第 8 次作业

1. 编译优化的策略有哪些（至少 5 种不同策略）？说明各种优化策略能提高程序运行速度的原理。

要点：

(1) 不仅要说策略是什么，而且要说出背后的原理；

(2) 出题的本意：利用哪些硬件特性来提速，即优化后能更好地发挥硬件的作用，加快运行速度。计算机系统是计算机软件和硬件组成的整体。单纯地从软件层面介绍优化（如去掉没有用的废代码减少了执行指令的数目；算出可以计算的表达式的值而不产生相应的机器指令等等），这就与硬件特性无关了。

答案：

(1) 循环展开：将程序执行流程变成一个顺序结构。消除引起循环的跳转指令，使指令流水线利用更充分，避免在指令流水线上产生要被丢弃的“半成品”而浪费时间。

(2) 分支语句向无分支语句转换：使用条件传送 cmov* 等指令，可以提高指令流水线的利用率，原因同(1) [参见“L03_Intel 中央处理器 : 流水线的控制分支冒险”]。

(3) 调整指令执行顺序：后面的指令用到前面指令的结果，前面的指令结果还未产生，后面的指令就要等待，产生阻塞就会影响指令流水线的速度。调整指令顺序的目的是减少可能的阻塞 [参见 “L03_Intel 中央处理器: 流水线的数据冒险”]。

(4) 使用执行速度更快的机器指令：例如，将一个变量中的内容乘 2，可以用变量自己与自己相加，也可以用左移运算。此处指的是一条指令被另一条指令所代替，不是用多条机器指令来代替一条慢速指令。

(5) 使用串操作指令代替用循环一个数据一个数据的处理（传送、比较、串置初值等等）：串操作指令产生的根源就是加快速度。

(6) 使用 SIMD：一条指令成组操作，节约了操作次数。

(7) 使用位数更长的寄存器：使用字节数更大的寄存器，一次就可以处理更多的内容，充分利用硬件中已有数据线宽度；

(8) 对一个二维数组调整数据处理顺序（按列顺序操作调整为按行序操作）：提高 CPU 中 cache 的命中率，减少 cache 与内存之间来回的数据交换，从而节约时间。

(9) 变量与寄存器绑定：访问变量变成访问绑定的对应寄存器，访问寄存器的速度要快于访问内存（包括 cache）的速度。

(10) 并行优化：利用多线程、多核等特性。

2. 为了提高程序的执行速度，在编写 C 语言程序时，可进行哪些优化（不考虑编译

器的优化和算法层面的优化）？（至少给出 5 种优化场景，可举例说明）

答案：

(1) 优化数据的访问顺序，如在 for 循环里对于二维数组，按照先行序、再列序访问每个元素。
(2) 减少重复计算，比如 for(int i=0; i<strlen(a); i++)中 strlen(a)多次计算。
(3) 调用封装了串操作指令的函数，如 memcpy，memset、memcmp 等。
(4) 变递归程序为迭代程序，函数调用传递参数，断点压栈等多种操作，既慢又有栈溢出的风险。
(5) 用移位实现乘除法运算，比如 x*2 变为 x<<1。
(6) 调整条件语句中组合条件的子条件顺序。例如 if （A && B），假设 90%的情况下 A 会成立，10%的情况下 B 成立，就应该写成 if (B && A)，在 90%的情况下，减少了对条件 A 的判断。
(7) 使用封装了 SIMD 指令的函数调用。
(8) 利用多线程。
(9) 去除冗余指令（像编译器一样）。
(10) 有一些优化是编译器无法无做到的（例如与指针相关的数据访问）。

3. 分析优化题。下面的 C 语言程序段（32 位段）实现了一个数组求和的功能。给出了编译后的调试版本的汇编语言代码（斜体部分为 C 语句）。

```
int sum = 0;
00FB24C8    mov     dword ptr [ebp-30h], 0
for (i = 0; i < 5; i++)
00FB24CF    mov     dword ptr [ebp-8], 0
00FB24D6    jmp     00FB24E1
00FB24D8    mov     eax, dword ptr [ebp-8]
00FB24DB    add     eax, 1
00FB24DE    mov     dword ptr [ebp-8], eax
00FB24E1    cmp     dword ptr [ebp-8], 5
00FB24E5    jge     00FB24F6
sum += a[i];
00FB24E7    mov     eax, dword ptr [ebp-8]
00FB24EA    mov     ecx, dword ptr [ebp-30h]
00FB24ED    add     ecx, dword ptr [ebp+eax*4-24h]
00FB24F1    mov     dword ptr [ebp-30h], ecx
00FB24F4    jmp     00FB24D8
00FB24F6    … …
```

(1) 指出该段程序执行效率不高的原因。

答案：很多冗余语句，数据在寄存器和内存反复拷贝（如 i++），没有充分利用寄存器。

(2) “00FB24E5 jge 00FB24F6”处指令的机器码为 7DH 0FH，解释 0FH 代表的含义。

答案：表示目标地址到（EIP）的偏移量：00FB24F6 - 00FB24E7 = 0FH。

(3) 保留循环结构，改编相应的汇编语言程序，以提高程序的执行效率。要求写出变量与寄存器对应关系（可以用标号来代表指令地址）。

答案：eax 表示 sum，ebx 表示 a 的首地址，ecx 表示 i

```
        mov     eax, 0              ; sum
        lea     ebx, [ebp-24h]      ; a
        mov     ecx, 0              ; i
L1:     cmp     ecx, 5
        jge     L2
        add     eax, [ebx+ecx*4]
        inc     ecx
        jmp     L1
L2:     mov     [ebp-30h], eax      ; sum
        …… 
```

(4) 用循环展开的方法（即去除循环）优化程序段。

答案：eax 表示 sum

```
        mov     eax, 0    ; sum
        add     eax, [ebp-24h+00h]
        add     eax, [ebp-24h+04h]
        add     eax, [ebp-24h+08h]
        add     eax, [ebp-24h+0ch]
        add     eax, [ebp-24h+10h]
        mov     [ebp-30h], eax     ; sum
        …… 
```

(5) 请用一条语句实现：将（eax）* 5 + 10 的结果送到 ebx，不用考虑溢出。

答案：lea ebx, [eax+eax*4-10]

4. 程序分析题。阅读下面的程序，回答问题。

```
.section    .data
 array:     .long    10, -20, 30, -40, 50
```

```
length = (. -array) / 4              # length = 5 为 array 中元数的个数
format:   .ascii    "%d\n\0"
.section  .text
.global   _start
_start:   mov       $0, %eax
          mov       $length, %ecx
          lea       array, %edi       # ①
L1:       cmpl      $0, (%edi)
          jl        L2                # ②
          inc       %eax
L2:       add       $4, %edi
          sub       $1, %ecx          # ③
          jne       L1
          push      %eax
          push      $format
          call      printf
          mov       $1, %eax          # 程序正常退出
          mov       $0, %ebx
          int       $0x80
```

(1) 上述程序的功能是什么？运行后，屏幕上显示的是什么？

答案：统计 array 数组中非负数的个数并显示。显示 3

(2) 若标号 L1 写到 ① 处语句前，程序运行的结果是什么？为什么？

答案：显示 5。每次循环都将 array 的地址送 edi, 每次循环都是判断数组的第一个元素是否为负数。

(3) 若将 ② 处的语句改为 “jb L2”,程序运行的结果是什么？

答案：显示 5。jb 是无符号数比较转移，任何无符号数都不低于 0。

(4) 若漏写了 ③ 处的语句，程序运行会出现什么现象？为什么？

答案：程序死循环，随着edi每次加4，cmpl $0, (%edi) 访问的内存单元超出程序空间范围，引起异常而程序奔溃。

5. 程序分析题。阅读下面的程序（部分语句列出了反汇编时看到的结果），回答问题。

```
.686P
.model          flat, c
 exitProcess    proto stdcall :dword
 includelib     kernel32.lib
 includelib     libcmt.lib
```

```
includelib    legacy_stdio_definitions.lib
printf        proto   :vararg
.data
 lpfmt        db      "%s --> %d", 0dh, 0ah, 0
 value        dd      0
 string       db      "123", 0
.stack        200
.code
 main    proc c
         push    offset string  ; 00108280   push   179010h
         call    atoui          ; 00108285   call   001082B1
         add     esp, 4         ; 0010828A   add    esp,4
         mov     value, eax     ; 0010828D   mov    dword ptr ds:[0017900Ch], eax
         invoke  printf, offset lpfmt, offset string, value
         invoke  ExitProcess, 0
 main    endp
 atoui   proc
         push    ebp                ; 001082B1   push   ebp
         mov     ebp, esp
         push    edx
         push    esi
         mov     esi, [ebp+8]       ; ①
         mov     eax, 0             ; ②
 atoi_convert:
         mov     dl, [esi]
         cmp     dl, 0
         jz      atoi_convert_over
         sub     dl, 30h
         movzx   edx, dl
         imul    eax, 10
         add     eax, edx
         inc     esi
         jmp     atoi_convert
 atoi_convert_over:
         pop     esi
         pop     edx
         pop     ebp
         ret
 atoui   endp
         end
```

(1) 上述程序运行后，屏幕上显示的结果是什么？

答案： 123 --> 123

(2) 子程序 atoui 的功能是什么？它的入口参数和出口参数分别是什么？

答案： 将以 0 结尾的 10 进制数字字符串转换为整数。

入口参数：10 进制数字字符串的首地址（堆栈传递参数）

(3) 若标号 atoi_convert 写在②处语句之前，即有 atoi_convert: mov eax, 0，程序的运行后，屏幕上显示的结果是什么？

答案： 123 --> 3

(4) 画出执行到 ① 处语句时堆栈的示意图，要求在单元外标明单元的地址，并画出 ESP 指向的单元。

设开始执行子程序 atoui 时，寄存器内容如下：

EAX = 0017AF34　　EBX = 00307000

ECX = 00000000　　EDX = C10E52E8

ESI = 00636BD0　　EBP = 004FF9D8

ESP = 004FF994　　EIP = 00108280

00636BD0	ESP
C10E52E8	004FF98C
004FF9D8	EBP
0010828A	004FF994
179010	

6. 程序改错。在一个以 0 结束的字符串中，将所有的大写字母转换为对应的小写字母，并将转换结果输出。请将程序中的语法错误和逻辑错误圈出来，并在其右侧写出正确的形式（重点关注带*的行）。

```
.686P
.model        flat, c
ExitProcess   proto   stdcall :dword
printf        proto   :ptr sbyte, :vararg
includelib    kernel32.lib
includelib    libcmt.lib
includelib    legacy_stdio_definitions.lib
.data
fmt           db      "%s", 0
buf           db      "Assembly Language"     ; * , 0
.code
main          proc
              mov     edi, buf                ; * lea edi, buf  或  mov edi, offset buf
 loop_start:  mov     dl, dword ptr [edi]     ; * mov dl, [edi]
              cmp     dl, '0'                 ; * cmp dl, 0
              jz      exit
              cmp     dl, 'A'
```

```
                jbe     loop_end                ; * jb loop_end
                cmp     [edi], 'Z'              ; * cmp byte ptr [edi], 'Z'
                jg      loop_start              ; * ja loop_end
                add     dl, 'a'-'A'
                mov     buf, dl                 ; * mov [edi], dl
  loop_end:     inc     edi
                jne     loop_start              ; * jmp loop_start
  exit:         invoke  printf, offset fmt, buf ; * buf   =>   offset buf
                invoke  ExitProcess, 0
  main          endp
                end
```

7. 程序填空。下面程序的功能是：找出 buf 中最大和最小的数，分别保存到 max_v、min_v 中。

```
... ...
.data
buf       dd   10, 30, -25, -10, 23, -20, 20
n    =    ($ - buf) / 4      ; n 为 buf 中有符号双字类型数据的个数
max_v     dd   0
min_v     dd   0
.code
          main     proc   c
          mov      eax,   buf      ; buf 中的第 0 个数，作为当前的最大数
          mov      edx,   eax
          mov      ecx,   n - 1
          mov      esi,   offset buf + 4
L1:       mov      ebx,   [esi]
          cmp      eax,   ebx
          jge      L2
          mov      eax,   ebx
L2:       cmp      edx,   ebx
          jle      next
          mov      edx,   ebx
L3:       add      esi, 4
          dec      ecx
          jnz      L1
          mov      max_v,   eax
          mov      min_v,   edx
          invoke   ExitProcess, 0
```

```
main    endp
        end
```