ASSEMBLY XIX

Oi povo...

Estou retomando o desenvolvimento do curso de assembly aos poucos e na nova série: Otimização de código para programadores C. Well... väo algumas das rotinas para aumentar a velocidade dos programas C que lidam com strings:

strlen()

A rotina strlen() é implementada da seguinte maneira nos compiladores C mais famosos:

```
int strlen(const char *s)
{
    int i = 0;
    while (*s++) ++i;
    return i;
}
```

Isso gera um código aproximadamente equivalente, no modelo small, a:

```
PROC    _strlen NEAR
ARG     s:PTR
    push    si                  ; precisamos preservar
    push    di                  ;  SI e DI.
    xor     di,di               ; i = 0;
    mov     si,s
@@_strlen_loop:
    mov     al,[si]
    or      al,al               ; *s == '\0'?
    jz      @@_strlen_exit      ; sim... fim da rotina.
    inc     si                  ; s++;
    inc     di                  ; ++i;
    jmp     short @@_strlen_loop ; retorna ao loop.
@@_strlen_exit:
    mov     ax,si               ; coloca i em ax.
    pop     si                  ; recupara SI e DI.
    pop     di
    ret
ENDP
```

Eis uma implementação mais eficaz:

```
#ifdef __TURBOC__
#include <dos.h>      /* Inclui pseudo_registradores */
#define _asm  asm
#endif

int     Strlen(const char *s)
{
    _asm push    es
```

```
#ifndef __TURBOC__
      _asm push    di
#endif

#if defined(__LARGE__) || defined(__HUGE__) || defined(__COMPACT__)
      _asm les     di,s
#else
      _asm mov     di,ds
      _asm mov     es,di
      _asm mov     di,s
#endif

      _asm mov     cx,-1
      _asm sub     al,al
      _asm repne   scasb

      _asm not     cx
      _asm dec     cx
      _asm mov     ax,cx

#ifndef __TURBOC__
      _asm pop     di
#endif

      _asm pop     es

#ifdef __TURBOC__
      return _AX;
#endif
  }
```

Essa nova Strlen() [Note que é Strlen() e näo strlen(), para näo confundir com a funçäo que já existe na biblioteca padräo!] é, com certeza, mais rápida que strlen(), pois usa a instrução "repne scasb" para varrer o vetor a procura de um caracter '\0', ao invés de recorrer a várias instruções em um loop. Inicialmente, CX tem que ter o maior valor possível (-1 não sinalizado = 65535). Essa função falha no caso de strings muito longas (maiores que 65535 bytes), dai precisaremos usar strlen()!

Uma vez encontrado o caracter '\0' devemos inverter CX. Note que se invertermos 65535 obteremos 0. Acontece que o caracter '\0' tambem é contado... dai, depois de invertermos CX, devemos decrementá-lo também, excluindo o caracter nulo!

Não se preocupe com DI se vc usa algum compilador da BORLAND, o compilador trata de salvá-lo e recuperá-lo sozinho...

strcpy()

Embora alguns compiladores sejam espertos o suficiente para usar as intruções de manipulação de blocos a implementação mais comum de strcpy é:

```
char *strcpy(char *dest, const char *src)
{
    char *ptr = dest;
    while (*dest++ = *src++);
    return ptr;
}
```

```
+-----------------------------------------------------------------+
```

Para maior compreenção a linha:

```
+-----------------------------------------------------------------+
|     while (*dest++ = *src++);                                   |
+-----------------------------------------------------------------+
```

Pode ser expandida para:

```
+-----------------------------------------------------------------+
|     while ((*dest++ = *src++) != '\0');                         |
+-----------------------------------------------------------------+
```

O código gerado, no modelo small, se assemelha a:

```
+-----------------------------------------------------------------+
|  PROC    _strcpy                                                |
|  ARG     dest:PTR, src:PTR                                      |
|      push    si          ; Salva SI e DI                        |
|      push    di                                                 |
|                                                                 |
|      mov     si,[dest]  ; Carrega os pointers                   |
|                                                                 |
|      push    si                  ; salva o pointer dest         |
|                                                                 |
|      mov     di,[src]                                           |
|                                                                 |
|  @@_strcpy_loop:                                                |
|      mov     al,byte ptr [di]    ; Faz *dest = *src;            |
|      mov     byte ptr [si],al                                   |
|                                                                 |
|      inc     di                  ; Incrementa os pointers       |
|      inc     si                                                 |
|                                                                 |
|      or      al,al               ; AL == 0?!                    |
|      jne     short @@_strcpy_loop ; Não! Continua no loop!      |
|                                                                 |
|      pop     ax                  ; Devolve o pointer dest.      |
|                                                                 |
|      pop     di          ; Recupera DI e SI                     |
|      pop     si                                                 |
|                                                                 |
|      ret                                                        |
|  ENDP                                                           |
+-----------------------------------------------------------------+
```

Este código foi gerado num BORLAND C++ 4.02! Repare que as instruções:

```
+-----------------------------------------------------------------+
|      mov        al,byte ptr [di]    ; Faz *dest = *src;         |
|      mov        byte ptr [si],al                                |
+-----------------------------------------------------------------+
```

Poderiam ser facilmente substituidas por um MOVSB se a ordem dos registradores de índice não estivesse trocada. Porém a substituição, neste caso, causaria mais mal do que bem. Num 386 as instruções MOVSB, MOVSW e MOVSD consomem cerca de 7 ciclos de máquina. No mesmo microprocessador, a instrução MOV, movendo de um registrador para a memória consome apenas 2 ciclos. Perderiamos 3 ciclos em cada iteração (2 MOVS = 4 ciclos). Numa string de 60000 bytes, perderiamos cerca de 180000 ciclos de máquina... Considere que cada ciclo de máquina NAO é cada ciclo de clock. Na realidade

um único ciclo de máquina equivale a alguns ciclos de clock - vamos pela média... 1 ciclo de máquina ¸ 2 ciclos de clock, no melhor dos casos!

Vamos dar uma olhada no mesmo código no modelo LARGE:

```
PROC _strcpy
ARG  dest:PTR, src:PTR
LOCAL temp:PTR
    mov        dx,[word high dest]
    mov        ax,[word low dest]
    mov        [word high temp],dx
    mov        [word low temp],ax

@@_strcpy_loop:
    les        bx,[src]

    inc        [word low src]

    mov        al,[es:bx]

    les        bx,[dest]

    inc        [word low dest]

    mov        [es:bx],al

    or         al,al
    jne        short @@_strcpy_loop

    mov        dx,[word high temp]
    mov        ax,[word low temp]
    ret
_strcpy    endp
```

Opa... Cade os registradores DI e SI?! Os pointers são carregados varias vezes durante o loop!!! QUE DESPERDICIO! Essa strcpy() é uma séria candidata a otimizaçäo!

Eis a minha implementaçäo para todos os modelos de memória (assim como Strlen()!):

```
char *Strcpy(char *dest, const char *src)
{
      _asm    push    es
#if defined(__LARGE__) || defined(__HUGE__) || defined(__COMPACT__)
      _asm    push    ds
      _asm    lds     si,src
      _asm    les     di,dest
#else
      _asm    mov     si,ds
      _asm    mov     es,si
      _asm    mov     si,src
      _asm    mov     di,dest
#endif
      _asm    push    si

Strcpy_loop:
      _asm    mov     al,[si]
      _asm    mov     es:[di],al
```

```
        _asm    inc     si
        _asm    inc     di

        _asm    or      al,al
        _asm    jne     Strcpy_loop

        _asm    pop     ax
 #if defined(__LARGE__) || defined(__HUGE__) || defined(__COMPACT__)
        _asm    mov     ax,ds
        _asm    mov     dx,ax
        _asm    pop     ds
 #endif
        _asm    pop     es
   }
```

Deste jeito os pointers são carregados somente uma vez, os registradores de segmento DS e ES são usados para conter as componentes dos segmentos dos pointers, que podem ter segmentos diferentes (no modelo large!), e os registradores SI e DI são usados como indices separados para cada pointer!

A parte critica do código é o interior do loop. A única diferença entre essa rotina e a rotina anterior (a não ser a carga dos pointers!) é a instrução:

```
        _asm    mov     es:[di],al
```

Que consome 4 ciclos de máquina. Poderiamos usar a instrução STOSB, mas esta consome 4 ciclos de máquina num 386 (porém 5 num 486). Num 486 a instrução MOV consome apenas 1 ciclo de máquina! Porque MOV consome 4 ciclos neste caso?! Por causa do registrador de segmento explicitado! Lembre-se que o registrador de segmento DS é usado como default a não ser que usemos os registradores BP ou SP como indice!

Se vc está curioso sobre temporização de instruções asm e otimização de código, consiga a mais nova versão do hypertexto HELP_PC. Ele é muito bom. Quanto a livros, ai vão dois:

¦ Zen and the art of assembly language
¦ Zen and the art of code optimization

Ambos de Michael Abrash.

AHHHHHHHH... Aos mais atenciosos e experientes: Não coloquei o prólogo e nem o epílogo das rotinas em ASM intencionalmente. Notem que estou usando o modo IDEAL do TURBO ASSEMBLY para não confundir mais ainda o pessoal com notações do tipo: [BP+2], [BP-6], e detalhes do tipo decremento do stack pointer para alocação de variáveis locais... Vou deixar a coisa o mais simples possível para todos...

Da mesma forma: Um aviso para os novatos... NAO TENTEM COMPILAR os códigos em ASM (Aqueles que começão por PROC)... Eles são apenas uma demonstração da maneira como as funções "C" são traduzidas para o assembly pelo compilador, ok?

Well... próximo texto tem mais...