Procedimentos e Macros

João Marcelo Uchôa de Alencar

11 de maio de 2023

1/51

2/51

Introdução

- Começamos falando de procedimentos.
- Vamos apresentar as macros.
- São conceitos que permite ao desenvolvedor poupar tempo.
- Por fim, falamos de montagem condicional.

3/51

Procedimentos

- Subprocedimentos, subprogramas, subrotinas, funções ou métodos.
- Funções e métodos em geral retornam apenas um valor.
- Procedimentos retornam ou mais valores.
- No lugar de parâmetros, a maneira mais simples de comunicação entre um programa e seus procedimentos são variáveis globais e registradores.

4/51

Procedimentos

- A instrução *call* recebe um operando que especifica o nome do procedimento a ser invocado.
- Após o fim do procedimento, a execução continua na instrução após a instrução *call*:

```
call pname
```

- O código do procedimento pode estar em vários locais.
- Por convenção, vamos colocá-lo após a sentença *endp* do programa *main*, mas antes do *end*.

5/51

Procedimentos

- Procedimento chamado *pname*:

```
pname proc
        ; body of the procedure
        ret
pname endp
```

- A instrução *ret* não retorna valor, apenas indica que a execução deve retornar para o programa que invocou o procedimento.

6/51

Procedimentos

- Assim como o *return* nas linguagens de alto nível, pode haver mais de um *ret* em um procedimento.
- Entretanto, é bom estruturar procedimentos com um único ponto de entrada e saída.

```
sample1   proc
          .if eax == 0
          mov edx, 1
          ret
          .else
          mov edx, 0
          ret
          .endif
sample1   endp
```

```
sample1   proc
          .if eax == 0
          mov edx, 1
          .else
          mov edx, 0
          .endif
          ret
sample1   endp
```


7/51

Procedimentos

- Sempre é bom projetar o código para que a instrução *ret* seja a última antes do *endp*.

```
; *** Caution: Contains a logic error ***
sample2   proc
          add eax, ebx
          add eax, ecx
          ret
          add eax, edx
sample2   endp
```

- *add eax, edx* nunca é executado, código morto.

8/51

Procedimentos

- Procedimento para fazer a multiplicação através da soma.

```
mult    proc
        mov eax, 0          ; initialize eax to 0
        .if x != 0
        mov ecx, 1          ; initialize i to 1
        .while ecx <= y
        add eax, x          ; add x to eax
        inc ecx             ; increment i by 1
        .endw
        .endif
        ret
mult    endp
```


9/51

Procedimentos

- Se *ecx* estiver sendo usado pelo programa que invoca *mult*, pode haver problemas.
- Uma alternativa é no momento de invocar o procedimento, salvar na pilha ou na memória o *ecx*. Ao retornar, o *ecx* seria restaurado.
- A melhor opção é o próprio procedimento salvar e restaurar o registrador.
- Usando a pilha, o processo é direto.
- Apesar de chamar um procedimento ser um pouco mais lento do que código direto, ele economiza memória com menos código replicado.

10/51

Procedimentos

```
mult  proc
      push ecx              ; save ecx
      mov eax, 0            ; initialize eax to 0
      .if x != 0
      mov ecx, 1            ; initialize i to 1
      .while ecx <= y
      add eax, x            ; add x to eax
      inc ecx               ; increment i by 1
      .endw
      .endif
      pop ecx               ; restore ecx
      ret
mult  endp
```


11/51

Programa Completo: Implementando a Função Potência em um Procedimento

- Calcular o valor de x^n.
- Retiramos o código para calcular x^n do programa principal e colocamos em um procedimento.
- A função *power* é um procedimento.
- x, n e *ans* são variáveis globais.
- Na versão em MASM, usamos *ecx* no lugar de i.

12/51

Programa Completo: Implementando a Função Potência em um Procedimento

```c
#include <stdio.h>
int x, n, ans;
void power();
int main() {
    printf("%s", "Enter x: ");
    scanf("%d", &x);
    printf("%s", "Enter n: ");
    scanf("%d", &n);
    power();
    printf("\n%s%d\n\n",
             "The answer is: ", ans);
    return 0;
}
```

```c
void power() {
    int i;
    ans = -1;
    if (x < 0 || n < 0)
        printf("\n%s\n",
                 "Error: Negative x and/or y");
    else
        if (x == 0 && n == 0)
            printf("\n%s\n",
                     "Error: Undefined answer");
        else {
            i = 1;
            ans = 1;
            while (i <= n) {
                ans = ans * x;
                i++;
            }
        }
}
```

Programa Completo: Implementando a Função Potência em um Procedimento

```
            .686
            .model  flat, c
            .stack  100h
printf      PROTO   arg1:Ptr Byte, printlist:VARARG
scanf       PROTO   arg2:Ptr Byte, inputlist:VARARG
            .data
in1fmt      byte  "%d", 0
msg1fmt     byte  "%s", 0
msg3fmt     byte  "%s%d", 0Ah, 0Ah, 0
errfmt      byte  "%s", 0Ah, 0
errmsg1     byte  0Ah, "Error: Negative x and/or y", 0
errmsg2     byte  0Ah, "Error: Undefined answer", 0
msg1        byte  "Enter x: ", 0
msg2        byte  "Enter n: ", 0
msg3        byte  0Ah, "The answer is: ", 0
x           sdword ?
n           sdword ?
ans         sdword ?
            .code
main        proc
            INVOKE printf, ADDR msg1fmt, ADDR msg1
            INVOKE scanf, ADDR in1fmt, ADDR x
            INVOKE printf, ADDR msg1fmt, ADDR msg2
            INVOKE scanf, ADDR in1fmt, ADDR n
            call power
            INVOKE printf, ADDR msg3fmt, ADDR msg3, ans
            ret
main        endp
```

```
power       proc
            push eax ; save registers
            push ecx
            push edx
            mov ans, -1 ; default value for ans
            .if x < 0 || n < 0
            INVOKE printf, ADDR errfmt, ADDR errmsg1
            .else
            .if x == 0 && n == 0
            INVOKE printf, ADDR errfmt, ADDR errmsg2
            .else
            mov ecx, 1  ; initialize ecx loop counter
            mov ans, 1  ; initialize ans
            .while ecx <= n
            mov eax, ans ; load eax with ans
            imul x       ; multiply eax by x
            mov ans, eax ; store eax in ans
            inc ecx      ; increment eax loop counter
            .endw
            .endif
            .endif
            pop edx      ; restore registers
            pop ecx
            pop eax
            ret
power       endp
            end
```


14/51

Programa Completo: Implementando a Função Potência em um Procedimento

- Poderíamos usar registradores no lugar de variáveis globais (x e n) no programa MASM?
  - As diretivas de alto nível como *.if* consideram números sem sinal a não ser que uma variável *sdword* em memória seja utilizada.
  - INVOKE apaga os registradores.
- Para procedimentos curtos, registradores são uma opção. Mas procedimentos maiores, o melhor é usar variáveis.
- Perceba que todos os registradores usados no procedimento são salvos e restaurados da pilha.

15/51

Salvando e Restaurando Registradores

- Se registradores não forem utilizados para retornar valores ao programa principal, é possível salvar e restaurar todos os registradores, alterados ou não.
- Ao salvar e restaurar apenas os registradores que são alterados, o código fica mais claro.
- Se um procedimento utilizar todos os registradores, salvar e restaurar todos com *pop* e *push* pode ficar bagunçado.
- É possível salvar todos os registradores gerais (*eax*, *ebx*, *ecx* e *edx*) e alguns adicionais (*esi*, *edi*, *ebp* e *esp*) com apenas uma instrução:
  - *pushad* coloca todos os registradores na pilha.
  - *popad* retira todos os registradores da pilha.
- Lembre que INVOKE destrói o conteúdo de *eax* e *ecx*.

16/51

Salvando e Restaurando Registradores

```
blankln         proc
                push eax
                push ebx
                push ecx
                push edx
                .repeat
                INVOKE printf, ADDR blnkfmt
                dec ebx
                .until ebx <= 0
                pop edx
                pop ecx
                pop ebx
                pop eax
                ret
blankln         endp
```


17/51

Salvando e Restaurando Registradores

```
blankln         proc
                pushad
                .repeat
                INVOKE printf, ADDR blnkfmt
                dec ebx
                .until ebx <= 0
                popad
                ret
blankln         endp
```

- Como não há valores retornados, o código fica mais limpo.
- Quando um registrador é usado para retornar um valor, precisamos usar a abordagem de salvar e restaurar individualmente.

18/51

Macros

- Alternativa a procedimentos, executa mais rápido, consome mais memória.
- Em geral, são escritas antes da diretiva *.code*.
- A declaração não ocupa espaço na memória.

```
mname           macro
                ; body of the macro
                endm
```

- Não há *ret* ao final da macro, pois não há desvio para o código da macro, como ocorre nos procedimentos.
- O montador recorta e cola todo o código da macro no local de cada invocação.
- Não há necessidade da instrução *call*, basta colocar o nome da macro.

19/51

Macros

- Macro para trocar duas posições de memória:

```
swap    macro
        mov ebx, num1   ; copy num1 into ebx
        xchg ebx, num2  ; exchange ebx and num2
        mov num1, ebx   ; copy ebx into num1
        endm
```

- Não há a necessidade da instrução *call*:

```
...
swap
...
...
swap
...
```


20/51

Macros

- Por debaixo dos panos, a declaração da macro existe apenas no arquivo *.asm*.
- O montador cria um arquivo intermediário *.lst* no qual o código da macro é copiado.
- Ao final, o *.exe* binário é gerado a partir do *.lst*, com toda menção ao nome da macro substituída pelo código da mesma.
- Se nós colocarmos as instruções *push* e *pop* na macro, essas instruções serão copiadas várias vezes no código final.
- O uso de registradores em macros deve ter bem projetado para evitar código grande consumindo memória.

21/51

Macros

- Duas menções à macro *swap* geraria o seguinte arquivo *.lst*:

```
                        swap
00000000  8B 1D 00000046 R   1   mov ebx,num1     ; copy num1 into ebx
00000006  87 1D 0000004A R   1   xchg ebx,num2    ; exchange ebx and num2
0000000C  89 1D 00000046 R   1   mov num1,ebx     ; copy ebx into num1
                        swap
00000012  8B 1D 00000046 R   1   mov ebx,num1     ; copy num1 into ebx
00000018  87 1D 0000004A R   1   xchg ebx,num2    ; exchange ebx and num2
0000001E  89 1D 00000046 R   1   mov num1,ebx     ; copy ebx into num1
```

- As colunas da esquerda são os endereços de memória e a representação em linguagem de máquina em hexadecimal.
- Veja que mesmo instruções idênticas ocupam endereços diferentes.

22/51

Macros

- Em linguagens de alto nível, o programa que invoca um procedimento envia argumentos que correspondem aos parâmetros no procedimento ou função:
  - Passagem por referência: C++, C, etc.
  - Passagem por valor: C++, Java, etc.
- As macros tem **passagem por nome**.
- Os nomes dos argumentos são apenas substituídos pelos nomes dos parâmetros.

23/51

Macros

- Se na macro *swap*, quiséssemos trocar os valores de variáveis arbitrárias?

```
swap macro p1, p2
     mov ebx, p1          ; copy p1 into ebx
     xchg ebx, p2         ; exchange ebx and p2
     mov p1, ebx          ; copy ebx into p1
     endm
```

- Na invocação:

```
...
swap num1, num2
...
...
swap x, y
...
```


24/51

Macros

- A macro vai ficar com cara de uma instrução memória-memória.
- Mas na expansão, o que acontece é substituição na cópia do código:

```
...
swap num1, num2
     mov ebx, num1        ; copy p1 into ebx
     xchg ebx, num2       ; exchange ebx and p2
     mov num1, ebx        ; copy ebx into p1
...
...
swap x, y
     mov ebx, x           ; copy p1 into ebx
     xchg ebx, y          ; exchange ebx and p2
     mov x, ebx           ; copy ebx into p1
...
```


25/51

Macros

- Usando : *REQ* na lista de parâmetros, podemos definir o parâmetro como essencial:

```
swap macro p1:REQ, p2:REQ
     mov ebx, p1          ;; copy p1 into ebx
     xchg ebx, p2         ;; exchange ebx and p2
     mov p1, ebx          ;; copy ebx into p1
     endm
```

- Se o programador não informar os parâmetros, haverá um erro de sintaxe.
- Podemos usar ;; para evitar que os comentários sejam copiados na expansão.

26/51

Macros

- Uma invocação *swap num, 1* levaria um erro de sintaxe:

```
swap num1, 1
     mov ebx, num1
     xchg ebx, 1
error A2070: invalid instruction operands
     mov x, ebx
```

- O que aconteceria se usássemos registradores no lugar de variáveis na memória?

```
...
swap eax, ecx
...
...
swap ebx, ecx
...
```


27/51

Macros

- No primeiro caso, *xchg eax, ecx* seria suficiente.
- No segundo caso ainda tenho duas cópias *mov ebx, ebx* desnecessárias.

```
...
swap eax, ecx
     mov ebx, eax
     xchg ebx, ecx
     mov eax, ebx
...
...
swap ebx, ecx
     mov ebx, ebx
     xchg ebx, ecx
     mov ebx, ebx
...
```


28/51

Macros

- Poderíamos ter um caso pior.
- *ebx* é apagado, e o valor de *ecx* é copiado nele mesmo duas vezes!

```
...
swap ecx, ebx
     mov ebx, eax
     xchg ebx, ecx
     mov eax, ebx
...
...
```

- É um erro lógico.
- Não é emitido um erro de sintaxe.

29/51

Montagem Condicional

- Montagem condicional permite controlar qual será o trecho de código a ser montado.
- Parece com um *if-then-else*, mas não está controlando a execução, e sim a montagem.

Diretiva	Significado
if	Se (pode usar EQ, NE, LT, LE, GT, GE, OR e AND)
ifb	Se em branco
ifnb	Se não está em branco
ifidn	Se idêntico
ifidni	Se idêntico maiúscula ou minúscula
ifdif	Se diferente
ifdifi	Se diferente maiúscula ou minúscula

30/51

Montagem Condicional

- Considere a necessidade de examinar se há ou não um argumento na invocação de uma macro.
- Nós já vimos que : *REQ* pode emitir um erro se o argumento estiver ausente.
- Mas se no lugar de gerar um erro, pudéssemos escolher outra versão da macro para ser incluída no código gerado?
- Considere uma macro chamada *addacc*:
  - Quando invocada sem argumentos, adicionar 1 a *eax*. A instrução *inc* é a mais indicada.
  - Quando uma constante, registrador ou memória é usada como argumento, o mesmo é adicionado a *eax*. A instrução *add* é a mais indicada.

31/51

Montagem Condicional

- : *REQ* não é usado.
- *ifb* é uma diretiva, orienta o montador.

```
addacc macro parm
       ifb <parm>
       inc eax
       else
       add eax, parm
       endif
       endm
```

- Ao processar o arquivo *.asm*, o montador verifica se o argumento está ausente, em branco.
- Se estive em branco, inclui *inc eax* no arquivo *.lst*, caso contrário, *add eax, parm*.

32/51

Montagem Condicional

```
                                        addacc
                                        1           ifb <>
                    0000000A 40         1           inc eax
                                        1           else
                                        1           add eax,
                                        1           endif
.
addacc                                  addacc 5
                                        1           ifb <5>
.                                       1           inc eax
.                                       1           else
addacc 5            0000000B 83 C0 05   1           add eax,5
.                                       1           endif
.                                       addacc edx
addacc edx                              1           ifb <edx>
.                                       1           inc eax
.                                       1           else
                    0000000E 03 C2      1           add eax,edx
addacc num                              1           endif
.                                       addacc num
                                        1           ifb <num>
                                        1           inc eax
                                        1           else
                    00000010 03 05 0000003A R   1   add eax,num
                                        1           endif
```


33/51

Montagem Condicional

- O código anterior tem alguma economia de memória.
- A instrução *inc eax* tem apenas 1 *byte*.
- A instrução *add eax, num* tem 6 *bytes*.
- O código *.asm* pode até aparentar ter muitas instruções, mas o *.lst* gerado pode ter bem menos.

34/51

Macro *Swap* Revisitada Usando Montagem Condicional

- Considere as invocações:

```
...
swap num1, num1
...
swap eax, eax
...
```

- Usando a seguinte definição:

```
swap macro p1:REQ, p2:REQ
     mov ebx, p1          ;; copy p1 into ebx
     xchg ebx, p2         ;; exchange ebx and p2
     mov p1, ebx          ;; copy ebx into p1
     endm
```


35/51

Macro *Swap* Revisitada Usando Montagem Condicional

- O resultado seria:

```
...
swap num1, num1
     mov ebx, num1
     xchg ebx, num1
     mov num1, ebx
...
...
swap eax, eax
     mov ebx, eax
     xchg ebx, eax
     mov eax, ebx
...
```

- Novamente, código redundante.

36/51

Macro *Swap* Revisitada Usando Montagem Condicional

- *ifidn* e *ifidni* (se igual) verificam se dois argumentos são iguais.
- *ifdif* e *ifdifi* (se diferente) verificam se dois argumentos são diferentes.

```
swap macro p1:REQ, p2:REQ
     ifdifi <p1>,<p2>
     mov ebx, p1          ;; copy p1 into ebx
     xchg ebx, p2         ;; exchange ebx and p2
     mov p1, ebx          ;; copy ebx into p1
     endif
     endm
```


37/51

Macro *Swap* Revisitada Usando Montagem Condicional

```
swap num1,num2
swap num1,num1
swap eax,ecx
swap eax,eax
```

```
swap    num1,num2
        mov ebx,num1
        xchg ebx,num2
        mov num1,ebx

swap    num1,num1

swap    eax,ecx
        mov ebx,eax
        xchg ebx,ecx
        mov eax,ebx

swap    eax,eax
```

- O segundo e o quarto exemplos não geram código.
- O primeiro exemplo funciona corretamente.
- O terceiro exemplo ainda é redundante, poderia ser substituído por *xchg*.

38/51

Macro *Swap* Revisitada Usando Montagem Condicional

- Se um dos argumentos for *ebx*, a troca direta pode ser feita:

```
swap macro p1:REQ, p2:REQ
     ifidni <ebx>,<p2>
     xchg p1, ebx
     else
     mov ebx, p1
     xchg ebx, p2
     mov p1, ebx
     endif
     endm
```


39/51

Macro *Swap* Revisitada Usando Montagem Condicional

- Podemos ainda verificar se qualquer um dos dois argumentos é *ebx*:

```
swap macro p1:REQ, p2:REQ
     ifidni <ebx>,<p2>
     xchg p1, ebx
     elseifidni <p1>, <ebx>
     xchg ebx, p2
     else
     mov ebx, p1
     xchg ebx, p2
     mov p1, ebx
     endif
     endm
```


40/51

Macro *Swap* Revisitada Usando Montagem Condicional

- Se a macro for utilizada poucas vezes, não vale a pena testar todos os casos para os argumentos.
- Agora se fizer parte de uma biblioteca para ser distribuída, é bom tratar todos os casos.
- Além da economia de memória, temos proteção adicional contra programadores desavisados.

41/51

Macro da Função de Potência Usando Montagem Condicional

- Considere o cálculo da potência, retornando -1 no lugar de mensagens de erro:

$x^n =$ Se $x < 0$ ou $n < 0$, então -1.
Senão se $x = 0$ e $n = 0$, então -1.
Senão se $n = 0$, então 1.
Caso contrário $1 * x * x * x * ... * x$ (n vezes).

- Poderíamos fazer um procedimento, mas vamos aproveitar como exemplo para macros.
- Nós vamos usar a montagem condicional para evitar que o código final tenha as estruturas de seleção.
- Apenas quanto tanto x e n forem maiores que 1 é que o laço é necessário.

42/51

Macro da Função de Potência Usando Montagem Condicional

- Vamos revisar o pseudocódigo anterior para limitar ainda mais a necessidade do laço.

x^n = Se $x < 0$ ou $n < 0$, então -1.
Senão se $x = 0$ e $n = 0$, então -1.
Senão se $x = 0$ ou $x = 1$, então x.
Senão se $n = 0$, então 1.
Senão se $n = 1$, então x.
Caso contrário $1 * x * x * x * ... * x$ (n vezes).

43/51

Macro da Função de Potência Usando Montagem Condicional

```
swap macro x:REQ, n:REQ
     if (x lt 0) or (n lt 0)
     mov eax, -1
     elseif (x eq 0) and (n eq 0)
     mov eax, -1
     elseif (x eq 0) or (x eq 1)
     mov eax, x
     elseif n eq 0
     mov eax, 1
     elseif n eq 1
     mov eax, x
     else
     mov eax, x
     mov ebx, eax
     mov ecx, n
     dec ecx
     .repeat
     imul ebx
     .untilcxz
     endif
     endm
```


44/51

Macro da Função de Potência Usando Montagem Condicional

- O exemplo anterior só funciona se x e n são constantes, não podem ser registradores ou variáveis.

```
power 2, -1
      mov eax, -1
power 0, 0
      mov eax, -1
power 0, 2
      mov eax, 0
power 1, 2
      mov eax, 1
power 2, 0
      mov eax, 1
```

```
power 3, 1
      mov eax, 3
power 2, 3
      mov eax, 2
      mov ebx, eax
      mov ecx, 3
      dec ecx
@C0001:
      imul ebx
      loop @C0001
```


45/51

Programa Completo: Implementando uma Calculadora Usando Macros

- Vamos criar uma calculadora macro, que usa um acumulador para ir construindo o resultado de várias operações aritméticas.
- As macros vão simular *novas* instruções disponíveis para o desenvolvedor.
- A acumulador será o *eax*.

Instrução	Implementação	Descrição
INACC	procedimento	Ler entrada
OUTACC	procedimento	Imprime saída
LOADACC	macro	Carrega operando no acumulador
STOREACC	macro	Armazena acumulador no operando
ADDACC	macro	Adiciona operando ao acumulador
SUBACC	macro	Subtrai operando do acumulador
MULTACC	macro	Multiplica operando pelo acumulador
DIVACC	macro	Divide acumulador pelo operando

46/51

Programa Completo: Implementando uma Calculadora Usando Macros

- Apenas *eax* deve ser modificado, devemos ter cuidado para preservar os outros.
- Para otimizar as macros, vamos salvar e restaurar apenas os registradores alterados.
- Por exemplo, MULTACC poderia usar *imul*, que altera *edx*.
- Mas LOADACC usará apenas *mov* e *ADDACC* usará *add*, instruções que não alteram outros registradores.

```
LOADACC macro operand
        mov eax, operand
        endm
```

```
ADDACC macro operand
       add eax, operand
       endm
```


47/51

Programa Completo: Implementando uma Calculadora Usando Macros

```
MULTACC macro operand
        push ebx            ;; save ebx and ecx
        push ecx
        mov ebx, eax        ;; mov eax to ebx
        mov eax, 0          ;; clear accumulator to zero
        mov ecx, operand    ;; load ecx with operand
        if operand LT 0     ;; if operand is negative
        neg ecx             ;; make ecx positive for loop
        endif
        .while ecx > 0
        add eax, ebx        ;; repetitively add
        dec ecx             ;; decrement ecx
        .endw
        if operand LT 0   ;; if operand is negative
        neg eax             ;; negate accumulator, eax
        endif
        pop ecx             ;; restore ecx and ebx
        pop ebx
        endm
```


48/51

Programa Completo: Implementando uma Calculadora Usando Macros

```
OUTACC  proc
        push ebx            ;; save ebx and ecx, and edx
        push ecx
        push edx
        mov temp, eax
        INVOKE printf, ADDR msg1fmt, ADDR msg1, temp
        pop edx             ;; save ebx and ecx, and edx
        pop ecx
        pop eax
        ret
OUTACC  endp
```


49/51

Resumo

- Procedimentos criam apenas uma cópia do código, macros criam uma nova cópia por invocação.
- Procedimentos em geral salvam e restauram registradores, macros devem evitar fazer isso.
- Procedimentos poupam memória, macros ajudam no desempenho.
- Para invocar um procedimento, use a instrução *call*. Para invocar uma macro, use apenas o nome.
- Procedimentos sempre terminam com *ret*. É melhor ter apenas uma instrução *ret*.

50/51

Resumo

- Inclua o nome do procedimento como rótulo em *endp*, mas não inclua o nome da macro como rótulo em *endm*.
- Quando argumentos da macro são obrigatórios, use : *REQ*.
- As diretivas de montagem condicional não tem . antes do nome.
- A diretiva *if* só aceita constantes e os parênteses são necessários ao usar *or* ou *and*.
- *ifb* e *ifnb* servem para testar argumentos vazios.
- *ifid* e *ifdif* servem para testar se argumentos são idênticos.

51/51