# Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)

*Universitat Politècnica de València*

fso

## Práctica 8 v2.0
## Análisis del Mapa de Memoria (I)

# 1 Objetivos

El objetivo principal de esta práctica es **"analizar el mapa de memoria de un proceso" y comprender** la evolución que sufre durante la ejecución del mismo. Para ello, visualizaremos el archivo ***/proc/PID/maps*** de Linux.

# 2 Mapa de Memoria de un proceso

El espacio de direcciones lógicas de un programa contiene las diferentes secciones en las que se estructura su memoria: área de código, área de datos, áreas para las bibliotecas de enlace dinámico, área para pila, etc. Esta colección de secciones marca la estructura del mapa de memoria de un proceso y cada sistema operativo utiliza la suya propia.

El mapa de memoria de un proceso es gestionado por el sistema operativo y evoluciona durante la ejecución del mismo. Existe una gran similitud o correspondencia entre el contenido del archivo ejecutable que soporta un proceso y su mapa de memoria inicial.

En Linux, el espacio de direcciones lógicas de un programa define la organización interna del código de usuario que el sistema operativo ubicará en memoria para generar el proceso. Esta práctica trabaja con el modelo de direcciones lógicas de los archivos *ELF (Executable and Linking Format)* para versiones Linux con arquitectura PC.

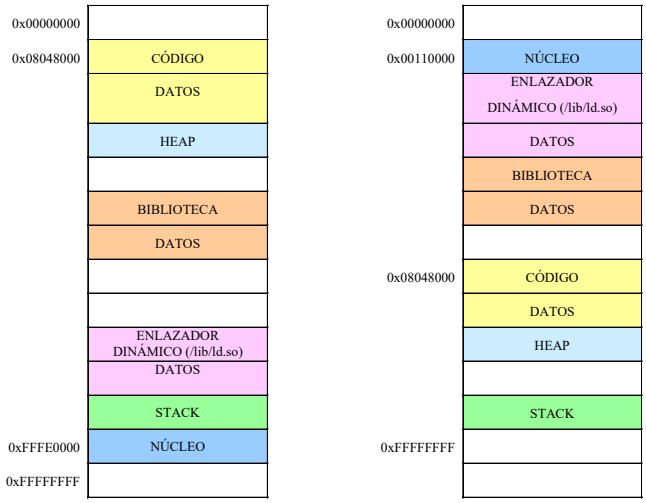


Figura1. Dos posibles organizaciones de la memoria lógica de un proceso para versiones Linux 2.6.x en maquinas de 32 bits.

## *2.1 Mapas de memoria de 32 y 64 bits*

Existen sistemas operativos para arquitecturas de 32 bits y de 64 bits, es decir, para procesadores de 32 y 64 bits de dirección. Con 32 bits se direccionan espacios lógicos de hasta 4GB ($2^{32}$= 4 GB), mientras que con 64 bits se direccionan espacios lógicos de hasta 16 EB ($2^{64}$ = 16 ExaByte).

Dentro de la familia Intel de procesadores, la compatibilidad hacia atrás permite instalar un sistema operativo de 32 bits sobre un procesador de 64 bits y ejecutar procesos de 32 bits bajo un sistema operativo de 64 bits. Sin embargo, no es posible gestionar procesos de 64 bits en un sistema operativo de 32 bits, tampoco funciona un sistema operativo de 64 bits sobre un procesador de 32 bits. Pueden coexistir procesos de 64 y de 32 bits en un sistema operativo de 64 bits, pero en un sistema operativo de 32 bits sólo hay procesos de 32 bits.

El compilador de Linux permite escoger qué tipo de mapa de direcciones usará el código que genere. Lo habitual es que el compilador sólo disponga de las bibliotecas apropiadas para el sistema en que opera y que, por omisión, genere el código acorde a ello. Pero puede hacer compilación "cruzada" si dispone de las bibliotecas necesarias, es decir, podrá generar código para un tipo de sistema diferente. Para procesadores Intel, gcc genera código de 32 bits usando la opción "-m32". Esta es la opción por omisión en las versiones de linux 32 bits, pero hay que explicitarla en la línea de órdenes para sistemas de 64 bits.

En esta práctica vamos a utilizar mapas de memoria de 32 bits. En las **órdenes de compilación**, debe poner la opción **"-m32"** ya que el sistema operativo instalado en el laboratorio es de 64 bits. Si quiere, puede probar a analizar un mapa de 64 bits, y comprobar su complejidad.

## *2.2 Regiones del mapa de memoria y sus propiedades*

El mapa de memoria de un proceso está formado por varias regiones, como se muestra en la figura-1. Cada **región** es una **zona contigua del espacio lógico del proceso** caracterizada por la dirección dentro del mapa, el tamaño y una serie de propiedades:

•**Soporte de la región**. Existen dos posibilidades

- *Región con soporte en archivo*: El contenido de la región se almacena en un archivo.
- *Región sin soporte*: La región no tiene contenido inicial.

•**Tipo de uso de la región**. Indica si está permitido compartir la región entre procesos.

- *Privada (p):* El contenido de la región sólo es accesible por el proceso.
- *Compartida (s):* El contenido de esta región puede ser accesible por varios procesos.

•**Protección:** Tipo de acceso permitido a la región: lectura (r), escritura (w), ejecución (x).

•**Tamaño fijo o variable**: Existen regiones cuyo tamaño puede variar durante la ejecución del proceso y otras que el tamaño es fijo (no varía).

**Las regiones** en el **mapa de memoria inicial** del proceso son:

•**Código o texto.** Región compartida de lectura (r) /ejecución (x), tamaño fijo y soportada por el fichero ejecutable. Una región marcada como "p" sin permisos de escritura, en la práctica estará compartida. En versiones anteriores del núcleo de Linux, dichas regiones recibían la marca "s".
r-xp Región de código

•**Datos con valor inicial**. Región privada (p), cada proceso necesita una copia propia de las variables, es de lectura(r)/escritura(w), tamaño fijo y soportada por el fichero ejecutable.
rw-p. Región de datos inicializados cuando está soportada por un fichero (DATOS).

•**Datos sin valor inicial**. Región privada (p), es de lectura(r)/escritura(w), tamaño fijo y no está soportada ya que su contenido inicial es irrelevante.
rw-p. Región de datos no inicializados cuando no está soportada por un fichero (BSS).

•**Pila**. Región privada (p) de lectura(r)/escritura(w) de tamaño variable, sirve de soporte para almacenar las variables locales, parámetros y direcciones de retorno de las llamadas a funciones.

El mapa de memoria de un proceso es dinámico, con regiones que pueden añadirse o eliminarse durante su ejecución. Algunas de las nuevas regiones que pueden crearse son:

•**Heap.** Región privada de r/w sin soporte (inicialmente se rellena a ceros) y que crece según reserva memoria el proceso y decrece cuando la libera. Sirve de soporte a la memoria dinámica que un proceso reserva en tiempo de ejecución (en lenguaje C utilizando la función "malloc").

•**Archivos proyectados**. Cuando se proyecta un archivo sobre el espacio lógico del proceso se crea una región para ello, cuya protección la especifica el proceso a la hora de proyectarla.

•**Memoria Compartida**. Cuando se crea una zona de memoria compartida, aparece una región de carácter compartido en el mapa.

•**Pilas de threads**. Cada *thread* necesita una pila propia.

## 2.3 El archivo /proc/PID/maps

El directorio */proc* contiene archivos con información sobre el sistema y los procesos. La información propia de cada proceso se encuentra en un directorio identificado por el PID del proceso. Para facilitar el acceso de un proceso a su propia información existe un directorio ***self***, para cada uno, que en realidad se implementa mediante un enlace simbólico al directorio correspondiente a dicho proceso. Así cuando un proceso con PID 2975 accede a */proc/self/* en realidad está accediendo al directorio */proc/2975/*.

Esta práctica trabaja con la información contenida en el archivo *maps* de un proceso.

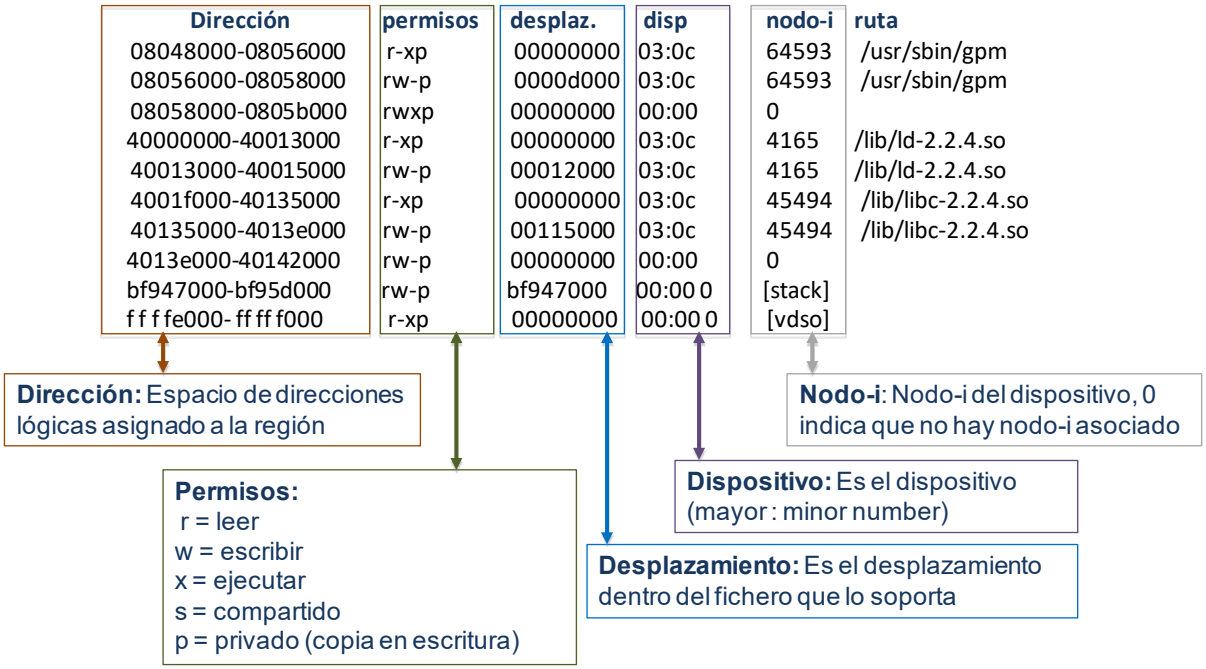


Figura 2. Contenido de las diferentes columnas que aparecen al listar el archivo *maps* de un proceso.

En la siguiente tabla se resumen los diferentes tipos de regiones observados en la figura 2.

| | |
|---|---|
| Regiones de código | Código propio del programa<br>Código de biblioteca dinámica (.so)<br>Región VDSO |
| Regiones de datos | Región de datos respaldada por el fichero ejecutable<br>Región de datos no respaldada<br>Pila ([stack]) |

Tabla-1 Algunas regiones que se observan en un mapa de memoria

La línea del [vdso] de la figura-2, corresponde a código del sistema operativo. En concreto se trata del *Virtual Dynamically-linked Shared Object*, una librería compartida que permite al proceso realizar unas pocas acciones del kernel sin la sobrecarga de una llamada a sistema.

### *Ejercicio 1: Experimente con /proc/self*

| |
|---|
| Ejecute dos veces seguidas la orden: $ ls -l /proc/self<br>Explique porqué salen dos resultados distintos. |
| La orden *head -1* muestra la primera línea de un archivo de texto o de la entrada estándar. Pruebe las siguientes órdenes:<br>$ head -1 /proc/self/maps<br>$ cat /proc/self/maps \| head -1<br>Justifique los resultados obtenidos |

## *2.4 Herramientas y órdenes: system() y size*

Analizaremos el mapa de memoria de un proceso LINUX visualizando el archivo */proc/PID/maps*, donde PID representa el identificador del proceso. Durante la ejecución de un proceso utilizaremos la orden *cat* y la ejecutaremos con *system()* para mostrar el contenido del fichero *maps*. Para ello se debe construir un string con la cadena *"cat /proc/PID/maps"* e invocar la llamada *system(cadena)*. Las siguientes líneas de código se deben incluir en aquellos puntos del programa donde desee mostrar el mapa de memoria del proceso para analizarlo.

```
sprintf(path_maps,"cat /proc/%d/maps",getpid()); //Crea orden para mostrar MAPA
flush(stdout);                                    /*vacia buffer*/
system(path_maps);                 /*Llamada al sistema para ejecutar orden**/
```

Figura 3. Código para visualizar el mapa de memoria de un proceso

La orden del ***shell size*** muestra el tamaño de cada una de las secciones de un archivo ejecutable, imprimiendo en pantalla el tamaño de cada una de las diferentes regiones que componen el proceso, para más información sobre dicha orden haga *$man size*.

```
$ size /bin/ls
   text    data     bss     dec     hex filename
 101164    1896    3424  106484   19ff4 /bin/ls
```

# 3 Mapa con variables locales, globales y parámetros de funciones

Como ya conoce el alumno, cuando se habla de variables se puede distinguir entre:

-**Variables locales**: se encuentran definidas dentro de una función, y sólo pueden ser accedidas dentro de dicha función.

-**Variables globales:** se definen fuera del cuerpo de cualquier función, y pueden ser accedidas por cualquier función del fichero fuente.

-**Parámetros de una función**: son los valores que recibe una función por el código que la invoca.

Comprobaremos que estos tres tipos de variables no tienen por qué encontrarse ubicadas en la misma región del mapa de memoria de un proceso, aunque podrían estarlo. Trabaje con el archivo "map1.c" proporcionado como material de prácticas que contiene el código mostrado en la figura-4.

```
/* map1.c code*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char path_maps[80]; /* Global variable */
char ni_glob[4095];
long i_glob=20;      /* Initialized global variable */

void f(int param)
{
    printf("Address of funcion f parameter: %p\n",&param);
}

int main() {
    long i_loc=20; /* Initialized local variable */
    long ni_loc; /* Non initialized local variable */
    printf("Process PID: %d\n\n", getpid()); /* Shows process PID */

     /**** ADDRESS VISUALIZATION ***/
    printf("main function address: %p \n", main);
    printf("f function address: %p \n", f);
    printf("Initialiazed global variable i_glob address: %p\n", &i_glob);
    printf("Non initialized global variable ni_glob address \n");
    printf(" 1st ni_glob element address: %p\n", &ni_glob[0]);
    printf(" Last ni_glob element address: %p\n", &ni_glob[4095]);
    printf("Initialized local variable i_loc address: %p\n", &i_loc);
    printf("Non initialized local variable ni_loc address: %p\n", &ni_loc);
    f(40);
    printf("\n PROCESS MEMORY MAP /proc/%d/maps \n", getpid());
    // Create command "path_maps" to show memory map
    sprintf(path_maps, "cat /proc/%d/maps",getpid());
    // Empty the buffer
    fflush(stdout);
    // Execute command "path_maps"
    system(path_maps);
    printf ("            ---------------\n\n");
    return 0;
}
```

Figura 4. Código del archivo fuente "map1.c"

## *3.1 Ejercicio2: Mapa de memoria de un proceso básico*

Compile y ejecute el código de *map1.c*, para ello haga lo siguiente:

```
$ gcc –m32 map1.c –o map1
$ ./map1
```

***Cuestión-1***: Analice el mapa de memoria mostrado por map1 y consultando tanto la figura 2 como el manual del sistema ($man proc), responda a las siguientes preguntas:

| |
|---|
| 1. ¿Cuál es el rango de direcciones lógicas ocupado por el proceso "*map1*"? |
| 2. Identifique la región del código y anote su dirección inicial y final. Haga lo mismo con la región de pila. |
| 3. ¿Cuántas regiones de las estudiadas puede identificar en el mapa de memoria del proceso "map"? Identifique al menos, una región de datos respaldada, una región de datos no respaldada y una de código de biblioteca |
| 4. ¿Qué regiones de "*map1*" se encuentran soportadas por dispositivos físicos? Justifique la necesidad de dicho soporte. |
| 5. ¿Qué número de nodo-i tienen asignadas las regiones que no se encuentran soportadas? |
| 6. ¿Qué regiones de "*map1*" podrían ser compartidas por varios procesos? |
| 7. Identifique en qué región, de las indicadas en la Tabla 1, se encuentra la función *main*, así como la función f. |

***Cuestión-2***: Indique en qué regiones, de las indicadas en la Tabla 1, ha ubicado el sistema cada una de las diferentes variables del programa.

| **Tipo de Variable** | **Región en la que se ubica** |
|---|---|
| Variables globales inicializadas | |
| Variables globales no inicializadas:<br>`path_maps[80]`<br>`ni_glob[4095]` | |
| Variables locales inicializadas | |
| Variables locales no inicializadas | |
| Parámetros de función | |

### 3.2 Ejercicio3: Mapa de un proceso con variables locales de gran tamaño

Trabaje con el archivo map2.c, proporcionado con el material de prácticas, cuyo código es similar al del ejercicio anterior, figura-4, salvo que contiene una nueva función f2 como muestra la figura 5.

```
void f2() {
    int vloc[1024*1024];
    printf("Local variable vloc address: %p\n", vloc);
    printf("\n PROCESS MEMORY MAP (in function f2) \n");
    fflush(stdout);
    system(path_maps);
    printf(" ---------------\n\n");
}
```

Figura 5. Código de la función f2, incluida en map2.c, con variable local de gran tamaño.

En la función f2, se define una variable local de gran tamaño: *int vloc[1024*1024]*. El programa *map2.c* imprime el mapa de memoria antes de llamar a la función f2 y dentro de dicha función. Compare ambos mapas de memoria y observar qué cambios se han producido relacionándolos con la variable *vloc*.

Para ello compile *map2.c* y ejecútelo:

```
$gcc –m32 map2.c –o map2
$ ./map2
```

Analice el mapa de memoria mostrado por map2 y responda a las siguientes cuestiones:

***Cuestion-3**:* Indique en qué región del mapa de memoria se encuentra la variable local *vloc*.

***Cuestion-4**:* Compare los mapas resultantes de los procesos *map1 y map2,* e indique si el tamaño de sus respectivas pilas es diferente. Indique en cuál de ellos es mayor y justifíquelo.

# 4 Mapa de memoria: Reserva dinámica de memoria

La reserva dinámica de memoria es aquella que solicita un proceso durante su ejecución. En Java, la reserva dinámica de memoria se hace mediante el operador *new*, mientras que en C se realiza mediante la función *malloc*. En el anexo de esta práctica tiene más información sobre la función *malloc* y la reserva dinámica de memoria en C.

La reserva dinámica de memoria está soportada por una región de memoria denominada ***heap***. Se trata de una región de datos no respaldada por archivo (inicialmente se rellena a ceros), que crece según reserva memoria el proceso y decrece cuando la libera.

## 4.1 Ejercicio 4: mapa de un proceso con reserva dinámica

Trabaje con el archivo map3.c, que se proporciona con el material de prácticas, y que contiene las siguientes declaraciones e instrucciones:

```
/* map3.c code */
int *vdin;
/*** Dynamic memory allocation ***/
vdin = (int *) malloc(100*sizeof(int));
……
/** Free previously allocated memory**/

free(vdin);
```

Figura 6. Líneas de código de map3.c donde se hace la reserva dinámica de memoria

La línea **`malloc`**`(100*sizeof(int))` hace una reserva dinámica de memoria para un array de 100 enteros y es equivalente a la sentencia de Java: *vdin = new int[100];*
Compile *map3.c* y ejecútelo, para ello haga lo siguiente:

```
$gcc -m32 map3.c –o map3
$ ./map3
```

En *map3* imprime el mapa de memoria antes y después de realizar la reserva de memoria. Compare ambos mapas de memoria y observe qué cambios se han producido y su relación con la variable *vdin*.

***Cuestion-5****:* Analice los mapas de memoria mostrados por map3 y compárelos

| |
|---|
| 1. Indique si aparece o desaparece alguna región. Intente justificar a qué se deben estos cambios. |
| 2. En el caso de que aparezca alguna región, ¿Qué tipo de región es? ¿Cuáles son sus permisos? |
| 3. Indique en qué región se encuentra el contenido del vector *vdin*. |

## 4.2 Ejercicio 5: Aumentando la reserva dinámica

Trabaje con el código de map3.c y aumente la reserva dinámica de memoria como se indica en la tabla, compile y ejecute cada vez.

**Cuestión-6**: Analice qué ocurre con la región donde se encuentra la variable *vdin*, e intente calcular su tamaño

| Reserva dinámica a realizar | Efectos sobre la región |
|---|---|
| `malloc(1000*sizeof(int));` | |
| `malloc(10000*sizeof(int));` | |
| `malloc(20000*sizeof(int));` | |
| `malloc(30000*sizeof(int));` | |

# 5 Mapa de Memoria: Uso de Bibliotecas

Las bibliotecas son archivos binarios que contienen código de rutinas o subprogramas útiles para el usuario que pueden provenir de diferentes entornos como: librerías matemáticas, del sistema operativo, biblioteca para efectuar llamadas al sistema (API del so.) o bien crearlas el propio usuario. Bajo esta perspectiva las librerías son una forma sencilla y versátil de hacer modular y reutilizable el código.

El mecanismo que hace accesible a las aplicaciones dichos módulos se le llama enlace y son de dos tipos enlace estático (no se puede compartir) y enlace dinámico (permite compartir). En Windows, archivos de bibliotecas dinámicas poseen extensión `.DLL` *(Dynamic Link Library),* mientras que las estáticas generalmente terminan en `.LIB`. En *Unix y Linux*, las bibliotecas dinámicas tienen extensión `.so` (*Shared Object)* y las estáticas `.a` *(Archive)*.

En Linux una biblioteca puede ser enlazada al código de un programa de dos formas diferentes:

- **Enlace estático**: El ejecutable es autocontenido: incluye todo el código que necesita la aplicación, es decir, el código propio del proceso más el de las funciones externas que necesita. En el caso de que se disponga de dos versiones de la misma biblioteca, estática y dinámica hay que usar la opción `-static` del compilador. La orden de compilación tendría la forma:

```
$gcc -m32 program.c -static -lXYZ -o program
```

donde `-lXYZ` indica utilizar la librería `libXYZ`. Por ejemplo, `-lm` para utilizar la librería `libm` (librería matemática).

- **Enlace dinámico implícito:** La carga y el montaje de la biblioteca se lleva a cabo en tiempo de ejecución del proceso. Es por tanto en tiempo de ejecución cuando se ha de resolver las referencias del programa a constantes y funciones (símbolos) de la biblioteca y la reubicación de regiones. Esta es la opción por defecto del compilador (sino se especifica lo contrario) ya que busca en primer lugar la versión dinámica de la biblioteca. La orden de compilación es como la anterior pero sin `-static`:

```
$gcc -m32 program.c -lXYZ -o program
```

Como parte del proceso de montaje, siempre que se use al menos una biblioteca dinámica, se incluye en el ejecutable un módulo de montaje dinámico, que se encargará de realizar en tiempo de ejecución,

la carga y el montaje de las bibliotecas dinámicas usadas en el programa. Esta opción genera un archivo ejecutable de menor tamaño el estático.

## 5.1 Ejercicio 6: Mapa de un proceso que hace uso de bibliotecas

El archivo *lib_cos.c* proporcionado con el material de prácticas, contiene el código de un programa que utiliza la función coseno la cual se encuentra dentro de la biblioteca matemática. La figura-7 corresponde a dicho código, cuyo mapa de memoria se imprime antes y después de utilizar la función coseno con el fin de analizar los cambios que experimenta dicha el mapa del proceso.

```
/* lib_cos.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define PI 3.14159265
/* Global variables */
char path_maps[80];
char ni_glob[4095];
long i_glob=20;

void f(int param)
{
    printf("Address of funcion f parameter: %p\n",&param);
}
int main()
{
    /* Local variables */
    long i_loc=20; /* Initialized local variable */
    long ni_loc; /* Non initialized local variable */
    float c;

    printf("Process PID: %d\n\n", getpid()); /* Shows process PID */

    /**** ADDRESS VISUALIZATION ***/
    printf("main function address: %p \n", main);
    printf("f function address: %p \n", f);
    printf("Initialiazed global variable i_glob address: %p\n", &i_glob);
    printf("Non initialized global variable ni_glob address \n");
    printf(" 1st ni_glob element address: %p\n", &ni_glob[0]);
    printf(" Last ni_glob element address: %p\n", &ni_glob[4095]);
    printf("Initialized local variable i_loc address: %p\n", &i_loc);
    printf("Non initialized local variable ni_loc address: %p\n", &ni_loc);
    f(40);

    /** Mathematical operation **/
    c = cos(45*PI/180);
    printf("The mathematical operation result is: %f\n", c);

    printf("\n PROCESS MEMORY MAP /proc/%d/maps \n", getpid());
    sprintf(path_maps,"cat /proc/%d/maps",getpid());
    fflush(stdout);
    system(path_maps);
    printf(" ---------------\n\n");

    return 0;

}
```

Figura 8. Líneas de código de *lib_cos.c*

Compile el programa, tanto con enlace de la librería estática, como dinámica implícita y ejecútelo redireccionando la salida a un archivo, para ello haga:

```
$ gcc  –m32 lib_cos.c –static –lm –o biblio_estatica
$ gcc  –m32 lib_cos.c  –lm –o biblio_dinamica
$ ./biblio_estatica > resul_biblioe
$ ./biblio_dinamica > resul_bibliod
```

**Cuestión 7:** Visualice los ficheros generados, *resul_biblioe* y *resul_bibliod,* y compárelos y responda

| |
|---|
| 1. Qué diferencias encuentra al comparar los mapas de generados con los dos tipos de enlaces |
| 2. Ejecute la orden "*ls -lh*" en los dos archivos ejecutables e intente justificar las diferencias de tamaño observada |
| 3. Aplique el comando *size* a los archivos ejecutables resultantes e intente justificar las diferencias de tamaño encontradas en las regiones. |

# 6 Anexo

## *6.1 Generación de un ejecutable*

En general, una aplicación estará formada por un conjunto de módulos que contienen código fuente y que han de ser compilados y montados, como describe la figura-6. El compilador genera el código máquina de cada módulo fuente y el montador genera un único archivo ejecutable agrupando todos los módulos y resolviendo las referencias entre ellos.

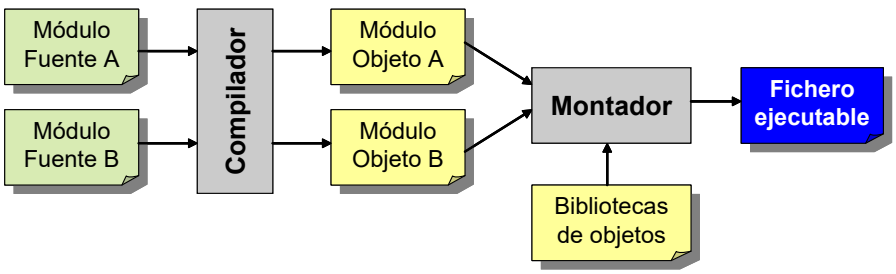


Figura-6.- Etapas en la generación del fichero ejecutable

## 6.2 Formato del archivo ejecutable

Un archivo ejecutable Linux está estructurado en una cabecera y un conjunto de secciones (figura-7). La cabecera contiene información de control que permite interpretar el contenido del ejecutable. Cada ejecutable tiene un conjunto de secciones diferentes, pero como mínimo aparecen tres: código, datos con valor inicial y datos sin inicializar. Esta última sección aparece en la tabla de secciones de la cabecera, pero no se almacena normalmente en el archivo ejecutable ya que su contenido es irrelevante.

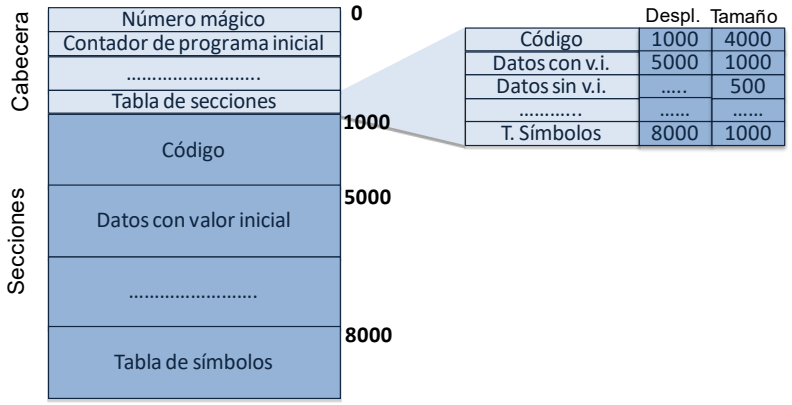


Figura-7.- Formato simplificado de un archivo ejecutable en Linux (ELF).

Cuando se solicita la ejecución de un proceso, se crean varias regiones del mapa a partir de la información del archivo ejecutable, figura-7. Las regiones iniciales del mapa del proceso se corresponden básicamente con las distintas secciones del archivo ejecutable.

## 6.3 Reserva dinámica de memoria en lenguaje C

La reserva dinámica de memoria es aquella que solicita un proceso durante su ejecución. Esto significa que la reserva de memoria se realiza dinámicamente en tiempo de ejecución, no siendo necesario tener que especificar en la declaración de variables la cantidad de memoria que se va a requerir. La reserva de memoria dinámica añade una gran flexibilidad a los programas ya que permite al programador reservar la cantidad de memoria exacta en el preciso instante en que se necesita, sin tener que realizar una reserva por exceso en prevención de que la pueda llegar a necesitar.

Para reservar memoria dinámica en C se utiliza la función ***malloc(),*** que reservar una porción contigua de memoria. Está definida como:

*void *malloc(size_t size);*

malloc devuelve un puntero de tipo void *, con la dirección a partir de la cual se encuentra reservada la porción de memoria de tamaño *size*. Si no puede reservar esa cantidad de memoria la función devuelve un puntero a NULL. Un puntero es una variable que contiene la dirección de otro objeto.

La función malloc devuelve un puntero a *void,* o puntero genérico. El compilador de C requiere hacer una conversión del tipo, mediante un casting. Por ejemplo:

```
char *cp;
cp = (char *) malloc(1024);
```

En este ejemplo se intenta reservar 1024 bytes y la dirección de inicio se almacena en cp. El puntero genérico devuelto por malloc es convertido a un puntero de tipo char mediante el casting "(char *)".

Es usual usar la función sizeof() para indicar el número de bytes. La función sizeof() puede ser usada para encontrar el tamaño de cualquier tipo de dato, variable o estructura, por ejemplo:

```
int *ip;
ip = (int *) malloc(1024 * sizeof(int) );
```

En este ejemplo se esta intentando reservar memoria para 1024 enteros (no bytes).

Cuando se ha terminado de usar una porción de memoria siempre se deberá liberar usando la función free(). Esta función permite que la memoria liberada este disponible nuevamente quizás para otra llamada de la función malloc(). Ejemplo

*free(ip);*

La función free() libera la memoria a la cual el puntero *cp* hace referencia.