Sistemas Operativos Avanzados

CC-571

César Lara Avila

Universidad Nacional de Ingeniería

(actualización: 2020-06-25)

Bienvenidos

Sesión 4

Virtualización

Temario de la sesión 4

- Espacio de direcciones
- API de la memoria
- Traducción de memoria
- Segmentación

¿Qué es la virtualización de memoria?

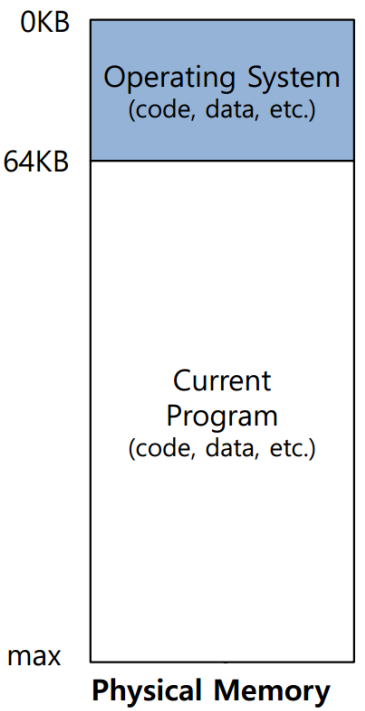
- El sistema operativo virtualiza su memoria física
- El sistema operativo proporciona un espacio de memoria de ilusión para cada proceso
- Parece verse que cada proceso usa toda la memoria.

Y por que virtualizar la memoria:

- ¡Porque la visión real de la memoria es desordenada!
- Anteriormente, la memoria solo tenía el código de un proceso en ejecución (y el código del sistema operativo)
- Ahora tiene múltiples procesos activos CPU de tiempo compartido
    - La memoria de muchos procesos debe estar en la memoria
    - No contiguos
    - Y se necesita ocultar esta complejidad al usuario.

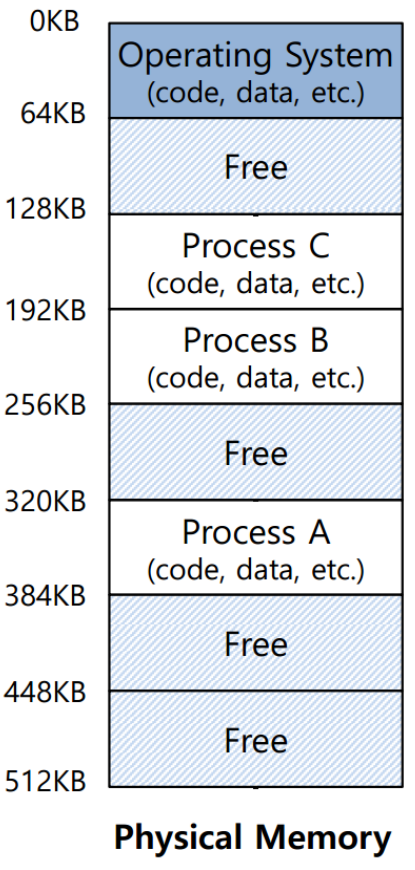
OS para los primeros sistemas

- Las primeras máquinas no proporcionaban mucha abstracción a los usuarios
- Se carga solo un proceso en la memoria y el OS era un librería en memoria
    - Comenzaba en la dirección física de 64 KB (ejemplo).
- Mala utilización y eficiencia.



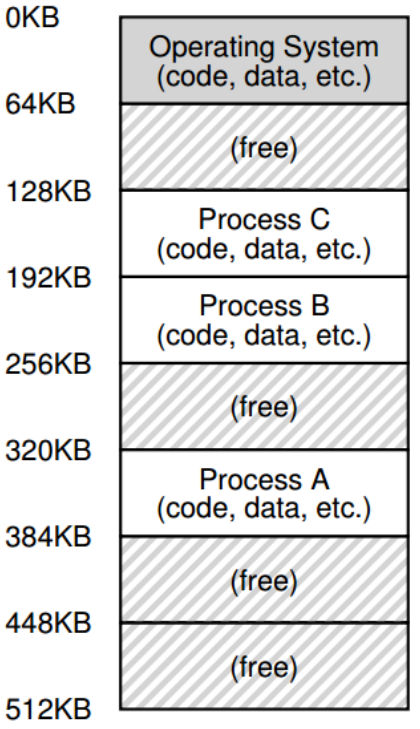
Multiprogramación y tiempo compartido

- Cargar múltiples procesos en la memoria
  - Ejecuta un proceso por un corto tiempo (tiempo compartido)
  - Cambia procesos entre ellos en memoria
  - Aumenta la utilización y la eficiencia.
- Causa un importante problema de protección
  - El proceso es demasiado lento
  - Accesos errantes a la memoria de otros procesos.



Espacio de direcciones virtuales

- El sistema operativo crea una abstracción de la memoria física
  - El espacio de direcciones contiene todo sobre un proceso en ejecución
  - Consiste en código del programa, heap, stack, etc.
    - El stack y el heap crecen durante el proceso de ejecución
  - El CPU emite cargas y almacena en direcciones virtuales.



- Cuando coexisten múltiples hilos en un espacio de direcciones, se toma en cuenta como dividir el espacio de direcciones.

Dirección virtual

- Cada dirección en un programa en ejecución es virtual.

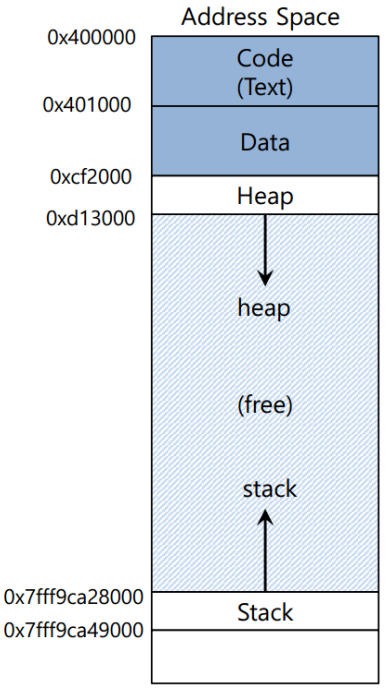
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

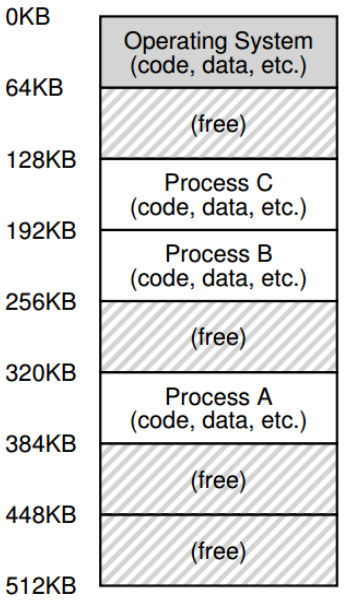
```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```

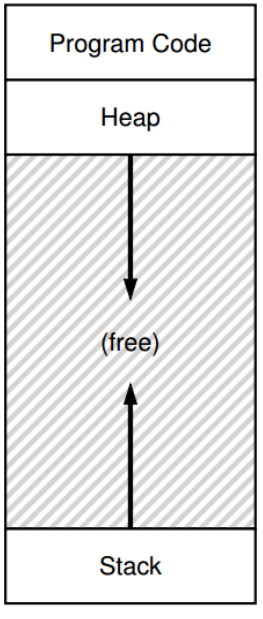


- El OS y el hardware traslada las direcciones virtuales para obtener valores de sus ubicaciones físicas reales.

¿Cómo se alcanza la memoria real?

- Traducción de direcciones de direcciones virtuales (VA) a direcciones físicas (PA) con el uso del MMU (Unidad de administración de Memoria)





- La CPU emite cargas/almacenamiento a VA y el hardware de memoria accede a PA
- OS asigna memoria y rastrea la ubicación de los procesos y pone a disposición la información necesaria.

Objetivos de la virtualización de la memoria.

- Transparencia: los programas de usuario no deben conocer los detalles desordenados
- Eficiencia: minimiza los gastos generales y el desperdicio en términos de espacio de memoria y tiempo de acceso
- Aislamiento y protección: un proceso de usuario no debe poder acceder a nada fuera de su espacio de direcciones.

API de memoria

- Interfaces de asignación de memoria en sistemas UNIX
- Tipos de memoria
  - Stack
  - Heap
- Debido a su naturaleza explícita y a su uso más variado, el heap presenta más desafíos tanto para los usuarios como para los sistema.

malloc()

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Asigna una región de memoria en el heap.
  - Argumento
    - *size_t size*: tamaño del bloque de memoria (en bytes)
    - *size_t* es un tipo entero sin signo.
  - Retorno
    - Éxito: un puntero de tipo void al bloque de memoria asignado por malloc.
    - Fallo: un puntero nulo.

sizeof()

- Las rutinas y las macros se utilizan para *size* en *malloc* en lugar de escribir un número directamente.
- Dos tipos de resultados de sizeof con variables
  - El tamaño actual de x se conoce en tiempo de ejecución

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

```
4
```

  - El tamaño actual de x se conoce en tiempo de compilación

```
int x[10];
printf("%d\n", sizeof(x));
```

```
40
```

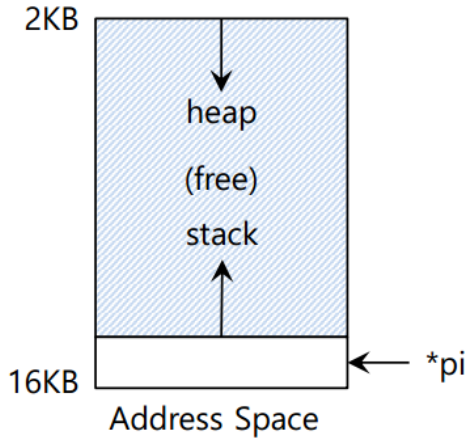
free()

```
#include <stdlib.h>

void free(void* ptr)
```

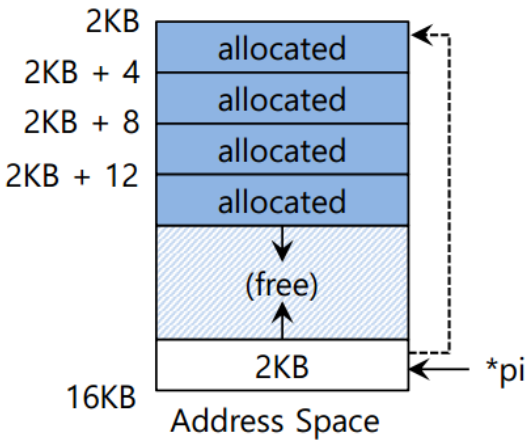
- Libere una región de memoria asignada por una llamada a malloc()
  - Argumento
    - *void* ptr *: un puntero a un bloque de memoria asignado con malloc
  - Retorno
    - None.
- Asignar memoria es la parte fácil de la ecuación; saber cuándo, cómo e incluso si liberar memoria es la parte difícil.
- Si pasas otro valor a free(), cosas malas pueden suceder.

Asignación de memoria



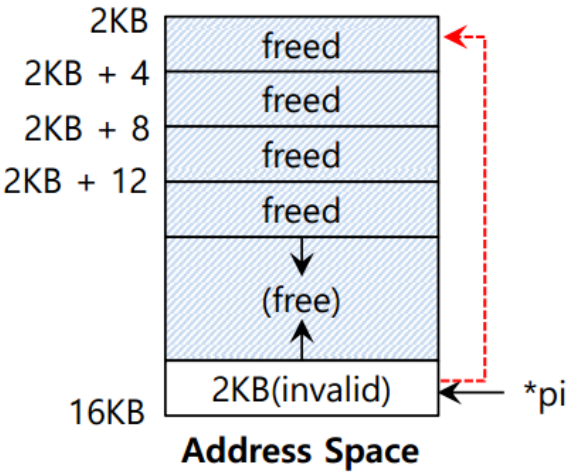
pointer

```
int *pi; // local variable
```

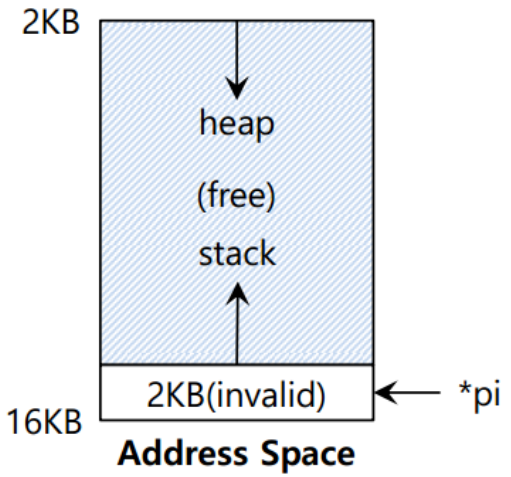



```
pi = (int *)malloc(sizeof(int)* 4);
```

Liberación de memoria



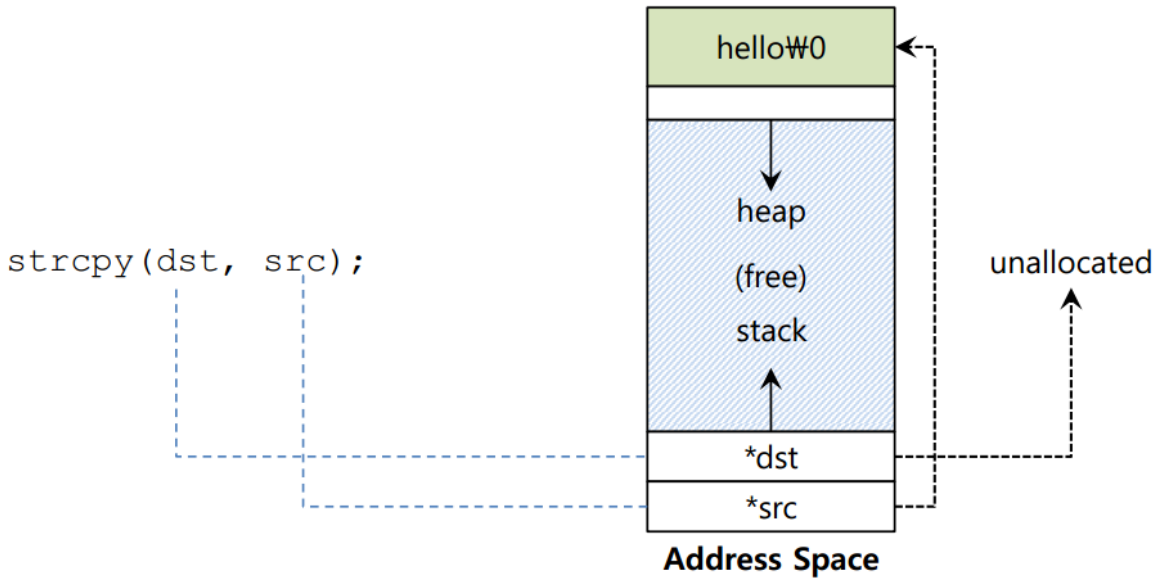

```
free(pi);
```



Errores comunes: olvidar asignar memoria

- Código incorrecto

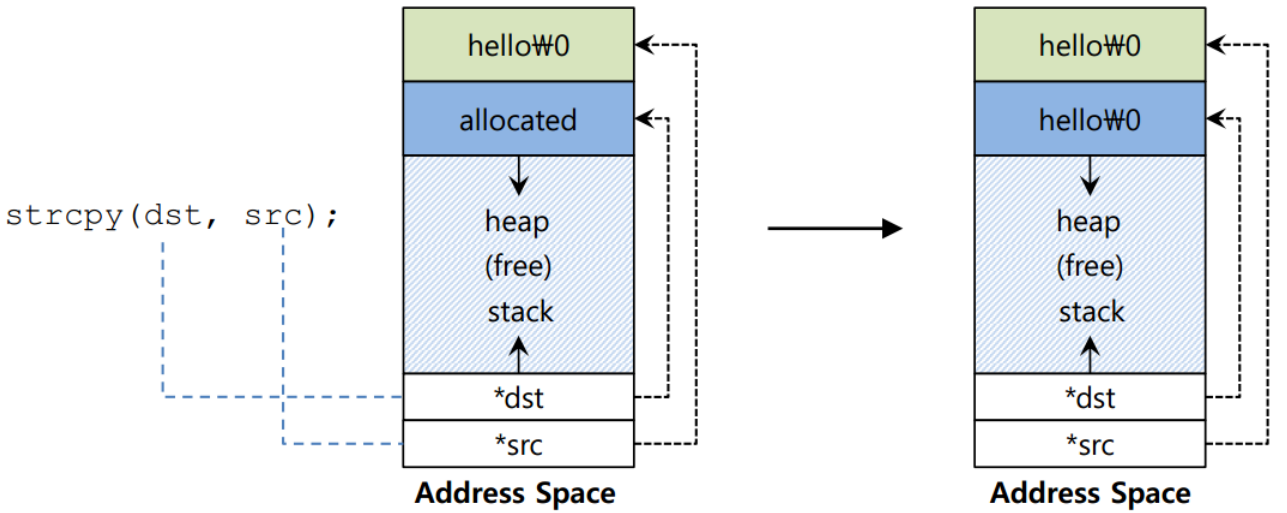
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



Errores comunes: olvidar asignar memoria (continuación)

- Código correcto

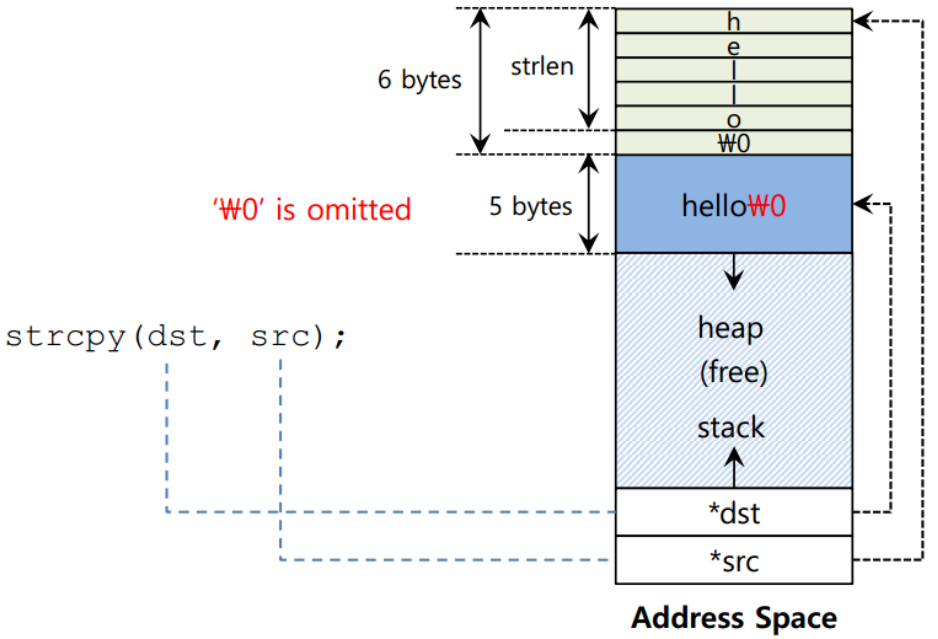
```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Errores comunes: no asignar suficiente memoria

- Buffer overflow
- Código incorrecto, pero funciona correctamente

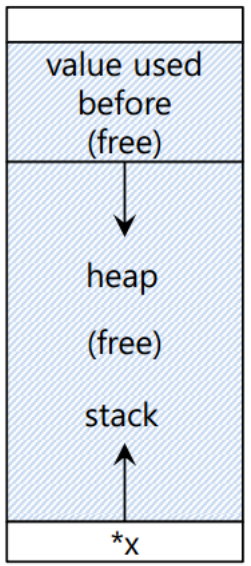
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);    //work properly
```



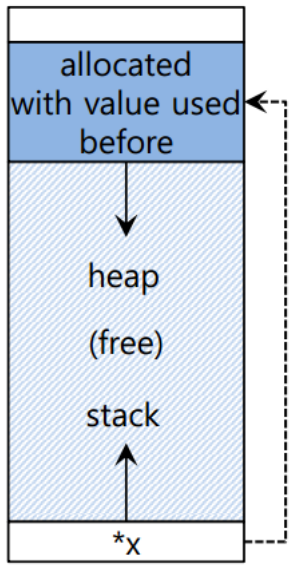
Errores comunes: olvido de inicializar

- Encuentra una lectura no inicializada (no se sabe que se encuentra?)

```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x); // uninitialized memory access
```



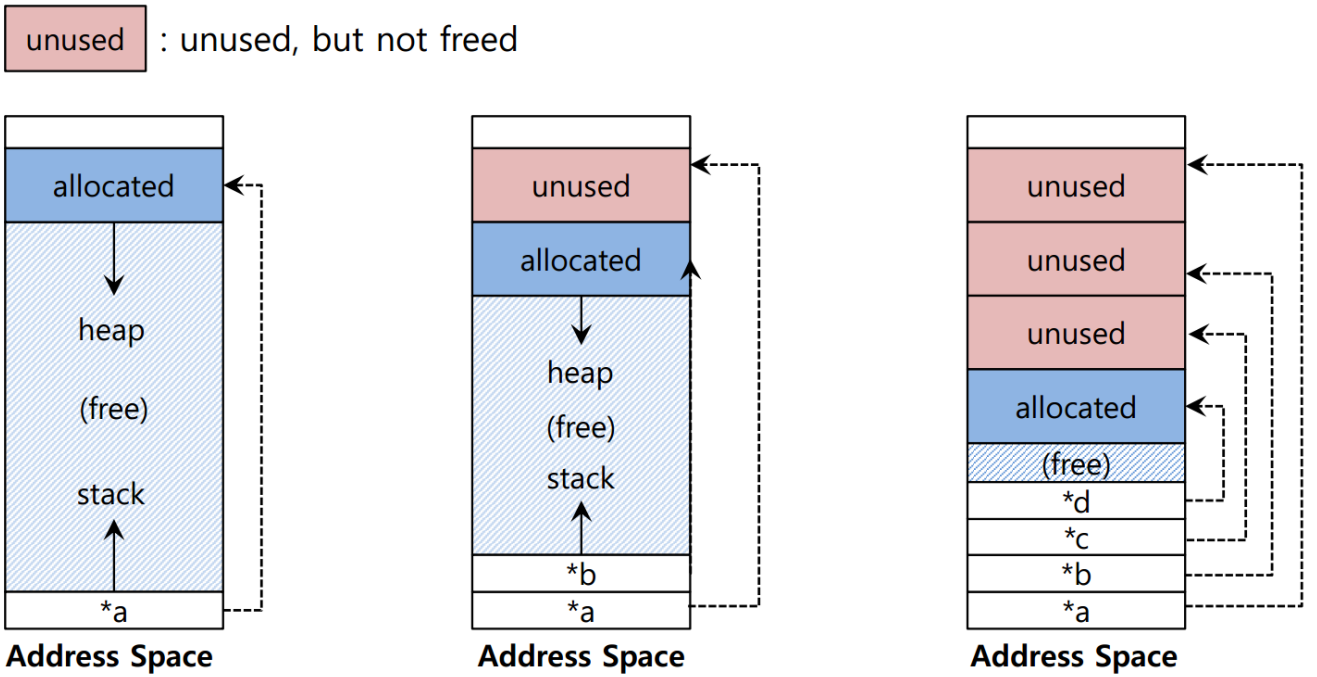
Address Space



Address Space

Pérdida de memoria

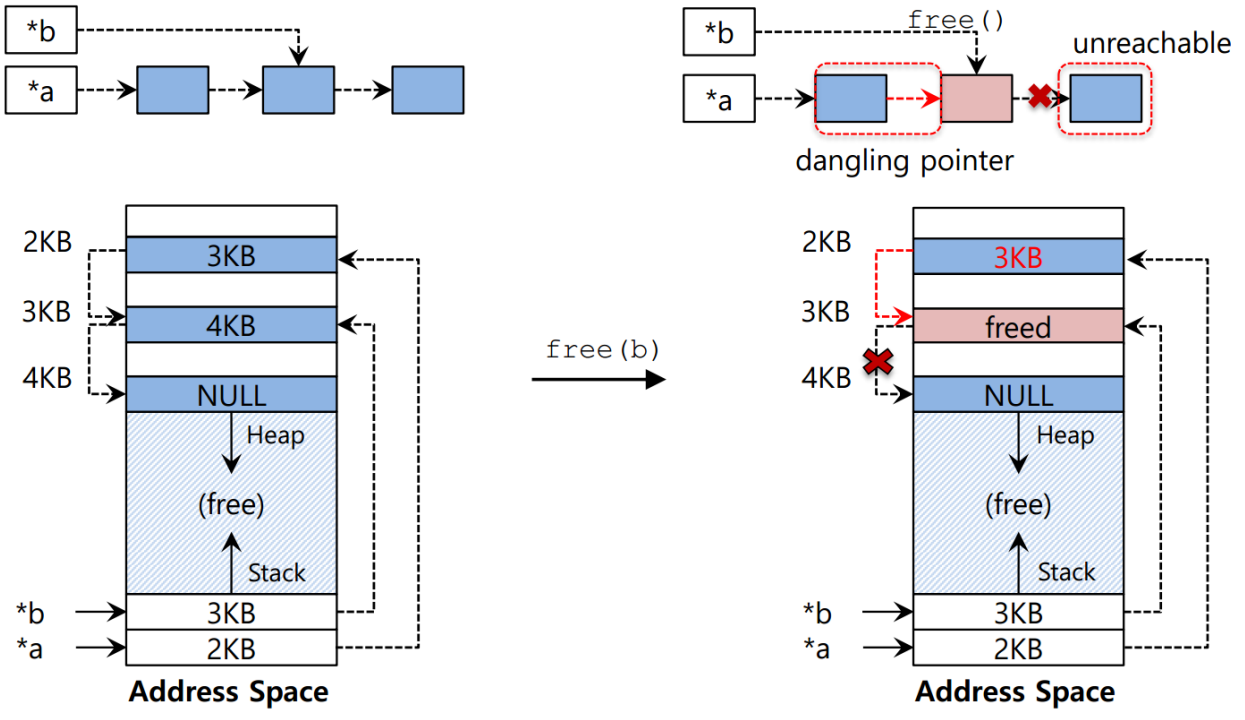
- Un programa se queda sin memoria y finalmente muere.
  - El OS limpiará todas sus páginas asignadas y, por lo tanto, no se producirá ninguna pérdida de memoria per se.



- Es un mal hábito para desarrollar.

Punteros colgantes

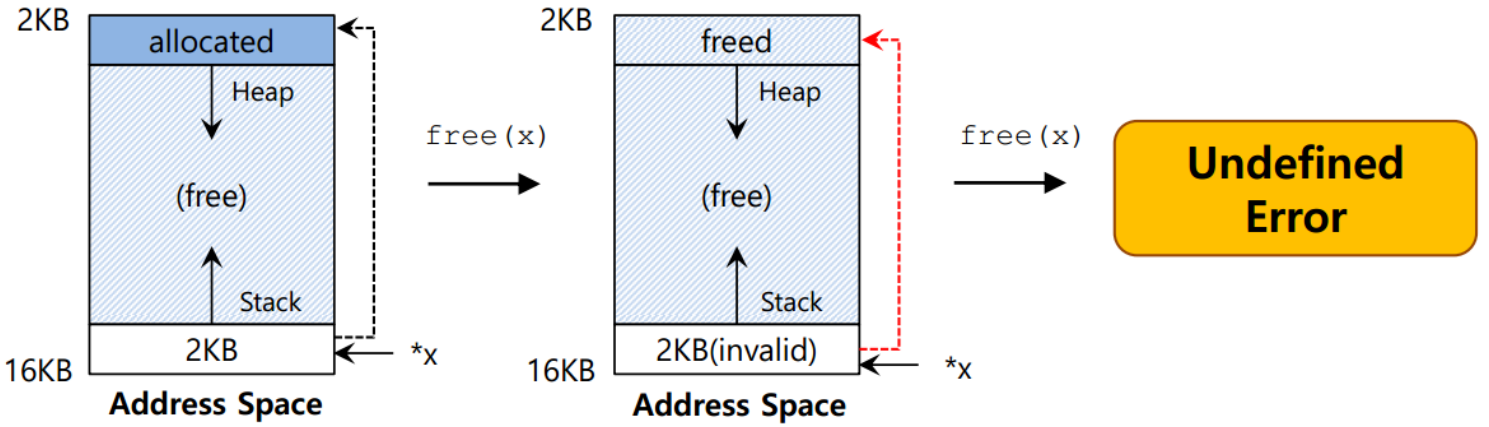
- Liberar memoria antes de que se termine de usar
  - Un programa accede a la memoria con un puntero no válido.



Double free

- Los programas a veces también liberan memoria más de una vez

```c
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory
free(x); // free repeatedly
```



Llamadas al sistema

```
#include <unistd.h>

int brk(void *addr)
void *sbrk(intptr_t increment);
```

- La llamada a la biblioteca malloc() usa brk.
  - Se llama a brk para expandir el *break* del programa
  - *break*: la ubicación del final del heap en el espacio de direcciones
  - sbrk es una llamada adicional similar a brk.
  - Los programadores nunca deben llamar directamente a brk o sbrk.

```
#include <sys/mman.h>

void *mmap(void *ptr, size_t length, int port, int flags,
int fd, off_t offset)
```

- La llamada al sistema mmap puede crear una región de memoria anónima.

Otras API de memoria: calloc()

```
#include <stdlib.h>

void *calloc(size_t num, size_t size)
```

- Asigna memoria en el heap y asigna a cero antes de regresar.
  - Argumento
    - *size_t num*: cantidad de bloques a asignar
    - *size_t size*: tamaño de cada bloque (en bytes).
  - Retorno
    - Éxito: un puntero de tipo void al bloque de memoria asignado por calloc.
    - Fallo: un puntero nulo.

Otras API de memoria: realloc()

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Cambiar el tamaño del bloque de memoria
  - Un puntero devuelto por realloc puede ser el mismo que ptr o uno nuevo
  - Argumento
    - *void* ptr*: Puntero al bloque de memoria asignado con malloc, calloc o realloc.
    - *size_t size*: Nuevo tamaño para el bloque de memoria (en bytes).
  - Retorno
    - Éxito: un puntero de tipo void al bloque de memoria
    - Fallo: un puntero nulo.

Mecanismo de traducción de direcciones

- La virtualización de memoria toma una estrategia similar como ejecución directa limitada (LDE) para la eficiencia y el control
- En la virtualización de la memoria, la eficiencia se logran con el soporte de hardware
    - Por ejemplo: registros, TLB (Translation Look-aside Buffer)s, tabla de páginas
- El control implica que el OS asegura que ninguna aplicación tenga acceso a ninguna memoria que no sea la suya.
    - Se necesita ayuda del hardware para las aplicaciones y el OS.

Traducción de direcciones basadas en hardware

- El hardware transforma una dirección virtual en una dirección física
  - La información deseada se almacena realmente en una dirección física.
- El OS debe involucrarse en los puntos clave para configurar el hardware
  - El OS debe administrar la memoria para intervenir juiciosament en mantener el control sobre cómo se usa la memoria.

Ejemplo

- Código en C

```
void func()
        int x;
        ...
        x = x + 3; // this is the line of code we are interested in
```

  - Cargar un valor de la memoria
  - Incrementa en 3
  - Almacena el valor nuevamente en la memoria

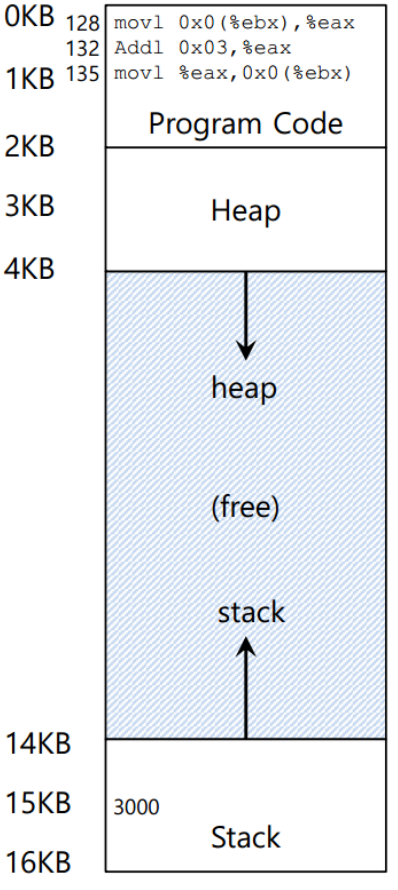
- Ensamblador

```
128 : movl 0x0(%ebx), %eax      ; load 0+ebx into eax
132 : addl $0x03, %eax          ; add 3 to eax register
135 : movl %eax, 0x0(%ebx)      ; store eax back to mem
```

  - Carga el valor en esa dirección en el registro eax.
  - Agregue 3 al registro eax.
  - Almacena el valor en eax nuevamente en la memoria.

Ejemplo

- Cuando se ejecutan estas instrucciones, desde la perspectiva del proceso, tienen lugar los siguientes accesos a la memoria:
  - Obtención de la instrucción en la dirección 128
  - Ejecución de la instrucción (carga desde la dirección 15 KB)
  - Se obtiene la instrucción en la dirección 132
  - Ejecución de esta instrucción (sin referencia de memoria)
  - Obtención de las instrucciones en la dirección 135.
  - Ejecuta esta instrucción (almacenar en dirección 15 KB).



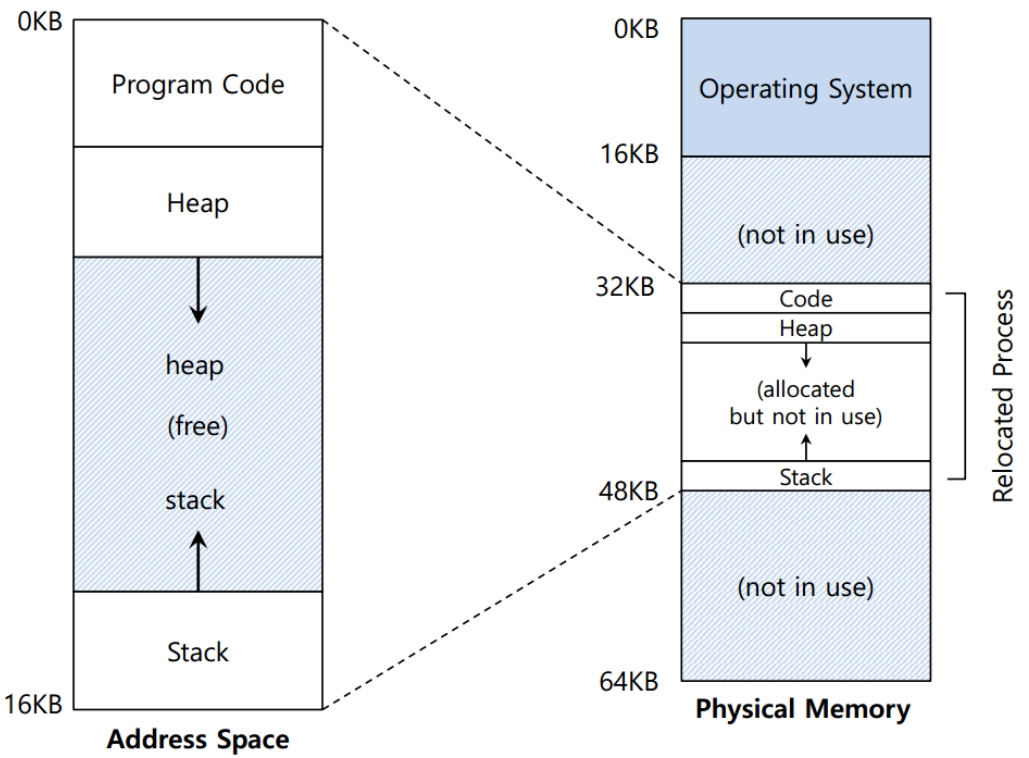
Reasignación del espacio de direcciones

- El OS quiere colocar el proceso en otro lugar en la memoria física, no en la dirección 0.

¿cómo podemos reubicar este proceso en la memoria de una manera que sea transparente para el proceso?

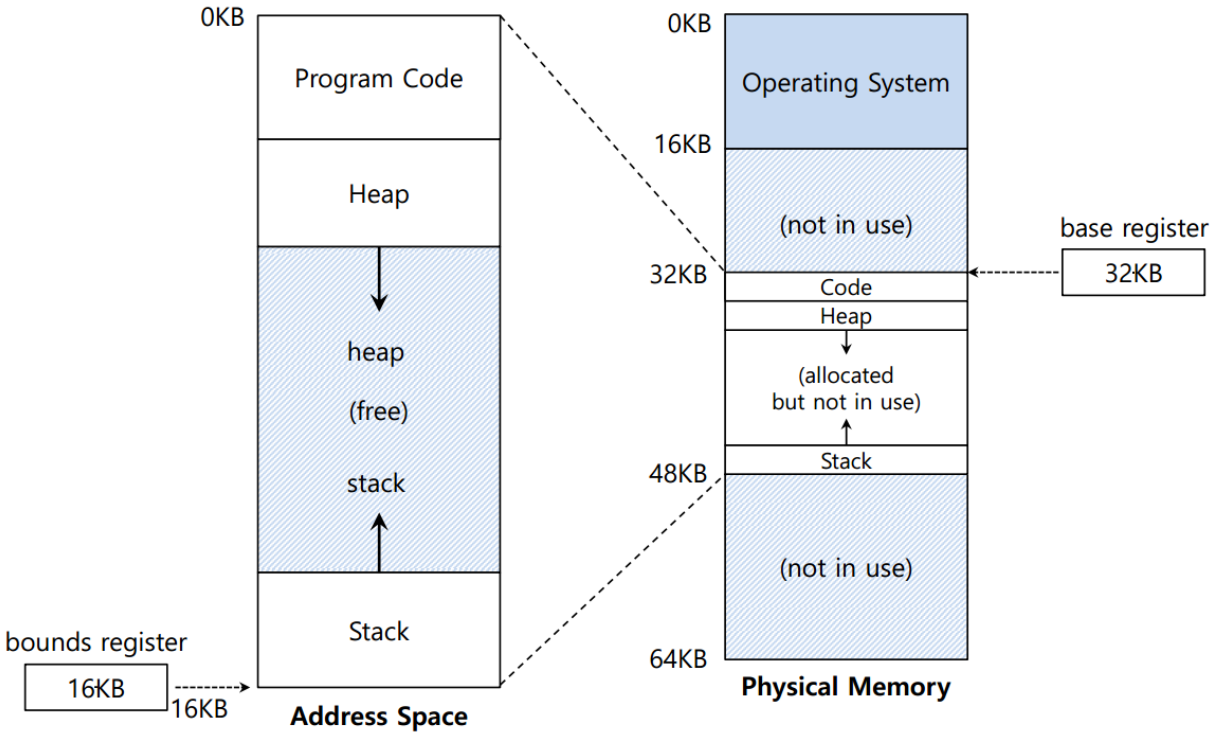
¿Cómo podemos proporcionar la ilusión de un espacio de direcciones virtual que comienza en 0, cuando en realidad el espacio de direcciones se encuentra en alguna otra dirección física?.

Un proceso único reasignado



- Un ejemplo de cómo se vería la memoria física una vez que el espacio de direcciones de este proceso se haya colocado en la memoria.

Registros base y límites



- Nos permiten colocar el espacio de direcciones en cualquier lugar que deseemos en la memoria física
    - Garantizan que el proceso solo pueda acceder a su propio espacio de direcciones.

Reasignación dinámica (base de hardware)

- Cuando un programa comienza a ejecutarse, el sistema operativo decide dónde se debe cargar un proceso en la memoria física.
  - Establece el registro base de un valor y los valores límites.
  - Cuando el proceso genera cualquier referencia de memoria, el MMU traduce de la siguiente manera:

$$\text{direccion física} = \text{dirección virtual} + \text{base}$$

- Cada dirección virtual no debe ser mayor que el valor limitada y no negativo. El MMU se encarga de comprobar esta condición

$$0 \leq \text{dirección virtual} < \text{límites}$$

- El OS no participa en todas las traducciones.

Reasignación y traducción de direcciones

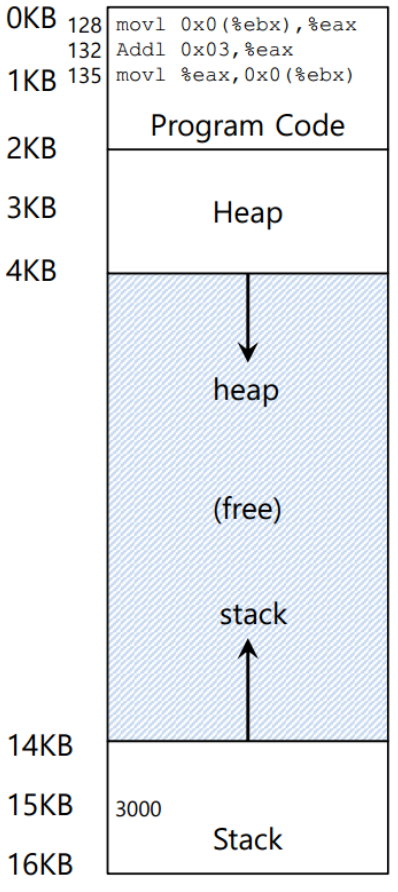
```
128 : movl 0x0(%ebx), %eax
```

- Se obtiene instrucciones en la dirección 128

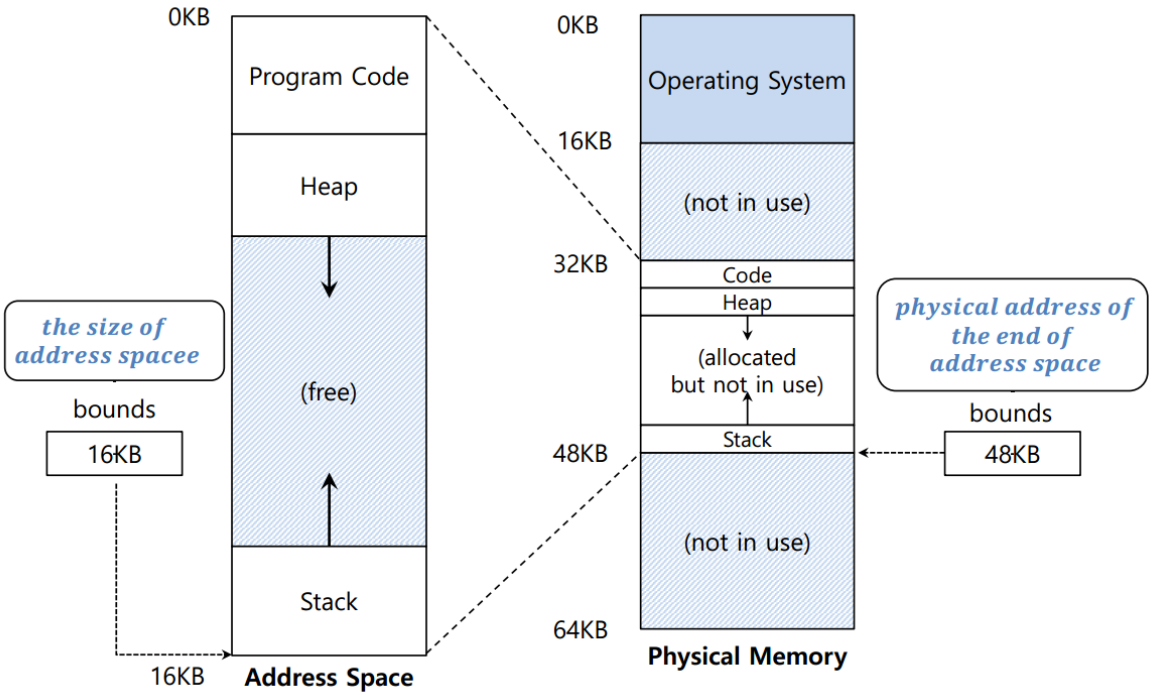
  $32896 = 128 + 32KB(\text{base})$

- Ejecuta esa instrucción
  - Carga desde la dirección 15 KB?

  $47KB = 15KB + 32KB(\text{base})$



Dos definiciones del registro de límites



- Contiene el tamaño del espacio de direcciones y, por lo tanto, el hardware verifica primero la dirección virtual antes de agregar la base.
- Contiene la dirección física del final del espacio de direcciones y, por lo tanto, el hardware agrega primero la base y luego se asegura de que la dirección esté dentro de los límites.

Rol del hardware en la traducción

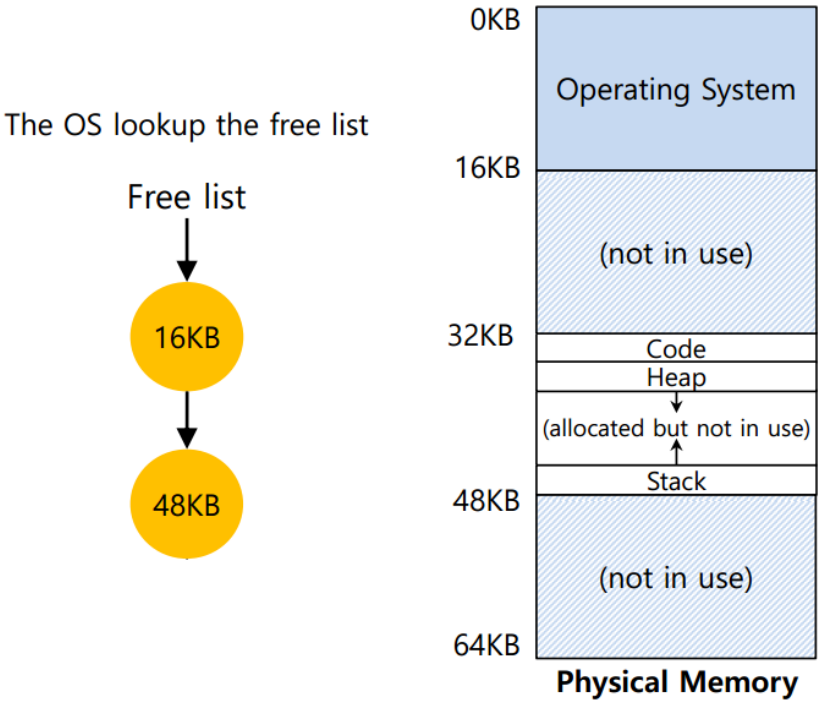
- CPU proporciona un modo de ejecución privilegiado
- El conjunto de instrucciones tiene instrucciones privilegiadas para configurar la información de traducción (por ejemplo, base, límites)
- El hardware (MMU) usa esta información para realizar la traducción en cada acceso a la memoria
- MMU genera fallas para el OS cuando el acceso es ilegal (por ejemplo, cuando la dirección virtual está fuera de límite).

Problemas del OS con la virtualización de la memoria

- El sistema operativo mantiene una lista libre de memoria
- Asigna espacio cuando se crea un proceso (y cuando se le solicite) y lo libera cuando finaliza
- Mantiene información de dónde se asigna el espacio a cada proceso (en PCB)
- Establece la información de traducción de direcciones (por ejemplo, base y límites) en el hardware
- Actualiza esta información al cambiar de contexto
- Maneja fallas debido al acceso ilegal a la memoria (excepciones).

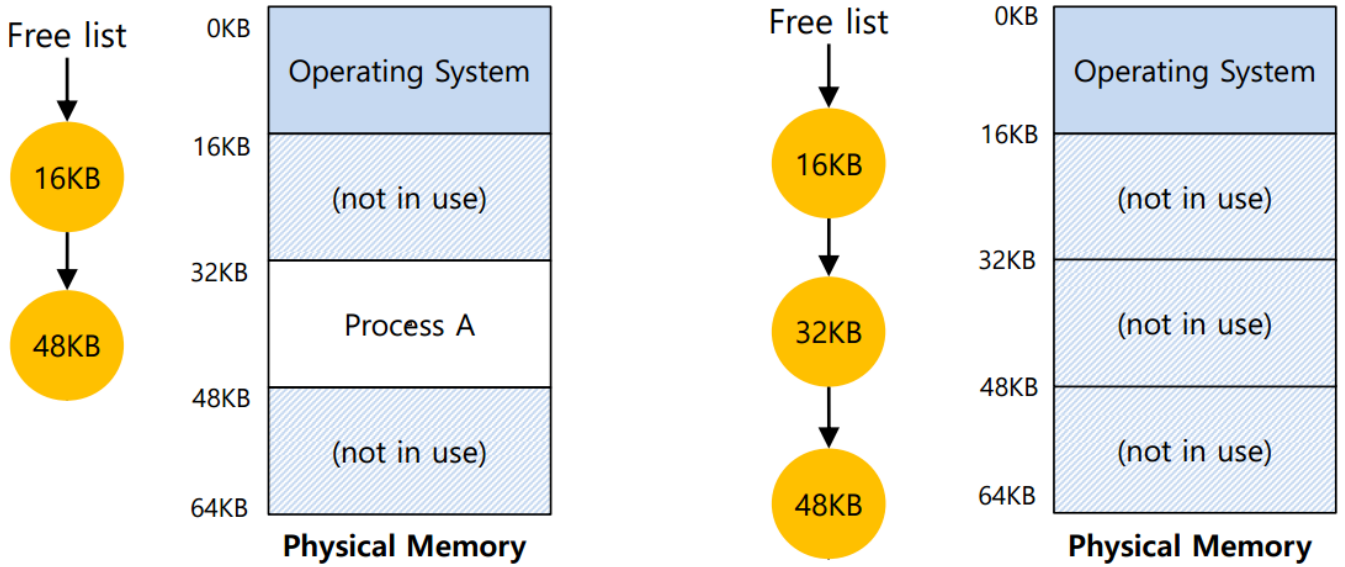
Problemas del OS: cuando un proceso comienza a ejecutarse

- El sistema operativo debe encontrar un espacio para un nuevo espacio de direcciones
  - Lista libre: una lista del rango de la memoria física que no está en uso.



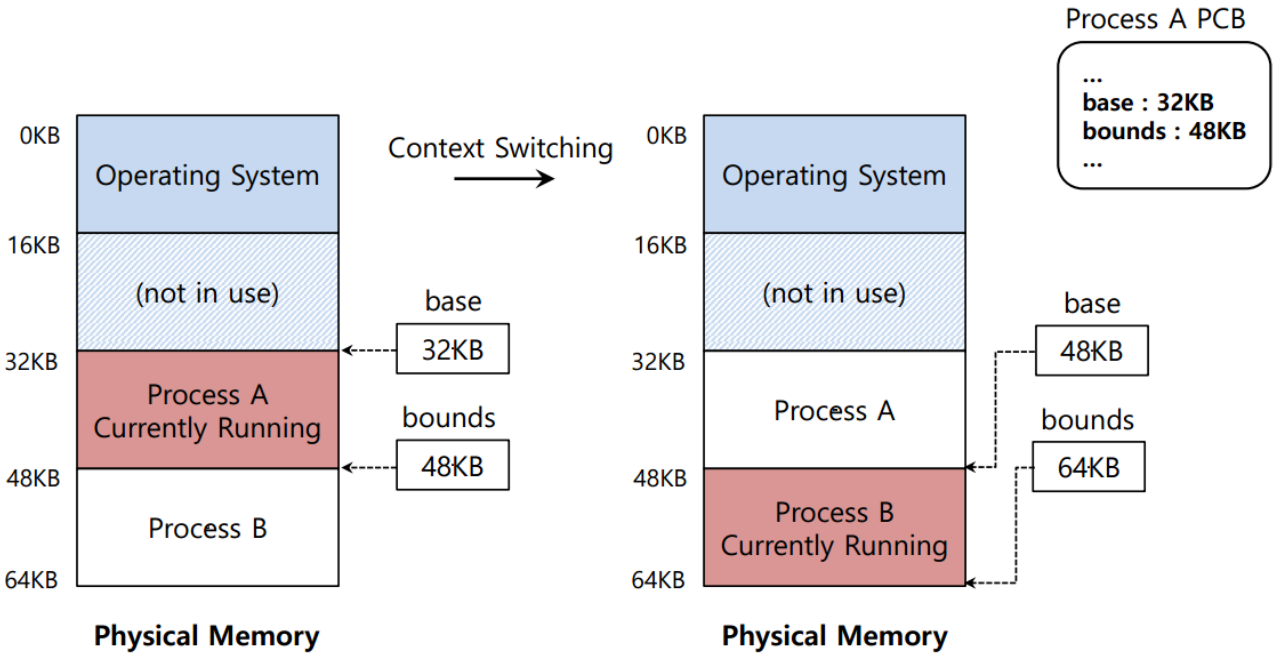
Problemas del OS: cuando se termina un proceso

- El sistema operativo debe volver a colocar la memoria en la lista libre.



Problemas del sistema operativo: cuándo se produce el cambio de contexto

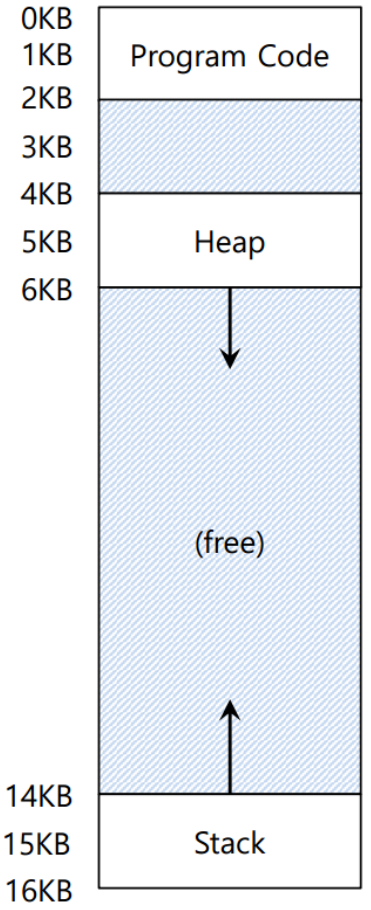
- El sistema operativo debe guardar y restaurar el registro base y límite
  - Estructura de proceso o bloque de control de proceso (PCB).



Ineficiencia del enfoque base y límite

- Mucho espacio *libre* produce fragmentación interna
- El espacio dentro de la unidad asignada no se usa todo (es decir, está fragmentado) y, por lo tanto, se desperdicia
- Difícil de ejecutar cuando un espacio de direcciones no cabe en la memoria física.

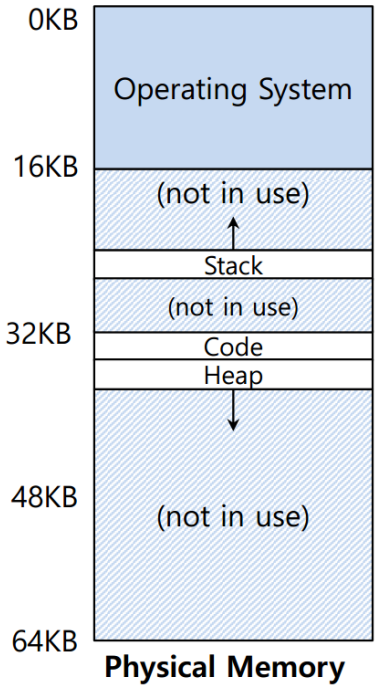
`Mejora del enfoque con la segmentación.`



Segmentación

- El segmento es solo una parte contigua del espacio de direcciones de una longitud particular.
    - Segmento lógicamente diferente: código, stack, heap
- Cada segmento se puede colocar en diferentes partes de la memoria física
    - Base y límites existen para cada segmento.
- Permite al OS colocar cada uno de esos segmentos en diferentes partes de la memoria física, y así evitar llenar la memoria física con el espacio de direcciones virtuales no utilizado.

Colocación de un segmento en la memoria física



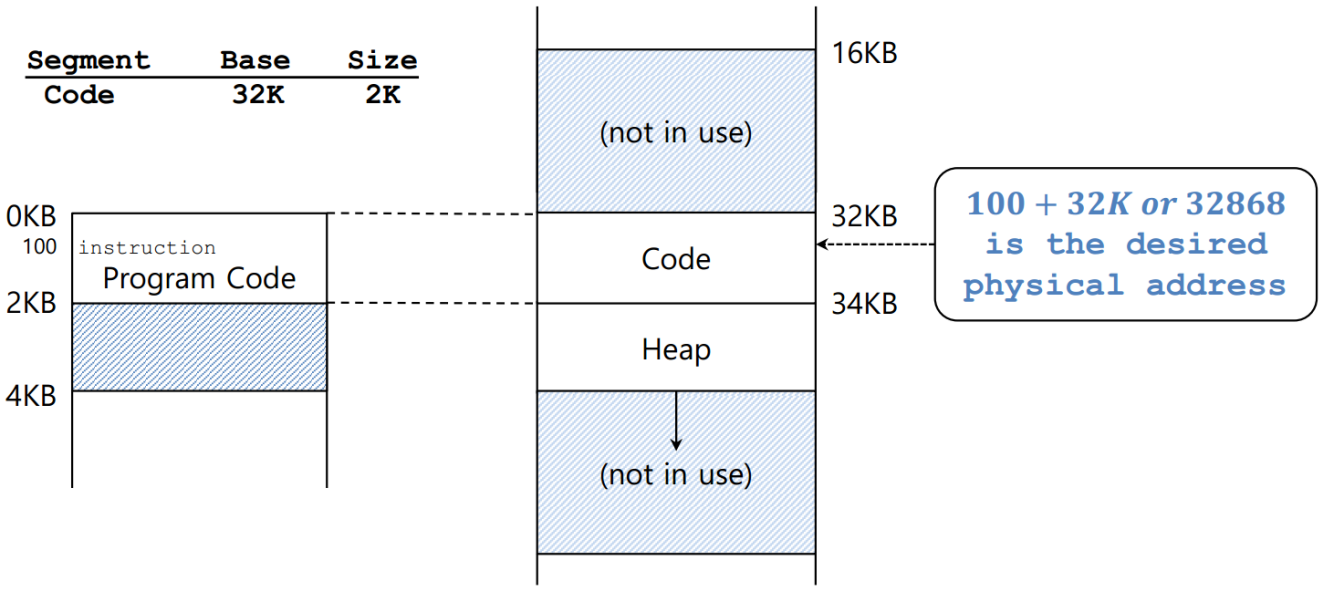
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

- Se muestra los valores de registro.

Traducción de direcciones en segmentación

$$\text{dirección física} = \text{offset} + \text{base}$$

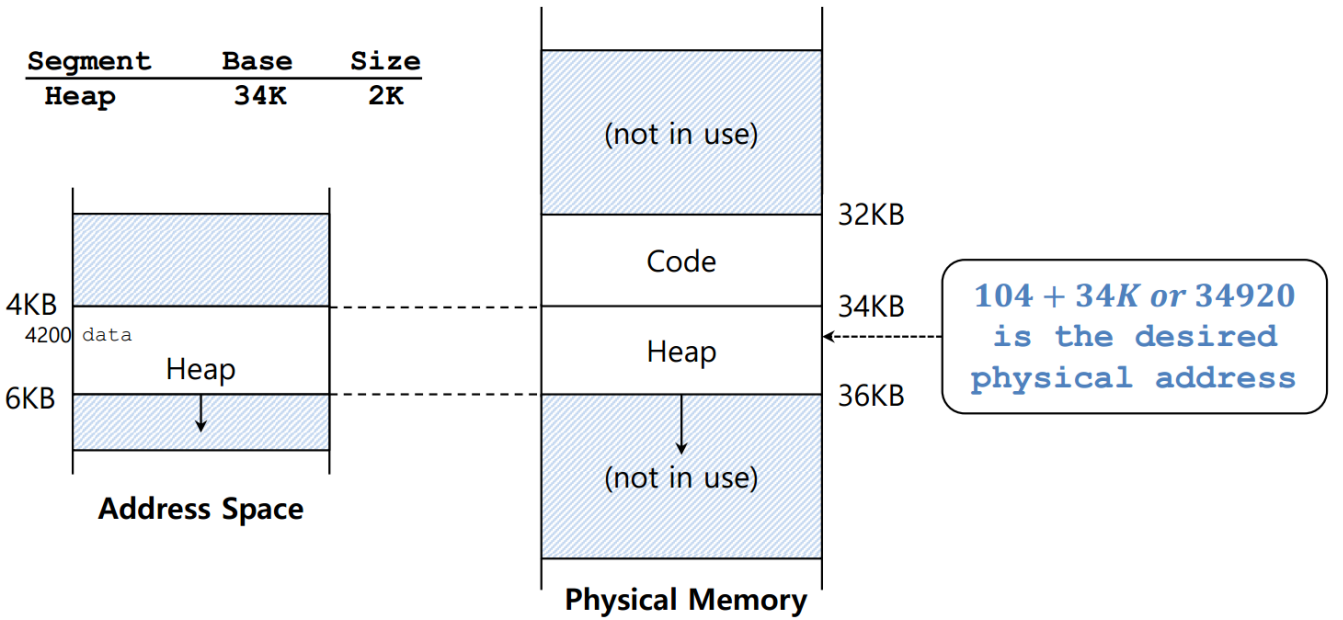
- El *offset* de la dirección virtual 100 es 100
  - El segmento de código comienza en la dirección virtual 0 en el espacio de direcciones.



- Se verifica que la dirección está dentro de los límites y se emite la referencia a la dirección de memoria física 32868.

Traducción de direcciones en segmentación

- El *offset* de la dirección virtual 4200 es 104.
  - El segmento heap comienza en la dirección virtual 4096 en el espacio de direcciones.



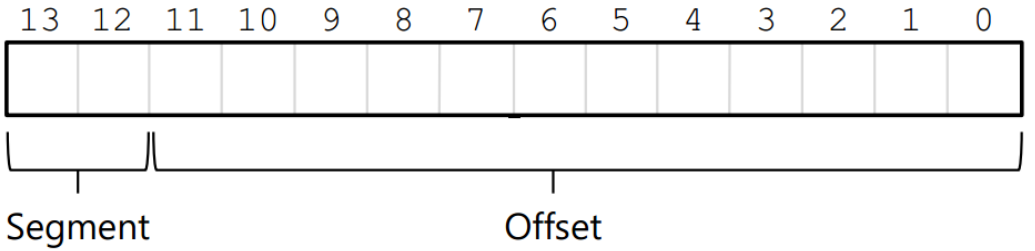
dirección virtual + base no es la dirección física correcta

Falla o violación de segmentación

- Si se hace referencia a una dirección ilegal, como 7 KB en el ejemplo, que está más allá del final del heap, el sistema operativo produce una falla de segmentación.
    - El hardware detecta que la dirección está fuera de los límites.
- En general, el término fallo o violación de segmentación surge de un acceso a la memoria en una máquina segmentada a una dirección ilegal.

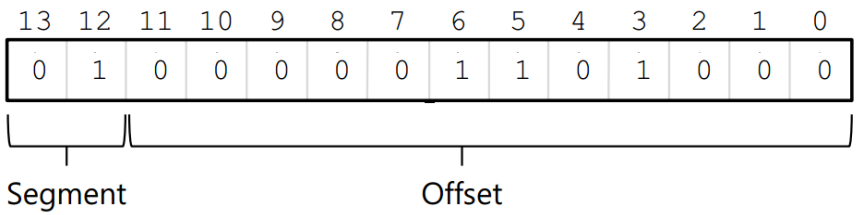
¿ A qué segmento nos referimos?

- Enfoque explícito
  - Corta el espacio de direcciones en segmentos basados en los pocos bytes superiores de la dirección virtual
  - Se utiliza en el sistema VAX/VMS.



- Ejemplo: dirección virtual $4200 (01000001101000)$

Segment	bits
Code	00
Heap	01
Stack	10
-	11



¿A qué segmento nos referimos (continuación)

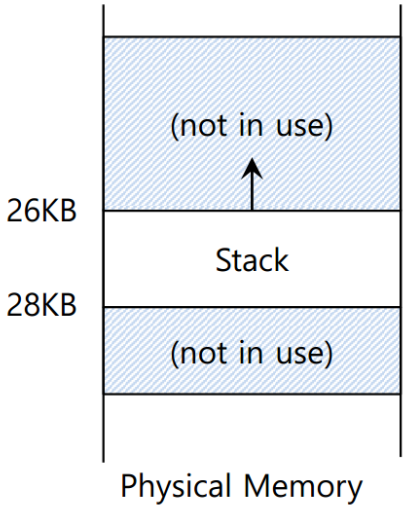
```
1   // get top 2 bits of 14-bit VA
2   Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3   // now get offset
4   Offset = VirtualAddress & OFFSET_MASK
5   if (Offset >= Bounds[Segment])
6        RaiseException(PROTECTION_FAULT)
7   else
8        PhysAddr = Base[Segment] + Offset
9        Register = AccessMemory(PhysAddr)
```

```
SEG_MASK = $0x3000 (11000000000000)$
SEG_SHIFT = $12$
OFFSET_MASK = $0xFFF (00111111111111)$
```

- Enfoque implícito
  - El hardware determina el segmento al notar cómo se formó la dirección.

Refiriéndose al segmento Stack

- Crece hacia atrás.
- Se necesita soporte de hardware adicional
  - El hardware comprueba en qué dirección crece el segmento
    - 1: dirección positiva, 0: dirección negativa
- Con la comprensión del hardware de que los segmentos pueden crecer en la dirección negativa, el hardware ahora debe traducir dichas direcciones virtuales de manera ligeramente diferente



Segment Register(with Negative-Growth Support)

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Soporte para compartir

- El segmento se puede compartir entre el espacio de direcciones
  - El código compartido todavía se usa en los sistemas hoy en día
- Se necesita soporte de hardware adicional para la forma de bits de protección
  - Se agrega unos cuantos bits más por segmento para indicar permisos de lectura, escritura y ejecución.

Segment Register Values(with Protection)

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

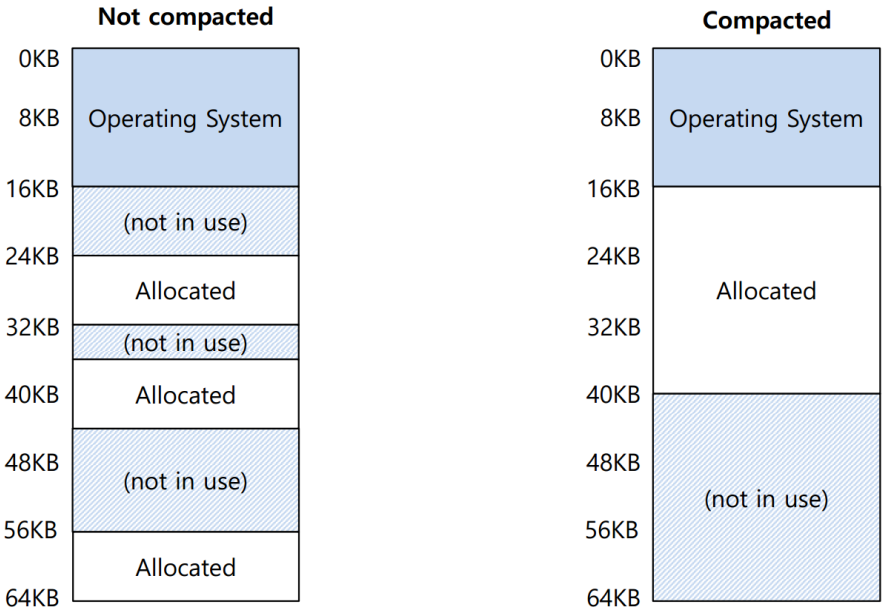
- El algoritmo de hardware cambia, ya que debe verificar si un acceso en particular está permitido.

Segmentación de grano fino y de grano grueso

- De grano grueso significa segmentación en un sistema con un número pequeño de segmentos
    - Por ejemplo, código, heap, stack.
- La segmentación de grano fino permite más flexibilidad para el espacio de direcciones en algunos sistemas
    - Para admitir muchos segmentos, se requiere soporte de hardware con una tabla de segmentos
    - Máquinas como Burroughs B5000 tenían soporte para miles de segmentos.

Problemas con la segmentación

- ¿Qué debe hacer el sistema OS en un cambio de contexto?
  - Los registros de segmento deben guardarse y restaurarse
- ¿ Cómo es la administración del espacio libre en la memoria física?



  - Se tiene una serie de segmentos por proceso, y cada segmento podría ser diferente tamaño.

Soporte del OS: fragmentación

- Fragmentación externa: pequeños agujeros del espacio libre en la memoria física que dificultan la asignación de nuevos segmentos.
    - Hay 24KB libres (ejemplo), pero no en un segmento contiguo
    - El OS no puede satisfacer la solicitud de 20 KB.
- compactación: reorganizando los segmentos existentes en la memoria física.
    - La compactación es costosa
        - Deja de ejecutar el proceso
        - Copia datos a alguna parte
        - Cambia el valor del registro de segmento.
- Otros enfoques minimizan la fragmentación externa son los algoritmos **best fit**, **worst-fit, first-fit** y **algoritmo buddy**.

Fin!

[1] Algunos de los gráficos de esta presentación se basan en el texto **Operating Systems: Three Easy Pieces** de Remzi H. Arpaci-Dusseau y Andrea C. Arpaci-Dusseau.

[2] Algunos de los gráficos de esta presentación se basan en las notas de Youjip Won, Hanyang University, Embedded Software Systems Laboratory.