

## Procedimientos Índice

---

1. Definición y fundamentos
2. Uso en entorno x86-16bits
3. X86 *calling conventions*
4. Programación multimódulo
5. Librerías

## Procedimientos 1. Definición y fundamentos

---

### 1. Definición y fundamentos Índice

1. Concepto
2. Argumentos
3. Valor devuelto
4. Variables locales
5. Soporte *hardware*
6. ABI

## Procedimientos

### 1.1. Concepto

---

- **Términos sinónimos:**

- ➔ Subrutina
- ➔ Subprograma
- ➔ Función
- ➔ Rutina

➔ En algunos casos se reserva el término **función** para una subrutina que devuelve un valor

## Procedimientos

### 1.1. Concepto

---

- Funcionalmente, un procedimiento es un subalgoritmo de un algoritmo principal que resuelve una **tarea específica**
- En la práctica, un procedimiento es una **fracción de código separada** de la secuencia principal que puede ser invocado desde cualquier parte del programa, ya sea desde el flujo principal o desde otro procedimiento

## Procedimientos

### 1.1. Concepto

---

- La invocación de un procedimiento supone una **doble transferencia de control** (salto o bifurcación)
  - ➔ Primero al código de la subrutina; y
  - ➔ Posteriormente un retorno al código principal
- ➔ Algunos compiladores detectan cuantas veces se llama cada procedimiento y cuando son pocas sustituyen la llamada por el código <sup>1</sup>
  - ➔ Así se evitan la **sobrecarga** debida a las transferencias de control (saltos) y al paso de parámetros

<sup>1</sup> *Inline expansion* o *inlining* es una optimización que realiza el compilador consistente en reemplazar la llamada a una subrutina por el código de la misma

5/66

© Rafael Rico López

## Procedimientos

### 1.1. Concepto

---

- Desde el punto de vista de la ingeniería del software
  - ➔ Un procedimiento se escribe una vez y se llama siempre que haga falta
  - ➔ Facilita:
    - ➔ La vista del código
    - ➔ El mantenimiento del código
    - ➔ La modularidad del código <sup>1</sup>

<sup>1</sup> La programación estructurada y su "hija", la programación modular, consideran que un desarrollo de *software* óptimo sólo debe utilizar subrutinas

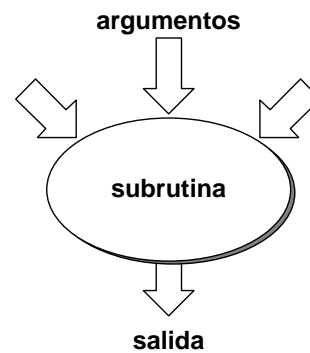
6/66

© Rafael Rico López

## Procedimientos

### 1.1. Concepto

- **Elementos de una subrutina**
  - ➔ **Declaración**
    - ➔ Nombre
    - ➔ Código
  - ➔ **Argumentos o parámetros de entrada**
  - ➔ **Tipo de valor devuelto (si lo hay)**



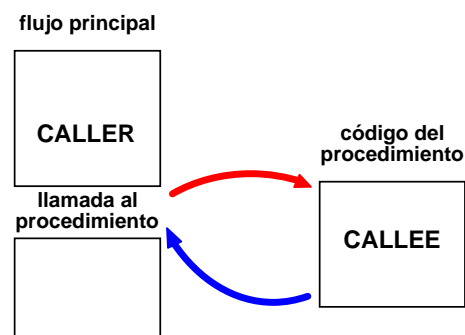
© Rafael Rico López

7/66

## Procedimientos

### 1.1. Concepto

- **Nomenclatura**
  - ➔ El flujo de código que invoca el procedimiento se conoce como código llamador (*caller*)
  - ➔ El flujo de código que se ejecuta en el cuerpo del procedimiento se conoce como código llamado (*callee*)



© Rafael Rico López

8/66

## Procedimientos

### 1.2. Argumentos

- Son los datos que va a tratar el procedimiento
  - ➔ Podemos pasar argumentos
    - ➔ Por **valor** → el argumento es el dato a tratar
    - ➔ Por **referencia** → el argumento es un puntero a una posición de memoria en la que está el dato a tratar
  - ➔ Los argumentos son datos de entrada pero también pueden recibir **resultados** obtenidos como consecuencia del procesamiento de la subrutina (si se pasan como referencia)


© Rafael Rico López

9/66

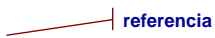
## Procedimientos

### 1.2. Argumentos

- Ejemplos<sup>1</sup>:

`int putchar(int char)` 

- ➔ Escribe en *stdout* el carácter pasado como argumento; si tiene éxito devuelve el carácter y EOF en caso de error

`char *strcpy(char *str1, const char *str2)` 

- ➔ Copia (destruktiva) de la cadena *str2* en la cadena *str1*; devuelve el puntero a *str1* o NULL si error

<sup>1</sup> Usaremos funciones de ANSI C como ejemplo

© Rafael Rico López

10/66

## Procedimientos

### 1.3. Valor devuelto

- Un procedimiento puede (opcionalmente) devolver un valor (que no es de la lista de argumentos)
  - ➔ A veces, es un entero que indica si el procesamiento ha tenido éxito o se ha producido algún tipo de **error**
    - ➔ En estos casos, el valor devuelto puede NO usarse sin que eso afecte a la funcionalidad del programa
  - ➔ En otros casos, el valor devuelto es el **resultado principal** del procedimiento

## Procedimientos

### 1.3. Valor devuelto

- Modos de uso:

```
error = funcion(argumentos)
if (error = valor) then ...
```

el valor devuelto puede indicar si ha ocurrido un error

- ➔ O mejor:

```
if (funcion(argumentos) = valor) then ...
```

- ➔ A veces:

```
funcion(argumentos) ...
```

## Procedimientos

### 1.3. Valor devuelto

- Ejemplos:

el valor devuelto es el resultado principal

```
FILE *fopen(const char *nombre, const char *modo)
```

- ➔ Abre el fichero cuyo nombre pasamos como primer argumento; si tiene éxito devuelve un puntero a un descriptor de fichero y NULL en caso de error

```
int atoi(const char *str)
```

- ➔ Convierte la cadena pasada como argumento a entero; devuelve el entero ó 0 en caso de error

## Procedimientos

### 1.4. Variables locales

- Un procedimiento puede declarar y utilizar variables locales, es decir, variables **sólo accesibles desde la subrutina**, que se crean cuando se invoca y que desaparecen cuando termina

- ➔ Se dice que su ámbito es local y su **tiempo de vida limitado** al procesamiento de la subrutina

- ➔ ¡Recordamos! que si el ámbito es local pero el tiempo de vida es todo el tiempo de ejecución, es una variable estática

## Procedimientos

### 1.4. Variables locales

- Ejemplos:

```
int mifuncion(argumentos)
{
    int i=10;
    while(i)
        i--;
    ...
    ...
    return i;
}
```

variable local

valor devuelto

© Rafael Rico López

15/66

## Procedimientos

### 1.5. Soporte *hardware*

- Control de flujo

- ➔ Instrucciones específicas

- ➔ **Llamada** → salvaguarda de contexto y salto incondicional a subrutina

- ➔ **Retorno** → recuperación de contexto y salto incondicional a código ppal.

- ➔ El **contexto** puede ser simplemente el PC o puede ser una estructura de datos más compleja

- ➔ Salvaguarda del contexto:

- En registros
- En la pila

© Rafael Rico López

16/66



## Procedimientos

### 1.5. Soporte *hardware*

- Paso de argumentos
- Valor devuelto
- Variables locales

#### ➔ En registros

- ➔ Organizados como **banco de registros** o
- ➔ como **ventanas de registros**<sup>1</sup>

#### ➔ En la pila

- ➔ Organización LIFO de memoria

<sup>1</sup> Las ventanas de registros encuentran justificación en estudios que establecen que, en promedio, el máximo número de argumentos es 6, el máximo número de variables locales es 6 y la profundidad máxima de las llamadas es 5

17/66

© Rafael Rico López

## Procedimientos

### 1.5. Soporte *hardware*

- Ventajas e inconvenientes:

#### ➔ En registros

- ➔ Pocos registros → hay que garantizar que no se sobrescriben<sup>1</sup>
- ➔ No dan lugar a subrutinas reentrantes ni anidables
- ➔ Los registros son **rápidos** | criterio si se usan registros
- ➔ Generan pocas dependencias

#### ➔ En la pila

- ➔ Acceso lento a memoria
- ➔ Sobrecarga de dependencias y cálculo de direcciones
- ➔ Las subrutinas son **reentrantes y anidables** | criterio si se usa la pila

<sup>1</sup> La salvaguarda de los registros se suele hacer en la pila. Esto genera un tráfico con memoria en forma de PUSH y POP

18/66

© Rafael Rico López

## Procedimientos

### 1.5. Soporte *hardware*

---

- Teoría de compiladores (I):
  - ➔ **Concepto de subrutina reentrante**
    - ➔ En general, se dice que un código es reentrante si se puede llamar simultáneamente desde varios puntos sin alterar su funcionamiento
  - ➔ **Concepto de subrutina anidable**
    - ➔ O **recursiva**, es aquella que puede llamarse a sí misma
    - ➔ Una subrutina recursiva es reentrante

## Procedimientos

### 1.5. Soporte *hardware*

---

- Teoría de compiladores (II):
  - ➔ Desde un punto de vista práctico, el código reentrante no usa variables estáticas en sucesivas llamadas
  - ➔ Para garantizar que un código es reentrante, todos los datos que se deban mantener entre llamadas deberán ser suministrados por el llamador (*caller*)
  - ➔ Una subrutina reentrante no puede usar registros para pasar argumentos; **es imperativo usar la pila**

## Procedimientos

### 1.5. Soporte *hardware*

---

- Teoría de compiladores (III):
  - ➔ El lenguaje C está orientado hacia el código reentrante
  - ➔ Las llamadas al sistema no son reentrantes ya que pasan argumentos en registros (para que sean rápidas)
  - ➔ Para que un sistema operativo sea reentrante, debemos deshabilitar las interrupciones durante cada llamada al sistema; esto aumenta la latencia (ralentiza el sistema)

## Procedimientos

### 1.6. ABI

---

- El *Application Binary Interface* (ABI) describe la interfase de bajo nivel que relaciona una aplicación con el sistema operativo u otras aplicaciones
- En concreto el ABI detalla:
  - ➔ Los tipos de datos, su tamaño y el alineamiento
  - ➔ La **convenio de llamadas** (*calling convention*) a subrutinas
  - ➔ El modo de realizar las llamadas al sistema
  - ➔ Opcionalmente, el formato de los ficheros OBJ y librerías
- ➔ El ABI no debe confundirse con el API (librerías de llamadas al sistema)
- ➔ El ABI debe cumplirlo el compilador

## Procedimientos

### 1.6. ABI

- **Convenio de llamadas indica:**

- ➔ **Cómo se pasan los argumentos**

- ➔ Pila → orden (**RTL** *right to left* ó **LTR** *left to right*)

- ➔ Registros → cuales se usan y en qué orden

funcion (arg\_1, arg\_2, ... arg\_n)

pila ← arg\_1 arg\_2 ... arg\_n

reg\_1 ← arg\_1

reg\_2 ← arg\_2

pila ← arg\_n ... arg\_2 arg\_1

reg\_n ← arg\_n

- ➔ **Cómo se comunican los valores devueltos**

- ➔ **Quién limpia la pila al terminar**

- ➔ *Caller clean-up* o *callee clean-up*

## Procedimientos

### 2. Uso en entorno x86-16bits

#### 2. Uso en entorno x86-16bits

##### Índice

1. Declaración
2. Invocación y retorno
3. Paso de argumentos por pila
4. Variables locales
5. Uso de estructuras

## Procedimientos

### 2.1. Declaración

- Utilizamos las directivas PROC y ENDP
  - ➔ Marcan el principio y el final de los procedimientos

1 etiqueta PROC [NEAR|FAR]  
: : : ;código  
RET [constante]  
etiqueta ENDP

2 etiqueta:  
: : : ;código  
RETN [constante]

3 etiqueta: LABEL FAR  
: : : ;código  
RETF [constante]

25/66

© Rafael Rico López

## Procedimientos

### 2.2. Invocación y retorno

- Instrucciones
  - ➔ CALL ➔ salva el *contexto* (CS:IP *-far-* o IP *-near-*) en la pila y salta al procedimiento
  - ➔ RET ➔ recupera el *contexto* y vuelve al programa principal; puede hacer el *clean-up* de la pila
    - ➔ *contexto* ➔ información relativa al entorno de procesamiento principal

26/66

© Rafael Rico López

## Procedimientos

### 2.2. Invocación y retorno

`CALL {registro | memoria}`

- Salva en la pila la dirección de la siguiente instrucción <sup>1</sup>
- Salta a la dirección especificada

```
[SP = SP - 2]      ; si salto far
[CS → pila]
[CS = nuevo CS]

SP = SP - 2
IP → pila
IP = nuevo IP
```

<sup>1</sup> Un CALL a la siguiente instrucción permite obtener el CS:IP en curso

27/66

© Rafael Rico López

## Procedimientos

### 2.2. Invocación y retorno

`CALL {registro | memoria}`

- La dirección de salto se puede dar de forma:
  - ➔ Directa → etiqueta
  - ➔ Indirecta → puntero en registro o en memoria
- Los saltos pueden ser:
  - ➔ *near* → sólo se da y se salva en la pila IP
  - ➔ *far* → la dirección de salto se muestra como CS:IP

Ejemplo:

`CALL far ptr TAREA → salto far`

problemática de las etiquetas adelantadas

28/66

© Rafael Rico López

## Procedimientos

### 2.2. Invocación y retorno

---

RET [ constante ]

- Devuelve el *contexto* original de la secuencia de código principal sacando de la pila CS:IP o IP (dependiendo de si el salto es *far* o *near*)
- Como operando opcional tenemos una constante cuyo significado es el número de bytes a sumar a SP después de retornar (*clean-up* de la pila)

RET ( <i>far</i> )	RET ( <i>near</i> )	RETN 4
POP IP    SP+2	POP IP    SP+2	POP IP    SP+2
POP CS    SP+2		SP+4

© Rafael Rico López

29/66

## Procedimientos

### 2.2. Invocación y retorno

---

- Juntando todo:

<pre>CALL tarea : : : : : :  tarea PROC : : : : : :  RET 1  tarea ENDP</pre>	<pre>CALL near ptr tarea : : : : : :  tarea: : : : : : :  RETN 2</pre>
------------------------------------------------------------------------------	------------------------------------------------------------------------

<sup>1</sup> Asume el tipo de salto del CALL    <sup>2</sup> Supone CALL *near*; si es *far* emite error

30/66

© Rafael Rico López

## Procedimientos

### 2.3. Paso de argumentos por pila

- Si usamos la pila para pasar argumentos...
  - ➔ Los argumentos se pasan a la pila antes de la llamada al procedimiento (tarea del módulo llamador o *caller*)
  - ➔ Transferido el control, el procedimiento recupera los argumentos y los procesa (tarea del llamado o *callee*)
  - ➔ Los registros que vaya a usar el procedimiento deben ser preservados copiándolos en la pila
    - ➔ Lo puede hacer el *caller* o el *callee*
  - ➔ Finalmente se ajusta el puntero de pila para soslayar los argumentos
    - ➔ En secuencia principal `ADD SP, n` (*caller clean-up*), o
    - ➔ Con `RET n` (*callee clean-up*)

© Rafael Rico López

31/66

## Procedimientos

### 2.3. Paso de argumentos por pila

- Tareas del *caller*
  - ➔ Antes de invocar el procedimiento
    - ➔ Si es el caso, preservar en la pila los registros que vaya a usar el procedimiento (*caller-saved*)
      - Podrían ser todos por simplicidad pero es lento
    - ➔ Pasar a la pila los argumentos en el orden establecido por el ABI; pueden ser valores o referencias (*near* o *far*)
    - ➔ Invocar el procedimiento
  - ➔ Después de retornar del procedimiento
    - ➔ Si es el caso, soslayar argumentos (*caller clean-up*) (`ADD SP, n`)
    - ➔ Si es el caso, salvar el valor devuelto
    - ➔ Si es el caso, recuperar los registros preservados

© Rafael Rico López

32/66



## Procedimientos

### 2.3. Paso de argumentos por pila

- **Tareas del *callee***

- ➔ Copiar el registro BP en la pila (**PUSH BP**)
  - El registro BP es el **marco de pila**; es un puntero que sirve de referencia de argumentos y variables locales para el procedimiento en curso; salvamos el marco de pila del procedimiento llamador
- ➔ Copiar el puntero de pila SP en BP (**MOV BP, SP**)
  - Cargamos el marco de pila del procedimiento en curso
- ➔ Si es el caso, preservar en la pila los registros que vaya a usar el procedimiento (*callee-saved*)
- ➔ Ejecutar el código
- ➔ Si es el caso, recuperar los registros preservados
- ➔ Recuperar el marco de pila llamador (**POP BP**)
- ➔ Si es el caso, soslayar argumentos (*callee clean-up*) (**RET n**)

© Rafael Rico López

33/66

## Procedimientos

### 2.3. Paso de argumentos por pila

- **Clean-up**

- ➔ Después de ejecutar el procedimiento, el estado de la pila debe ser el que tendría de no haberse ejecutado
- ➔ Debemos ajustar la cima de la pila para que no crezca con los argumentos ya que a la larga se producirá un desbordamiento de pila
- ➔ Esto es lo que llamamos soslayar argumentos
  - ➔ Si lo hace el *caller*, usa **ADD SP, n**
  - ➔ Si lo hace el *callee* usa **RET n** donde *n* es el tamaño en bytes de los argumentos
- ➔ Si lo hace el *caller*, admite un número de argumentos variable (caso de funciones como `vprintf`)

© Rafael Rico López

34/66

## Procedimientos

### 2.3. Paso de argumentos por pila

---

- Ejemplo:

- ➔ Sea el procedimiento

```
int madd (int a, int b, int m)
```

que devuelve el valor del cálculo  $a \times b + m$

- ➔ Los argumentos se pasan **por valor** de derecha a izquierda

- ➔ El valor devuelto se escribe en AX

- ➔ AX es *caller-saved*

- ➔ Asumimos *caller clean-up*

© Rafael Rico López

35/66

## Procedimientos

### 2.3. Paso de argumentos por pila

---

- Código del *caller*:

```
PUSH AX      ;preservo AX (caller-saved)
```

```
PUSH m      ;paso el 3er parámetro a la pila
```

```
PUSH b      ;paso el 2o
```

```
PUSH a      ;paso el 1o
```

```
CALL madd   ;llamo al procedimiento
```

```
ADD SP,6    ;soslayo argumentos (caller clean-up)
```

```
MOV r,AX    ;salvo el valor devuelto
```

```
POP AX      ;recupero AX (caller-saved)
```

© Rafael Rico López

36/66

## Procedimientos

### 2.3. Paso de argumentos por pila

- Código del *callee*:

```

madd PROC NEAR
    PUSH BP           ;salvo el marco de la pila antiguo
    MOV BP,SP        ;actualizo el marco para este proc.

    MOV AX,[BP+4]    ;recupero el 1er argumento (a)
    MUL [BP+6]       ;multiplico el 2º (b) por el 1º (a)
    ADD AX,[BP+8]    ;sumo con el 3º (m) con axb (DX:AX)
                    ;asumo que DX es 0

    POP BP           ;recupero el marco antiguo
    RET              ;devuelvo el control al programa ppal.
madd ENDP
    
```

¡OJO! Habría que preservar DX

37/66

© Rafael Rico López

## Procedimientos

### 2.3. Paso de argumentos por pila

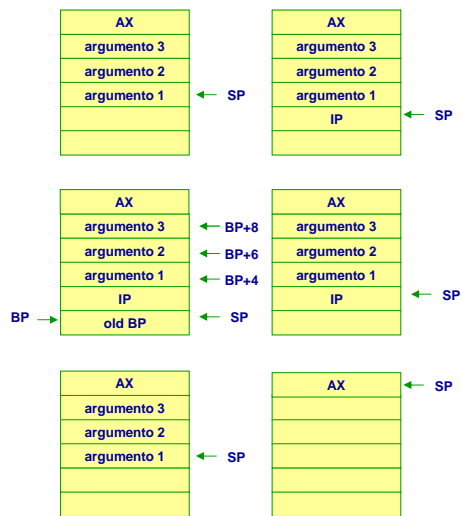
```

PUSH AX
PUSH m
PUSH b
PUSH a
CALL madd
ADD SP,6
MOV r,AX
POP AX

madd PROC NEAR
    PUSH BP
    MOV BP,SP

    MOV AX,[BP+4]
    MUL [BP+6]
    ADD AX,[BP+8]

    POP BP
    RET
madd ENDP
    
```



© Rafael Rico López

38/66

## Procedimientos

### 2.4. Variables locales

- **Uso de variables locales (I):**
  - ➔ **Las variables locales tienen un tiempo de vida limitado a la ejecución del procedimiento**
    - ➔ Se reserva memoria para ellas en la pila al comenzar el procedimiento y después de ajustar el marco de pila desplazando el puntero de pila (**SUB SP, tamaño**)
    - ➔ Se opera con ellas mientras se está ejecutando el procedimiento
    - ➔ Se destruyen al finalizar el procedimiento
  - ➔ **Se crean en la pila y se acceden por su posición en la pila (respecto al marco de pila BP)**

© Rafael Rico López

39/66

## Procedimientos

### 2.4. Variables locales

- **Ejemplo:**
  - ➔ **Sea el procedimiento**

```
int cuenta_char (char *str, char c)
```

que devuelve cuántas veces aparece el carácter *c* en la cadena *str*
    - ➔ La cadena se pasa **por referencia** como puntero *far*
    - ➔ El procedimiento se declara como *far*
    - ➔ En el procedimiento, la cuenta se lleva en una variable local
  - ➔ Los argumentos se pasan de derecha a izquierda
  - ➔ El valor devuelto se escribe en *AX*
  - ➔ *AX* es *caller-saved*
  - ➔ ¡OJO! Asumimos *callee clean-up*

© Rafael Rico López

40/66

## Procedimientos

### 2.4. Variables locales

- Código del *caller*:

```
PUSH AX      ;preservo AX (caller-saved)
PUSH c       ;paso el 2º parámetro a la pila
PUSH DS      ;paso el 1º (base del puntero far)
LEA BX, str
PUSH BX      ;offset del puntero far

CALL far ptr cuenta_char

MOV r, AX    ;salvo el valor devuelto

POP AX       ;recupero AX
```

hay que convertir el char en entero word

AX no se puede preservar en *callee* porque es el registro en el que se devuelve el valor

41/66

© Rafael Rico López

## Procedimientos

### 2.4. Variables locales

- Código del *callee*:

```
cuenta_char PROC FAR
    PUSH BP      ;salvo el marco de pila
    MOV BP, SP   ;actualizo el marco
    SUB SP, 2    ;reserva para la variable local

    ;[BP + 6] → argumento 1: offset del puntero
    ;[BP + 8] → argumento 1: base del puntero
    ;[BP +10] → argumento 2: carácter c
    ;[BP - 2] → variable local

    MOV SP, BP   ;soslayo var. locales
    POP BP      ;recupero el marco antiguo
    RET 6        ;retorno y soslayo argumentos
cuenta_char ENDP
;el resultado se devuelve en AX
```

42/66

© Rafael Rico López

## Procedimientos

### 2.4. Variables locales

- Código del cuerpo de la función:

```

                PUSH BX, ES, SI ;preservo registros
                MOV [BP-2],0    ;inicio el contador
                MOV BX,[BP+10] ;carga carácter en BX
                MOV ES,[BP+8]  ;base de la cadena
                MOV SI,[BP+6]  ;offset de la cadena
seguir:        CMP ES:[SI],0   ;miro si es el final
                JZ salir
                CMP ES:[SI],BL ;miro si es el carácter
                JNZ nocontar
                INC [BP-2]     ;incremento el contador
nocontar:     INC SI          ;incremento el puntero
                JMP seguir
salir:        POP SI, ES, BX  ;recupero registros
                MOV AX,[BP-2] ;devuelvo el resultado en AX
    
```

43/66

AX no se puede preservar en *callee*

aunque es  
word, BH es 0

© Rafael Rico López

## Procedimientos

### (ejemplo de uso de EQU)

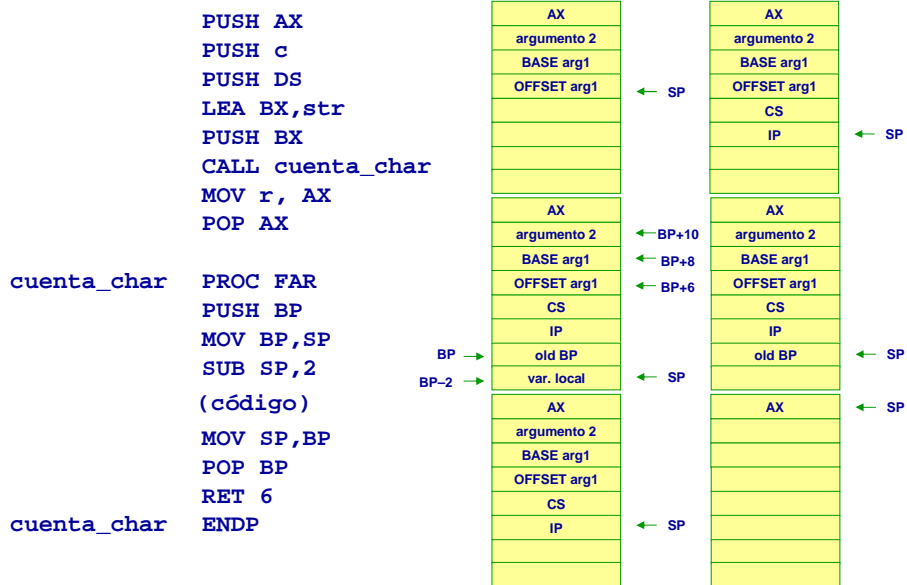
```

argloffset EQU <[BP+6]> ;posición del arg1 (offset)
arglbase   EQU <[BP+8]> ;posición del arg1 (base)
arg2       EQU <[BP+10]> ;posición del argumento2
loc        EQU <[BP-2]>  ;posición de la var. local

                PUSH BX, ES, SI ;preservo registros
                MOV loc,0       ;inicio el contador
                MOV BX,arg2     ;carga carácter en BX
                MOV ES,arglbase ;base de la cadena
                MOV SI,argloffset ;offset de la cadena
seguir:        CMP ES:[SI],0   ;miro si es el final
                JZ salir
                CMP ES:[SI],BL ;miro si es el carácter
                JNZ nocontar
                INC loc        ;incremento el contador
nocontar:     INC SI          ;incremento el puntero
                JMP seguir
salir:        POP SI, ES, BX  ;recupero registros
                MOV AX,loc    ;devuelvo el resultado en AX
    
```

© Rafael Rico López

## Procedimientos 2.4. Variables locales



© Rafael Rico López

## Procedimientos 2.4. Variables locales

- **Juntando todo:**
  - ➔ Pasamos argumentos mediante PUSH
  - ➔ Antes de comenzar las tareas del procedimiento se salva el antiguo marco (PUSH BP) y se actualiza el marco en curso (BP ← SP; MOV BP, SP)
    - ➔ el nuevo marco siempre apunta a la posición de la pila que almacena el antiguo (precedente)
  - ➔ Las variables locales se reservan a partir del nuevo marco moviendo el SP hacia la cima de la pila (SUB SP, n)
  - ➔ Los argumentos están por encima del marco: [BP+n]
  - ➔ Las variables locales están por debajo: [BP-n]

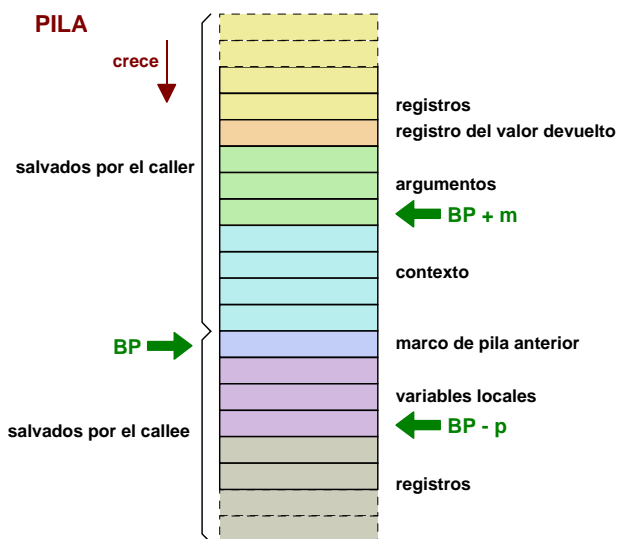
© Rafael Rico López

## Procedimientos Juntando todo

CALLER		CALLEE	
salvar reg. valor devuelto	PUSH AX		
salvar resto de registros	PUSH reg		
pasar argumentos según ABI	PUSH arg		
invocar procedimiento	CALL		
		salvar marco de pila	PUSH BP
		ajustar marco de pila nuevo	MOV BP, SP
<b>CÓDIGO DE COLORES</b>		declarar variables locales	SUB SP, n
preservar registros		salvar regs. (excepto v. dev.)	PUSH reg
<i>clean-up</i>		<b>CÓDIGO PROCEDIMIENTO</b>	
manejo de variables locales		recuperar registros	POP reg
		soslayar variables locales	MOV SP, BP
		recuperar marco de pila	POP BP
		retorno	RET
		soslayar argumentos	RET s
soslayar argumentos	ADD SP, s		
salvar valor devuelto	AX → var		
recuperar registros	POP reg		
recuperar reg. valor devuelto	POP AX		47/66

© Rafael Rico López

## Procedimientos Juntando todo



© Rafael Rico López

48/66



## Procedimientos

### 2.5. Uso de estructuras

---

- Algunos aspectos de la programación de procedimientos son muy propensos a error; concretamente:
  - ➔ Las referencias a parámetros
  - ➔ El tamaño de los parámetros a soslayar
- Con el fin de minimizar los errores, podemos usar la declaración de estructuras

## Procedimientos

### 2.5. Uso de estructuras

---

- La estructura representa el orden de los datos en la cima de la pila, desde el marco de pila hasta el último parámetro
- Permite referenciar cada uno de los parámetros por su etiqueta y calcular el tamaño de los mismos
  - ➔ Las etiquetas son el *offset* de cada parámetro respecto al marco de pila (BP)
- La **estructura no se reserva en memoria** ya que no se declara en el segmento de datos; sólo se usa para simplificar la programación del *callee*

## Procedimientos

### 2.5. Uso de estructuras

---

- Ejemplo:

```
                                ;cada miembro representa el espacio
                                ;que ocupa en la cima de la pila

parametros  STRUC
             DW ?  ;marco de pila BP      OFFSET 0
retorno     DD ?  ;puntero far           OFFSET 2
param1      DD ?  ;puntero far a cadena  OFFSET 6
param2      DW ?  ;valor del carácter    OFFSET 10
parametros  ENDS

size_params EQU $ - param1 ;tamaño de los parámetros
```

© Rafael Rico López

51/66

## Procedimientos

### 2.5. Uso de estructuras

---

- Ejemplo:

```
MOV BX, [BP].param2 ;carga carácter en BX
LES SI, [BP].param1 ;puntero far de la cadena

: : :

MOV SP, BP
POP BP
RET size_params ;retorno y soslayo argumentos
```

© Rafael Rico López

52/66

## Procedimientos

### 3. X86 *calling conventions*

---

#### 3. X86 *calling conventions*

##### Índice

1. Convenciones *caller clean-up*
2. Convenciones *callee clean-up*

## Procedimientos

### 3.1. Convenciones *caller clean-up*

---

- Admiten procedimientos con un número variable de argumentos

#### ➔ `cdecl` (*C declaration*)

- ➔ Todos los argumentos se pasan por la pila
- ➔ El orden de paso es RTL (*right to left*)
- ➔ Los valores devueltos se pasan en EAX
- ➔ Los registros EAX, ECX y EDX son *caller-saved*
- ➔ El resto de registros son *callee-saved*

#### ➔ `syscall`

#### ➔ `optlink`

## Procedimientos

### 3.2. Convenciones *callee clean-up*

---

- ➔ **pascal**
  - ➔ Todos los argumentos se pasan por la pila
  - ➔ El orden de paso es LTR (*left to right*)
- ➔ **fastcall**
  - ➔ Los 2 primeros argumentos por la izquierda se pasan en ECX y EDX
  - ➔ El resto de argumentos se pasan por la pila en orden RTL
- ➔ **Borland fastcall**
  - ➔ Los 3 primeros argumentos por la izquierda se pasan en EAX, EDX y ECX
  - ➔ El resto de argumentos se pasan por la pila en orden LTR
- ➔ **stdcall**
- ➔ **safecall**

## Procedimientos

### 4. Programación multimódulo

---

#### 4. Programación multimódulo

##### Índice

1. Directivas
2. Ensamblado

## Procedimientos

### 4. Programación multimódulo

- Programación multimódulo es aquella en la que el ejecutable es el resultado de enlazar varios módulos OBJ diferentes
  - ➔ Cualquier módulo puede utilizar nombres simbólicos del resto de módulos
  - ➔ Los nombres simbólicos pueden corresponder a declaración de variables o nombres de procedimientos
  - ➔ Es habitual que un módulo contenga la declaración de los procedimientos y otro el código principal
    - ➔ El módulo de procedimientos tendrá sólo segmento de código (a no ser que use variables estáticas)
    - ➔ El módulo de procedimientos puede usarse para crear **librerías**

© Rafael Rico López

57/66

## Procedimientos

### 4.1. Directivas

- Para que un nombre simbólico esté disponible a otros módulos se utiliza la directiva PUBLIC
- Ejemplo:

```
PUBLIC nombre_proc
PUBLIC nombre_variable
      : : :
DOSSEG
.MODEL SMALL
.DATA
nombre_variable DB 17
      : : :
.CODE
nombre_proc PROC NEAR
      : : :
nombre_proc ENDP
```

© Rafael Rico López

58/66

## Procedimientos

### 4.1. Directivas

- Para que un módulo utilice un nombre simbólico declarado en otro módulo se utiliza la directiva **EXTRN**

- Ejemplo:

```
EXTRN nombre_proc: far
EXTRN nombre_variable
      : : :
DOSSEG
.MODEL SMALL
      : : :
```

© Rafael Rico López

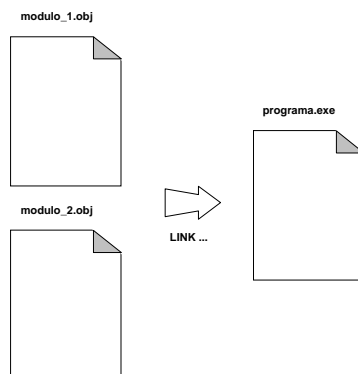
59/66

## Procedimientos

### 4.2. Ensamblado

- Para enlazar un ejecutable a partir de varios módulos **OBJ** invocamos el programa **link** pasándole como argumentos el listado de módulos

```
link [opciones] mod1.obj+mod2.obj, [nombre_ejecutable]
```



© Rafael Rico López

60/66

## Procedimientos

### 5. Librerías

---

#### 5. Librerías

##### Índice

1. Fundamentos
2. Manejando librerías con LIB

## Procedimientos

### 5.1. Fundamentos

---

- Una librería es una colección de procedimientos compilados (ensamblados) que proporcionan un conjunto de rutinas orientadas a un tipo de tarea
  - ➔ Manejo de cadenas de caracteres
  - ➔ Cálculos matemáticos
  - ➔ Presentación gráfica
  - ➔ Acceso al sistema de ficheros
  - ➔ ... etc.
- Las librerías se crean reuniendo en un único fichero diferentes objetos compilados separadamente

## Procedimientos

### 5.1. Fundamentos

---

- Cuando un procedimiento es suficientemente eficaz y genérico como para poder ser útil en diferentes tareas, se puede añadir a una librería de funciones semejantes
- Para ello necesitamos una herramienta que nos permita gestionar las librerías
  - ➔ Crear librería
  - ➔ Añadir módulos
  - ➔ Borrar módulos
  - ➔ Copias módulos
  - ➔ Reemplazar módulos
  - ➔ ... etc.

## Procedimientos

### 5.2. Manejando librerías con LIB

---

- En nuestro entorno de trabajo, la herramienta de gestión de librerías es `lib`
- La herramienta `lib` tiene como fuente ficheros objeto ya compilados (ficheros `.obj`)
- Cuando `lib` añade los ficheros objeto a un fichero de librería se convierten en **módulos**
- Los ficheros de librería tienen extensión `.lib`



## Procedimientos

### 5.2. Manejando librerías con LIB

- Programa `lib`

```
lib  libreria_antigua [comandos]
    [, [fichero_ref_cruzadas]
    [, [libreria_nueva]] [;]
```

- ➔ `librería_antigua` → fichero de librería a modificar
  - ➔ Extensión por defecto: `.lib`
- ➔ `comandos` → +(añadir); -(reemplazar); \*(copiar); -(mover)
- ➔ `fichero_ref_cruzadas` → listados de símbolos públicos y de módulos
- ➔ `librería_nueva` → fichero de librería nuevo (si se desea crear uno nuevo con las modificaciones)
- ➔ `;` → evita que nos proponga los nombres de los ficheros

© Rafael Rico López

65/66

## Procedimientos

### 5.2. Manejando librerías con LIB

- Ejemplo:

```
lib mi_lib + mi_objeto;
```

- ➔ Este comando crea la librería (o añade a la librería, si ya existe) `mi_lib.lib` el módulo del fichero objeto `mi_objeto.obj`
- ➔ Si ahora queremos enlazar algún procedimiento público de dicha librería con un fichero objeto para crear un ejecutable, haremos

```
link nuevo_objeto,,mi_lib;
```

© Rafael Rico López

66/66