Computadores de Programação (DCC/UFRJ)

Aula 11: Implementação de subrotinas na arquitetura IA32

Prof. Paulo Aguiar

Recordação: movimentação de dados sobre a pilha

Operações sobre a pilha

- **pushl S**: aponta para a próxima posição vaga na pilha fazendo %esp←%esp-4, e armazena o valor S na pilha (M[%esp]←S)
- **popl D**: copia o topo da pilha para D (D←M[%esp]) e atualiza ponteiro da pilha (%esp←%esp+4)

O registrador %esp SEMPRE armazena o endereço do topo da pilha, que cresce no sentido de endereços decrescentes

Equivalências das operações sobre a pilha

- **pushl %ebp** ≡ `subl $4,%esp`; `movl %ebp,(%esp)`
- **popl %ebp** ≡ `movl (%esp),%ebp`; `addl $4,%esp`

As operações sobre a pilha seguem a regra LIFO (*Last-in, first-out*) ou "primeiro a entrar, último a sair"

Subrotinas

Uma **chamada de subrotina** requer:

1. **passagem de valores** (parâmetros de entrada e de saída/retorno)
2. **transferência de controle** (desvio da execução para outra parte do programa)
3. **alocação/desalocação de espaço de memória** para as variáveis locais

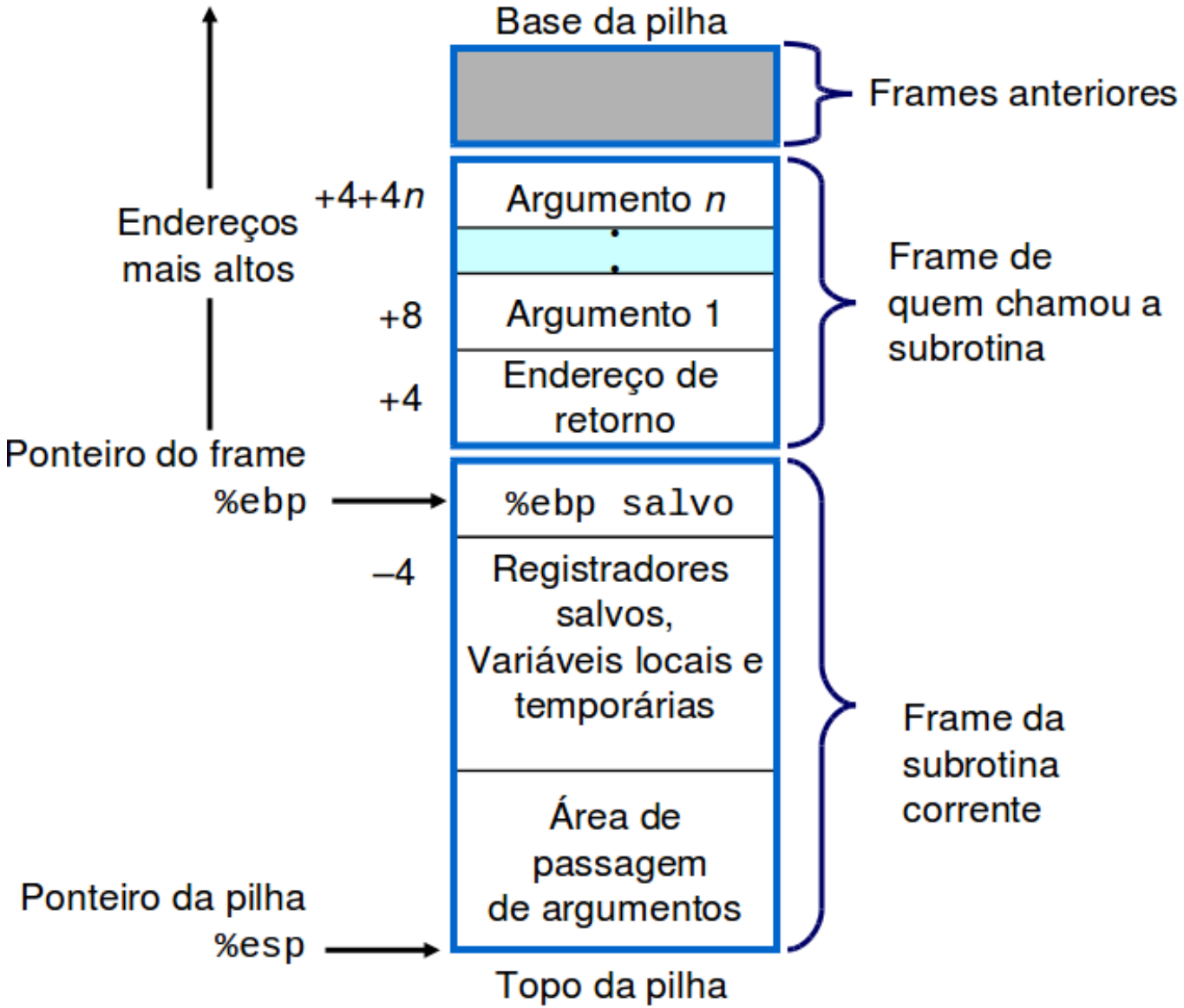
Frames (registros de ativação) da pilha

Os programas IA32 usam a pilha na implementação de chamadas de subrotinas para:

1. **passar argumentos**
2. **salvar o estado de registradores**
3. **armazenar variáveis locais**
4. **armazenar informações de retorno**

A porção da pilha alocada para uma chamada particular de subrotina é chamada de **stack frame** (ou registro de ativação)

Estrutura dos registros de ativação (*stack frames*)



Ponteiro da pilha versus ponteiro do frame

- O registrador **%ebp** é o ponteiro para a base do frame e o registrador **%esp** é o ponteiro da pilha
- O ponteiro da pilha pode variar durante a execução, todavia o ponteiro do frame deve permanecer inalterado, apontando para o início do frame, até o final da execução
- Por isso a informações são acessadas (na maioria das vezes) tomando como **endereço relativo o ponteiro do frame**

Armazenamento de variáveis locais na pilha

Sempre que possível, as variáveis locais são armazenadas em registradores (otimização do tempo de acesso), mas há casos em que isso não é possível:

1. não há registradores suficientes para armazenar todas as variáveis locais
2. as variáveis locais são **vetores** ou **estruturas de dados** (são acessadas por referência ao endereço base)
3. o operador de endereço **&** é aplicado a uma variável local e um endereço deve ser gerado para essa variável

Alocação/desalocação de espaço na pilha

- É possível **alocar** espaço na pilha previamente, sem um valor inicial, apenas **decrementando o ponteiro da pilha** (lembrar que a pilha cresce no sentido de endereços menores)
- Para **desalocar** espaço na pilha basta **incrementar o ponteiro da pilha**

Exemplo de alocação da pilha para variável local

```
void troca(int *xp, int *yp) {
    int tmp;
    tmp = *xp;
    *xp = *yp;
    *yp = tmp;
}
```

Atenção

Observar que os parâmetros da rotina são ponteiros, isto é, endereços de memória

Exemplo de alocação da pilha para variável local

Compilado sem otimização (usa %eax, %edx e pilha)

```
troca:
pushl    %ebp          // salva %ebp
movl %esp, %ebp        // cria registro de ativacao
subl $16, %esp         // aloca espaco na pilha
movl 8(%ebp), %eax     // pega xp
movl (%eax), %eax      // pega *xp
movl %eax, -4(%ebp)    // salva *xp em tmp
movl 12(%ebp), %eax    // pega yp
movl (%eax), %edx      // pega *yp
movl 8(%ebp), %eax     // pega xp
movl %edx, (%eax)      // *xp = *yp
movl 12(%ebp), %eax    // pega yp
movl -4(%ebp), %edx    // copia tmp
movl %edx, (%eax)      // *yp = tmp
leave                  // libera registro de ativacao
ret                    // retorna da subrotina
```

Exemplo de alocação da pilha para variável local

Compilado com otimização -O1 (usa %eax, %edx, %ecx e %ebx)

```
troca:
pushl %ebx              // salva %ebx
movl 8(%esp), %edx      // pega xp
movl 12(%esp), %eax     // pega yp
movl (%edx), %ecx       // copia *xp
movl (%eax), %ebx       // copia *yp
movl %ebx, (%edx)       // *xp = *yp
movl %ecx, (%eax)       // *yp = *xp
popl %ebx               // restaura %ebx
ret                     // retorna da subrotina
```

Exemplo de código com chamada de subrotina

```
int troca_soma (int *xp, int *yp) {
   int x = *xp;
   int y = *yp;
   *xp = y;
   *yp = x;
   return x + y;
}

int f () {
  int arg1 = 534;
  int arg2 = 1057;
  int soma = troca_soma(&arg1, &arg2);
  int dif = arg1 - arg2;
  return soma * dif;
}
```

Exemplo de código com chamada de subrotina

```
f: pushl   %ebp
   movl    %esp, %ebp
   subl    $24, %esp            aloca 24 bytes na pilha
   movl    $534, -4(%ebp)       arg1=534
   movl    $1057, -8(%ebp)      arg2=1057
   leal    -8(%ebp), %eax       computa &arg2
   movl    %eax, 4(%esp)        armazena na pilha
   leal    -4(%ebp), %eax       computa &arg1
   movl    %eax, (%esp)         armazena na pilha
   call    troca_soma
   movl    -4(%ebp), %edx       recupera arg1
   subl    -8(%ebp), %edx       recupera arg2  e subtrai
   imull   %edx, %eax           retorno em %eax
   leave
   ret
```

Etapas do processamento de uma subrotina

- Chamada da subrotina
- Criação de registro de ativação e leitura de argumentos
- Retorno da subrotina

CALL: instrução para chamada de subrotina

call label: chamada direta da subrotina pelo nome

O argumento `label` é o **endereço alvo de desvio**, isto é, endereço da primeira instrução da subrotina

call *operando: chamada indireta da subrotina por endereço

Assim como a instrução **jmp**, `call` pode ser **indireto** (via endereço de memória)

Efeito da instrução call

Coloca o endereço de retorno na pilha, como último objeto do registro de ativação do chamador e **desvia para a primeira instrução da subrotina**

Entrada em subrotina com criação de registro de ativação

- **push %ebp**
  salva a base do frame chamador na pilha
- **mov %esp, %ebp**
  fixa o endereço da pilha onde está salva a base anterior como a base de frame para a subrotina
- ⋮

Observe que neste caso

- `4(%ebp)` aponta para o endereço de retorno inserido pelo `call` do chamador
- `8(%ebp)` aponta para o primeiro argumento
- `12(%ebp)` aponta para o segundo argumento e assim sucessivamente

Saída da subrotina, restaurando registro de ativação do chamador

- **mov %ebp, %esp**
  aponta o ponteiro da pilha para a base do frame da subrotina, onde está salvo a base do frame chamador
- **pop %ebp**
  restaura em %ebp a base do frame chamador, deixando o topoda pilha apontando para o endereço de retorno

Entrada em subrotina sem criação de registro de ativação

No início de execução da subrotina, assumindo que a passagem de argumentos é feita pela pilha e não por registradores, e nenhuma operação de pushl tenha acontecido

- `(%esp)` aponta para o endereço de retorno inserido pelo `call` do chamador
- `4(%esp)` aponta para o primeiro argumento
- `8(%esp)` aponta para o segundo argumento, e assim sucessivamente

Todavia, se `pushl %ebx` acontece antes da coleta dos argumentos, então

- `4(%esp)` agora aponta para o end. de retorno inserido pelo `call` do chamador
- `8(%esp)` agora aponta para o primeiro argumento
- `12(%esp)` agora aponta para o segundo argumento, e assim sucessivamente

Transferência de controle: retorno de subrotina

As instruções usadas para retorno de subrotinas são:

- **leave**
  preparação da pilha para retorno da subrotina
- **ret**
  retorno da subrotina, resultado em %eax por default

A instrução **leave**

`leave`

usada para preparar a pilha para o retorno da subrotina e equivalente à seguinte sequência de código:

- **movl %ebp, %esp**
  atualiza o ponteiro da pilha para o início do frame, onde deve estar salva a base do frame de quem chamou a subrotina
- **popl %ebp**
  restaura %ebp e atualiza o ponteiro da pilha para o final do frame anterior, onde deve estar inserido o endereço de retorno (pelo `call` do chamador)

A instrução **ret**

- A instrução **ret** retira do topo da pilha um endereço de retorno, armazena no PC (program counter) e desvia a execução para esse endereço
- Para o uso correto dessa instrução, o ponteiro da pilha tem que estar apontando para o local onde a chamada call precedente armazenou o endereço de retorno

Convenções sobre o uso dos registradores

- Os registradores são recursos compartilhados por todas as subrotinas!
- Embora apenas uma subrotina possa estar ativa a cada momento, **é preciso garantir que quando uma subrotina chama outra, a subrotina chamada não sobreescreva os valores dos registradores que ainda serão usados pela subrotina anterior**

Atenção!

A arquitetura IA32 adota um conjunto de convenções para o uso dos registradores que devem ser seguidas por todas as subrotinas!

Convenções sobre o uso dos registradores em Linux

Caller e *Callee*

- Quem chama é intitulado *caller* (subrotina chamadora) e a subrotina chamada é intitulada *callee* (subrotina chamada)
- O **caller** deve cuidar de **eax**, **ecx** e **edx**
  - O **callee** pode usar esses registradores à vontade sem preocupar-se em restaurar o conteúdo alterado
- O **callee** deve salvar e restaurar os registradores **ebx**, **esi** e **edi**, se a subrotina alterar estes registros
  - Subrotinas anteriores podem ficar seguras de que esses registradores serão preservados em chamadas subsequentes

Exemplo de código com chamada de subrotina

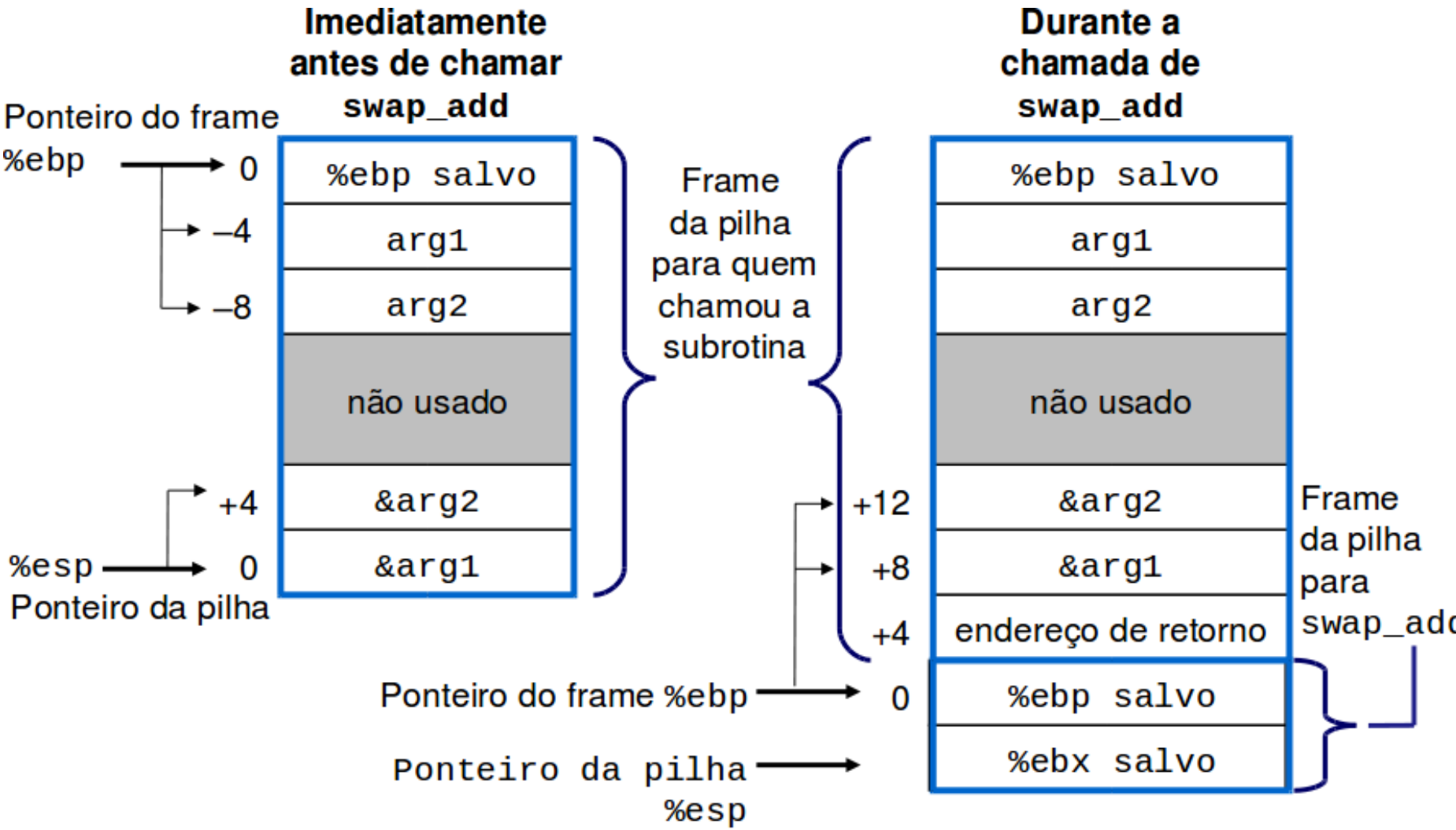
```
int troca_soma (int *xp, int *yp) {
   int x = *xp;
   int y = *yp;
   *xp = y;
   *yp = x;
   return x + y;
}

int f () {
  int arg1 = 534;
  int arg2 = 1057;
  int soma = troca_soma(&arg1, &arg2);
  int dif = arg1 - arg2;
  return soma * dif;
}
```

Exemplo de código com chamada de subrotina

```
f: pushl   %ebp
   movl    %esp, %ebp
   subl    $24, %esp              aloca 24 bytes na pilha
   movl    $534, -4(%ebp)         arg1=534
   movl    $1057, -8(%ebp)        arg2=1057
   leal    -8(%ebp), %eax         computa &arg2
   movl    %eax, 4(%esp)          armazena na pilha
   leal    -4(%ebp), %eax         computa &arg1
   movl    %eax, (%esp)           armazena na pilha
   call    troca_soma
   movl    -4(%ebp), %edx         recupera arg1
   subl    -8(%ebp), %edx         recupera arg2  e subtrai
   imull   %edx, %eax             retorno em %eax
   leave
   ret
```

Frame da pilha para o exemplo de chamada de subrotina



Exemplo de código com chamada de subrotina

```
troca_soma:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        movl    8(%ebp), %ebx
        movl    12(%ebp), %ecx
        movl    (%ebx), %eax
        movl    (%ecx), %edx
        movl    %edx, (%ebx)
        movl    %eax, (%ecx)
        leal    (%edx,%eax), %eax
        popl    %ebx
        popl    %ebp
        ret
```

Frame da pilha para o exemplo de chamada de subrotina

FRAME F antes de CALL TROCA_SOMA

end	conteúdo
%ebp	%ebp ant.
%ebp-4	534 (arg1)
%ebp-8	1057 (arg2)
%ebp-12	vazio
%ebp-16	vazio
%ebp-20	yp (&arg2=%ebp-8)
%esp	xp (&arg1=%ebp-4)

FRAME TROCA_SOMA (antes de RET)

end	conteúdo
%ebp	%ebp ant.
%ebp-4	1057 (arg1)
%ebp-8	534 (arg2)
%ebp-12	vazio
%ebp-16	vazio
%ebp+12	yp (&arg2=%ebp-8)
%ebp+8	xp (&arg1=%ebp-4)
%ebp+4	**end retorno**
%ebp	%ebp salvo
%esp	%ebx

Referências bibliográficas

1. *Computer Systems–A Programmer's Perspective* (**Cap.3**, seções 3.7.1, 3.7.2, 3.7.3, 3.7.4)