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.

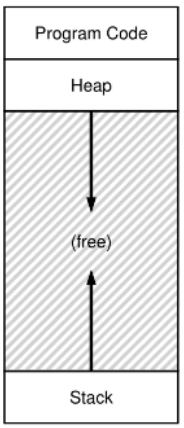
Extra: ¿Dónde se ubica el arreglo global si lo declaramos inicializado a cero? `int a[N] = {0};`

```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;
}
```



N program code 1024
a[N] segmento particular que está entre el program code y el heap. El SO inicializa siempre en '0' este tipo de variable
i stack
s register
*b heap
b stack
argv stack
argc stack
return s stack

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;
```

1- man gets nos dice que nunca lo usemos pues toma cualquier input sin importar el buffer. Por lo cual un mal usuario puede generar problemas.
2- el malloc se está olvidando del +1 para el signo '\0'
3- se está alocando memoria dos veces pues strdup ya se encarga de eso
4- Uso poco formal de malloc. Notemos que los int suelen ser de 4 bytes o de 8 bytes aunque está mal asumir lo que ocupan, en lugar de eso usar sizeof(int). En este caso posiblemente haya memoria para llegar a a[3], debería haberse hecho int *a = malloc(sizeof(int)*16).

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, es una función de una librería.
b) Falso, en caso de que ya exista espacio disponible no genera ningún syscall.
c) Verdadero, puede llamar a brk para ampliar el break del heap.
d) Verdadero.
e) Falso, solo reserva verdaderamente memoria cuando escribo sobre la memoria reservada y solo la necesaria.

Ejercicio 4. Mostrar la secuencia de accesos a la memoria física que se produce al ejecutar este 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
```

movl	0	
movl	5	128
shll	8	
movl	10	256
retq	13	

0; 5; 128; 8; 10; **256**; 13 serían los accesos a memoria
donde con un bound de 256 daría segmentation fault en el 256 en negrita.

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.

NO

Ejercicio 6.
Distinguir relocalización estática y relocalización dinámica.

NO

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.

NO

Segmentación

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.

Manejo del Espacio Libre

Ejercicio 9. Suponga un sistema de memoria contiguo con la siguiente secuencia de tamaños de huecos: 10 KiB, 4 KiB, 20 KiB, 18 KiB, 7 KiB, 9 KiB, 12 KiB, 15 KiB.
Para la siguiente **secuencia de solicitudes** de segmentos de memoria: 12 KiB, 10 KiB, 9 KiB.
¿Cuáles huecos se toman para las distintas políticas?

(a) Primer ajuste (*first fit*).
(b) Mejor ajuste (*best fit*).
(c) Peor ajuste (*worst fit*).
(d) Siguiente ajuste (*next fit*).

Secuencia libre: 10Kib; 4KiB; 20 KiB; 18KiB; 7KiB; 9KiB; 12KiB; 15KiB

1 2 3 4 5 6 7 8

Solicitudes: 12 KiB, 10KiB; 9KiB

a) 12KiB va a 3; 10KiB va a 1; 9KiB va a 4
b) 12KiB va a 7; 10KiB va a 1; 9 KiB va a 6
c) 12KiB va a 3; 10KiB va a 4; 9KiB va a 8
d) 12KiB va a 3; 10 KiB va a 4; 9KiB va a 6

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?

TLB HIT (95%):

10 + 120 = 130

TLB MISS (5%):
10 + 120 + 120 = 250

Average time:
130 * 0.95 + 250 * 0.05 =136 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.

TLB 0...63
Pages 4KiB = 2^{12} = 4096 bytes
int x[N]; osea ocupan 4 bytes => cada Page puede mantener 4096/4 elementos, osea 1024.
El TLB puede almacenar 64 Pages, por lo tanto 64 * 4096 =262144 bytes que son 64 * 1024 = 65536 elementos del arreglo.

a) Para que TLB falle M debería ser mayor o igual a 1024, lo que causaría que cada iteración del loop caiga en una Page distinta. Para N causaría TLB miss si N>65536 por lo que excedería la capacidad del TLB y darían constantes TLB misses.
b) Sí, si el ciclo se sigue repitiendo los TLB misses se irán cacheando a medida que vayan ocurriendo. Si N > 65536 no sucedería lo mismo porque se excedería la capacidad de TLB.

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:

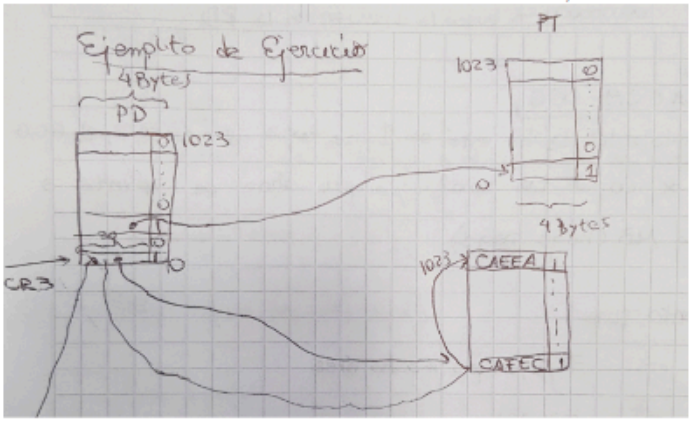
Chequear todo esto…

a) Hay 4 bits de direccionamiento. 0000 = 0 .. 1111 = 15

b) 39424 = (1001 101000000000) base 2, dirección 9 => la dirección física es 0110 101000000000 = 27136
   26112 = 0x6600. Dirección 6, no válida page fault
   63008 = 0xF620. Dirección 15 => 0x2620
   21760 = 0x5500. Dirección 5 => 0x1500
   32512 = 0x7F00. Dirección 7, no válida page fault
   43008 = 0xA800. Dirección 10 => 0x4800
   36096 = 0x8D00. Dirección 8 => 0x3D00
   7424 = 0x1D00. Dirección 1 => 0x7D00
   4032 = 0x0FC0. Dirección 0 => 0x0FC0

c) 16385 = 0x4001. 0x4 = 0100 => 4 y 10 => 0x4001 y 0xA001
   4321 = 0x10E1. 0x1 = 0001 => 5 => 0x50E1

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



0x003ff666 = 0000 0000 00 11 1111 1111 0110 0110 0110

PD = 0 PGE = 1023

se traduce en 0xCAEEA666

0x00000AB0 = 0..0 0..0 00 00 0..0 0..0 1010 1011 0..0

PD = 0 PGE = 0

0xCAFECAB0

0x00800B0B = 0000 0000 10 00 0000 0000 1011 0000 1011

PD = 2 PTE = 0

0x??

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)

PTsize = 4KiB

PDE = 1023 entradas

PTE = 1023 entradas

Para mapear 32MiB que son 2^{25} divido por el pagesize $2^{25}/2^{12}$=8192 páginas, como cada PT apunta a 1024 entradas se necesitan 8 PT. Por lo tanto el total ocupado es de

4KiB + 8 * 4KiB = 36 KiB

b)

Mismo procedimiento $2^{32}/2^{12}=2^{20}$=1048576 paginas

como c/ page table apunta a 1024 paginas fisicas se necesitan 1024 page tables para mapear 4GiB. entonces el total ocupado por el directorio y las tablas de páginas es :

4KiB + 1024 * 4KiB = 4KiB + 4MiB = $2^{12} + 2^{22}$ = 4198400 bytes = 4100 KiB aprox 4MiB

c)

Si fuese una tabla de un solo nivel, se necesitaría menos espacio pues no se estarían tomando en cuenta los 4KiB de el PD pero sería menos eficiente.

d)

Notar que la primera de 4MiB y que una PT mapea a 1024 * 4KiB = 4MiB por lo tanto solo hace falta una página virtual para mapear la física. En el segundo caso sólo se mapean 8 entradas válidas para los 32KiB

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

1024 * 1024 * 4KiB = 4GiB pero hay espacio reservado para el OS y para las mismas PT enontonces no puede mapear a ese espacio restringido

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].

Como siempre es el caso eso se resuelve con más indirección, se necesitarán 3 niveles de tablas. **RECHEQUEAR**

Con PAE, el proceso de traducción de direcciones funciona de la siguiente manera:

1. El procesador toma una **dirección virtual de 32 bits**.
2. Utiliza los bits más altos de la dirección virtual para indexar en el **Page Directory Pointer Table (PDPT)**.
3. Luego, con otros bits de la dirección virtual, accede al **page directory** y a la **tabla de páginas**.
4. Finalmente, el procesador obtiene una dirección **física de 36 bits** que corresponde a la página en la memoria física.

Resumen:

Para extender el esquema de memoria virtual del i386 para que el sistema pueda utilizar **64 GiB** de memoria física (36 bits), mientras cada proceso tiene un espacio de direcciones virtuales de 4 GiB, se implementa **PAE** (Physical Address Extension). Esta extensión permite al sistema direccionar más memoria física agregando un tercer nivel de paginación y aumentando el tamaño de las entradas en las tablas de páginas, sin cambiar el tamaño de las direcciones virtuales de los procesos individuales.

Ejercicio 17. ¿Verdadero o Falso? Explique.

(a) Hay una page table por cada proceso. V

(b) La MMU siempre mapea una memoria virtual más grande a una memoria física más pequeña. F

(c) La dirección física siempre la entrega la TLB. F

(d) Dos páginas virtuales de un mismo proceso se pueden mapear a la misma página física. V

(e) Dos páginas físicas de un mismo proceso se pueden mapear a la misma página virtual. F

(f) En procesadores de 32 bits y gracias a la memoria virtual, cada proceso tiene 2^{32} direcciones de memoria. VERFA si bien se pueden direccionar 4GiB, no todos están disponibles.

(g) La memoria virtual se puede usar para ahorrar memoria. V

(h) Toda la memoria virtual tiene que estar mapeada a memoria física. F

(i) El page directory en i386 se comparte entre todos los procesos. F

(j) Puede haber marcos de memoria física que no tienen un marco de memoria virtual que los apunte. V

(k) Por culpa de la memoria virtual hacer un fork resulta muy caro en términos de memoria. F

(l) Los procesadores tienen instrucciones especiales para acceder a la memoria física evitando la MMU. V

(m) Es imposible hacer el mapeo inverso de física a virtual. F

(n) No se puede meter un todo un Sistema Operativo completo con memoria paginada i386 en 4 KiB. F

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) 0xFFC00 000 = 1111 1111 11 00 0000 0000 0000 0000 0000 => PD apunta a 1023 que es su propia base y luego va a la dirección 0 del PD
b) 0xFFFE0000 = 1111 1111 11 11 1110 0000 0000 0000 0000 => PD apunta a 1023 que es su propia base y luego va a la dirección 992
c) 0xFFFFF00 apunta a 1023 y nuevamente a 1023
d)

Este esquema se denomina **mapeo recursivo del directorio de páginas**, y tiene varios usos importantes en la gestión de memoria:

1. **Acceso a las estructuras de paginación**: Este mapeo recursivo permite que el sistema acceda directamente al **page directory** y a las **tablas de páginas** a través del espacio de direcciones virtuales. Es muy útil para los sistemas operativos, ya que permite manipular las tablas de paginación sin necesidad de realizar conversiones entre direcciones virtuales y físicas.
2. **Facilita la modificación de las tablas de páginas**: Cuando el sistema operativo necesita modificar o inspeccionar las estructuras de paginación (por ejemplo, para cambiar una entrada en el directorio o las tablas de páginas), puede hacerlo a través de estas direcciones virtuales especiales, sin tener que manejar directamente direcciones físicas.
3. **Eficiencia**: Este mecanismo simplifica la implementación del sistema operativo, ya que las tablas de páginas y el directorio de páginas son accesibles desde el mismo espacio de direcciones virtuales, sin necesidad de realizar complicados cálculos de direcciones físicas.

En resumen, este mapeo recursivo es un truco eficiente para permitir que el sistema operativo acceda y modifique sus propias estructuras de paginación directamente desde el espacio de direcciones virtuales.

Ejercicio 19. Explique como se usa la paginación para hacer:

(a) Dereferenciamiento de puntero a `NULL` tira excepción.
(b) Archivo de intercambio o *swap file*.
(c) *Demand paging* para la carga de programas.
(d) Auto-growing stack.
(e) Non-executable stacks.
(f) Memory mapped files `mmap()`.
(g) Copy-on-write (COW) para el `fork()`.
(h) `sbrk()` barato y por lo tanto `malloc()` barato.
(i) `malloc();memset(0)` = `calloc()` barato.
(j) Código compartido entre procesos: *shared libraries*, código de kernel, etc.
(k) Memoria compartida entre procesos.

a) No se mapea a la direccion 0 (NULL) por lo que si se quiere acceder se obtendrá una excepción
b) Mapea a disk en vez de a RAM
c) Las páginas no se cargan hasta que hagan falta, cuando se intentan acceder se genera una excepción y el SO decide si trae esa página o no desde el disco.
d) Si se intenta acceder a una región no mapeada del stack el sistema genera una excepción y de ser necesario se asigna una página al stack de manera dinámica.
e) Con paginación, las páginas de la memoria pueden tener permisos de acceso específicos. Por ejemplo, las páginas que contienen la pila pueden marcarse como **no ejecutables** (NX bit), lo que previene que el código malicioso sea ejecutado desde la pila. Esto es una medida de seguridad para evitar ciertos tipos de ataques, como el **stack-based buffer overflow**.
f) El sistema operativo puede mapear archivos a la memoria virtual de un proceso usando la llamada al sistema **mmap**(). Cuando el proceso accede a las páginas mapeadas, el sistema operativo carga las partes necesarias del archivo desde el disco a la memoria, permitiendo trabajar con archivos de gran tamaño sin tener que cargar todo el archivo en memoria a la vez.
g) En principio cuando se llama a fork() no se genera otro espacio de memoria física para el otro proceso si no que se usan otras páginas para mapear a la misma PFM, solamente uno de los procesos intenta escribir se genera un espacio físico para el proceso que lo necesite, compartiendo el resto de memoria.

h) La llamada **sbrk**() se utiliza para aumentar o disminuir el tamaño del segmento de datos de un proceso. Debido a la paginación, el sistema operativo puede asignar grandes cantidades de espacio virtual sin necesidad de mapear inmediatamente las páginas en memoria física. Solo cuando el proceso realmente utiliza las páginas asignadas por **sbrk**() (por ejemplo, al invocar **malloc**()), las páginas son mapeadas a memoria física.

i) Cuando se usa **calloc**(), el sistema no necesita asignar inmediatamente la memoria física y limpiar cada byte. En lugar de eso, el sistema operativo puede usar la paginación para marcar las páginas solicitadas como "vacías" (no asignadas). Cuando el proceso intenta escribir en esas páginas, el sistema operativo las asigna físicamente y las inicializa a cero en ese momento, haciendo que la operación sea más eficiente.

j) El código que es común entre procesos (como las bibliotecas compartidas o el código del kernel) puede estar mapeado a las mismas páginas físicas en la memoria, pero mapeado a diferentes espacios de direcciones virtuales en cada proceso. Esto permite que múltiples procesos usen el mismo código sin necesidad de duplicar la memoria.

k) El sistema operativo puede mapear la misma página física a diferentes espacios de direcciones virtuales de diferentes procesos. Esto permite que los procesos compartan memoria de manera eficiente. Por ejemplo, la función **shmget**() permite crear segmentos de memoria compartida que pueden ser accedidos por varios procesos, y la paginación permite mapear esos segmentos al espacio de direcciones de cada proceso.