Práctico 2 SO - Virtualización de la memoria

Ejercicio 1. Para cada una de las variables de este código indicar si están en el segmento de código, de pila o de montículo (heap). Si hay punteros indicar a que segmento apunta.

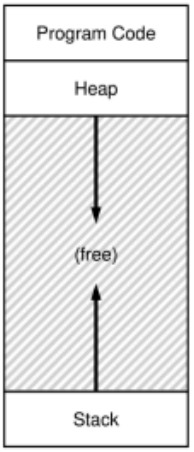
Extra:¿Dónde se ubica el arreglo global si lo declaramos inicializado a cero? `int a[N] = {0};`

```c
#include <stdlib.h>
#define N 1024

int a[N];

int main(int argc, char ** argv)
{
  int i;
  register int s = 0;
  int *b = calloc(N, sizeof(int));
  for (i=0; i<N; ++i)
    s += a[i]+b[i];

  free(b);
  return s;
}
```



Definamos primero, los segmentos mencionados para darnos una idea de dónde podrían estar esas variables.

- **Stack:** área de memoria utilizada para almacenar datos de forma estructurada en forma de marcos de pila. Cada función llamada crea un nuevo marco en la pila, que almacena variables locales y la dirección de retorno de la función.
- **Heap:** área de memoria utilizada para almacenar datos dinámicos durante la ejecución de un programa. Estos son datos que no tienen un tamaño fijo y cuya memoria se asigna y libera en tiempo de ejecución.
- Código: Esta área de memoria contiene el código ejecutable del programa. Es la parte del programa que se compila a instrucciones de máquina y se ejecuta cuando el programa se inicia.
- **.data:** Esta sección contiene datos estáticos inicializados que tienen un tamaño y ubicación fija en el programa. Los datos en esta sección pueden ser modificados durante la ejecución, pero su tamaño y ubicación inicial están definidos en tiempo de compilación. Se utiliza, por ejemplo, para almacenar variables globales y estáticas que deben conservar su valor entre llamadas a funciones o a lo largo de la vida del programa.

int A[N] → se encuentra en .data, ya que es una variable de tamaño fijo y estática.
int i → se encuentra en heap, el scope de la variable es main.
register int s → depende del compilador, puede estar en la memoria de registros o en el Stack.
int *b → está en Stack y en Heap, “b*” es la dirección de memoria en donde se apunta al Heap, donde está el valor entero.

Ejercicio 2. Debuggear el mal uso de memoria en los siguientes pedacitos de código.

```c
char *s = malloc(512);
gets(s);
```

```c
char *s = "Hello Waldo";
char *d = malloc(strlen(s));
strcpy(d,s);
```

```c
char *s = "Hello Waldo";
char *d = malloc(strlen(s));
d = strdup(s);
```

```c
int * a = malloc(16)
a[15] = 42;
```

char *s = malloc(512) → asigna 512 bytes en memoria dinámica.
gets(s) → se utiliza para leer una cadena de caracteres desde la entrada estándar (generalmente el teclado) y almacenarla en el array de caracteres que se pasa como argumento.

- El problema en este primer programa, es que gets(s) no tiene límite de entrada, por lo que se podría leer más elementos de los que contiene “s”. En otras palabras, no es segura y propensa a desbordamientos de búfer, por lo que su uso se desaconseja en la programación moderna debido a problemas de seguridad.

char *s = “Hello World” → se encuentra en .data
char *d = malloc(strlen(s)) → falta un elemento en el malloc debido al “/0”.
strcpy(d, s)

- La solución es “char *d = malloc(strlen(s))”.

char *s = "Hello Waldo"; → se encuentra en .data
char *d = malloc(strlen(s)); → este malloc se pierde, debido a que strdup ya mallocquea.
d = strdup(s);

- La solución es reemplazar la línea dos por char *d = strdup(s), que duplica el string pasado como parámetro.

int * a = malloc(16); → arreglo de 4 elementos debido a que cada int ocupa 4 bytes.
a[15] = 42;

- No se puede indexar a[15] debido a que se va de los límites.

Ejercicio 3. Verdadero o falso. Explique.

(a) `malloc()` es una syscall.
(b) `malloc()` siempre llama a una syscall.
(c) `malloc()` a veces produce una llamada a una syscall.
(d) Idem con `free()`.
(e) El tiempo de cómputo que toma `malloc(x)` es proporcional a `x`.

a. Falso, malloc() no es una syscall, es una función de C.
b. Falso, aunque malloc() en sí mismo no es una syscall, sí puede implicar llamadas al sistema operativo, ya sea directamente o a través de las funciones internas de administración de memoria de la biblioteca de C utilizadas por malloc().
c. Verdadero, por lo antes mencionado.
d. Verdadero.
e. Falso, no es directamente proporcional a x, donde x es el tamaño de la memoria que se está solicitando. En cambio, el tiempo de cómputo de malloc() depende de diversos factores, como la implementación específica de malloc() en la biblioteca estándar del compilador, el estado del sistema, la disponibilidad de memoria y otros factores del entorno de ejecución.

Ejercicio 4. Mostrar la secuencia de accesos a la memoria física que se produ
programa assembler `x86_32`, donde el registro `base=4096` y `bounds=256`.

```
 0: movl $128,%ebx
 5: movl (%ebx),%eax
 8: shll $1, %ebx
10: movl (%ebx),%eax
13: retq
```

0: movl $128,%ebx → ebx = 128
5: movl (%ebx),%eax → eax = &[ebx]
8: shll $1, %ebx → eax * 2
10: movl (%ebx),%eax → eax = &[ebx]
13: retq

Práctico 2 SO - Virtualización de la memoria

Traza de memoria de la ejecución del programa(virtual): 0, 5, 128, 8, 10, 256, 13.
Para pasar a memoria física, basta con sumar cada dirección de memoria virtual a la base, que en este caso es 4096. Los límites de memoria para este programa, están especificados en bounds, y si nos pasamos de tal dirección, se produce un segfault.
Traza de memoria de la ejecución del programa (física): 4096 + 0, 4096 + 5, 4096 + 128, 4096 + 8, 4096 + 10, 4096 + 256, 4096 + 13.

Ejercicio 5. Mostrar con un ejemplo de disposición de memoria física de varios procesos como el esquema de traducción de direcciones con base+límite puede producir **fragmentación interna** y **fragmentación externa**.

Definamos primero los conceptos involucrados en este ejercicio.

- **Fragmentación Interna:** Ocurre cuando hay espacio no utilizado dentro de un bloque de memoria asignado a un proceso. Esto puede deberse a que los bloques de memoria asignados no son del tamaño exacto requerido por el proceso.
- **Fragmentación Externa:** Ocurre cuando hay suficiente memoria total para satisfacer una solicitud de asignación de memoria, pero la memoria está dispersa en bloques pequeños y no contiguos. Como resultado, no se puede asignar un bloque de memoria contiguo lo suficientemente grande para satisfacer la solicitud, aunque la memoria total disponible es suficiente.

Segmentación Interna: consideremos un escenario con tres procesos (A, B y C), con tamaños de memoria requeridos de 200, 300 y 150 KB respectivamente.
Supongamos que tenemos una memoria física de 1 MB (1024 KB) y usamos el esquema de traducción de direcciones con base + límite.

- **Proceso A:**
  → Tamaño requerido de 200 KB
  → Se asigna memoria física contigua de 200KB, la dirección base es 0 y la dirección límite es 199.

- **Proceso B:**
  → Tamaño requerido de 300 KB.
  → No hay bloques contiguos de 300 KB, pero si de 200 y 100 KB.
  → Se le asigna un bloque de 200 KB para satisfacer la solicitud y otro de 100KB.
- **Proceso C:**
  → Tamaño requerido de 150 KB.
  → Se le asigna dos bloques de 100 KB para satisfacer la solicitud.
  → **Genera segmentación interna ya que no ocupa la totalidad de un bloque de 100 KB.**

Segmentación externa: supongamos que tenemos una memoria física de 1000 KB y tres procesos: A, B y C, con tamaños de memoria requeridos de 400 KB, 300 KB y 200 KB respectivamente.

- **Proceso A:**
  → Tamaño requerido: 400 KB
  → Se asigna un bloque contiguo de 400 KB en las direcciones de memoria 0 a 399.
- **Proceso B:**
  → Tamaño requerido: 300 KB
  → Se asigna un bloque contiguo de 300 KB en las direcciones de memoria 400 a 699.
- **Proceso C:**
  → Tamaño requerido: 200 KB

→ No hay un bloque contiguo de 200 KB disponible en la memoria, aunque hay espacio libre en total. Por ejemplo, hay 100 KB de espacio libre entre los bloques asignados a A y B y otros 300 KB de espacio libre después del bloque asignado a B.
→ **Debido a la falta de bloques de 200 KB contiguos, el proceso C no puede ser asignado en la memoria, aunque hay suficiente memoria total para satisfacer la solicitud, produciendo segmentación externa.**

Ejercicio 6. Distinguir **relocalización estática** y **relocalización dinámica**.

La **relocalización** se refiere a la capacidad de un sistema operativo para cargar programas y datos en diferentes ubicaciones de memoria física. Tanto la relocalización dinámica como la estática están relacionadas con este proceso, pero difieren en cómo y cuándo se realiza la asignación de direcciones.
Relocalización Estática: en la relocalización estática, las direcciones de memoria de un programa se asignan en tiempo de compilación y no cambian durante la ejecución del programa. Esto significa que las direcciones específicas de memoria se escriben directamente en el código del programa durante la fase de compilación.
Si varios programas se ejecutan simultáneamente y están vinculados a direcciones específicas de memoria, podrían haber conflictos si intentan acceder a las mismas direcciones, ya que las direcciones son fijas y no se pueden cambiar durante la ejecución. La relocalización estática no es tan flexible y no permite una gestión dinámica de la memoria.
Relocalización Dinámica: en la relocalización dinámica, las direcciones de memoria de un programa se asignan durante la carga o ejecución del programa en memoria. Las direcciones específicas de memoria se determinan y se ajustan en tiempo de ejecución, según la disponibilidad de memoria en ese momento.
Esto permite una mayor flexibilidad, ya que los programas pueden cargarse en diferentes ubicaciones de memoria cada vez que se ejecutan, lo que ayuda a evitar conflictos de memoria cuando varios programas se ejecutan simultáneamente. La relocalización dinámica es especialmente útil en sistemas multitarea, donde varios programas pueden ejecutarse al mismo tiempo y necesitan compartir la memoria de manera eficiente.

Ejercicio 7. Verdadero o falso. Explique.

(a) Modificar los registros `base` y `bounds` son instrucciones privilegiadas.

(b) Hay un juego de registros `(base, bounds)` por cada proceso.

a. Verdadero, solo el sistema operativo o el núcleo del sistema operativo tienen el privilegio de modificar estos registros para evitar que los programas de usuario accedan a regiones de memoria fuera de sus límites asignados.
b. Falso, hay solo un par de registros (base, bounds) físicos por core. Lo que se guarda por proceso son los valores de los mismos.
   En los sistemas de segmentación de hardware, hay registros físicos por cada núcleo del procesador para mantener la información de segmentación, como la dirección base y el límite (bounds) para cada segmento. Sin embargo, cuando se cambia de un proceso a otro, los valores de estos registros se guardan y restauran adecuadamente para asegurar que cada proceso tenga su propio espacio de memoria protegido.
   Por lo tanto, aunque solo hay un conjunto de registros físicos (base, bounds) por cada núcleo del procesador, estos registros se utilizan de manera dinámica para mantener los valores específicos del proceso que está siendo ejecutado

actualmente. De esta manera, cada proceso puede tener su propio espacio de memoria protegido, lo que proporciona aislamiento entre los procesos.

Ejercicio 8. Una computadora proporciona a cada proceso 65536 bytes de espacio de direcciones dividido en páginas de 4 KiB. Un programa específico tiene el segmento código de 32768 bytes de longitud, el segmento montículo de 16386 bytes de longitud, y un segmento pila de 15870 bytes. ¿Cabrá el programa en el espacio de direcciones? ¿Y si el tamaño de página fuera de 512 bytes? Recuerde que una página no puede contener segmentos de distintos tipos asi se pueden proteger cada uno de manera adecuada.

Tenemos páginas de 4 KiB (no es lo mismo que KB) → 4096 bytes.
Páginas totales → 65536 / 4096 = 16.

- Páginas del segmento code: 32768 / 4096 = 8.
- Páginas del segmento heap: 16386 / 4096 = 5.
- Páginas del segmento stack: 15870 / 4096 = 4.

Son necesarias 17 páginas (8 + 5 + 4), y teníamos disponibles 16 páginas, por lo que el programa no entra en el espacio de direcciones.

Teniendo páginas de 512 bytes, el calculo seria el siguiente:
Páginas totales → 32768 / 512 = 128.

- Páginas del segmento code: 32768 / 512 = 64.
- Páginas del segmento heap: 16386 / 512 = 33.
- Páginas del segmento stack: 15870 / 512 = 31.

Son necesarias 128 páginas (64 + 33 + 31), y teníamos disponibles 128, por lo que el programa entra en el espacio de direcciones.

Paginación

Ejercicio 10. La TLB de una computadora con una pagetable de un nivel tiene una eficiencia del 95 %. Obtener un valor de la TLB toma 10ns. La memoria principal tarda 120ns. ¿Cuál es el tiempo promedio para completar una operación de memoria teniendo en cuenta que se usa tabla de páginas lineal?

- Eficiencia del 95% → Page table Hit = 0.95
- La probabilidad de que la obtención de un valor de la TLB sea de 10 ns es 0.95.
- El resto de los casos, tarda 120 ns.

0.95 * (Page table Hit) + 0.5 (-Page table Hit) = 0.95 * 10 ns + 0.05 * 120 ns = 15.50 ns

Respuesta final: el tiempo promedio para completar una operación es de 15.50 ns.

Ejercicio 11. Considere el siguiente programa que ejecuta en un microprocesador con soporte de paginación, páginas de 4 KiB y una TLB de 64 entradas.

```
int x[N];
int step = M;
for (int i=0; i<N; i+=step)
  x[i] = x[i]+1;
```

(a) ¿Qué valores de `N`, `M` hacen que la TLB falle en cada iteración del ciclo?
(b) ¿Cambia en algo si el ciclo ser repite muchas veces? Explique.

Práctico 2 SO - Virtualización de la memoria

Ejercicio 12. Dado un tamaño de página de 4 KiB = 2^{12} bytes y la tabla de paginado de la Fig. 1.

(a) ¿Cuántos bits de direccionamiento hay para cada espacio?

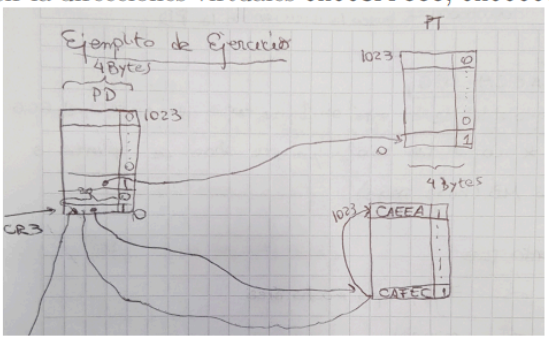
(b) Determine las direcciones físicas a partir de las virtuales: 39424, 12416, 26112, 63008, 21760, 32512, 43008, 36096, 7424, 4032.

(c) Determine el mapeo inverso, o sea las direcciones virtuales a partir de las direcciones físicas: 16385, 4321.

V	F	¿Válida?
0	000	1
1	111	1
2	000	0
3	101	1
4	100	1
5	001	1
6	000	0
7	000	0
8	011	1
9	110	1
10	100	1
11	000	0
12	000	0
13	000	0
14	000	0
15	010	1

Figura 1:

Ejercicio 13. Dado el siguiente esquema de paginación i386 (10, 10, 12) traducir la direcciones virtuales `0x003FF666`, `0x00000AB0`, `0x00800B0B` a físicas.



12.

a. Páginas de 2^12 bytes, luego:
→ Bits de direccionamiento virtual = bits de offset + bits de páginas virtuales.
→ bits de offset = 12.
→ bits de páginas virtuales (16) = 2^4 = 4.
→ **Bits de direccionamiento virtual** = 16 bits.
→ Bits de direccionamiento físico = bits de offset + bits de páginas físicas.
→ bits de offset = 12.
→ bits de páginas físicas = 3.
→ **Bits de direccionamiento físico** = 15 bits.

b. 0d39424 →0b1001 1010 0000 0000 → DF = 0b110101000000000
0d12416 →0b0011 0000 1000 0000 → DF = 0b101000010000000
0d26112 →0b0110 0110 0000 0000 → Ausente, no se puede acceder.
…
Análogos
…

c. 0d16385 →0b100 0000 0000 0001 → DV = 01000000 0000 0001 , DV = 1010000000000001
0d4321 →0b001 0000 1110 0001 → DV = 1010000 1110 0001

13. Esquema (10, 10, 12) → PD = 10 bits, PT 10 bits ,OFF = 12 bits.

0x003FF666: 0000 0000 0011 1111 1111 0110 0110 0110
→PD = 0, PT = 1023, OFF = 0x666
→DV = 0xCAEEA666

0x00000AB0: 0000 0000 0000 0000 0000 1010 1011 0000
→ PD = 0, PT = 0, OFF = 0xAB0
→DV = 0xCAFECAB0

0x00800B0B: 0000 0000 1000 0000 0000 1011 0000 1011
→PD = 2, PT = 0, OFF = 0xB0B
→ DV = 0xCAEEBB0B

Ejercicio 14. Dado el sistema de paginado de dos niveles del i386 direcciones virtuales de 32 bits, direcciones físicas de 32 bits, 10 bits de índice de *page directory*, 10 bits de índice de *table directory*, y 12 bits de *offset* dentro de la página, o sea un $(10, 10, 12)$, indicar:

(a) Tamaño de total ocupado por el directorio y las tablas de página para mapear 32 MiB al principio de la memoria virtual.

(b) Tamaño total del directorio y tablas de páginas si están mapeados los 4 GiB de memoria.

(c) Dado el ejercicio anterior ¿Ocuparía menos o más memoria si fuese una tabla de un solo nivel? Explicar.

(d) Mostrar el directorio y las tablas de página para el siguiente mapeo de virtual a física:

Virtual	Física
$[0MiB, 4MiB)$	$[0MiB, 4MiB)$
$[8MiB, 8MiB + 32KiB)$	$[128MiB, 128MiB + 32KiB)$

a. Primero obtenemos la cantidad de páginas físicas que necesitamos para mapear los 32 MiB.
→2^25 / 2^12 = 2^13 páginas físicas.
→Cada PT tiene 1024 entradas ⇒ 8192 / 1024 ⇒ 8 PT
→PD ocupa 4 KiB más.
→ (8 PT + 1 PD) * 4 KiB
→**36 KiB**

b. Mismo proceso:
→2^32 / 2^12 = 2^20 páginas físicas.
→Cada PT tiene 1024 entradas ⇒ 1.048.576 / 1024 ⇒ 1024 PT.
→PD ocupa 4 KiB más.
→ (1024 PT + 1 PD) * 4 KiB.
→**4100 KiB.**
→**4 KiB + 4 MiB.**

c. Ocupa menos, debido a que nos ahorramos los 4 KiB del PD. Osea, en una estructura de un nivel, la estructura de datos ocuparía 4 MiB.

Ejercicio 15. Explique porque un i386 no puede mapear los 4 GiB completos de memoria virtual. ¿Cuál es el máximo?

Los metadatos ocupan lugar, por ende nunca vamos a tener mapeados los 4 GiB completos de memoria virtual, siempre vamos a ocupar memoria para saber dónde están los datos.

Ejercicio 16. Explique como podría extender el esquema de memoria virtual del i386 para que, aunque cada proceso tenga acceso a 4 GiB de memoria virtual (32 bits), en total se puedan utilizar 64 GiB (36 bits) de memoria física[1].

Se hace con PAE, el esquema sería ahora de (2, 9, 9, 12)

Ejercicio 17. ¿Verdadero o Falso? Explique.

(a) Hay una *page table* por cada proceso.
(b) La MMU siempre mapea una memoria virtual más grande a una memoria física más pequeña.
(c) La dirección física siempre la entrega la TLB.
(d) Dos páginas virtuales de un mismo proceso se pueden mapear a la misma página física.

[1]Esto se conoce como *page address extension* – PAE.

(e) Dos páginas físicas de un mismo proceso se pueden mapear a la misma página virtual.
(f) En procesadores de 32 bits y gracias a la memoria virtual, cada proceso tiene 2^{32} direcciones de memoria.
(g) La memoria virtual se puede usar para ahorrar memoria.
(h) Toda la memoria virtual tiene que estar mapeada a memoria física.
(i) El *page directory* en i386 se comparte entre todos los procesos.
(j) Puede haber marcos de memoria física que no tienen un marco de memoria virtual que los apunte.
(k) Por culpa de la memoria virtual hacer un `fork` resulta muy caro en términos de memoria.
(l) Los procesadores tienen instrucciones especiales para acceder a la memoria física evitando la MMU.
(m) Es imposible hacer el mapeo inverso de física a vitrual.
(n) No se puede meter un todo un Sistema Operativo completo con memoria paginada i386 en 4 KiB.

a. Verdadero, puesto que surge como solución al problema de fragmentación externa.
b. Falso, se puede mapear en todos los tamaños respectivamente.
c. Verdadero.
d. Verdadero.
e. Falso, cada dirección virtual tiene una sola dirección física sino genera ambigüedad.
f. Verdadero.
g. Verdadero.
h. Es falso, ya que hay partes de la memoria virtual marcadas como ausente.
i. Falso, cada proceso tiene su propio directorio de páginas.
j. Verdadero.
k. Es falso, gracias a la memoria virtual, hacer fork() es sumamente barato en términos de memoria.
l. Falso.
m. Falso.
n. Falso.

Ejercicio 18. Se define un *page directory* donde la última entrada, la 1023, apunta a la base del mismo[2].

(a) ¿A dónde apunta la dirección virtual `0xFFC00000`?
(b) ¿Y la dirección virtual `0xFFFE0000`?
(c) Indique a donde apunta la dirección virtual `0xFFFFF000`.
(d) Finalmente, describa para que sirve este esquema de memoria virtual.

a. 0xFFC00000 = 1111 1111 1100 0000 0000 0000 0000 0000
→PD = 1023, PT = 0, OFF = 0
→Esta dirección virtual apunta a la base del PD, en la PT 0.
b. 0xFFFE0000 = 1111 1111 1111 1110 0000 0000 0000 0000
→PD = 1023, PT = 0x3E0, OFF = 0
→Esta dirección virtual apunta a la base del PD, en la PT = 0x3E0.
c. 0xFFFFF000 = 1111 1111 1111 1111 1111 0000 0000 0000

→PD = 1023, PT = 0x3FF, OFF = 0
→Apunta a la base del PD.