

Introducción a los Sistemas Operativos / Conceptos de Sistemas Operativos

Administración de Memoria –Ejemplos



- ✓ Versión: Septiembre 2023
- ✓ Palabras Claves: Procesos, Espacio de Direcciones, Memoria Mapa de Memoria, Paginación, Memoria Virtual, Tablas de Páginas

Referencia: <https://juncotic.com/mapa-de-memoria-de-un-proceso-en-linux/>



Mapa de Memoria

- ✓ Un mapa de memoria, (memory map o memory layout) es una estructura de datos que indica al sistema operativo cómo está distribuida la memoria de un proceso, los segmentos que la componen, y los datos almacenados en cada uno de ellos.
- ✓ Cuando se ejecuta un programa en GNU/Linux, se crea un mapa de memoria en la que se carga variables inicializadas, variables no inicializadas, segmentos de memoria dinámica, la pila, y el código binario de la aplicación (o una parte de el).



Mapa de Memoria

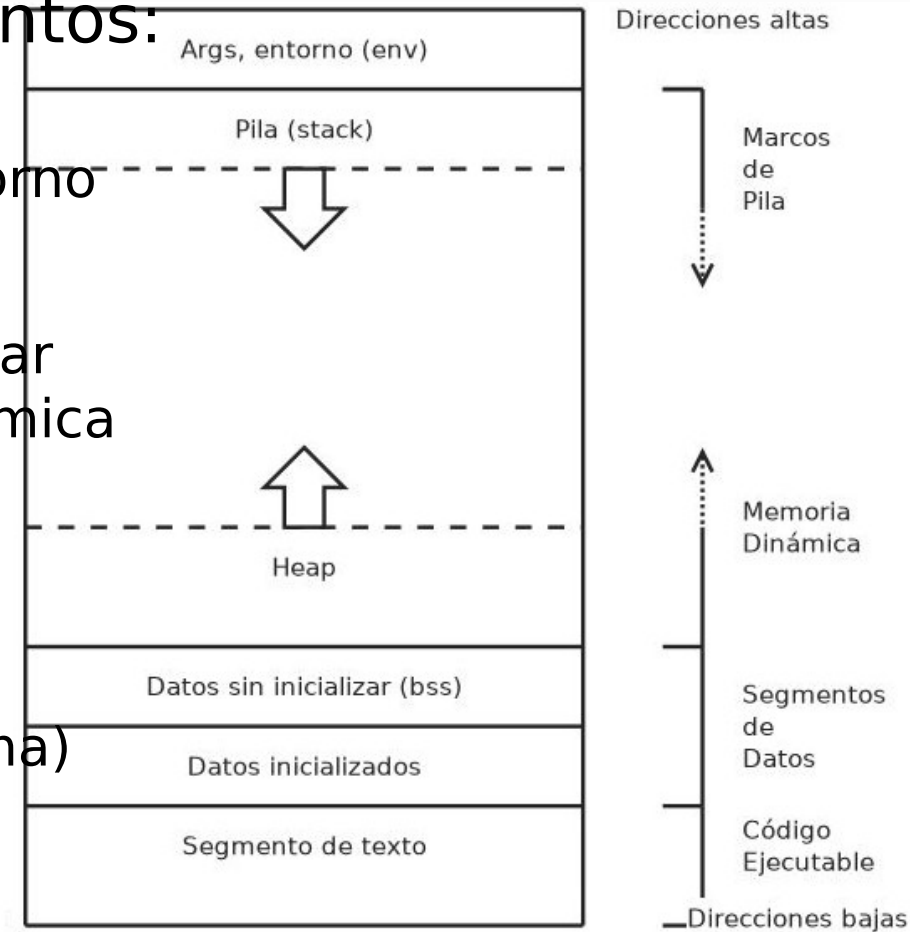
- ✓ GNU/Linux utiliza gestión de memoria basada en segmentación y paginación
- ✓ Las páginas de memoria son de tamaño fijo, por ejemplo, 4 KiB (podemos leer el tamaño de la página con el comando `getconf PAGESIZE`), mientras que la segmentación implica segmentos de tamaño variable.
- ✓ GNU/Linux divide el mapa de memoria de un proceso en segmentos de diferentes tamaños, cada uno dividido, a su vez, en páginas de tamaño fijo.



Mapa de Memoria

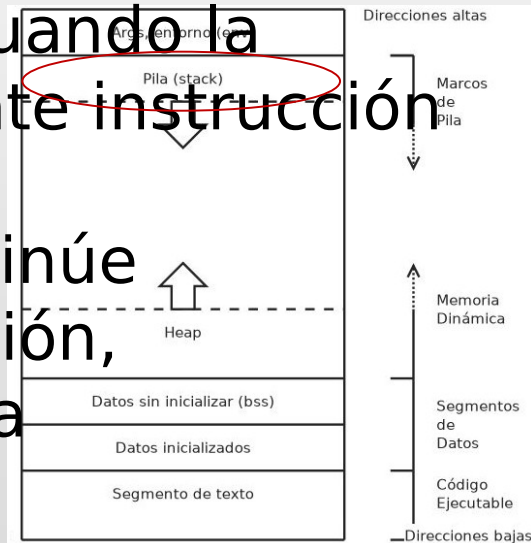
✓ El mapa de memoria de un proceso se divide generalmente en 6 segmentos:

- ✓ Argumentos de la línea de comandos y variables de entorno
- ✓ Stack o pila del proceso.
- ✓ Heap o espacio para almacenar segmentos de memoria dinámica
- ✓ Datos no inicializados (BSS)
- ✓ Datos inicializados
- ✓ Segmento de texto (código binario de un programa)



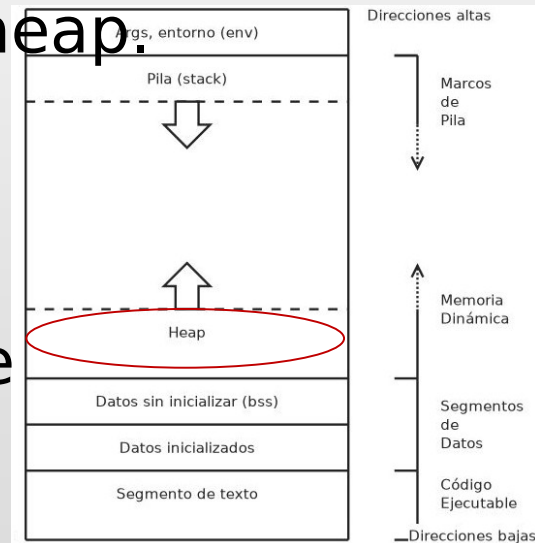
Mapa de Memoria - Stack

- ✓ En este segmento se almacenan las variables locales de las funciones, los argumentos pasados a cada función llamada, y los punteros de retorno de las mismas.
- ✓ Cuando se llama a una función se crea un marco de pila, se almacenan variables locales, argumentos y la dirección de retorno de la función. Cuando la función termina, se carga como siguiente instrucción la dirección de retorno, de modo que la ejecución continúe por donde iba antes de llamar a la función, y se procede a eliminar el marco de pila.



Mapa de Memoria - Heap

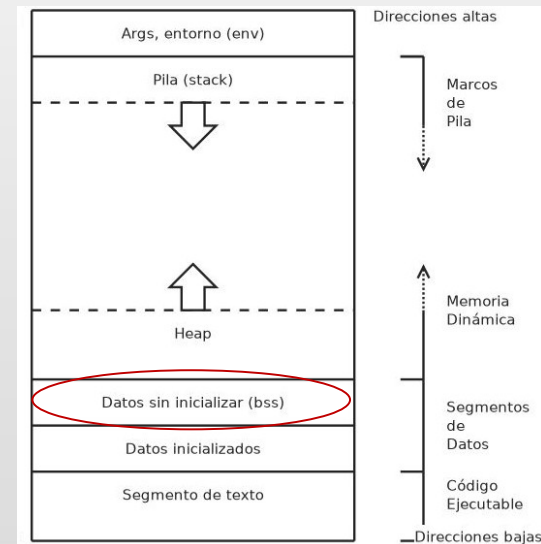
- ✓ En este segmento se almacena la memoria dinámica creada por el proceso (malloc(), calloc(), realloc() por ejemplo).
- ✓ Cuando se reserva una posición para memoria dinámica el programa adquiere dicho espacio desde el heap. Si se libera dicha memoria (free()), se reduce el espacio ocupado dentro del heap.
- ✓ Cuando se libera espacio de memoria dinámica se devuelve al heap, pero no a la memoria del sistema operativo, por lo que el heap puede que comience a fragmentarse.



Mapa de Memoria – Datos sin inicializar

- ✓ Este segmento, también conocido como BSS (heredado de lenguaje ensamblador) almacena todas las variables globales y estáticas que no están inicializadas a cero o no tienen un valor de inicialización en el código fuente

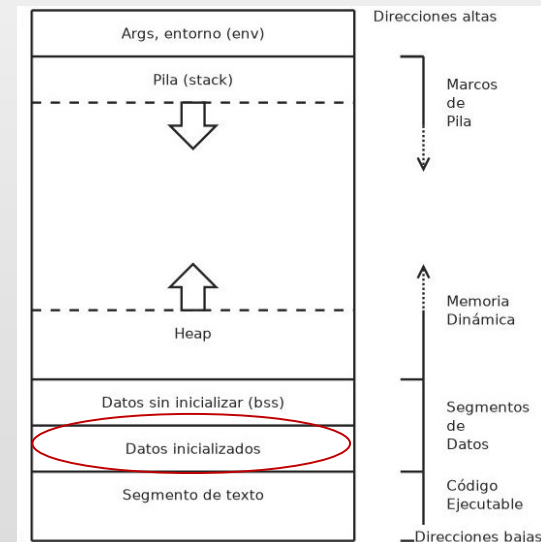
```
1 | int num; //global
2 | static int x=0;
```



Mapa de Memoria – Datos inicializados

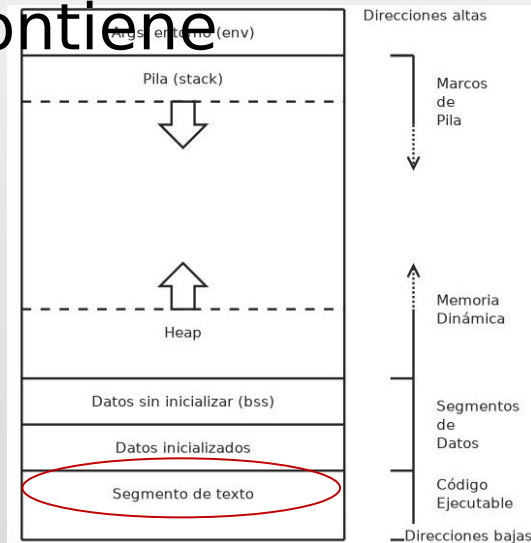
- ✓ Contiene las variables globales y estáticas del programa, que a su vez fueron inicializadas con valores distintos de cero. Este segmento puede clasificarse como de sólo lectura y de lectura-escritura.

```
1 char cadena[] = "Hola Mundo"; //global
2 int contador = 1; //global
3 const int num = 5; //global
```



Mapa de Memoria – Texto

- ✓ Este segmento almacena las instrucciones ejecutables del programa, por lo que también se lo denomina segmento de código.
- ✓ Almacena la representación en código de máquina de las instrucciones del programa. Este segmento en general puede ser compartido entre diferentes procesos, ya que no se modifica, y si contiene código de librerías compartidas no es necesario duplicarlo en memoria
- ✓ Es un segmento de sólo lectura

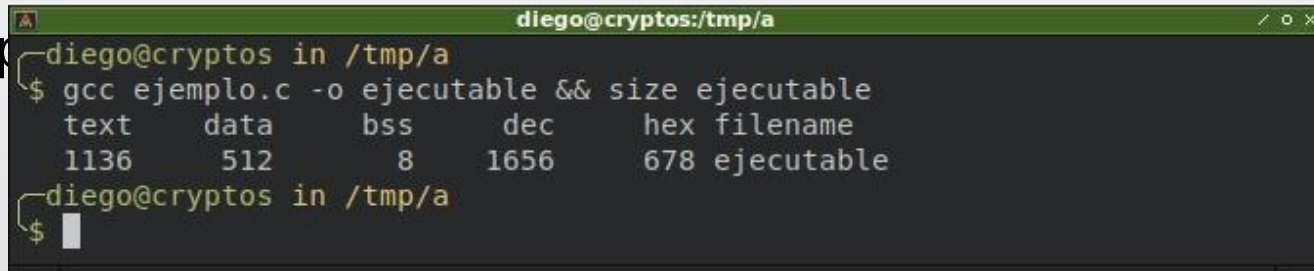


Mapa de Memoria – Ejemplos

- ✓ Para los siguientes ejemplos usaremos el comando `size`, que en su salida muestra el tamaño del segmento de texto, de datos inicializados, y el bss.
- ✓ Tomando como punto de partida el siguiente código fuente de C:

```
1  #include<stdio.h>
2
3  int main(){
4      return 0;
5  }
```

- ✓ Compilamos el código fuente con el comando `gcc` y usamos el comando `size`:



```
diego@cryptos:/tmp/a
$ gcc ejemplo.c -o ejecutable && size ejecutable
text    data    bss      dec     hex filename
1136    512      8      1656    678 ejecutable
diego@cryptos:/tmp/a
$
```

- ✓ Se ve que el segmento de datos contiene 512 bytes, mientras que el bss solamente 8 bytes.



Mapa de Memoria – Ejemplos

- ✓ Para los siguientes ejemplos usaremos el comando `size`, que en su salida muestra el tamaño del segmento de texto, de datos inicializados, y el bss.

- ✓ Creemos ahora, por ejemplo, una variable global, de tipo `double`:

```
1 #include<stdio.h>
2 double numero;
3
4 int main(){
5     return 0;
6 }
```

- ✓ Comprobemos el tamaño de los segmentos de memoria con el comando `size`:

```
diego@cryptos:/tmp/a
$ gcc ejemplo.c -o ejecutable && size ejecutable
text    data    bss     dec     hex filename
1136    512      16    1664    680 ejecutable
diego@cryptos:/tmp/a
$
```

- ✓ Se ve que bss aumentó 8 bytes. Esto es correcto puesto que la variable global no está inicializada, y una variable de tipo `double` en C ocupa 8 bytes en memoria.



Mapa de Memoria – Ejemplos

- ✓ Para los siguientes ejemplos usaremos el comando `size`, que en su salida muestra el tamaño del segmento de texto, de datos inicializados, y el bss.
- ✓ Veamos ahora qué pasa si la inicializamos con un valor distinto

```
1 #include<stdio.h>
2 double numero = 123;
3
4 int main(){
5     return 0;
6 }
```

- ✓ Com `size`:

```
diego@cryptos:~/tmp/a
$ gcc ejemplo.c -o ejecutable && size ejecutable
text    data    bss     dec     hex filename
1136    520      8      1664    680 ejecutable
diego@cryptos:~/tmp/a
$
```

- ✓ Aquí se ve que se redujo en 8 bytes el bss, ya no tenemos esa variable no inicializada. Sin embargo, se incrementó en 8 bytes el segmento de datos inicializados



Mapa de Memoria – pmap

- ✓ Un interesante comando para leer el mapa de memoria de un proceso en ejecución es pmap

```
1 | #include<stdio.h>
2 |
3 | char cadena[8192*10];
4 |
5 | int main(){
6 |     getchar();
7 |     return 0;
8 | }
```

- ✓ Compilamos el código y vemos la salida del comando size:

```
diego@cryptos:/tmp/a
diego@cryptos in /tmp/a
$ gcc ejemplo.c -g -o ejecutable && size ejecutable
   text    data     bss     dec      hex filename
   1279      584    81952    83815   14767 ejecutable
diego@cryptos in /tmp/a
$
```

- ✓ Como puede observarse, el segmento de datos no inicializados (bss) contiene 80 KiB de datos, el arreglo del 80 KiB definido de manera global



Mapa de Memoria – pmap

- ✓ Si ejecutamos el código éste se detendrá en la penúltima línea, getchar(), esperando que presionemos una tecla. Si lo dejamos corriendo y en otra terminal ejecutamos el comando pmap pasando por argumento el PID de nuestro ejecutable, veremos

```
diego@cryptos in /tmp/a
$ pmap -x $(pidof ejecutable)
48216: ./ejecutable
Address      Kbytes      RSS      Dirty Mode  Mapping
00005616523d9000      4         4        4 r---- ejecutable
00005616523da000      4         4        4 r-x-- ejecutable
00005616523db000      4         0        0 r---- ejecutable
00005616523dc000      4         4        4 r---- ejecutable
00005616523dd000      4         4        4 rw--- ejecutable
00005616523de000     80         0        0 rw--- [ anon ]
00005616532a5000    132         4        4 rw--- [ anon ]
00007f1cfb6c4000      8         4        4 rw--- [ anon ]
00007f1cfb6c6000    136        136        0 r---- libc.so.6
00007f1cfb6e8000   1384        672        0 r-x-- libc.so.6
00007f1cfb842000    352         64        0 r---- libc.so.6
00007f1cfb89a000     16         16       16 r---- libc.so.6
00007f1cfb89e000      8         8         8 rw--- libc.so.6
00007f1cfb8a0000     60         28       28 rw--- [ anon ]
00007f1cfb8f3000      4         4         0 r---- ld-linux-x86-64.so.2
00007f1cfb8f4000    152        152        0 r-x-- ld-linux-x86-64.so.2
00007f1cfb91a000     40         40        0 r---- ld-linux-x86-64.so.2
00007f1cfb924000      8         8         8 r---- ld-linux-x86-64.so.2
00007f1cfb926000      8         8         8 rw--- ld-linux-x86-64.so.2
00007ffffe684b000   136         16       16 rw--- [ stack ]
00007ffffe69c4000    16         0         0 r---- [ anon ]
00007ffffe69c8000      8         4         0 r-x-- [ anon ]
fffffffffff600000     4         0         0 --x-- [ anon ]
-----
total kB          2572    1180    108
```

- ✓ Se llama anónima a la memoria mapeada no asociada a archivos en el disco
- ✓ La línea marcada indica que tenemos un mapeo anónimo de 80 KiB.
- ✓ Columnas:
 - ✓ Address: la dirección de inicio de la posición de memoria en cuestión.
 - ✓ Kbytes: tamaño de esa región en KiB.
 - ✓ RSS: Resident set size, parte de la memoria realmente en RAM.
 - ✓ Dirty: estado de las páginas de memoria.
 - ✓ Mode: permisos de acceso a esa región de memoria por parte del proceso.
 - ✓ Mapping: el nombre de la app o librería asociada a esa región de memoria.



Mapa de Memoria – `pmap`

- ✓ Inicialicemos la memoria con un valor, y analicemos la salida de `size` y de `pmap` nuevamente.

```

1  #include<stdio.h>
2
3  char cadena[8192*10] = "hola";
4
5  int main(){
6      getchar();
7      return 0;
8  }
```

- ✓ Compilamos el código y vemos la salida del comando `size`:

```

diego@cryptos:/tmp/a
$ gcc ejemplo.c -g -o ejecutable && size ejecutable
text    data    bss     dec      hex filename
1279    82520      8    83807    1475f ejecutable
diego@cryptos:/tmp/a
$
```

- ✓ El comando `size` nos muestra que los 80 KiB que antes estaban en el segmento `bss` ahora han pasado al segmento de datos inicializados.



Mapa de Memoria – pmap

- ✓ La salida de pmap muestra que ahora el segmento ya no es anónimo, ha pasado a datos inicializados, y se sumó a los 4 KiB

```
diego@cryptos in /tmp/a
$ pmap -x $(pidof ejecutable )
48768:  ./ejecutable
Address      Kbytes      RSS      Dirty Mode  Mapping
000055688033b000      4         4         4 r---- ejecutable
000055688033c000      4         4         4 r-x-- ejecutable
000055688033d000      4         0         0 r---- ejecutable
000055688033e000      4         4         4 r---- ejecutable
000055688033f000     84        64        64 rw--- ejecutable
00005568818b6000    132         4         4 rw--- [ anon ]
00007fa150bad000      8         4         4 rw--- [ anon ]
00007fa150baf000    136       132         0 r---- libc.so.6
00007fa150bd1000   1384       700         0 r-x-- libc.so.6
00007fa150d2b000    352         64         0 r---- libc.so.6
00007fa150d83000     16         16        16 r---- libc.so.6
00007fa150d87000      8         8         8 rw--- libc.so.6
00007fa150d89000     60        28        28 rw--- [ anon ]
00007fa150ddc000      4         4         0 r---- ld-linux-x86-64.so.2
00007fa150ddd000    152       152         0 r-x-- ld-linux-x86-64.so.2
00007fa150e03000     40        40         0 r---- ld-linux-x86-64.so.2
00007fa150e0d000      8         8         8 r---- ld-linux-x86-64.so.2
00007fa150e0f000      8         8         8 rw--- ld-linux-x86-64.so.2
00007ffc9e5ff000    136        20        20 rw--- [ stack ]
00007ffc9e658000     16         0         0 r---- [ anon ]
00007ffc9e65c000      8         4         0 r-x-- [ anon ]
fffffffffff60000      4         0         0 --x-- [ anon ]
-----
total kB          2572       1268       172
```

✓ Columnas:

- ✓ Address: la dirección de inicio de la posición de memoria en cuestión.
- ✓ Kbytes: tamaño de esa región en KiB.
- ✓ RSS: Resident set size, parte de la memoria realmente en RAM.
- ✓ Dirty: estado de las páginas de memoria.
- ✓ Mode: permisos de acceso a esa región de memoria por parte del proceso.
- ✓ Mapping: el nombre de la app o librería asociada a esa región de memoria.

