Módulo 09
Lenguaje Ensamblador

Organización de Computadoras
Depto. Cs. e Ing. de la Comp.
Universidad Nacional del Sur

Copyright

Contenidos

- Entorno de programación.
- Partes de un programa.
- Directivas al compilador.
- Arquitectura **i386**.
- Estructuras de control.
- Principales llamadas al sistema.
- Pasaje de parámetros.
- Proceso de ensamblado, vinculación y carga.

Entorno de programación

- En lo que resta de la materia, usaremos al sistema operativo **GNU/Linux** como entorno de programación en lenguaje ensamblador.
- ¿Qué es un sistema operativo?
  - Un sistema operativo es el encargado de gestionar la interacción entre el usuario y la computadora y de administrar sus recursos de manera eficiente.
- ¿Qué es **GNU/Linux**?
  - Es un **SO** libre: estamos autorizado para usarlo, copiarlo, estudiar su código fuente, etc.

Entorno de programación

- El entorno para programar en lenguaje ensamblador se entrega en un archivo que contiene una máquina virtual (VM).
- Se recomienda trabajar en un equipo con adecuada cantidad de memoria RAM.
- Formatos de VM:
  - OVA: Open Virtual Appliance. Brinda compatibilidad con diferentes aplicaciones para virtualización.
  - VDI: Formato de imagen de disco virtual de VirtualBox.

Inicio de la sesión

- El sistema operativo cuenta con dos usuarios previamente creados:
  - El administrador del sistema (**root**).
  - Un usuario convencional (**alumno**).
- La contraseña para el usuario convencional es: **ocuns**
- Para acceder a las tareas de administración del sistema también se usa la contraseña del usuario convencional (al estilo **Ubuntu**).

Arquitecturas de 32 bits

- Escribir el código fuente:

```
$ nano prueba.asm
```

- Compilar el código fuente:

```
$ yasm -f elf prueba.asm
```

- Enlazar (vincular) el código objeto:

```
$ ld -o prueba prueba.o
```

- Ejecutar el programa resultante:

```
$ ./prueba
```

Arquitecturas de 64 bits

- La distribución que utilizaremos se denomina: Devuan Jessie de **32** bits.
- El lenguaje ensamblador con el cual trabajaremos es para una arquitectura tipo **i386** de **32** bits.
- Por otra parte, quien tenga hardware de **64** bits y a su vez tenga **GNU/Linux** de **64** bits también puede cross-compilar código de **32** bits.

Arquitecturas de 64 bits

- Escribir el código fuente:

```
$ nano prueba.asm
```

- Compilar el código fuente:

```
$ yasm -f elf -m x86 prueba.asm
```

- Enlazar (vincular) el código objeto:

```
$ ld -o prueba -m elf_i386 prueba.o
```

- Ejecutar el programa resultante:

```
$ ./prueba
```

Depuración de un ejecutable

- Compilar el código fuente:

```
$ yasm -f elf -g dwarf2 prueba.asm
```

- Enlazar (vincular) el código objeto:

```
$ ld -o prueba prueba.o
```

- Depurar el programa resultante:

```
$ ddd prueba
```

Estructura de un fuente

- El código fuente, como suele ser el caso en los lenguajes de programación, son en esencia archivos de texto.
- Deben contener las siguientes secciones:
  - **`.data:`** conteniendo datos inicializados.
  - **`.bss:`** conteniendo datos no inicializados.
  - **`.text:`** conteniendo el programa propiamente dicho.
  - **`.stack:`** conteniendo información acerca de la configuración inicial de la pila del programa.

Directivas al compilador

- Las directivas instruyen al compilador, sin generar código máquina.
- Las directivas **%define** e **%include** funcionan de manera análoga a **C**:

```
%include “mylib.inc”

%define tope 100
```

- La directiva **EQU** asocia el resultado de una expresión a una etiqueta:

```
LF EQU 10
```

Directivas al compilador

- Existen dos clases de directivas para reservar locaciones de memoria:
  - Las que reservan memoria sin inicializar.
  - Las que reservan memoria con un determinado valor inicial.
- Estas directivas sólo pueden aparecer en las secciones **`.bss`** o **`.data`**, respectivamente.
- Cada locación de memoria puede o no tener una etiqueta asociada.

Locaciones sin inicializar

- Las siguientes directivas reservan locaciones de memoria sin inicializar:
  - **resb** para reservar bytes (**8** bits)
  - **resw** para reservar words (**16** bits)
  - **resd** para reservar doubles (**32** bits)
  - **resq** para reservar quads (**64** bits)
- Ejemplo:

```
section .bss
    buffer resb 256 ; reserva 256 bytes
```

Locaciones inicializadas

- Las siguientes directivas reservan locaciones de memoria inicializadas:
  - **db** para reservar bytes (**8** bits)
  - **dw** para reservar words (**16** bits)
  - **dd** para reservar doubles (**32** bits)
  - **dq** para reservar quads (**64** bits)
- Ejemplo:

```
section .data
    contador dd 0 ; reserva espacio para
                  ; una variable de 32 bits
```

Etiquetas

- Las etiquetas son básicamente direcciones de memoria.
- Pueden apuntar a una locación que contiene datos o bien que contiene código.
- En caso de comenzar con un punto, se consideran locales (esto es, pueden repetirse varias veces en el mismo archivo):

```
.loop  ...
    jnz loop ; salta a la etiqueta
             ; .loop más cercana.
```

Registros del procesador

- La arquitectura **i386** tiene **8** registros:
  - **ESP** y **EBP**: suelen tener un rol específico.
  - **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**: registros de propósito general.
- La memoria funciona como un gran arreglo.
  - Se direcciona al byte.
- El procesador está en modo protegido.
  - La ejecución de un programa no puede afectar a los restantes programas en ejecución.

Registros del procesador

- La versión de **16** bits de los registros (esto es, sus **16** bits menos significativos) también pueden ser accedidos:
  - Por caso: **AX**, **BX**, **SI**, **SP**, etc.
- Algunos de estos registros permiten a su vez acceder a los dos campos de **8** bits que lo componen.
  - Por caso: **AX** se compone de **AH** y **AL**.
- En contraste, los **16** bits más significativos no son referenciables de manera directa.

Instrucción MOV

- Sintaxis: **`mov dest, origen`**

```
mov ebx, 3    ; guarda un 3 en EBX
mov eax, ebx  ; copia EBX en EAX
mov eax, CONT ; guarda CONT en EAX
```

- Restricciones:
  - El destino no puede ser una constante.
  - Se puede hacer referencia de a lo sumo una dirección de memoria.
  - Origen y destino tienen que ser compatibles.

Accediendo a memoria

- Los corchetes denotan al modo de direccionamiento indirecto:

```
mov eax, CONT   ; guarda CONT en EAX
mov ecx, [eax]  ; guarda el contenido
                ; de CONT en ECX
mov ecx, [CONT] ; inst. equivalente
```

- La arquitectura **i386** permite acceder a memoria mediante el siguiente modo:

[base + escala × índice + offset]

Accediendo a memoria

- El requisito de compatibilidad de los operandos implica que a veces tengamos que explicitar el tamaño de los mismos:

```
mov dword [CONT], 1
mov eax, CONT
mov ebx, [CONT]
mov [eax], byte 1
```

- Los modificadores posibles son **byte**, **word** y **dword**.

Instrucciones aritméticas

- Operaciones aritméticas disponibles:
  - **`add destino, operando`**
  - **`sub destino, operando`**
  - **`inc destino`**
  - **`dec destino`**
  - **`not destino`**
  - **`neg destino`**
  - **`mul operando`** / **`imul operando`**
  - **`div divisor`** / **`idiv divisor`**

Estructuras de control

- Los lenguajes de alto nivel cuentan con diversas estructuras de control.
- El lenguaje ensamblador es más elemental, sólo cuenta con las siguientes facilidades:
  - La instrucción **`cmp`** para comparar valores entre sí.
  - Los flags del procesador para recordar el resultado de la última comparación u otras operaciones.
  - Los saltos para alterar el flujo de ejecución de forma condicionada o no.

Principales flags

- La arquitectura **i386** cuenta con diversos flags del procesador, los cuales reflejan el resultado de la última comparación.
  - Para los enteros no signados se usan los flags zero (**ZF**) y carry (**CF**).
  - Para los enteros signados se usan los flags overflow (**OF**) y sign (**SF**).
- El resultado de la mayoría de las operaciones aritméticas también afectan los flags.

Instrucción CMP

- Sintáxis: **`cmp primero, segundo`**
- Esta instrucción computa **`primero – segundo`**, modificando los flags de manera acorde.
  - Si resultado = 0 (**`primero == segundo`**):
    `ZF = 1; CF = 0;`
  - Si resultado > 0 (**`primero > segundo`**):
    `ZF = 0; CF = 0; SF = OF;`
  - Si resultado < 0 (**`primero < segundo`**):
    `ZF = 0; CF = 1; SF != OF;`

Instrucción JMP

- Sintaxis: **`jmp dest`** / **`jmp short dest`**

  ```
  jmp infinite-loop

  jmp short label-cercano
  ```

- Los saltos incondicionales siempre transfieren el control a una cierta dirección de memoria sin tener en cuenta el estado de los flags.
  - La instrucción a continuación de un **`jmp`** jamás será ejecutada...
  - ...salvo que sea ¡el destino casualmente de un salto!

Saltos condicionales

- En los saltos condicionales no siempre se produce la transferencia de control, habida cuenta que el salto se realiza o no dependiendo del estado de uno o más flags.
  - Los saltos más sencillos dependen del estado de sólo un flag (no son tan frecuentes).
  - Los saltos más complejos dependen del estado de múltiples flags a la vez (son más frecuentes).

Saltos condicionales

- Saltos simples:
  - **`jz dest`** / **`jnz dest`**: depende de **ZF**.
  - **`jo dest`** / **`jno dest`**: depende de **OF**.
  - **`js dest`** / **`jns dest`**: depende de **SF**.
  - **`jc dest`** / **`jnc dest`**: depende de **CF**.
  - **`jp dest`** / **`jnp dest`**: depende de **PF**.
- Si el flag está activo se produce el salto.
- Si el flag no está activo, la ejecución continúa en la instrucción siguiente al salto condicional.

Saltos condicionales

- Saltos condicionales para valores signados (se asume que se acaba de ejecutar la instrucción **`cmp eax, ebx`**):
  - **`je dest`**: salta si **`eax = ebx`**.
  - **`jne dest`**: salta si **`eax ≠ ebx`**.
  - **`jl dest`** / **`jnge dest`**: salta si **`eax < ebx`**.
  - **`jle dest`** / **`jng dest`**: salta si **`eax ≤ ebx`**.
  - **`jg dest`** / **`jnle dest`**: salta si **`eax > ebx`**.
  - **`jge dest`** / **`jnl dest`**: salta si **`eax ≥ ebx`**.

Saltos condicionales

- Saltos condicionales para valores no signados (se asume que se acaba de ejecutar la instrucción **`cmp eax, ebx`**):
  - **`je dest`**: salta si **`eax = ebx`**.
  - **`jne dest`**: salta si **`eax ≠ ebx`**.
  - **`jb dest`** / **`jnae dest`**: salta si **`eax < ebx`**.
  - **`jbe dest`** / **`jna dest`**: salta si **`eax ≤ ebx`**.
  - **`ja dest`** / **`jnbe dest`**: salta si **`eax > ebx`**.
  - **`jae dest`** / **`jnb dest`**: salta si **`eax ≥ ebx`**.

Estructuras de control

- La estructura de control condicional se puede codificar con facilidad en ensamblador:

```
; if (a > 15) { b = 32; } else { a = a + 1; }
mov eax, [a]
cmp eax, 15          ; comparo a con 15
jng else             ; ir a “else” si ≤ 15
mov [b], word 32     ; brazo “then”
jmp seguir           ;
else: inc eax        ; brazo “else”
seguir: ...          ; resto del programa
```

Estructuras de control

- Las restantes estructuras de control también se pueden codificar de una manera similar.
- Otra posibilidad es inspeccionar el código ensamblador generado por un compilador de lenguaje **C** ante las distintas estructuras de control:

```
$ gcc -S prueba.c
$ cat prueba.s
...
```

Llamadas al sistema

- El sistema operativo brinda sus servicios a los distintos programas bajo la forma de llamadas al sistema.
- La mayor parte de las llamadas al sistema involucra conceptos que serán abordados recién en otras asignaturas.
- En esta materia haremos uso principalmente de las llamadas al sistema que permiten manipular archivos.

Llamadas al sistema

- La invocación de llamadas al sistema bajo el **SO GNU/Linux** es similar a la invocación de un procedimiento en **C**.
- La interrupción **80h** es el portal de acceso a todos los servicios de este **SO**.
- El servicio solicitado se indica en el registro **EAX** y sus parámetros usualmente en los restantes registros del procesador.
- Si el servicio retorna algún valor, lo hará también haciendo uso del registro **EAX**.

Llamadas al sistema

- Principales llamadas al sistemas provistas por **GNU/Linux**:
  - **sys_exit**: Para finalizar de un proceso.
  - **sys_open**: Para abrir un archivo.
  - **sys_close**: Para cerrar un archivo.
  - **sys_read**: Para leer un archivo preexistente.
  - **sys_write**: Para escribir un archivo preexistente.
  - **sys_creat**: Para crear un nuevo archivo.

Fuentes de información

- La sección **2** de la documentación que acompaña al **SO GNU/Linux** contiene una detallada descripción de la forma de invocación a los distintos servicios.

```
$ man 2 write
```

- La documentación está pensada para invocar la llamada al sistema desde el lenguaje **C**, pero la adaptación para el caso de estar invocándola desde lenguaje ensamblador es relativamente directa.

Convenciones

- Por ejemplo, para invocar a la llamada al sistema **`sys_write`**, se requiere:
  - Cargar el número de servicio solicitado (**`sys_write`** es el **4**) en el registro **EAX**.
  - Cargar el descriptor del archivo que queremos escribir en el registro **EBX**.
  - Cargar el puntero al buffer conteniendo lo que queremos escribir en el registro **ECX**.
  - Cargar la cantidad de bytes que queremos escribir en el registro **EDX**.

Convenciones

- En general, el pasaje de parámetros en las llamadas al sistema adopta dos variantes en función de la cantidad de argumentos:
  - Para pasar de cero a seis argumentos se usa pasaje de parámetros en registros, usando los registros **EBX**, **ECX**, **EDX**, **ESI**, **EDI** y **EBP**, en ese orden.
  - Para pasar más de seis argumentos se usa pasaje de parámetros en memoria, usando el registro **EBX** para indicar la dirección de comienzo de la lista de parámetros.

Nuestro primer programa

```
section .data
texto db 'Hola mundo!',10
largo equ $ - texto ; $ denota el offset
                    ; actual

section .text
global _start ; etiqueta global que
              ; marca el comienzo
              ; del programa
```

Nuestro primer programa

```
_start:
mov eax, 4      ; servicio sys_write
mov ebx, 1      ; standard output
mov ecx, texto  ; offset de mensaje
mov edx, largo  ; largo del mensaje
int 80h         ; invocación al servicio
mov eax, 1      ; servicio sys_exit
mov ebx, 0      ; terminación sin errores
int 80h         ; invocación al servicio
```

Invocación a procedimientos

- La invocación a un procedimiento no difiere en gran medida de los saltos convencionales.
- Se adopta la convención de dejar en el tope la pila del programa la dirección a la que se debe retornar una vez finalizada la ejecución del procedimiento.
- Las instrucciones **`call`** y **`ret`** simplifican la invocación a procedimientos y el posterior retorno una vez finalizados.

Ejemplo

```
start_:
    mov eax, [UNVALOR]
    call triplicar      ; procedimiento para triplicar
    ...                 ; el valor almacenado en EAX
triplicar:
    push ecx            ; preservo ECX en la pila
    push edx            ; preservo EDX en la pila
    mov ecx, 3
    mul ecx             ; multiplico por 3
    pop edx             ; recupero EDX de la pila
    pop ecx             ; recupero ECX de la pila
    ret                 ; retorno
```

Pila del programa

- La pila del programa (**stack**) es una de las regiones que forma parte de los programas en ejecución.
  - Su tamaño es dinámico, crece o disminuye según se requiera.
  - Crece hacia abajo desde la dirección más alta.
  - Es utilizada para almacenar datos temporalmente, tales como la dirección de retorno al invocar a un procedimiento o el valor actual de un registro el cual será alterado, por caso, durante la invocación a un servicio del sistema operativo.

Pila del programa

- Operaciones sobre la pila del programa:
  - Instrucción **push**: resta el tamaño en bytes del objeto apilado al valor actual del registro **ESP** (stack pointer) y lo almacena en el lugar ahora apuntado por **ESP**.
  - Instrucción **pop**: recupera el objeto a ser desapilado de la locación apuntada por el registro **ESP** y lo actualiza sumándole el tamaño en bytes de ese objeto. Nótese que el objeto sigue en memoria, pero será sobrescrito cuando el tamaño de la pila vuelva a crecer.

Caller save vs. callee save

- Para preservar el contenido de los registros que serán alterados durante la ejecución de un procedimiento surgen dos alternativas:
  - Una posibilidad (denominada “caller save”) es que quien llama al procedimiento se encargue de salvar en la pila del programa aquellos registros que se necesiten preservar.
  - Otra posibilidad (denominada “callee save”) es que el procedimiento que es llamado se encargue de preservar en la pila del programa sólo aquellos registros que serán modificados.

Caller save vs. callee save

- Cada alternativa tiene ventajas y desventajas:
  - Caller save permite optimizar múltiples llamados a un procedimiento, evitando que cada una de las invocaciones preserve y recupere de la pila múltiples registros.
  - Callee save, por otra parte, posibilita que el procedimiento sólo preserve los registros que serán afectados (el llamador usualmente no tiene acceso a esta información).
- La idea es adoptar sólo uno de estos esquemas a lo largo de un mismo programa.

Pasaje de parámetros

- Los procedimiento en ocasiones requieren recibir información adicional al ser invocados, esto es, requieren un conjunto de parámetros.
- El pasaje de estos parámetros se puede resolver de diversas formas:
  - Usando los registros del procesador.
  - Usando una región de la memoria.
  - Usando la pila del programa.

Pasaje de parámetros

- Usando registros:
  - Este es por lejos el mecanismo más eficiente (pues no implica accesos adicionales a memoria).
  - La cantidad de parámetros está limitada a la cantidad de registros disponibles.
- Usando la memoria:
  - Este mecanismo es más flexible que el anterior, pues permite una cantidad mucho mayor de parámetros.
  - Naturalmente, tiene como desventaja que implica accesos adicionales a memoria.

Pasaje de parámetros

- Usando la pila del programa:
  - El pasaje de parámetros usando registros o usando la memoria no resulta del todo apropiado a la hora de implementar procedimientos recursivos.
  - En contraste, el pasaje de parámetros usando la pila del programa permite la implementación de procedimientos recursivos con relativa simpleza.
  - Nótese que la dirección de retorno seguirá ocupando el tope de la pila, es decir, los parámetros serán apilados por debajo (lo que implica que la pila no será accedida como una pila).

Reentrancia

- ¿Por qué resulta más conveniente usar la pila, que está memoria, que directamente un región de la memoria?
  - En otras palabras, ¿por qué en un caso podemos implementar soluciones recursivas y en el otro no?
- La diferencia entre las dos alternativas radica en que usando la pila obtenemos lo que se denomina un procedimiento reentrante o puro.

Reentrancia

- Para alcanzar la reentrancia hace falta que el código no sea automodificable y que sólo se opere sobre los registros o la pila del programa.
  - Para apreciar las ventajas de este tipo de código hay que recordar que los sistemas en la actualidad son multiprogramados, es decir, existe más de un programa en ejecución a la vez.
  - Bajo esta definición, la recursión consiste meramente en sacar provecho de la reentrancia en una determinada rutina.

Comandos de la consola

- La consola que brinda **GNU/Linux** nos da acceso a diversos comandos:
  - **`ls ruta`**: lista el contenido de un directorio.
  - **`ls -l ruta`**: lista el contenido de un directorio mostrando más detalles.
  - **`cd directorio`**: cambia el directorio actual
  - **`rm archivo`**: borra un archivo.
  - **`cp origen destino`**: copia un archivo.
  - **`mv origen destino`**: mueve un archivo.

Comandos de la consola

- Continúa:
  - **`mkdir directorio`**: crea un directorio.
  - **`rmdir directorio`**: borra un directorio.
  - **`cat archivo`**: lista el contenido de un archivo sin paginar.
  - **`man tópico`**: muestra la ayuda (si es que existe) acerca del tópico indicado.
  - **`echo cadena`**: muestra la cadena de caracteres por pantalla.

Comandos útiles

- El comando **alias** permite crear sinónimos de los comando usados frecuentemente:

```
$ alias n=nano
$ n prueba.asm
$ alias asm=”yasm -f elf”
$ asm prueba.asm
```

- La invocación al comando **alias** sin argumentos lista por pantalla el conjunto de los sinónimos activos.

Consejo práctico

- El intérprete de comandos del **SO GNU/Linux** es muy poderoso.
- Presionando la tecla **TAB**, estamos solicitando al intérprete que intente autocompletar lo que estamos escribiendo.
- Si nos acostumbramos a usarlo, es posible alcanzar una velocidad de escritura mediante el teclado antológica.

Parámetros en la línea de comandos

- El **SO GNU/Linux** simplifica el acceso desde el lenguaje ensamblador a los parámetros pasados en la línea de comandos.
- Se adopta una convención similar a la empleada bajo el lenguaje **C**.
- El cargador se encarga que al comenzar la ejecución el tope de la pila del sistema esté ocupado por las conocidas estructuras de datos **ARGC** y **ARGV**.

Parámetros en la línea de comandos

- La instrucción **pop** permiten acceder a los parámetros en la línea de comandos recibidos:

```
_start:
pop eax ; EAX contiene a ARGC
pop ebx ; EBX almacena un puntero a ARGV[0],
        ; el nombre del programa actual
pop ecx ; ECX almacena un puntero a ARGV[1],
        ; el primer argumento recibido
...
```

Ejemplo

```
section .text
global _start

_start:
pop eax    ; EAX contiene a ARGC
cmp eax, 2 ; ¿nos pasaron exactamente un argumento?
je seguir  ; de ser así, seguir;
           ; caso contrario, indicar el error.
mov eax, 1 ; sys_exit
mov ebx, 1 ; terminación con error
int 80h    ; invocación del servicio del SO
```

Ejemplo

```
seguir:
mov eax, 4    ; sys_write
mov ebx, 1    ; stdout
pop ecx       ; descartamos ARGV[0]
pop ecx       ; conservamos ARGV[1]
mov edx, 1    ; imprimir sólo un carácter
int 80h       ; invocación del servicio del SO

mov eax, 1    ; sys_exit
xor ebx, ebx ; terminación normal
int 80h       ; invococión del servicio del SO
```

Resultado de la ejecución

- La variable **$?** contiene el resultado de la ejecución del último programa:

```
$ yasm -f elf pasaje.asm && ld -o pasaje pasaje.o
$ ./pasaje
$ echo $?
1
$ ./pasaje abc
a
$ echo $?
0
```

Ejecución de un programa

- Como hemos visto, un programa atraviesa distintas etapas desde su concepción hasta llegar a estar corriendo en una arquitectura:
  - En primer lugar hay que escribir el código fuente.
  - Más tarde hay que convertir ese código fuente en código objeto.
  - Luego, se debe convertir el código objeto (de uno o más archivos fuente) en el código ejecutable.
  - Finalmente, se debe cargar en memoria el código ejecutable y comenzar su ejecución.

Etapa de compilación

- Características:
  - Durante esta etapa el código fuente es convertido en código objeto instrucción por instrucción.
  - Se resuelven la totalidad de las referencias locales.
  - Se identifican y dejan pendientes de resolver a la totalidad de las referencias externas.
  - El código objeto es prácticamente ejecutable, sólo resta incorporar el código objeto asociado a otros módulos o bien a las funciones de librería que se hayan utilizado.

1-pasada vs. 2-pasadas

- En relación a la resolución de las referencias locales, es interesante tener en cuenta cómo se lleva adelante esa resolución.
  - Es decir, al encontrar una referencia local, es decir, una etiqueta, ¿cómo se puede saber a qué locación de memoria se está referenciando?
- El compilador cuenta con una estructura de datos interna denominada tabla de símbolos, donde anota la dirección que asocia a las distintas declaraciones de etiquetas que va encontrando durante el ensamblado.

1-pasada vs. 2-pasadas

- Naturalmente, la tabla de símbolos sólo contiene la dirección de las etiquetas cuyas declaraciones ya visitó.
- Es decir, en principio no es posible resolver las referencias a etiquetas aún no declaradas (por caso, saltos hacia adelante).
- Esto implica que el ensamblador deba realizar dos pasadas sobre el código fuente.
  - Ahora bien, ¿será factible resolver las referencias en sólo una pasada?

1-pasada vs. 2-pasadas

- La respuesta es que sí, veamos cómo:
  - La clave radica en tener presente que tanto programa fuente como tabla de símbolos coexisten en el mismo espacio de direccionamiento (la memoria principal).
  - Al encontrar una referencia a una etiqueta aún no declarada, se coloca en la tabla de símbolos una referencia a esa locación (en la cual posteriormente se deberá actualizar la dirección de la etiqueta).
  - De aparecer otras referencias a esa misma etiqueta, se van enlazando las locaciones en donde eventualmente se deberá poner la dirección correcta.

1-pasada vs. 2-pasadas

- Continúa:
  - Cuando eventualmente se alcance la declaración de esa etiqueta, se recorre la “lista enlazada” arrancando en la dirección almacenada en la tabla de símbolos reemplazando las referencias con la dirección recién descubierta.
  - Caso contrario, de alcanzar el fin del archivo fuente simplemente se señala el error de compilación.
  - Cabe acotar que esta idea sólo es aplicable si en el espacio asignado a la etiqueta dentro de la instrucción cabe una dirección completa.

Etapa de vinculación

- Características:
  - Durante esta etapa el código objeto de uno o más archivos fuentes es convertido en código ejecutable.
  - La principal tarea del vinculador es resolver la totalidad de las referencias externas.
  - Las referencias a otros módulos se resuelven a partir de código objeto de ese módulo (el cual debe suministrarse en ese momento).
  - Las referencias a librerías se resuelven usando el código objeto de la librería, el cual estará disponible en alguna ubicación conocida de antemano.

Etapa de carga

- Características:
  - El cargador, un componente del sistema operativo, es el encargado de cargar a memoria el código ejecutable de un cierto programa.
  - A esta altura todas las referencias externas están resueltas, sólo resta conocer la dirección inicial a partir de la cual será cargado el código ejecutable.
  - Los modelos más avanzados de manejo de memoria han simplificado notablemente esta tarea (por caso, al contar con un espacio de direccionamiento de uso exclusivo para cada proceso).

Vinculación dinámica

- Como es frecuente que múltiples programas compartan la misma librería (por caso, **`stdio`**), en la actualidad es posible vincular múltiples programas a una misma instancia en memoria de la librería.
  - Desde ya, esta librería debe estar compuesta exclusivamente de procedimientos reentrantes.
  - Nótese que esas referencias externas no podrán ser resueltas en tiempo de vinculación, razón por la cual es este tipo de librería se las denomina de vinculación dinámica (por caso, las librerías **`.DLL`**).

¿Preguntas?