7. Los procedimientos

Los procesadores tienen el equivalente a los procedimientos de los lenguajes de alto nivel. La idea es la misma: tenemos dos o más grupos de instrucciones, y queremos, desde uno de ellos, ejecutar el otro. Pero, a diferencia del `jmp`, cuando se acabe de ejecutar el segundo grupo de instrucciones, queremos regresar al punto donde estábamos en el primero (fig. 7.1).

Esto se logra por medio de una instrucción parecida a las instrucciones de salto. Dicha instrucción, además de efectuar el salto, almacena la dirección donde se encuentra en ese momento (el `eip`). El procedimiento debe terminar con una instrucción que recupera la dirección guardada y la repone en el `eip`. El efecto neto es el regreso al punto donde estaba antes de llamar al procedimiento.

Pero, ¿dónde almacenar la dirección de retorno? La sección que sigue responde a esta pregunta.

7.1 La pila de la máquina

La recuperación de las direcciones de retorno no es tan simple, sobre todo si se considera una generalización del mecanismo: el procedimiento 1 llama al 2, el cual llama al 3, etc. En este caso, se tendrá que almacenar dos direcciones de retorno. Estudiemos cómo debe ser el funcionamiento del mecanismo:

Procedimiento 1 llama a procedimiento 2 ⇒ debemos guardar dirección 1.
Procedimiento 2 llama a procedimiento 3 ⇒ debemos guardar dirección 2.
Procedimiento 3 termina ⇒ debemos recuperar dirección 2.
Procedimiento 2 termina ⇒ debemos recuperar dirección 1.

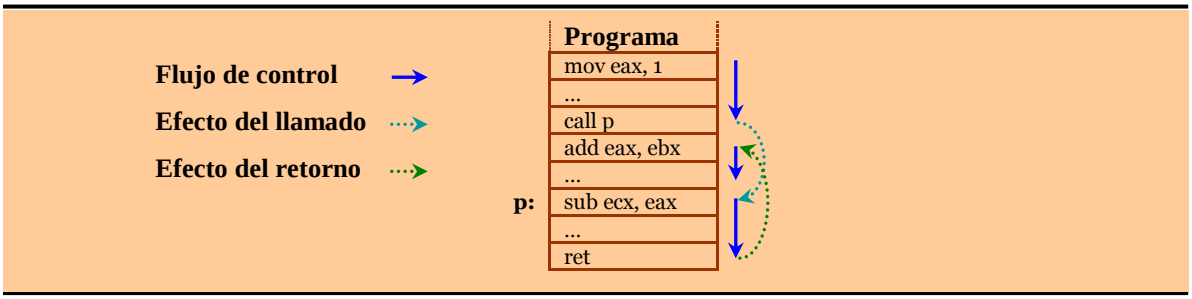


Fig. 7.1. Llamado de procedimientos.

Puede notar que el mecanismo de llamada-regreso funciona como una pila. Esta es la solución que se toma: en una zona de la memoria, se tiene una pila manejada por el hardware; al hacer un llamado, la dirección se guarda en el tope de la pila; cuando se retorna, la dirección se obtiene del tope de la pila. El tope de la pila lo señala el registro `esp` (*Extended Stack Pointer*).

La instrucción de llamada de procedimiento es `call`. Tiene un operando que es la dirección a donde se quiere ir (igual que el `jmp`). Como dijimos anteriormente, guarda el `eip` (la dirección donde está) en la pila, y pone la dirección del procedimiento en el `eip`, produciendo de esta manera el llamado. La instrucción de retorno es `ret`: toma lo que está en el tope de la pila —que debe ser la dirección desde donde se hizo el llamado— y se lo asigna al `eip`.

Veamos en detalle cómo funciona la pila. Dentro del área de memoria asignada a la pila, `esp` apunta al tope de la pila. La pila de la IA32 es decreciente, es decir, `esp` apunta al final del área; a medida que se guardan valores, `esp` se decrementa; cuando se sacan, `esp` aumenta. En la figura 7.2, se puede ver un ejemplo.

Tenemos entonces que, cuando se ejecuta "`call` dirección", las acciones tomadas por la máquina son:

`esp` ← `esp` - 4
[`esp`] ← `eip`
`eip` ← dirección del procedimiento

La instrucción RET efectúa las acciones que se describen a continuación:

`eip` ← [`esp`]
`esp` ← `esp` + 4

Hay otras instrucciones que permiten guardar valores temporalmente en la pila. Estas instrucciones son `push` y `pop`, las dos tienen un operando. `push` decrementa el `esp` y después guarda su operando en la pila. `pop` le asigna el tope de la pila a su operando, y después incrementa el `esp`. Las dos aceptan varios modos de direccionamiento —directo, indirecto, etc. También inmediato, en el caso del `push`—.

Por supuesto, `esp` es un registro; así que puede ser manipulado como cualquier otro registro. En particular, se le puede sumar o restar valores de manera que se desplace sobre la pila. También, usando indirección, es posible sacar valores de la

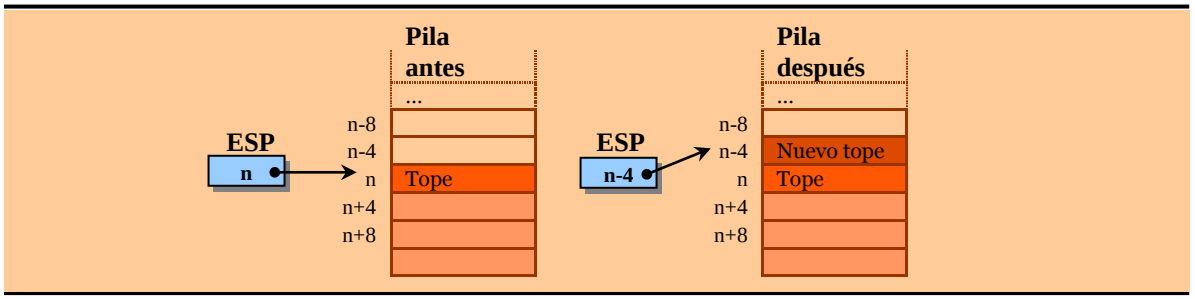


Fig. 7.2. Funcionamiento de la pila.

pila directamente; por ejemplo: "`mov eax, [esp]`" le asigna a `eax` lo que está en el tope de la pila.

7.2 USO DE LOS PROCEDIMIENTOS

En primer lugar, un programa se divide en *regiones*. Las regiones se especifican por medio de las directivas al ensamblador[1]:

`.data` Es una región de datos; es donde se declaran las variables. Las variables pueden tener valores iniciales.

`.data?` También es una región de datos, pero las variables no pueden tener valores iniciales —se inician con: ?—.

`.const` Otro tipo de región de datos: las variables no pueden ser modificadas, lo cual las convierte en constantes.

`.code` Es una región de código; es donde se escriben las instrucciones.

Una región se extiende desde el punto donde aparece la directiva hasta la siguiente directiva de región. Puede haber varias regiones del mismo tipo —varias .data, .code, etc.—. El fin de todo programa se indica con el comando `end`.

En las regiones de código —`.code`— se declaran los procedimientos. La sintaxis para escribir un procedimiento en ensamblador es la siguiente:

```
nombre proc
 ;Aquí van las instrucciones
 …
nombre endp
```

Recuerde que la última instrucción ejecutada por un procedimiento debe ser `ret`, no necesariamente tiene que escribirse al final del procedimiento —antes del `endp`—, pero toda secuencia de ejecución sí debe terminar en un `ret`.

El programa principal se declara como un procedimiento. Al ensamblador se le indica cuál es el programa principal colocando su nombre después del `end`, como se puede ver en el ejemplo que se muestra en el prog. 7.1.

Note que el programa principal no acaba en `ret` puesto que debe devolver el control al sistema operacional. Este retorno incluye varias acciones de las cuales se encarga la función de Windows *`ExitProcess`*. `NULL` es una constante que vale cero, y le indica a Windows que el programa terminó normalmente —sin errores—. En cuanto al invoke, es una forma de llamar la función *`ExitProcess`* pasándole como parámetro el código de retorno — `NULL`, en este caso—.

---

[1] Una *directiva al ensamblador* es un comando que no corresponde a instrucciones, no genera código, sino a indicaciones que se le dan al ensamblador sobre cómo generar el código del programa.

Cualquier procedimiento puede llamar a cualquier otro procedimiento, incluso se pueden hacer llamados recursivos. Además, igual que los `jmp`, se puede hacer `call` indirecto del estilo:

```
call variable
;variable debe tener un apuntador a un procedimiento
call ebx
call [ebx]
```

```
.data
  cadena BYTE 50 DUP (?)

.code
  principal PROC;El programa principal.

    lea  esi, cadena
    call leerCadena
    call longitud
    ;Retornar a Windows; fin del programa principal.
    invoke ExitProcess, NULL

  principal ENDP

  longitud PROC ;Retorna en eax la longitud de una cadena de
                ;caracteres apuntada por esi.
    mov eax, 0
    contarBytes:
    cmp byte ptr [esi+eax], 0
    je  finLongitud   ;¿[esi+eax] = 0? (fin de la cadena)
                      ;[esi+eax] ≠ 0
      inc  eax
      jmp  contarBytes
    finLongitud:;[esi+eax] = 0. eax = longitud
    ret

  longitud ENDP

  leerCadena PROC     ;Este procedimiento debería lee una cadena
          …           ;de caracteres en la zona apuntada por esi
  leerCadena ENDP     ;pero no la vamos a especificar.

  END principal       ;"Principal" es el programa principal.
```

Prog. 7.1 Ejemplo de procedimientos

```
call v[esi]
```

Para generar un apuntador a un procedimiento, se procede como con los apuntadores a variables; por ejemplo, si se ha declarado un procedimiento *p*:

```
mov ebx, offset p
```

O, equivalentemente:

```
mov ebx, p
```

También se puede hacer en declaraciones:

```
variable        dword     offset p
```

O, equivalentemente:

```
variable        dword     p
```

Los `call` indirectos son bastante útiles; en particular, en la programación orientada por objetos, sirven para implementar los métodos. En efecto, un objeto se puede representar por medio de una estructura cuyos campos corresponden a los atributos; los métodos también pueden ser campos de una estructura que tienen un apuntador al procedimiento que implementa el método en cuestión:

```
S struct
  atributo_1    dword     ?
      …
  atributo_n    dword     ?
  metodo_1 dword     offset proc_1
      …
  metodo_m dword     offset proc_m
S ends
```

Luego, si se tiene un objeto *o* de tipo *S*, el llamado de un método se hace por medio de un `call` indirecto a través del campo respectivo; por ejemplo:

```
call o.metodo_1
```

Esta es una aproximación bastante simple; en realidad, para manejar aspectos como herencia y polimorfismo, se necesitan estructuras un tanto más complejas.

7.3 CONVENCIONES DE LLAMADO

Cuando pasa el control de un procedimiento a otro, es necesario que los procedimientos estén de acuerdo en una serie de convenciones que permiten pasar información de uno al otro, y que impiden que se estorben mutuamente; en esta sección, veremos algunos de los aspectos que se deben tener en cuenta.

Paso de parámetros

Uno de los primeros aspectos en que deben estar de acuerdo los procedimientos es en cómo se pasan los parámetros de uno al otro. Esto se puede hacer por los registros y por la pila.

Paso de parámetros por los registros

La forma más simple de pasar parámetros es por los registros, como hicimos en el primer ejemplo: pasamos la dirección de la cadena en el registro `esi`. Por supuesto, los procedimientos deben estar de acuerdo en cuáles registros usar y en qué parámetro va en cada uno de ellos.

Este método tiene dos defectos: en primer lugar, complica los procedimientos recursivos, puesto que, al hacer la llamada recursiva, habría que salvar el parámetro recibido para poder cargar el nuevo parámetro. En segundo lugar, si hay muchos parámetros, los registros pueden ser insuficientes.

El primer problema se puede resolver así: antes de hacer el llamado recursivo, se guarda el parámetro en la pila por medio de la instrucción `push`.[2]

Como ejemplo, el prog. 7.2 presenta el factorial recursivo. Este procedimiento recibe el parámetro en el registro `ebx` y retorna el resultado en el registro `eax`.

A todo `push` le debe corresponder un `pop`, de lo contrario se dejaría basura en la pila. Esto es muy importante, pues si en un procedimiento se deja basura en la pila, el `ret` interpretará la basura como si fuera la dirección de retorno; el resultado de lo cual no será particularmente razonable.

```
;Precondición: ebx ∈ ℕ
;Poscondición: eax = ebx!

factorial PROC          ;ebx = n
  cmp ebx, 1
  jle  resultado1       ;¿n ≤ 1?
    push  ebx           ;preservar n
    dec   ebx           ;ebx = n-1
    call  factorial
                        ;eax = (n-1)!
    pop   ebx           ;ebx = n
    imul  eax, ebx      ;eax = n!
    ret
  resultado1:           ;n! = 1
    mov  eax, 1
  ret
factorial ENDP
```

Prog. 7.2 Factorial recursivo

[2] De hecho, este método funciona en general: siempre que se desee preservar un valor, de un registro o de una variable, se puede guardar en la pila para restaurarlo posteriormente.

Paso de parámetros por la pila

Todavía queda un problema: ¿qué hacer si hay más parámetros que registros? Se puede utilizar la pila pero de una manera diferente. En lugar de salvar los registros en ella, puede ser usada para pasar los parámetros. El procedimiento llamador pone los parámetros en la pila e invoca al otro, por medio de código del estilo:

```
push  parámetro_m
 …
push  parámetro_1
call  procedimiento
```

En la fig. 7.3(a) se puede ver el estado de la pila inmediatamente antes del `call`; en la fig. 7.3(b), el estado inmediatamente después del `call`, es decir, como la recibe el procedimiento llamado. Note que la dirección de retorno dejada por el `call` se encuentra encima de los parámetros; también note que, puesto que se trata de una pila, los parámetros se meten en orden "inverso" a como se quiere que queden.

La pregunta ahora es cómo hace el procedimiento llamado para utilizar los parámetros. La respuesta se encuentra en el direccionamiento basado: el primer parámetro se encuentra a un desplazamiento de 4 con respecto al `esp`; el segundo, a un desplazamiento de 8 y así hasta el último que se encuentra a 4**m*. En consecuencia, los parámetros se podrían referenciar como [esp+4], [esp+8], etc. Así, para cargar el primer parámetro en `eax`, se escribiría:

```
mov eax, [esp+4]
```

Algunos compiladores direccionan de esta manera, pero otros no consideran conveniente, o cómodo, direccionar con respecto a `esp`, ya que `esp` se mueve bastante —para guardar valores temporalmente en la pila, para guardar parámetros durante un llamado, etc.—, y esto implica que los desplazamientos cambian dinámicamente. Si no se desea manejar este cambio dinámico, se puede usar el `ebp`; para esto, al comienzo del procedimiento, se le asigna `esp` a `ebp`[3]:

```
mov ebp, esp
```

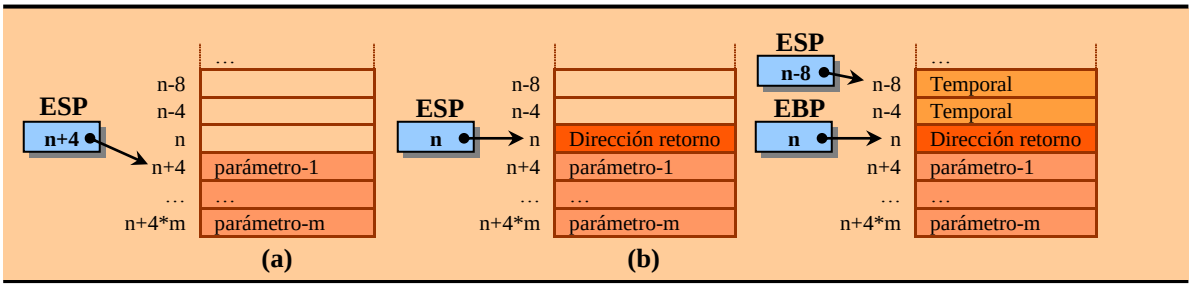


Fig. 7.3. Estado de la pila: (a) antes del call, (b) después del call, (c) durante el call.

[3] ¿Por qué el `ebp` y no otro registro? En realidad no hay una razón de fondo: un poco por tradición, un poco por razones históricas; en el Intel 8086 sí era importante usar el `bp`. En las circunstancias actuales podría ser otro, pero, dadas las características de la IA32, es mejor mantener esta convención.

Con lo cual la situación queda como se muestra en la fig. 7.3(c); donde se supone que `esp` se ha desplazado, pero los parámetros se pueden referenciar como `[ebp+4]`, `[ebp+8]`, etc. En estas circunstancias, podemos obtener el primer parámetro escribiendo "`mov eax, [ebp+4]`"; en general, obtenemos el *i*-ésimo parámetro escribiendo: "`mov eax, [ebp+4*i]`".

Si bien el anterior mecanismo nos da flexibilidad, introduce un inconveniente: si el procedimiento *p1* llama a otro procedimiento *p2*, este también utilizará `ebp` para manejar sus parámetros; al retornar de *p2*, `ebp` no estaría donde lo puso *p1*. Es necesario, entonces, proteger `ebp`; lo cual se logra guardándolo en la pila al comienzo, con lo cual el código del comienzo es:

```
push ebp
mov ebp, esp
```

Y, por supuesto, al final, antes del `ret`, se debe hacer el `pop` correspondiente.

Note que este mecanismo implica que los desplazamientos de los parámetros se aumentan en 4, puesto que ahora `ebp` está en la pila encima de la dirección de retorno.

A manera de ejemplo, el prog. 7.3 calcula el máximo entre sus parámetros, devolviendo el resultado en `eax`. El programa que llame a este procedimiento debe guardar los parámetros en la pila. Por ejemplo, si queremos obtener el máximo entre `ebx` y una variable *i*, debemos escribir:

```
push  ebx
push  i
```

```
;Precondición: [esp+4] y [esp+8] son números enteros
;Poscondición: eax = Max([esp+4], [esp+8])

maximo PROC           ;ebx = n

  push  ebp           ;salvar ebp, ya que los vamos a modificar.
  mov  ebp, esp       ;parametro1=[ebp+8], parametro2=[ebp+12]
  mov  eax, [ebp+8]   ;eax = parametro1
  cmp  eax, [ebp+12]
  jge  finMaximo;¿parametro1 ≥ parametro2?
                      ;parametro1 < parametro2
    mov  eax, [ebp+12]     ;eax = parametro2
  finMaximo:          ;eax = max (parametro1, parametro2)
  pop  ebp            ;recuperar ebp anterior
  ret

maximo ENDP
```

Prog. 7.3 Máximo de dos números

```
call  maximo          ;eax = max (ebx, i)
```

Al terminar el procedimiento *maximo*, los parámetros todavía están en la pila. Alguien debe encargarse de desocupar la pila; esto lo puede hacer tanto el procedimiento llamado como el llamador.

El procedimiento llamador puede desocupar la pila avanzando ESP manualmente:

```
add   esp, 4*n
;Siendo n una constante  (el número de parámetros).
```

El procedimiento llamado también puede desocupar la pila por medio de una variante de la instrucción RET:

```
ret  4*n ;n es una constante (el número de parámetros).
```

Esta variante de `ret`, además de recuperar la dirección de retorno, suma su operando a `esp`, con lo cual quedan descartados los parámetros.

Es de anotar que, así como se puede mover el `esp` manualmente para vaciar la pila, también se puede mover para meter los parámetros, con código del estilo:

```
sub   esp, 8
mov  [esp], eax
mov  [esp+4], ebx
call  procedimiento
```

En realidad esto no es muy útil para parámetros escalares, pero facilita pasar entidades más complejas como vectores, cadenas de caracteres y estructuras.

Parámetros por referencia

Varios lenguajes de programación tienen dos formas de pasar los parámetros: por valor y por referencia.

El paso por valor es el que hemos venido tratando: el parámetro es una copia del valor del argumento pasado. Esto implica que, si el procedimiento modifica al parámetro, el argumento original se mantiene intacto.

Cuando se pasan parámetros por referencia, el parámetro hace referencia al argumento original. En consecuencia, si el procedimiento modifica al parámetro, en realidad está modificando al argumento original y no una copia del mismo.

Para implementar el paso por referencia, se pasa como parámetro un apuntador al argumento; esto le permite al procedimiento llamado modificar el argumento mismo. Los parámetros por referencia son una especie de "apuntadores escondidos", con "indirección automática"; es decir, el programador los utiliza como variables comunes y corrientes, pero, en realidad, son apuntadores y cada uso implica una indirección.

A manera de ejemplo, veamos un procedimiento que recibe dos números (x y y), y retorna el cociente de la división x/y. Los números x y y los vamos a pasar por valor, y el cociente por referencia (prog. 7.4).

Para invocar este procedimiento, el procedimiento llamador debe tener las instrucciones que siguen:

```
push  offset c ;parámetro por referencia: dirección de c.
push  y        ;parámetro por valor: segundo número.
push  x        ;parámetro por valor: primer número.
call  division
add   esp, 12  ;desocupar parámetros de la pila.
```

Note que los parámetros por referencia son una abstracción que crean los lenguajes de alto nivel para facilitar el desarrollo de programas, pero, en ensamblador, no hay diferencia entre pasar un parámetro por referencia o pasar un apuntador a una variable. En realidad, sería más preciso decir que, en ensamblador, no existen los parámetros por referencia.

Hay otras formas de pasar parámetros que tienen un parecido con el paso por referencia, sin embargo, dado que tienen efectos colaterales distintos, no se consideran paso por referencia. Estas formas son:

- Paso por valor-resultado: el procedimiento llamado modifica los parámetros en la pila. El procedimiento llamador la desocupa, pero, al hacerlo, toma los valores de la pila y se los asigna de vuelta a las variables originales.
- Paso por nombre: no se pasa ni el valor del argumento, ni un apuntador al mismo; en su lugar, se pasa un apuntador a un procedimiento que calcula y retorna la dirección del parámetro. Esto hace que el argumento pueda cambiar dinámicamente durante la ejecución del procedimiento.

Para ilustrar estos dos métodos vamos a suponer que tenemos un procedimiento *p* que recibe un parámetro *x*. Si el parámetro se pasa por valor-resultado, no hay que

```
;Precondición: [esp+4] y [esp+8] son números enteros.
               [esp+12] es un apuntador a un entero
;Poscondición: [[esp+12]]= [esp+4]/[esp+8]

division  PROC

  push  ebp           ;salvar ebp, ya que los vamos a modificar.
  mov   ebp, esp      ;x=[ebp+8], y=[ebp+12]
  mov   eax, [ebp+8]  ;eax = x
  mov   edx, 0        ;la división se efectúa sobre edx:eax.
  idiv  [ebp+12]      ;eax = x/y
  mov   ecx, [ebp+16] ;ecx = apuntador a la variable resultado.
  mov   [ecx], eax    ; [[esp+16]]= x/y
  pop   ebp
  ret

division  ENDP
```

Prog. 7.4 Paso de parámetros por referencia

hacer nada especial en el procedimiento llamado; en cuanto al llamador, haría lo siguiente:

```
push  x    ;pasa el parámetro por valor.
call  p    ;p posiblemente modifica el valor en la pila.
pop   x
;desocupa la pila.
;Asigna a x el valor dejado por p en la pila.
```

Si el parámetro se pasa por nombre, el procedimiento llamador, haría lo siguiente:

```
push  offset d  ;d es un procedimiento.
call  p
add   esp, 4    ;desocupa la pila.
```

El procedimiento *d*, hace lo siguiente:

```
mov   eax, offset x  ;eax apunta a x.
ret
```

En cuanto al procedimiento *p*, cada vez que necesite usar el parámetro, hace lo siguiente:

```
call  [ebp+8]
;llama a d. d retorna en eax un apuntador al argumento.
```

Después de lo cual puede usar `[eax]` para leer o escribir en el argumento. ¿Cuál es la diferencia con el paso por referencia? En este caso ninguna, pero, si el argumento es *v*[*i*], el procedimiento *d* sería:

```
mov   eax, offset v  ;eax apunta al vector.
add   eax, i         ;eax apunta a v[i].
ret
```

Ahora bien, si el procedimiento *p* modifica la variable *i*, cada vez que llame a *d*, obtendrá un apuntador a un *v*[*i*] diferente; el argumento cambiará dinámicamente.

Salvaguardia de los registros

Como mencionamos anteriormente, es necesario preservar `ebp` en la pila al comienzo y recuperarlo al final, puesto que puede ser modificado por otras invocaciones que se hagan desde el procedimiento que está ejecutando.

Esto se puede generalizar y aplicar a todos los registros que se utilicen en el procedimiento: si se invocan otros procedimientos, estos podrían modificar los registros que están siendo usados en el procedimiento llamador; por supuesto, se generarían inconsistencias en este procedimiento.

En consecuencia, cuando se invoca un procedimiento es necesario tomar alguna de las siguientes medidas para proteger los registros:

- No usar los mismos registros en el procedimiento llamado y en el llamador.
- Guardar los registros que están siendo usados en el llamador antes de hacer la llamada. Para esto, se pueden guardar los registros en la pila antes de la llamada, y recuperarlos después del retorno —salvaguardia en el llamador—.

- Al comienzo del procedimiento llamado, guardar en la pila los registros que este va a utilizar y recuperarlos inmediatamente antes de retornar —salvaguardia en el llamador—.

La primera aproximación es difícil de garantizar cuando se tienen pocos registros o cuando se hacen muchas invocaciones anidadas —los registros se agotan—. En cuanto a las otras dos, la salvaguardia se hace "a ciegas", con lo cual se podría estar desperdiciando trabajo: se podrían estar protegiendo registros que, en realidad, no son usados por el otro procedimiento.

Por supuesto, también se puede recurrir a estrategias mixtas: algunos registros se guardan en el llamador —solo si los está utilizando—; el procedimiento llamado intenta usar esos mismos registros, y, si necesita más, es su responsabilidad proteger el contenido previo.

Con esto, el código al comienzo de un procedimiento queda así:

```
push ebp
mov ebp, esp
push reg1
 …
push regn  ;regi son los registros que se van a preservar.
```

Note que los registros se salvaron después de asignar `esp` a `ebp`. Se podría hacer antes, pero no es conveniente por la siguiente razón: si se hace antes, los registros quedarían encima de los parámetros, lo cual haría que los desplazamientos del `ebp` hasta los parámetros dependiera del número de registros guardados en la pila; esto dificulta la labor del compilador —o del programador—. Por otro lado, la idea de tener dos apuntadores a la pila, `ebp` y `esp`, es justamente para que uno se encargue de lo estático —`ebp`— y otro se encargue de lo dinámico —`esp`—, y el número de los registros guardados puede cambiar dinámicamente.

El código al final del procedimiento se convierte en:

```
pop regn
 …
pop reg1   ;los regi son los registros que se preservaron.
pop ebp
ret
```

Retorno de valores

Hasta el momento hemos tratado el caso de procedimientos que no retornan valores. ¿Cómo tratar el caso de las funciones?

Se aplica todo lo visto hasta el momento, pero, además, se debe decidir cómo retornar valores. Las dos opciones típicas son: por la pila y por registros.

- Por registros: se elige un registro en el cual el procedimiento llamado —la función— deja el resultado. Por ejemplo, en el prog. 7.3, usamos `eax` para esta tarea. Este método solo sirve para los valores escalares —y lo que es más, para escalares de máximo 32 bits—.

- Por la pila: el procedimiento llamador separa espacio para el resultado —posiblemente en la pila— y pasa un parámetro extra con un apuntador a este espacio separado. El procedimiento llamado calcula el resultado y lo deja donde le indica el apuntador. Se podría decir que se crea un parámetro por referencia adicional que apunta al sitio donde se deja resultado.

Es de anotar que las dos opciones no son excluyentes; se puede retornar los valores escalares por un registro y las estructuras por la pila.

Algunas convenciones usuales

Los diversos compiladores y sistemas toman decisiones diferentes a la hora de elegir entre los mecanismos antes mencionados. En esta sección mostraremos algunas de las posibilidades.

Orden de los parámetros

En los lenguajes de alto nivel, se hacen invocaciones de procedimiento del estilo:

p (*parámetro*$_1$, … , *parámetro*$_n$)

Es necesario decidir el orden en que se pasan los parámetros en la pila: *parámetro*$_1$ en el tope y *parámetro*$_n$ en el fondo, o al contrario. En principio, no importa mucho cuál convención se adopte, mientras se sea consistente con la decisión adoptada; es decir, mientras el procedimiento llamador y el llamado estén de acuerdo en la misma convención.

En la práctica, se encuentran las dos posibilidades, dependiendo del lenguaje. Por ejemplo, las implementaciones de Pascal suelen dejar el *parámetro*$_n$ en el tope; en el caso de C, el *parámetro*$_1$ se guarda en el tope. Esta decisión no se tomó arbitrariamente: C permite manejar procedimientos con un número variable de parámetros[4], es decir, procedimientos que, en cada invocación, pueden recibir un número diferente de parámetros. En estas condiciones, es necesario que el primer parámetro esté en el tope; así, por lo menos, se puede identificar cuál es el primer parámetro. Si fuera al contrario, ni siquiera se sabría cuál parámetro está en el tope —en realidad, no se podría identificar ningún parámetro—.

El API —*Application Programming Interface*— de Windows funciona con las convenciones de C.

Paso por la pila y por registros

En muchas máquinas el paso de parámetros es por la pila; no se utilizan los registros puesto que limitan el número de parámetros que puede tener un procedimiento. Esta aproximación permite manejar cualquier cantidad de parámetros, pero se pierde algo de eficiencia puesto que los parámetros están en memoria.

Sin embargo, algunas máquinas, notoriamente los RISC —*Reduced Instruction Set Computer*—, usan un esquema mixto: los primeros parámetros se pasan por los registros, los restantes por la pila. Es una buena aproximación, ya que, si el

[4] Este es el caso, por ejemplo, de las funciones *printf* y *scanf*.

procedimiento tiene pocos parámetros, se podrán pasar por los registros y funcionará eficientemente; si tiene muchos, de todas maneras no habrá problemas para pasarlos, así sea perdiendo algo de eficiencia. Debemos anotar que estadísticamente la mayoría de procedimientos tiene pocos parámetros, así que, en general, se usarán los registros.

Vaciado de la pila

Como se mencionó anteriormente, los parámetros se pueden sacar en el procedimiento llamado o en el llamador.

En la práctica, se encuentran las dos posibilidades: en Pascal, el procedimiento llamado desocupa la pila; en C lo hace el llamador. La razón es la siguiente: en Pascal, el número de parámetros de un procedimiento es fijo; en consecuencia, el procedimiento llamado puede encargarse de desocupar la pila porque sabe cuánto ocupan exactamente. En C, dado que puede haber procedimientos con un número variable de parámetros, debe ser el llamador quien desocupe la pila, ya que sólo él sabe cuántos parámetros pasó.

En el API de Windows, en general, desocupa la pila el llamado. Sin embargo, hay una función del API, parecida a *printf*, donde es necesario que sea el llamador quien desocupa la pila.

Parámetros por referencia

Como se mencionó anteriormente, en varios lenguajes de alto nivel se manejan dos tipos de parámetros: por valor y por referencia. Cuando se diseña un lenguaje de alto nivel es necesario tomar varias decisiones al respecto.

En primer lugar, es necesario decidir si se va a soportar los dos tipos de parámetros o solo uno de ellos y, en ese caso, cuál. Pascal, por ejemplo, soporta los dos; C solo tiene parámetros por valor[5]; PL/I, solo por referencia.

En segundo lugar, es necesario definir la semántica precisa del paso de parámetros. Por ejemplo, Occam considera que los parámetros por valor son "valores" y no "variables", por ende, no pueden ser modificados por el procedimiento llamado. En cuanto a los parámetros por referencia, es necesario decidir si se permite pasar expresiones o solo variables; PL/I, por ejemplo, permite el paso de expresiones, en cuyo caso el procedimiento llamador crea una nueva variable temporal —anónima—, le asigna la expresión y la pasa por referencia.

C solo maneja parámetros por valor; para modificar una variable externa desde un procedimiento, es necesario pasar un apuntador a la misma, y hacer la indirección explícitamente. En C++ se decidió incluir las referencias, con lo cual C++ tiene paso por valor y por referencia.

---

[5] Esta afirmación puede parecer extraña: ¿acaso los vectores no se pasan por referencia? Podríamos decir que se trata de una excepción; o, también, que el concepto de base en C no es el de "vector" sino el de "apuntador"; cuando se está pasando un "vector por referencia", en realidad se está pasando un apuntador por valor.

En los cuadros que se encuentran a continuación, se compara el paso por valor, por referencia y de un apuntador a una variable:

Valor		Apuntador		Referencia	
C	**Ensamblador**	**C**	**Ensamblador**	**C**	**Ensamblador**
Declaración					

void f (int **i**)
{
...
i = 17;
... }
	f proc
	...
	mov **[ebp+8]**, 17
	...
	f endp
		void f (int * **i**)
		{
		...
		***i** = 17;
		i = NULL;
		...}
			f proc
			...
			mov esi, **[ebp+8]**
			mov **[esi]**, 17
			mov **[ebp+8]**, 0
			...
			f endp
				void f (int **&** **i**)
				{
				...
				i = 17;
				...}
					f proc
					...
					mov esi, **[ebp+8]**
					mov **[esi]**, 17
					...
					f endp

Invocación					
f (**k**);	push **k** call f	f (**&** **k**);	push **offset k** call f	f (**k**);	push **offset k** call f

Salvaguardia de los registros

Se utilizan todos los métodos: salvaguardia en el llamador, en el llamado y mixta. El método mixto es usado con frecuencia en los RISC.

Los RISC plantean el asunto un tanto distinto, pero acaba siendo lo mismo. Por convención se definen registros de dos tipos: los *registros de trabajo* son registros que cualquier procedimiento puede utilizar en cualquier momento —están disponibles—; los *registros locales* son para mantener valores locales que tienen una cierta permanencia —como variables locales del procedimiento—. Cuando un procedimiento invoca a otro, sabe que este último puede, eventualmente, modificar los registros de trabajo; pero espera que no modifique los registros locales.

En consecuencia, si un procedimiento tiene valores en los registros de trabajo que le interesa conservar, debe preservarlos en la pila antes de invocar a otro procedimiento. Por otro lado, si un procedimiento desea utilizar los registros locales, antes de hacerlo debe preservar el valor anterior en la pila.

En el caso de Windows, por convención, los registros `eax`, `ecx` y `edx` son de trabajo —con salvaguardia del llamador—; los registros `ebx`, `ebp`, `esi` y `edi` son considerados locales —con salvaguardia del llamado—. Como hemos visto, `ebp` tiene un papel especial, pero lo podemos considerar como local.

Retorno de valores

Con frecuencia se mezclan los métodos de retorno por registro y por la pila. Por ejemplo, en Windows, los valores de 8 bits se retornan en `al`, los de 16 en `ax`, los de 32 en `eax`, los de 64 en la pareja `edx:eax`.

Para las entidades que midan más de 64 bits —en particular, las estructuras—, el llamador separa una zona de memoria para el resultado, y le pasa al llamado, como un parámetro adicional, un apuntador a esa zona. El procedimiento llamado usa el apuntador para dejar el resultado en la zona de memoria elegida por el llamador.

7.4 VARIABLES LOCALES

En los lenguajes de alto nivel, los procedimientos manejan variables que les son propias —variables locales— y que solo existen mientras el procedimiento está ejecutando. En esta sección estudiaremos la forma de implementarlas.

Una posibilidad consiste en declararlas en la sección de datos, como si fueran variables globales. Esto tiene varios inconvenientes: si se está programando en ensamblador, no es una buena metodología de programación —las variables y el

```
;Precondición: [esp+4] es un número entero.
;Poscondición: eax = fibonacci ([esp+4])

fibonacci  PROC

  push  ebp
  mov   ebp, esp;n = [ebp+8]
  sub   esp, 4        ;variable local =[ebp-4]
  mov   eax, [ebp+8]  ;eax = n
  cmp   eax, 1
  jle   retornar1     ;¿n ≤ 1?
                      ;n > 1
    dec   eax         ;eax = n-1
    push  eax
    call  fibonacci   ;eax = fibonacci (n-1)
    mov   [ebp-4], eax     ;variable local = fibonacci(n-1)
    mov   eax, [ebp+8]
    sub   eax, 2;eax = n-2
    push  eax
    call  fibonacci   ;eax = fibonacci(n-2)
    add   eax, [ebp-4]     ;eax = fibonacci(n-2)+fibonacci(n-1)
    jmp   finFibonacci
  retornar1:
  mov   eax, 1
  finFibonacci:
  add   esp, 4        ;devolver el espacio de variables locales.
  pop   ebp
  ret   4             ;retornar y vaciar pila

fibonacci  ENDP
```

Prog. 7.5 Fibonacci: variable local en la pila

código están separados—; por otro lado, no es posible hacer llamadas recursivas, ya que al hacerlas se dañarían las variables de la llamada anterior. Por último, malgasta memoria, ya que habría que separar espacio en memoria para todas las

variables de todos los procedimientos del programa aunque el procedimiento en cuestión no haya sido llamado.

Por estas razones, las variables locales se crean en la pila, separándoles espacio cada vez que se invoca el procedimiento, y manejándolas con direccionamiento basado. A manera de ejemplo, el prog. 7.5 calcula recursivamente el n-ésimo número de Fibonacci. Como se hacen dos llamados recursivos, se crea una variable local para guardar el resultado del primer llamado mientras se efectúa el segundo.

"`sub esp, 4`" se encarga de abrir campo en la pila para una variable local. Al desplaza el `esp`, queda un espacio libre en la pila. En el caso general, se debe restar 4**m* a `esp` (donde *m* es el número de variables locales). Al final hay que sumarle la misma cantidad a `esp` para retornar el espacio separado en la pila.

Los compiladores suelen separar espacio para las variables locales como se muestra en la fig. 7.4(a). En este esquema, se usa direccionamiento basado con desplazamiento positivo para los parámetros, y con desplazamiento negativo para las variables locales. También es posible hacerlo como en la fig. 7.4(b): desplazamiento positivo para todos; en este caso, primero se separa el espacio para las variables locales y después se le asigna el `esp` al `ebp`. En general se prefiere el primer esquema porque genera desplazamientos más cortos —sean ellos negativos o positivos—.

Vectores y estructuras locales

La creación de vectores y estructuras locales es parecida a la de los escalares: se separa espacio en la pila restándole el tamaño de la entidad al `esp`. En el caso de los vectores, se multiplica el número de elementos por el tamaño de cada elemento. Para la estructura, se separa una zona de tamaño igual a la suma de los campos que la componen.[6]

El manejo es un tanto más complicado. Los vectores globales —declarados en el .data— se pueden manejar con direccionamiento indexado; los vectores locales no se pueden manejar así porque la dirección inicial no es fija —puesto que se generan dinámicamente en la pila—. Ahora bien, se conoce su desplazamiento con respecto al `ebp`, lo cual permite trabajar con direccionamiento basado del estilo: `[ebp`

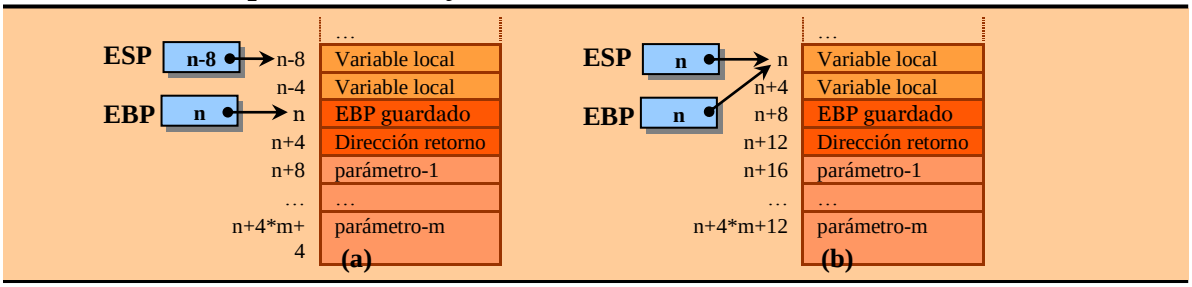


Fig. 7.4. Variables locales: (a) desplazamiento negativo, (b) desplazamiento positivo.

[6] Estrictamente hablando es un poco más complejo, porque, con frecuencia, los compiladores alinean las entidades en memoria, lo que puede causar que se necesite más memoria que el mínimo necesario.

`+desplazamientoV+k*esi]`, donde *desplazamientoV* es el desplazamiento del vector con respecto al `ebp`, `esi` contiene el índice del elemento deseado y *k* es el factor de escalamiento.

Las estructuras son un poco más simples: se conoce su desplazamiento con respecto a `ebp` y el desplazamiento de cada uno de sus campos con respecto al inicio de la estructura; en consecuencia, se usan direccionamientos del estilo `[ebp+desplazamientoE+desplazamientoC]`, donde *desplazamientoE* es el desplazamiento de la estructura con respecto al `ebp`, y *desplazamientoC* el desplazamiento del campo con respecto a la estructura.

7.5 MEMORIA DINÁMICA

El objetivo de la memoria dinámica es permitir al usuario determinar, en ejecución, el tamaño de las zonas de memoria que necesita. Se usa cuando no es posible determinar estáticamente los requerimientos de memoria; típicamente cuando se usan estructuras dinámicas, del tipo listas encadenadas, árboles, etc. Igualmente, es útil en otras ocasiones, por ejemplo, cuando se trabaja con vectores cuyo tamaño solo se determina en ejecución.

Uso de la memoria dinámica

Para manejar la memoria dinámica, se necesitan dos procedimientos: el primero permite pedir memoria y el segundo, liberarla; corresponden a las funciones *malloc* y *free* de C, o los operadores *new* y *delete* de C++. Por medio de estos procedimientos, es posible solicitar áreas de memoria de diferentes tamaños y devolverlas cuando ya no sean necesarias.

Cuando se pide memoria, de una manera u otra, se debe indicar el tamaño deseado; el manejador de memoria dinámica retorna un apuntador a una zona disponible del tamaño deseado —o un apuntador nulo si no es posible satisfacer el requerimiento—. Al liberar memoria, se debe indicar la dirección del área que se desea liberar —un apuntador—; el sistema toma la zona y la agrega a la memoria disponible.

La parte que concierne al usuario es sencilla: él pide una zona por medio de los procedimientos del manejador y la manipula por medio del apuntador que recibe. Cuando termina de usarla, la devuelve al sistema usando los procedimientos del manejador.

Los diversos lenguajes de alto nivel utilizan varios tipos de mecanismo para llamar estas funciones. Incluso, en algunos casos, como en Java, los ocultan en buena medida —el sistema mismo se encarga de pedir y liberar la memoria sin intervención del usuario, o con una intervención mínima—. Pero, en cualquier caso, sea por medio del usuario, del compilador o del sistemas de soporte en tiempo de ejecución, se acabará llamando los procedimientos de administración de la memoria dinámica.

Normalmente, los sistemas operativos ofrecen soporte para esta tarea; es decir, tienen procedimientos que se encargan de la administración de la memoria

dinámica. En el caso de Windows, se dispone, entre otras, de las funciones *GlobalAlloc* y *GlobalFree*.

GlobalAlloc[7] recibe dos parámetros: *uFlags* y *dwBytes*; el segundo es la cantidad de bytes que se desea reservar; el primero permite seleccionar algunas opciones sobre la zona de memoria, del estilo: si se desea iniciarla en ceros o no, si es movible o no —esto se explicará posteriormente—, etc. Aquí solo diremos que la opción `GMEM_FIXED` se limita a pedir memoria sin características especiales. Esta función se puede invocar así:

```
invoke GlobalAlloc, uFlags, dwBytes
```

La función retorna un apuntador a la zona o `NULL` si no hay suficiente memoria.

El procedimiento *GlobalFree* se encarga de liberar memoria. Su único parámetro es un apuntador a la zona que se está liberando.

Administración de la memoria dinámica

La parte del manejador es más delicada. En primer lugar, se debe reservar una región de memoria para entregar las zonas de memoria solicitados por el usuario; esta región se conoce como *el montón* (*heap*).

Pero hay más: el usuario no necesariamente devuelve la memoria en el mismo orden en que la pidió. Luego, el manejador de memoria dinámica debe hacer un seguimiento a cuáles zonas están libres y cuáles, ocupadas.

Una forma usual de lograrlo es por medio de una lista encadenada cuyos nodos son los pedazos libres de los que se dispone. Los nodos son de tamaño variable, ya que, después de un cierto número de pedidos y retornos de memoria, las zonas libres estarán distribuidas de una manera aleatoria. En consecuencia, el manejador debe incluir dos datos en cada nodo: el tamaño del nodo y un apuntador al siguiente.

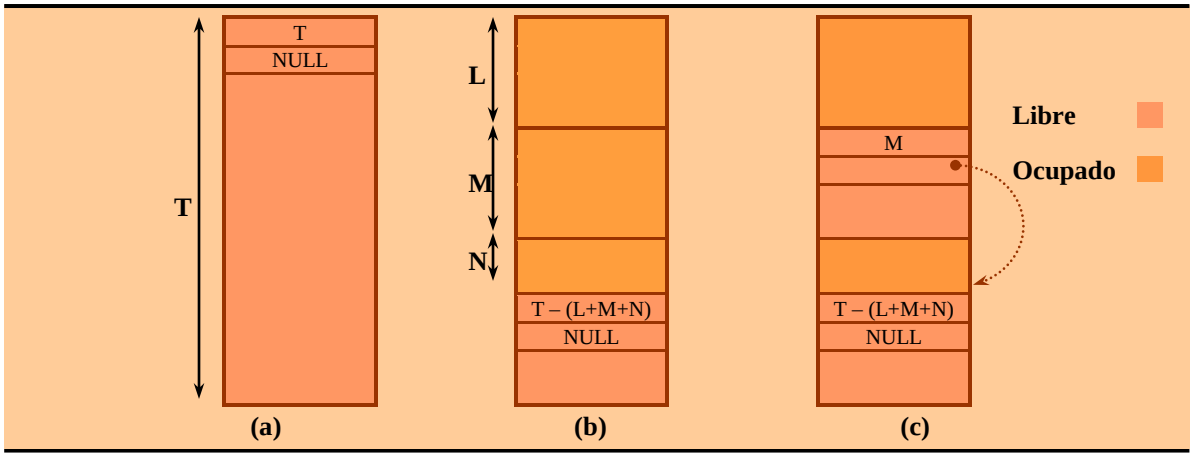


Fig. 7.5. Varios estados de la memoria dinámica.

[7] En el API de Windows hay otra función, *LocalAlloc*, que se mantiene por razones de compatibilidad pero que, actualmente, es equivalente.

Inicialmente, el manejador tiene un solo nodo que cubre toda la zona de memoria disponible —todo el *heap*—; el campo de longitud debe indicar el tamaño total de la zona —*T*— y el apuntador al siguiente debe estar en `NULL`, como se muestra en la figura 7.5(a). Después de haber solicitado zonas de tamaños *L*, *M* y *N*, debe quedar como se muestra en la figura 7.5(b). Cada vez que se hace una solicitud, se debe partir el nodo: una parte se le entrega al usuario y la otra se conserva como memoria disponible.

Si el usuario devuelve la zona de tamaño *M*, esta se debe adicionar a la lista como se muestra en la figura 7.5(c). En este estado, puede ocurrir que el usuario pida más memoria; en dicho caso, se recorre la lista buscado un bloque que pueda satisfacer el pedido. Se pueden usar diferentes criterios para seleccionar el bloque: el que mejor se ajuste, el más grande, el primero que se encuentre, etc.; en este ejemplo, usaremos el tercero: seleccionar el primer bloque en la lista con tamaño suficiente para satisfacer el pedido. En cualquier caso, el bloque se parte y el pedazo que sobra se encadena a la lista de memoria disponible.

También podría ocurrir que el usuario libere el pedazo de tamaño *L*. En tal caso, este pedazo debe unirse con el bloque siguiente (*M*) para formar un solo nodo de tamaño *L+M*, de lo contrario la memoria empezaría a fragmentarse; es decir, con el tiempo, se generaría una multitud de pequeños bloques inutilizables.

Si el bloque liberado es el de tamaño *N*, hay que fundirlo con el anterior y con el siguiente. En cada caso, hay que corregir los apuntadores y los tamaños como convenga.

La descripción general de la estructura es la siguiente:

- Se mantiene una lista encadenada de las zonas libres de memoria.
- La lista se ordena por direcciones de memoria —por la dirección donde empieza el nodo—; es decir, los apuntadores siempre van “hacia adelante”, nunca “hacia atrás”. Esto facilita la recuperación de los bloques liberados.
- Cada nodo se divide en tres partes: un campo que indica el tamaño del nodo, un campo con un apuntador al nodo siguiente y el bloque de memoria como tal. Al comienzo hay un solo nodo con toda la memoria disponible.

Cuando se solicita memoria se procede de la siguiente manera:

- El usuario invoca el procedimiento para pedir memoria, pasándole como parámetro el tamaño deseado.
- Se busca un nodo que satisfaga el pedido —con algún criterio; puede ser el primero encontrado—. Si no se encuentra, se suspende el proceso y se retorna un apuntador nulo al usuario.
- El nodo se parte en dos pedazos: uno es la memoria solicitada, que se entrega al usuario, y el otro, la memoria que sobró, se convierte en un nodo de la lista encadenada.
- Se retorna al usuario un apuntador a la zona que se encontró.

Cuando se retorna memoria se procede de la siguiente manera:

- El usuario invoca el procedimiento para liberar memoria, pasándole como parámetro un apuntador a la zona que está devolviendo.
- Se busca la posición en la lista en la que debe ir esta zona de memoria —recuerde: la lista está ordenada por direcciones de las zonas—.
- Se revisa si la zona de memoria es adyacente al nodo anterior o al siguiente; si es así, se funde con el nodo respectivo —los dos nodos se vuelven uno solo, con tamaño igual a la suma de los tamaños individuales—; puede ocurrir que se funda con los dos nodos vecinos.
- Si no se funde con los vecinos, simplemente se encadena a la lista.

En este último proceso hay un problema adicional: el usuario pasa un apuntador a la zona que está retornando, ¿cómo saber cuánto mide esa zona? Se suele proceder de la siguiente manera: cuando el usuario pide memoria, se separa un bloque con la siguiente estructura: un encabezado, seguido de la memoria que se está entregando. En el encabezado se guarda la información de cuánto mide la zona entregada. El apuntador que se devuelve al usuario apunta a la memoria en sí, así que el encabezado queda inmediatamente antes. Cuando el usuario retorna una zona, el procedimiento de liberación de memoria puede revisar la información del encabezado para saber cuánto mide.

Fragmentación y manijas

Como se mencionó anteriormente, después de múltiples pedidos y retornos de memoria, se puede generar una multitud de pequeños bloques que, sumados, puede ser una cantidad considerable de memoria, pero que no se puede aprovechar porque está fragmentada.

Se podría pensar en compactar la memoria, es decir, desplazar en la memoria los bloques ocupados, agrupándolos, para así eliminar los huecos entre ellos. Sin embargo, esto no es posible, puesto que el usuario tiene apuntadores a estas zonas y quedarían desactualizados —ver figs. 7.6 (a) y (b)—.

Una técnica para manejar este problema se basa en el uso de la doble indirección:

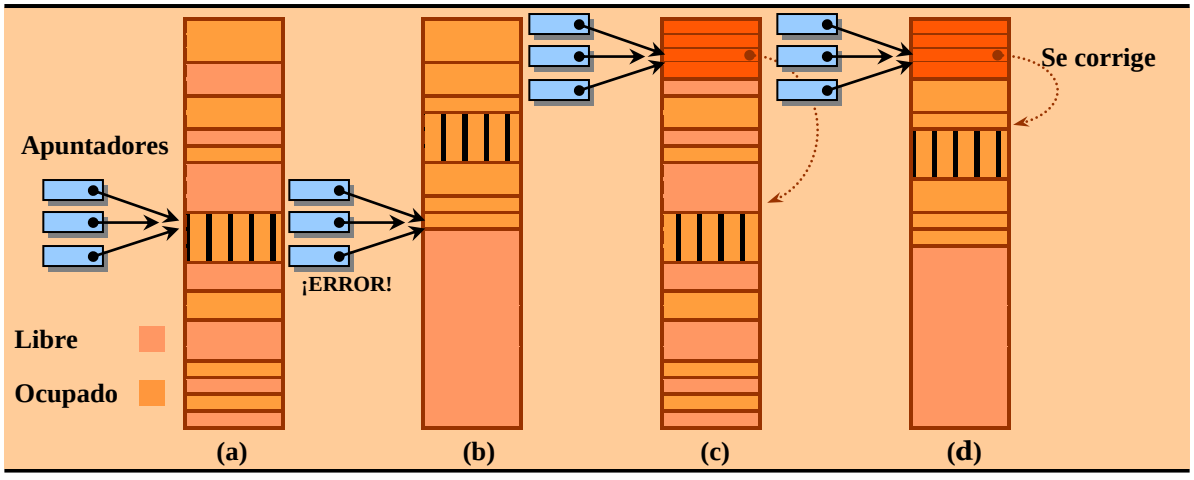


Fig. 7.6. Compactación: (a) y (b) sin manijas, (c) y (d) con manijas.

un apuntador a un apuntador a la zona de memoria, lo cual se conoce como *manija*. La idea es la siguiente: la región de la memoria dinámica se divide en dos zonas; una de las zonas se destina a la memoria dinámica en sí; la otra, es un vector de apuntadores —inicialmente todos en `NULL`—.

Cuando un usuario pide memoria, el manejo es igual al antes descrito; pero no se le entrega un apuntador a la memoria. En su lugar, se busca una posición libre del vector, se guarda en ella el apuntador a la memoria, y se retorna un apuntador a esa posición del vector.

Se supone que el usuario utiliza la doble indirección para acceder a la memoria; que no usa el apuntador directo. Así, toda copia del apuntador que haga el usuario apuntará al vector y no a la memoria en sí. De esta manera sí es posible compactar: se pueden mover los bloques de memoria y se actualizan los apuntadores en el vector —ver figs. 7.6 (c) y (d)—.

Ahora bien, el uso de la doble indirección puede introducir bastante ineficiencia. Por esta razón, se ideó un mecanismo que le permite al usuario utilizar, temporalmente, el apuntador directo. Para esto, se tienen dos procedimientos que permiten asegurar —*lock*— y desasegurar —*unlock*— la memoria. El procedimiento de asegurar marca la zona como no-movible; esto quiere decir que, si se hace una compactación, esa zona se debe dejar donde está. El procedimiento de desasegurar le quita la marca a la zona. Mientras la zona está asegurada, el usuario puede utilizar el apuntador directo. Se supone que el usuario asegura la zona, usa temporalmente el apuntador directo y desasegura la zona.

Es de anotar que con la memoria virtual —que estudiaremos posteriormente— este mecanismo pierde interés puesto que la memoria virtual se encarga, hasta un cierto punto, del control de la fragmentación.

7.6 SEGURIDAD Y LA MEMORIA

Tanto la pila como la memoria dinámica son susceptibles a errores y a manipulación malintencionada; constituyen, en efecto, uno de los puntos más frágiles de los computadores modernos, tanto del punto de vista de la seguridad como del de la programación.

Para estudiar estos problemas es conveniente separar dos factores: una cosa son las estructuras de datos que sirven para administrar las zonas de memoria, y otra es la memoria física como tal. Por ejemplo:

- Cuando liberamos memoria en la pila, movemos el `esp`, y decimos que la memoria por encima de él está “libre”. Así es conceptualmente, pero, físicamente esa memoria “libre” sigue manteniendo la misma información que antes puesto que nadie la ha borrado.
- Cuando liberamos una zona de memoria dinámica, también decimos que está “libre”, pero, de nuevo, físicamente sigue manteniendo la misma información de antes.

- Si tenemos un vector de 20 posiciones, y asignamos un valor en la posición 21, desde el punto de vista lógico —del lenguaje de programación—, las posiciones de la 21 en adelante no existen, pero, desde el punto de vista físico, eso generará una escritura en memoria —en la zona inmediatamente adyacente al vector—.

La pila está compuesta, en principio, por ambientes de procedimientos que son disyuntos y teóricamente inaccesibles el uno del otro, pero, en la práctica, están en la memoria y son adyacentes. Lo mismo es aplicable al *heap*: se trata de zonas de memoria entregadas independientemente, pero en realidad son vecinas en la memoria. Cuando se declaran dos variables, en teoría son independientes y no tienen nada que ver la una con la otra, pero, de nuevo, pueden ser vecinas en la memoria y accesible la una desde la otra.

Estos puntos débiles, junto con errores de los programadores, son aprovechados para penetrar la seguridad de los sistemas.

A continuación describiremos los principales problemas que se presentan en estos dos tipos de memoria.

La memoria dinámica

La memoria dinámica presenta dos características que explican en buena medida su fragilidad:

- Para su correcto funcionamiento depende mucho del manejo que de ella haga el usuario; es decir, en buena medida, es controlada por el usuario y no por el sistema.[8]
- Es una "zona de tránsito". Es decir, la memoria dinámica se asigna para diversas tareas y no tiene orden; un mismo pedazo de memoria puede servir para diversas funciones a lo largo de la ejecución del programa, y puede ser asignado, reasignado, dividido y fusionado múltiples veces.

Hay errores típicos en la memoria dinámica que son la causa de los accidentes y vulnerabilidades. A continuación presentaremos los principales.

Falta de inicialización

Esto es aplicable a las variables en general, a los apuntadores en particular y especialmente a los apuntadores a memoria dinámica.

Un error frecuente consiste en olvidar darle un valor inicial a los apuntadores. Cuando se declara un apuntador, solo se está separando el espacio en memoria para el mismo, pero el apuntador todavía no apunta a nada:

```
int * p = "Hola";
int * q;
```

[8] Estamos hablando en términos generales. Java, por ejemplo, le quita esta responsabilidad al programador, y lo hace, justamente, porque considera "peligroso" que este se encargue de esas tareas. Java va más allá e intenta controlar el manejo de apuntadores en general. Los apuntadores son, en algún grado, el "GO TO" de las estructuras de datos.

En estas declaraciones, tanto `p` como `q` existen, así como `*p`; pero `*q` no existe.

En el caso de los apuntadores a memoria dinámica, es necesario invocar la función que separa espacio en el *heap*, y asignarle al apuntador el valor devuelto por dicha función (`malloc` en el caso de C).

Incluso, aunque se llame la función, puede haber errores, puesto que la memoria se puede agotar, en cuyo caso el valor retornado por la función no corresponderá a una zona válida de memoria.

Pérdida de memoria

Esto ocurre cuando el programador pide memoria dinámica y olvida devolverla. Aunque la memoria no está siendo físicamente usada, las estructuras de datos de la memoria dinámica la tendrán marcada como ocupada. Tendremos memoria libre que no está siendo utilizada, con el consiguiente desperdicio.

Uso de memoria liberada

De cierta manera este también es un error de inicialización: el programador pide memoria, la retorna, y sigue utilizando el apuntador sin volver a pedir memoria.

Se puede decir que el apuntador no está inicializado, pero, en realidad, es un error más insidioso. Esto porque el contenido del apuntador es una dirección válida en el *heap*, en consecuencia, podremos seguir escribiendo allí. Si la memoria sigue libre, el programa continuará funcionando normalmente, pero si la memoria es reasignada —porque se solicitó más memoria dinámica—, tendremos dos apuntadores apuntando a la misma zona de memoria y creyendo que se trata de dos zonas diferentes. En la figura 7.7 (b), `p` sigue apuntando a una zona liberada, y en la (c), se hizo un `malloc` para `q`; `p` y `q` apuntan a la misma zona, pero el programador cree que son zonas diferentes.

Replicas de apuntadores

Una variante del error anterior ocurre cuando se tiene varios apuntadores a la misma zona, y por medio de uno de ellos se libera la zona, y luego se vuelve a pedir memoria, pero se olvida actualizar a los otros apuntadores; el programador cree que todos apuntan a la misma zona cuando en realidad apuntan a dos zonas distintas: una válida y la otra inválida.

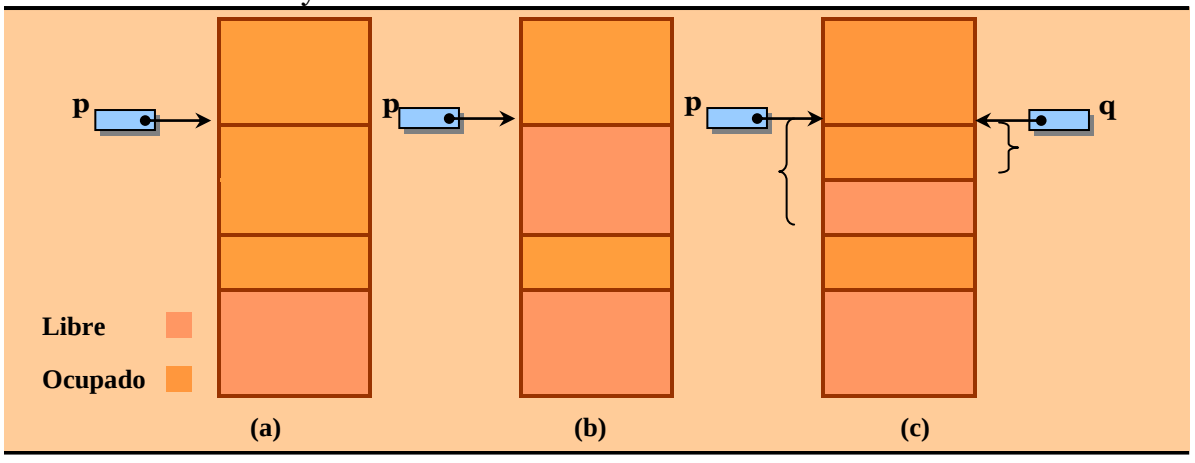


Fig. 7.7. (a) Memoria separada (b) Liberada pero en uso. (c) Reutilizada.

Múltiples liberaciones

Este error se presenta cuando el programador libera múltiples veces la misma zona de memoria —es decir, hace dos `free` del mismo apuntador sin que medie un `malloc`—.

En principio parecería que no es un error: ¿qué mal puede hacer devolver algo que ya se devolvió? Hay dos circunstancias:

- Si, después de que se liberó la memoria por primera vez, se hace un `malloc`, la zona puede ser reasignada para otra cosa; en consecuencia, cuando se haga el segundo `free` se retornará una zona que puede estar siendo utilizada para otras labores.
- Aunque la memoria siga libre —no se ha hecho ningún `malloc`—, el hecho de liberar dos veces la misma memoria puede hacer que funcionen mal las estructuras de datos de la memoria dinámica, con lo cual pueden quedar en un estado inconsistente.

La pila

La pila tiene tres problemas básicos:

- Lo que se destruye no desaparece; la información de los procedimientos que ya terminaron, puede permanecer un tiempo en la pila.
- Los ambientes que se crean en la pila, aunque teóricamente inaccesibles el uno desde el otro, en la práctica, se pueden alcanzar.
- La información que controla el funcionamiento de la pila, está en la pila misma, y, por ende, puede ser modificada por lo programas usuarios.

Acceso a ambientes desaparecidos

Cuando un procedimiento termina, retorna su espacio en la pila; esto consiste en desplazar el `esp`, pero no se borra la información. Por ejemplo:

```
1:   void f () {
2:     char s[100];
3:      ...
4:   }
5:   void g (int i) {
6:     char * c;
7:     f();
8:     c = (char *) &i;
9:     c -= 108;
10:   printf ("%s\n", c);
11:  }
```

En la línea 7 se invoca `f`, y esta retorna en 4; en ese punto se supone que la variable `s` deja de existir. Después de la invocación, en la línea 8, `c` queda apuntando a `i`. Ahora bien, `i` es un parámetro, luego esta en la pila; en consecuencia, `c` apunta a la

pila. En la instrucción 9 se retrocede el apuntador `c`;[9] esto no tiene sentido desde el punto de vista del lenguaje, pero, desde el punto de vista físico, `c` retrocede en la pila y queda apuntando a `s`. En consecuencia, la instrucción 10 imprime la cadena `s`. La figura 7.8 ilustra el proceso.

Acceso a otros ambientes

En el programa de la sección anterior, si en lugar de restarle al apuntador `c` le sumamos una cantidad, podremos localizarlo en el ambiente del procedimiento llamador. Una vez localizado allí, es posible manipular las variables locales del procedimiento llamador.

Manipulación de la estructura de la pila

En las secciones anteriores localizamos el apuntador `c` para modificar o usar variables locales de otros procedimientos, sean anteriores, sean los que ya ejecutaron. De la misma manera es posible usar el apuntador para modificar otras informaciones que se encuentran en la pila.

En concreto, es posible cambiar las direcciones de retorno o los `ebp` almacenados en la pila, con lo cual se puede modificar el flujo de control: retornar a una función que no es la llamadora, o incluso poner una dirección de un pedazo de código arbitrario. En efecto, si se altera la dirección de retorno, cuando se ejecute la instrucción `ret` del procedimiento que está ejecutando en ese momento, el procedimiento saltará a ejecutar al punto indicado, sin importar que no sea el punto de la invocación original.

Existen otras técnicas para lograr este efecto; la más conocida se llama *buffer overflow*, y es una falla de seguridad ampliamente reconocida. A continuación la explicaremos someramente.

Estudiemos el comportamiento de la siguiente función:

```
void f (char * s ){
  char t[100];
  int i = 0;

  while ( t[i++] = *s++ );
}
```

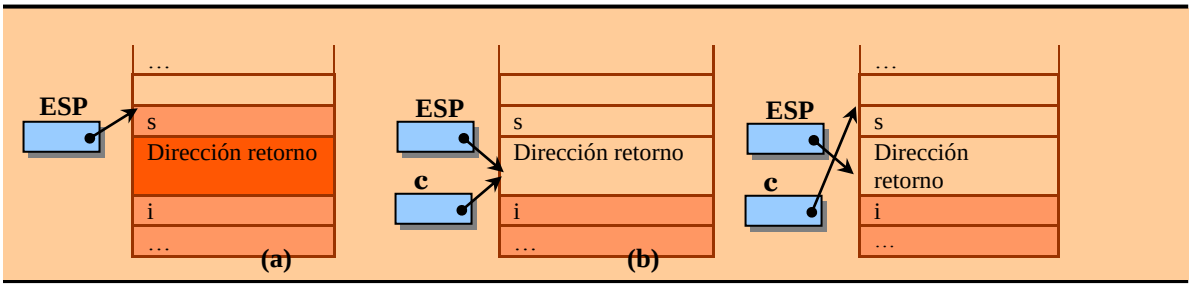


Fig. 7.8. (a) Durante el call de f, (b) Después del retorno, (c) Uso del ambiente de f.

[9] El valor de la constante —108 en este caso— es solo un ejemplo, puesto que en la práctica depende de varios factores; principalmente de los parámetros y variables locales de `f` y `g`.

La función `f` recibe como parámetro un apuntador a una cadena, la cual copia sobre una variable local —un vector de caracteres de 100 posiciones—.

Ahora, ¿qué ocurre si la cadena apuntada por `s` tiene más de 100 caracteres? Puesto que `f` no revisa la longitud de la cadena, procederá a copiarla aunque no quepa en el espacio reservado en `t`. Esto quiere decir que copiara por fuera del espacio asignado a `t`, y alterará variables vecinas. Pero `t`, al ser una variable local, está localizada en la pila, lo cual quiere decir que algunos caracteres de `s` se copiarán sobre entidades en la pila; en particular, si la cadena apuntada por `s` es lo suficientemente larga, se copiarán sobre la dirección de retorno.

Si se logra identificar cuáles caracteres de `s` se copiarán sobre la dirección de retorno, se podrán asignar estos caracteres para que conformen la dirección del punto a donde deseamos saltar. Dicho de otra manera, la dirección que nos interesa ejecutar se asignará a la cadena en la posición de los bytes que se copian sobre la dirección de retorno. Luego se invocará `f` pasándole como parámetro la cadena así traficada.

EJERCICIOS

1- **a-** Desarrolle una función que compare dos cadenas (usando el orden lexicográfico; las mayúsculas son menores que las minúsculas, como en el código ASCII).

El procedimiento recibe en `esi` y `edi` apuntadores a sendas cadenas. Debe poner `eax` en 0 si son iguales; en -1, si la primera es menor que la segunda y en 1, si es mayor.

b- Se tiene un vector de apuntadores a cadenas. Escriba un procedimiento que revise si una cadena determinada se encuentra en él.

El procedimiento recibe en `esi` un apuntador a la cadena en cuestión, en `edi` un apuntador al vector de cadenas y en `ecx` el número de elementos del vector. En `ebx` retorna el índice de la posición donde se encuentra la cadena, ó -1 si no está.

c- Para el mismo vector de apuntadores a cadenas, escriba un procedimiento para abrir un espacio en el vector. El procedimiento recibe un índice del vector y desplaza todos los apuntadores una posición de ese punto en adelante.

El procedimiento recibe en `edi` un apuntador al vector de cadenas y en `ebx` el índice.

d- Se tiene un vector de apuntadores a cadenas. Escriba un procedimiento para adicionar una cadena al vector. El vector está en orden lexicográfico, por lo tanto, al adicionar la nueva cadena, hay que buscar la posición que le corresponde dentro del vector.

El procedimiento recibe como parámetro, en `esi`, un apuntador a la cadena que se quiere adicionar; en edi, un apuntador al vector y en ecx el tamaño actual del vector (hasta donde está ocupado). Suponga que el vector tiene espacio disponible para adicionar la cadena.

e- Modifique los procedimientos anteriores para que reciban los parámetros por la pila.

2- Se quiere desarrollar procedimientos para manejar conjuntos. Los conjuntos se representan por medio de apuntadores a listas encadenadas. El conjunto vacío se representa por la lista vacía —apuntador en `NULL`—. Cada nodo debe tener dos campos: número y siguiente.

a- Escriba la función *pertenece*, que recibe en `esi` un apuntador a un conjunto y en `eax` un entero. La función retorna `eax` en 1 si el entero pertenece al conjunto; si no, retorna cero.

b- Desarrolle un procedimiento *adicionar* con dos parámetros: un apuntador a un conjunto en `esi` y un apuntador a un nodo en `edi`. El procedimiento debe adicionar el nodo al conjunto si no se encuentra en el mismo.

c- Escriba un procedimiento para realizar la unión de dos conjuntos (*unir*). Los apuntadores a los conjuntos son pasados como parámetros en `esi` y `edi`. `eax` debe quedar apuntando al conjunto resultado; retire los elementos de los conjuntos apuntados por `esi` y `edi` y páselos al conjunto apuntado por `ebx`.

d- Desarrolle un procedimiento que realice la intersección de dos conjuntos (*intersecar*). Los parámetros y las restricciones son los mismos del punto anterior.

e- Escriba un procedimiento *diferencia* que recibe en `esi` y `edi` dos conjuntos y retorna su diferencia en `eax`.

f- Modifique los procedimientos anteriores para que reciban los parámetros por la pila. Los resultados se retornan en `eax`.

3- Vamos a utilizar una estructura que llamamos *texto*. Un texto es una secuencia de caracteres que termina con el carácter ETX —ASCII 3—. Supondremos que los textos tienen un tamaño máximo `MAX`. Las palabras y líneas miden máximo 80 caracteres.

a- Escriba un procedimiento que informe cuál es la siguiente palabra en una estructura de tipo texto. Se considera una palabra cualquier secuencia de caracteres alfabéticos en mayúsculas o minúsculas.

El procedimiento recibe en `ebx` un apuntador a un texto, en `esi` un apuntador a la posición del texto a partir de la cual se quiere buscar la palabra —`esi` no necesariamente apunta al principio del texto— y en `edi`, un apuntador a una variable de tipo cadena de caracteres donde se debe dejar la palabra encontrada. Debe retornar la palabra como una cadena de caracteres, no como un texto.

b- En una estructura de tipo texto, se considera como línea cualquier secuencia de caracteres que acaba en CR (*Carriage Return*, ASCII 0DH).

Escriba un procedimiento que extraiga la siguiente línea de una estructura de tipo texto.

El procedimiento recibe en `ebx` un apuntador a un texto, en `esi` un apuntador a la posición del texto a partir de la cual se quiere extraer la línea y en `edi`, un apuntador a una cadena donde se debe dejar la línea leída.

c- Modifique los procedimientos anteriores para que reciban los parámetros por la pila.

4- Vamos a trabajar con programas usualmente llamados *filtros*; son programas que leen un texto —ver punto anterior— y lo transforman de alguna manera. Por ejemplo, un programa que convierte todos los caracteres de un texto a mayúsculas es un filtro.

a- Desarrolle un procedimiento que recibe como parámetros un vector de cadenas y un texto. El procedimiento debe recorrer el texto por palabras, mirando si cada palabra está en el vector o no; si está, la pasa a mayúsculas en el texto de salida.

Los parámetros son: `esi` es un apuntador al texto original; `edi` apunta al texto resultante; `ebx` apunta al vector de cadenas y `ecx` tiene el número de elementos del vector.

b- En un texto, el carácter HT —*Horizontal Tabulation*, ASCII 9— se utiliza para poner los caracteres que lo siguen en una posición determinada dentro de la línea. En este caso, vamos a suponer que el carácter HT indica que los caracteres que siguen se deben poner en la siguiente posición divisible por 8.

Escriba un procedimiento que cambie los HT por blancos de manera que los caracteres queden en las posiciones indicadas por los HT.

En `esi` recibe un apuntador al texto original; en `edi` recibe un apuntador a un texto donde se deja el resultado.

c- Escriba un procedimiento que justifique un texto. Es decir, que distribuya las palabras de cada línea de manera que queden alineadas a la izquierda y a la derecha —excepto si la línea acaba en punto, en cuyo caso, la deja igual—. Recuerde que las líneas del texto son de 80 caracteres.

En `esi` recibe un apuntador al texto original y en `edi` un apuntador a un texto donde se deja el resultado.

d- Modifique los procedimientos anteriores para que reciban los parámetros por la pila.

5- Vectores de bits.

a- Escriba un procedimiento que informa si el *i*-ésimo bit de un vector de palabras vale 1 ó 0. Numere los bits de "izquierda a derecha". El

procedimiento recibe un apuntador al vector en `esi` y en `edi`, la posición del bit que se quiere obtener; el resultado se retorna en `eax`.

b- Escriba un programa que modifica el *i*-ésimo bit de un vector de palabras sin modificar los otros. El procedimiento recibe un apuntador al vector en `esi` y en `edi`, la posición del bit que se quiere modificar; el valor del bit se envía en `eax`.

c- Modifique los procedimientos anteriores para que reciban los parámetros por la pila.

6- Una forma de representar conjuntos de tamaño limitado es por medio de vectores de bits. Se tienen vectores de palabras de un cierto tamaño *N*, si el *i*-ésimo bit del vector vale 1, el número *i* pertenece al conjunto.

Resuelva el ejercicio 7.2 usando esta representación.

7- Un vector disperso es aquel que tiene muchos elementos en cero, lo cual desperdicia memoria. Para evitar esto, se desarrolla una representación especial para este tipo de vector.

La representación es la siguiente: se tiene un arreglo de bits —vector de control— con tantos elementos como el vector disperso; el valor de cada bit indica si el elemento correspondiente del vector disperso vale 0 —bit de control en cero— o no —bit de control en 1—. Por otro lado, se tiene el vector de contenido que es un arreglo de palabras con los valores de los elementos diferentes de cero del vector disperso. Por ejemplo, el vector disperso:

0	0	0	3	0	0	7	0	0	-1	0	0	0	0	4	1

Se representa por el vector de control:

0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1

Y el vector de contenido:

3	7	-1	4	1

Desarrolle un procedimiento que sume dos vectores dispersos —usando la representación anterior—, cuyo resultado es otro vector en la misma representación. Decida cuántos parámetros necesita y cómo los pasa.

Atención: al sumar 2 elementos diferentes de 0, el resultado puede ser 0.

8- Vamos a trabajar con matrices cuadradas de dobles palabras.

11	6	0	0
10	65	15	0
2	0	5	17
42	7	23	21

11	0	0	0
10	65	0	0
2	0	5	0
42	7	23	1

a- Una matriz inferior diagonal-n es aquella que tiene *n* diagonales en cero, contando las diagonales desde la esquina superior derecha. Por ejemplo, las dos matrices anteriores son diagonal-2 y diagonal -3, respectivamente:

Escriba un procedimiento que detecte si una matriz es inferior diagonal-*n*. El procedimiento recibe como parámetros un apuntador a la matriz, el valor de *n* y la dimensión de la matriz —recuerde que es cuadrada—. El resultado se retorna en `eax`: 1, sí es, cero, no es.

b- Una matriz simétrica-*n* es aquella cuyas *n* diagonales superiores son iguales a las n diagonales inferiores. Ejemplo:

11	**6**	**2**	**42**
0	**65**	**15**	**7**
2	**51**	**5**	**17**
42	**7**	**6**	**2**

Desarrolle un procedimiento que reconozca si una matriz es simétrica-*n*. Utilice las mismas convenciones del punto anterior.

c- Una matriz banda-*m* es aquella cuyas *m* diagonales centrales son las únicas con valores diferentes de cero. Ejemplo:

11	**6**	**0**	**0**
0	**65**	**15**	**0**
0	**51**	**5**	**17**
0	**0**	**6**	**2**

Escriba un procedimiento que determine si una matriz es banda-*m*. Las convenciones son las mismas de los puntos anteriores.

En cada ejercicio, decida cuántos parámetros necesita y cómo los pasa

9- En ocasiones se quiere trabajar con subvectores de un vector o de una matriz. Esto es muy útil en ciertos casos; por ejemplo, si se tiene una matriz almacenada por filas y se quiere trabajar con columnas o diagonales, hay que hacer algoritmos específicos para cada caso. En lugar de esto, se puede crear una representación general para subvectores.

La representación es la siguiente: se tienen dos arreglos; uno de ellos es el vector original con todos los elementos disponibles —vector de contenido—; el otro es un vector de bits —vector de control— con la misma cantidad de elementos. El valor de cada bit del vector de control indica si el elemento correspondiente del vector de contenido hace parte o no del subvector; el bit en 1 indica que sí hace parte. Por ejemplo, si se tiene el vector:

9	**13**	**0**	**3**	**6**	**57**	**7**	**3**	**2**	**-1**	**-8**	**27**	**5**	**5**	**4**	**1**

Y el vector de control:

0	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**

El subvector representado es:

3	**7**	**-1**	**4**	**1**

Escriba un procedimiento que sume dos subvectores, usando la representación anterior; el resultado debe quedar en otro subvector en la misma representación.

Los vectores se representan con estructuras que tienen tres campos: un apuntador al vector de contenidos, un apuntador al vector de control y el tamaño del subvector. Los parámetros serán apuntadores a estructuras de este tipo. Decida cuántos parámetros necesita y cómo los pasa.

10- Otra representación de subvectores se logra con la estructura siguiente:

```
Subvector struct
   nElementos   dword  ?
   base         dword  ?
   factor       dword  ?
   apContenido  dword  ?
Subvector ends
```

nElementos es el número de elementos del subvector. *apContenido* es un apuntador al vector original. *base* y *factor* sirven para obtener la posición de cada elemento del subvector de la siguiente manera:

```
Subvector [i] = (*apContenido)[Base + Factor*i]
```

Por ejemplo, si se tiene una matriz de tamaño *n* representada por filas, la diagonal se puede representar por: *base* = 0 y *factor* = *n*+1; la segunda columna se representa por: *base* = 1 y *factor* = *n*; la primera fila se representa por: *base* = 0 y *factor* = 1.

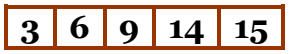
En cada punto, decida cómo pasa los parámetros.

a- Escriba un procedimiento que sume dos subvectores, usando esta representación; el resultado queda en otro subvector con la misma representación.

b- Escriba un procedimiento que calcula el producto punto de dos vectores representados con las convenciones anteriores.

11- Otra representación de subvectores consiste en lo siguiente: se tienen dos arreglos; uno de ellos es el vector original con todos los elementos —vector de contenido—; el otro es un vector de índices con tantos elementos como el subvector. El valor de cada elemento del vector de índices denota un elemento del vector de contenido. Por ejemplo, si se tiene el vector:

Y el vector de índices:



El subvector representado es:

Los índices no necesariamente están en orden ascendente, e incluso pueden estar repetidos.

Los vectores se representan con estructuras que tienen tres campos: un apuntador al vector de contenidos, un apuntador al vector de índices y el tamaño del subvector. Los parámetros serán apuntadores a estas estructuras.

En cada punto, decida cómo pasa los parámetros.

a- Escriba un procedimiento que sume dos subvectores, usando esta representación; el resultado queda en otro subvector con la misma representación.

b- Escriba un procedimiento que implemente la operación *recolectar*. Esta operación toma un subvector y un vector resultado normal, y arma el subvector en el vector resultado como un vector regular. Por ejemplo, si se tiene el vector:

9	13	0	3	6	57	7	3	2	-1	-8	27	5	5	4	1

Y el vector de índices:

3	6	9	14	15

El vector regular será:

3	7	-1	4	1

c- Escriba un procedimiento en ensamblador que implemente la operación *esparcir*. Esta operación toma un subvector y un vector normal, y pasa los elementos del vector normal al subvector. Por ejemplo, si se tiene el vector:

9	13	0	8	6	57	23	3	2	0	-8	27	5	5	76	54

Y el vector de índices:

3	6	9	14	15

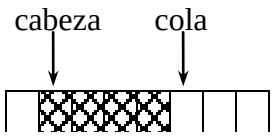
Y el vector regular:

3	7	-1	4	1

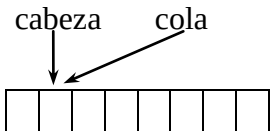
El vector se transformará en:

9	13	0	3	6	57	7	3	2	-1	-8	27	5	5	4	1

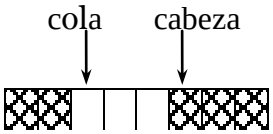
12- Cola circular. Esta es una implantación particular de las colas. Se utiliza un vector y dos apuntadores: un apuntador a la cabeza y un apuntador a la cola. El apuntador de cabeza apunta al primer elemento de la cola; el apuntador de cola, a la primera posición libre en el vector después del final de la cola:



Si la cola está vacía, los dos apuntadores están en la misma posición:



La particularidad de está implantación es que si uno de los apuntadores, de cabeza o de cola, llega al final del vector, la siguiente vez que toque incrementarlo pasará al principio del vector; de cierta manera, "da la vuelta"; como si el final del vector estuviera conectado con el principio:



En la gráfica anterior, después de retirar 3 elementos, la cabeza apuntará al principio del vector. Note que, en el ejemplo anterior, no se pueden adicionar más de 2 elementos, ya que la cola quedaría igual a la cabeza, y se interpretaría como que la cola está vacía.

Escriba procedimientos para adicionar y para retirar un elemento de la cola. Decida cuántos parámetros necesita y cómo los pasa.

13- Traduzca los siguientes segmentos de código en C a ensamblador. Escriba el prólogo del procedimiento (incluyendo la separación de espacio para las variables locales) y el epílogo (incluyendo retornar espacio) en los puntos apropiados. No puede usar los nombres simbólicos; solo los desplazamientos en la pila.

Código en C

`int main(int argc, char* argv[]) {`
`  int x, z, w, * y;`
`  ...`
`  f (5, & x, y, z + w);`

`int f(int a, int * b, int * c, int d){`
`  int e, h[10];`
`  char * i;`
`  double * j;`

`//en el contexto anterior (el de f)` `  a = d;`
`//en el contexto de f` `  *b = e;`
`//en el contexto de f` `  h[e] = d;`
`//en el contexto de f` `  e = *i;`

`//en el contexto de f`
`  return a + e;`
`}`

14- Traduzca la siguiente función de C a ensamblador (suponga que `f` ya fue declarada).

```
int g ( int c, char * p ){
  int x, z, *y;
  double j, * d;
  char v[4];

  x = z;
  c = *p;
  y = (int *)p;
  v[x] = z;
  *(y+4) = z;
  f (5, & x, y, z + c);
  return * y;
}
```

15- Traduzca los siguientes procedimientos de C a ensamblador.

```
void f (int n, char ** p){
  if ( n > 9 )
    f ( n / 10, p );
  **p = n % 10 + '0';
  (*p)++;
}

void g ( int n ){
  char c[100];
  char * q;

  q = c;
  f ( n, &q );
  *q = 0;
}
```

16- Traduzca el siguiente procedimiento de C a ensamblador.

```
int  f ( int n, int * p ){
  int v1, v2;

  if ( n == 1 )  return *p;
  v1 = f ( n/2, p );
  v2 = f ( n - n/2, p + n/2 );
  if ( v1 > v2 )
    return v1;
  else
    return v2;
}
```

17- Traduzca el siguiente procedimiento de C a ensamblador.

```
int diasenMes (int year, int mes) {
  int nDias;

  switch (mes) {
    case 4:
    case 6:
```

```
    case 9:
    case 11:
      nDias = 30;
      break;
    case 2:
      if (!(year % 4) && (year % 100)) || !(year % 400)
        nDias = 29;
      else
        nDias = 28;
      break;
    default: nDias = 31;
  }
  return nDias;
}
```

18- Traduzca el siguiente procedimiento de C a ensamblador.

```
int  f ( int n, int * p ){
  int v[8], i;
  int k = (n <= 8? n/2 : 8);

  if ( n == 1)  return *p;
  for ( i = 0; i < n; i++)
    v[i % k] += *p++;
  return f ( k, &v );
}
```

19- Se tiene la siguiente función en C:

```
int *  f (){
   int  i = 3;
   return & i;
}
```

¿Que opina de esta función? Piense en lo que ocurre en la pila y las consecuencias que puede traer más adelante en el programa.

20- Una manera de pasar un vector como parámetro por valor es hacer una copia del vector en la pila. El procedimiento usa la copia en la pila y no modifica el vector original.

Escriba el código necesario para copiar un vector en la pila. Cuidado con el orden de los elementos.