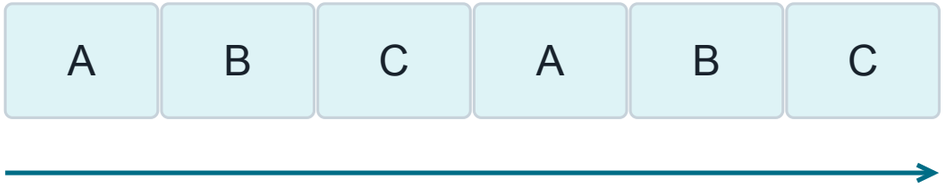
CPU'yu paylaştık; sırada bellek var

The Abstraction: Address Spaces

Scheduler, işlemciyi süreçler arasında paylaştırır. O anda çalışmayan sürecin kodu ve verileri RAM'de kalabilir. Bellek sanallaştırma, aynı fiziksel belleği kullanan süreçlerin birbirinden korunmasını sağlar.

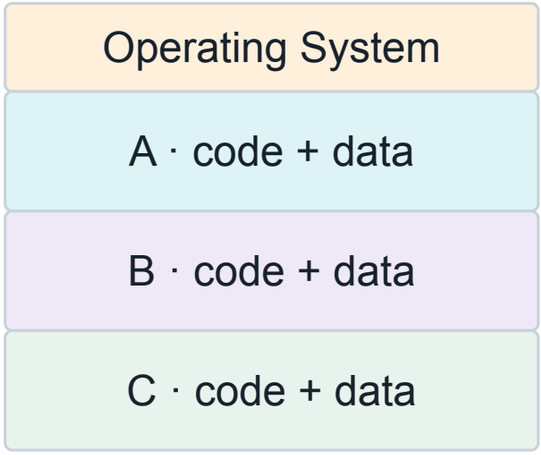
CPU · over time



Next process runs.

Context switch ≠ copying all memory

RAM · at one moment

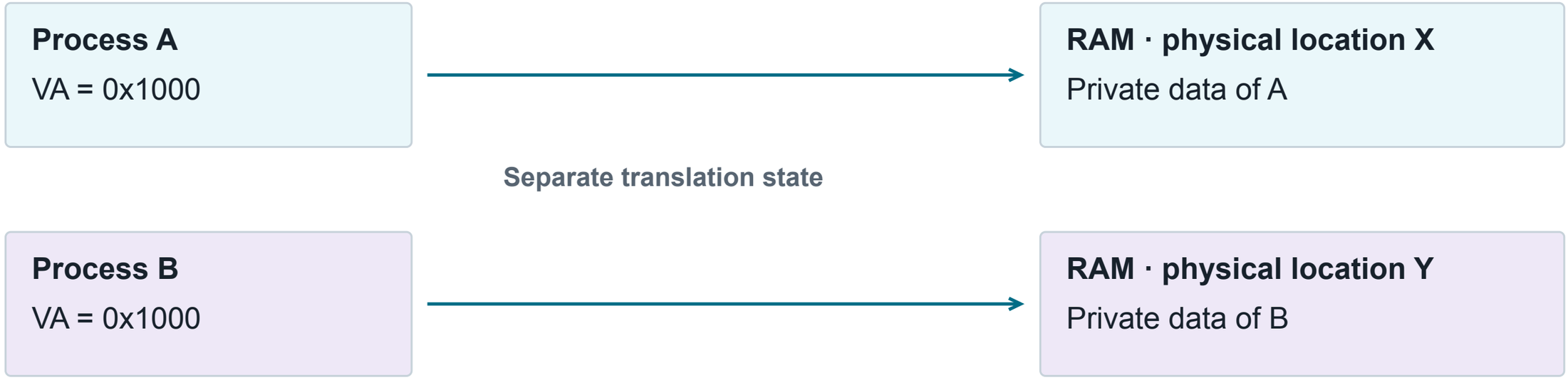


Sonuç: İşlemcide çalışan süreç değişse de diğer süreçlerin verileri bellekte kalabilir.

Programın kullandığı adres sanaldır

The Address Space

Bir pointer, sürecin adres uzayındaki bir konumu belirtir. Aynı sayısal adres başka bir süreçte farklı bir fiziksel konuma karşılık gelebilir. Bu karşılığı, her sürece ait adres çevirme bilgileri belirler.



Sonuç: Programın yazdırdığı adres sanaldır; fiziksel RAM adresi değildir.

Bu soyutlama bize ne kazandırır?

Goals

Program kendi adresleriyle çalışır; diğer süreçlerin RAM'de nereye yerleştirildiğini hesaplamaz. İşletim sistemi ve donanım, bu görünümü düşük maliyetle ve güvenli biçimde sağlar. Programlama kolaylığı, verimlilik ve koruma birlikte hedeflenir.

Transparency

Use my own address space.
No need to know
physical placement.

Efficiency

Low translation overhead.
Manage memory resources
according to demand.

Protection

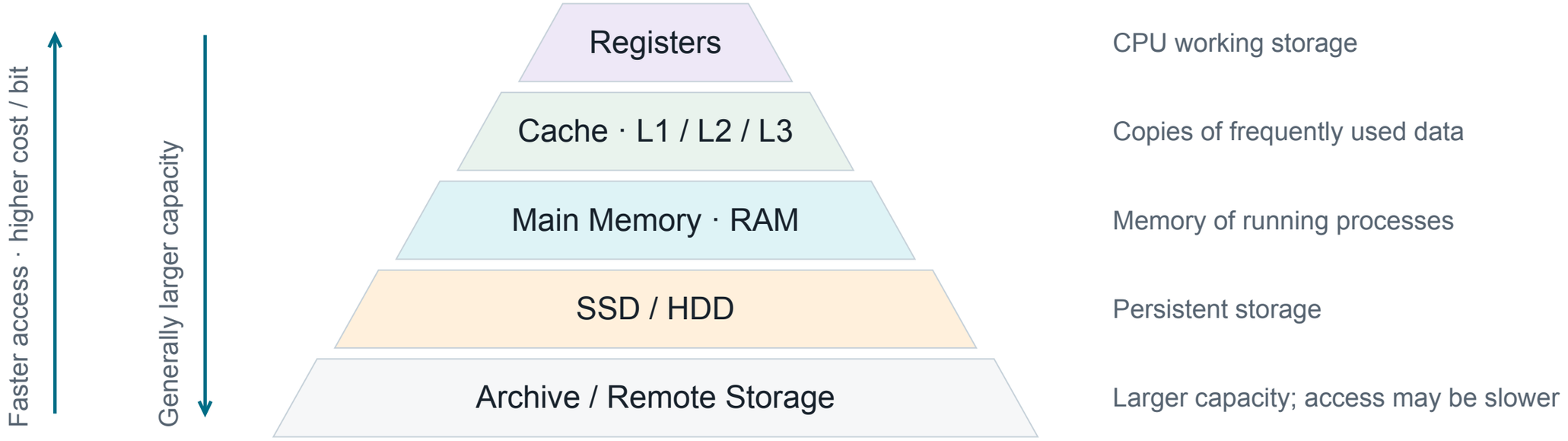
Invalid access from A
must not overwrite
private data of B.

Sonuç: Kullanışlı bir bellek görünümü, verimli paylaşım ve süreçler arasında koruma gerekir.

Bellek hiyerarşisi: neden her şey RAM değil?

Memory Hierarchy

İşlemciye yakın bellek katmanları genellikle daha hızlı ve bit başına daha pahalıdır. Daha büyük veriler için geniş kapasiteli, ancak daha yavaş katmanlar kullanılır. RAM ile kalıcı depolama arasındaki hız farkı, bellek yönetimi kararlarını etkiler.



Sonuç: Cache, RAM ve SSD donanım katmanlarıdır; stack ve heap ise programın belleği kullanma biçimiyle ilgilidir.

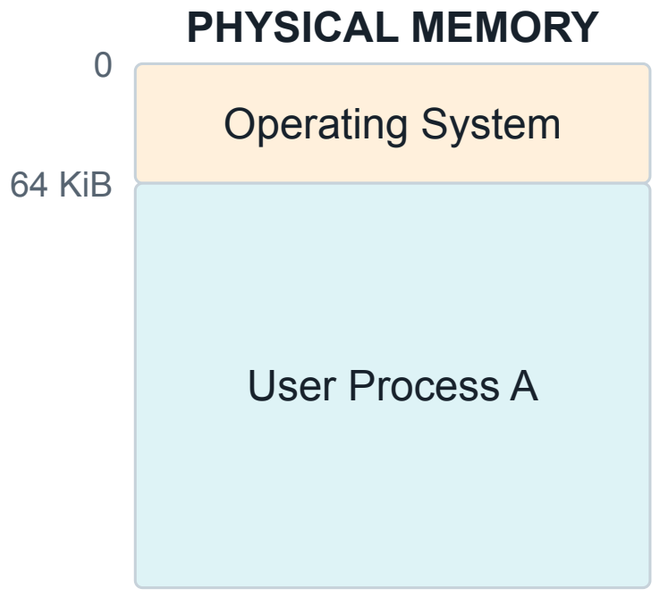
İlk model: bellekte tek kullanıcı programı

Early Systems

İlk basit modelde işletim sisteminin yanında tek bir kullanıcı programı bulunur. Her kısa işlemci geçişinde programın bütün belleğini diske taşıyıp geri getirmek pahalıdır. Birden fazla süreci RAM'de tutmak bu maliyeti azaltır.

One user process

Moving a complete memory image to storage on every CPU switch is expensive for time sharing.

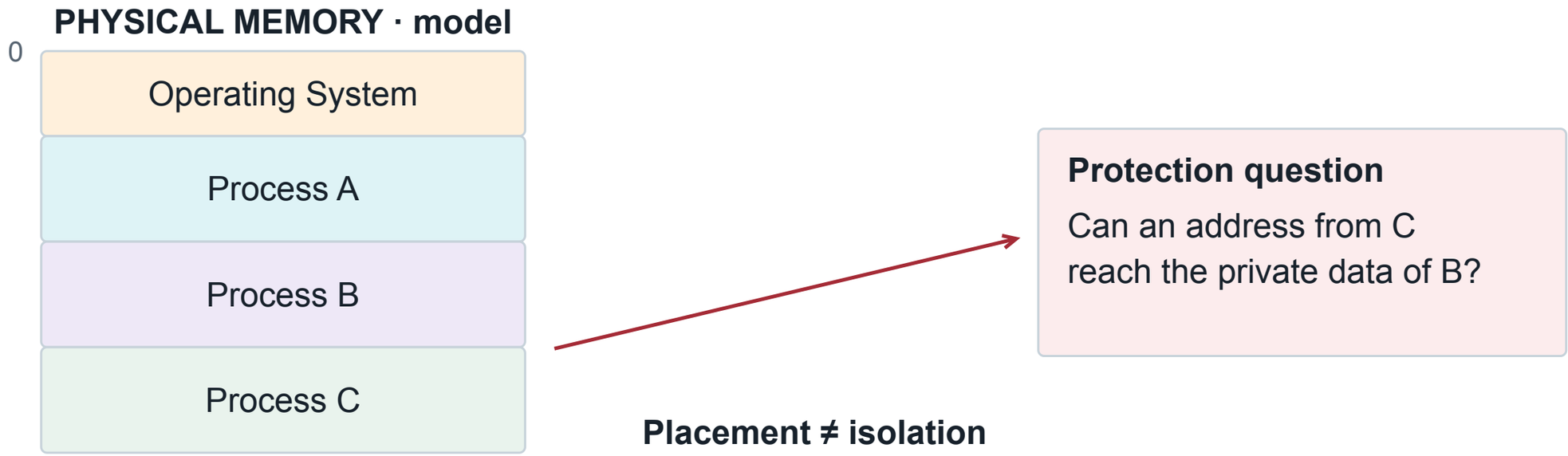


Sonuç: Hızlı geçiş için birden fazla sürecin kodu ve verileri RAM'de kalabilir.

Birden fazla süreç RAM'de kalır

Multiprogramming and Time Sharing

A, B ve C süreçleri RAM'de dururken işlemci aralarında geçebilir. Ancak ayrı yerlere yerleştirmek tek başına koruma sağlamaz. C'nin ürettiği bir adres B'nin özel verisini değiştirebiliyorsa yalıtım sorunu hâlâ çözülmemiştir.

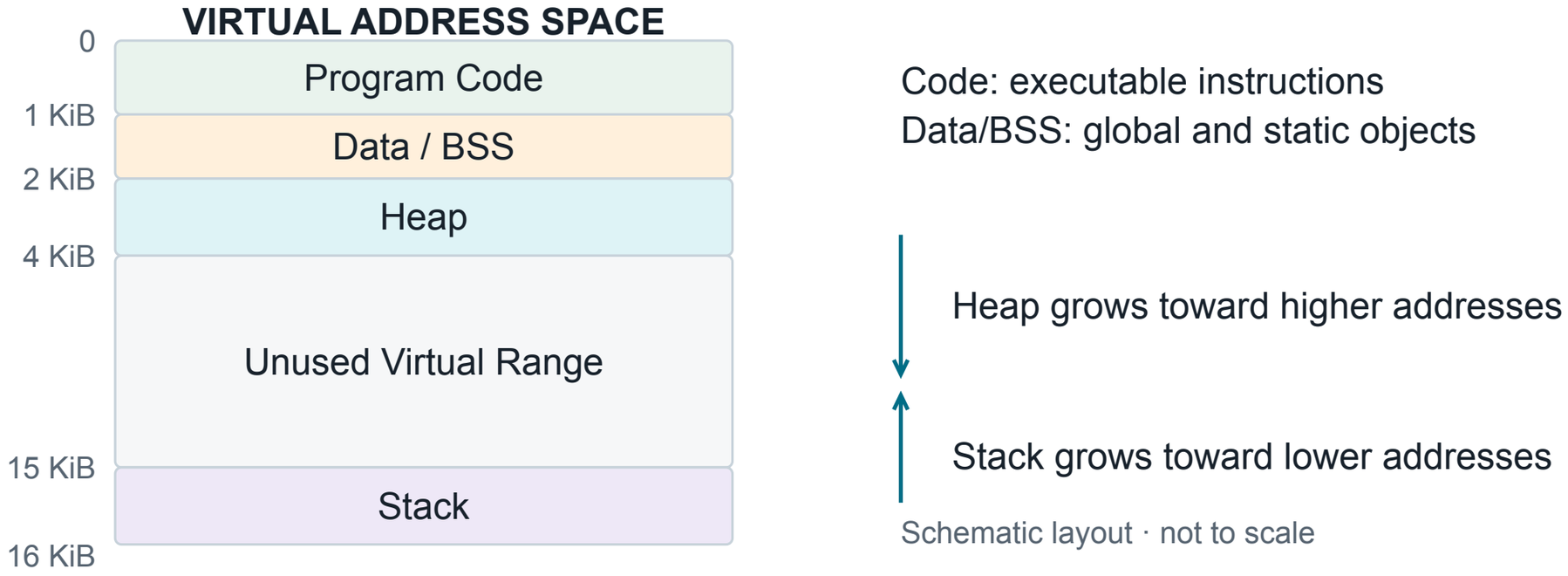


Sonuç: Aynı belleği paylaşan süreçlerin özel verileri birbirinden korunmalıdır.

Address space: sürecin gördüğü bellek düzeni

The Address Space

Sürecin gördüğü bellek düzeninde program komutları, global değişkenler, dinamik nesneler ve çağrı bilgileri bulunur. Buradaki 0–16 KiB aralığı küçük bir öğretim örneğidir; fiziksel RAM'in ilk 16 KiB'si anlamına gelmez.

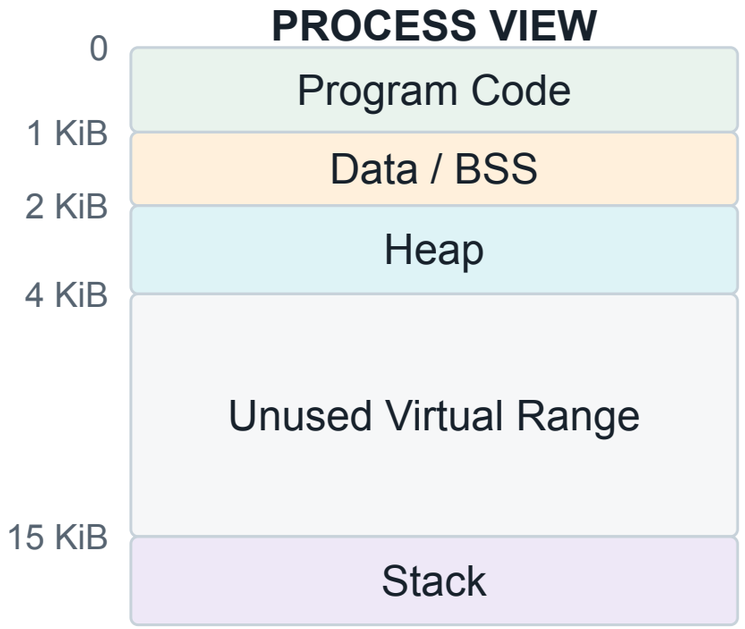


Sonuç: Adres uzayı, bir sürece sunulan sanal bellek görünümüdür.

Code, data, heap ve stack ne tutar?

The Address Space

Code, çalıştırılacak komutları; data/BSS, global ve statik verileri tutar. Heap, program çalışırken ayrılan dinamik nesneler için kullanılır. Stack modelinde yerel değişkenler ve fonksiyon çağrılarıyla ilgili bilgiler bulunur.



Code → instructions in main
Data → int global = 10;
Heap → dynamically allocated object
Stack → local, p, call frame

Heap and stack are address-space regions.

They are not separate hardware types.
Real maps contain additional regions.

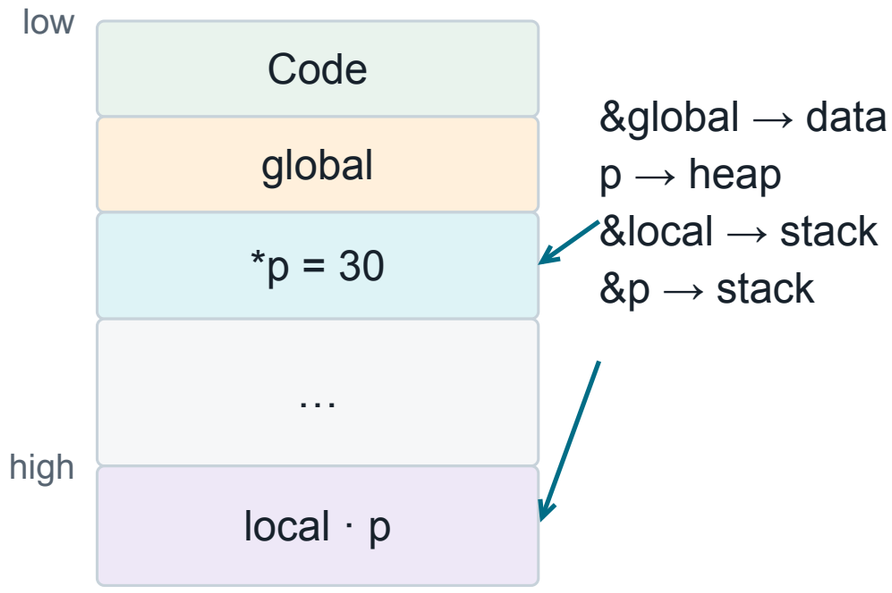
Sonuç: Heap ve stack farklı amaçlarla kullanılır; ikisi de sürecin sanal belleğindedir.

C programında hangi adresi yazdırıyoruz?

The Address Space

&global, global değişkenin; &local, yerel değişkenin adresidir. p, dinamik nesnenin adresini tutar. &p ise pointer değişkeninin kendi adresini verir. %p ile yazdırılan bu değerlerin hepsi sanal adrestir.

```
int global = 10;

int main(void) {
    int local = 20;
    int *p = malloc(sizeof *p);
    /* p != NULL kontrolünden sonra */
    printf("%p", (void *)&global);
    printf("%p", (void *)&local);
    printf("%p", (void *)p);
    printf("%p", (void *)&p);
}
```



Sonuç: p ile &p aynı nesnenin adresini göstermez.

0x ile başlayan sayı ne demek?

Reading Hexadecimal Addresses

0x öneki, sayının onaltılık düzende yazıldığını belirtir. Her basamak 4 bit temsil eder. Bir adresin kısa yazılması, pointer'ın az yer tuttuğu anlamına gelmez; baştaki sıfırlar gösterilmeyebilir.

One value, two notations

0x1000 = 4096 (decimal)
1 0 0 0 · hex
0001 0000 0000 0000 · bit

Pointer size vs. printed notation

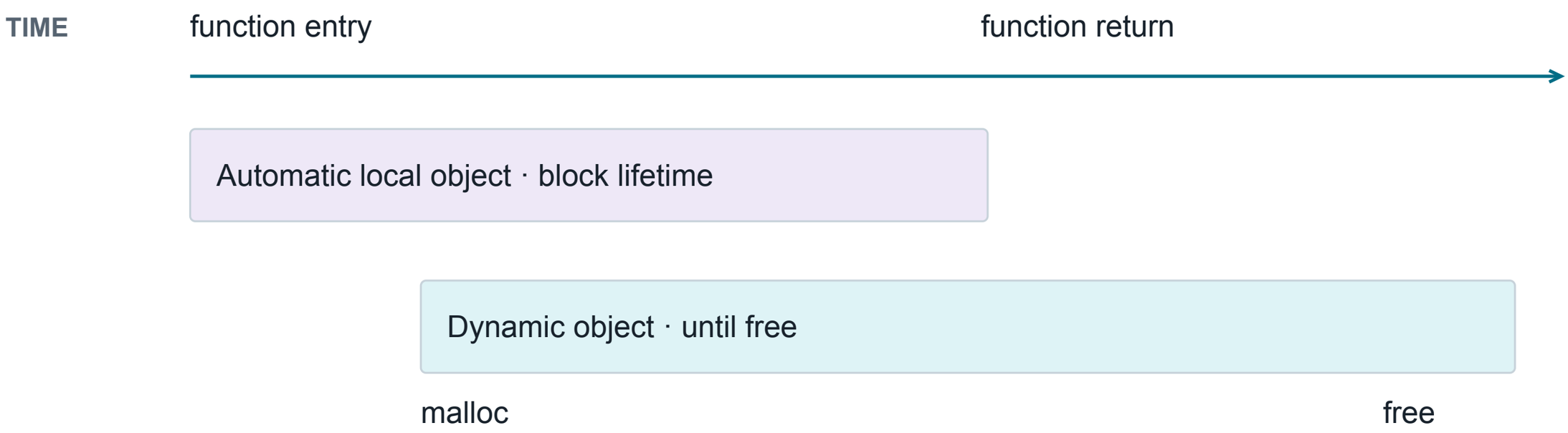
32-bit guest: pointer = 4 bytes
Typical 64-bit Linux: pointer = 8 bytes
int is often 4 bytes in both

Sonuç: Adresin yazılışı, pointer boyutu ve fiziksel RAM miktarı farklı bilgilerdir.

İki bellek türü: otomatik ve dinamik

Types of Memory

Yerel bir nesnenin yaşam süresi bulunduğu blokla sınırlıdır. malloc ile ayrılan dinamik nesne ise onu oluşturan fonksiyon döndüğünde kendiliğinden serbest kalmaz. Uygun zamanda free çağırmak programın sorumluluğudur.



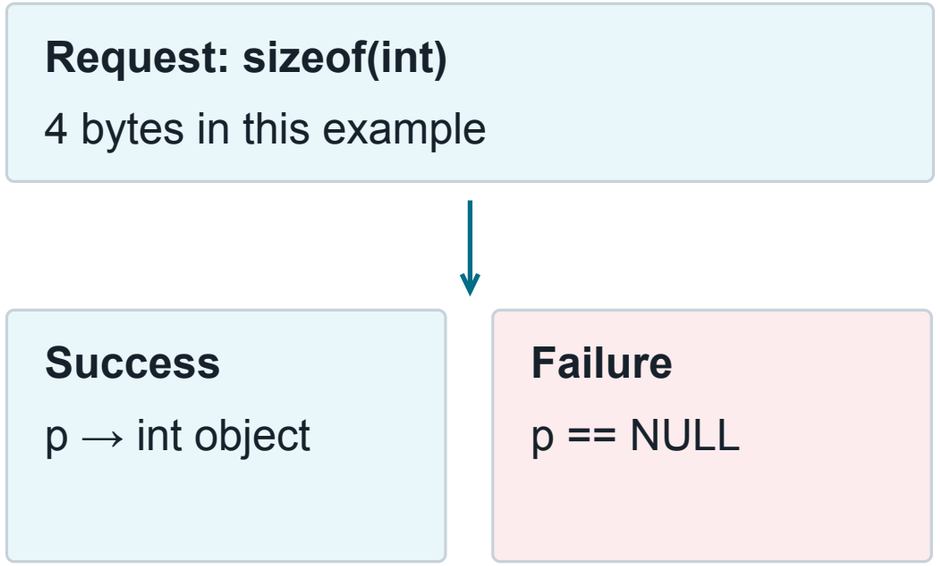
Sonuç: Yerel pointer ile gösterdiği dinamik nesne farklı süreler boyunca yaşayabilir.

malloc: yer iste, sonucu kontrol et

The malloc() Call

malloc, istenen byte sayısını alır. Başarılıysa ayrılan alanın başlangıç adresini, başarısızsa NULL döndürür. Alanın ayrılmış olması, içine geçerli bir başlangıç değeri yazıldığı anlamına gelmez.

```
int *p = malloc(sizeof *p);
if (p == NULL) {
    return 1;
}
*p = 30;
/* ... */
free(p);
```



Sonuç: Alanı ayır → NULL kontrolü yap → ilk değeri yaz → kullan → free ile bırak.

sizeof(pointer), ayrılan alanın boyutu değildir

The malloc() Call

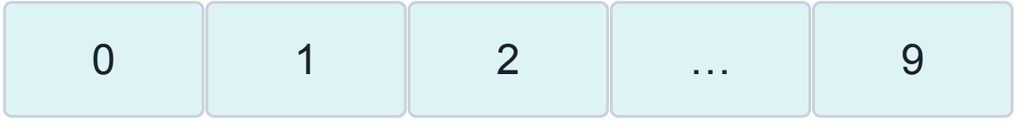
p bir pointer, a ise 10 elemanlı bir dizidir. sizeof p pointer değişkeninin, sizeof a dizinin tamamının boyutunu verir. malloc ile ayrılan alanın uzunluğu ayrıca tutulmalıdır.

```
int *p = malloc(10 * sizeof *p);
int a[10];

sizeof *p;  /* int boyutu: 4 */
sizeof p;   /* pointer: 4 veya 8 */
sizeof a;   /* bu kapsamda: 40 */

free(p);
```

p: pointer object

sizeof p = 4 or 8 bytes

a: array of 10 int objects



sizeof a = 10 × 4 = 40 bytes

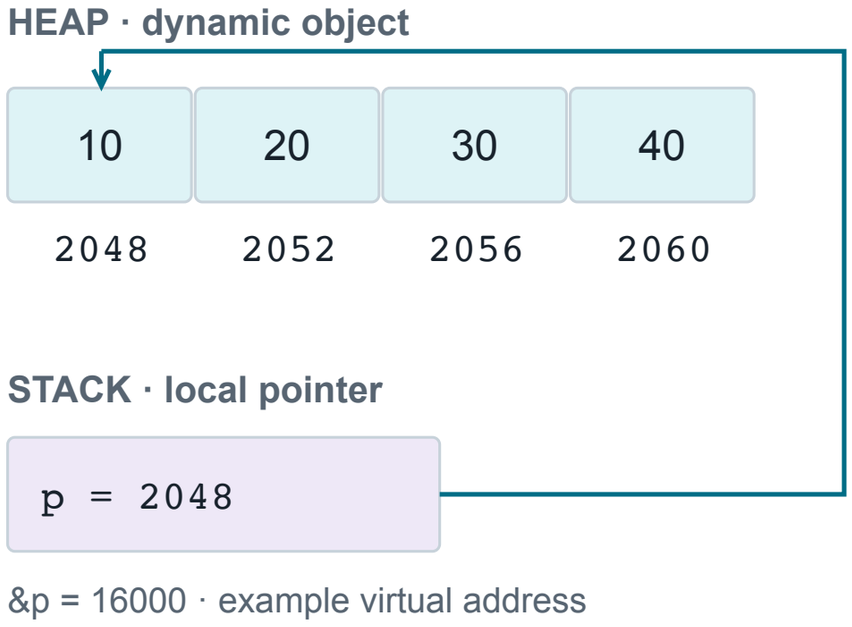
Sonuç: 10 int için yer ayırmak, pointer değişkenini 10 int büyüklüğüne getirmez.

Tek satır, iki ayrı nesne

Types of Memory

int *p bildirimi, adres tutacak bir pointer değişkeni oluşturur. malloc(4 * sizeof *p) ise dört int için ayrı bir alan ister. Tahsis başarılı olduğunda bu alanın başlangıç adresi p'ye yazılır.

```
int *p = malloc(4 * sizeof *p);
if (p == NULL) return 1;

p[0] = 10;
p[1] = 20;
p[2] = 30;
p[3] = 40;
```



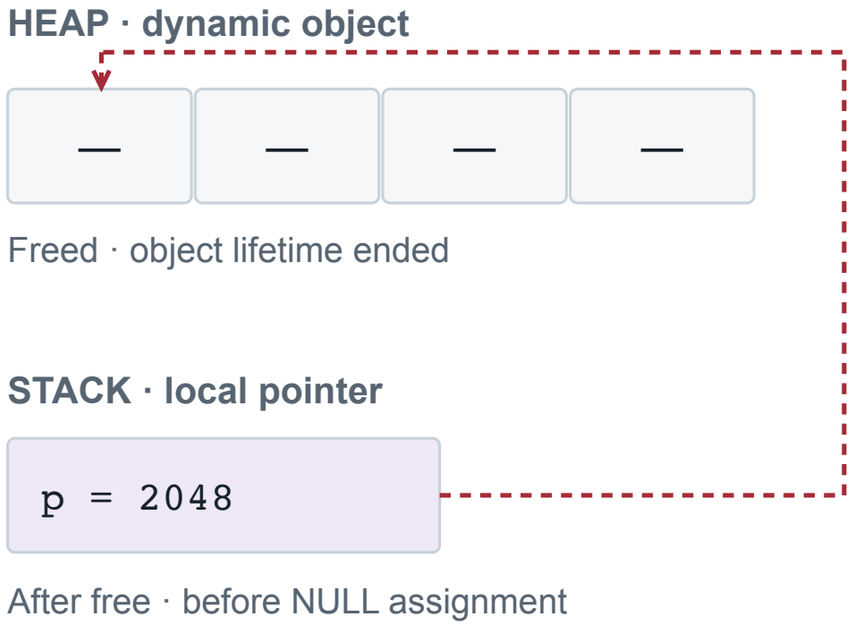
Sonuç: Pointer değişkeni ile gösterdiği dinamik alan ayrı nesnelerdir.

free: nesnenin kullanım süresi biter

The free() Call

free(p), p'nin gösterdiği alanı serbest bırakır. Pointer değişkeni p var olmaya devam eder, ancak eski nesneye erişim artık geçerli değildir. free, p'yi kendiliğinden NULL yapmaz.

```
free(p);
/* p[0] artık okunamaz. */

p = NULL;
/* p değişkeni hâlâ var. */
```



Sonuç: Alanı serbest bırakmak, pointer değişkenini silmek değildir.

Bellek yönetiminde sık yapılan hatalar

Common Errors

Hatalar yalnızca yanlış adrese erişmekten doğmaz. Alanın yeterli büyüklükte olması, kullanılmadan önce başlatılması ve doğru zamanda serbest bırakılması da gerekir.

OSTEP'teki hata başlığı	Sorun
Forgetting To Allocate Memory	Pointer var, ancak göstereceği alan ayrılmamış.
Not Allocating Enough Memory	Ayrılan alan verinin tamamına yetmiyor.
Forgetting to Initialize Allocated Memory	Bellek okunmadan önce başlangıç değeri yazılmamış.
Forgetting To Free Memory	Kullanımı biten alan serbest bırakılmamış.
Freeing Memory Before You Are Done With It	Alan, kullanımı tamamlanmadan serbest bırakılmış.
Freeing Memory Repeatedly	Aynı tahsis ikinci kez serbest bırakılıyor.
Calling free() Incorrectly	free çağrısına geçersiz bir adres veriliyor.

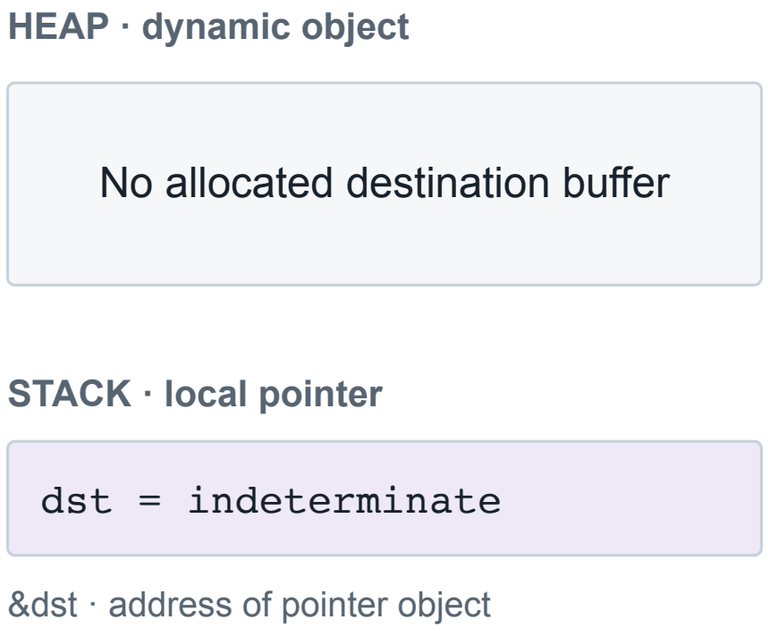
Sonuç: Doğru bellek kullanımı için alanın büyüklüğü, başlangıç değeri ve yaşam süresi birlikte düşünülür.

Hata: pointer var, hedef alan yok

Forgetting To Allocate Memory

char *dst bildirimi, karakterlerin kopyalanacağı alanı ayırmaz. Başlatılmamış dst üzerinden yazmak tanımsız davranıştır. Pointer tanımlamak ile onun göstereceği nesne için yer ayırmak farklı işlemlerdir.

```
const char *src = "hello";
char *dst;  /* hedef alan ayrılmadı */

strcpy(dst, src);  /* ERROR */
```



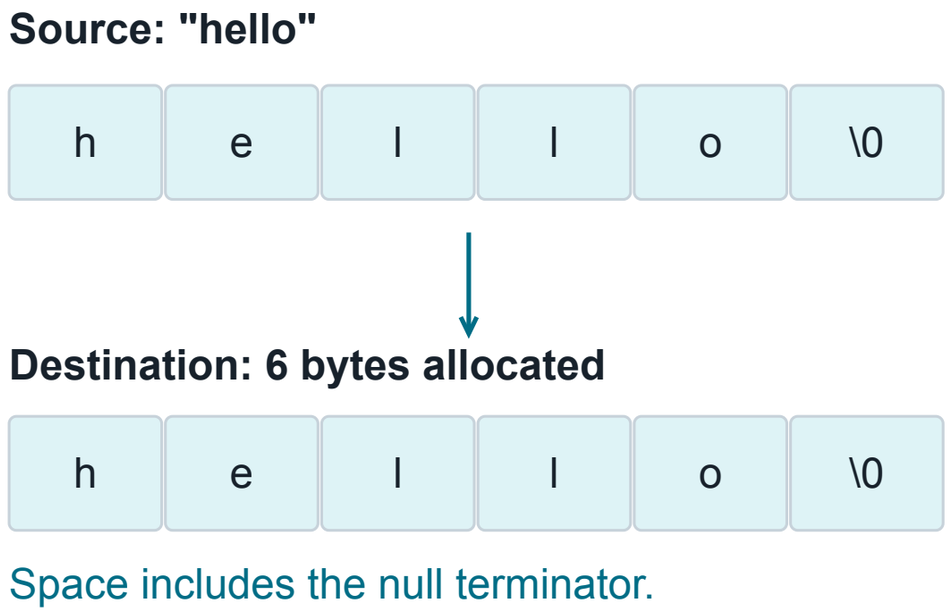
Sonuç: Bir pointer'ın varlığı, yazılabilir bir hedef alan bulunduğunu garanti etmez.

Düzeltme: karakterler ve sonlandırıcı için yer

Forgetting To Allocate Memory

"hello" beş görünür karakter içerir. C dizgesinin sonunda ayrıca \0 sonlandırıcısı bulunduğu için hedefte altı byte yer gerekir. Başarılı tahsis kontrolünden sonra kopyalama yapılabilir.

```
const char *src = "hello";
char *dst = malloc(strlen(src) + 1);
if (dst == NULL) return 1;

strcpy(dst, src);
/* dst kullanılır. */
free(dst);
```



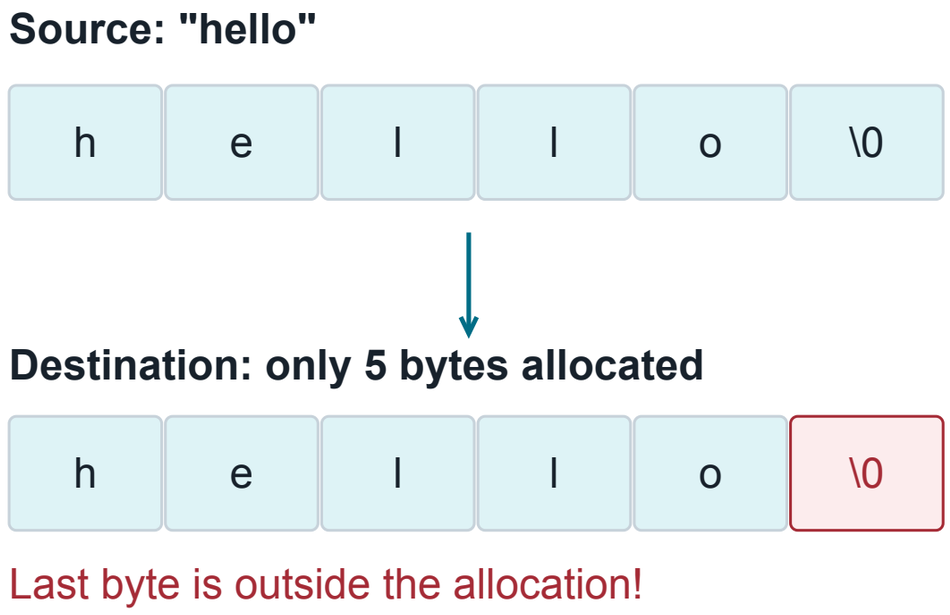
Sonuç: Dizge için gereken alan: strlen(src) + 1 byte.

Hata: beş byte'a altı byte yazmak

Not Allocating Enough Memory

Yalnızca strlen(src) kadar yer ayrılırsa dizge sonlandırıcısına alan kalmaz. Kopyanın son byte'ı ayrılan alanın dışına yazılır. Program doğru çıktı verse bile bu erişim hatalıdır.

```
char *dst = malloc(strlen(src));
if (dst == NULL) return 1;
strcpy(dst, src); /* HATA: +1 yok */
```



Sonuç: Doğru görünen çıktı, bellek erişiminin doğru olduğunu kanıtlamaz.

Hata: tahsis etmek, değer atamak değildir

Forgetting to Initialize Allocated Memory

malloc ile yer ayırmak, nesneye başlangıç değeri yazmaz. *p = 0 ataması nesneyi başlatır; ardından değeri okunabilir. Başlatılmamış int okumak güvenilir bir sonuç vermez.

```
int *p = malloc(sizeof *p);
if (p == NULL) return 1;

/* printf("%d", *p);  HATA */
*p = 0;
printf("%d", *p);
free(p);
```

After allocation

No initial value has been written.

↓

After *p = 0

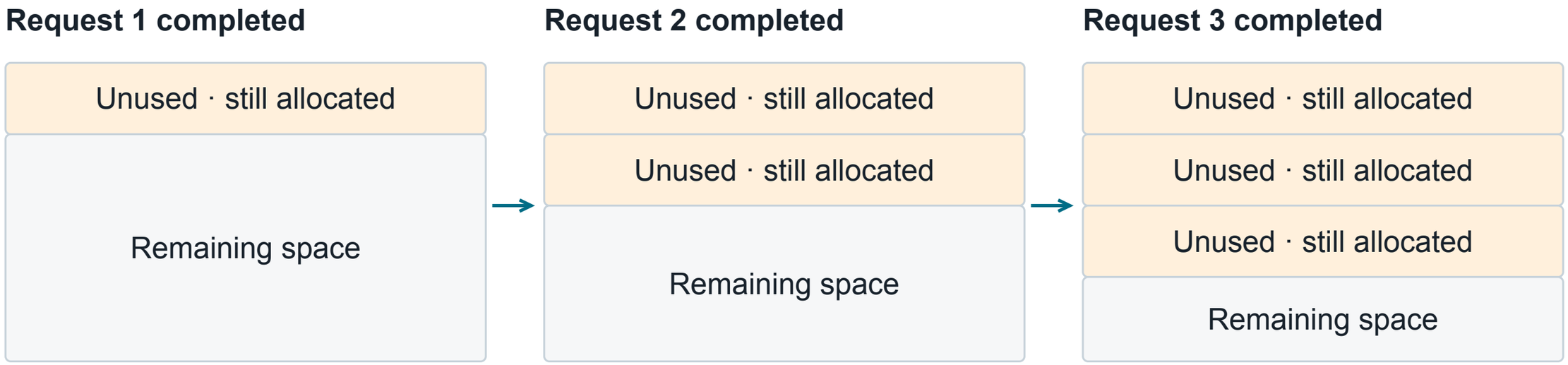
The object now stores 0.

Sonuç: Yer ayırmak ve başlangıç değeri yazmak ayrı adımlardır.

Hata: kullanım bitti, tahsisler kaldı

Forgetting To Free Memory

Artık gerekmeyen alanlar serbest bırakılmazsa tahsisler birikir. Kısa programlarda bu sorun gözden kaçabilir. Uzun süre çalışan süreçlerde ise bellek tüketimi giderek artabilir.



Requests finish; unreleased allocations accumulate.

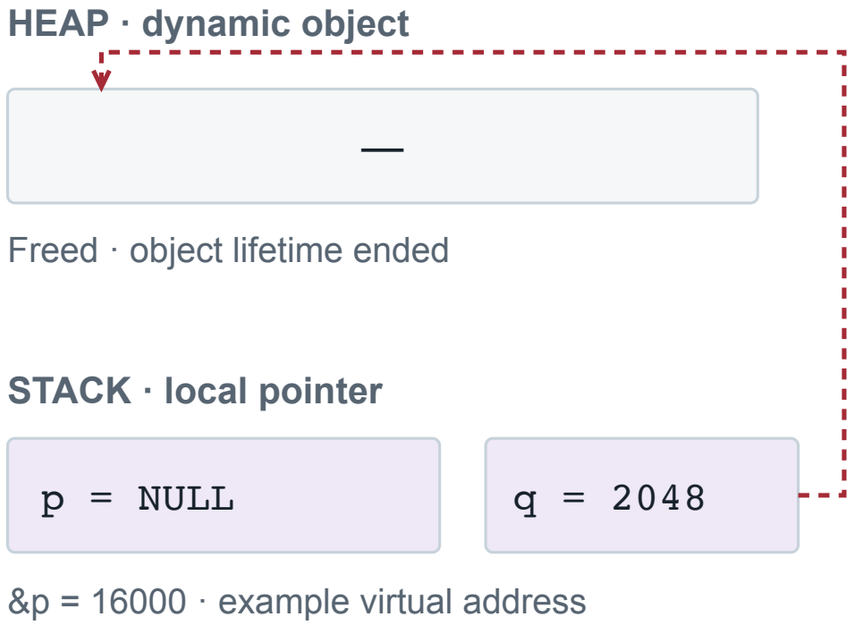
Sonuç: İşin bitmesi, dinamik alanın kendiliğinden serbest kalması demek değildir.

Hata: nesne bitti, başka pointer hâlâ gösteriyor

Freeing Memory Before You Are Done With It

p ve q aynı dinamik nesneyi gösterebilir. free(p), o nesnenin yaşam süresini bitirir; q üzerinden erişimi geçerli tutmaz. Serbest bırakılmış nesneye q ile erişmek hatadır.

```
int *p = malloc(sizeof *p);
if (p == NULL) return 1;
int *q = p;
*p = 42;
free(p);
p = NULL;
/* *q okumak: HATA */
```



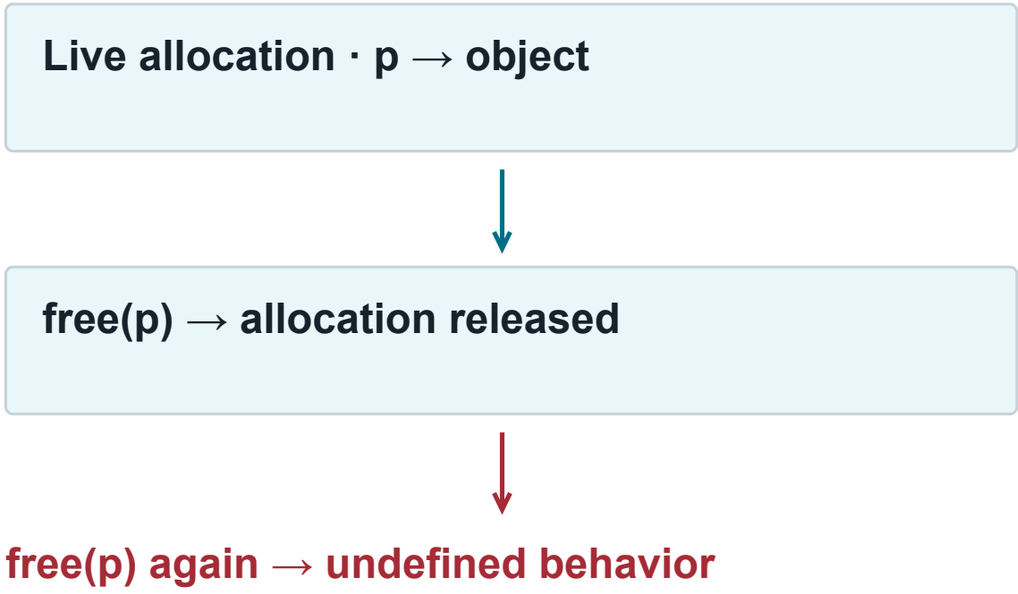
Sonuç: Bir nesnenin yaşamı bitince onu gösteren diğer pointer'lar da erişim için kullanılamaz.

Hata: aynı tahsisi iki kez bırakmak

Freeing Memory Repeatedly

İlk free, dinamik alanı serbest bırakır. Aynı tahsisi eski pointer ile yeniden bırakmak hatadır ve tanımsız davranış oluşturur. Alanı kimin serbest bırakacağı açık olmalıdır.

```
int *p = malloc(sizeof *p);
if (p == NULL) return 1;
free(p);
free(p); /* ERROR */
```



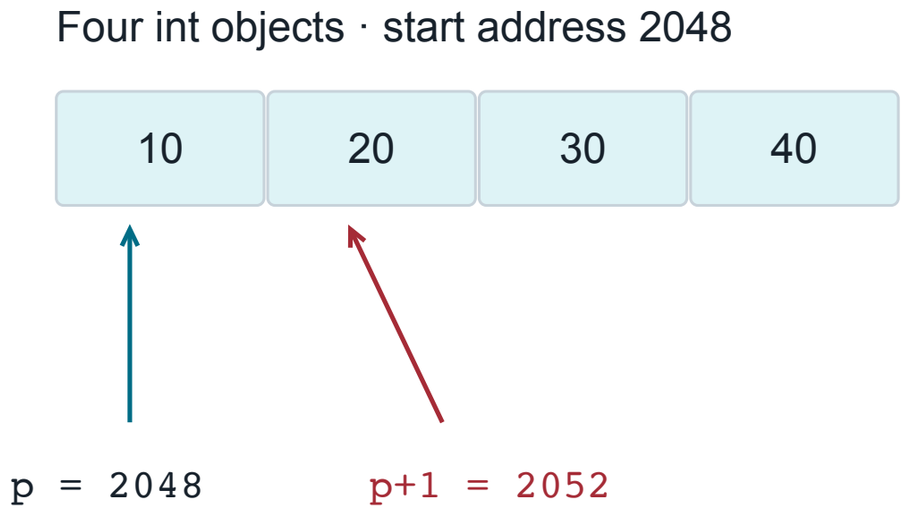
Sonuç: Her tahsis yalnızca bir kez, kullanım bittikten sonra serbest bırakılır.

free'ye hangi adres verilir?

Calling free() Incorrectly

free'ye canlı tahsisin başlangıç adresi veya NULL verilir. Dizinin içindeki p+1, tahsisin başlangıcı değildir. Yerel x değişkeninin adresi &x de malloc ile alınmış bir alanı göstermez.

```
int *p = malloc(4 * sizeof *p);
if (p == NULL) return 1;

/* free(p + 1);  HATA */
/* free(&x);     HATA */
free(p);         /* valid */
```



free(p) is valid; free(p+1) is invalid.

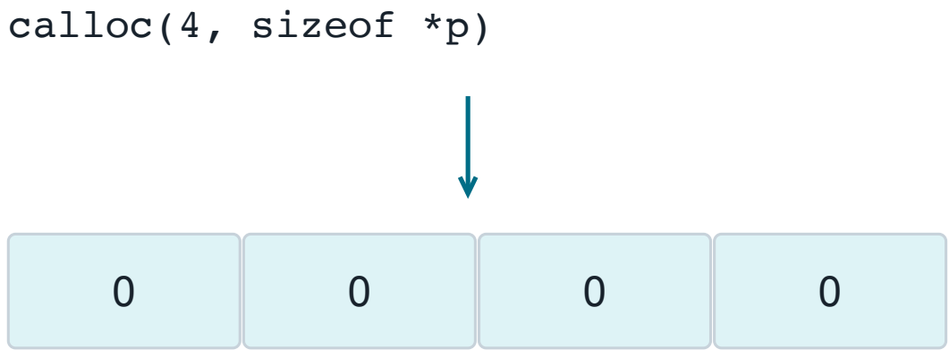
Sonuç: free(p) doğru olabilir; free(p+1) aynı alanı bırakmanın geçerli bir yolu değildir.

calloc: tahsis et ve byte'ları sıfırla

Other Calls

calloc, eleman sayısını ve bir elemanın boyutunu ayrı alır. Başarılı olduğunda ayrılan byte'lar sıfırlanmıştır. Sonuç için NULL kontrolü ve kullanım sonunda free yine gerekir.

```
int *p = calloc(4, sizeof *p);
if (p == NULL) return 1;

/* p[0] ... p[3] == 0 */
free(p);
```



Allocated bytes are zeroed.

The result can still be NULL.

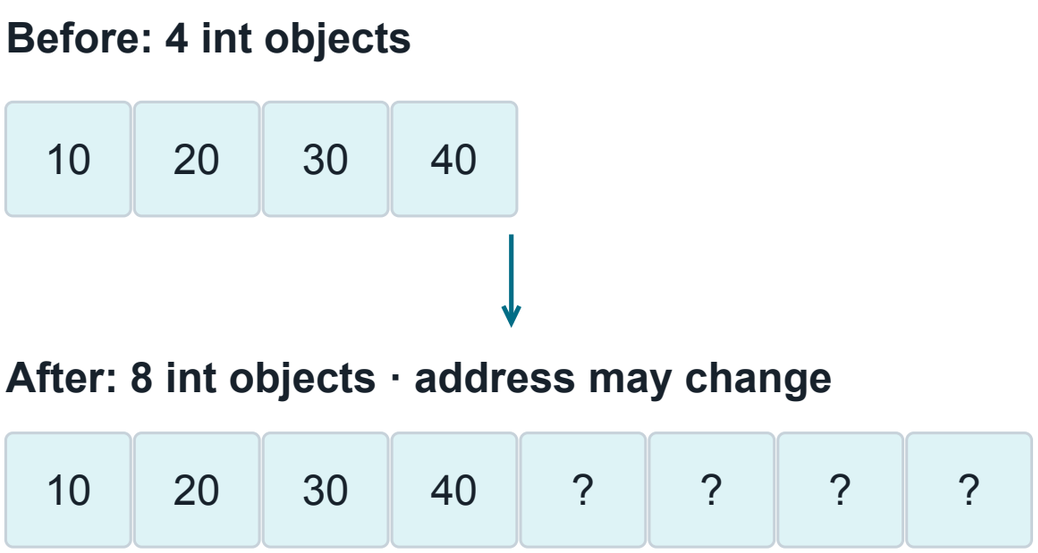
Sonuç: calloc(4, sizeof(int)), bu örnekte dört sıfır değerli int için alan ister.

realloc: alan büyürken adres değişebilir

Other Calls

realloc, ayrılmış alanı aynı yerde büyütebilir veya başka bir konuma taşıyabilir. Başarısızlıkta eski alan korunur. Sonuç önce geçici pointer'da kontrol edilirse eski tahsise erişim kaybedilmez.

```
/* p: önceden alınmış 4 int */
int *tmp = realloc(p, 8 * sizeof *p);
if (tmp != NULL) {
    p = tmp;
} else {
    /* p hâlâ eski alanı gösterir. */
}
/* ... */
free(p);
```



Old contents retained; new part uninitialized.

Sonuç: Başarıda yeni pointer kullanılır; başarısızlıkta eski tahsis korunur.

malloc bir sistem çağrısı mı?

Underlying OS Support

malloc ve free, C kütüphanesi arayüzleridir. Bellek ayırıcı elindeki uygun boş bloğu kullanabiliyorsa her çağrıda çekirdeğe geçmesi gerekmez. Yeni alana ihtiyaç duyduğunda işletim sisteminden yardım alabilir.



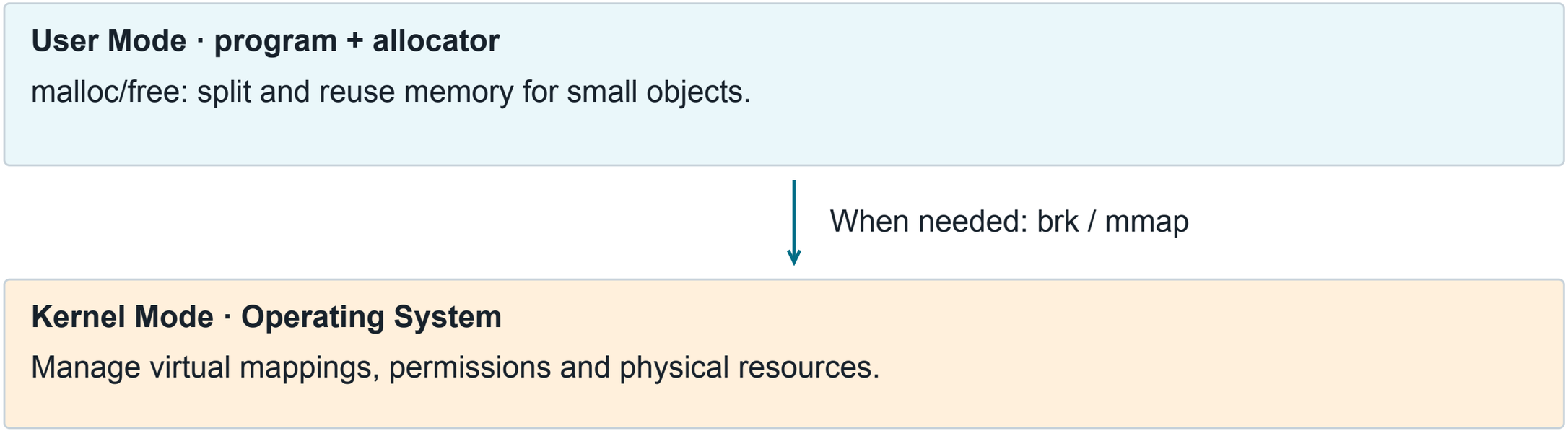
A system call is not required for every malloc.

Sonuç: Kütüphane çağrısı ile sistem çağrısı farklı katmanlardır.

Allocator'ın altında OS var

Underlying OS Support

Bellek ayırıcı, sürece ait bölgeleri küçük tahsislere böler ve yeniden kullanır. Gerektiğinde brk veya mmap üzerinden işletim sisteminden alan edinir. Serbest bırakılan alanın hemen fiziksel RAM'den çıkarılması gerekmez.



Sonuç: Bellek ayırıcı nesneler için alanı paylaştırır; işletim sistemi eşlemeleri ve kaynakları yönetir.

Canlı Linux / C: adresler ve nesnelerin yaşam süresi

C Memory Lab

adresler.c, &local, p ve &p değerlerini ayrı ayrı gösterir. heap.c dört int için alan ayırır, elemanları kullanır ve alanı bırakır. Çıktıdaki adresler sanaldır; 32-bit konuk Linux'ta pointer boyutu 4 byte'tır.

adresler.c

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

int global = 10;

int main(void) {
    int local = 20;
    int *p = malloc(sizeof *p);
    if (p == NULL) return 1;
    *p = 30;
    printf("Pointer boyutu: %zu byte\n", sizeof(void *));
    printf("global: %p, deger: %d\n", (void *)&global, global);
    printf("local : %p, deger: %d\n", (void *)&local, local);
    printf("p     : %p, deger: %d\n", (void *)p, *p);
    printf("&p    : %p (pointer degiskeninin adresi)\n", (void *)&p);
    free(p);
    p = NULL;
    return 0;
}
```

heap.c

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

int main(void) {
    size_t n = 4;
    if (n == 0 || n > 1000) return 1;
    int *values = malloc(n * sizeof *values);
    if (values == NULL) {
        puts("Tahsis basarisiz.");
        return 1;
    }
    for (size_t i = 0; i < n; i++) {
        values[i] = (int)(i + 1) * 10;
        printf("values[%zu] = %d\n", i, values[i]);
    }
    free(values);
    values = NULL;
    puts("Dinamik nesne serbest birakildi.");
    return 0;
}
```

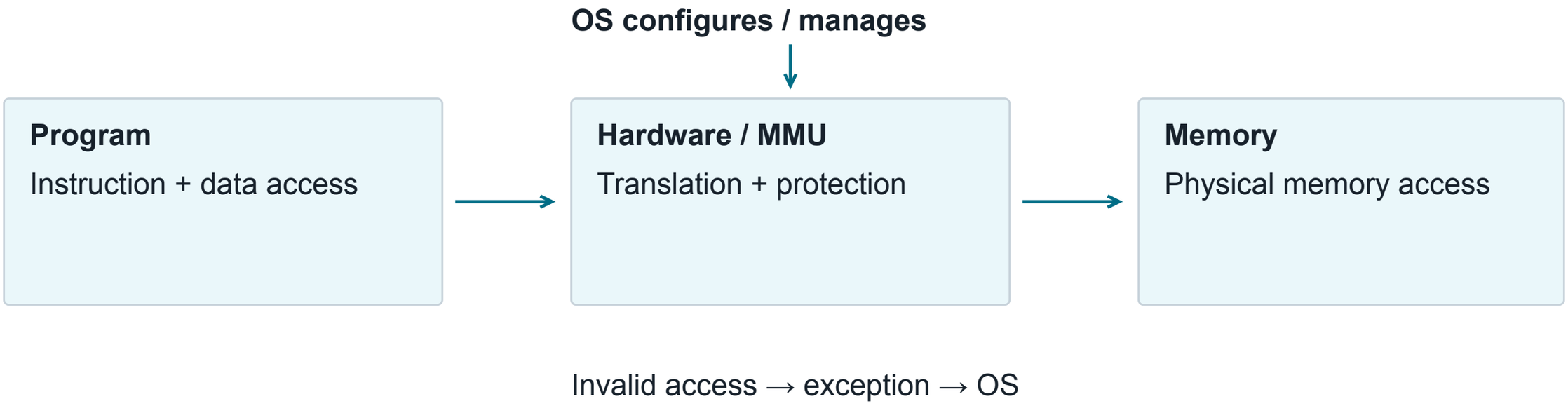
Örnek çıktı: heap.c → 10, 20, 30, 40; ardından "Dinamik nesne serbest birakildi." Adresler platforma göre değişir.

Sonuç: Gerçek çıktı, pointer değişkeni ile gösterdiği nesnenin ayrı olduğunu görünür kılar.

Her erişimde çekirdeğe girersek ne olur?

Mechanism: Address Translation

Program, verileri okumak ve yazmak kadar çalıştıracağı komutları getirmek için de belleğe erişir. Adres çevirmenin ve korumanın hızlı yapılması donanım desteği gerektirir. İşletim sistemi bu desteği kurar ve gerekli anlarda denetimi geri alır.

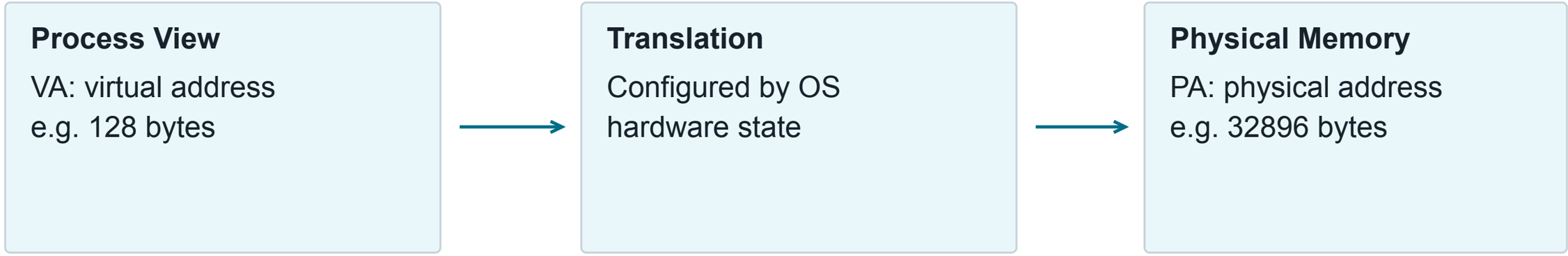


Sonuç: Her erişimde hızlı donanım yolu; kritik anlarda işletim sistemi müdahalesi.

Sanal adresten fiziksel adrese

Assumptions

İlk modelde her sürecin adres uzayı RAM'de tek bir bitişik bölgeye sığar. Donanım, programın sanal adresini bu bölgedeki fiziksel adrese çevirir. Bu varsayım basit bir adres çevirme mekanizması kurmayı sağlar.



Initial model: one contiguous physical block per process.

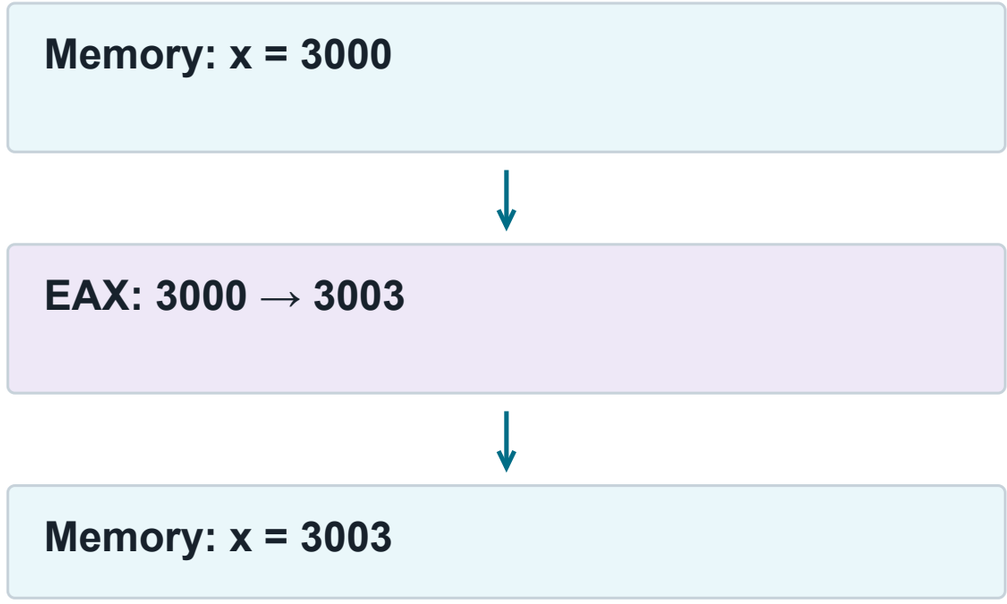
Sonuç: Sanal adresi program üretir; fiziksel konumu eşleme belirler.

x = x + 3 hangi işleri gerektirir?

An Example

Bellekteki x değeri bir yazmaca alınır. Burada 3 eklenir ve sonuç belleğe geri yazılır. Başlangıç değeri 3000 ise sonuç 3003 olur; x'in adresi bu değerden ayrı bir bilgidir.

```
void func(void) {
    int x = 3000;
    x = x + 3;
}
```



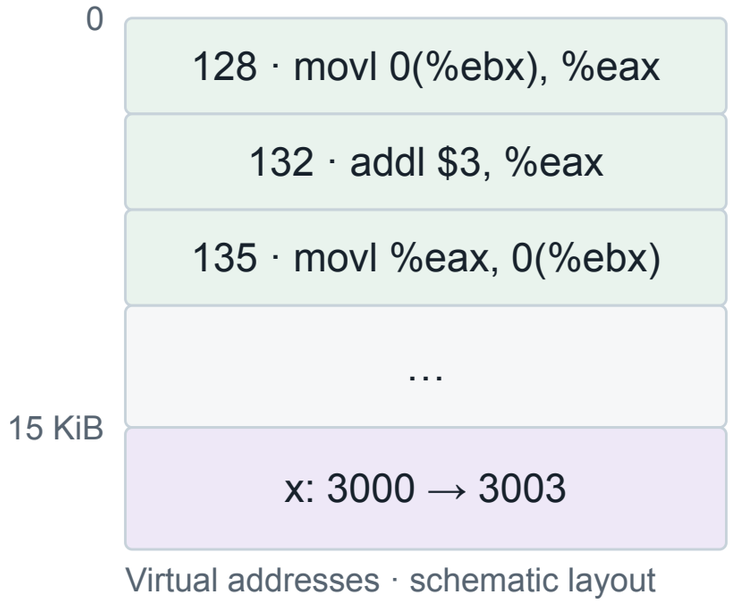
Sonuç: 3000 saklanan veridir; x'in adresi ise onun bulunduğu konumdur.

Aynı işlem: üç komut

An Example

EBX'in x'in adresini tuttuğunu varsayalım. İlk movl, değeri bellekten EAX'e alır; addl EAX'teki değeri artırır; son movl sonucu aynı adrese yazar. Bu komutların kendileri de bellekte tutulur.

```
128: movl 0(%ebx), %eax
132: addl $3, %eax
135: movl %eax, 0(%ebx)

/* EBX = 15 KiB */
/* [15 KiB] = 3000 */
```



Sonuç: Yazmaçta yapılan aritmetik işlem ile belleğe erişim farklı adımlardır.

Üç komut, beş bellek başvurusu

An Example

Her komut önce kendi adresinden getirilir. Okuma ve yazma komutları ayrıca x'in bulunduğu adrese erişir. Toplama yazmaçta yapıldığı için o adımda ek veri erişimi gerekmez.

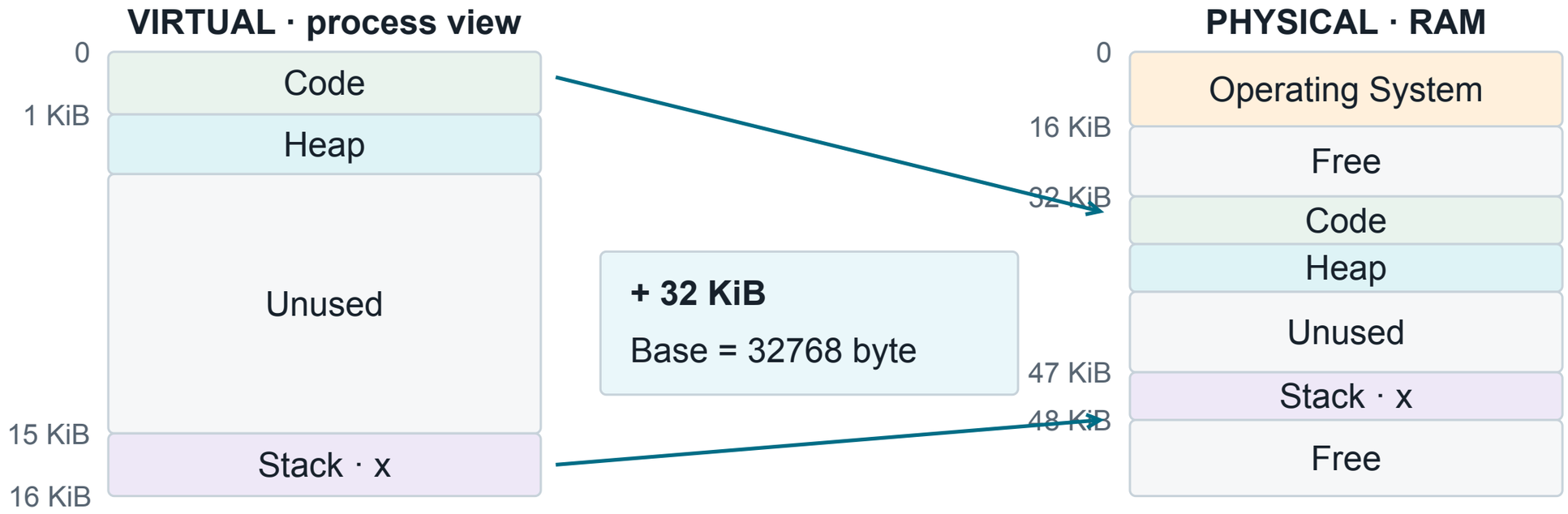
ORDER	REFERENCE	VIRTUAL ADDRESS	RESULT
1	Instruction fetch	`128`	movl · load
2	Data load	`15 KiB`	3000 → EAX
3	Instruction fetch	`132`	addl · EAX = 3003
4	Instruction fetch	`135`	movl · store
5	Data store	`15 KiB`	3003 → x

Sonuç: Bu modelde üç komut getirme, bir veri okuma ve bir veri yazma vardır.

Süreç RAM'de 0 adresinde değilse?

An Example

0–16 KiB sanal adres aralığı, fiziksel bellekte 32–48 KiB aralığına yerleşir. Program kendi sıfır tabanlı adresleriyle çalışır. Her geçerli sanal adrese 32 KiB eklenerek fiziksel karşılığı bulunur.

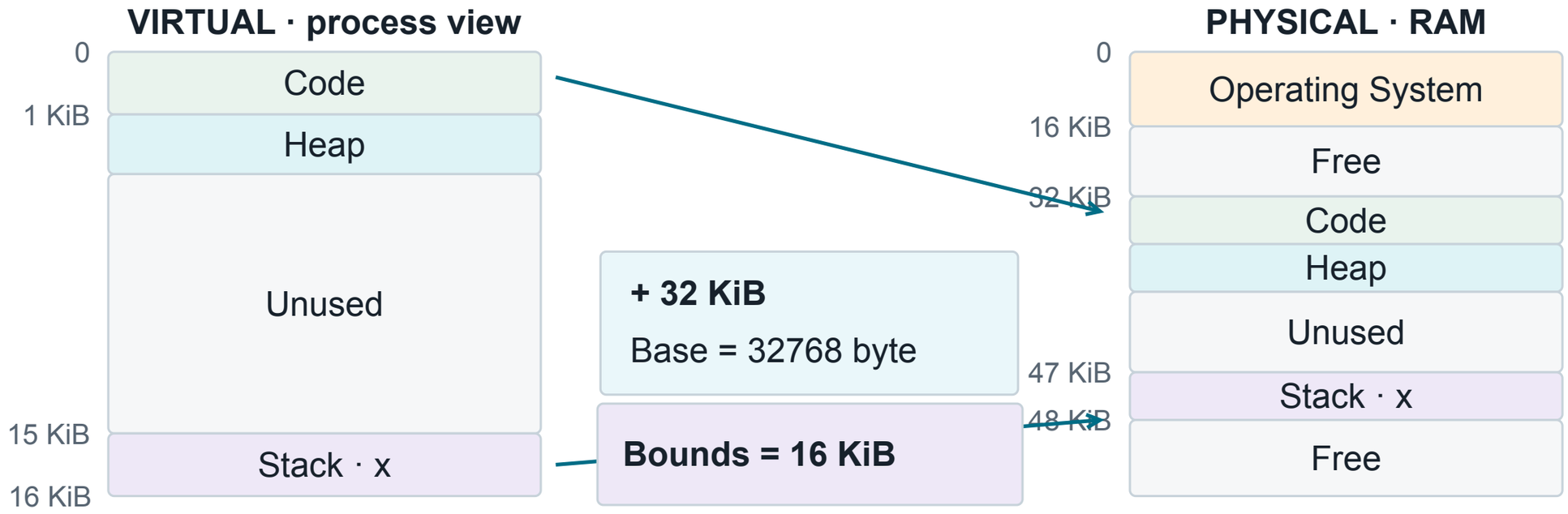


Sonuç: Programın adresleri aynı kalırken fiziksel yerleşimi değişebilir.

İki register: base ve bounds

Dynamic (Hardware-based) Relocation

Base, sürecin fiziksel bellekteki başlangıcını belirtir. Bounds burada sanal adres aralığının uzunluğudur. Donanım önce 0 ≤ VA < bounds koşulunu denetler, sonra base + VA hesaplar.



Sonuç: Örnekte base = 32 KiB, bounds = 16 KiB kullanılır.

Komut adresi de veri adresi de çevrilir

Dynamic (Hardware-based) Relocation

128 sanal adresindeki komut, fiziksel bellekte 32896 adresine karşılık gelir. x'in 15 KiB olan sanal adresi ise 47 KiB'ye çevrilir. Çevirme, 3000 değerini değil o değere erişilen adresi değiştirir.

VIRTUAL ADDRESS	BASE + VA	PHYSICAL ADDRESS
128	32768 + 128	32896
132	32768 + 132	32900
135	32768 + 135	32903
15360 (15 KiB)	32768 + 15360	48128 (47 KiB)

Sonuç: Adres çevirme, saklanan veriye değil erişimin adresine uygulanır.

Koruma: sınırda hangi erişim geçerlidir?

Dynamic (Hardware-based) Relocation

Bu ayrı örnekte sınır 4 KiB'dir ve bir byte'lık erişim incelenir. Geçerli sanal adresler 0–4095'tir. 4096, aralığın dışındaki ilk adrestir; bu adrese fiziksel erişim yapılmaz.

VA = 4095 byte

4095 < 4096 → valid
16384 + 4095 = 20479 byte

VA = 4096 byte

4096 < 4096 → invalid
No physical access.

Base = 16384 bytes · Limit = 4096 bytes · Access width = 1 byte

Sonuç: Üst sınır hariçtir: VA < limit; VA ≤ limit değildir.

OS hangi anlarda devreye girer?

Operating System Issues

İşletim sistemi süreç için bellek bulur ve adres çevirme bilgilerini kurar. Süreç değiştiğinde yenisinin bilgilerini yükler. Geçersiz erişimi işler; süreç sonlandığında kaynakları geri alır.

Process creation

Find space; initialize translation state.

Context switch

Save old state; restore new state.

Invalid access

OS handles the hardware exception.

Process termination

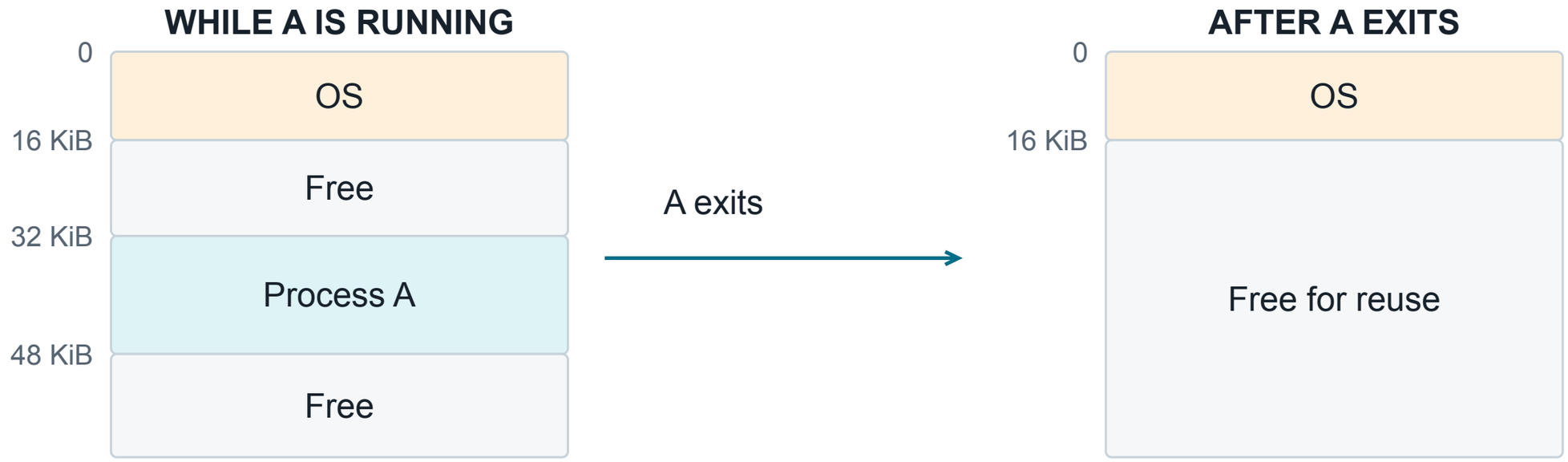
Reclaim memory resources.

Sonuç: İşletim sistemi eşlemeleri yönetir; donanım çevirme ve koruma kurallarını uygular.

Süreç bitince belleği yeniden kullanılabilir

Operating System Issues

A süreci sonlandığında ona ayrılan fiziksel alan yeniden kullanılabilir olur. İşletim sistemi bu alanı boş alan listesi gibi yapılarda izler. Başka bir süreç için yer aranırken bu bilgiler kullanılır.



Sonuç: Sürecin kaynaklarını geri almak ile tek bir malloc alanını bırakmak farklı düzeylerdeki işlemlerdir.

Context switch: aynı adres, başka süreç

Operating System Issues

A çalışırken onun base/bounds değerleri kullanılır. B'ye geçerken A'nın bilgileri saklanır ve B'ninkiler yüklenir. Böylece aynı sanal adres iki süreçte farklı fiziksel konumlara karşılık gelebilir.

A running

Base = 32 KiB · Limit = 16 KiB
VA = 1 KiB → PA = 33 KiB

→

B running

Base = 48 KiB · Limit = 16 KiB
VA = 1 KiB → PA = 49 KiB

State of A → saved in the PCB of A.
State of B → restored to hardware.

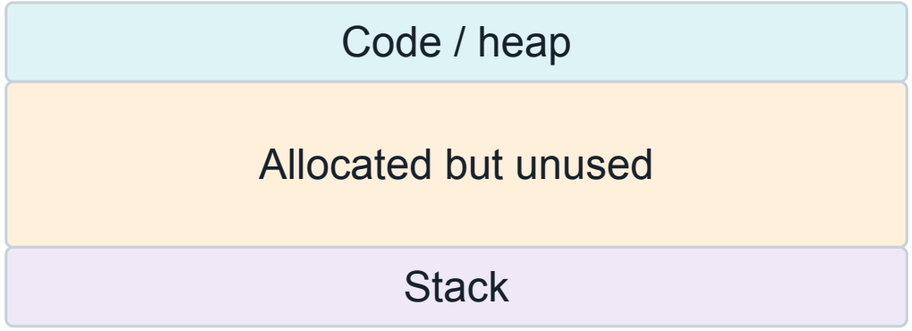
Sonuç: Süreç değiştiğinde adreslerin anlamını belirleyen çevirme bilgileri de değişir.

Bu kolay model neden yetmiyor?

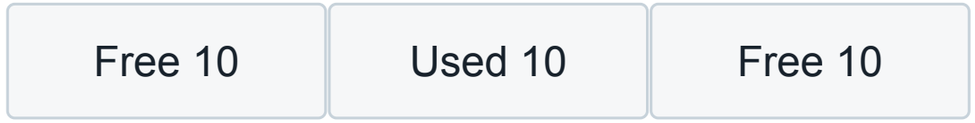
Summary

Tek parça yerleşimde kullanılmayan sanal aralık da ayrılan fiziksel bloğun içinde kalır. Ayrıca toplam boş bellek yeterli olsa bile istenen büyüklükte tek bir boş aralık bulunmayabilir. Bu iki sorun daha esnek bir yerleşim gerektirir.

1 · Unused space inside the block



2 · External fragmentation



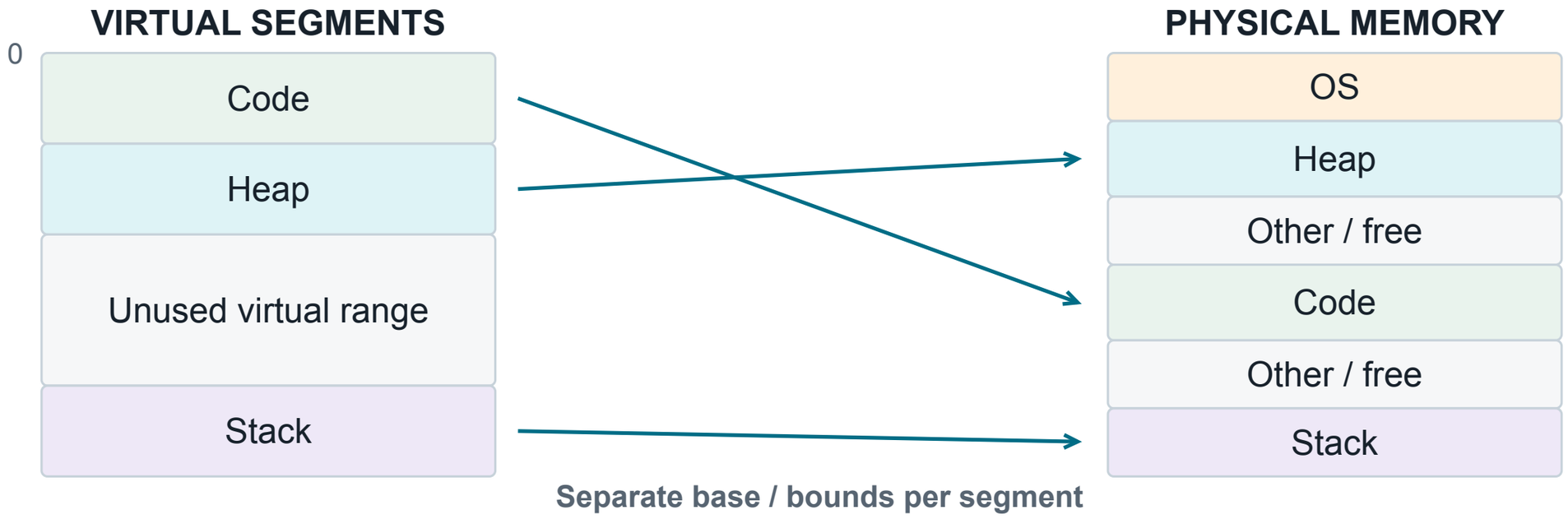
Request 15 → no contiguous block.

Sonuç: Blok içinde kullanılmayan alan ile dağınık boşluklar farklı problemlerdir.

Geçiş: her mantıksal bölgeye ayrı eşleme

Segmentation: Generalized Base/Bounds

Segmentation, code, heap ve stack gibi bölgeler için ayrı base/bounds bilgileri tutar. Aradaki kullanılmayan sanal boşluğu aynı fiziksel blokta ayırmak gerekmez. Değişken boyutlu bölgeleri yerleştirme sorunu ise devam eder.

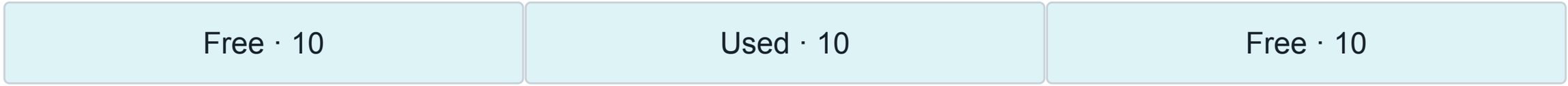


Sonuç: Segmentation boşluk israfını azaltır; dış parçalanmayı tek başına çözmez.

Fragmentation: toplam boşluk neden yetmeyebilir?

Low-level Mechanisms

İki boş bölgenin her biri 10 birimdir. Toplam 20 birim boş olsa da tek parça 15 birimlik alan yoktur. Ortadaki dolu alan serbest bırakılınca komşu boşluklar birleşir; oluşan 30 birimlik alandan 15 birim ayrılabilir.



Total free = 20 · Largest free block = 10 · Request 15 fails.

İç parçalanma örneği: 6 birim istenir, 8 birimlik blok ayrılır; bloğun içinde 2 birim artar.

Sonuç: Dış parçalanma boş alanların dağınık kalmasıdır; iç parçalanma ayrılmış bloğun içinde oluşan fazlalıktır.

Program, bellek ayırıcı, işletim sistemi ve donanım

Memory Virtualization: Review

Program nesneleri kullanır; bellek ayırıcı dinamik alanı paylaştırır. İşletim sistemi eşlemeleri ve fiziksel kaynakları yönetir; donanım adres çevirme ve korumayı uygular. Paging, tek parça fiziksel yerleşim zorunluluğunu kaldıran sonraki adımdır.

Pointer and Object

p = malloc(...)
p holds an address.
free(p) ends object lifetime.
Aliases become invalid.

Address and Protection

VA is the process view.
PA is the physical location.
4095 < 4096: valid.
4096 < 4096: invalid.

Placement and Paging

20 free; largest block 10.
Contiguous 15 cannot fit.
Next: fixed-size
pages and frames.

Sonuç: Sanal görünüm → nesnenin yaşam süresi → alan ayırma → adres çevirme → koruma.