Extraído de http://www.inf.puc-rio.br/~inf1612/corrente/aula8.html.
Material original desenvolvido, pela PUC-RIO.
Baseado no livro: Computer Systems, A programmer's perspective. Randal Bryant and David O'Hallaron. Prentice Hall. 2003

Procedimentos (subrotina)

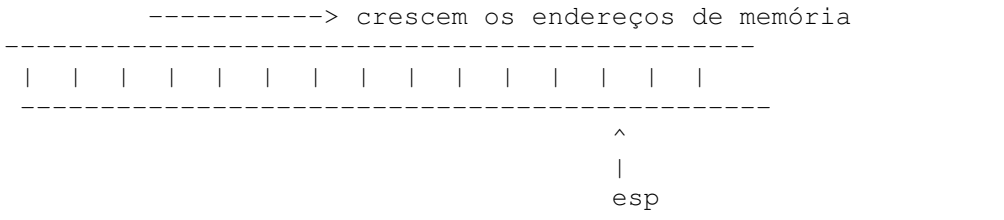
Pilha

Chamamos *área de pilha* uma espaço de memória especialmente reservado para organização de uma pilha de dados. Esta pilha é usada como memória auxiliar durante a execução de uma aplicação. As operações sobre esta área são *push* (empilha) e *pop* (desempilha).

Em geral, o hardware dá algum suporte à manutenção dessa pilha. No caso da máquina Pentium, um registrador específico, `esp`, é dedicado ao endereço do topo da pilha.

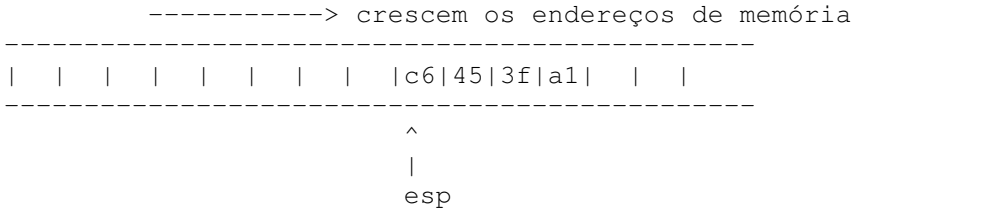
Por motivos históricos, a pilha cresce em direção aos endereços mais baixos de memória. Ou seja, para alocar espaço para um endereço, devemos subtrair 4 de `esp` (lembre-se que um endereço ocupa 4 bytes!) e para desalocar devemos somar 4 a `esp`.

Por exemplo:



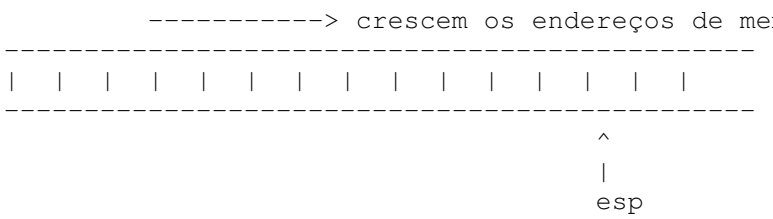

```
empilhar: pushl %eax  # suponha que o valor de eax e' a13f45c6

resultado:
```




```
desempilhar: popl %eax

resultado:
```



A operação `pushl %eax` é análoga à sequência

```
subl $4, %esp
movl %eax, (%esp)
```

e a operação `popl %eax` à sequência

```
movl (%esp), %eax
addl $4, %esp
```

Pilhas e Procedimentos

Ao chamarmos um procedimento, precisamos passar dados e controle de uma parte do código para outra. A maior parte das arquiteturas têm instruções que facilitam algumas dessas tarefas, e o restante delas tem que ser programado explicitamente, usando a pilha.

No IA32: intruções `call` e `ret` facilitam a transferência de controle.

Endereço de retorno

A instrução `call f` transfere o controle para `f`, antes armazenando o *endereço de retorno* (endereço da instrução seguinte a `call f`), para que o controle possa retornar para este endereço ao final da execução da função. A instrução `ret` faz com que o controle retorne para o último endereço de retorno armazenado.

Mas onde `call` pode armazenar esse endereço?

- `call f` armazena o endereço da próxima instrução na pilha de execução e faz um desvio para o endereço associado a *f*.
- `ret` desempilha um endereço da pilha e desvia para o endereço desempilhado.

(ou seja, ambas alteram `esp`).

Valores de Registradores

Um outro uso da pilha é *salvar* registradores durante chamadas de funções.

Cada função utiliza os registradores (eax, ebx, ecx, edx, ...) para armazenar valores de variáveis e temporários. Mas ao fazer uma chamada, esses mesmos registradores são usados pela nova função!

Para evitar que os valores anteriores se percam, podemos armazená-los na pilha. Mas quem deve armazená-los? Quem chama (caller) ou quem é chamada (callee)?

Deve existir uma convenção em cada sistema, de maneira que não se armazenem valores duas vezes e nem se deixe de armazená-los...

No Linux, caller deve armazenar `eax`, `ecx`e `edx`. Quando uma função P chama uma função Q, Q pode ficar a vontade para destruir os valores nesses registradores. Por outro lado, os registradores `ebx`, `esi` e `edi` devem ser armazenados pelo callee (Q, nesse caso).

(No windows: caller deve armazenar eax, ebx, ecx e edx.)

Passagem de Parâmetros

Uma forma simples de passar parâmetros para funções é colocá-los em registradores. No entanto, quando o número de parâmetros é grande essa técnica não é adequada...

Uma outra técnica, usada em C, é passar os parâmetros colocando-os na pilha!

A convenção adotada em C é empilhar os parâmetros na ordem inversa em que aparecem na declaração da função (da direita para a esquerda). Ou seja, dada a declaração:

```
int boba (int tam, int nums[]);
```

no momento da ativação de *boba* teremos a seguinte situação:

```
--------------
|            |
--------------
|            |
--------------
| end.ret    | <- esp
--------------
| tam        |
--------------
| nums (end) |
--------------
|            |
--------------
   ...
--------------
|            |
--------------
|            | <- ebp    ; base do registro de ativação anterior
--------------
```

nesse instante, poderíamos dizer que o primeiro parâmetro está no endereço `(esp) + 4`. No entanto, ao longo da execução da função o valor de `esp` pode variar. Por isso, utiliza-se um outro registrador, `ebp`, para apontar para a base da pilha durante toda a execução de uma função. (A porção da pilha referente a cada função é chamada de *registro de ativação*, como veremos depois. Assim, ebp aponta sempre para a base do registro de ativação corrente.)

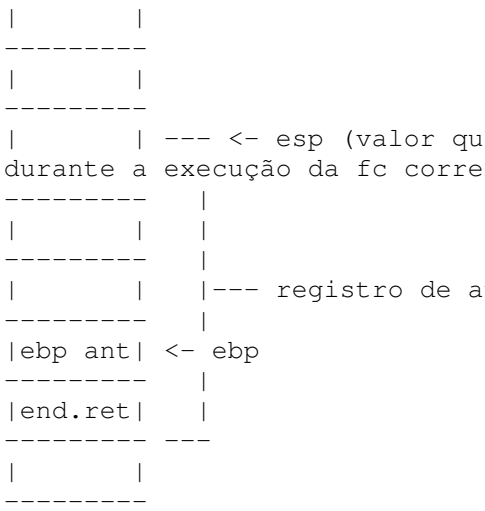
Para isso, adota-se a seguinte convenção (convenção do compilador C e outros). O código tipicamente gerado por um compilador para um procedimento começa com:

```
pushl %ebp ; salva o endereço do registro de ativação anterior
movl %esp, %ebp
```

e termina com:

```
movl %ebp, %esp
popl %ebp ; restaura o endereço do registro de ativação anterior
```

ou seja, em cada instante temos na pilha de execução:



Assim, para acessar o valor do primeiro parâmetro (tam), o código da rotina deverá acessar a posição de memória 8(%ebp). O segundo parâmetro estará em 12(%ebp), e, se tivermos mais parâmetros, assim por diante.

Obs: Cada parâmetro é colocado em quatro bytes, mesmo que seu tipo ocupe menos do que isso.

Referência

- CS:APP, 3.7.1, 3.7.2, 3.7.3.