval );
10         printf( "Result = %d\n", result );
11    }
```

```
/* test.c */
1   typedef int ret_type;
2   ret_type i_max(unsigned inval) {
3        ret_type sum, tmp;
4        unsigned i;
5        sum = 1;   tmp = 1;
6        for( i=0; i<=inval-1; i++ ) {
7             tmp = tmp* 2;
8             sum = sum + tmp;
9        }
10  return sum;
11  }
```

假设在**IA-32/Linux**平台上用GCC编译驱动程序处理，**main.c**和**test.c**的可重定位目标文件名分别是**main.o**和**test.o**，生成的可执行文件名为**test**。使用**“objdump –d test”**得到反汇编后的部分结果如下。

```
0804844b <i_max>:
 804844b:    55                    push   %ebp
 804844c:    89 e5                 mov    %esp,%ebp
 804844e:    83 ec 10              sub    $0x10,%esp
 8048451:    c7 45 fc 01 00 00 00  movl   $0x1,-0x4(%ebp)
 8048458:    c7 45 f8 01 00 00 00  movl   $0x1,-0x8(%ebp)
 804845f:    c7 45 f4 00 00 00 00  movl   $0x0,-0xc(%ebp)
 8048466:    eb 0d                 jmp    8048475 <i_max+0x2a>
 8048468:    d1 65 f8              shll   -0x8(%ebp)
 804846b:    8b 45 f8              mov    -0x8(%ebp),%eax
 804846e:    01 45 fc              add    %eax,-0x4(%ebp)
 8048471:    83 45 f4 01           addl   $0x1,-0xc(%ebp)
 8048475:    8b 45 08              mov    0x8(%ebp),%eax
 8048478:    83 e8 01              sub    $0x1,%eax
 804847b:    3b 45 f4              cmp    -0xc(%ebp),%eax
 804847e:    73 e8                 jae    8048468 <i_max+0x1d>
 8048480:    8b 45 fc              mov    -0x4(%ebp),%eax
 8048483:    c9                    leave
 8048484:    c3                    ret

08048485 <main>:
 8048485:    8d 4c 24 04           lea    0x4(%esp),%ecx
 .....    ................        ........................
```

```
8048499:    68 b0 97 04 08        push   $0x80497b0
804849e:    68 70 85 04 08        push   $0x8048570
80484a3:    e8 98 fe ff ff        call   8048340 <__isoc99_scanf@plt>
80484a8:    83 c4 10              add    $0x10,%esp
80484ab:    a1 b0 97 04 08        mov    0x80497b0,%eax
80484b0:    83 ec 0c              sub    $0xc,%esp
80484b3:    50                    push   %eax
80484b4:    e8 92 ff ff ff        call   804844b <i_max>
.....    ................        ........................
```

回答下列问题或完成下列任务。

（提示：IA-32为小端方式，按字节编址，字长为32位，即sizeof(int)=4，页大小为4KB，虚拟地址空间中的只读数据和代码段、可读写数据段都按4KB边界对齐）

一、执行程序时，该生在键盘上输入0后程序一直没有反应，为什么？此时，若在键盘上按下“Ctrl+C”即可让程序退出执行，这种导致程序终止的事件属于内部异常还是外部中断？从键盘上按下“Ctrl+C”到程序终止过程中计算机系统进行了哪些处理？（要求用100~200字简要描述处理过程）（8分）

答：

因为i_max函数的入口参数为0时，inval-1=-1，其机器数为11…1，因此，for循环的终止条件“i>inval-1”永远不会满足，for循环为死循环，因而程序没有反应。（2分）

用按键Ctrl+C终止程序属于外部中断。（1分）

中断过程（略）（5分）

二、若把test.c中所有unsigned改为int，则i_max函数的反汇编结果中哪条指令会发生改变？应改变成什么指令？（2分）

答：应该将jae改成jge。（2分）

三、在键盘上输入31时程序执行结果为Result=-1，为什么？若输入32，则程序执行结果是什么？（4分）

答：因为inval=31，函数i_max返回的机器数为32个1，按照%d格式解释，为带符号整数-1。（2分）

inval=32时，程序执行结果还是-1。（2分）

四、为了在更大输入值时也能正确执行，该生把程序中的ret_type改成float类型，i_max函数名改为f_max，

printf函数中%d改成%f。结果发现当键盘输入值超过23时，i_max的返回值总是比f_max的返回值小1，例如，输入24时，前者为33554431，后者为33554432.000000。为什么？当输入为127时，函数f_max的返回值为inf，为什么？此时返回结果的机器数是什么？（提示：inf为无穷大）（6分）

答：inval=24时，对应的结果应该为25个1，而float型有效数字只有23+1=24位，故函数f_max返回的值有舍入。舍入位为1，舍入后尾数加1，数值变大。（2分）

inval=127时，对应的精确结果应该为128个1，即$1.11\cdots1\times2^{127}$。但程序在计算过程中会发生舍入，用float型格式表示时，23位尾数后面的1舍入后会有进位，因而尾数变为10.00…0，进行右规操作，尾数右移，阶码加1，用float型格式表示：阶码为127+127+1=255，即阶码全1、尾数全0，结果为无穷大。（3分）

结果的机器数为0 1111 1111 000 0000 0000 0000 0000 0000B=7F80 0000H。（1分）

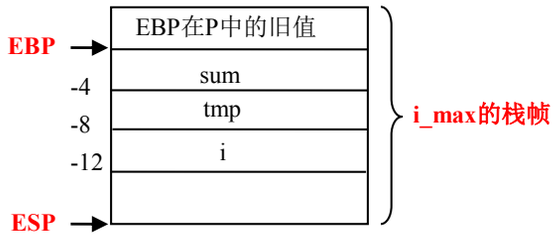
五、i_max函数中哪条指令实现了tmp=tmp*2的功能？对于第四题中提到的函数f_max，能否也不用乘法指令来实现tmp=tmp*2？为什么？（4分）

答：i_max的机器级代码中，shll指令是左移指令，左移一位相当于乘2。（2分）

对于f_max函数，可以用加法指令来实现乘2功能。（2分）

六、根据反汇编结果画出i_max函数（过程）的栈帧，要求分别用EBP和ESP标示出i_max函数的栈帧底部和顶部，并标出sum、tmp和i的位置。（6分）

答：（6分）



七、该进程的只读代码段和可读写数据段的起始地址各是什么？main.c和test.c中各定义了哪几个符号？这些符号分别在哪个段中定义？库函数scanf和printf中格式符（如"%d"）在哪个段中定义？（8分）

答：只读代码段和可读写数据段的起始地址分别是0x8048000和0x8049000。（2分）

main.c中定义了一个全局变量inval和一个函数main。（2分）

test.c中定义了一个函数i_max。（1分）

函数main和i_max定义在只读代码段内；inval定义在可读可写数据段内。（2分）

格式符（如"%d"）在只读代码段中定义。（1分）

八、填写下表中各标识符的情况，说明每个标识符是否出现在test.o的符号表（.symtab节）中，如果是的话，进一步说明定义该符号的模块是main.o还是test.o、该符号的类型是全局、外部还是本地符号、该符号出现在test.o中的哪个节（.text、.data或.bss）。（6分）

标识符	在test.o的符号表中？	定义模块	符号类型	节
i_max				
inval				
sum				

答：

标识符	在test.o的符号表中？	定义模块	符号类型	节
i_max	是	test.o	全局	text
inval	否	-	-	-
sum	否	-	-	-

九、地址0x8048480处的mov指令中，源操作数采用什么寻址方式？若R[ebp]=0xbfff f000，则源操作数的有效地址是多少？源操作数的访问过程需要经过哪些步骤？（要求从有效地址计算开始进行简要说明，包括何时判断及如何判断TLB缺失、缺页和cache缺失等，300字左右）（18分，若能结合题目中给出的具体例子清楚描述IA-32/Linux中的地址转换过程，则额外加10分）

答：采用“基址+位移量”寻址方式。（1分）

有效地址EA=R[ebp]-0x4 = 0xbfff f000-4 = 0xbfff effc（2分）

操作数的访问过程（略）。（15分）

十、已知页大小为4KB，主存地址位数为32位。假设代码Cache的数据区大小为32KB，采用8路组相联映射方式，主存块大小为64B，采用LRU替换算法和回写（Write Back）策略，则主存地址如何划分？代码Cache的总容量是多少？虚拟地址页内偏移（VPO）与Cache组索引（CI）和Cache块内偏移（CO）之间有什么关系？函数i_max的

代码将会映射到几个Cache组？各自映射到的Cache组号是什么？试分析执行i_max函数过程中可能发生几次缺失？（20分）

答：32位主存地址中，块内地址占$\log_2 64$=6位，代码Cache共有32KB/64B=512行，分成512/8=64组，因此组号（组索引）占6位，标记（Tag）字段占32-6-6=20位。（3分）

因为每组8路，故每行中LRU位占3位，还有1位修改位、1位有效位、20位标记、64B数据。总容量位数为512×(3+1+1+20+64×8)=274 944。（4分）

因为页大小为4KB，按字节编址，所以页内偏移量占12位。即物理地址和虚拟地址的低12位完全相同，因此，虚拟地址的低12位中，高6位为Cache组号，低6位为块内地址。（2分）

映射到两个Cache组。（1分）

i_max代码的虚拟地址中，804844b~804847f映射到同一个Cache组，8048480~8048484映射到同一组，前者对应组号为0100 01，后者的对应组号为0100 10。（2分）

执行i_max函数过程中可能会发生0次或1次cache缺失。（1分）

虚拟地址804844b~804847f范围内的指令中，可能第一条指令会缺失；若在执行i_max之前已经有同一个主存块的指令执行过，则不会发生缺失。因为8048480~8048484范围内的3条指令与main中的到80484b4处的call指令，都在同一个主存块中，对应的cache组号为18，从call指令转到i_max执行后，i_max中804844b~804847f范围内的指令对应的cache组号为17，不会冲掉已经装入到cache第18组中的8048480~8048484范围内的3条指令，因而不会发生缺失。综上所述，执行i_max函数过程中可能会发生0次或1次cache缺失。（7分）

十一、根据反汇编结果回答以下问题并给出理由：在执行i_max函数的过程中，是否可能发生缺页异常？是否可能因为溢出而转内核进行相应处理？（6分）

答：不会发生缺页。（1分）因为转到i_max执行之前，已经执行了main函数，它们在同一页，因而i_max的代码已经在主存。（2分）

不会因溢出转内核处理。（1分）

执行INTO指令时，如果OF=1，则触发异常，需转内核处理。而这里反汇编结果中没有INTO指令。（2分）

十二、printf语句用于实现从屏幕上显示一个字符串信息。计算机系统如何实现printf的功能？（要求从调用printf库函数开始简要说明，包括对应哪个系统调用、如何从用户态陷入内核态、内核的大致处理过程等。200字左右）（12分）

答：（略）。

中断过程：

中断概念：

1.中断是指由于接收到来自外围硬件（相对于中央处理器和内存）的异步信号或来自软件的同步信号，而进行相应的硬件／软件处理。发出这样的信号称为进行中断请求（interrupt request，IRQ）。硬件中断导致处理器通过一个上下文切换（context switch）来保存执行状态（以程序计数器和程序状态字等寄存器信息为主）；软件中断则通常作为 CPU 指令集中的一个指令，以可编程的方式直接指示这种上下文切换，并将处理导向一段中断处理代码。中断在计算机多任务处理，尤其是实时系统中尤为有用。这样的系统，包括运行于其上的操作系统，也被称为“中断驱动的”（interrupt-driven）。

2.中断是一种使 CPU 中止正在执行的程序而转去处理特殊事件的操作，这些引起中断的事件称为中断源，它们可能是来自外设的输入输出请求，也可能是计算机的一些异常事故或其它内部原因。

3.中断：在运行一个程序的过程中，断续地以“插入”方式执行一些完成特定处理功能的程序段，这种处理方式称为中断。

中断的作用：
◎并行操作
◎硬件故障报警与处理
◎支持多道程序并发运行，提高计算机系统的运行效率
◎支持实时处理功能

术语：
◎按中断源进行分类：发出中断请求的设备称为中断源。按中断源的不同，中断可分为
内中断：即程序运行错误引起的中断
外中断：即由外部设备、接口卡引起的中断
软件中断：由写在程序中的语句引起的中断程序的执行，称为软件中断
◎允许/禁止(开/关)中断： CPU 通过指令限制某些设备发出中断请求，称为屏蔽中断。从 CPU 要不要接收中断即能不能限制某些中断发生的角度 ，中断可分为
可屏蔽中断 ：可被 CPU 通过指令限制某些设备发出中断请求的中断
不可屏蔽中断：不允许屏蔽的中断如电源掉电
◎中断允许触发器：在 CPU 内部设置一个中断允许触发器，只有该触发器置“1”，才允许中断；置“0”，不允许中断。
指令系统中，开中断指令，使中断触发器置“1”
关中断指令，使中断触发器置“0”
◎中断优先级：为了管理众多的中断请求，需要按每个（类）中断处理的急迫程度，对中断进行分级管理，称其为中断优先级。在有多个中断请求时，总是响应与处理优先级高的设备的中断请求。
◎中断嵌套：当 CPU 正在处理优先级较低的一个中断，又来了优先级更高的一个中断请求，则 CPU 先停止低优先级的中断处理过程，去响应优先级更高的中断请求，在优先级更高的中断处理完成之后，再继续处理低优先级的中断，这种情况称为中断嵌套。
Intel 的官方文档里将中断和异常理解为两种中断当前程序执行的不同机制。这是中断和异常的共同点。不同点在于：
　中断(interrupt)是异步的事件，典型的比如由 I/O 设备触发；异常(exception)是同步的事件，典型的比如处理器执行某条指令时发现出错了等等。
　中断又可以分为可屏蔽中断和非可屏蔽中断，异常又分为故障、陷阱和异常中止 3 种，它们的具体区别很多书籍和官方文档都解释的比较清楚这里不再赘述。
关于它们的区别有两点是需要注意的：
1）平常所说的屏蔽中断是不包括异常的，即异常不会因为 CPU 的 IF 位被清（关中断，指令：cli）而受影响，比如缺页异常，即使关了中断也会触发 CPU 的处理。
2）通常说的 int 80h 这种系统调用使用的中断方式实际上硬件上是理解为异常处理的，因此也不会被屏蔽掉，这也很好理解，int 80h 这种中断方式是程序里主动触发的，对于 CPU 来说属于同步事件，因此也就属于异常的范畴。

2. 中断（异常）处理过程

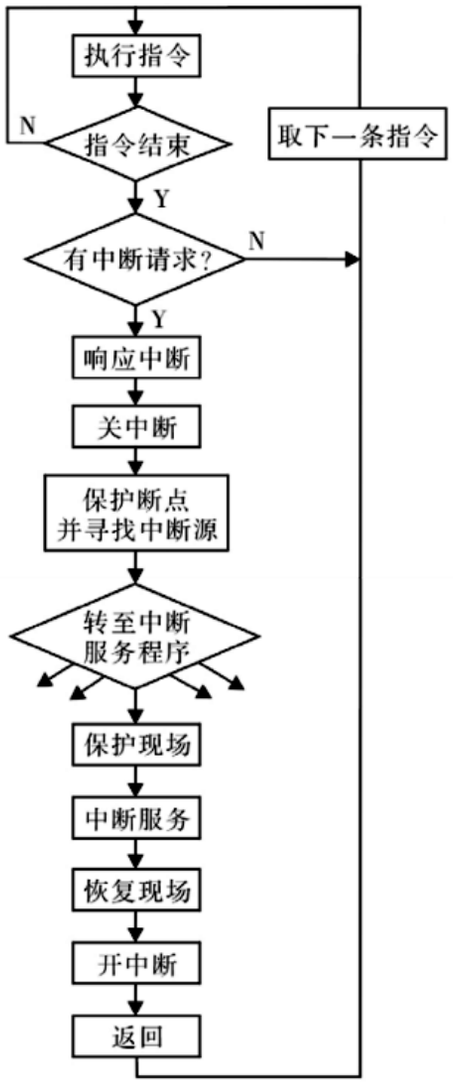


图7－1　中断处理过程的流程图

需要明确的一点是 CPU 对于中断和异常的具体处理机制本质上是完全一致的，即：
当 CPU 收到中断或者异常的信号时，它会暂停执行当前的程序或任务，通过一定的机制跳转到负责处理这个信号的相关处理程序中，在完成对这个信号的处理后再跳回到刚才被打断的程序或任务中。这里只描述保护模式下的处理过程，搞清楚了保护模式下的处理过程（更复杂），实模式下的处理机制也就容易理解了。

具体的处理过程如下：
0）中断响应的事前准备：
系统要想能够应对各种不同的中断信号，总的来看就是需要知道每种信号应该由哪个中断服务程序负责以及这些中断服务程序具体是如何工作的。系统只有事前对这两件事都知道得很清楚，才能正确地响应各种中断信号和异常。
[a]系统将所有的中断信号统一进行了编号（一共 256 个：0～255），这个号称为中断向量，具体哪个中断向量表示哪种中断有的是规定好的，也有的是在给定范围内自行设定的。
中断向量和中断服务程序的对应关系主要是由 IDT（中断向量表）负责。操作系统在 IDT 中设置好各种中断向量对应的中断描述符（一共有三类中断门描述符：任务门、中断门和陷阱门），留待 CPU 查询使用。而 IDT 本身的位置是由 idtr 保存的，当然这个地址也是由 OS 填充的。

[b]中断服务程序具体负责处理中断（异常）的代码是由软件，也就是操作系统实现的，这部分代码属于操作系统内核代码。也就是说从 CPU 检测中断信号到加载中断服务程序以及从中断服务程序中恢复执行被

暂停的程序，这个流程基本上是硬件确定下来的，而具体的中断向量和服务程序的对应关系设置和中断服务程序的内容是由操作系统确定的。

1）CPU 检查是否有中断/异常信号

CPU 在执行完当前程序的每一条指令后，都会去确认在执行刚才的指令过程中中断控制器（如：8259A）是否发送中断请求过来，如果有那么 CPU 就会在相应的时钟脉冲到来时从总线上读取中断请求对应的中断向量[2]。

对于异常和系统调用那样的软中断，因为中断向量是直接给出的，所以和通过 IRQ（中断请求）线发送的硬件中断请求不同，不会再专门去取其对应的中断向量。

2）根据中断向量到 IDT 表中取得处理这个向量的中断程序的段选择符

CPU 根据得到的中断向量到 IDT 表里找到该向量对应的中断描述符，中断描述符里保存着中断服务程序的段选择符。

3）根据取得的段选择符到 GDT 中找相应的段描述符

CPU 使用 IDT 查到的中断服务程序的段选择符从 GDT 中取得相应的段描述符，段描述符里保存了中断服务程序的段基址和属性信息，此时 CPU 就得到了中断服务程序的起始地址。

这里，CPU 会根据当前 cs 寄存器里的 CPL 和 GDT 的段描述符的 DPL，以确保中断服务程序是高于当前程序的，如果这次中断是编程异常（如：int 80h 系统调用），那么还要检查 CPL 和 IDT 表中中断描述符的 DPL，以保证当前程序有权限使用中断服务程序，这可以避免用户应用程序访问特殊的陷阱门和中断门[3]。

4）CPU 根据特权级的判断设定即将运行的中断服务程序要使用的栈的地址

CPU 会根据 CPL 和中断服务程序段描述符的 DPL 信息确认是否发生了特权级的转换，比如当前程序正运行在用户态，而中断程序是运行在内核态的，则意味着发生了特权级的转换，这时 CPU 会从当前程序的 TSS 信息（该信息在内存中的首地址存在 TR 寄存器中）里取得该程序的内核栈地址，即包括 ss 和 esp 的值，并立即将系统当前使用的栈切换成新的栈。这个栈就是即将运行的中断服务程序要使用的栈。紧接着就将当前程序使用的 ss,esp 压到新栈中保存起来。

6）保护当前程序的现场

CPU 开始利用栈保护被暂停执行的程序的现场：依次压入当前程序使用的 eflags，cs，eip，errorCode（如果是有错误码的异常）信息。

官方文档[1]给出的栈变化的示意图如下：

7）跳转到中断服务程序的第一条指令开始执行

CPU 利用中断服务程序的段描述符将其第一条指令的地址加载到 cs 和 eip 寄存器中，开始执行中断服务程序。这意味着先前的程序被暂停执行，中断服务程序正式开始工作。

8）中断服务程序处理完毕，恢复执行先前中断的程序

在每个中断服务程序的最后，必须有中断完成返回先前程序的指令，这就是 iret（或 iretd）。程序执行这条返回指令时，会从栈里弹出先前保存的被暂停程序的现场信息，即 eflags,cs,eip 重新开始执行。