Assembly IA32 – Procedimentos

1 Procedimentos

Uma chamada a um procedimento implica a transferência de dados (na forma de parâmetros do procedimento e na forma de valores retornados pelo procedimento) e do controlo de uma parte do programa para outra parte. Além disso, deve ser reservado espaço para as variáveis locais do procedimento, no seu início, e libertado no seu fim. A maioria das arquitecturas, incluindo a IA32, disponibilizam instruções base para a transferência do controlo de e para procedimentos. A alocação e libertação de variáveis locais, bem como a passagem de dados, é conseguida através da utilização da *program stack*.

1.1 *Estrutura do* Stack Frame

Os programas suportados pela arquitectura IA32 usam a *program stack* para implementarem as chamadas a procedimentos. A *stack* é usada para passar os respectivos argumentos aos procedimentos, para guardar valores de retorno, para guardar conteúdos de registos para posterior recuperação, e para armazenamento de variáveis locais. A uma porção da *stack* reservada para um procedimento chamamos *stack frame*. A figura 1 mostra a estrutura geral do *stack frame*. O *stack frame* de topo é delimitado por dois apontadores, o registo *%ebp*, que funciona como *frame pointer*, e o registo *%esp*, que funciona como *stack pointer*. O *stack pointer* pode ser deslocado à medida que o procedimento é executado e quando exista essa necessidade, daí que geralmente a informação contida na *stack* seja acedida relativamente ao *frame pointer*.

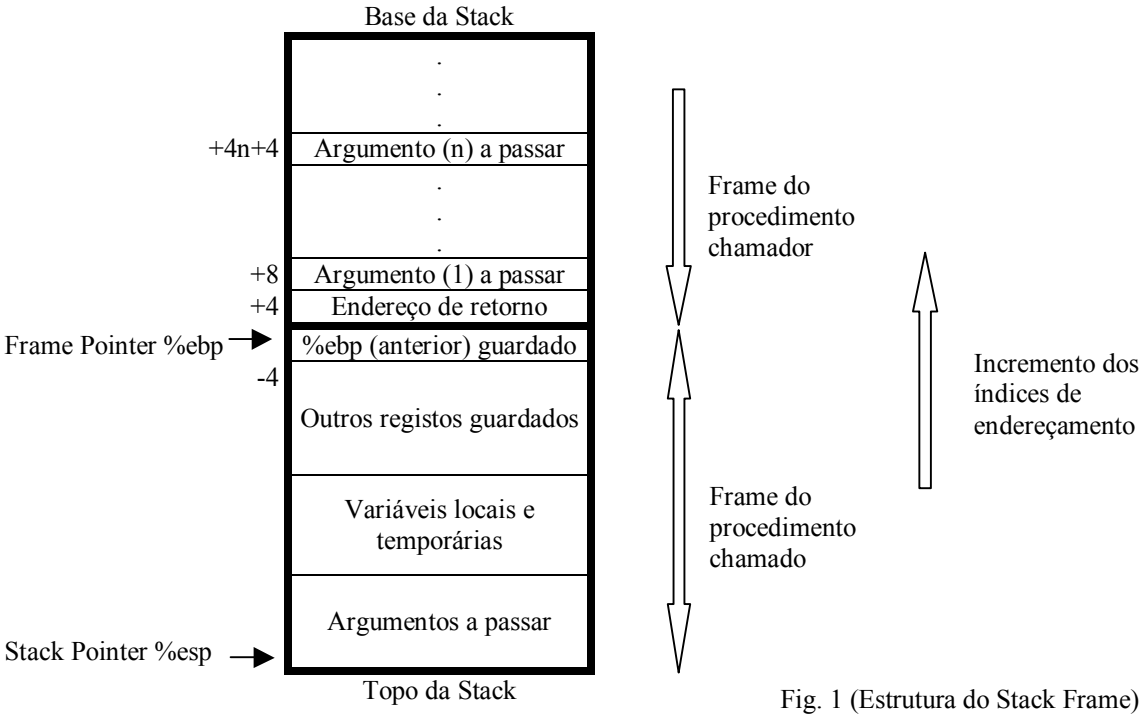

Fig. 1 (Estrutura do Stack Frame)

Considere que o procedimento P (o chamador) evoca o procedimento Q (o chamado). Os argumentos a receber por Q estão contidos no *stack frame* de P. Quando P chama Q, o endereço de programa, no qual, após término de Q, a execução deve prosseguir, é colocado (*pushed*) na *stack*, constituindo o último elemento do *stack frame* de P. O primeiro elemento do *stack frame* de Q guardará o valor do *frame pointer* de P, uma vez que *%ebp* deixará de apontar o *stack frame* de P, para apontar o *stack frame* de Q. Os elementos, do *stack frame*, situados imediatamente a seguir, poderão guardar valores de outros registos.

O procedimento Q também poderá usar a *stack* para variáveis locais que não possam ser guardadas em registos. Isto pode ocorrer pelas seguintes razões:

- Não existem registos suficientes para todas as variáveis locais;
- Algumas variáveis locais são *arrays* ou estruturas, daí que tenham de ser acedidas através de referências;
- O operador de endereço "*&*" encontra-se aplicado a uma variável local, assim terá que ser conhecido o endereço da posição de memória que a suporta (não pode ser um registo).

Os últimos elementos do *stack frame* de Q irão guardar argumentos para procedimentos que Q poderá chamar.

Como descrito atrás, o crescimento da *stack* faz-se no sentido das menores posições de memória, ficando o registo *%esp* a apontar para o elemento de topo da *stack*. Os elementos podem ser introduzidos e retirados através das instruções `pushl` e `popl`. Espaço para dados, sobre os quais ainda não se conhece o seu conteúdo, pode ser reservado na *stack*, decrementando o *stack pointer* (*%esp*) de acordo com o espaço a alocar. Similarmente, esse mesmo espaço poderá ser libertado através do incremento do *stack pointer*.

1.2 Transferência do controlo

As instruções que suportam as chamadas e os retornos de procedimentos são as seguintes:

Instrução	Descrição
`call Label`	Chama um procedimento
`call * Operand`	Chama um procedimento
`Leave`	Prepara a *stack* para o retorno
`Ret`	retorna a partir do procedimento chamado

A instrução `call` tem um parâmetro que indica o endereço da instrução onde o procedimento chamado inicia. Tal como nas instruções de salto, a chamada poderá ser directa ou indirecta. No código em *assembly*, o alvo de uma chamada directa é indicado por um *label*, enquanto que numa chamada indirecta, o alvo é dado por um operando precedido por um *.

A execução da instrução `call` provoca a adição de um novo elemento ao *stack frame* do procedimento chamador. Este novo elemento contém o endereço de programa (endereço de retorno) a atingir depois de o procedimento chamado terminar. Além disto, `call` faz com que o fluxo de programa passe para o início (primeira instrução) do procedimento chamado. O endereço de retorno é o endereço da instrução imediatamente a seguir à instrução de `call` (contidos no procedimento chamador). A instrução `ret` retira(`pop`) o elemento (último – endereço de retorno) do *stack frame* do procedimento chamador e salta para a posição de programa indicada por este. O objectivo da instrução `leave` consiste na preparação da *stack* de forma a que o *stack pointer* aponte para o elemento da *stack* onde a anterior instrução `call` guardou o endereço de retorno. Desta forma, a instrução `leave` pode ser usada para preparar a *stack* para a operação de retorno. No entanto, o mesmo pode ser feito com a seguinte sequência de código:

```
1 movl %ebp,%esp        desloca o valor do stack pointer para o início do frame (do p.chamado)
2 popl %ebp             repõe em %ebp o início do frame chamador e decrementa uma posição
```

ao *%esp*, assim este aponta agora para o final do *stack frame* do procedimento chamador.

1.3 Convenção sobre a utilização dos registos

O conjunto dos registos de programa actua como um único recurso partilhado por todos os procedimentos. Embora apenas um procedimento possa estar activo a cada instante, temos que garantir que, quando um procedimento (chamador) chama outro (chamado), este não destrua os valores guardados nos registos que o chamador pretende ainda usar a seguir. Por este motivo, na arquitectura IA32 adoptou-se uma convenção para a utilização dos registos. Esta terá que ser respeitada por todos os procedimentos.

Por convenção, registos *%eax*, *%edx* e *%ecx* estão classificados como registos guardados pelo procedimento chamador (*caller save registers*). Quando o procedimento Q é chamado por P, Q pode perder o valor guardado nesses registos sem correr o risco de destruir dados requeridos por P. Por outro lado, os registos *%ebx*, *%esi*, e *%edi* estão classificados como registos guardados pelo procedimento chamado (*callee save registers*). Isto significa que Q terá de guardar os valores destes registos na *stack*, antes de os poder utilizar, para que posteriormente possam ser repostos e utilizados pelo procedimento chamador.

Considere a seguinte situação:

```
int P( )
      {
         int x = f(); /* executa uma qualquer tarefa */
         Q();
         return x;
}
```

O procedimento P necessita que o valor computado para *x* se mantenha válido após a evocação do procedimento Q. Se *x* está num *caller save register*, então P(o chamador) tem que guardar o valor contido no registo, antes da evocação de Q, e repô-lo depois de Q terminar. Se *x* está num *callee save register*, e Q(o chamado) necessita de usar o registo, então Q tem que guardar o valor antes de usar o registo e restitui-lo antes de retornar para P. Em qualquer dos casos o acto de guardar o valor do registo passa por

copiá-lo (`push`) para a *stack*, enquanto que a recuperação do valor passa por removê-lo (`pop`) da *stack* para o registo.

Considere o seguinte exemplo:

```
int P(int x)
{
   int y = x*x;
   int Z = Q(y);

   return y + z;
}
```

O procedimento P estabelece um valor para *y* antes da chamada a Q, mas deve também assegurar que o valor de *y* continue o mesmo depois de Q retornar. Isto pode ser feito de uma das duas seguintes maneiras:

- Guardar o valor de *y* no seu próprio *stack frame*, antes de chamar Q. Quando Q retorna, P pode recuperar o valor de *y* através da *stack*.

- Guardar o valor de *y* num *callee save register*. Se Q, ou qualquer procedimento chamado por Q, necessita de usar esse registo, tem então que guardar o valor contido no registo no seu *stack frame* e recuperá-lo antes de retornar ao procedimento evocador. Assim, quando Q retornar para P, o valor de *y* encontrar-se-á no *callee save register*, ou porque o registo nunca foi alterado ou porque o seu conteúdo foi guardado e reposto.

O mais frequente é o GCC usar esta última convenção, já que tende a reduzir o número total de acessos à *stack*.

1.4 Um exemplo:

```
int swap_add(int *xp, int *yp)
{
      int x = *xp;
      int y = *yp;

      *xp = y;
      *yp = x;

      return x + y;

}

int caller()
{
      int arg1 = 534;
      int arg2 = 1057;
      int sum = swap_add(&arg1, &arg2);
      int diff = arg1 – arg2;

      return sum * diff;
}
```

Fig. 2

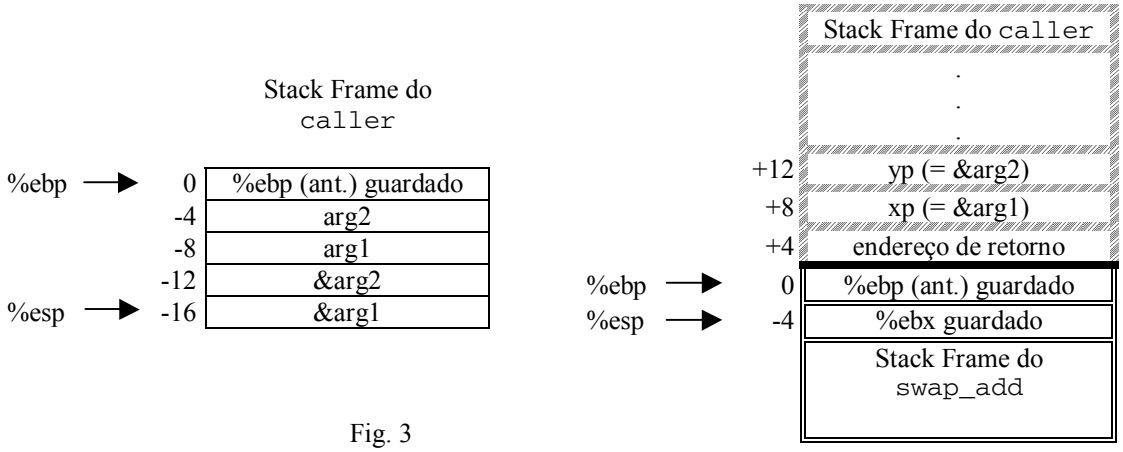

Fig. 3

Considere os procedimentos escritos em C apresentados na figura 2. A figura 3 mostra os *stack frame* para os dois procedimentos. Observe que `swap_add` retira os valores dos seu argumentos a partir do *stack frame* do procedimento `caller`. Estas localizações são obtidas por deslocamento relativo ao *frame pointer* localizado em *%ebp*. Os números colocados na parte esquerda dos *frames* representam os deslocamentos de endereço relativo ao *frame pointer*.

A *stack frame* do `caller` inclui elementos que guardam os valores das variáveis *arg1* e *arg2*, nas posições –8 e –4 relativamente ao *frame pointer*. Estas variáveis têm que estar guardadas na *stack* (e não em registos), já que temos que saber os endereços delas. O seguinte código em *assembly*, obtido da compilação do `caller`, mostra como este chama `swap_add`.

```
1     leal -4(%ebp), %eax     Obtém  o endereço de arg2
2     pushl %eax              Guarda na stack &arg2
3     leal -8(%ebp), %eax     Obtém  o endereço de arg1
4     pushl %eax              Guarda na stack &arg1
5     call swap_add           Chama a função swap_add
```

Observe neste código a obtenção do endereço das variáveis locais *arg2* e *arg1* (através da instrução `leal`) e o seu armazenamento na *stack*. Só depois é que `swap_add` é evocado.

O código resultante da compilação de `swap_add` é formado por três partes: o "setup", onde o *stack frame* é iniciado; o "body", onde a computação inerente ao procedimento é executada; e o "finish", onde o estado da *stack* é recuperado (adquire o estado anterior à chamada de `swap_add`) e o procedimento retorna.

O código seguinte representa o "setup" do procedimento `swap_add`. De notar que a instrução `call` inseriu um elemento na *stack*, cujo conteúdo é o endereço de retorno.

```
1     swap_add:
2     pushl %ebp          Guarda o %ebp que aponta p/ frame de caller
3     movl %esp,%ebp      Coloca %ebp como frame pointer de swap_add
4     pushl %ebx          Guarda %ebx na stack (adiciona uma unidade a %esp)
```

O procedimento `swap_add` necessita usar o registo *%ebx*. Sendo *%ebx* um *callee save register*, o procedimento `swap_add` tem que guardar o seu valor antes de poder usá-lo e fá-lo no "setup" adicionando um elemento à *stack*.

O código seguinte representa o "body" do procedimento `swap_add`:

```
1     movl 8(%ebp), %edx      Coloca xp em %edx
2     movl 12(%ebp), %ecx     Coloca yp em %ecx
3     movl (%edx), %ebx       x = *xp
4     movl (%ecx), %eax       y = *yp
5     movl %eax, (%edx)       Guarda y em *xp
6     movl %ebx, (%ecx)       Guarda x em *yp
7     addl %ebx, %eax         Estabelece o valor de retorno = x + y
```

Neste código podemos ver que o procedimento `swap_add` obtém os valores dos seus argumentos a partir do *stack frame* do procedimento `caller`. Daí que os valores de deslocamento sobre *%ebp* actual (*stack frame* do `caller`) sejam +12 e +8 e não –12 e –16 como acontecia com o valor de *%ebp* anterior (*stack frame* do `swap_add`). Observe que a soma sobre as variáveis *x* e *y* é guardada em *%eax*, este registo é usado na transferência do valor (ex: inteiros; apontadores) devolvido.

O código seguinte representa o "finish" do procedimento `swap_add`:

```
1     popl %ebx          Repõe %ebx
2     movl %ebp, %esp    Repõe %esp
3     popl %ebp          Repõe %ebp
4     ret                Retorna para caller
```

Este código recupera os valores de três registos *%ebx*, *%esp* e *%ebp*, a seguir executa a instrução `ret` . Repare que as instruções 2 e 3 podem ser substituídas pela instrução `leave`. Diferentes versões do GCC apresentam diferentes escolhas neste âmbito.

A seguinte linha de código aparece imediatamente a seguir à instrução que chama swap_add:

```
1     movl %eax, %edx
```

Depois de retornar do procedimento `swap_add`, `caller` prossegue a execução neste procedimento, copiando o conteúdo de *%eax* (valor devolvido) para o registo *%edx*.