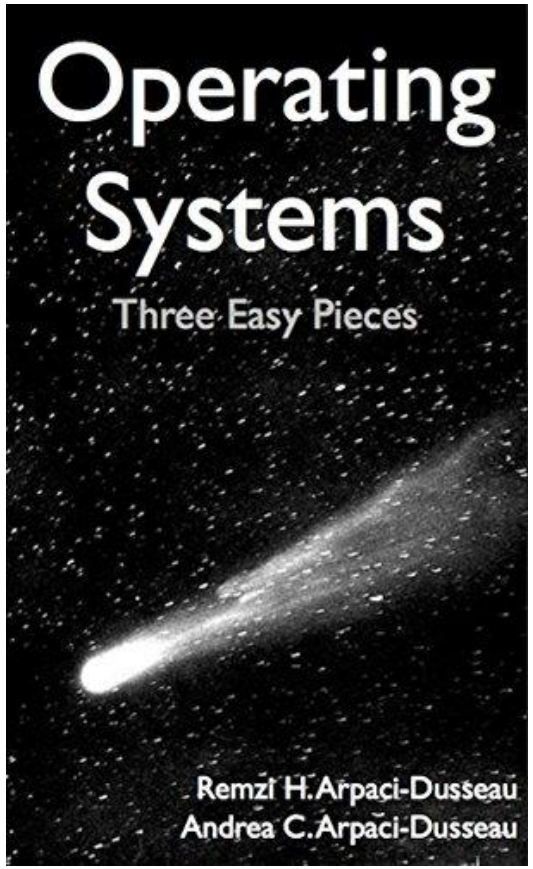
İşletim Sistemleri



4. Ders

Prof. Dr. Kemal Bıçakcı

Alıştırma Sorusu

- Aşağıdaki zamanlama algoritmalarından hangisi en düşük ortalama tepki süresine (response time) sahiptir?

a) Round Robin

b) Önce En Kısa İş

c) İlk Giren İlk Çıkar

d) Piyango Zamanlama (önceden dağıtılmış eşit olmayan sayıda biletle)

13. Soyutlama: Adres Uzayı

İşletim Sistemleri: Üç Basit Parça

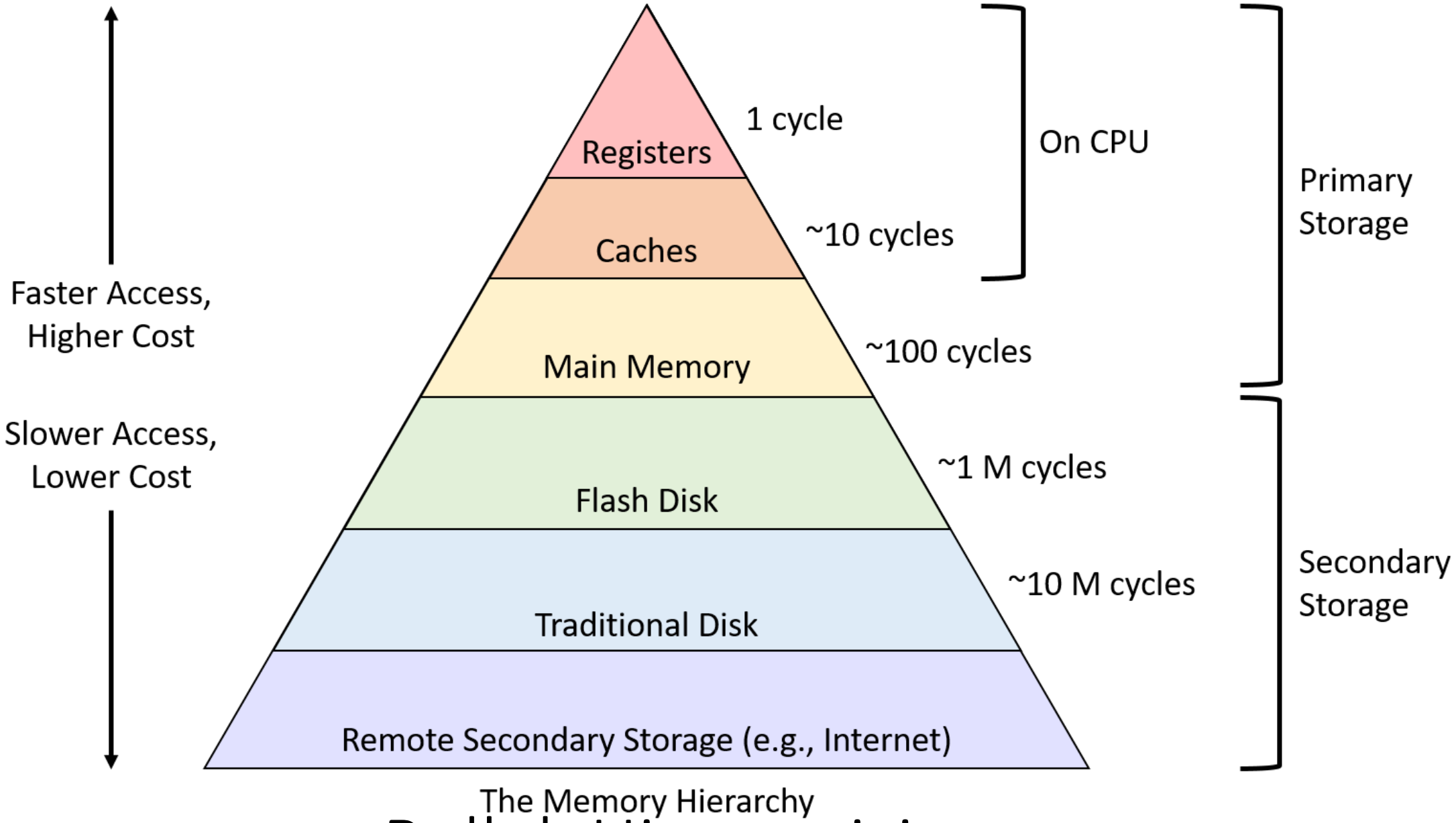
Bellek Sanallaştırma

- Bir kullanıcı programı tarafından üretilen her adres sanal bir adrestir.

- **Bellek Sanallaştırma** ne demektir?
  - İşletim Sistemi fiziksel belleği sanallaştırır.
  - İşletim Sistemi her işlem için bir bellek uzayı illüzyonu sağlar.
  - Her işlem sanki tüm belleği kullanıyormuş gibidir.

Bellek Sanallaştırmanın Yararları

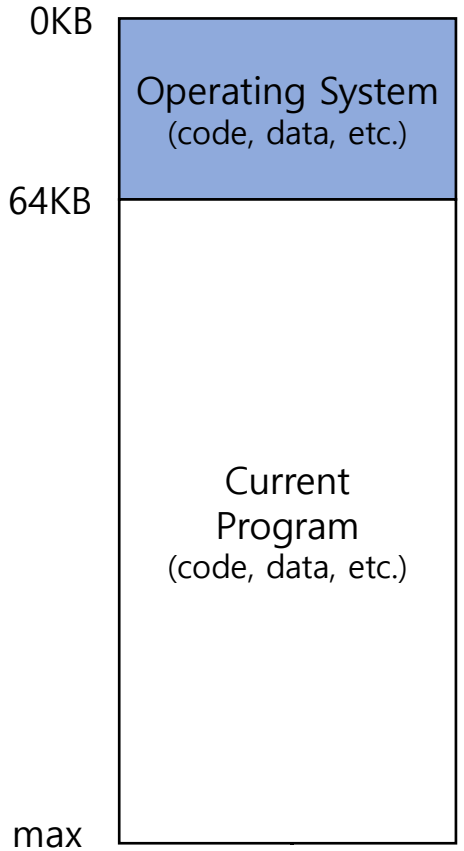
- Programlama kolaylığı
- Zaman ve yer açısından bellek verimliliği
- İşletim sistemi ve işlemler için izolasyon garantisi
  - Başka işlemlerin hatalı erişimlerinden korunma



Bellek Hiyerarşisi

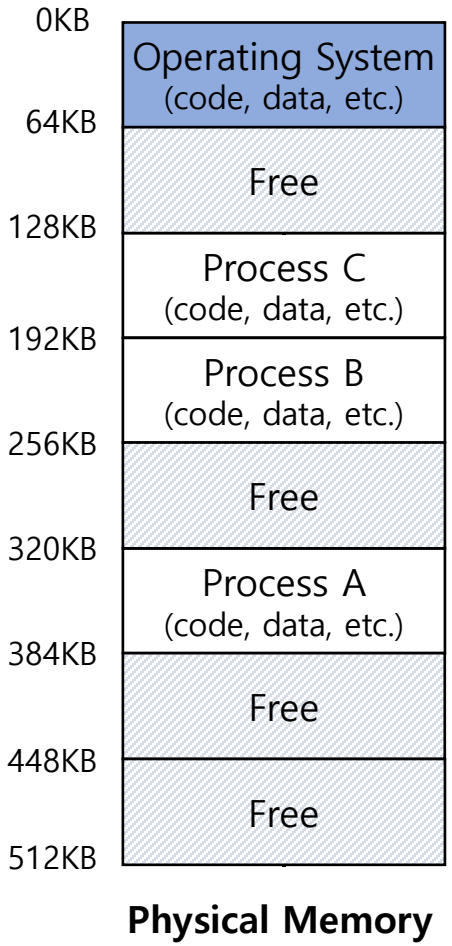
İlk Sistemler

- Belleğe sadece bir işlem yükle.
- Yetersiz kullanım ve verimsizlik



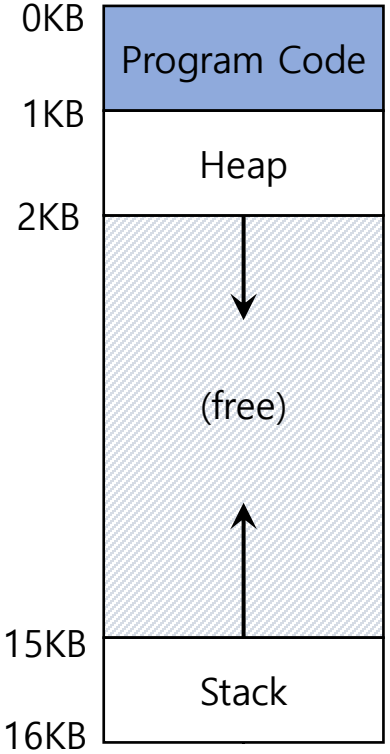
Çoklu Programlama ve Zaman Paylaşımı

- Belleğe **birden fazla işlem** yükle.
  - Birini kısa bir süre çalıştır.
  - Bellekteki diğer bir işleme geç.
  - Kullanım ve Verimlilik artar.
- Önemli bir koruma sorununa yol açar.
  - Diğer işlemlerin hatalı bellek erişimleri



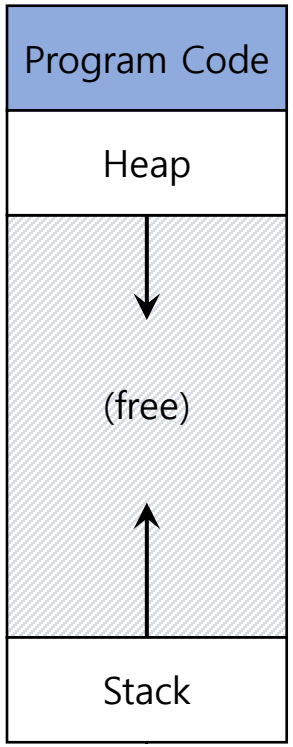
Adres Uzayı

- İşletim Sistemi fiziksel belleği sanallaştırır.
- Böylelikle, adres uzayı sadece o an çalışan işleme aittir.
- Bu uzay, program kodu, öbek, yığın, vb. alanlardan oluşur.



Adres Uzayı (devam)

- Kod
  - Buyrukların bulunduğu kısım
- Öbek
  - Dinamik olarak tahsis edilen bellek.
    - C dilindeki `malloc`
    - Nesne tabanlı dillerdeki `new`
- Yığın
  - Dönüş adresi ve değerlerini saklar.
  - Yerel değişkenler ve çağrılan rutinlerin argümanlarını tutar.



Sanal Adres

- Çalışan bir programdaki her adresin sanal olduğunu hatırlayalım.
- İşletim sistemi sanal adresi fiziksel adrese çevirir.

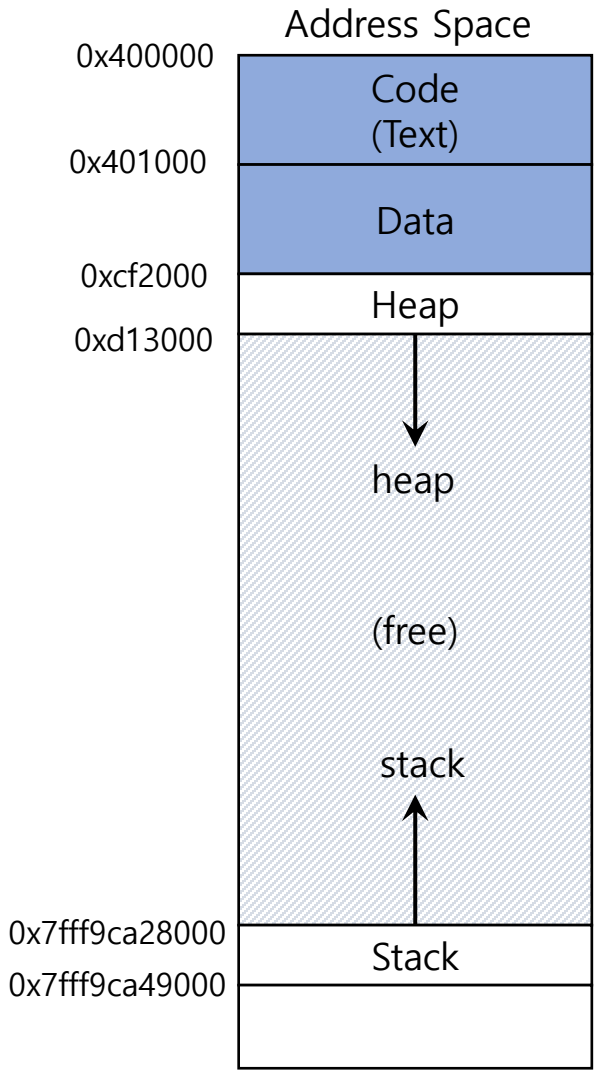
```c
#include <stdio.h>
#include <stdlib.h>

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;
}
```

Adresleri yazdıran basit bir program

Sanal Adres (Devam)

- Bir 64-bit Linux makinedeki çıktı:

```
Kod adresi    : 0x40057d
Öbek adresi   : 0xcf2010
Yığın adresi : 0x7fff9ca45fcc
```



14. Bellek API

İşletim Sistemleri: Üç Basit Parça

Bellek API: malloc()

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Öbekte bir bellek alanı tahsis eder.
  - Argüman
    - `size_t size`: bellek alanının büyüklüğü (byte cinsinden)
    - `size_t` işaretsiz tamsayı türünde.
  - Dönen değer
    - Başarılı: `malloc` ile tahsis edilen bellek alanını gösteren void türünde bir işaretçi (pointer)
    - Başarısız: null pointer

sizeof()

- `malloc` çağrısı yapılırken büyüklük (`size`) için rutinler, operatörler ve makrolar kullanılır (doğrudan bir sayı yazmak yerine).
- `sizeof` operatörünü bir değişken ile birlikte kullanırken ortaya çıkan iki kafa karıştırıcı sonuç:

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

```
4
```

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

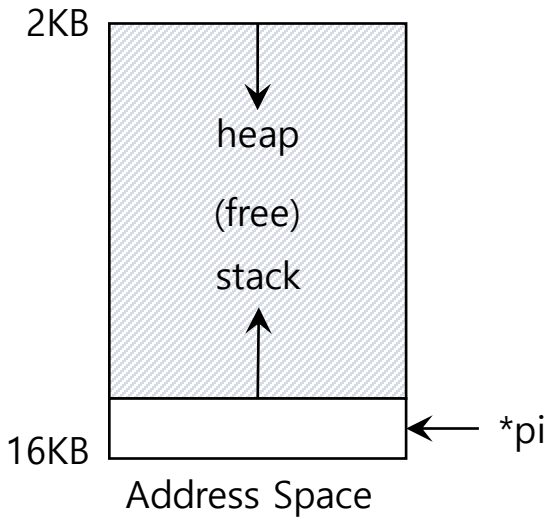
```
40
```

Bellek API: free()

```
#include <stdlib.h>

void free(void* ptr)
```

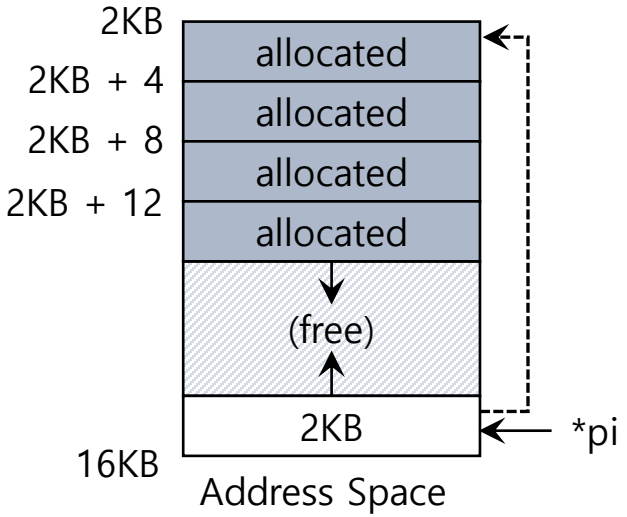
- `malloc` ile tahsis edilen bir bellek alanının boşaltır
- Argüman
  - `void *ptr`: `malloc` ile tahsis edilen alanı gösteren işaretçi (pointer)
- Dönüş değeri
  - yok

Bellek Tahsisi



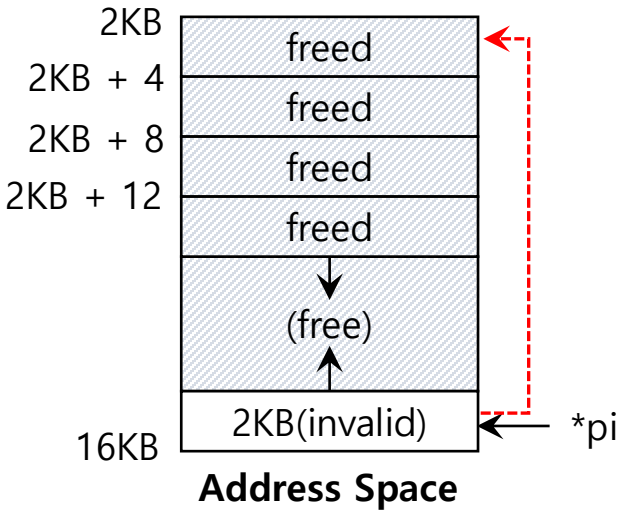
pointer

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

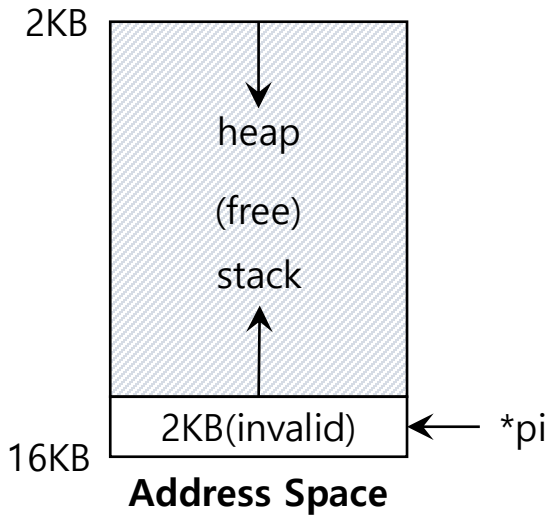



```
pi = (int *)malloc(sizeof(int)* 4);
```

Belleğin Boşaltılması




```
free(pi);
```



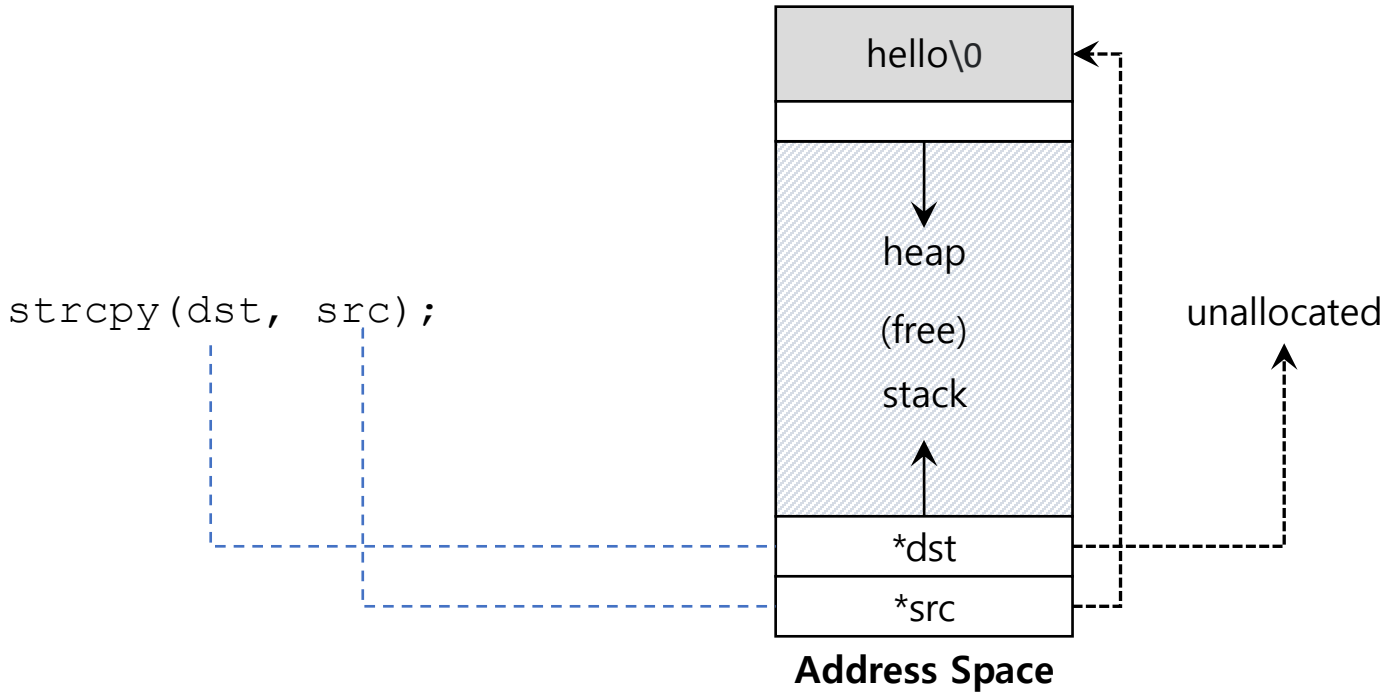
Yaygın Hatalar

- Bellek tahsisini unutmak
- Yeterli bellek alanı tahsis etmemek
- Tahsis edilen belleğe başlangıç değerlerini yazmayı unutmak
- Belleği boşaltmayı unutmak
- İş bitmeden belleği boşaltmak
- Belleği tekrar tekrar boşaltmak
- free()'yi yanlış çağırmak

Bellek Tahsisini Unutmak

Yanlış Kod:

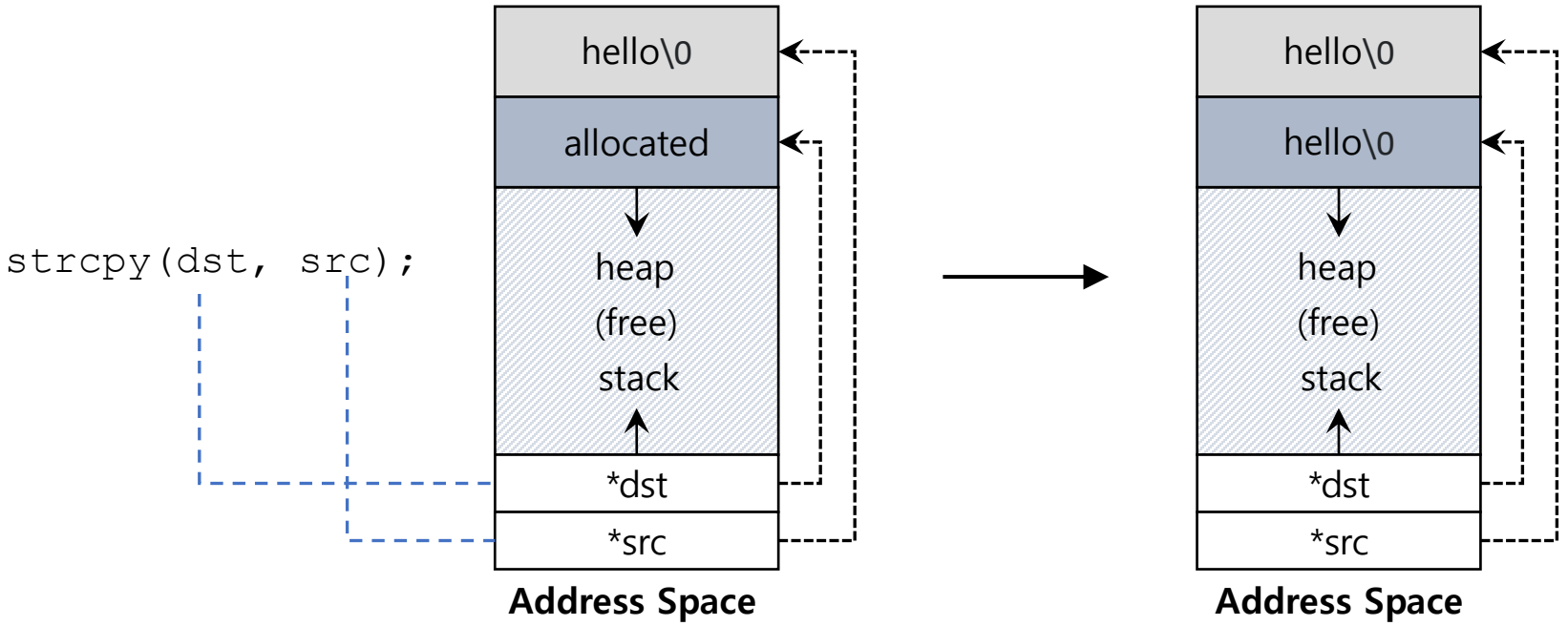
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



Bellek Tahsisini Unutmak(Devam)

Doğru Kod:

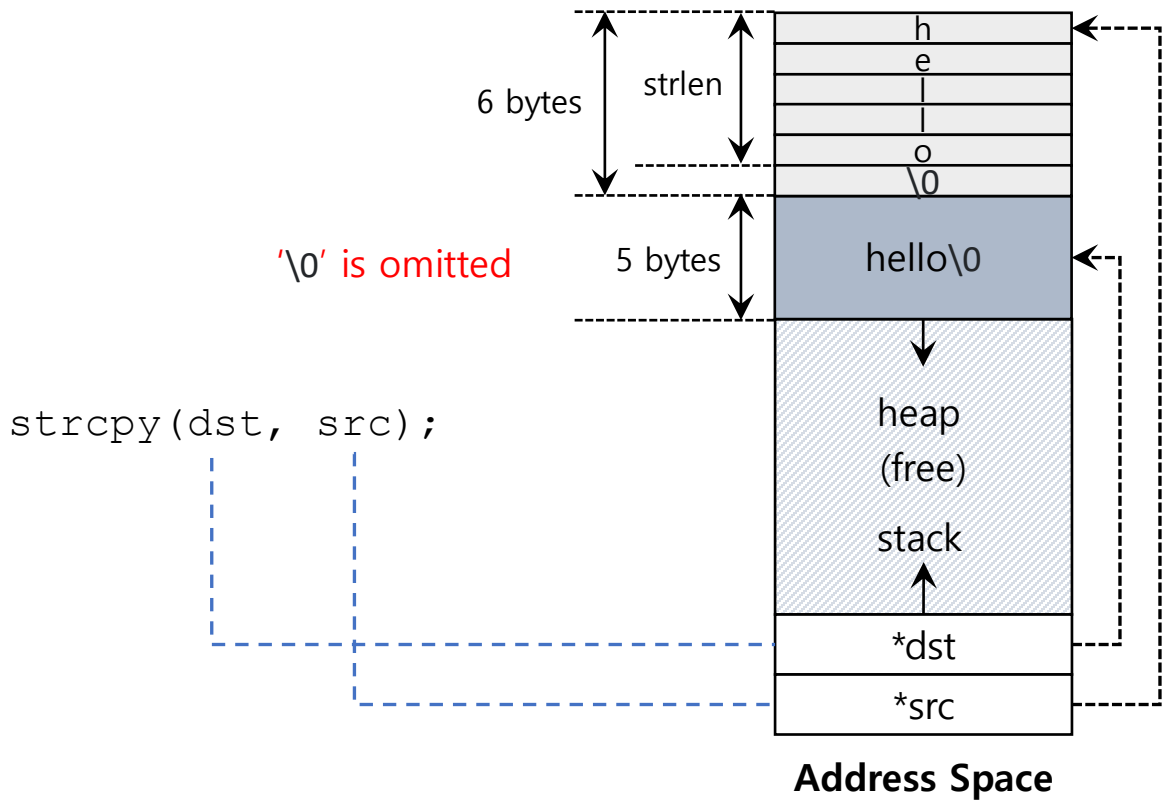
```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Yeterli bellek alanı tahsis etmemek

- Yanlış kod (fakat doğru çalışabilir):

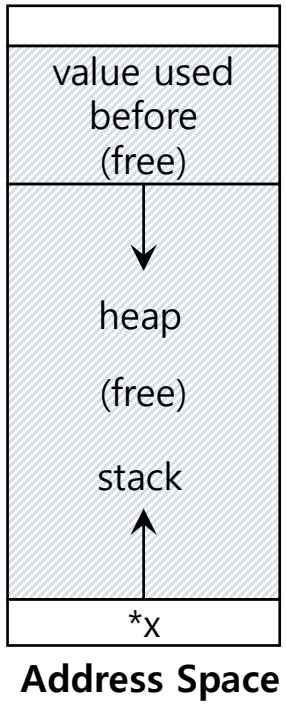
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);    //work properly
```



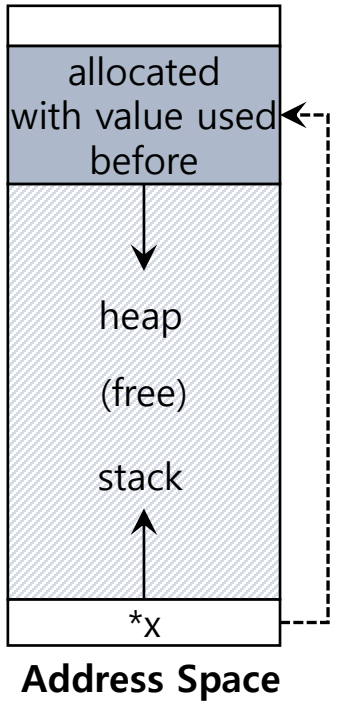
Tahsis edilen belleğe başlangıç değerlerini yazmayı unutmak

Başlangıç değeri yazılmamış bir bellek alanının okunması:

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



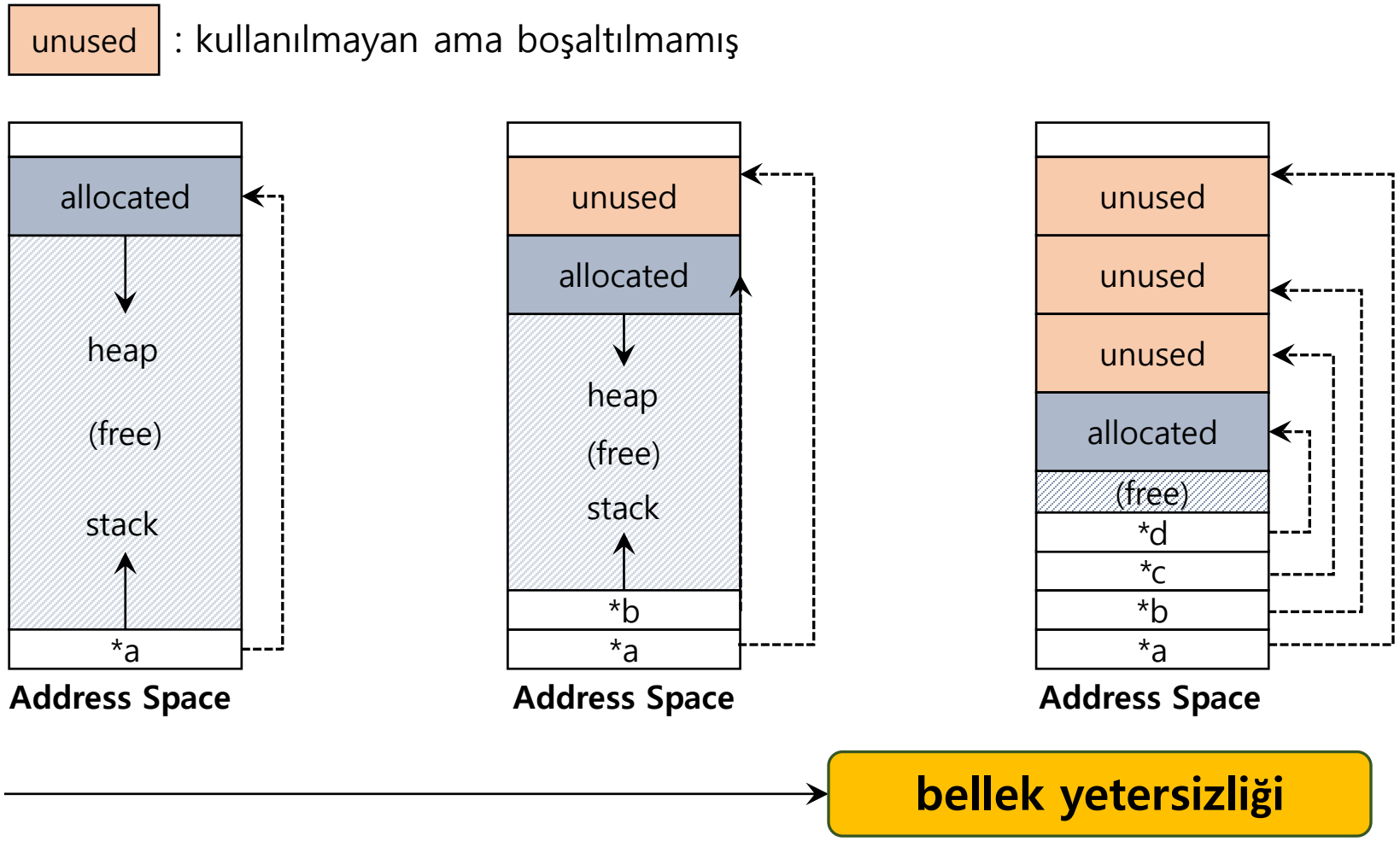
Address Space



Address Space

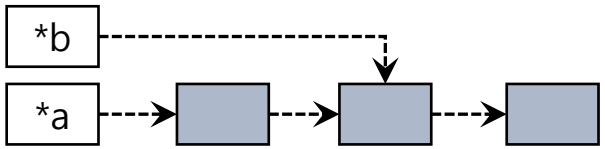
Belleği boşaltmayı unutmak (Memory Leak)

Bir programın yetersiz bellekten dolayı sonlandırılması gerekebilir.

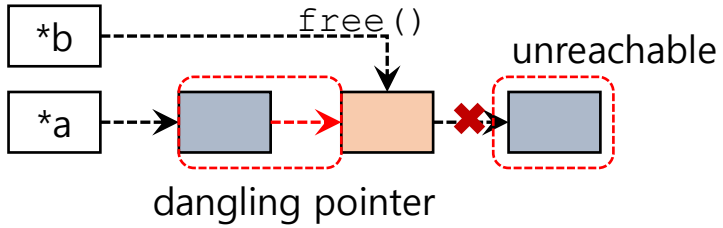


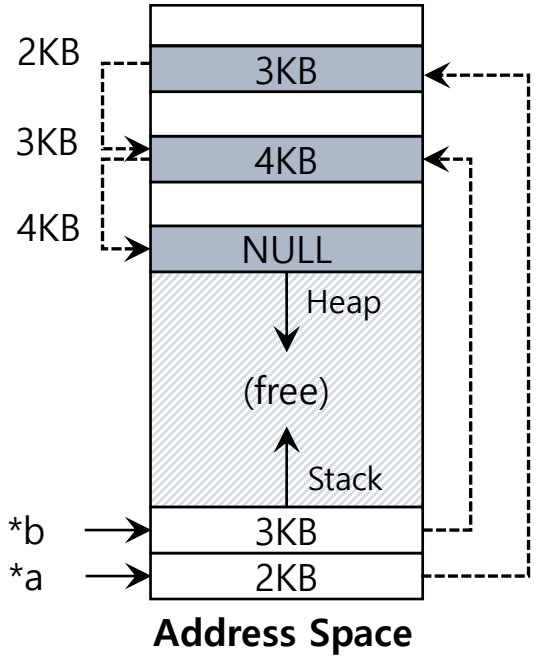
İş bitmeden belleği boşaltmak (Dangling Pointer)

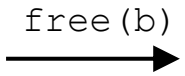
Dangling: Askıda kalan

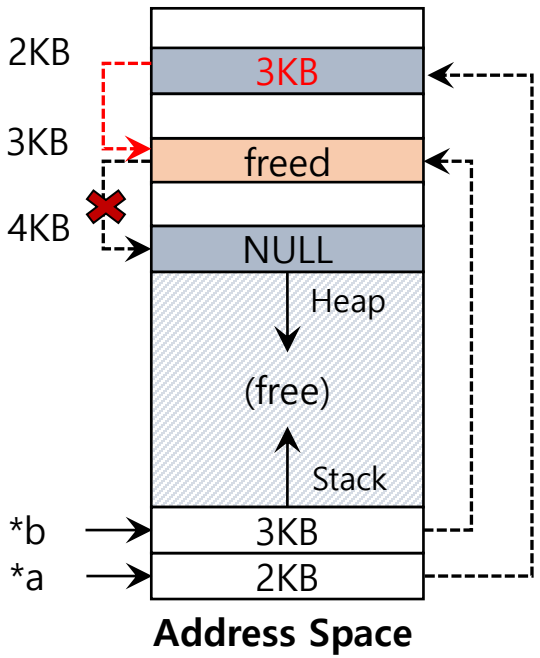


Bir program, geçersiz bir işaretçi ile belleğe erişiyor



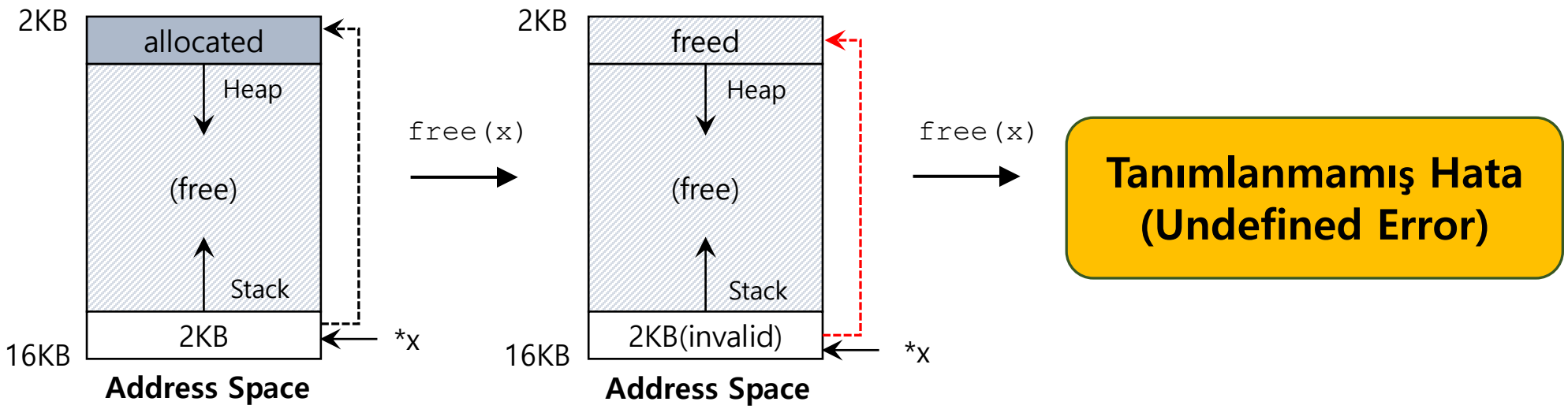






Belleği tekrar tekrar boşaltmak

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



Diğer Bellek API'leri: calloc()

```
#include <stdlib.h>

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

- Öbekte bellek alanı tahsis et ve dönmeden önce sıfırla
- Argüman
  - `size_t num`: kaç adet blok tahsis edileceği
  - `size_t size`: her bloğun boyutu (byte cinsinden)
- Dönüş
  - Başarılı: `calloc` ile tahsis edilen bellek alanını işaret eden void gösteren türünde işaretçi (pointer)
  - Başarısız: boş işaretçi (null pointer)

Diğer Bellek API'leri: realloc()

```
#include <stdlib.h>

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

- Tahsis edilmiş bellek alanının büyüklüğünü değiştirir
  - `realloc` ile dönen işaretçi ya aynıdır ya da değişir
  - Argüman
    - `void *ptr`: `malloc, calloc` veya `realloc` ile tahsis edilmiş bellek alanını gösteren işaretçi
    - `size_t size`: Yeni tahsis edilen bellek alanının (byte cinsinden) boyutu
  - Dönüş
    - Başarılı: Void türündeki bellek alanını gösteren işaretçi
    - Başarısız: boş işaretçi (Null pointer)

malloc() ve free() Sistem Çağrıları mıdır?

Sistem Çağrıları

```
#include <unistd.h>

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

- `malloc` kütüphane çağrısı (gerekirse) `brk` sistem çağrısını kullanır.
  - `brk`: program'ın *break* değerini genişletir.
    - *break*: Adres uzayında **öbek sonunun (end of the heap)** yerini değiştirir.
  - `sbrk`: `brk` ile benzerdir .
  - Program geliştiriciler hiçbir zaman doğrudan `brk` veya `sbrk` çağrısı yapmamalıdırlar.

15. Adres Çevrimi (Address Translation)

İşletim Sistemleri: Üç Basit Parça

Verimli ve Kontrollü Bellek Sanallaştırma

- Bellek sanallaştırma, verimlilik ve kontrol için **sınırlı doğrudan yürütme** (limited direct execution) olarak bilinen isismlendirdiğimiz yönteme benzer bir strateji kullanır.
- Bellek sanallaştırmada verimlilik ve kontrol, **donanım desteği** ile sağlanır.
  - Örnek: yazmaçlar, TLB (Translation Look-aside Buffers), sayfa tablosu (page-table)

Adres Çevrimi

- Donanım, sanal bir adresi fiziksel adrese dönüştürür.
  - Ulaşılmak istenen veri, bir fiziksel adreste saklanmaktadır.
- İşletim Sistemi donanımı kurmak ve ayarlamak için kilit noktalarda devreye girer.
  - Bellek yönetimini İşletim Sistemi yapar.

Örnek: Adres Çevrimi

- C dilinde bir kod parçası:

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

  - Bellekten bir değer yükle **(Load)**
  - Değeri üç artır
  - Değeri tekrar belleğe al **(Store)**

Örnek: Adres Çevrimi (Devam)

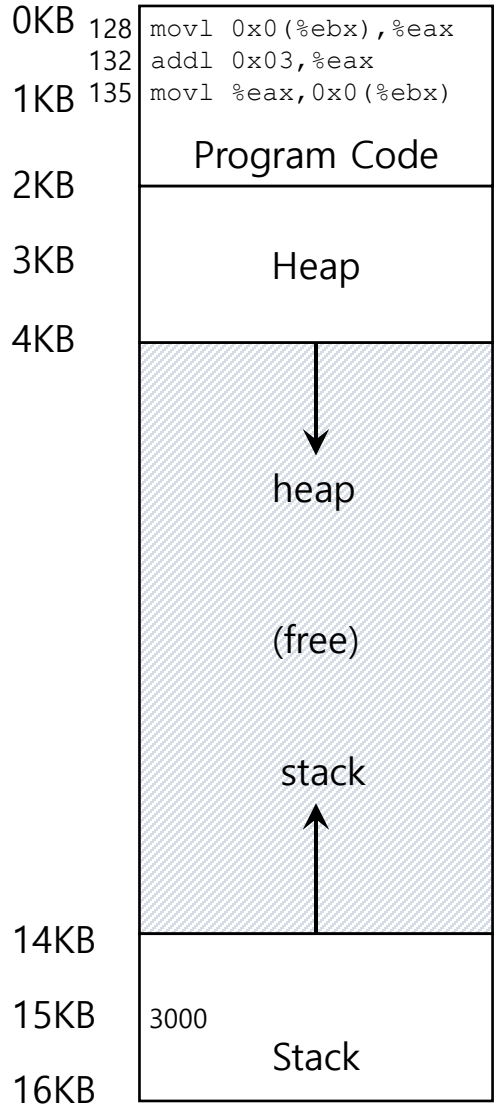
- Aynı kod Assembly dilinde:

```
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
```

  - ‘x’ değerinin adresinin ebx yazmacında olduğunu kabul edelim.
  - Bu adresteki değeri eax yazmacına yükle (**Load)**
  - eax yazmacına üç ekle.
  - eax değerini tekrar belleğe al (**Store)**

Örnek: Adres Çevrimi (Devam)

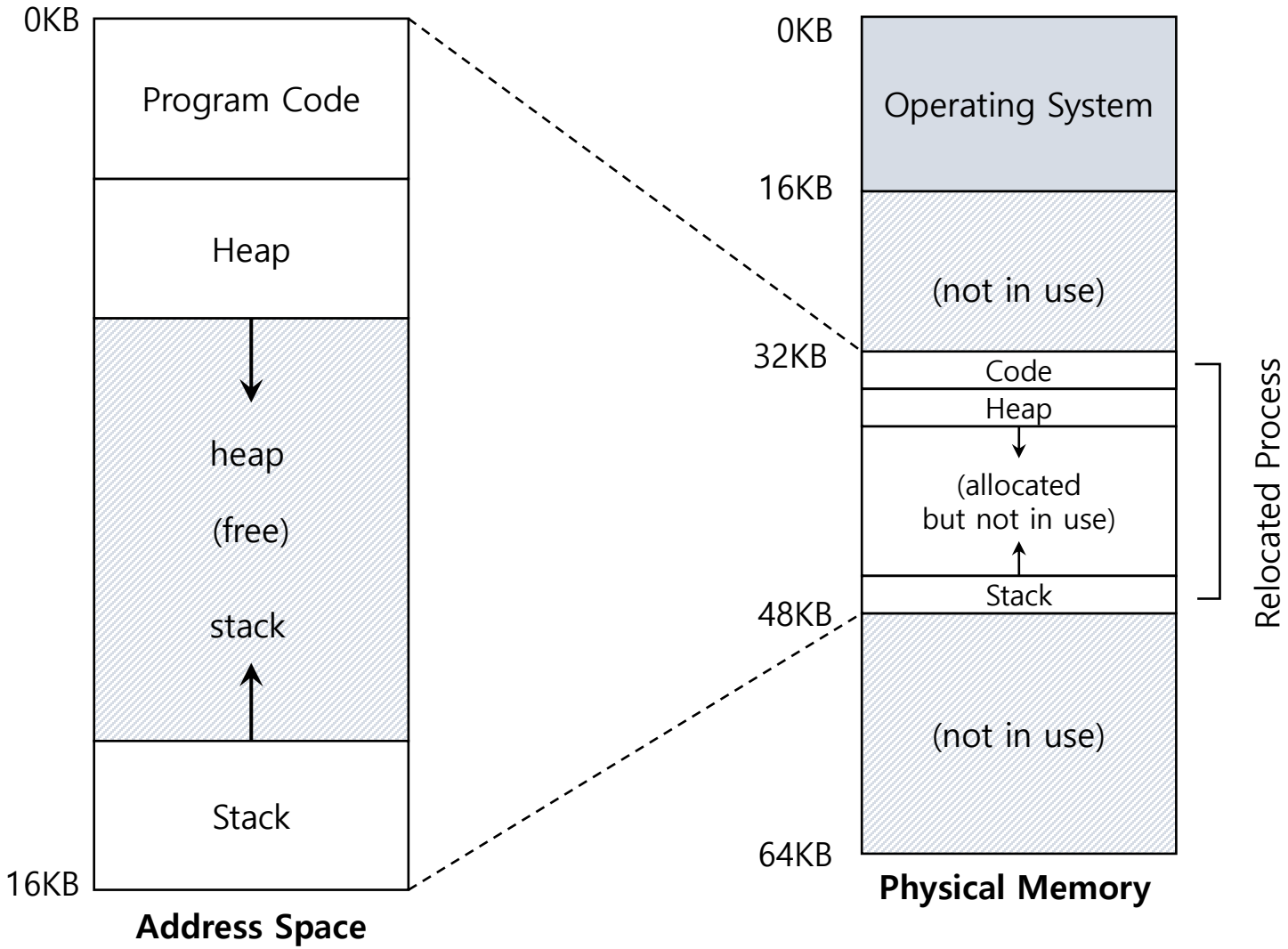
- adres 128’deki buyruğu getir (Fetch)
- Buyruğu çalıştır (adres 15KB’dan değer yükle)
- adres 132’deki buyruğu getir (Fetch)
- Buyruğu çalıştır (bellek referansı yok)
- adres 135’deki buyruğu getir (Fetch)
- Buyruğu çalıştır (değeri adres 15KB’daki belleğe al)



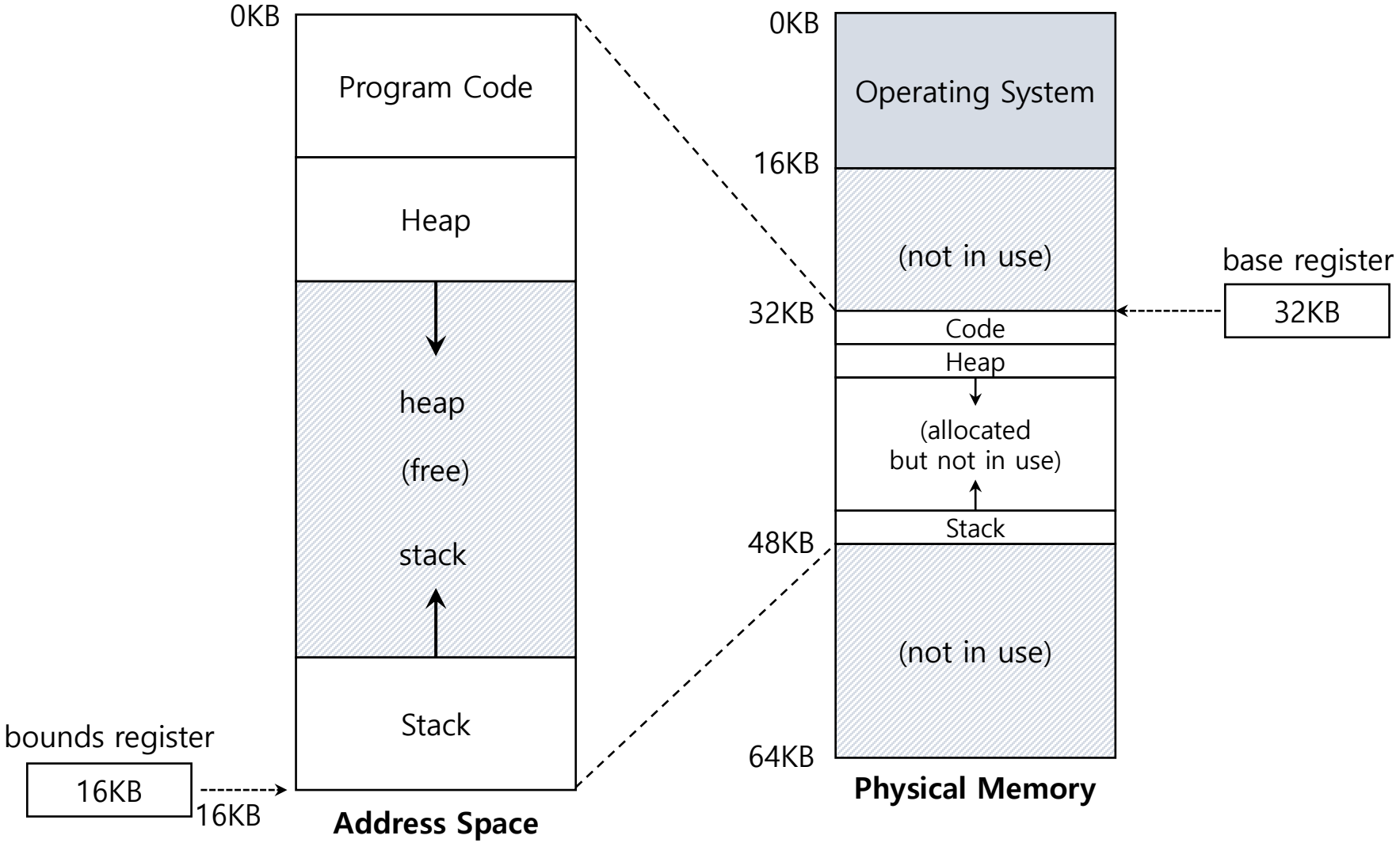
Adres Uzayını Fiziksel Belleğe Eşleme

- İşletim sistemi, işlemi adres 0'a değil, fiziksel bellekte başka bir yere yerleştirmek isterse ne olur?
  - Adres Uzayının (Address Space) adres 0'da başladığını hatırlayın.

Bir İşlem Var İse?



Taban ve Sınır Yazmacı (Base and Bounds Register)



Dinamik (Donanım Tabanlı) Çevrim

- Bir program çalışmaya başladığında, işletim sistemi bir işlemin fiziksel bellekte nereye yüklenmesi gerektiğine karar verir.
  - Taban (**base**) yazmacına ilgili değeri yükler.

$$physical\ address = virtual\ address + base$$

  - Her sanal adres sınır (**bound**) değerinden küçük olmalı ve pozitif bir değer almalıdır.

$$0 \leq\ virtual\ address < bounds$$

Adres Çevrimi

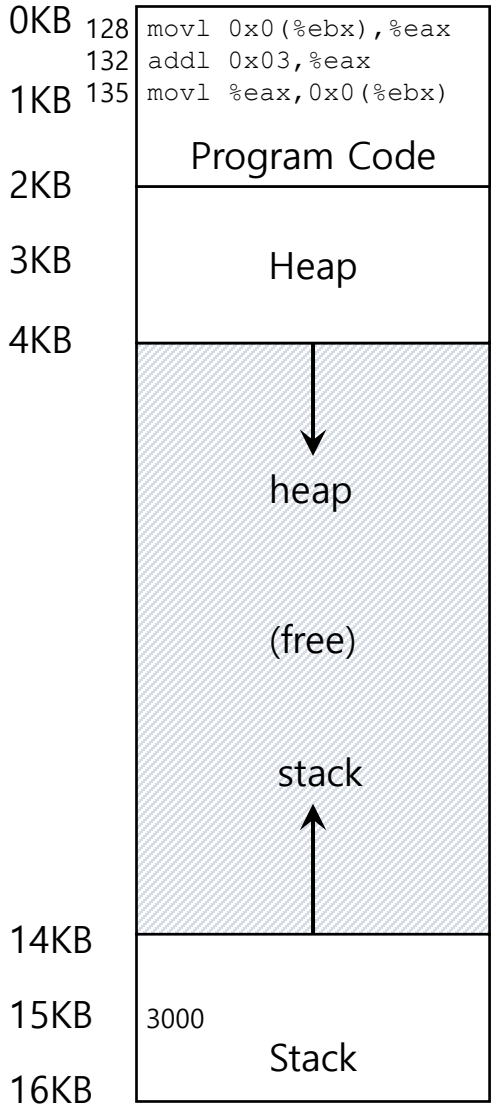
```
128: movl 0x0(%ebx), %eax
```

- Adres 128'dan buyruğu getir (**Fetch)**

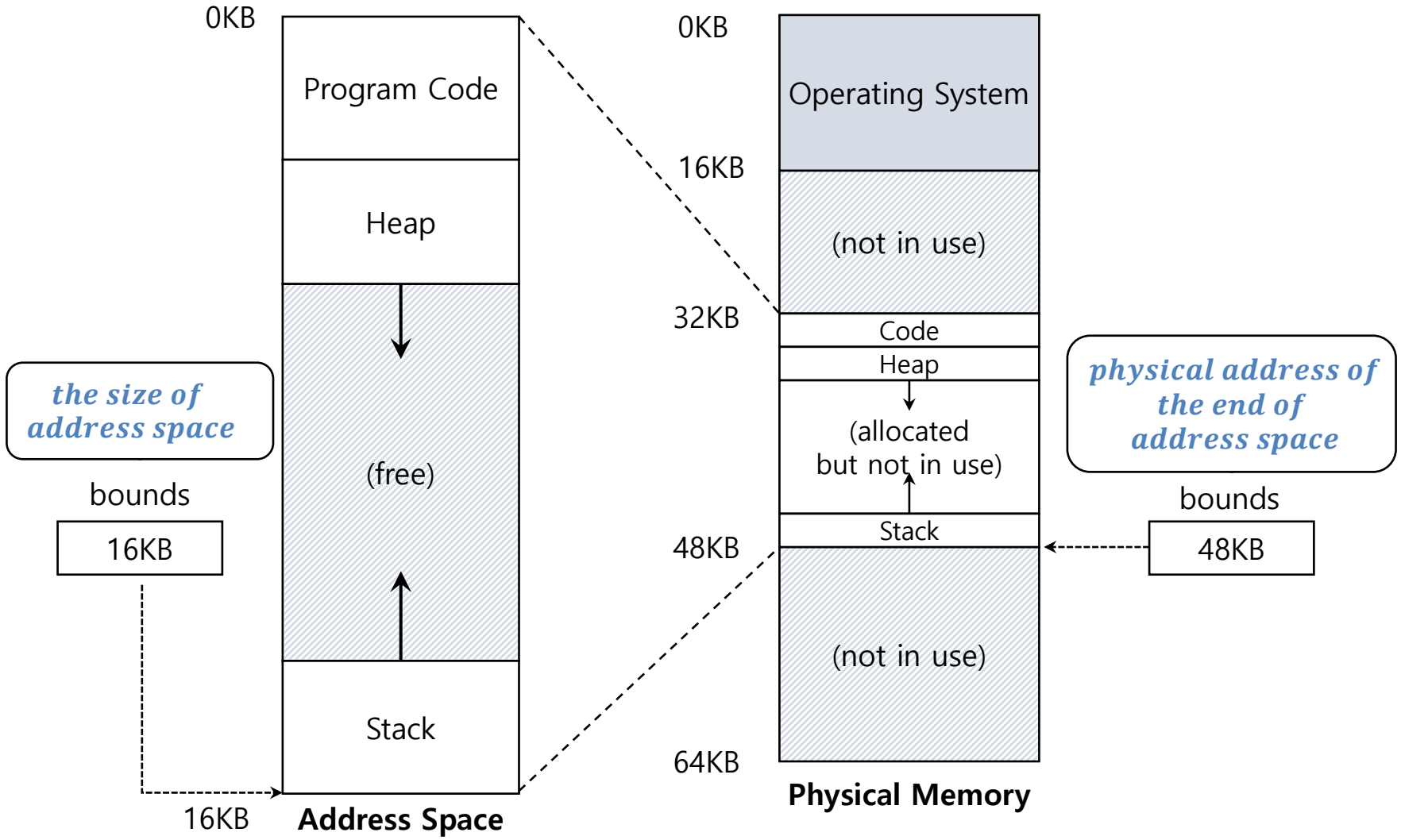
$$32896 = 128 + 32KB(base)$$

- Buyruğu çalıştır
  - Adres 15 KB'dan yükle **(Load)**

$$47KB = 15KB + 32KB(base)$$



Sınır Yazmacını Kullanmanın İki Yolu

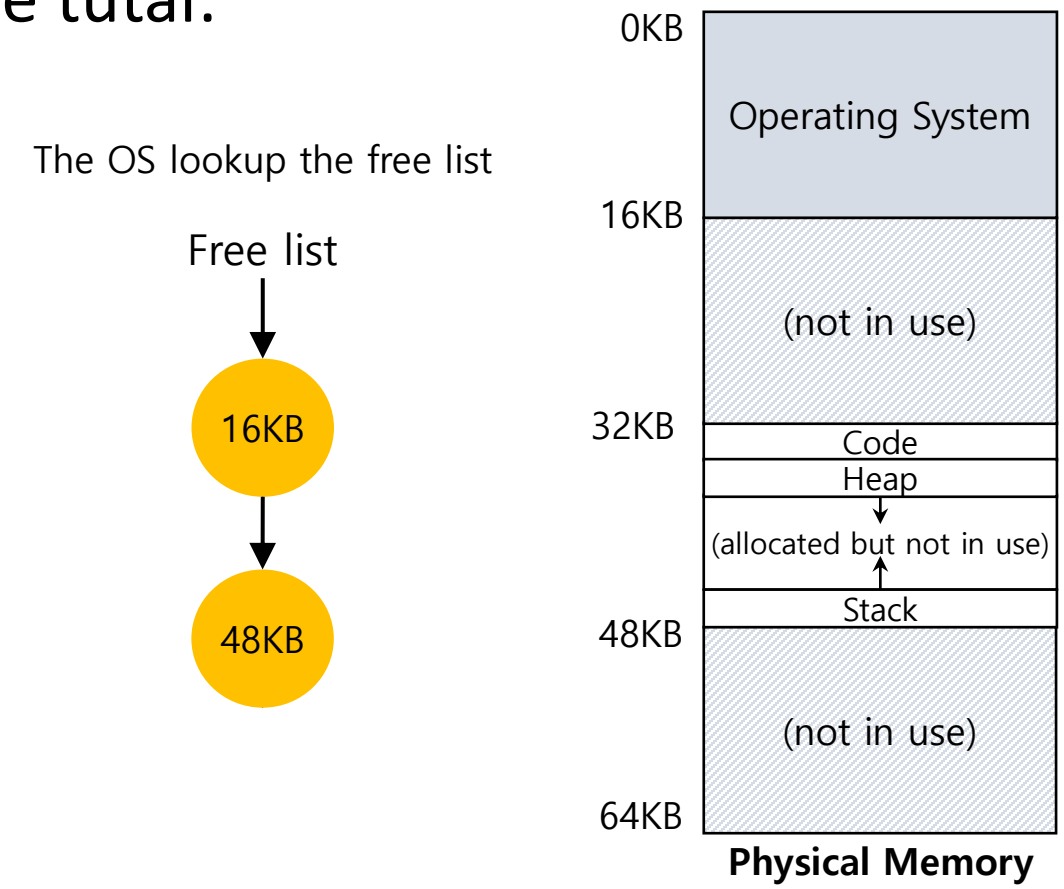


Bellek Sanallaştırmada İşletim Sistemi Konuları

- İşletim sistemi **taban ve sınır** yaklaşımını uygulamakla sorumludur.
- Üç kritik anda devreye girmelidir.
  - İşlem çalışmaya başladığında:
    - Fiziksel bellekte adres uzayı için yer bulmalıdır.
  - İşlem sonlandığında:
    - Belleğe sonrasında kullanmak için geri almalıdır.
  - Context switch olduğunda:
    - Mevcut taban ve sınır yazmaçlarını kaydetmeli ve yeni değerleri yüklemelidir.

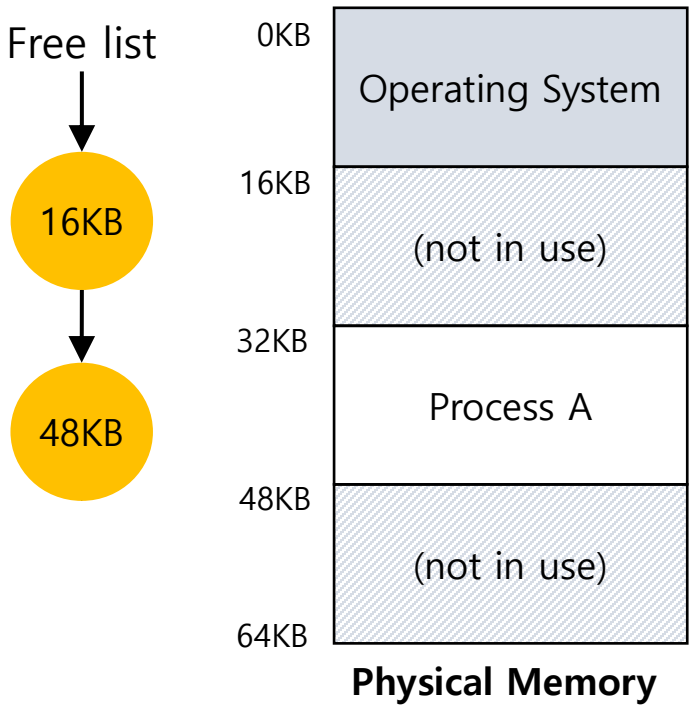
Bir İşlem Çalışmaya Başladığında:

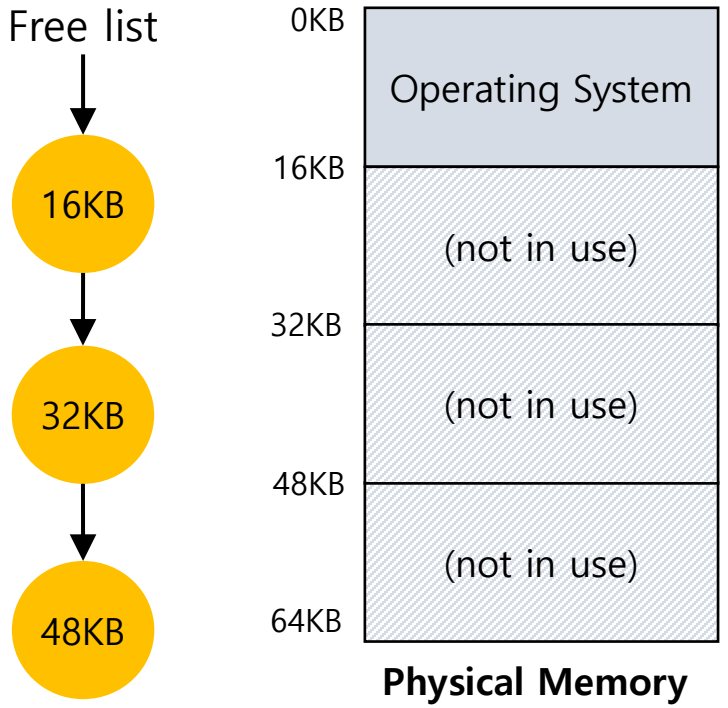
- İşletim Sistemi yeni adres uzayı için yer bulmalıdır.
- Bu amaçla hangi fiziksel adres aralıklarının boş olduğunu gösteren bir liste tutar.



Bir İşlem Sonlandığında:

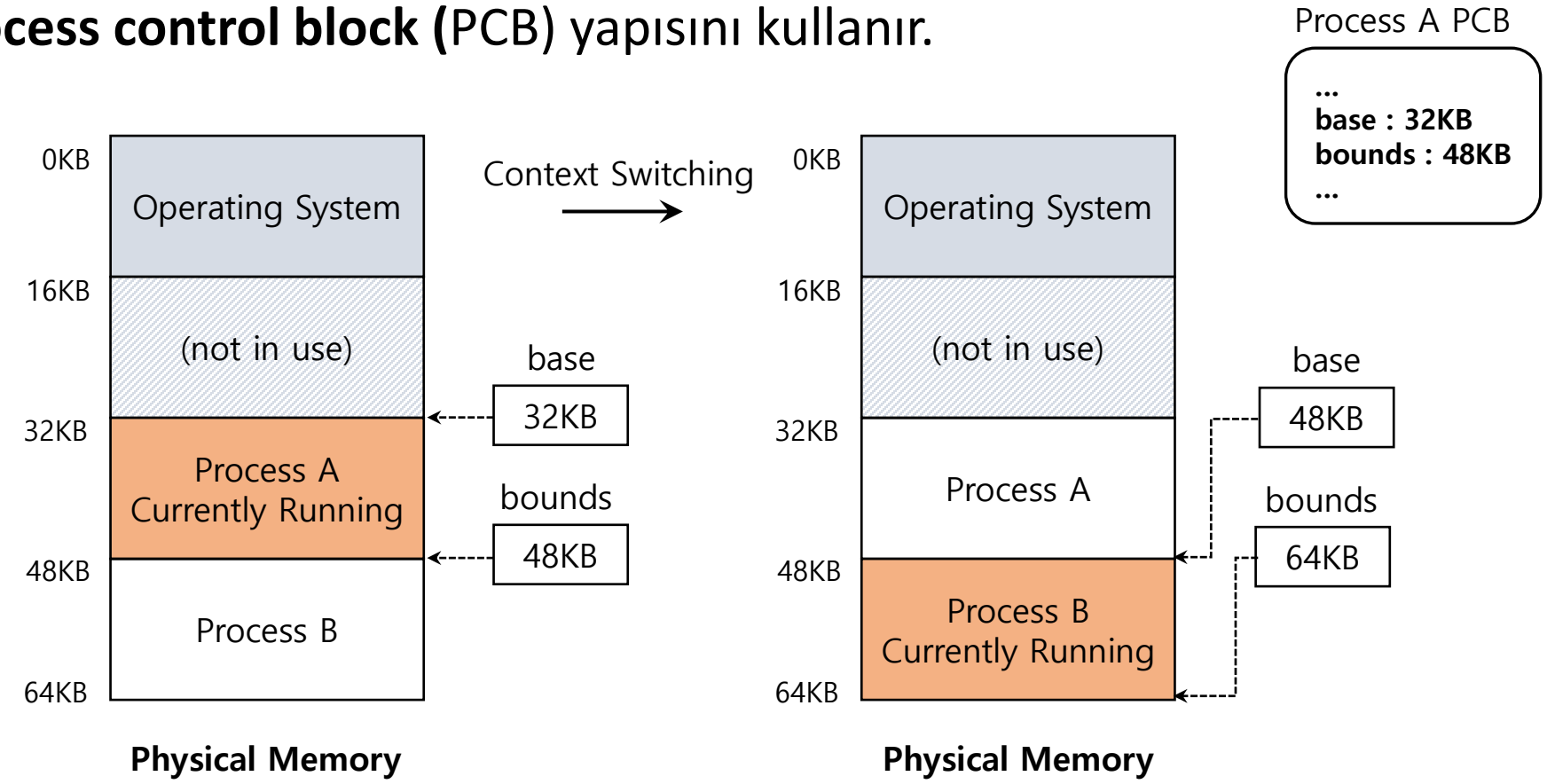
- İşletim Sistemi bellek aralığını tekrar boş listesine ekler.





Context Switch Olduğunda:

- İşletim Sistemi mevcut taban ve sınır yazmaçlarını kaydeder ve yeni değerleri yükler.
  - **process control block (**PCB) yapısını kullanır.



taban ve sınır yaklaşımı bizler için yeterli midir?