udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA

Stack

Adín Ramírez
adin.ramirez@mail.udp.cl

Sistemas Operativos (CIT2003-1)
1er. Semestre 2015

Adaptado de Dave Eckhardt

Objetivos de la clase

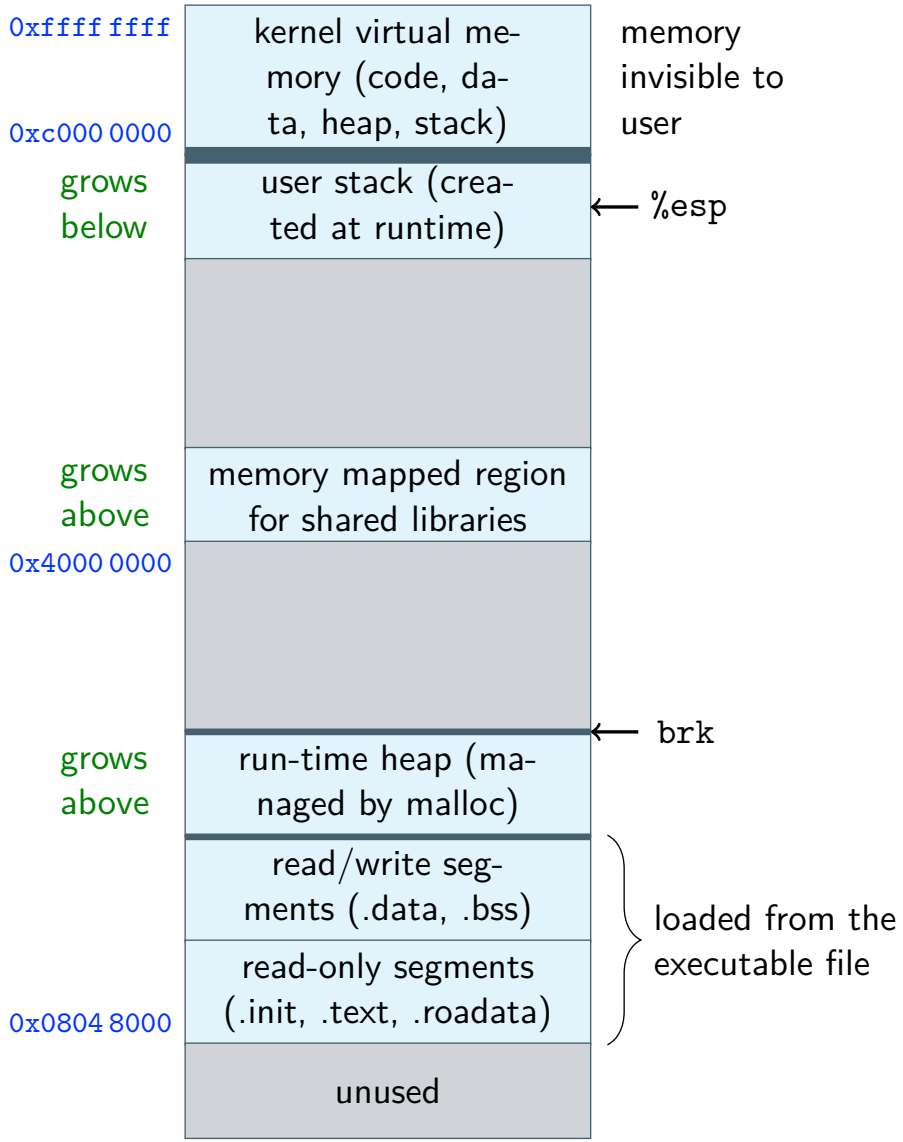
- Modelo del proceso en memoria
- Organización del stack
- Convenciones de registros
- Antes y después de `main()`

Trabajaremos con 32 bits

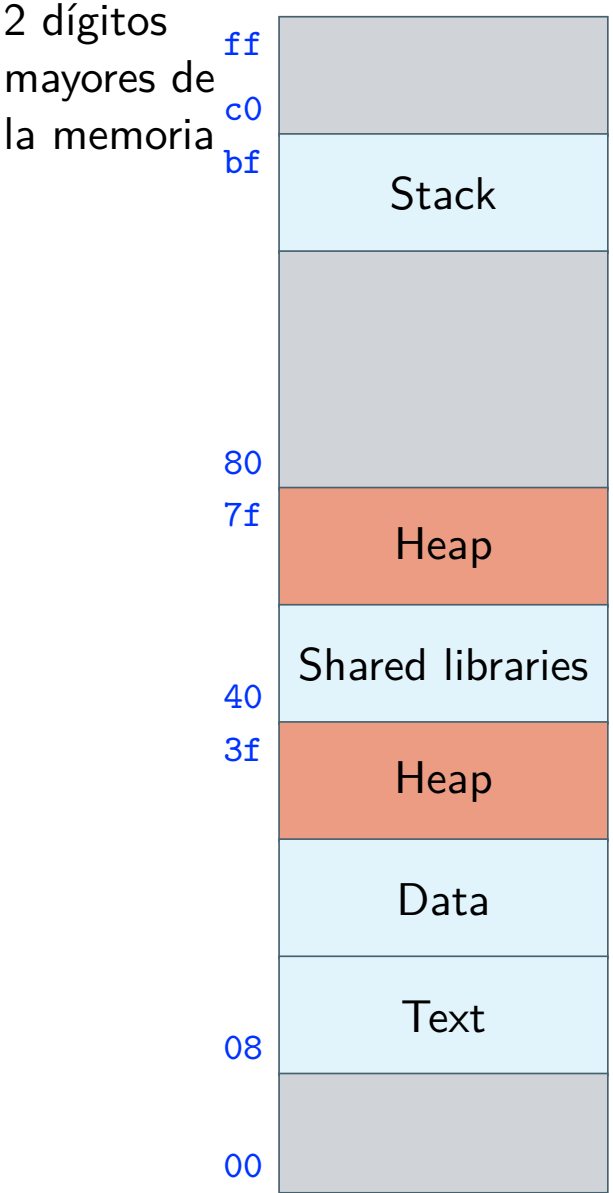
- Pueden haber aprendido en `x86-64`, a.k.a. `EMT64` a.k.a. `AMD64`
  - `x86-64` es más sencilla que `x86` para el código de programas de usuario
  - Más registros, más ortogonales
- Nosotros nos enfocaremos en `x86` (`IA32`)
  - `x86-64` no es simple para el código del kernel
    - La máquina empieza en modo de 16-bit, después en 32, y por último en 64 (necesita código de transición)
    - Las interrupciones son más complicadas
  - `x86-64` no es simple para depurar (debug)
    - Más registros significa más registros pueden tener valores incorrectos
  - la memoria virtual de `x86-64` es molesta
    - Más pasos que en la `x86-32`, pero no más estimulantes
  - Aún hay muchas máquinas de 32-bit en el mundo

Espacio privado de memoria

- Cada proceso tiene su propio espacio de memoria
- Las direcciones de memoria varían entre distintas arquitecturas
- Detalles pueden variar entre distintos sistemas operativos, y versiones de kernel
- `brk`: llamada del sistema (unix) para poder realojar de manera dinámica la memoria del segmento de datos (program break of process)

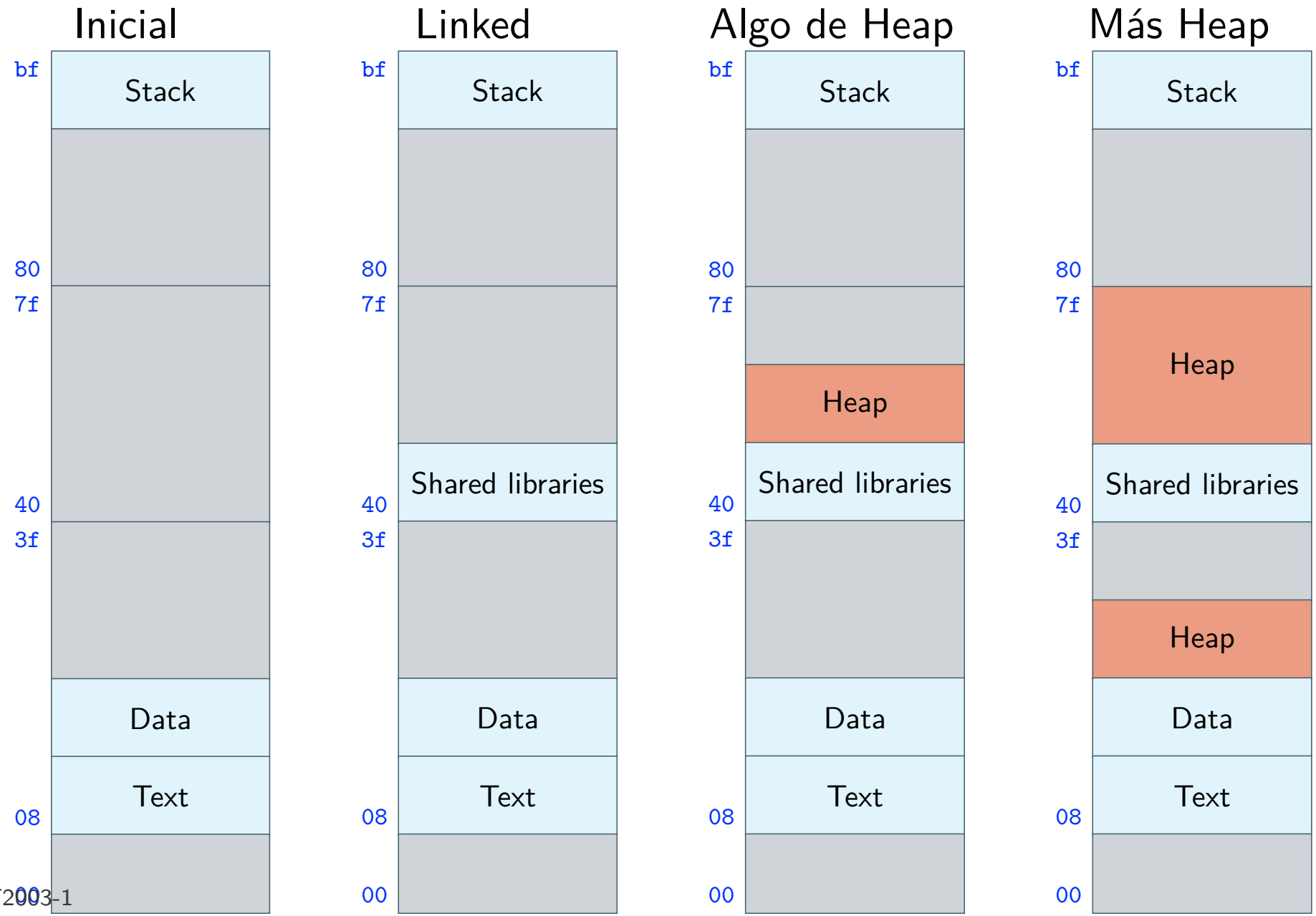


Memoria en Linux



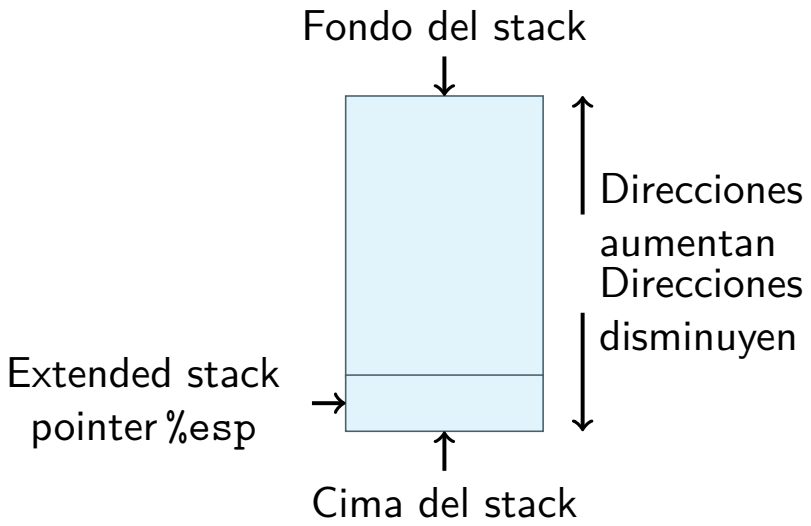
- **Stack**
  - Pila de tiempo de ejecución (8 MB por defecto)
- **Heap**
  - Almacenamiento asignado dinámicamente
  - Administrado por `malloc()`, `calloc()`, `new`
- **Shared/Dynamic Libraries** o **Shared Objects**
  - Funciones de librerias (e.g., `printf()`, `malloc()`)
  - Enlazados (link) a código objetivo cuando se ejecuta la primera vez
  - Windows tiene *DLL*, Linux *.so*
- **Data**, **BSS**
  - Datos asignados estáticamente (BSS —block started by symbol— inicializa todo en cero)
- **Text**, **RODATA**
  - Text: instrucciones ejectuables de máquina
  - RODATA: Read-only (e.g., `const`), strings literales

Memoria en Linux



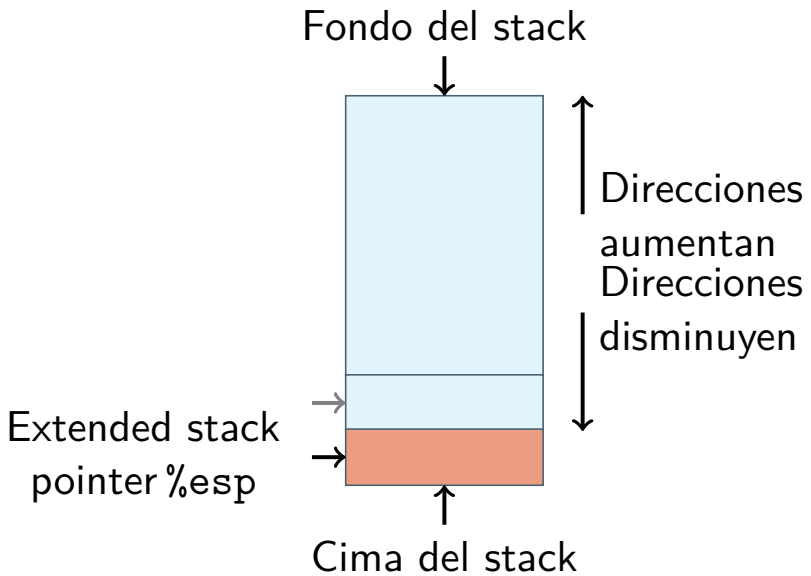
Stack

- Las regiones de memoria se administran como un stack
- *Crece* hacia las direcciones de menor memoria
- El registro %esp indica la menor dirección del stack
  - Dirección del tope del stack
  - Puntero del stack



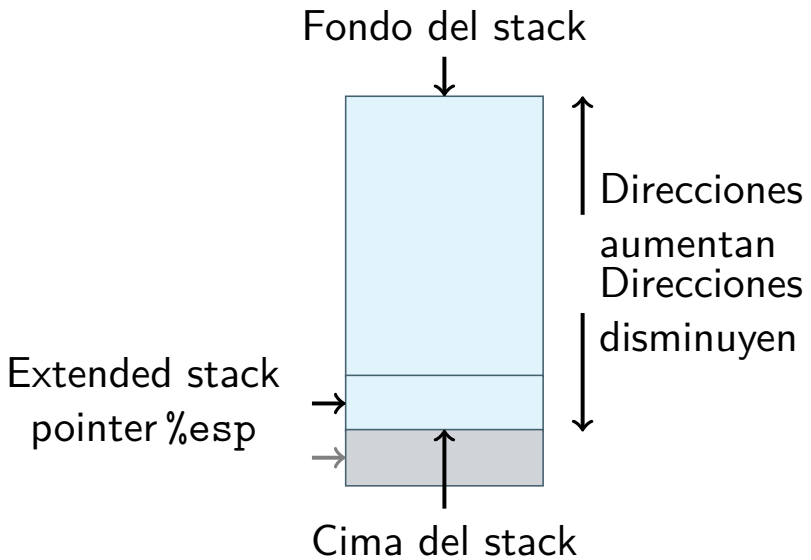
Stack Push

- `pushl src`
- Obtiene el operando desde `src`
  - Tal vez un registro: `%ebp`
  - Tal vez memoria: `8(%ebp)`
- Decrementa `%exp` por 4
- Almacena el operando en la memoria en la dirección apuntada por `%esp`

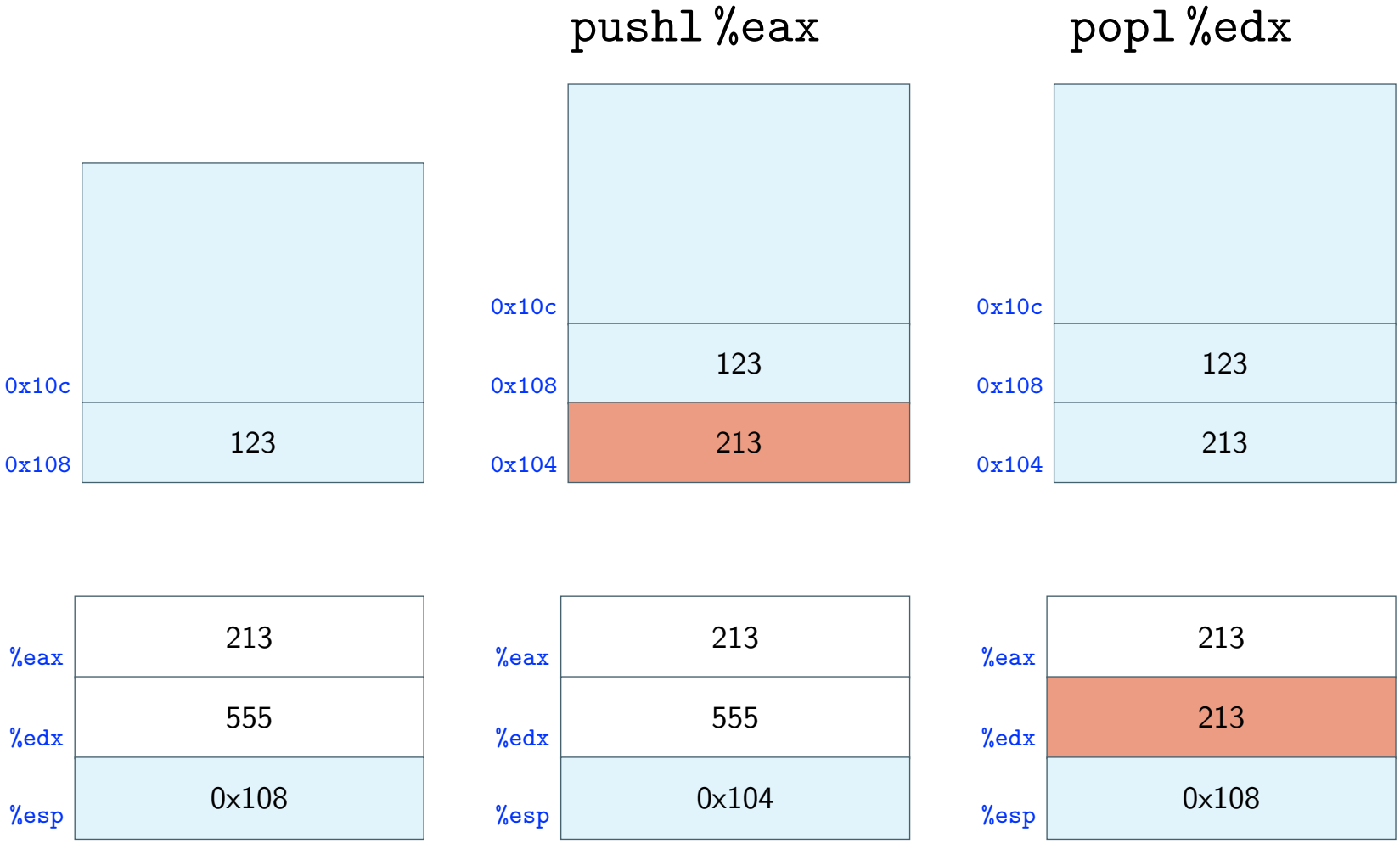


Stack Pop

- `popl dst`
- Lee la dirección de memoria apuntada por `%esp`
- Incrementa `%exp` por 4
- Almacena lo leído en el operando `dst`



Ejemplo de operaciones en el stack



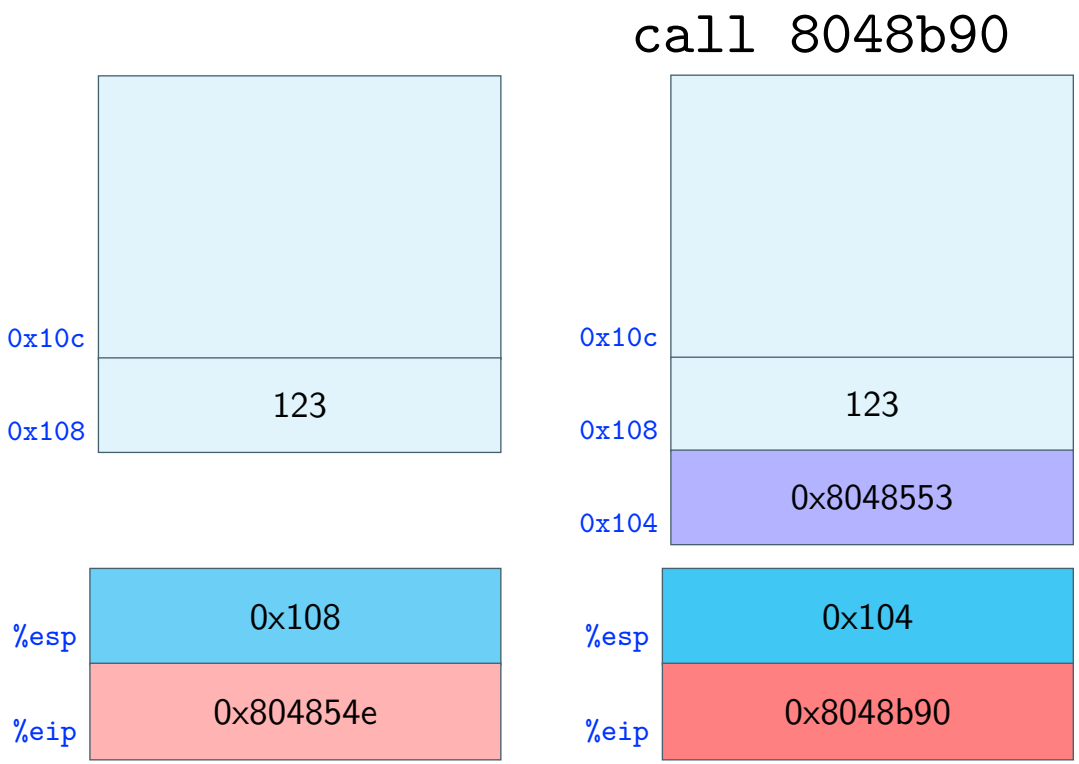
Control de flujo: Procedimientos

- Usamos el stack para ayudar la llamada y retorno de procedimientos
- Llamada a procedimiento: `call label`
  - Hace push a la dirección de retorno
  - Salta a la dirección de `label`
- Dirección de retorno
  - Dirección de la instrucción después de `call`
  - Ejemplo de desensamblado
    - `804854e: e8 3d 06 00 00 call 8048b90 <main>`
    - `8048553: 50 pushl %eax`
  - Dirección de retorno: `0x8048553`
- Retorno de procedimiento
  - `ret` pop a la dirección del stack
  - Salta a la dirección obtenida

Ejemplo de llamada a procedimiento

Assembler

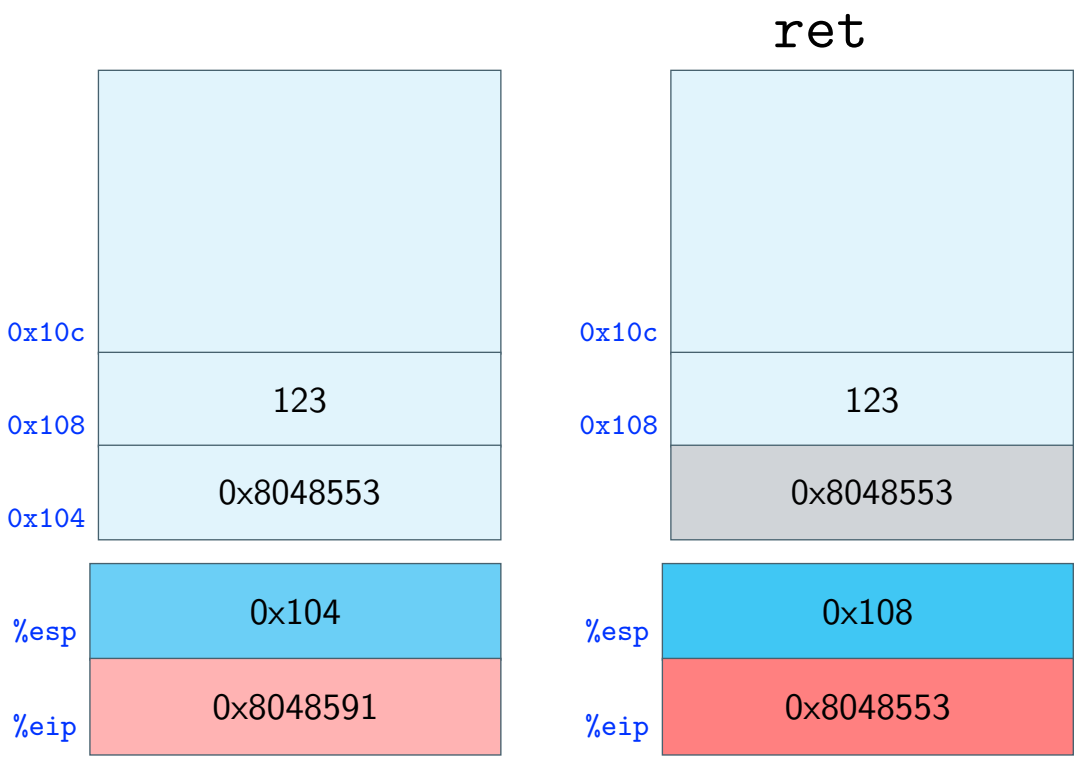
```
804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50             pushl %eax
```



Ejemplo de término de procedimiento

Assembler

```
8048591: c3                    ret
```



Lenguajes basados en el stack

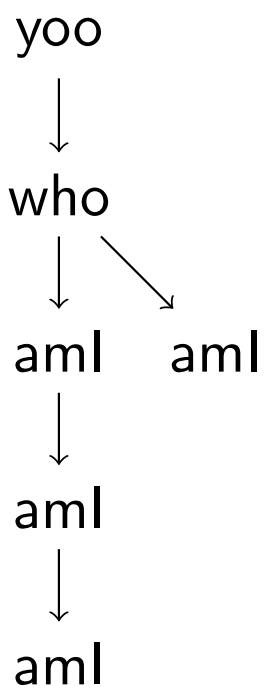
- Lenguajes que soportan recursión
  - e.g., C, Pascal, Java
  - Código debe de poder ser reentrante
    - Multiples instancias del mismo procedimiento deben poder vivir al mismo tiempo
  - Necesita algún lugar para almacenar el estado de cada instancia
    - Argumentos
    - Variables locales
    - Punteros de retorno (tal vez)
    - Cosas extrañas (static links, exception handling, . . .)
- Disciplina del stack —observación importante
  - El estado del procedimiento se necesita por tiempo limitado
    - Desde el momento de la llamada hasta el retorno
  - Nota: el procedimiento llamado termina antes que el que lo llamó
- El stack se almacena en *frames anidados*
  - Estado de cada instancia del proceso

Ejemplo de llamada en cadena

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

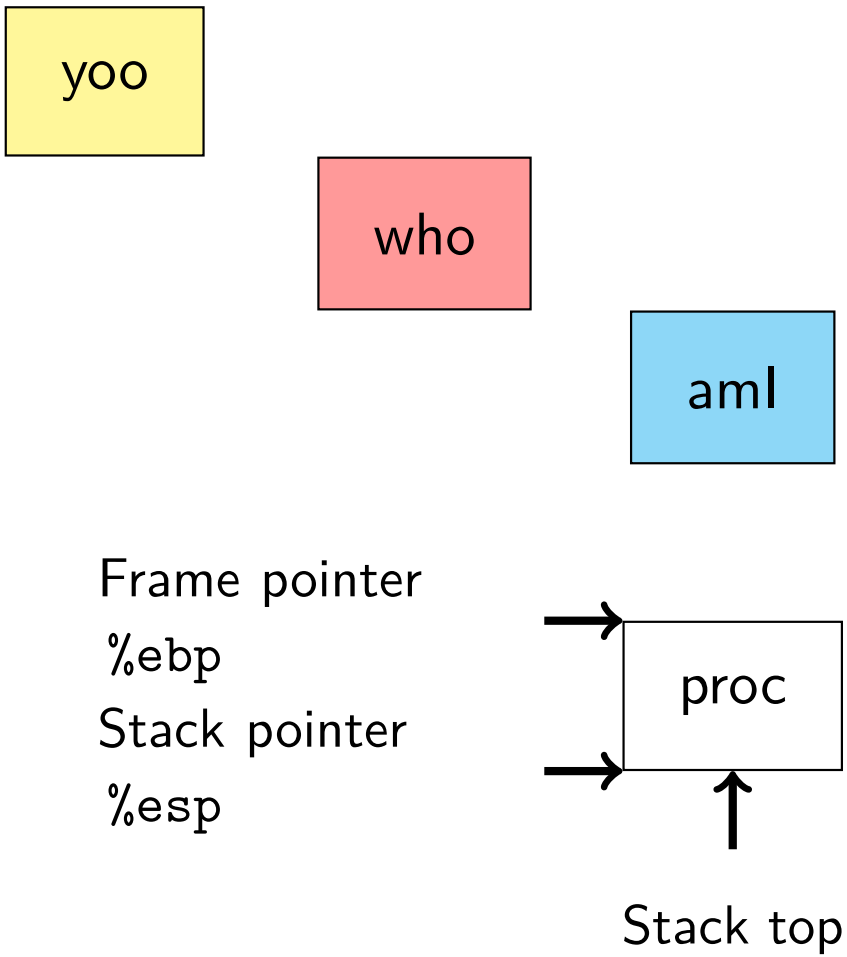
```
who(...){
   ...
   amI();
   ...
   amI();
}
```

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



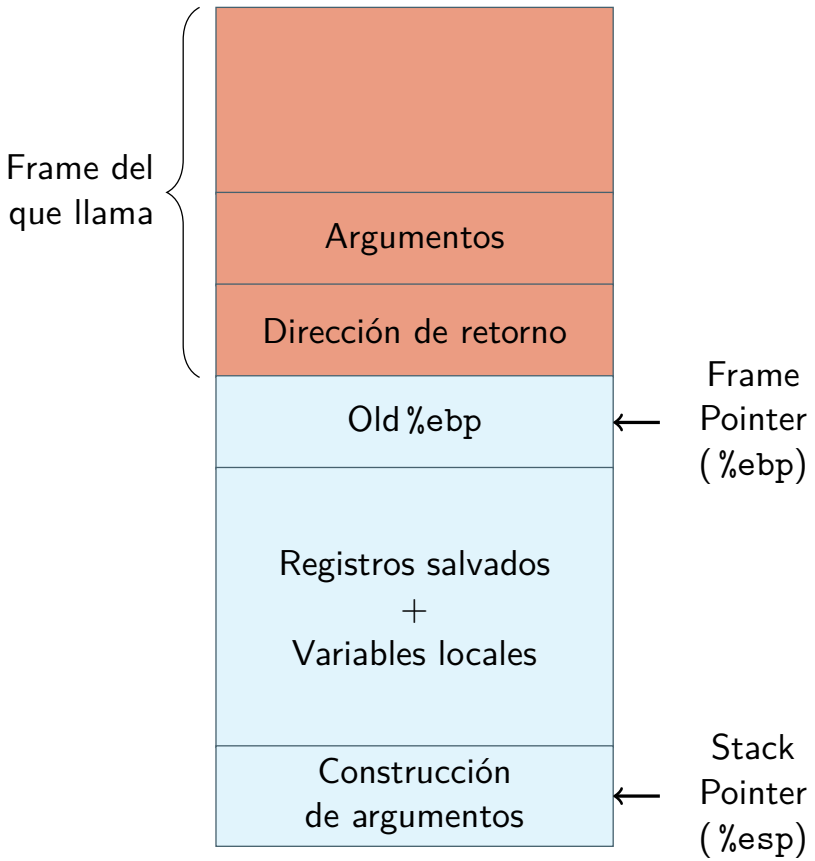
Stack frames

- Contenidos
  - Variables locales
  - Información de retorno
  - Espacio temporario
- Administración
  - Espacio asignado al entrar a un procedimiento
    - *Setup* code
  - Desasignar el espacio al retornar
    - *Finish* code
- Punteros
  - Puntero del stack %esp indica la cima del stack
  - Puntero del frame %ebp indica el inicio del frame actual



IA32/Linux Stack frames

- Stack frame actual (Cima hacia el fondo)
  - Parametros para la función que vamos a llamar
    - Construir los argumentos
  - Variables locales
    - Si no caben todas en los registros
  - Registros almacenados de la función que llama
  - Puntero al frame de la función que llama
- Stack frame de la función que llama
  - Dirección de retorno
    - Push por la instrucción `call`
  - Argumentos para la llamada



swap()

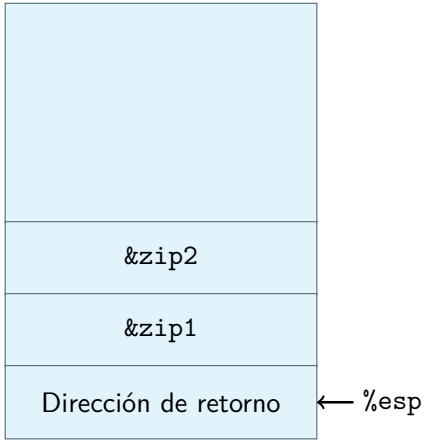
```
int zip1 = 15213;
int zip2 = 91125;

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

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

Llamando swap desde call_swap

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

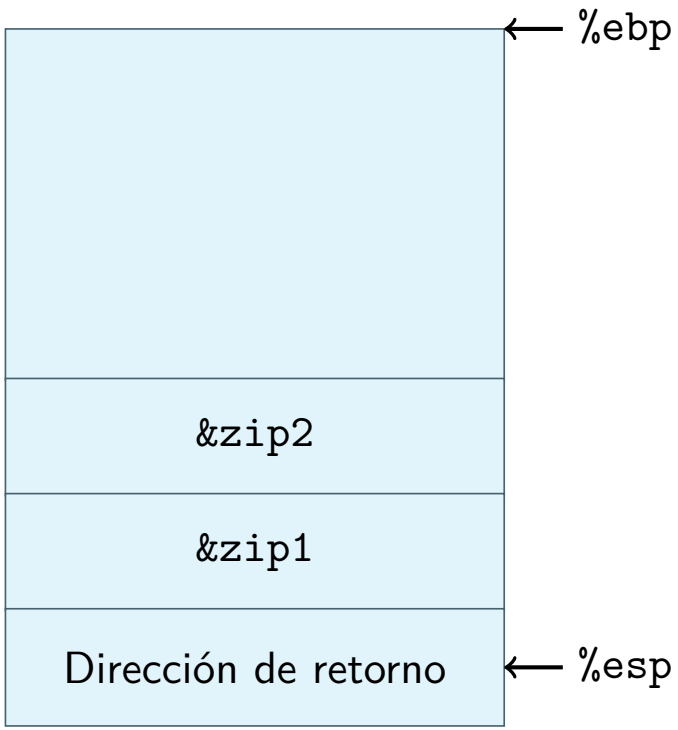


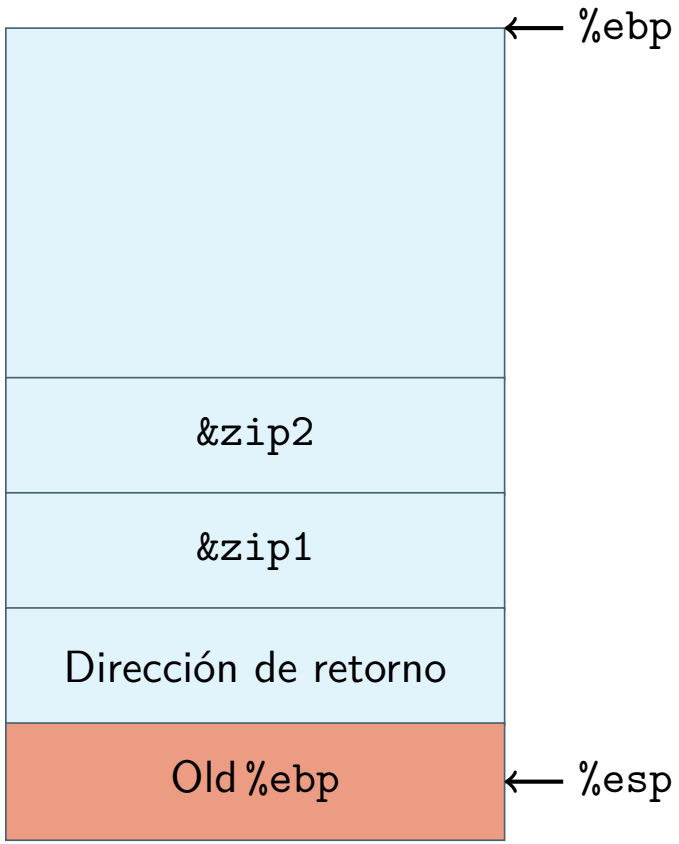
swap()

```
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
```

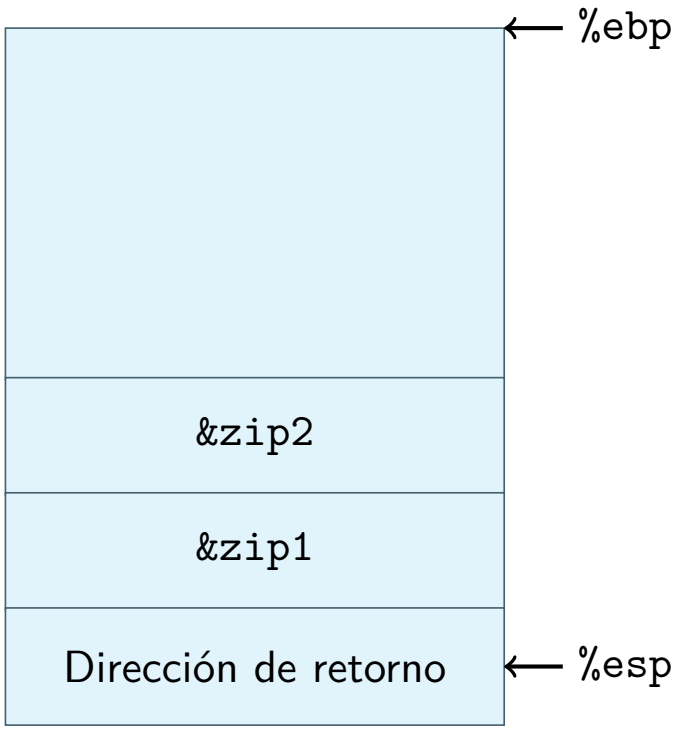
swap()

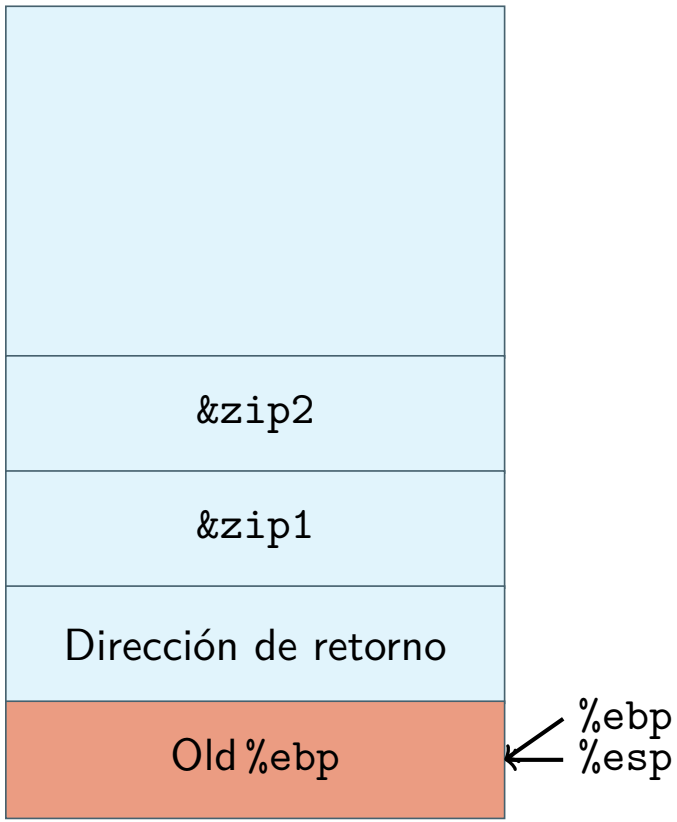





```
swap:
   ;setup
   pushl  %ebp
   movl   %esp , %ebp
   pushl  %ebx
```

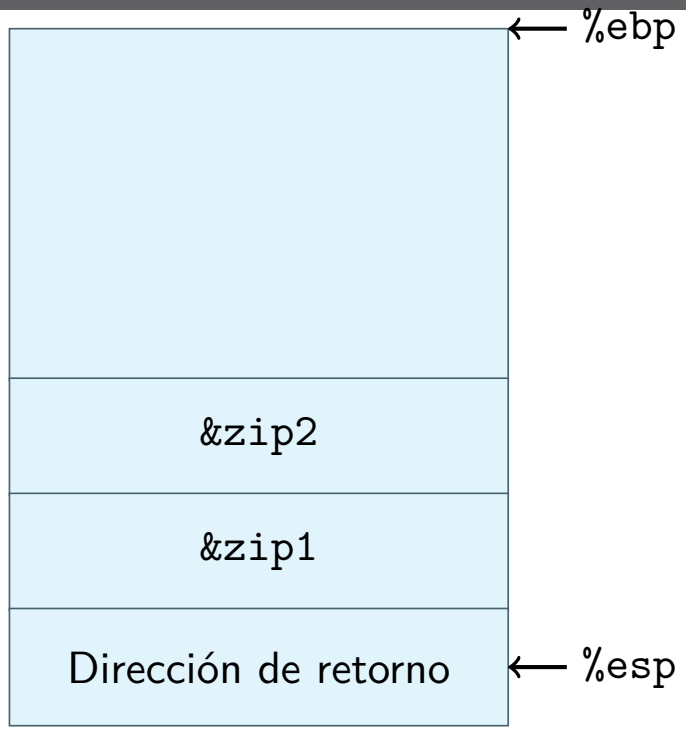
swap()

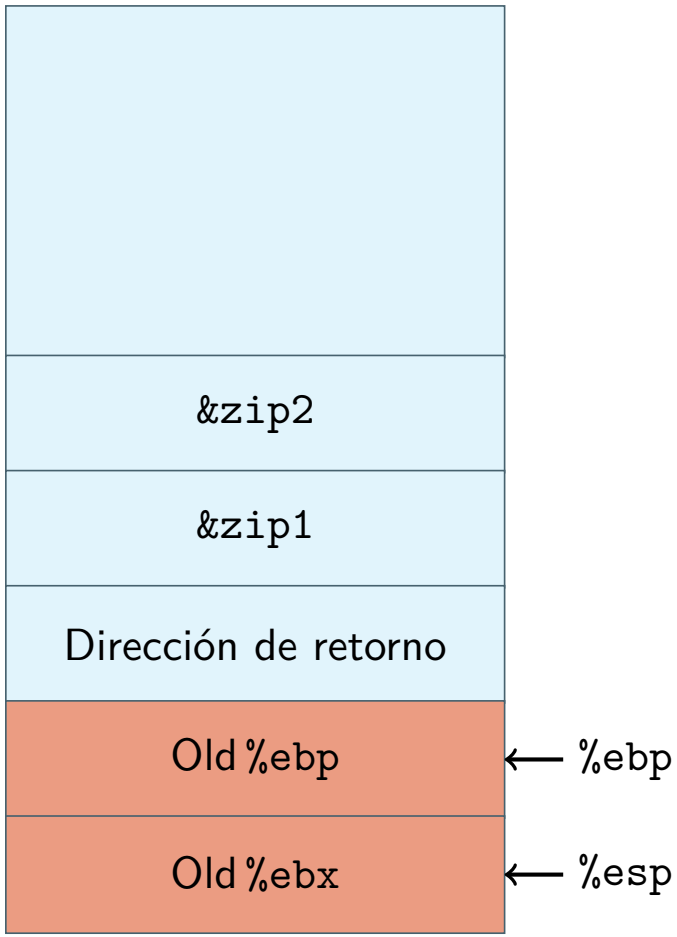





```
swap:
   ;setup
   pushl  %ebp
   movl   %esp , %ebp
   pushl  %ebx
```

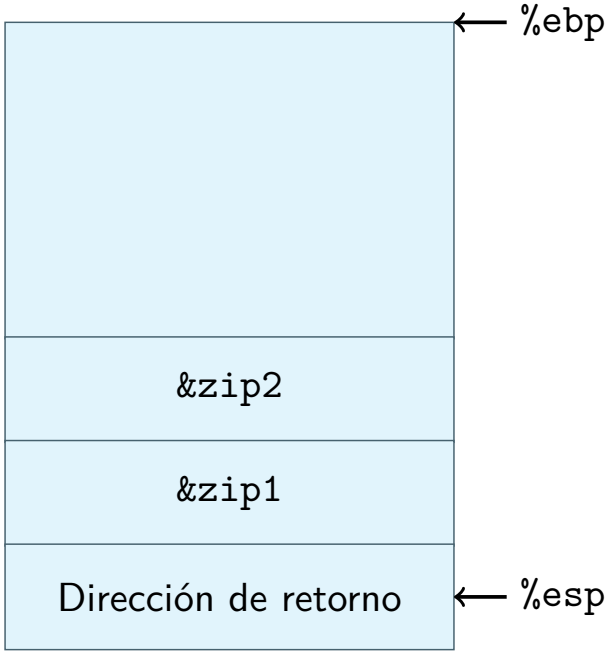
swap()

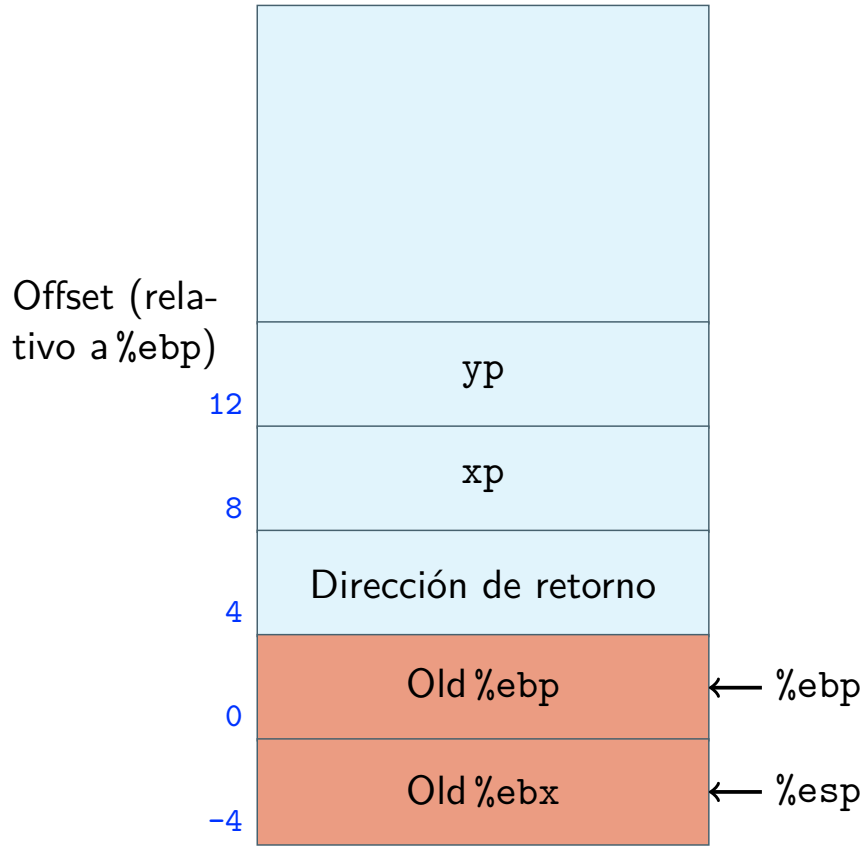





```
swap:
   ;setup
   pushl  %ebp
   movl   %esp, %ebp
   pushl  %ebx
```

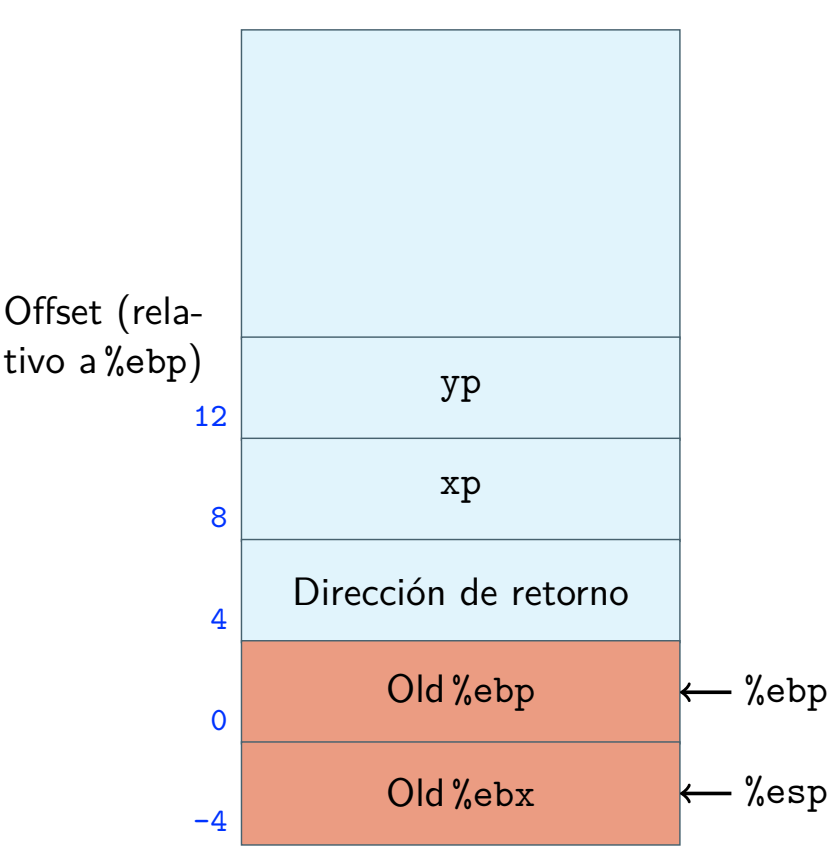
Effecto de swap() setup

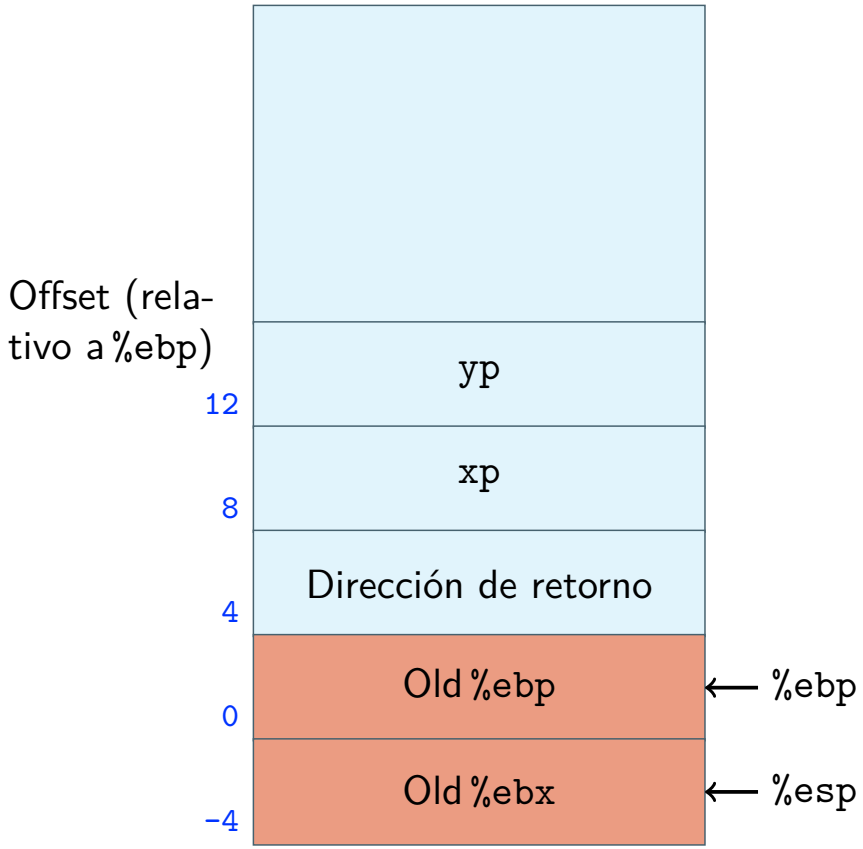





```
;body
movl 12(%ebp), %ecx ;get yp
movl 8(%ebp), %edx  ;get xp
```

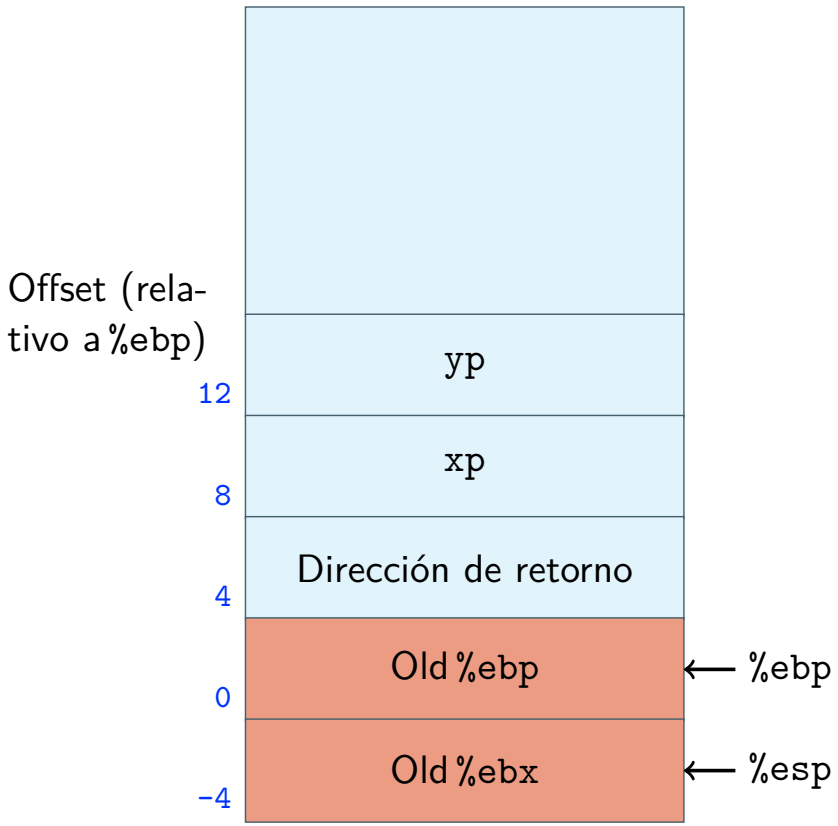
Effecto de swap() finish

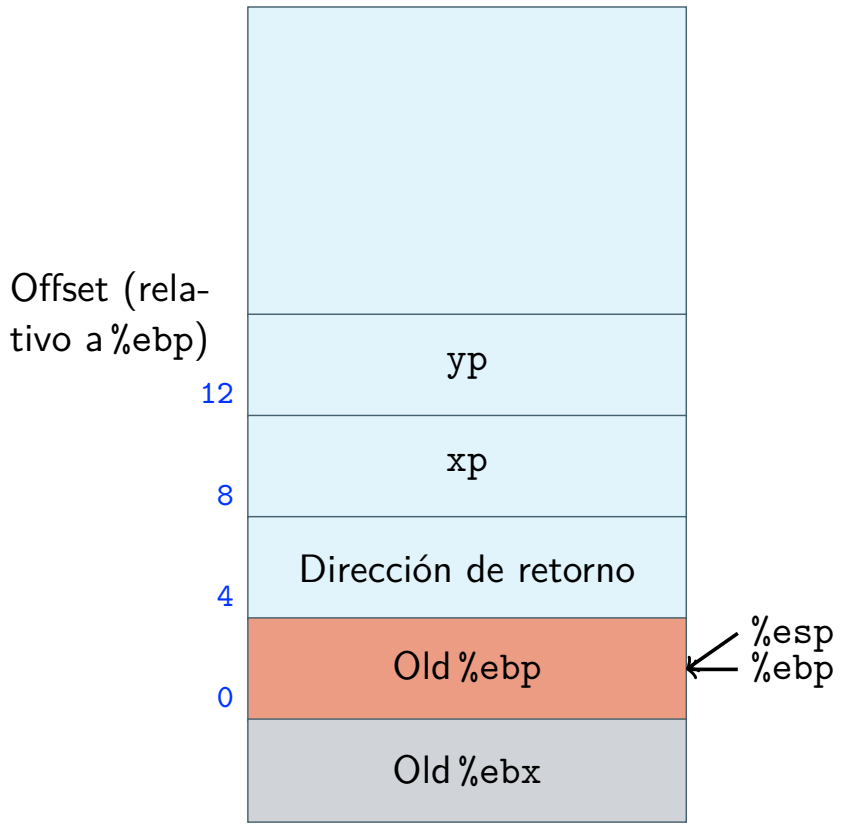





```
;finish
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

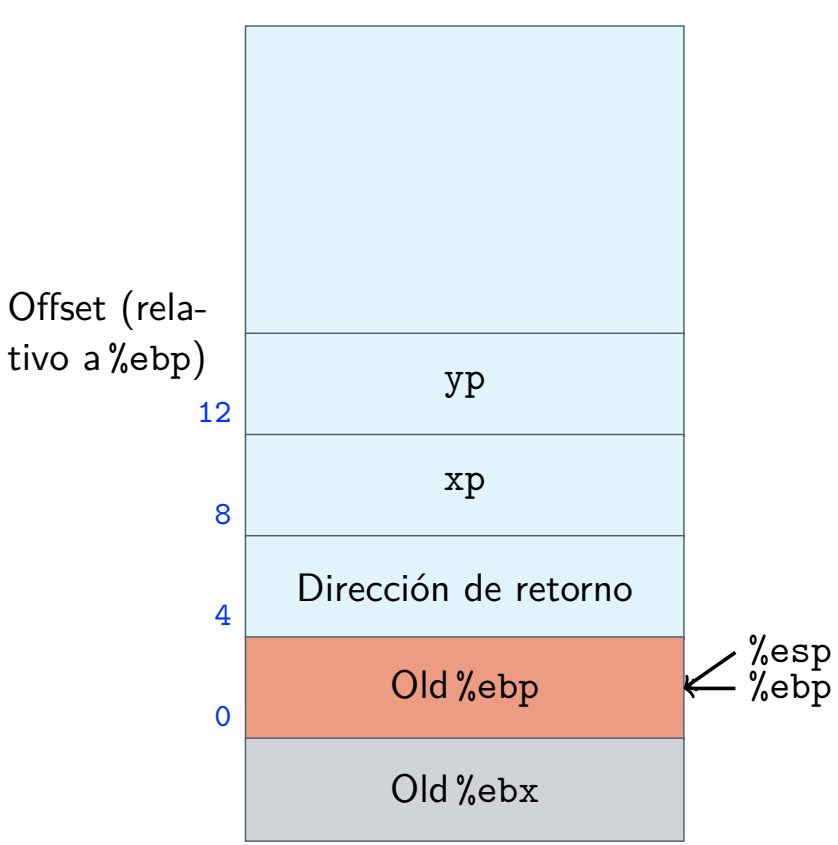
Effecto de swap() finish

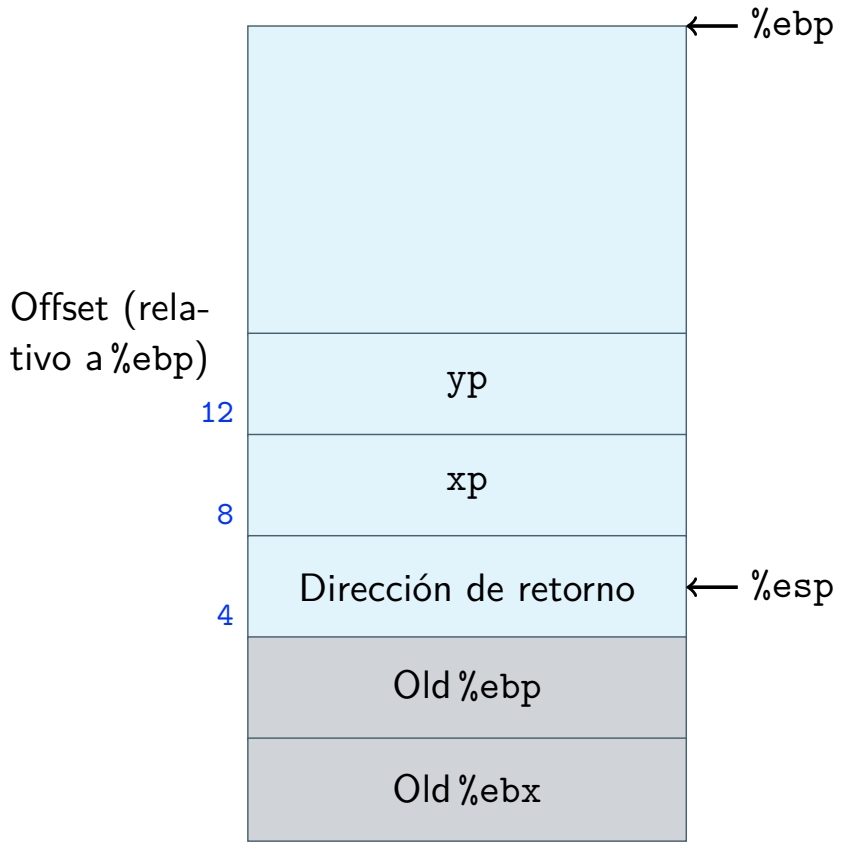





```
;finish
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

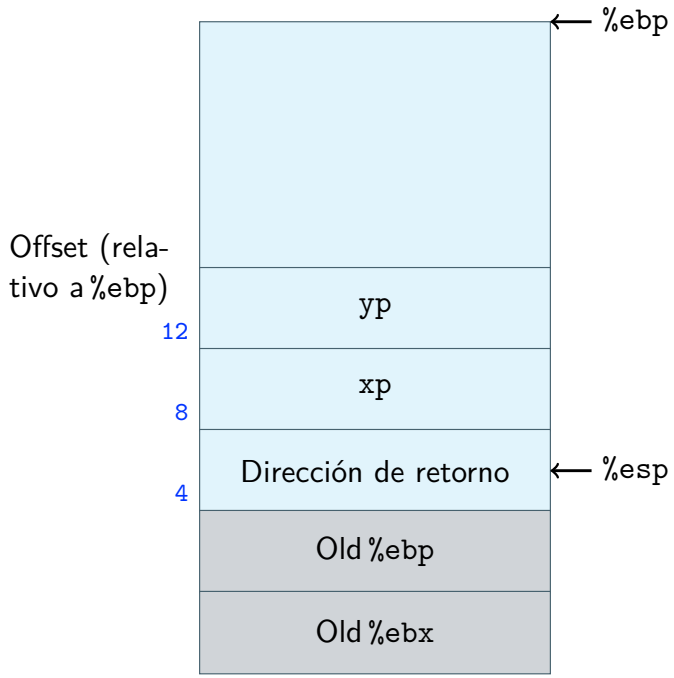
Effecto de swap() finish

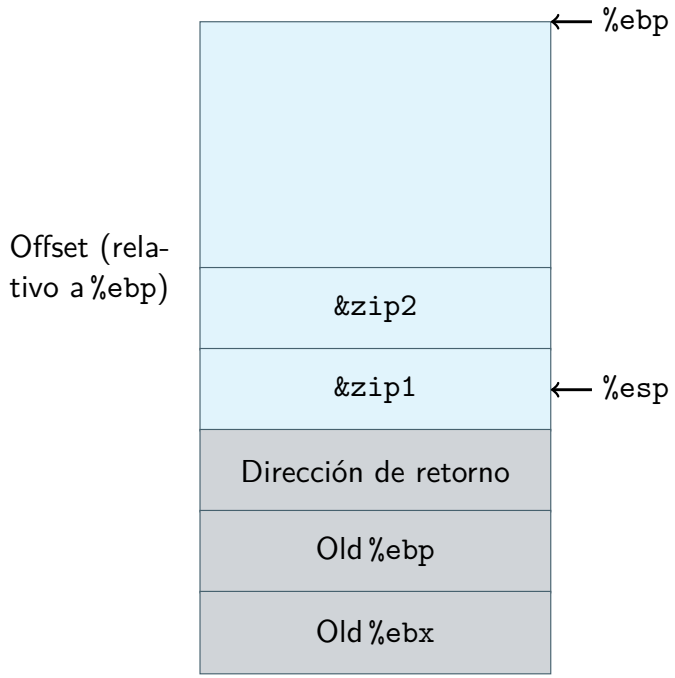





```
;finish
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

Effecto de swap() finish






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

- Se salva y reestablece el resgistro de la función que llamó %ebx
- No se hizo para los registros %eax, %ecx, o %edx

Convención para salvar en registros

- Cuando el procedimiento `yoo()` llama a `who()`
- ¿Puede un registro utilizarse para almacenamiento de memoria temporaria?

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

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

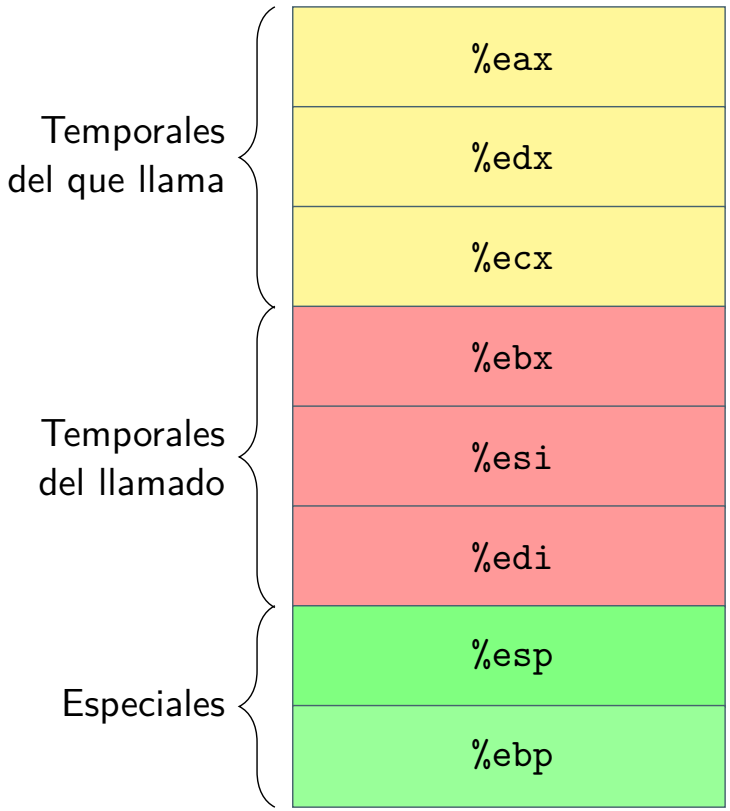
- El contenido del registro %edx es sobreescrito por `who()`

Convención para salvar en registros

- Cuando el procedimiento `yoo()` llama a `who()`
- ¿Puede un registro utilizarse para almacenamiento de memoria temporaria?
- Definiciones
  - *Registro del que llama*: el procedimiento que llama salva temporalmente los registros en su frame antes de llamar
  - *Registros del llamado*: el procedimiento llamado salva temporalmente los registros antes de usar
- Convención
  - ¿Qué registros son para salvar los del que llama y el llamado?

Uso de registros IA32/Linux

- Registros de enteros
  - Dos tienen usos especiales
    - %ebp, %esp
  - Tres se manejan para salvar en el procedimiento llamado
    - %ebx, %esi, %edi
    - Los valores anteriores se colocan en el stack antes de usarse
  - Tres para salvar en el procedimiento que llama
    - %eax, %edx, %ecx
  - El registro %eax almacena el valor de retorno



Resumen del Stack

- El stack hace que la recursión funcione
  - Almacenamiento privado para cada instancia del procedimiento (cada llamada)
    - Distintas instancias no interfieren entre ellas
    - Direcciones locales y argumentos son relativos a la posición del stack
  - Pueden administrarse como un stack
    - Los procedimientos retornan en el orden inverso de las llamadas
- Procedimientos IA32: instrucciones y convenciones
  - `call` y `ret` mezclan `%eip`, `%esp` en una manera establecida
  - Convención del uso de registros
    - Almacenamiento del que llama y el llamado
    - `%ebp` y `%esp`
  - Convención para organizar el stack frame
    - ¿Qué argumento es insertado en la pila antes?

Antes y después del `main()`

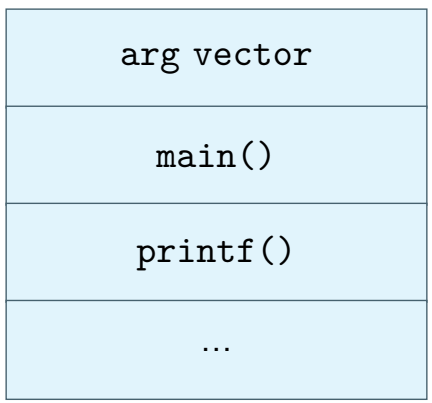
```
int main(int argc, char *argv[]){
  if (argc > 1) {
    printf("%s\n", argv[1]);
  } else {
    char* av[3] = {0, 0, 0};
    av[0] = argv[0];
    av[1] = "Fred";
    execvp(av[0], av); // carga el primer argumento y
        sobreescribe el código de esta instancia, y
        pasa los parametros del segundo argumento ---
        podemos hacer un shell usando esta instrucción
  }
  return (0);
}
```

Partes misteriosas

- `argc`, `argv`
  - Strings del programa
  - Disponibles cuando otro programa se está ejecutando
  - ¿En que parte de la memoria se encuentran?
  - ¿Cómo llegaron ahí?
- ¿Qué pasa cuando `main()` hace `return(0)`?
  - Ya no hay más programa que ejecutar, ¿cierto?
  - ¿Qué pasa con el `0`?
  - ¿Cómo llega a donde va?

No tan misteriosas

- `argc`, `argv`
  - Strings del programa
  - Disponibles cuando otro programa se está ejecutando
- La transferencia de información entre procesos es tarea del sistema operativo
  - El sistema operativo copia los strings de un espacio de memoria antiguo a uno nuevo cuando se llama a `exec()`
  - Tradicionalmente se colocan debajo del fondo del stack (implementaciones dependen del compilador, arquitectura, etc.)
  - Otras cosas extrañas (ambiente, vector auxiliar, etc.) —arriba de `argv`



No tan misteriosas

- ¿Qué pasa cuando `main()` hace `return(0)`?
  - Definido por `C` para tener el mismo comportamiento que `exit(0)`
  - ¿Cómo?
- El envoltorio (wrapper) de `main()`
  - Recibe `argc`, `argv` del sistema operativo
  - Llama `main()`, y luego llama a `exit()`
  - Está en la librería de `C`, tradicionalmente `crt0.s`
  - Tiene un nombre extraño

```
// no es el código real, pero es la idea
void main_wrapper(int argc, char* argv){
  exit( main(argc, argv) );
}
```

Puntos importantes

- Los procesos son instancias de un programa
  - Tienen una forma definida (convención) en memoria
  - Se administran como un stack
- Entendimos como evoluciona el stack en ejecución
- Importancia de los registros (y sus convenciones)
- El wrapper del `main()`

Texto

- Cubrimos partes hasta el capítulo 2 de OS:P+P
- Lean antes y después de clase
- Como ven no cubrimos el texto necesariamente en orden, y vemos cosas que no están ahí
- Además, hablamos de cosas complejas que requieren de que hayan leído
- Hay cosas que no hablamos (que están en el libro) y el resto de conceptos requieren que las entiendan (así que venir a clase y leer tampoco es suficiente)

Q: Entonces, ¿qué es suficiente?

A: Leer (antes y después para terminar de comprender), asistir a clases, ejercitar

Tarea

Shell clon

- ¿Qué hacer?
  - Escriban un clon de una terminal de comandos utilizando el comando `execvp`
  - Su aplicación debe de permitir escribir comandos que existan en Linux, y ejecutarlos desde la instancia de su aplicación
- Reporte
  - Escriban un reporte de no más de 3 páginas explicando
    - Introducción a las terminales, y cómo funciona una shell común
    - Además, expliquen el funcionamiento de `execvp`
    - ¿Cómo hicieron la aplicación?, y como usaron el funcionamiento `execvp` para poder resolverlo
    - Partes principales de su solución
    - Mostrando el funcionamiento de la aplicación: llamen a varios comandos, e.g., `cp`, `ls`, `mv`
- Entrega: Viernes 20-02-2015 (en el portal antes del deadline)