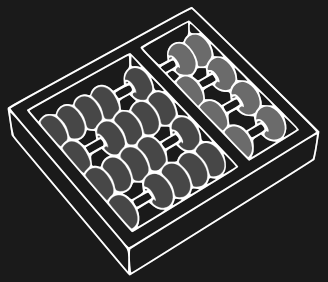


UNICAMP

Stack

Prof. Carlos A. Astudillo

castudillo@ic.unicamp.br

Sistemas Operacionais (MC504A)

2º Semestre 2023

Adaptado de Dave Eckhard

Objetivos da aula

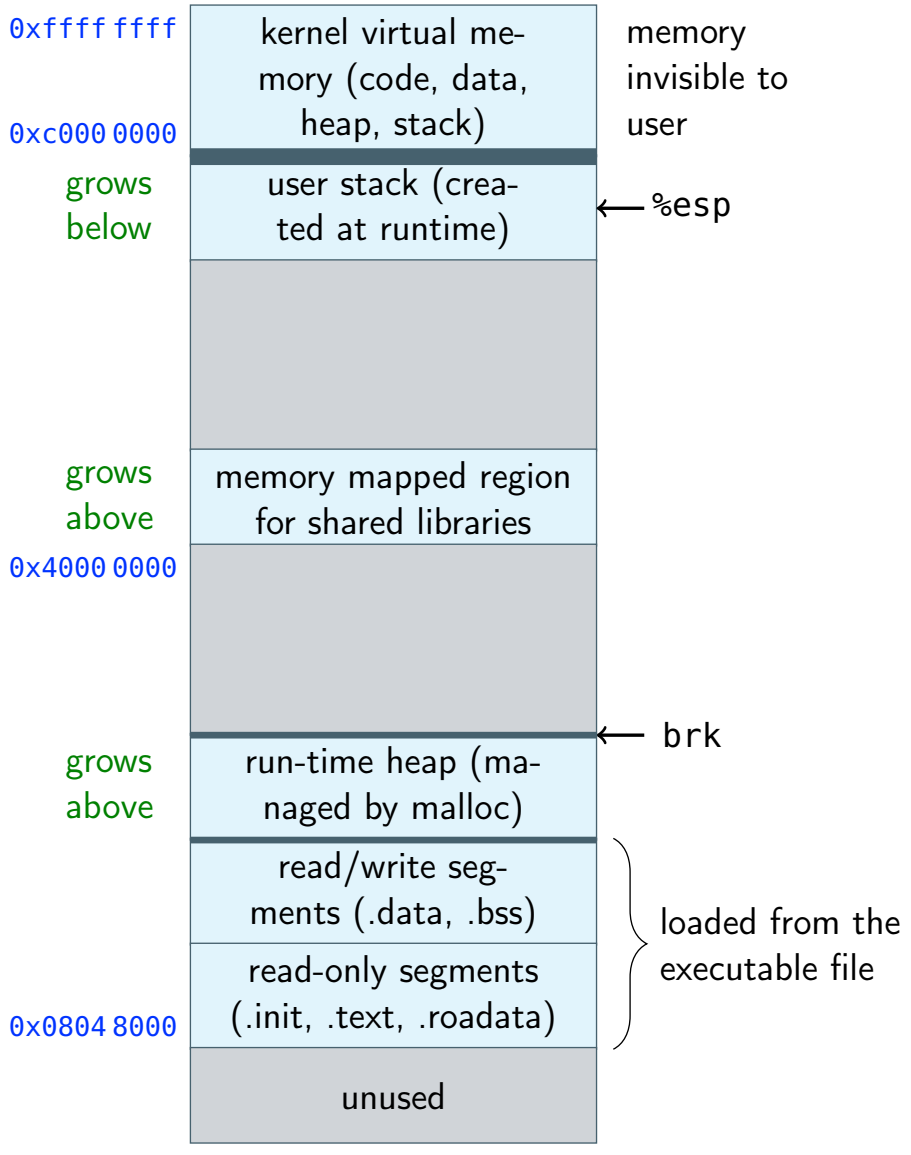
- Modelo do processo em memória
- Organização do stack
- Convenções dos registradores
- Antes e depois do `main()`

Trabalharemos com 32 bits

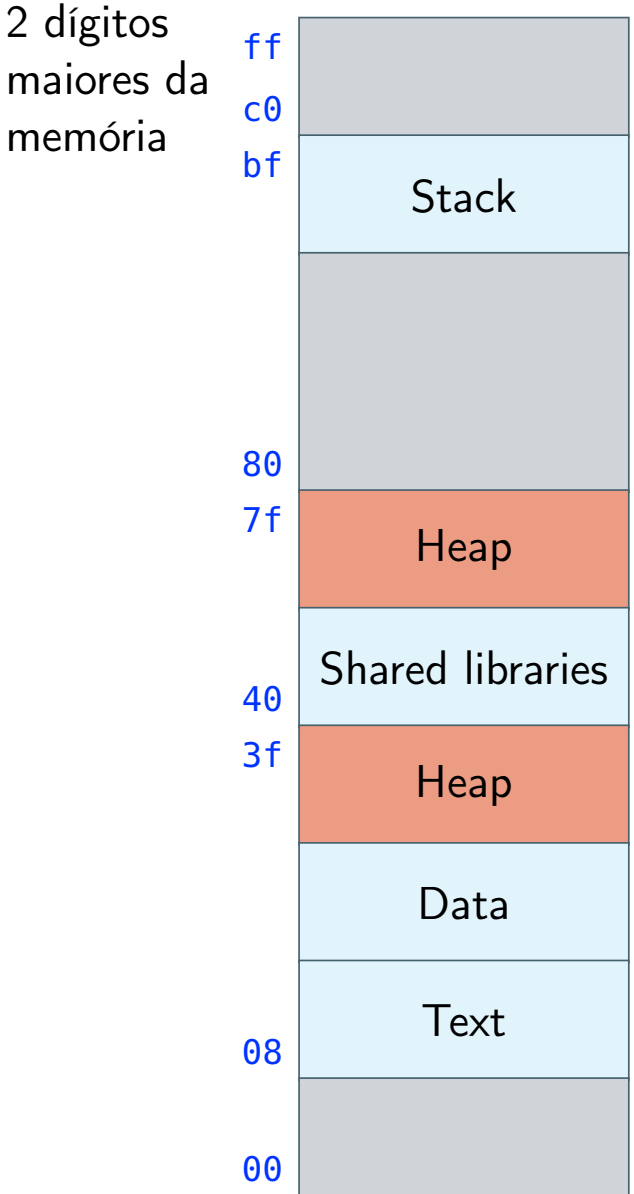
- Podem ter aprendido em x86-64, a.k.a. EMT64 a.k.a. AMD64
  - x86-64 é mais simples que x86 para o código de programas do usuário
  - Mais registradores, mais ortogonais
- Vamos nos concentrar em x86 (IA32)
  - x86-64 não é simples para o código do kernel
    - A máquina começa em modo de 16-bit, depois em 32, e por último em 64 (precisa de código de transição)
    - As interrupções são mais complicadas
  - x86-64 não é simples para depurar (*debug*)
    - Mais registradores significa que mais registradores podem ter valores incorretos
  - A memória virtual de x86-64 é feia
    - Mais passos que na x86-32, porém não mais estimulantes
  - Ainda há muitas máquinas de 32 bit no mundo

Espaço privado de memória

- Cada processo tem seu próprio espaço de memória
- Os endereços de memória varíam entre diferentes arquiteturas
- Detalhes podem variar entre diferentes sistemas operacionais, e versões do kernel
- `brk`: chamada do sistema (unix) para poder distribuir de maneira dinâmica a memória do segmento de dados (program break of process)

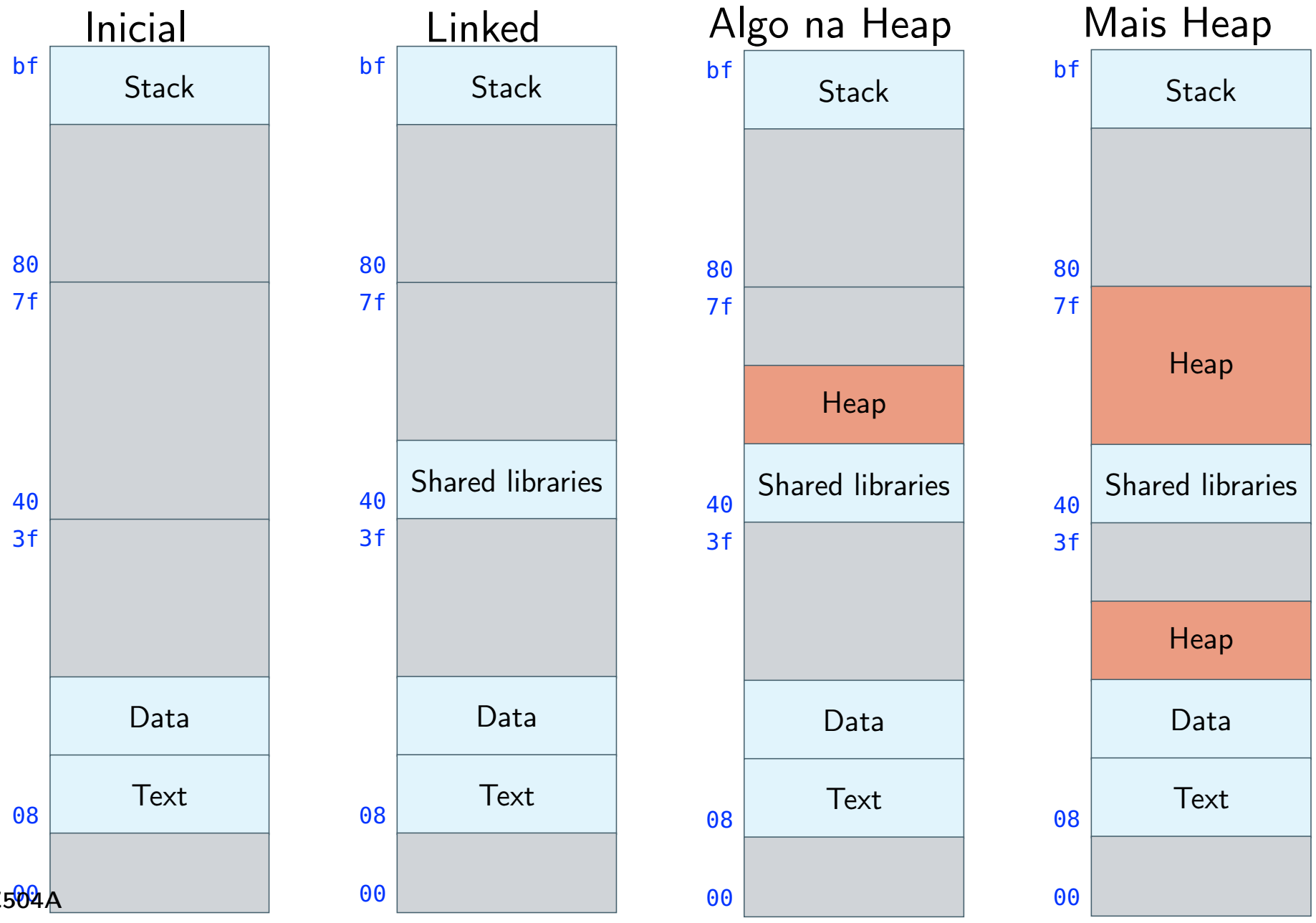


Memoria em Linux



- **Stack**
  - Pilha de tempo de execução (8 MB padrão)
- **Heap**
  - Armazenamento atribuído dinamicamente
  - Administrado por `malloc()`, `calloc()`, new
- **Shared/Dynamic Libraries** ou **Shared Objects**
  - Funções de bibliotecas (e.g., `printf()`, `malloc()`)
  - Enlaçados (link) ao código objetivo quando se executa por primeira vez
  - Windows tem *DLL*, Linux *.so*
- **Data**, **BSS**
  - Dados atribuídos estaticamente (BSS, block started by symbol, inicializa tudo em zero)
- **Text**, **RODATA**
  - Text: instruções executáveis de máquina
  - RODATA: Read-only (e.g., `const`), strings literais

Memoria em Linux



Segmentos de memória

```c
#include <stdio.h>

int main(void) {
  return 0;
}
```

Em x86-64 usar `gcc -m32`

```
$ gcc memory.c -o memory
$ size memory
text  data  bss   dec  hex  filename
1524   300    4  1828  724  memory
```

bss: *block starting symbol*

Segmentos de memória

```c
#include <stdio.h>
// var não inicializada em
    bss
int global;

int main(void) {
  return 0;
}
```

```
$ gcc memory.c -o memory
$ size memory
text  data  bss   dec  hex  filename
1524   300    8  1832  728  memory
```

Segmentos de memória

```c
#include <stdio.h>
// var não inicializada em
    bss
int global;

int main(void) {
  // var não inicializada em
       bss
  static int i;
  return 0;
}
```

```
$ gcc memory.c -o memory
$ size memory
text  data  bss   dec  hex  filename
1524   300   12  1836  72c  memory
```

Segmentos de memória

```c
#include <stdio.h>
// var não inicializada em
    bss
int global;

int main(void) {
  // var inicializada em
       data
  static int i=100;
  return 0;
}
```

```
$ gcc memory.c -o memory
$ size memory
text  data  bss   dec  hex  filename
1524   304    8  1836  72c  memory
```

Segmentos de memória

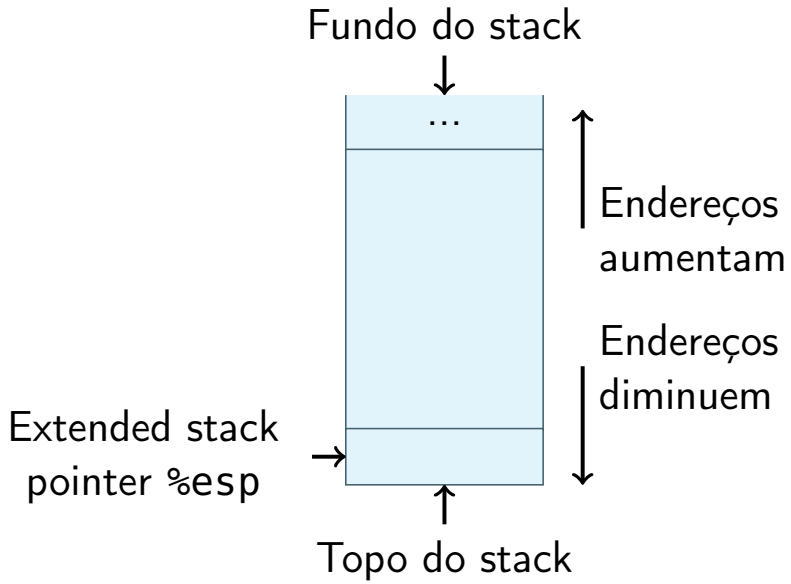
```c
#include <stdio.h>
// var inicializada em data
int global=20;

int main(void) {
  // var inicializada em
       data
  static int i=100;
  return 0;
}
```

```
$ gcc memory.c -o memory
$ size memory
text   data   bss    dec   hex   filename
1524    308     4   1836   72c   memory
```

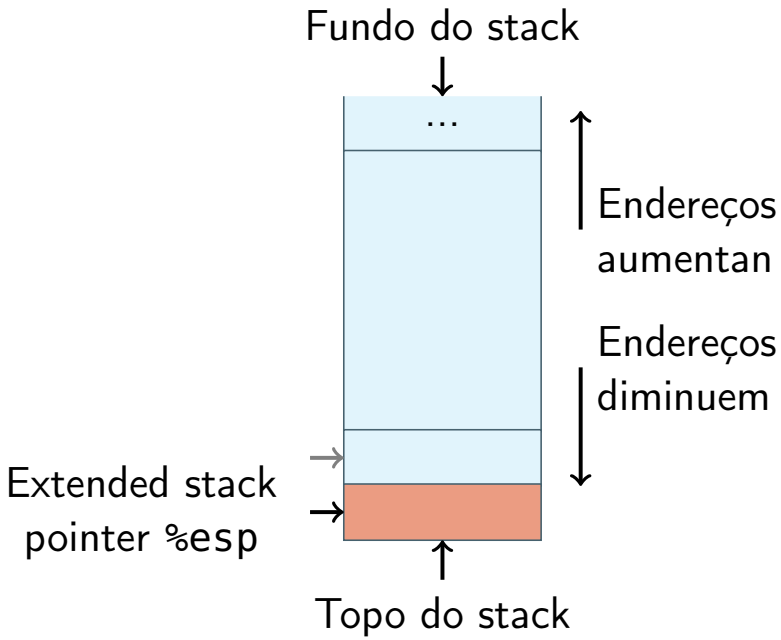
Stack

- As regiões de memória se administram como um stack
- *Cresce* para os endereços de menor memória
- O registrador `%esp` indica o menor endereço no stack
  - Endereço no topo do stack
  - Ponteiro do stack



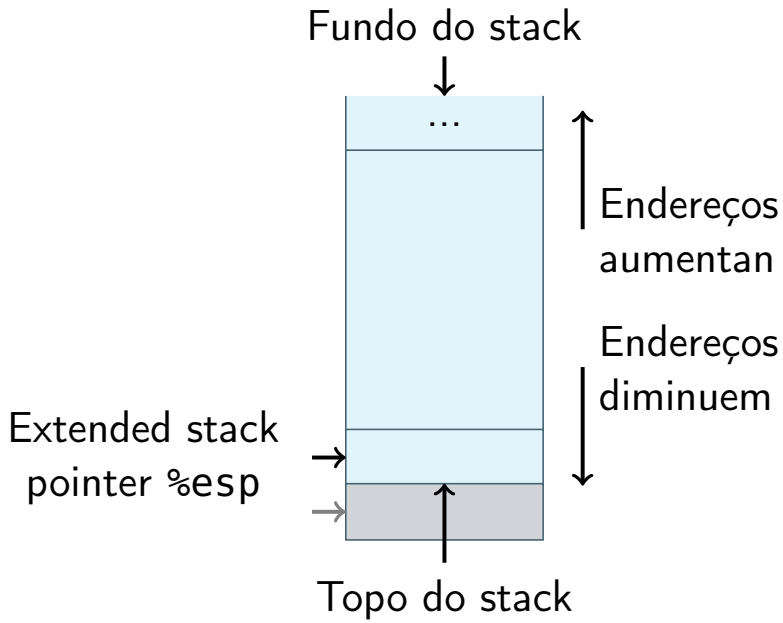
Stack Push

- `pushl src`
- Obtém o operando desde `src`
  - Talvez um registrador: `%ebp`
  - Talvez memória: `8(%ebp)`
- Diminui `%esp` por 4 (por que?)
- Armazena o operando na memória no endereço apontado por `%esp`

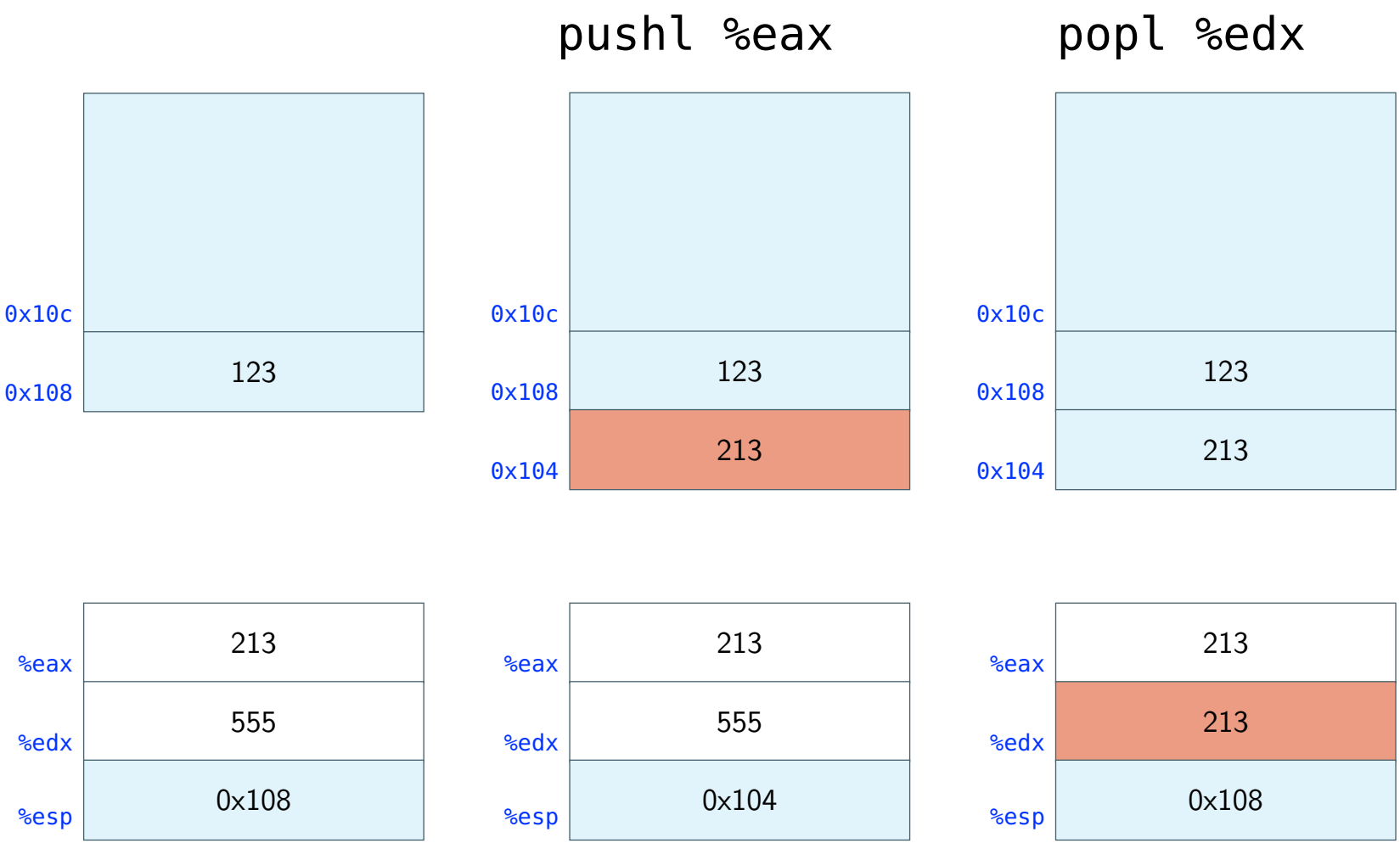


Stack Pop

- `popl dst`
- Lê o endereço de memória apontado por `%esp`
- Incrementa `%esp` por 4 (agora sabem porque)
- Armazena o lido no operando `dst`



Exemplo de operações no stack



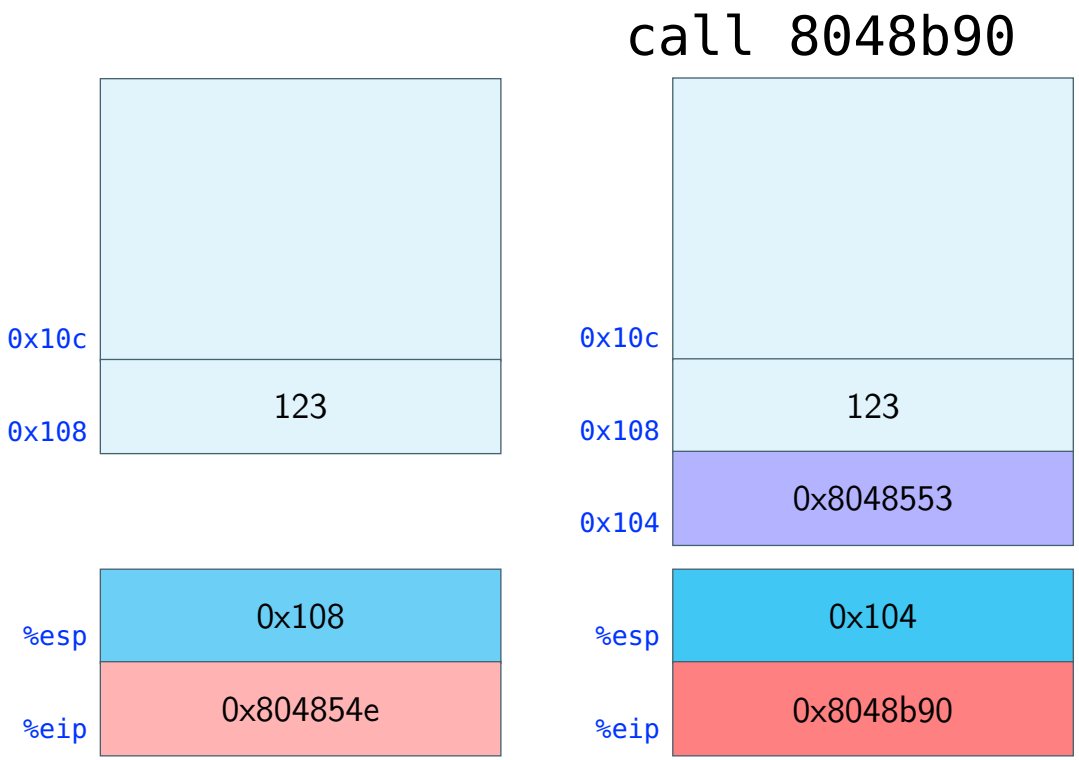
Controle de fluxo: Funções

- Usamos o stack para ajudar à chamada e retorno de funções
- Chamada a função: `call label`
  - Faz `push` ao endereço de retorno
  - Pula para o endereço de `label`
- Endereço de retorno
  - Endereço da instrução depois de `call`
  - Exemplo de dessasembly
    - `804854e: e8 3d 06 00 00 call 8048b90 <main>`
    - `8048553: 50 pushl %eax`
  - Endereço de retorno: `0x08048553`
- Retorno da função
  - `ret` pop ao endereço no stack
  - Pula para o endereço obtido

Exemplo de chamada a função

Assembler

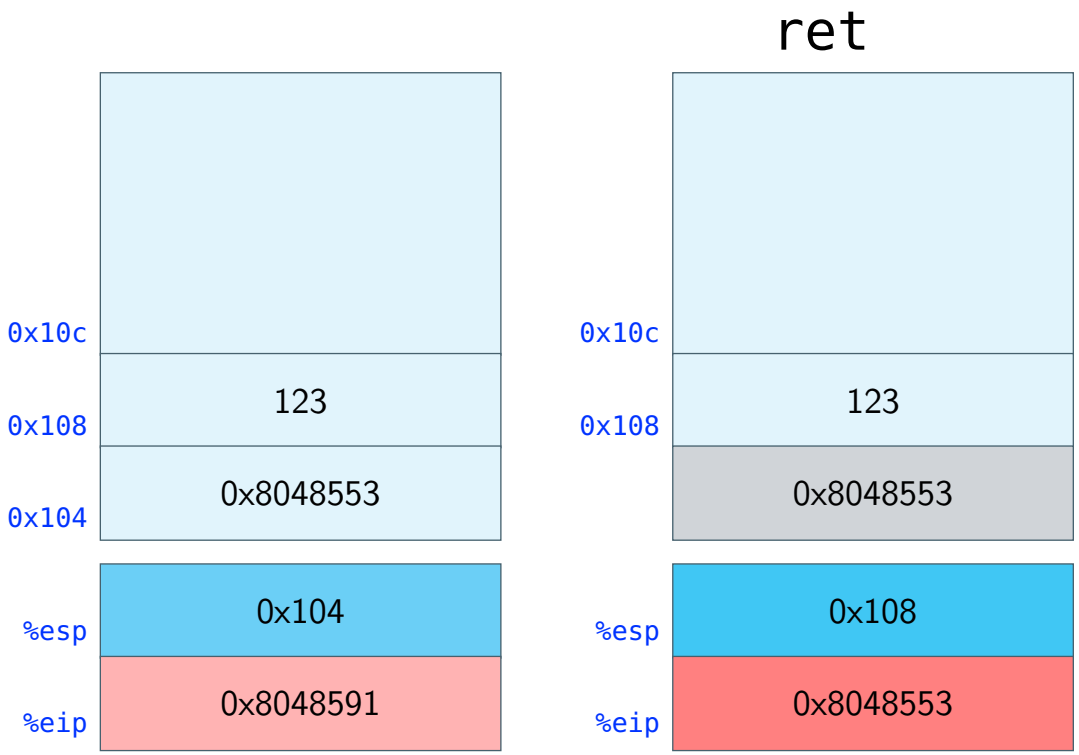
```
804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50             pushl %eax
```



Exemplo de término de função

Assembler

```
8048591: c3                    ret
```



Linguagens baseados no stack

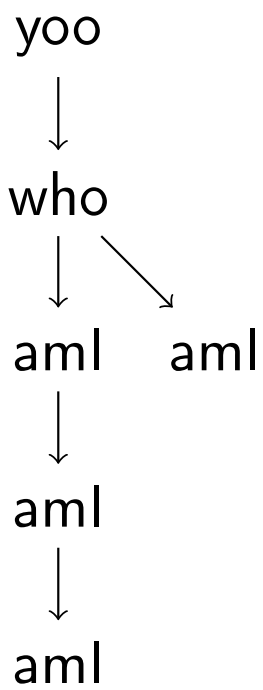
- Linguagens que suportam recursão
  - e.g., C, Pascal, Java
  - Código deve poder ser reentrante
    - Múltiplas instâncias da mesma função devem poder existir ao mesmo tempo
  - Precisam de algum lugar para armazenar o estado de cada instância
    - Argumentos
    - Variáveis locais
    - Ponteiros de retorno (tal vez)
    - Cosas esquisitas (static links, exception handling, . . .)
- Disciplina do stack —observação importante
  - O estado da função é precisada por tempo limitado
    - Desde o momento da chamada ate o retorno
  - Nota: a função chamada termina antes do quem a chamou
- O stack se armazena em *frames aninhados*
  - Estado de cada instância do processo

Exemplo de chamada em cadeia

```
yoo(...){
  ...
  who();
  ...
}
```

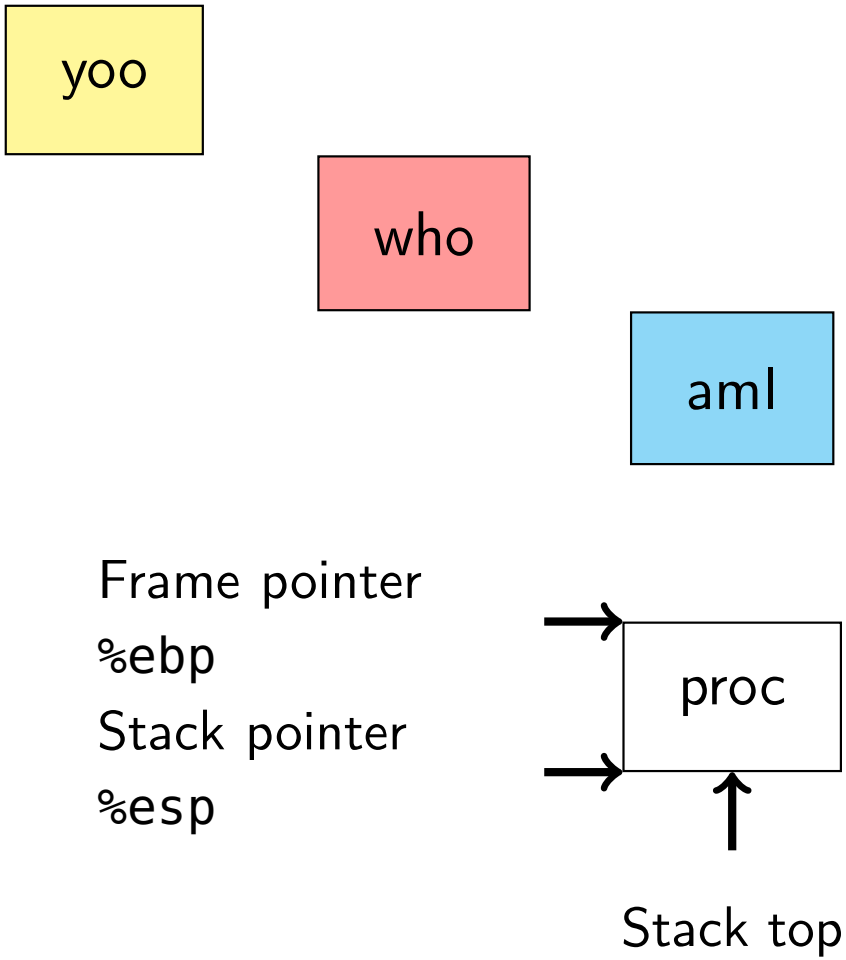
```
who(...){
  ...
  amI();
  ...
  amI();
}
```

```
amI(...){
  ...
  amI();
  ...
}
```



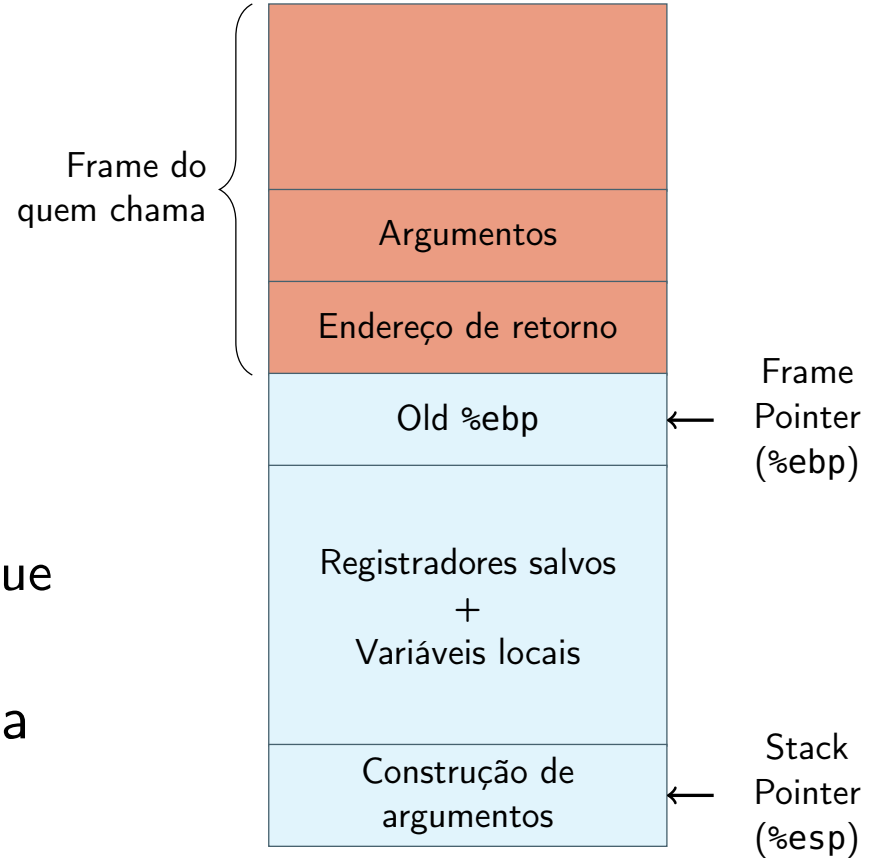
Stack frames

- Conteúdos
  - Variáveis locais
  - Informação de retorno
  - Espaço temporário
- Administração
  - Espaço atribuído quando se entra a uma função
    - *Setup* code
  - Desatribuir o espaço ao retornar
    - *Finish* code
- Ponteiros
  - Ponteiro do stack %esp indica o topo do stack
  - Ponteiro do frame %ebp indica o inicio do frame actual



IA32/Linux Stack frames

- Stack frame atual (topo no fundo)
  - Parâmetros para a função que vamos a chamar
    - Construir os argumentos
  - Variáveis locais
    - Se não cabem todas nos registradores
  - Registradores armazenados da função que chama
  - Ponteiro ao frame da função que chama
- Stack frame da função que chama
  - Endereço de retorno
    - Push pela instrução `call`
  - Argumentos para a chamada



swap()

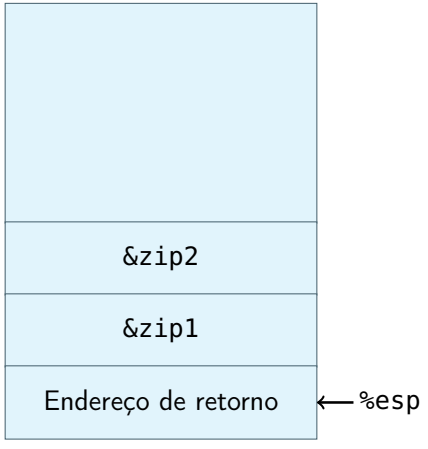
```
int zip1 = 15213;
int zip2 = 91125;

void call_swap(){
  swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp){
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Chamando swap desde call_swap

```
call_swap:
  ...
  pushl $zip2 ;Global var
  pushl $zip1 ;Global var
  call swap
  ...
```

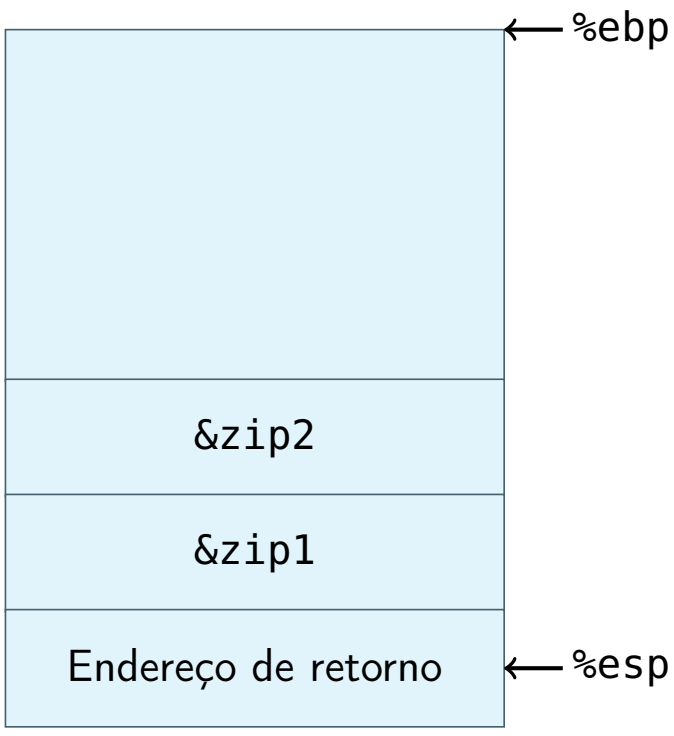


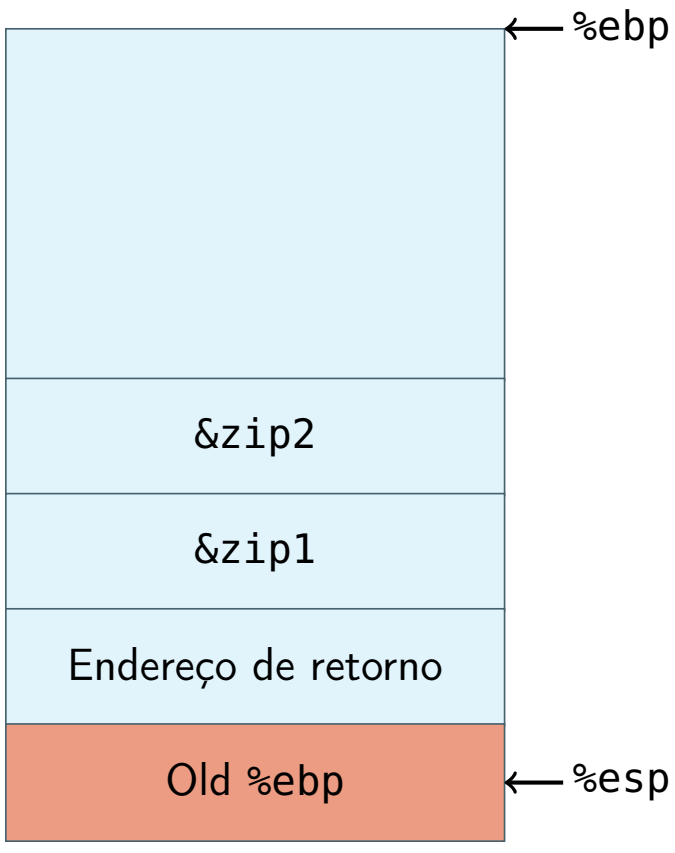
swap()

```c
void swap(int *xp, int *yp){
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  ;setup
  pushl %ebp
  movl  %esp, %ebp
  pushl %ebx
  ;body
  movl 12(%ebp), %ecx
  movl 8(%ebp), %edx
  ;core
  movl (%ecx), %eax
  movl (%edx), %ebx
  movl %eax, (%edx)
  movl %ebx, (%ecx)
  ;finish
  movl -4(%ebp), %ebx
  movl %ebp, %esp
  popl %ebp
  ret
```

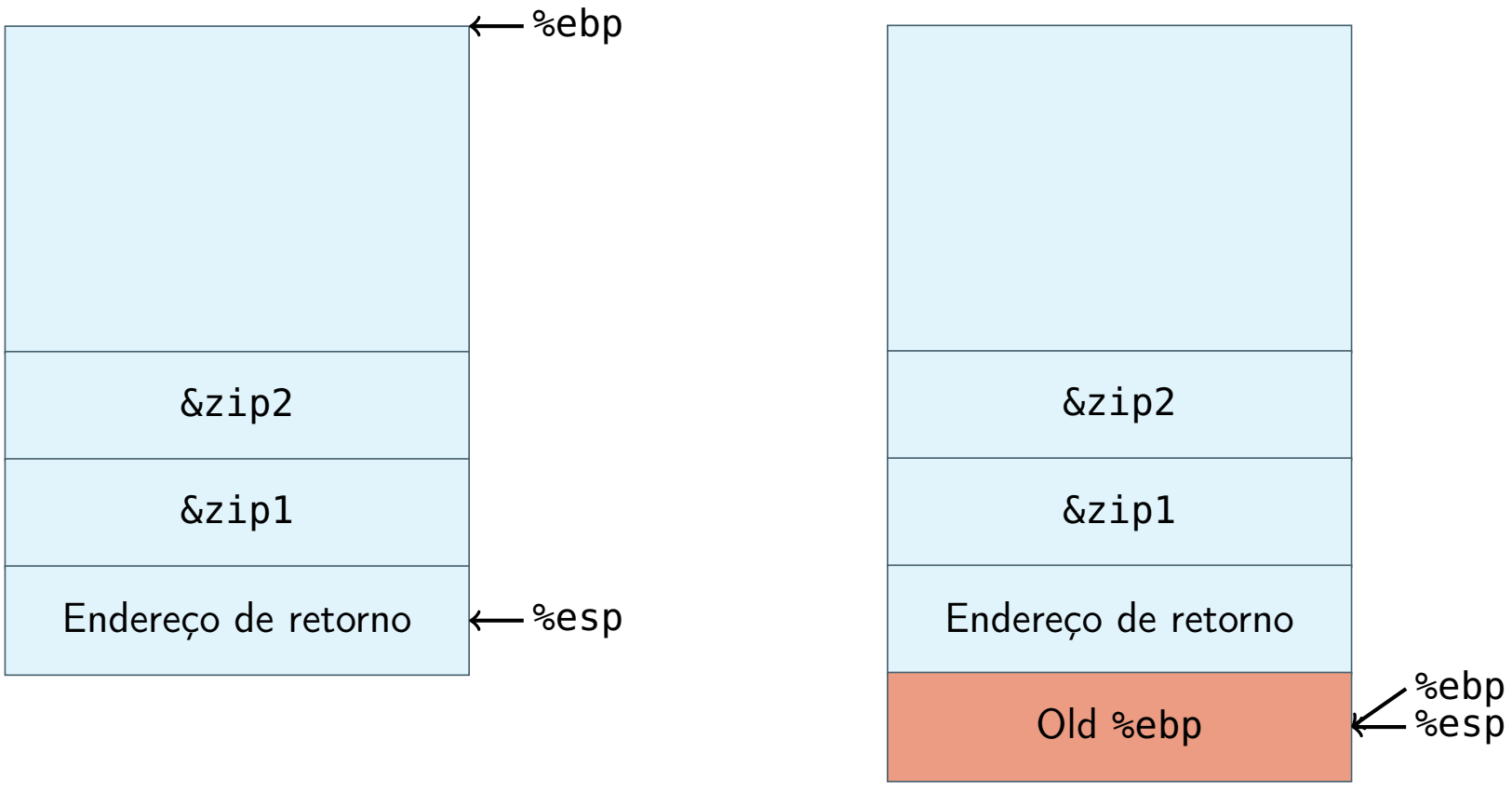
swap()





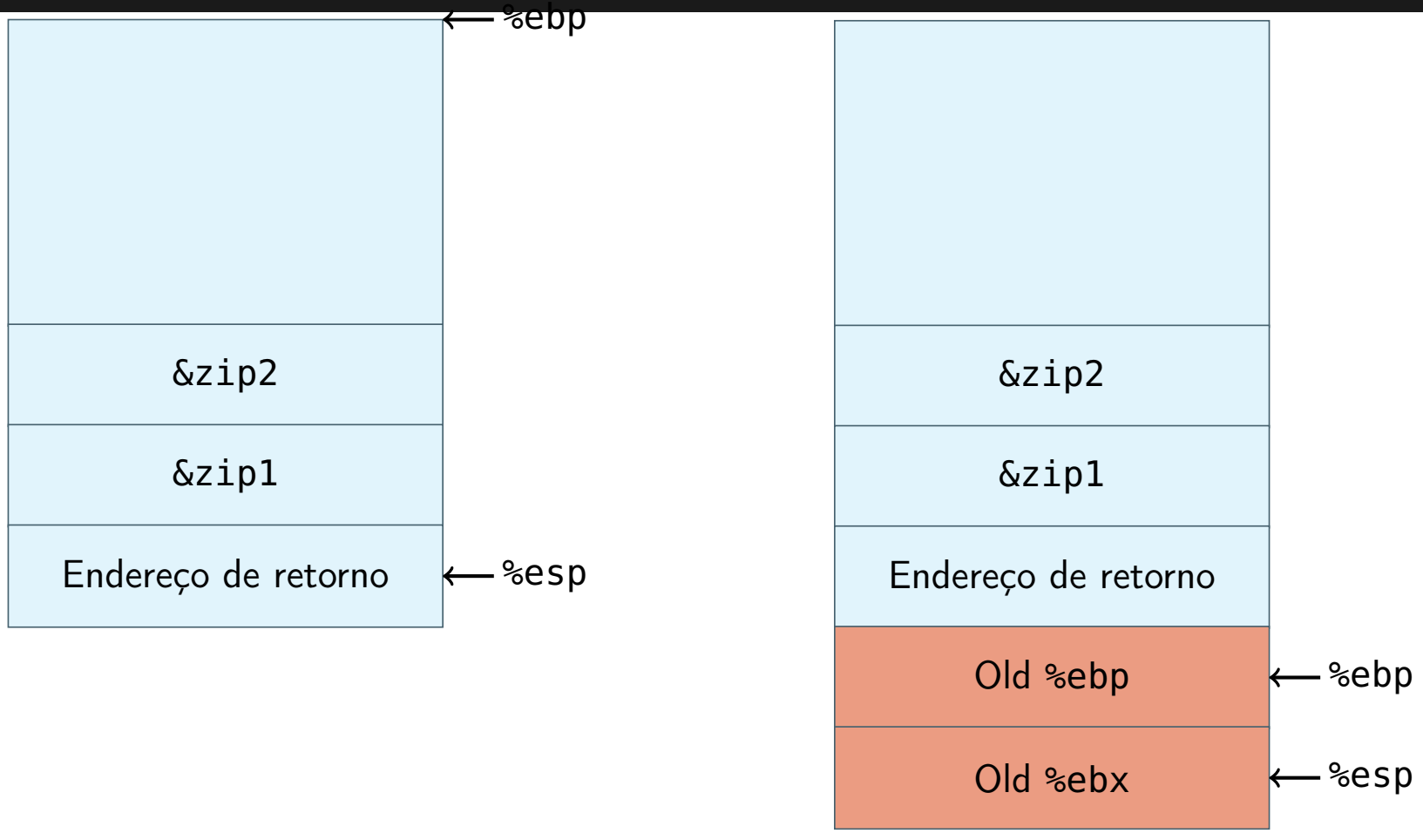

```
swap:
  ;setup
  pushl %ebp
  movl  %esp, %ebp
  pushl %ebx
```

swap()



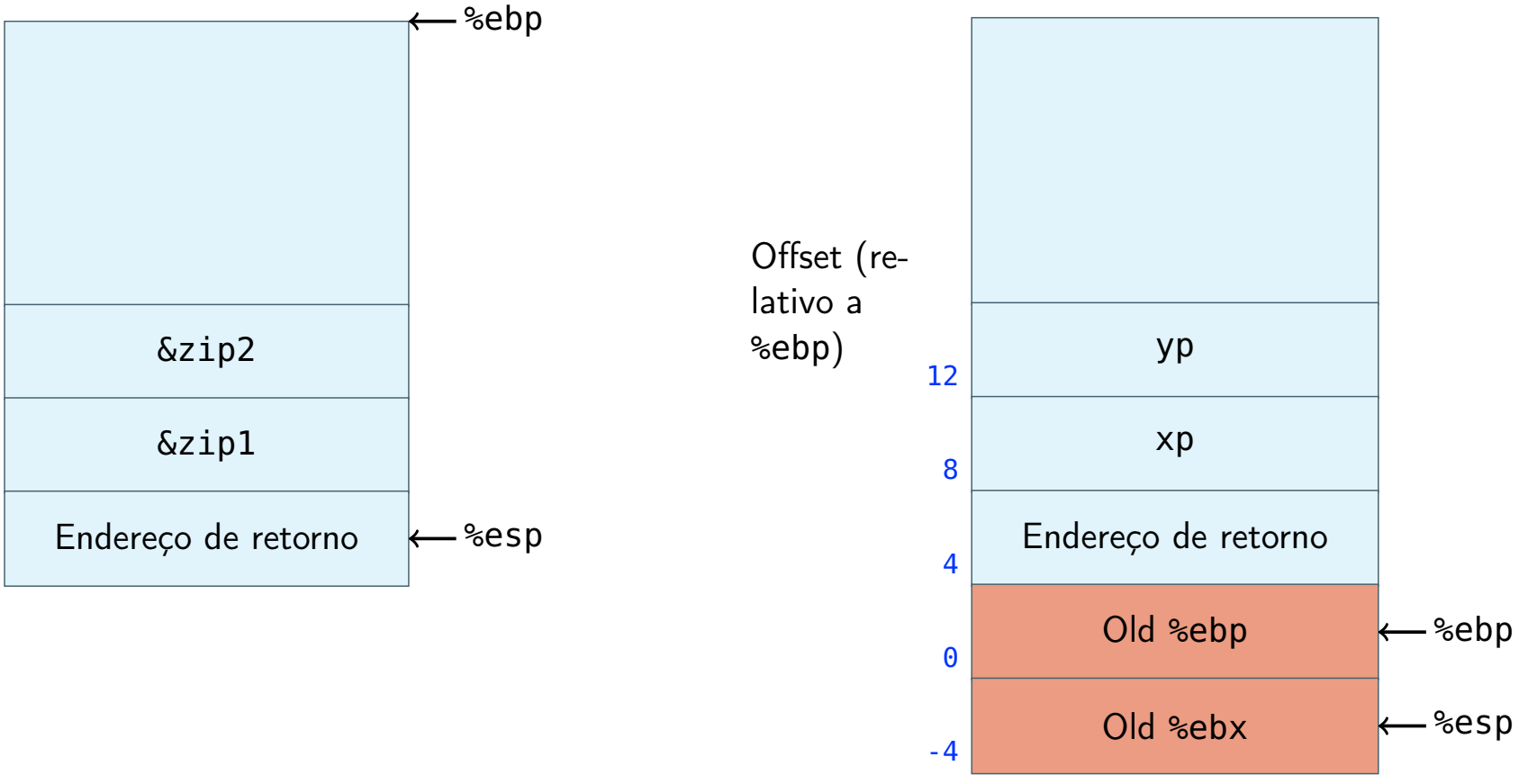

```
swap:
  ;setup
  pushl %ebp
  movl  %esp, %ebp
  pushl %ebx
```

swap()



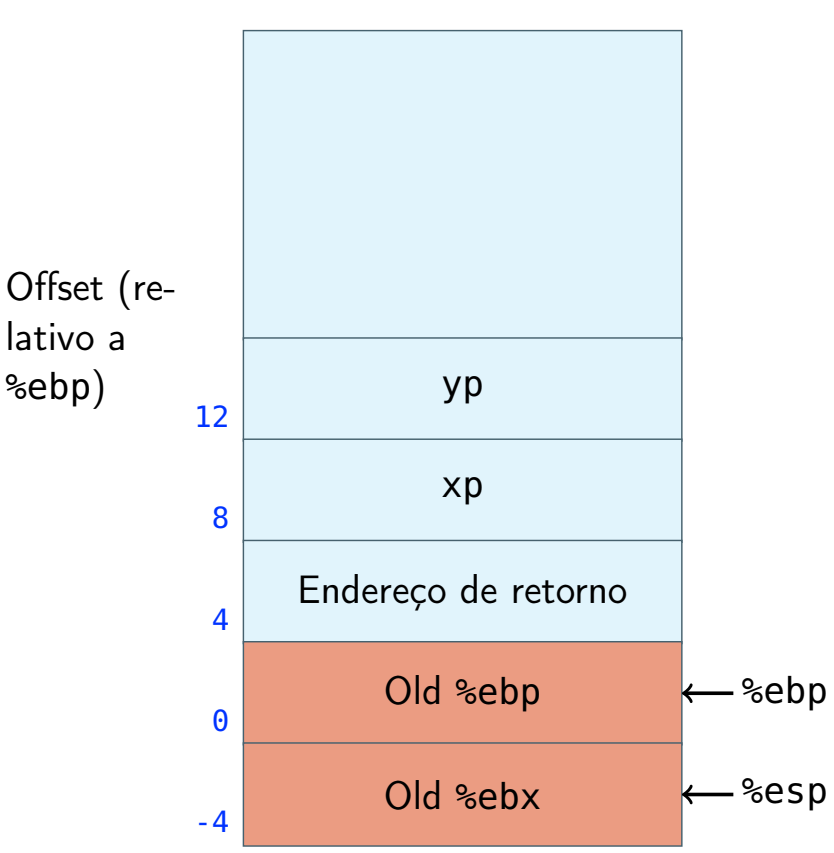

```
swap:
  ;setup
  pushl %ebp
  movl  %esp, %ebp
  pushl %ebx
```

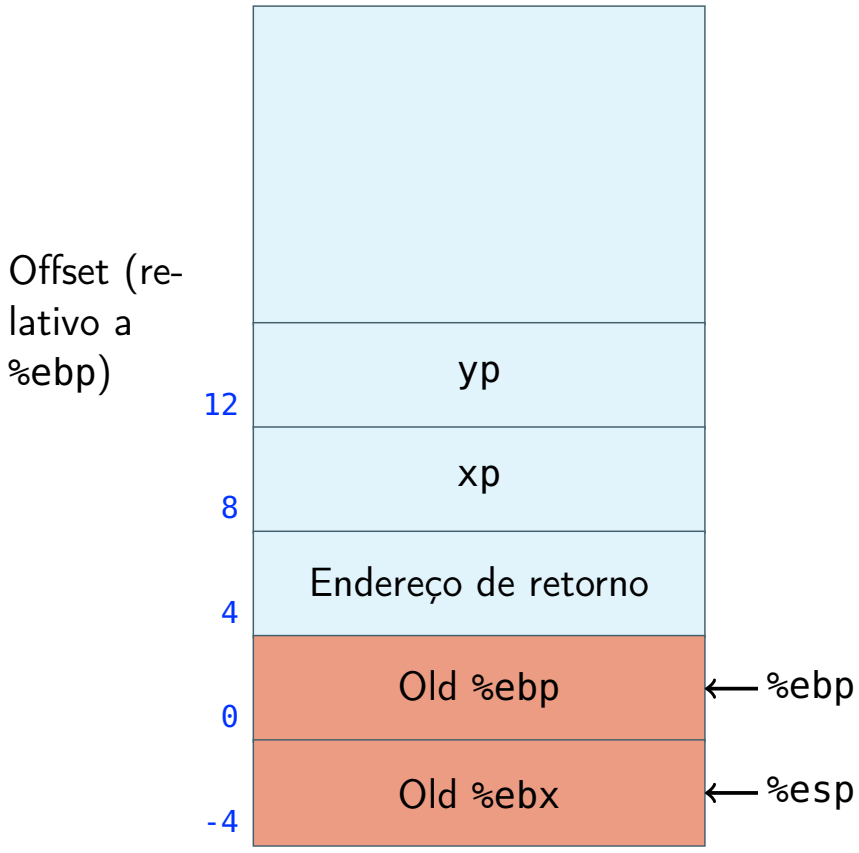
Efeito de swap() setup




```
;body
movl 12(%ebp), %ecx ;get yp
movl 8(%ebp), %edx  ;get xp
```

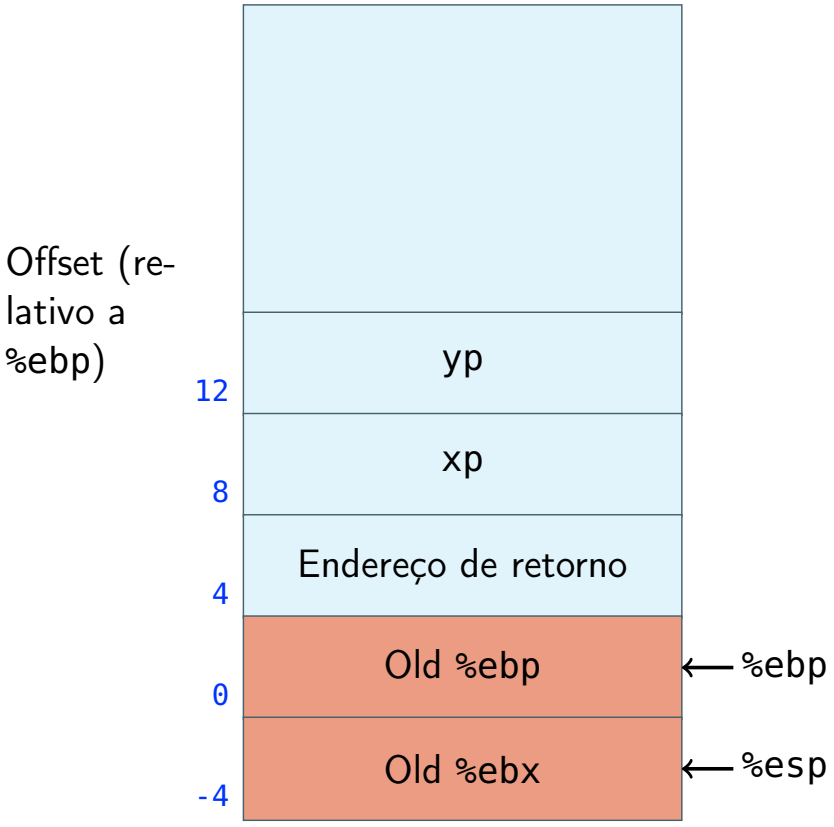
Efeito de swap() finish

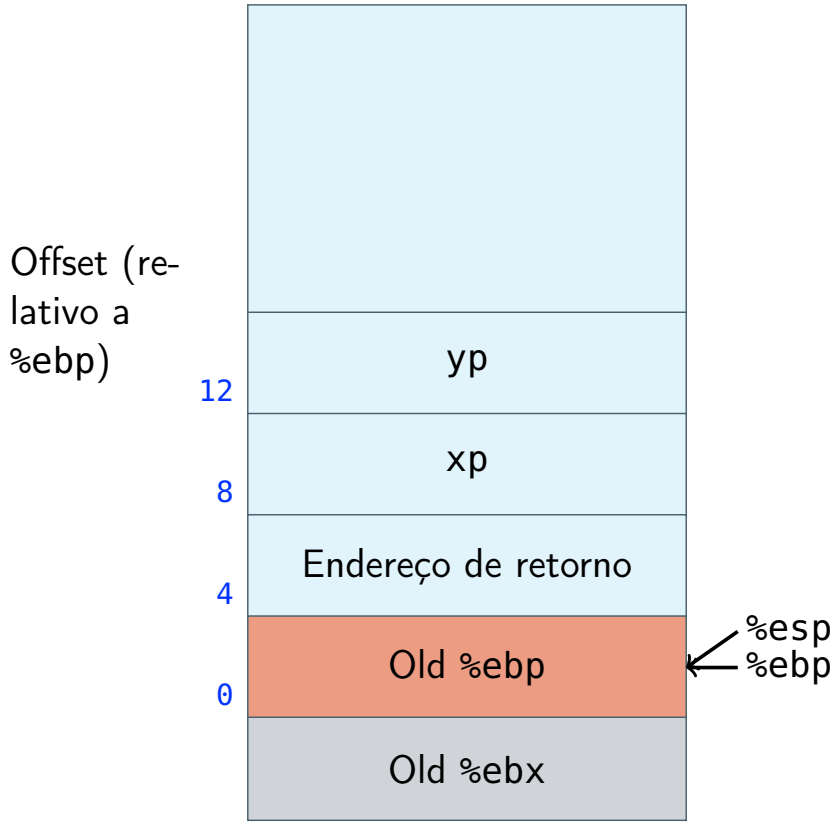





```
;finish
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

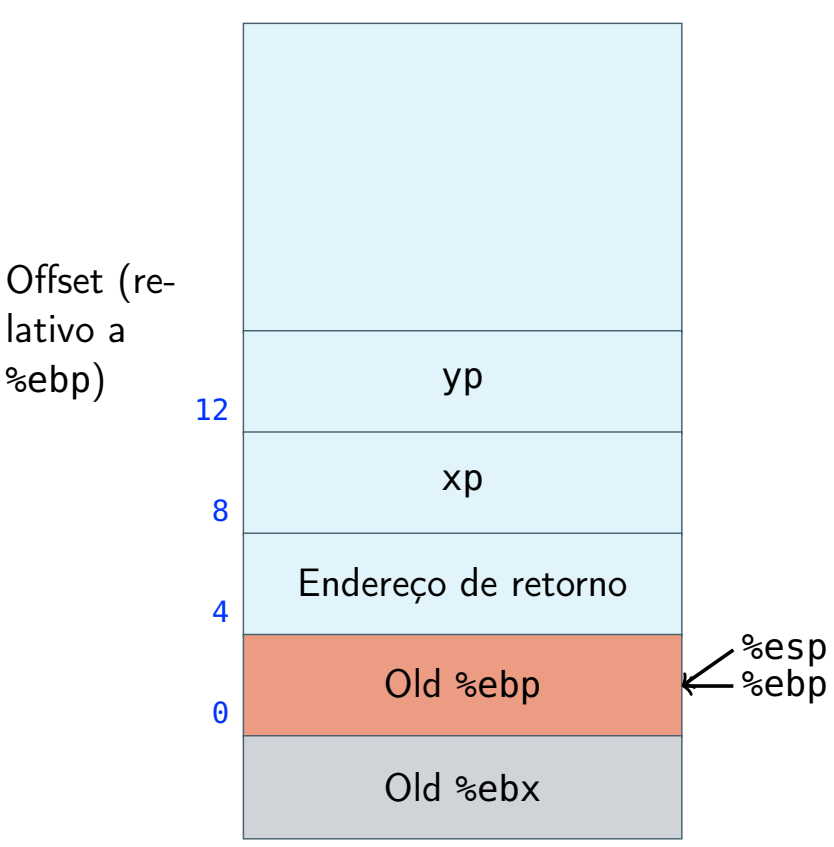
Efeito de swap() finish

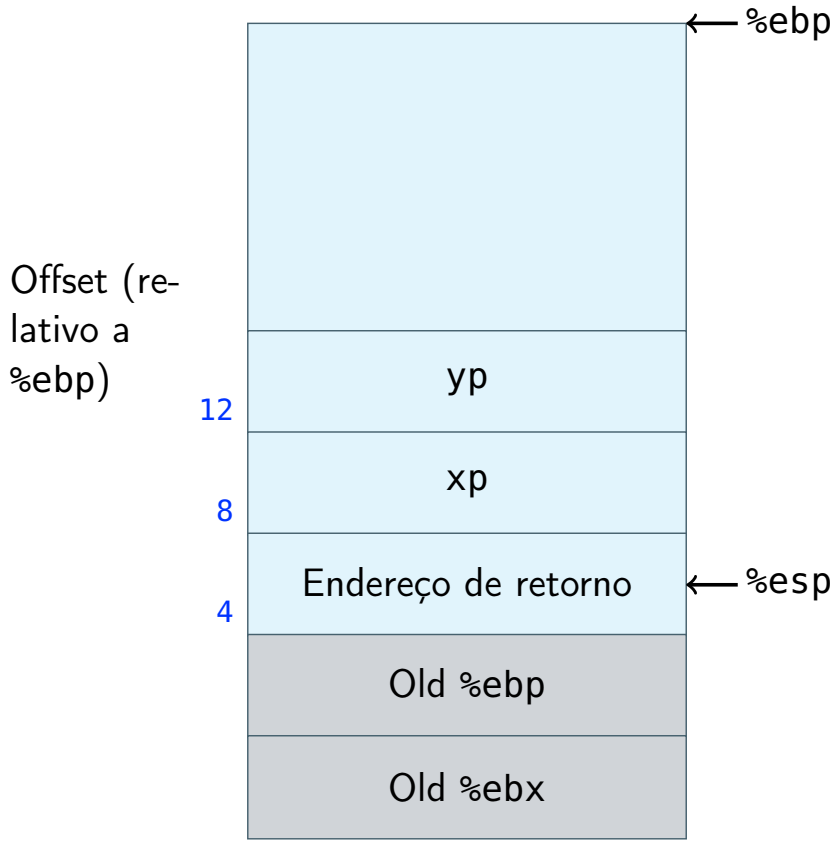





```
;finish
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

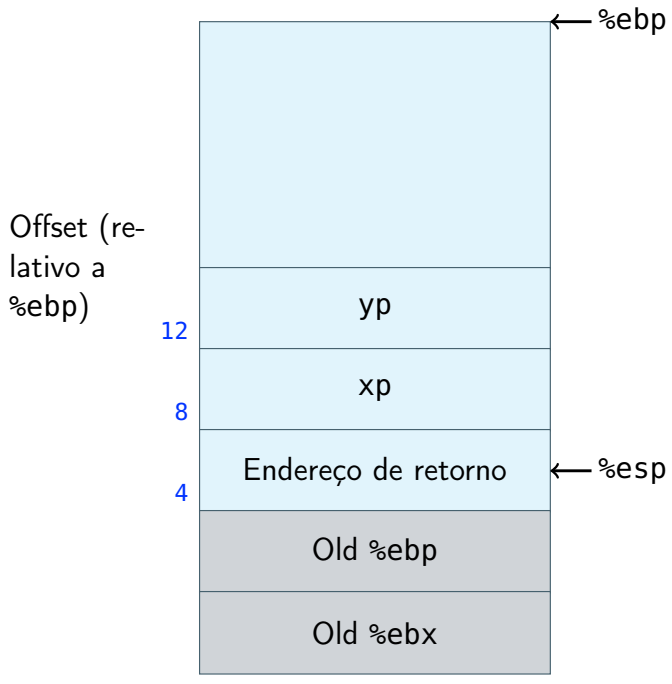
Efeito de swap() finish

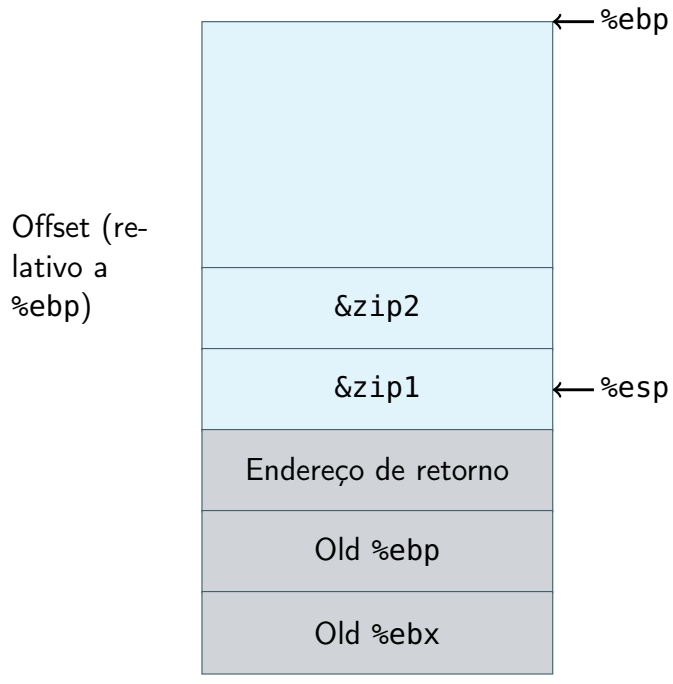





```
;finish
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

Efeito de swap() finish






```
;finish
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

- O registrador da função que chamou %ebx é armazenado e re-estabelecido
- Não foi feito para os registradores %eax, %ecx, o %edx

Convenção para armazenar em registradores

- Quando a função `yoo()` chama a `who()`
- Pode um registrador utilizar-se para armazenamento de memória temporária?

```
yoo:
  ...
  movl $15213, %edx
  call who
  addl %edx, %eax
  ...
  ret
```

```
who:
  ...
  movl 8(%ebp), %edx
  addl $91125, %edx
  ...
  ret
```

Convenção para armazenar em registradores

- Quando a função yoo() chama a who()
- Pode um registrador utilizar-se para armazenamento de memória temporária?

```
yoo:
  ...
  movl $15213, %edx
  call who
  addl %edx, %eax
  ...
  ret
```

```
who:
  ...
  movl 8(%ebp), %edx
  addl $91125, %edx
  ...
  ret
```

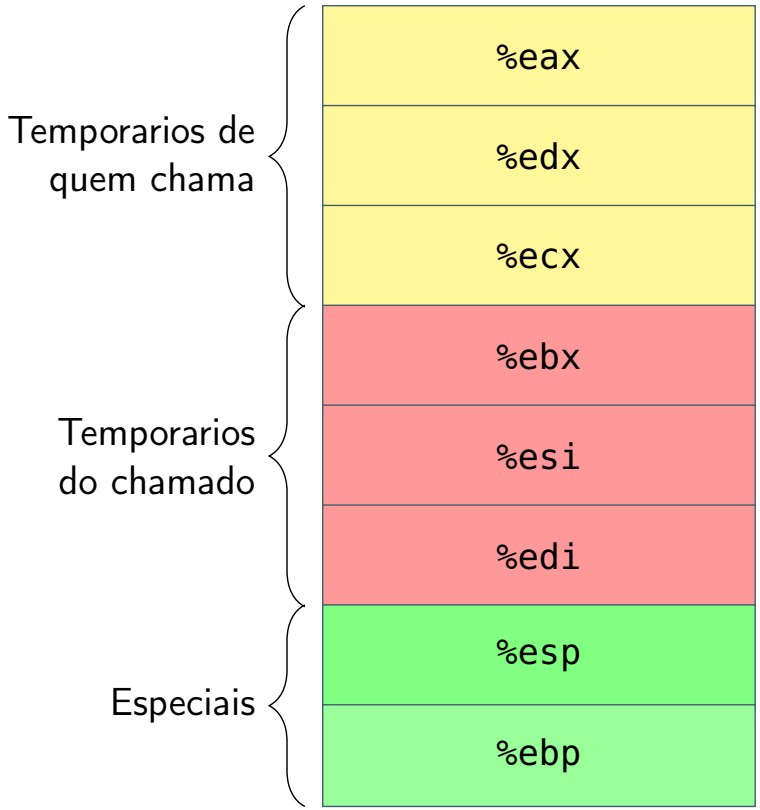
- O conteúdo do registrador %edx é sobre-escrito por who()

Convenção para armazenar em registradores

- Quando a função `yoo()` chama a `who()`
- Pode um registrador utilizar-se para armazenamento de memória temporária?
- Definições
  - *Registrador de quem chama*: a função que chama salva temporariamente os registradores em seu frame antes de chamar
  - *Registradores do chamado*: a função chamada salva temporariamente os registradores antes de usar
- Convenção
  - Quais registradores são para salvar a informação do quem chama e o chamado?

Uso de registradores IA32/Linux

- Registradores de inteiros
  - Dois têm usos especiais
    - `%ebp`, `%esp`
  - Três se usam para salvar na função chamada
    - `%ebx`, `%esi`, `%edi`
    - Os valores anteriores se colocam no stack antes de usar-se
  - Três para salvar na função que chama
    - `%eax`, `%edx`, `%ecx`
  - O registrador `%eax` armazena o valor de retorno



Resumo do Stack

- O stack faz que a recursão funcione
  - Armazenamento privado para cada instância da função (cada chamada)
    - Distintas instâncias não interferem entre elas
    - Endereços locais e argumentos são relativos à posição do stack
  - Podem administrar-se como um stack
    - As funciones retornam no ordem inverso das chamadas
- Funções IA32: instruções e convenções
  - `call` e `ret` misturam `%eip` e `%esp` de uma maneira estabelecida
  - Convenção do uso de registradores
    - Armazenamento do que chama e o chamado
    - `%ebp` e `%esp`
  - Convenção para organizar o stack frame
    - Qual argumento é inserido na pilha antes?

Antes e depois do `main()`

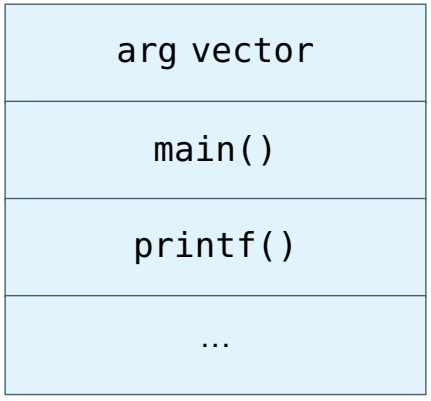
```c
int main(int argc, char *argv[]){
  if (argc > 1) {
    printf("%s\n", argv[1]);
  } else {
    char* av[3] = {0, 0, 0};
    av[0] = argv[0];
    av[1] = "Fred";
    execvp(av[0], av); // carrega o primeiro argumento e sobre-escreve
        o código de esta instância, e passa os parâmetros do segundo
        argumento---podemos fazer um shell usando esta instrução
  }
  return (0);
}
```

Partes misteriosas

- `argc`, `argv`
  - Strings do programa
  - Disponíveis quando outro programa está-se executando
  - Onde estão na memória?
  - Como chegaram lá?
- O que acontece quando `main()` faz `return(0)`?
  - Já não há mais programa que executar, certo?
  - O que acontece com o `0`?
  - Como chega a onde vá?

Não tão misteriosas

- `argc`, `argv`
  - Strings do programa
  - Disponíveis quando outro programa se está executando
- A transferência de informação entre processos é tarefa do sistema operacional
  - O sistema operacional copia os strings de um espaço de memória antigo a um novo quando se chama a `exec()`
  - Tradicionalmente se colocam embaixo do fundo do stack (implementações dependem do compilador, arquitetura, etc.)
  - Outras coisas esquisitas (ambiente, vector auxiliar, etc.) —arriba de `argv`



Não tão misteriosas

- O que acontece quando `main()` faz `return(0)`?
  - Definido por C para ter o mesmo comportamento que `exit(0)`
  - Como?
- O envoltório (wrapper) de `main()`
  - Recebe `argc`, `argv` do sistema operacional
  - Chama `main()`, e logo chama a `exit()`
  - Esta na biblioteca de C, tradicionalmente `crt0.s`
  - Tem um nome esquisito

```
// não é o código real, mas a ideia é esta
void main_wrapper(int argc, char* argv){
  exit( main(argc, argv) );
}
```

Pontos chave

- Os processos são instâncias de um programa
  - Têm uma forma definida (convenção) em memória
  - Se administram como um stack
- Entendemos como evoluciona o stack em execução
- Importância dos registradores (e suas convenções)
- O wrapper do `main()`

Texto

- Cobrimos partes do capítulo 2 de OS:P+P
- Leiam antes e después da aula
- Como vem não cobrimos o texto necessariamente em ordem, e vemos coisas que não estão lá
- Além disso, falamos de coisas complexas que requerem de que tenham lido antes
- Há coisas que não falamos (que estão no livro) e o resto de conceitos requerem que as entendam (assim que vir à aula e ler também não é suficiente)

Q: Então, o que és suficiente?

A: Ler (antes e depois para terminar de compreender), assistir a aula, exercitar

Exercício

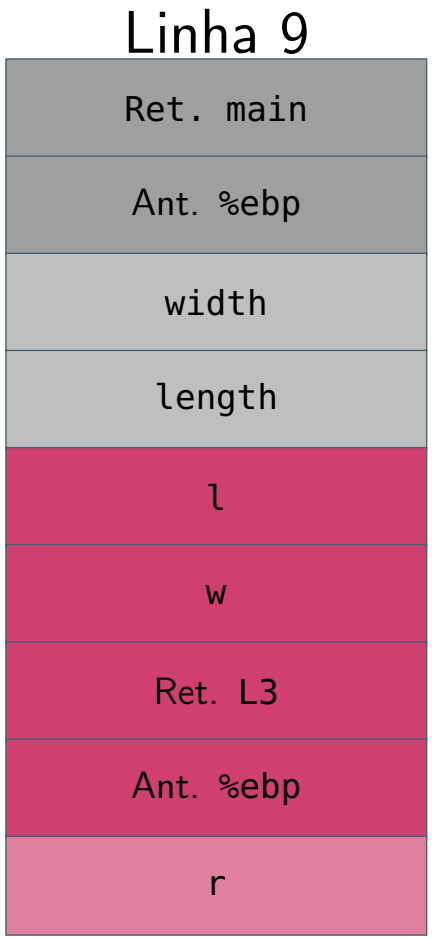
- Com a pessoa do lado
- Desenhem o stack dos códigos quando se executam
  - a linha 9
  - a linha 13

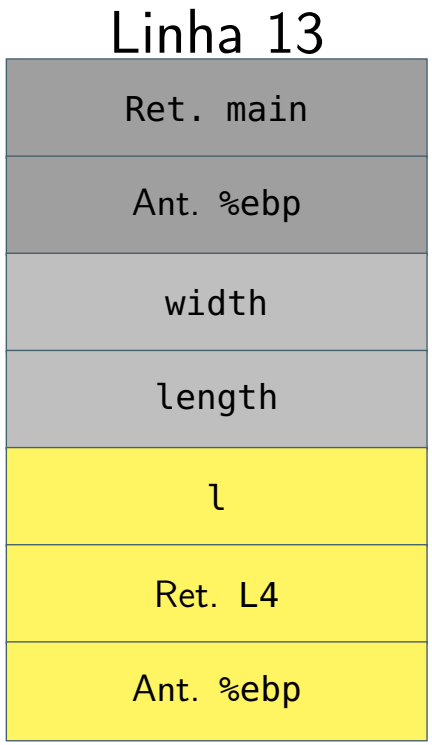
Código

```
1  int main(){
2    float length=3, width=2;
3    cout << area(length, width);
4    cout << foo(lenght);
5  }
6
7  float area(float l, float w){
8    float r = l*w;
9    return r;
10 }
11
12 float foo(float l){
13   return l+l+l+l;
14 }
```

Solução

Stack quando estamos executando





Exercício prático

Shell clone

Não precisa entregar, é trabalho independente (sem nota)

- O que fazer?
  - Escrever um clone de uma terminal de comandos usando o comando `execvp`
  - Sua aplicação deve permitir escrever comandos que existam em Linux, e executá-los desde a instância de sua aplicação
- Explique e resolva
  - Como funciona uma shell comum?
  - Como funciona o `execvp`?
  - Como pode usar `execvp` para implementar uma shell?