Bölüm 8: Bellek Yönetimi

İşletim Sistemleri

Bellek hiyerarşisi ve önbellek | Taban ve limit yazmaçları | Adres bağlama ve MMU

Dinamik yükleme ve bağlama | Takas | Bitişik tahsis ve uyan algoritmaları

Parçalanma ve biteşlem | Sayfalama | TLB ve erişim süresi

Hiyerarşik, hash, ters tablolar | x86-64 ve büyük sayfalar | Kesimleme, x86 ve ARMv8

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_08_Bellek_Yonetimi.html

GİRİŞ

Öğrenme Çıktıları

Soyutlama
Bellek hiyerarşisini, önbellek satırlarını ve ortalama erişim süresini, adres alanını, koruma ve yer değiştirme problemlerini açıklamak.

Adresler
Bağlama zamanlarını, mantıksal/fiziksel adresi ve MMU'nun taban-limit denetimini açıklamak.

Bağlama
Dinamik yüklemeyi, statik ve dinamik bağlamayı, PLT/GOT ile paylaşımlı kütüphaneleri ve takası tartışmak.

Bitişik tahsis
Uyan algoritmalarını uygulamak; iç/dış parçalanmayı, biteşlemi ve bağlı listeyi çözümlemek.

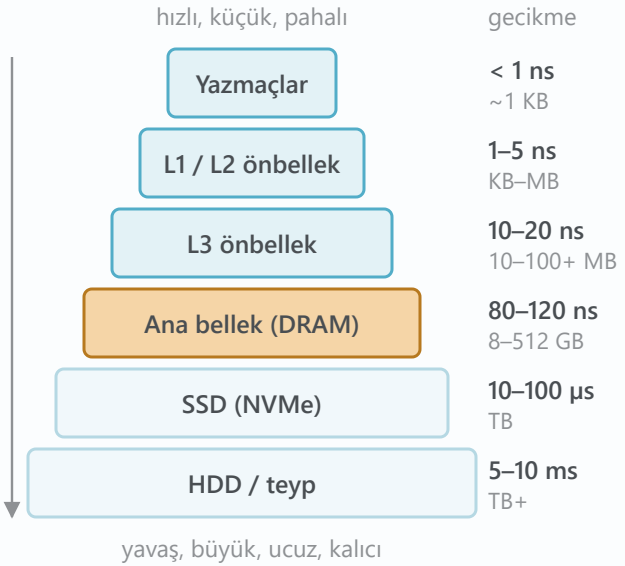
Sayfalama
Adresi (p, d) olarak bölmek, sayfa tablosu ve TLB ile çeviri yapmak, etkin erişim süresini hesaplamak.

Gerçek sistemler
Hiyerarşik, hash ve ters tabloları; x86-64 dört/beş düzeyli sayfalamayı, büyük sayfaları, kesimlemeyi ve ARMv8'i karşılaştırmak.

GİRİŞ

Bellek Hiyerarşisi ve Bellek Yöneticisi

- CPU'nun **doğrudan** erişebildiği depolama yalnızca **yazmaçlar** ve **ana bellektir**; disk ancak G/Ç ile.
- Bellek birimi yalnızca bir akış görür: *adres* + *okuma* ya da *adres* + *veri* + *yazma*. Adresin nereden geldiğini bilmez.
- Yazmaç bir saat döngüsünde okunur; DRAM erişimi **yüzlerce döngü** sürer ve CPU durur (stall). Aradaki boşluğu **önbellek** kapatır.
- Programcı belleğin *sınırsız, hızlı, kalıcı ve korumalı* olmasını ister; gerçek çözüm hiyerarşidir.
- **Bellek yöneticisi**: boş/dolu alanı izler, süreçlere tahsis eder, geri alır, süreçleri birbirinden korur.



Değerler 2026 itibarıyla tipik mertebelerdir; çekirdek ve modele göre değişir. Önbellek donanımca, ana bellek işletim sistemince yönetilir.

BELLEK DONANIMI

Ana Bellek Türleri

Tür	Özellik	Kullanım
RAM (rastgele erişimli)	Okunur ve yazılır, hızlı, **uçucu**. DRAM yenileme ister; SRAM daha hızlı ve pahalı.	Ana bellek (DRAM: DDR5, LPDDR5X), önbellekler (SRAM)
ROM (salt okunur)	Üretimde yazılır, değiştirilemez, kalıcı, ucuz.	Basit aygıtlar, mikrodenetleyicilerde sabit kod
EEPROM / Flash	Elektrikle silinip yeniden yazılabilir, kalıcı; yazma yavaş ve yazma ömrü sınırlı.	**UEFI/BIOS ürün yazılımı** (SPI flash), SSD, USB bellek, telefon ve müzik çalar depolaması (NAND flash)
CMOS / NVRAM	Pil ya da flash ile korunan küçük alan.	Saat, önyükleme ayarları

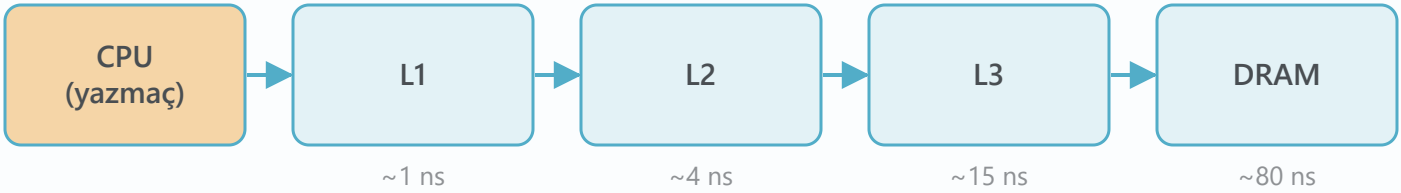
Bugünkü bilgisayarlarda BIOS/UEFI kodu ROM'da değil **flash bellekte** durur; bu yüzden güncellenebilir. Taşınabilir müzik çalarlar da disk değil NAND flash kullanır.

Önbellek Satırları

- Bellek, **önbellek satırlarına** (cache line, genelde **64 bayt**) bölünür: satır 0 = 0–63, satır 1 = 64–127, …
- Adres *a* için satır numarası ⌊a / 64⌋, satır içi konum a mod 64.
- CPU bir sözcük okurken donanım önce önbelleğe bakar: **isabet** (hit) → L1'de ~4–5 tur; **ıska** (miss) → alt düzey ya da DRAM, yüzlerce tur.
- Önbellek pahalı olduğundan küçüktür; bu yüzden düzeyler (L1, L2, L3) hâlinde kurulur.
- Neden işe yarar? **Zamansal yerellik** (az önce kullanılan yine kullanılır) ve **uzamsal yerellik** (komşu adresler birlikte kullanılır).



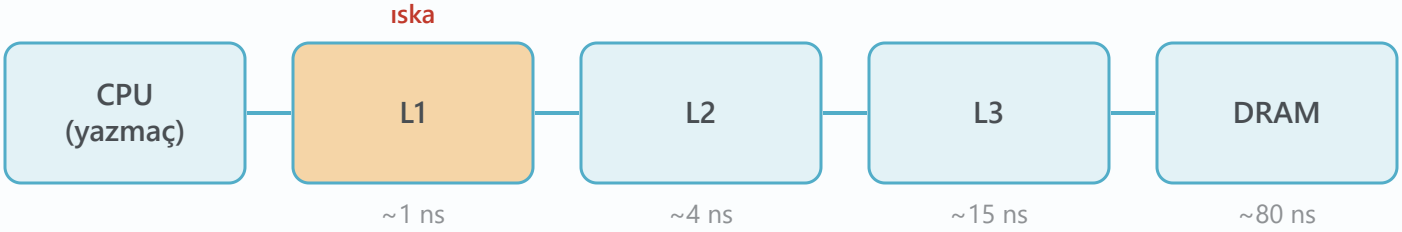
Bir Okuma Hiyerarşide Nasıl İlerler?



toplam gecikme: 0 ns

CPU `x` değişkenini okumak istiyor (`LOAD R1, [x]`). Önce en yakın ve en hızlı düzeye bakılır.

Bir Okuma Hiyerarşide Nasıl İlerler?



toplam gecikme: ~1 ns

L1'de yok: **ıska** (miss). ~1 ns kaybedildi; aramaya bir alt düzeyde devam edilir.

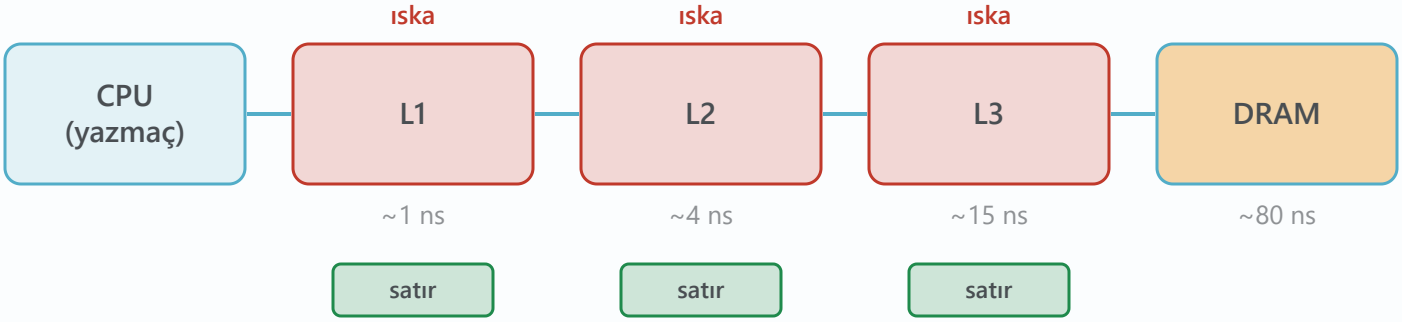
Bir Okuma Hiyerarşide Nasıl İlerler?



toplam gecikme: ~20 ns

L2 ve L3'te de ıska. Her düzey biraz daha büyük ama daha yavaştır; toplam ~20 ns.

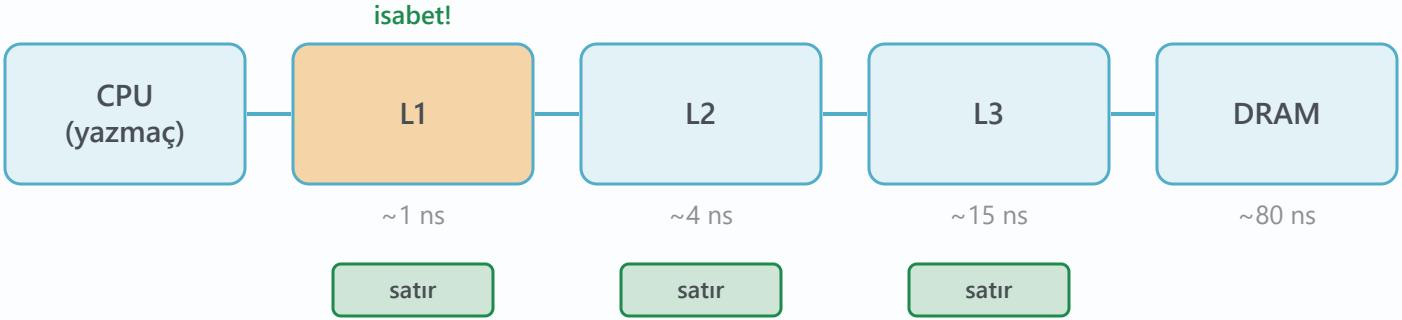
Bir Okuma Hiyerarşide Nasıl İlerler?



toplam gecikme: ~100 ns

Veri DRAM'den gelir (~80 ns). Tek bir sözcük değil, **64 baytlık satırın tamamı** L3, L2 ve L1'e yerleştirilir.

Bir Okuma Hiyerarşide Nasıl İlerler?



Aynı satırdaki bir veriye ikinci erişim artık **L1 isabeti** (hit): ~1 ns. Önbellek, **yerellik** sayesinde işe yarar.

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72

Önbellek (4 küme)

küme	blok (etiket)
0	–
1	–
2	–
3	–

Başlangıçta önbellek boş (−1). 64 B satır, **4 küme**, doğrudan eşleme: her blok tek bir kümeye gidebilir.

Adım 1 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
▶ int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5   int blok = adres[i] / 64;
6   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
8   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`etiket = [-1, -1, -1, -1]`

`isabet = 0` `iska = 0`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72

Önbellek (4 küme)

küme	blok (etiket)
0	–
1	–
2	–
3	–

adres[0] = 0 → blok = 0 / 64 = **0**, küme = 0 mod 4 = **0**.

Adım 2 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
▶   int blok = adres[i] / 64;
▶   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
8   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`i = 0` `blok = 0` `kume = 0`

`etiket = [-1, -1, -1, -1]`

`isabet = 0` `iska = 0`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M					

Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	–
2	–
3	–

Küme 0 boş → **ıska**; blok 0 yüklenir.

Adım 3 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5   int blok = adres[i] / 64;
6   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
▶   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`i = 0` `blok = 0` `kume = 0`

`etiket = [0, -1, -1, -1]`

`isabet = 0` `iska = 1`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M					

Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	–
2	–
3	–

adres[1] = 8 → blok = 8 / 64 = **0**, küme = 0 mod 4 = **0**.

Adım 4 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
▶   int blok = adres[i] / 64;
▶   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
8   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

i = 1 | blok = 0 | kume = 0

etiket = [0, -1, -1, -1]

isabet = 0 | iska = 1

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M	H				

Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	–
2	–
3	–

etiket[0] = 0 = blok → **isabet**. 8, 0 ile aynı 64 baytlık satırda: **uzamsal yerellik**.

Adım 5 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5   int blok = adres[i] / 64;
6   int kume = blok % 4;
▶   if (etiket[kume] == blok) isabet++;
8   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`i = 1` `blok = 0` `kume = 0`

`etiket = [0, -1, -1, -1]`

`isabet = 1` `iska = 1`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M	H				

Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	–
2	–
3	–

adres[2] = 64 → blok = 64 / 64 = **1**, küme = 1 mod 4 = **1**.

Adım 6 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
▶   int blok = adres[i] / 64;
▶   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
8   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`i = 2` `blok = 1` `kume = 1`

`etiket = [0, -1, -1, -1]`

`isabet = 1` `iska = 1`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M	H	M			

Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	blok 1
2	–
3	–

Küme 1 boş → **ıska**; blok 1 yüklenir.

Adım 7 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5   int blok = adres[i] / 64;
6   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
▶   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`i = 2` `blok = 1` `kume = 1`

`etiket = [0, 1, -1, -1]`

`isabet = 1` `iska = 2`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M	H	M			

Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	blok 1
2	–
3	–

adres[3] = 256 → blok = 256 / 64 = **4**, küme = 4 mod 4 = **0**.

Adım 8 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
▶   int blok = adres[i] / 64;
▶   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
8   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`i = 3` `blok = 4` `kume = 0`

`etiket = [0, 1, -1, -1]`

`isabet = 1` `iska = 2`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M	H	M	M		

Önbellek (4 küme)

küme	blok (etiket)
0	blok 4
1	blok 1
2	–
3	–

Küme 0'da blok 0 var → **ıska**; blok 4 onu **dışarı atar** (çakışma ıskası).

Adım 9 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5   int blok = adres[i] / 64;
6   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
▶   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`i = 3` `blok = 4` `kume = 0`

`etiket = [4, 1, -1, -1]`

`isabet = 1` `iska = 3`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M	H	M	M		

Önbellek (4 küme)

küme	blok (etiket)
0	blok 4
1	blok 1
2	–
3	–

adres[4] = 0 → blok = 0 / 64 = **0**, küme = 0 mod 4 = **0**.

Adım 10 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
▶   int blok = adres[i] / 64;
▶   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
8   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

i = 4 | blok = 0 | kume = 0

etiket = [4, 1, -1, -1]

isabet = 1 | iska = 3

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M	H	M	M	M	

Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	blok 1
2	–
3	–

Küme 0'da blok 4 var → **ıska**; blok 0 onu **dışarı atar** (çakışma ıskası).

Adım 11 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5   int blok = adres[i] / 64;
6   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
▶   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`i = 4` `blok = 0` `kume = 0`

`etiket = [0, 1, -1, -1]`

`isabet = 1` `iska = 4`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M	H	M	M	M	

Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	blok 1
2	–
3	–

adres[5] = 72 → blok = 72 / 64 = **1**, küme = 1 mod 4 = **1**.

Adım 12 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
▶   int blok = adres[i] / 64;
▶   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
8   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`i = 5` `blok = 1` `kume = 1`

`etiket = [0, 1, -1, -1]`

`isabet = 1` `iska = 4`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M	H	M	M	M	H

Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	blok 1
2	–
3	–

etiket[1] = 1 = blok → **isabet**. 72, 64 ile aynı satırda.

Adım 13 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5   int blok = adres[i] / 64;
6   int kume = blok % 4;
▶   if (etiket[kume] == blok) isabet++;
8   else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

`i = 5` `blok = 1` `kume = 1`

`etiket = [0, 1, -1, -1]`

`isabet = 2` `iska = 4`

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
M	H	M	M	M	H

Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	blok 1
2	–
3	–

Döngü bitti: **2 isabet, 4 ıska** (isabet oranı %33). 0 ve 256 aynı kümeye düştüğü için birbirini kovdu; 2 yollu küme ilişkili önbellek bunu önlerdi. (Burada etiket yerine blok numarasının tamamı saklandı.)

Adım 14 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5   int blok = adres[i] / 64;
6   int kume = blok % 4;
7   if (etiket[kume] == blok) isabet++;
8   else { etiket[kume] = blok; iska++; }
▶ }
```

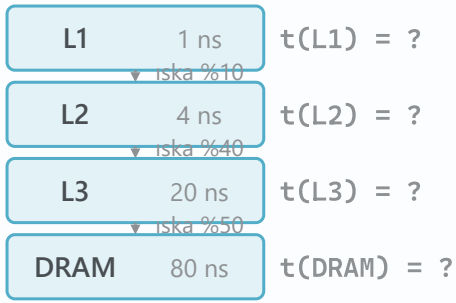
DEĞİŞKENLER

`etiket = [0, 1, -1, -1]`

`isabet = 2` `iska = 4`

Kod İzleme: Ortalama Bellek Erişim Süresi

```
/* L1, L2, L3, DRAM erişim süresi (ns) */
double sure[4] = {1, 4, 20, 80};
double iska[3] = {0.10, 0.40, 0.50};
double t = sure[3];          /* DRAM */
for (int k = 2; k >= 0; k--)
  t = sure[k] + iska[k] * t;
printf("ortalama = %.1f ns\n", t);
```



Dört düzey: erişim süreleri ve her önbellek düzeyinin **ıska oranı**. Bir düzeyde ıska olursa istek bir alttakine iner.

Adım 1 / 10

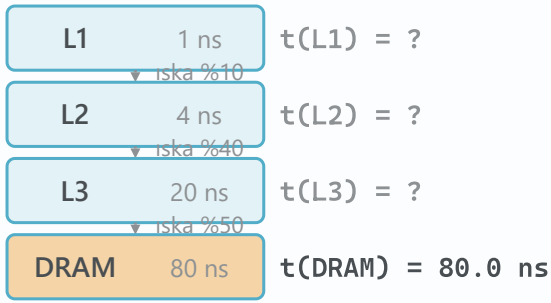
DEĞİŞKENLER

`t = ?` `k = ?`

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
▶ double t = sure[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
6   t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



`t = 80`

En alt düzey DRAM her isteği karşılar: oradan başlayınca **t = 80 ns**.

Adım 2 / 10

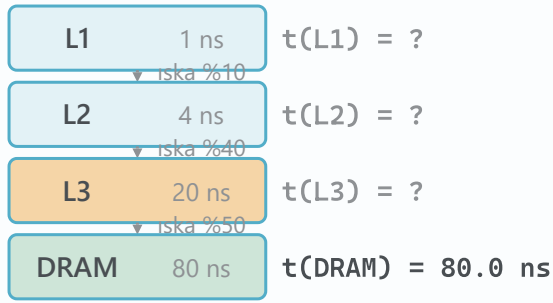
DEĞİŞKENLER

`t = 80.0` `k = ?`

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
▶ for (int k = 2; k >= 0; k--)
6   t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



k = 2 ≥ 0: bir üst düzeye (L3) çıkılır.

Adım 3 / 10

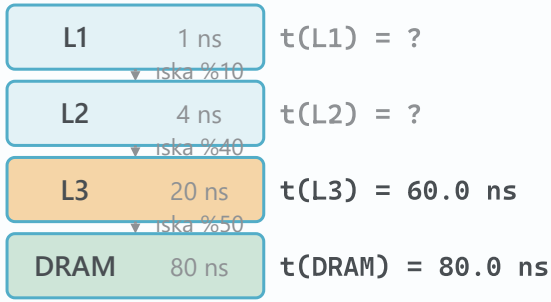
DEĞİŞKENLER

`t = 80.0` `k = 2`

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
▶   t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



`t = 20 + 0.50 × 80.0 = 60.0`

L3 her istekte 20 ns harcar; isteklerin %50'sinde alttaki düzeyin 80 ns'si eklenir: **t = 60 ns**.

Adım 4 / 10

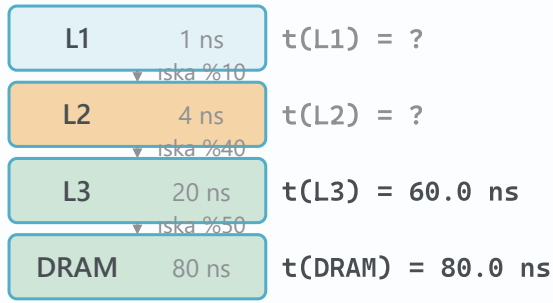
DEĞİŞKENLER

`t = 60.0` `k = 2`

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
▶ for (int k = 2; k >= 0; k--)
6   t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



k = 1 ≥ 0: bir üst düzeye (L2) çıkılır.

Adım 5 / 10

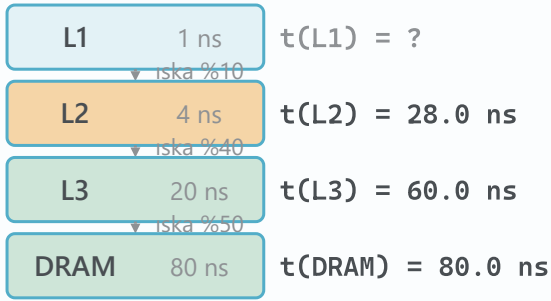
DEĞİŞKENLER

`t = 60.0` `k = 1`

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
▶   t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



`t = 4 + 0.40 × 60.0 = 28.0`

L2 her istekte 4 ns harcar; isteklerin %40'ında alttaki düzeyin 60 ns'si eklenir: **t = 28 ns**.

Adım 6 / 10

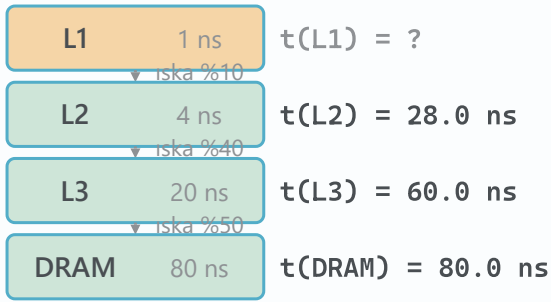
DEĞİŞKENLER

`t = 28.0` `k = 1`

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
▶ for (int k = 2; k >= 0; k--)
6   t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



k = 0 ≥ 0: bir üst düzeye (L1) çıkılır.

Adım 7 / 10

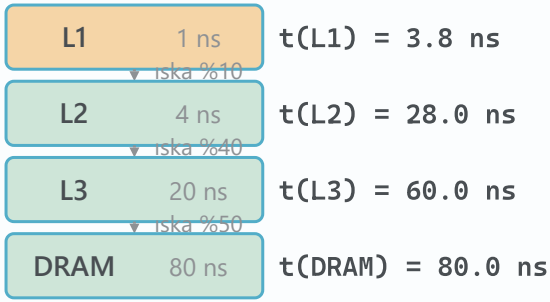
DEĞİŞKENLER

`t = 28.0` `k = 0`

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
▶   t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



`t = 1 + 0.10 × 28.0 = 3.8`

L1 her istekte 1 ns harcar; isteklerin %10'unda alttaki düzeyin 28 ns'si eklenir: **t = 3,8 ns**.

Adım 8 / 10

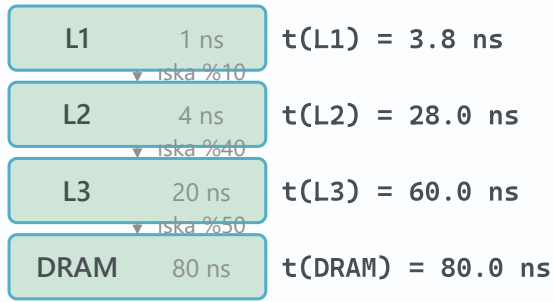
DEĞİŞKENLER

`t = 3.8` `k = 0`

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
▶ for (int k = 2; k >= 0; k--)
6   t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



k = −1: döngü biter.

Adım 9 / 10

DEĞİŞKENLER

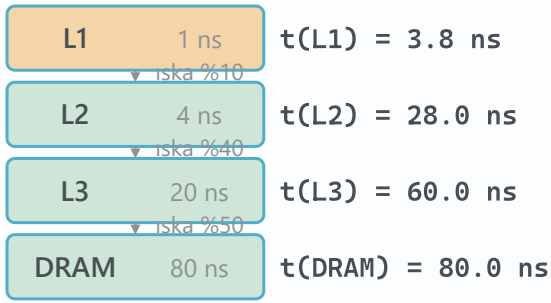
`t = 3.8` `k = -1`

ÇIKTI

```
ortalama = 3.8 ns
```

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
6   t = sure[k] + iska[k] * t;
▶ printf("ortalama = %.1f ns\n", t);
```



`ortalama = 3.8 ns`

Her erişimin %10'u L1'i ıskalasa da ortalama **3,8 ns**: DRAM'in 80 ns'sinden yaklaşık 21 kat hızlı. Hiyerarşiyi işe yarar kılan yüksek isabet oranıdır.

Adım 10 / 10

DEĞİŞKENLER

`t = 3.8` `k = -1`

ÇIKTI

```
ortalama = 3.8 ns
```

BELLEK DONANIMI

Önbellek Tasarım Soruları

1. Ne zaman yerleştirilir?
Genelde ıskada; donanım ayrıca sıradaki satırları önceden getirir (prefetch).

2. Hangi satıra konur?
Eşleme: doğrudan eşlemeli, *k* yollu küme ilişkili ya da tam ilişkili.

3. Yer gerekince hangisi çıkar?
Değiştirme politikası: LRU'ya yakın yaklaşımlar, rastgele, FIFO.

4. Çıkan nereye gider?
Değiştirilmişse ("kirli") alt düzeye yazılır (write-back); ya da her yazma hemen aşağı iletilir (write-through).

Boyut
Büyük önbellek daha çok isabet ama daha yavaş ve pahalı.

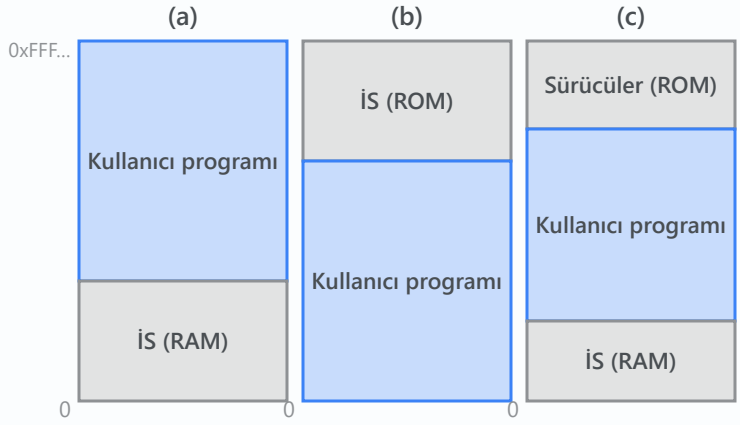
Değiştirme
Performans ile donanım karmaşıklığı arasında denge.

Tutarlılık
Çok çekirdekte aynı satırın kopyaları güncel kalmalı (MESI gibi protokoller).

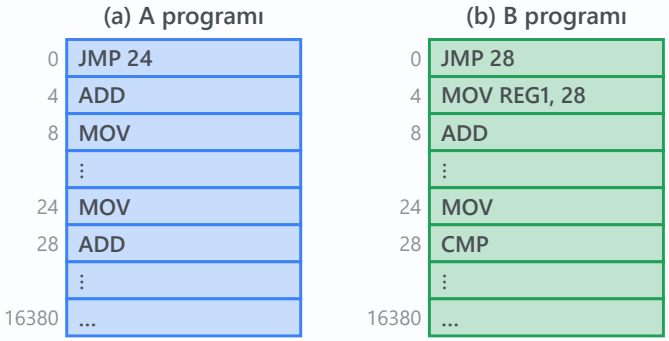
Aynı dört soru İS'nin yönettiği "önbelleklerde" de karşımıza çıkar: sayfa önbelleği, sanal bellek, TLB.

Soyutlama Olmadan Bellek

- İlk bilgisayarlarda (1960'lar öncesi ana bilgisayarlar, 1980'ler öncesi kişisel bilgisayarlar) her program **fiziksel belleği** doğrudan görürdü: `MOV REGISTER1, 1000` fiziksel 1000 adresini okur.
- Bellekte aynı anda **iki program olamaz**: ikisi de aynı fiziksel adreslere yazar, biri diğerini ezer.
- (a) ve (c)'de kullanıcı programındaki bir hata işletim sistemini **çökertebilir**.
- (b) bazı gömülü sistemlerde, (c) MS-DOS'lu ilk PC'lerde kullanıldı (BIOS sürücüleri ROM'da).
- Bugün de MMU'suz mikrodenetleyiciler böyle çalışır; çoğu en azından **MPU** (bellek koruma birimi) ile bölgeleri korur.



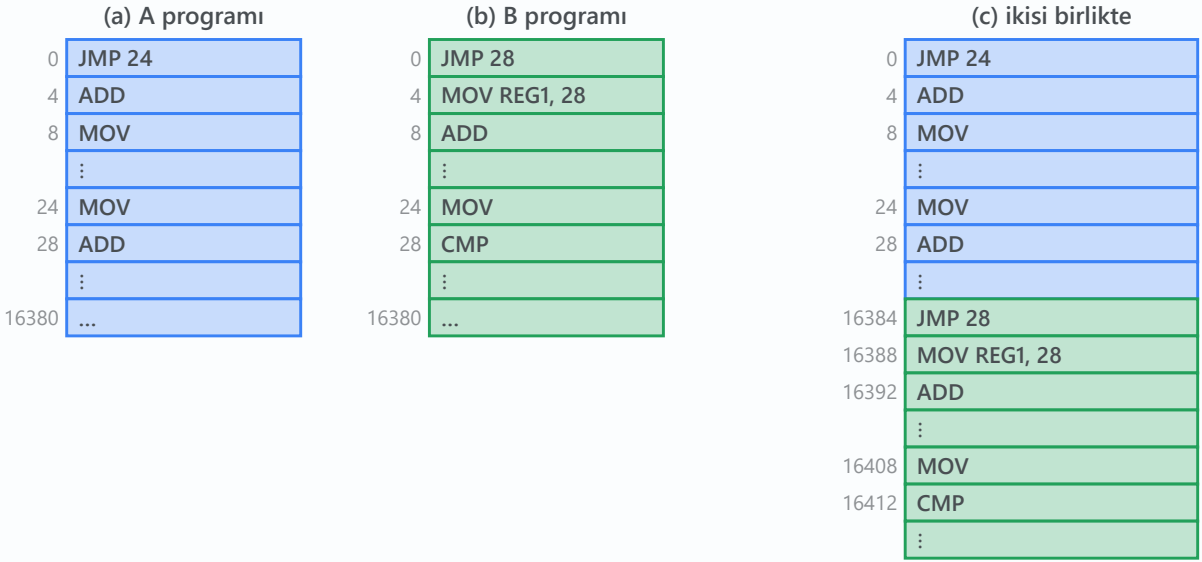
Yer Değiştirme Problemi



ikisi de 0'dan başlayacak
şekilde derlendi (16 KB)

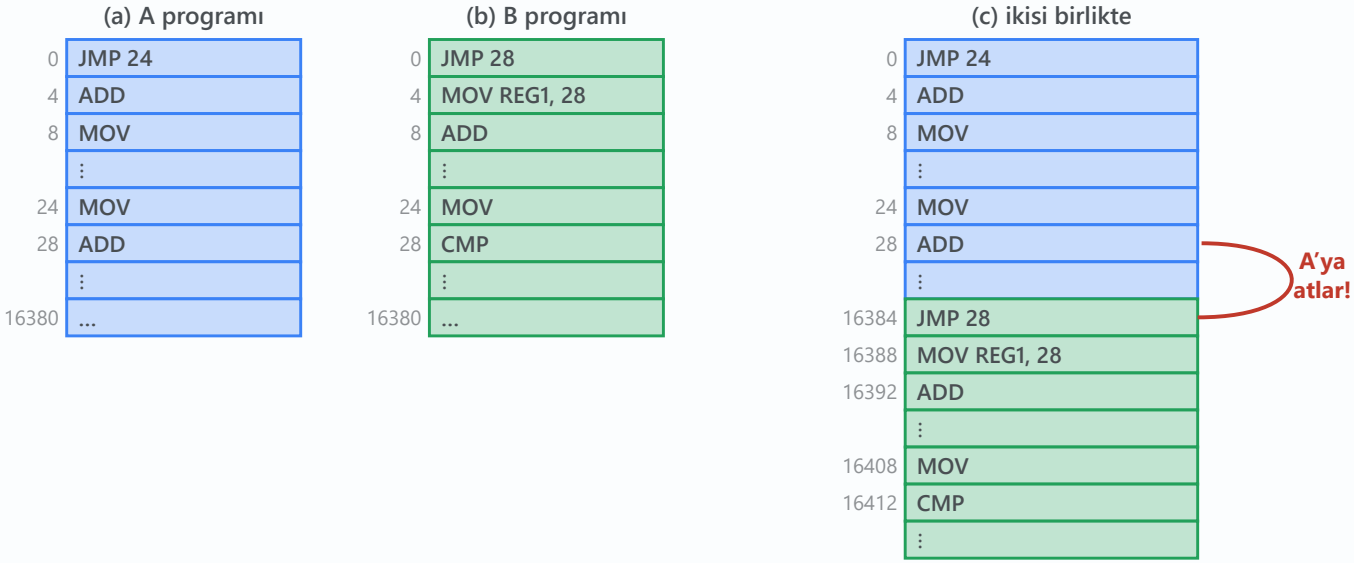
A ve B 16 KB'lık iki program; ikisi de **0 adresinden** başlayacağı varsayılarak derlendi.

Yer Değiştirme Problemi



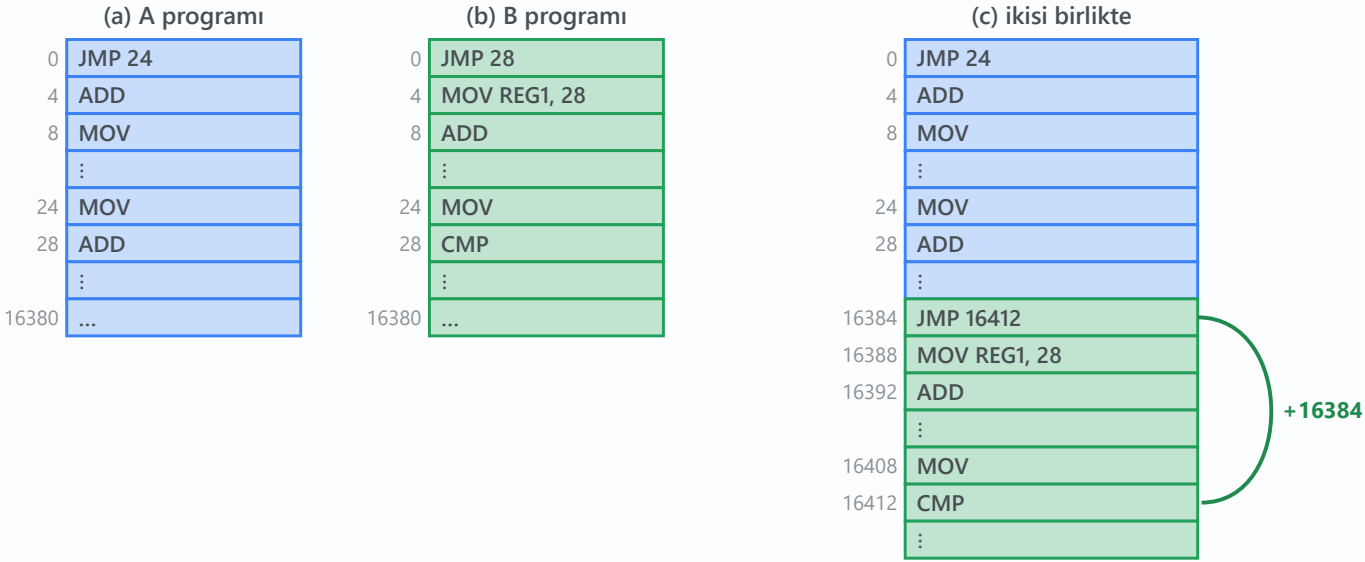
İşletim sistemi önce A'yı 0'a, ardından B'yi hemen üstüne, **16384**'e yükler.

Yer Değiştirme Problemi



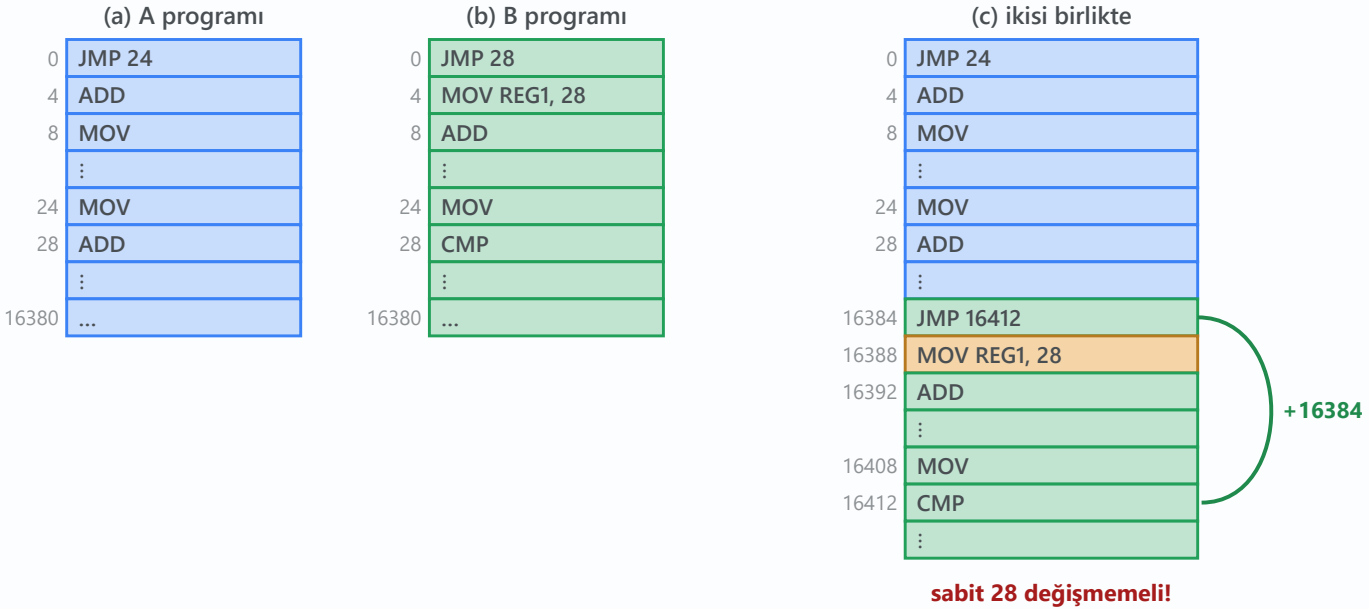
B'nin ilk komutu `JMP 28` mutlak adres kullanır: fiziksel 28'e, yani **A'nın** `ADD` komutuna atlar ve program çöker.

Yer Değiştirme Problemi



Statik yer değiştirme (IBM 360): yükleyici her adres sözcüğüne başlangıç adresini ekler: `JMP 16412` . Yükleme yavaşlar.

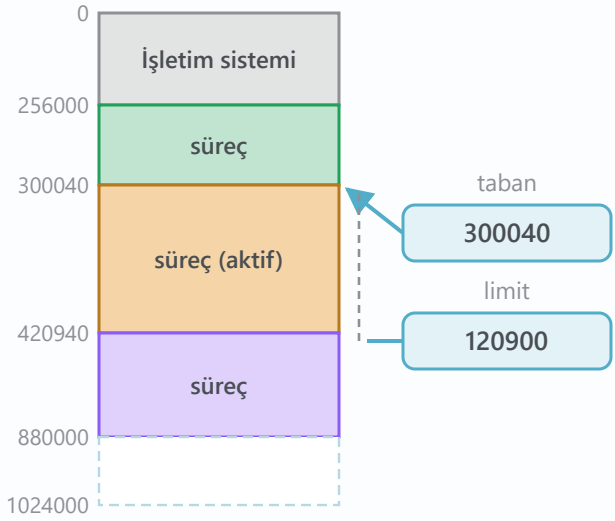
Yer Değiştirme Problemi



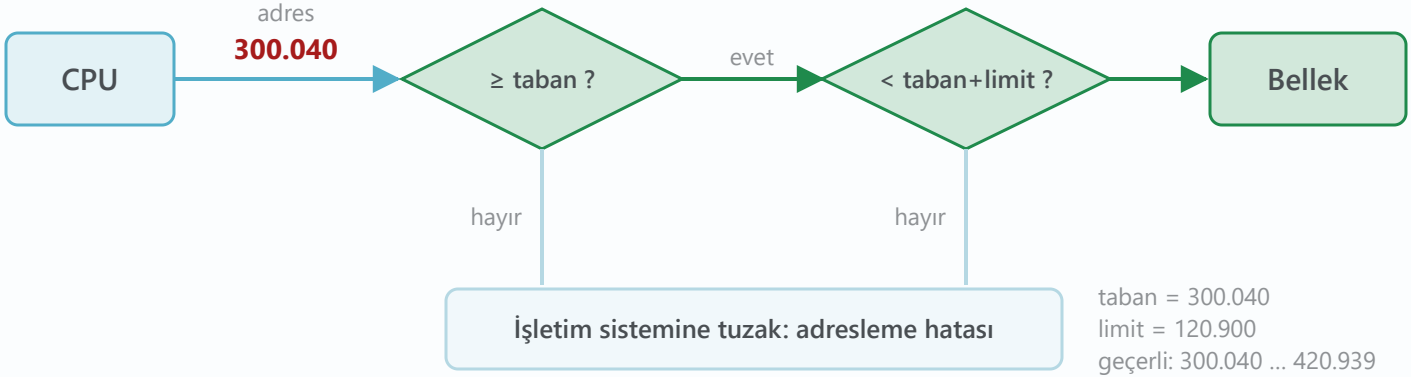
Ama `MOV REG1, 28`'deki 28 bir **sabittir**, değişmemelidir. Yükleyici hangi sözcüğün adres olduğunu bilmek için ek **yer değiştirme bilgisi** ister.

Adres Alanı: Taban ve Limit Yazmaçları

- **Adres alanı**: bir sürecin kullanabileceği adresler kümesi. Her sürecin kendine ait, diğerlerinden bağımsız adres alanı vardır.
- Çözülmesi gereken iki problem: **koruma** (başkasının alanına erişememe) ve **yer değiştirme** (her yere yüklenebilme).
- En basit donanım: **taban** (başlangıç fiziksel adresi) ve **limit** (uzunluk) yazmaçları. Program bitişik bir alana olduğu gibi yüklenir.
- Yazmaçlar yalnızca **çekirdek modunda** ayrıcalıklı komutlarla yüklenir; bağlam değiştirmede yeni sürecin değerleri konur.
- CDC 6600 böyle çalışıyordu. Intel 8088'in kesim yazmaçları ise **yalnızca taban** idi; limit olmadığından koruma yoktu.



Donanımsal Adres Koruması

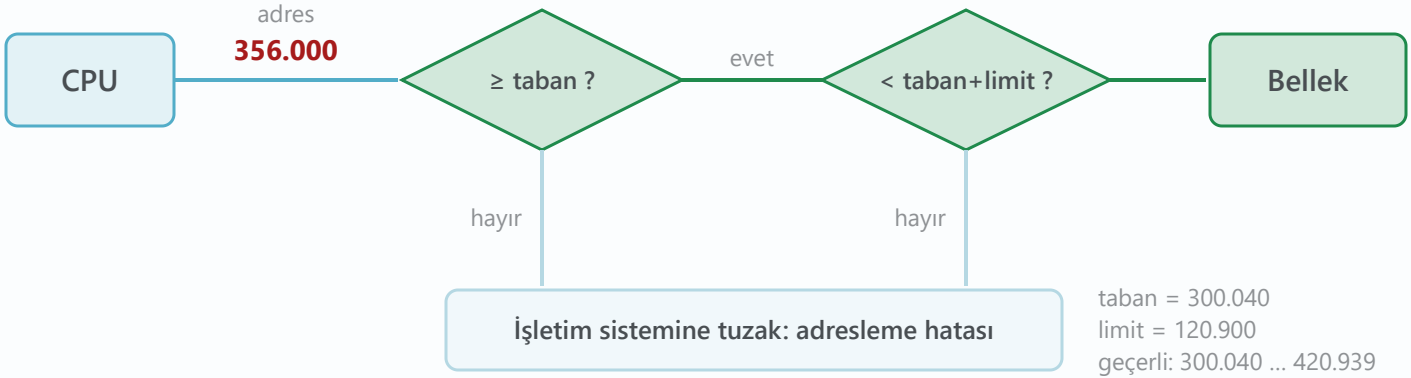


300.040 = taban: ilk geçerli adres; iki karşılaştırma da geçer, erişim yapılır.

Adım 1 / 5

Kullanıcı modundaki her erişim denetlenir. Çekirdek modunda denetim yapılmaz; işletim sistemi tüm belleğe erişebilmelidir (Silberschatz Şekil 9.2).

Donanımsal Adres Koruması

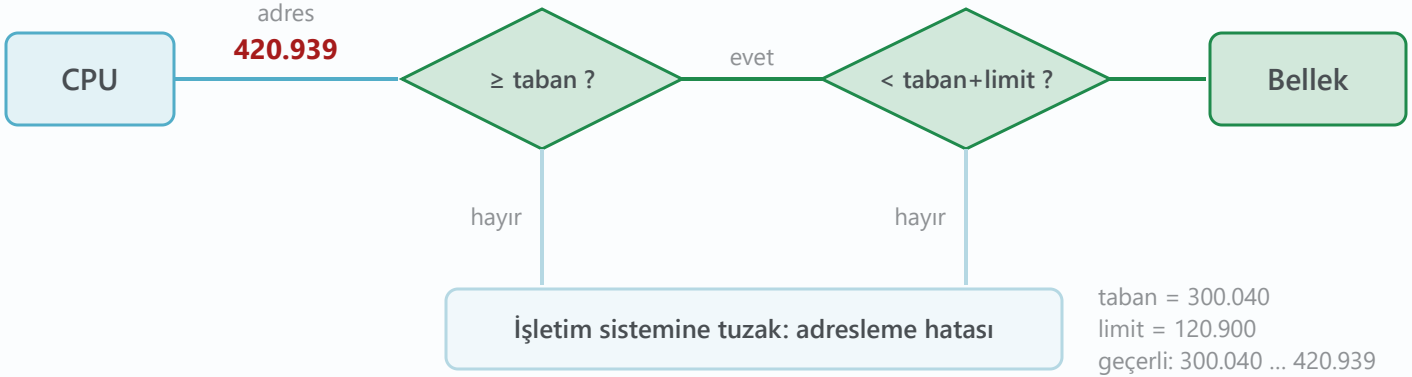


356.000 aralığın içinde; erişim yapılır. Denetim her erişimde, paralel donanımla yapılır.

Adım 2 / 5

Kullanıcı modundaki her erişim denetlenir. Çekirdek modunda denetim yapılmaz; işletim sistemi tüm belleğe erişebilmelidir (Silberschatz Şekil 9.2).

Donanımsal Adres Koruması

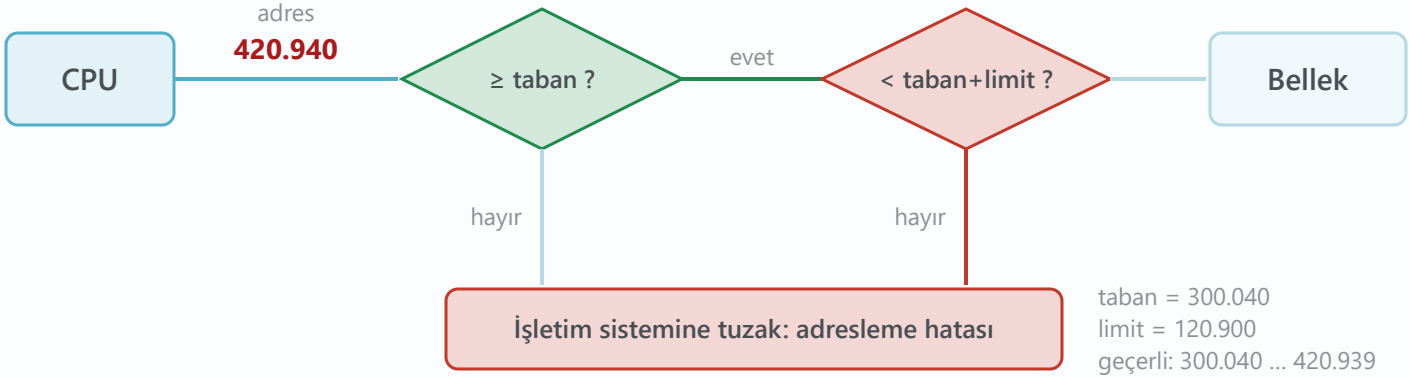


420.939 = taban + limit − 1: son geçerli adres.

Adım 3 / 5

Kullanıcı modundaki her erişim denetlenir. Çekirdek modunda denetim yapılmaz; işletim sistemi tüm belleğe erişebilmelidir (Silberschatz Şekil 9.2).

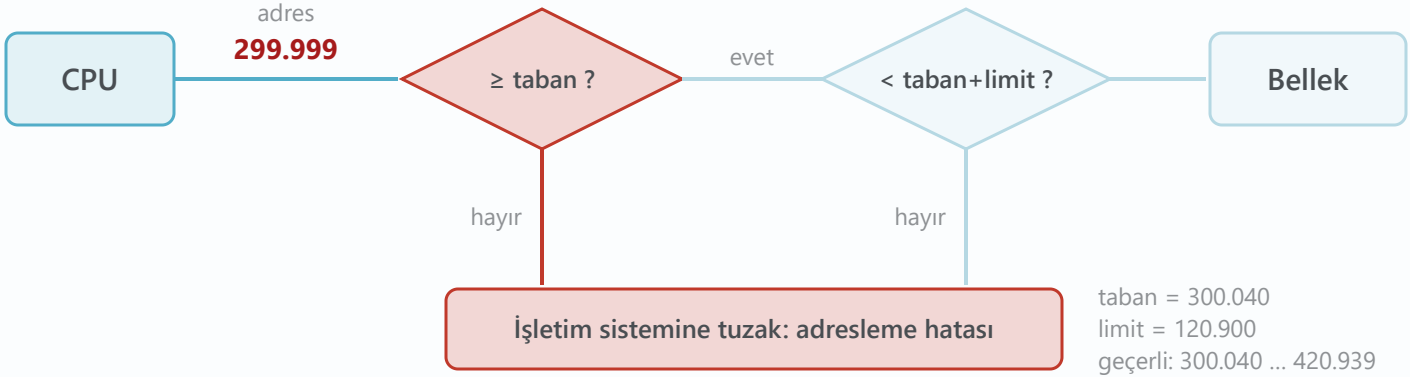
Donanımsal Adres Koruması



420.940 = taban + limit: ikinci karşılaştırma başarısız → **tuzak**. Çekirdek süreci genellikle sonlandırır (Unix'te `SIGSEGV`).

Kullanıcı modundaki her erişim denetlenir. Çekirdek modunda denetim yapılmaz; işletim sistemi tüm belleğe erişebilmelidir (Silberschatz Şekil 9.2).

Donanımsal Adres Koruması



299.999 tabandan küçük: ilk karşılaştırma başarısız → başka sürecin alanı korunur.

Adım 5 / 5

Kullanıcı modundaki her erişim denetlenir. Çekirdek modunda denetim yapılmaz; işletim sistemi tüm belleğe erişebilmelidir (Silberschatz Şekil 9.2).

Adres Bağlama Zamanları

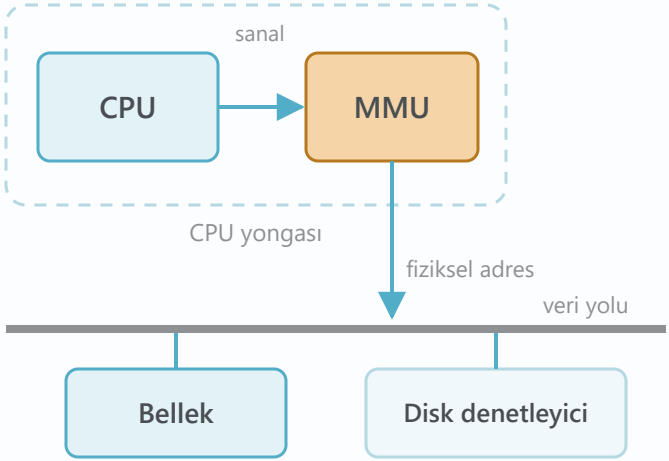


Bağlama zamanı	Üretilen kod	Konum değişirse	Donanım
Derleme	Mutlak kod (ör. MS-DOS `.COM`)	yeniden derle	gerekmez
Yükleme	Yeri değiştirilebilir kod; yükleyici adresleri düzeltir	yeniden yükle	gerekmez
Yürütme	Mantıksal adresler; süreç çalışırken taşınabilir	hiçbir şey	MMU (taban-limit, sayfalama)

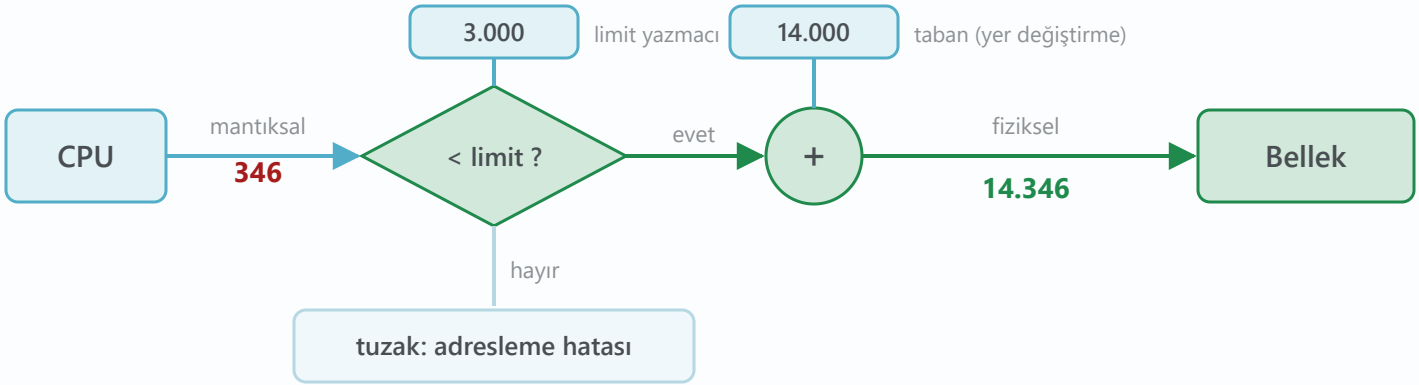
Kaynakta adresler *semboliktir* (`count`); derleyici onları *göreli* adrese ("modül başından 14 bayt"), bağlayıcı/yükleyici *mutlak* adrese (74014) bağlar. Modern genel amaçlı sistemlerin hepsi yürütme zamanı bağlama kullanır.

Mantıksal ve Fiziksel Adres, MMU

- **Mantıksal (sanal) adres**: CPU'nun ürettiği adres. **Fiziksel adres**: bellek biriminin gördüğü, adres yazmacına (MAR) yüklenen adres.
- Derleme ve yükleme zamanı bağlamada ikisi **aynıdır**; yürütme zamanı bağlamada **farklıdır**.
- Mantıksal adres alanı: programın ürettiği adresler kümesi; fiziksel adres alanı: bunlara karşılık gelen fiziksel adresler.
- Çalışma anındaki dönüşümü **MMU** (bellek yönetim birimi) donanımı yapar. Program fiziksel adresleri **hiç görmez**.
- Bugünkü işlemcilerde MMU, TLB'leri ve sayfa tablosu yürüyücüsüyle birlikte her çekirdeğin içindedir.



MMU ile Dinamik Yer Değiştirme

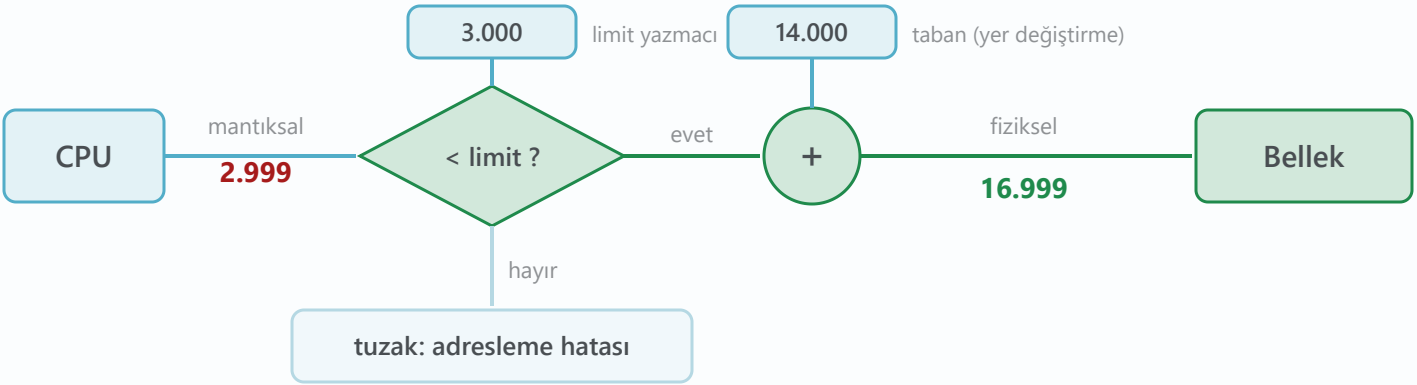


Mantıksal **346** < 3.000 → fiziksel 346 + 14.000 = **14.346**. Program 0 ile 2.999 arasını görür.

Adım 1 / 4

Silberschatz'ın *yer değiştirme yazmacı* Tanenbaum'un *taban yazmacıdır*. Her erişimde bir toplama ve bir karşılaştırma gerekir; toplama, elde yayılımı yüzünden özel devre olmadan yavaştır.

MMU ile Dinamik Yer Değiştirme

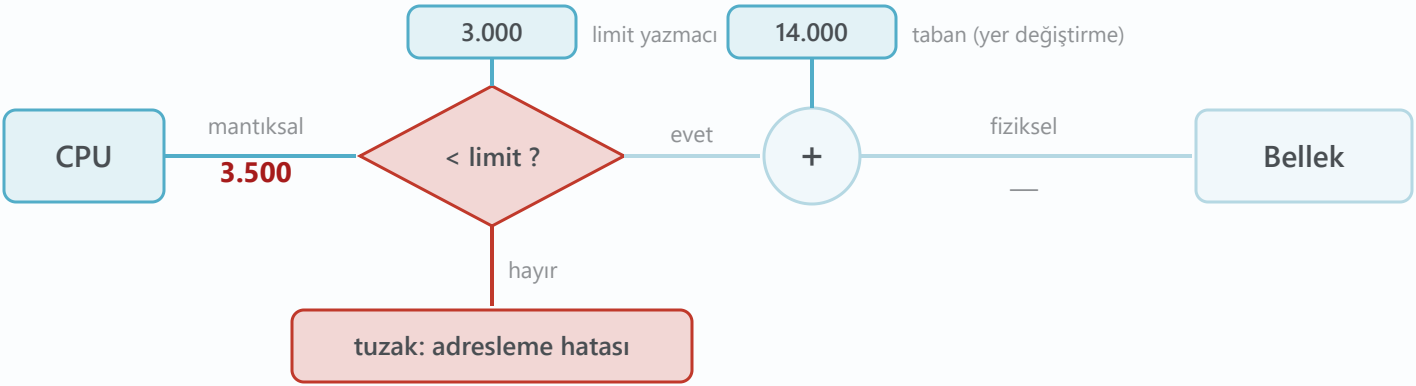


2.999 son geçerli mantıksal adres → **16.999**.

Adım 2 / 4

Silberschatz'ın *yer değiştirme yazmacı* Tanenbaum'un *taban yazmacıdır*. Her erişimde bir toplama ve bir karşılaştırma gerekir; toplama, elde yayılımı yüzünden özel devre olmadan yavaştır.

MMU ile Dinamik Yer Değiştirme

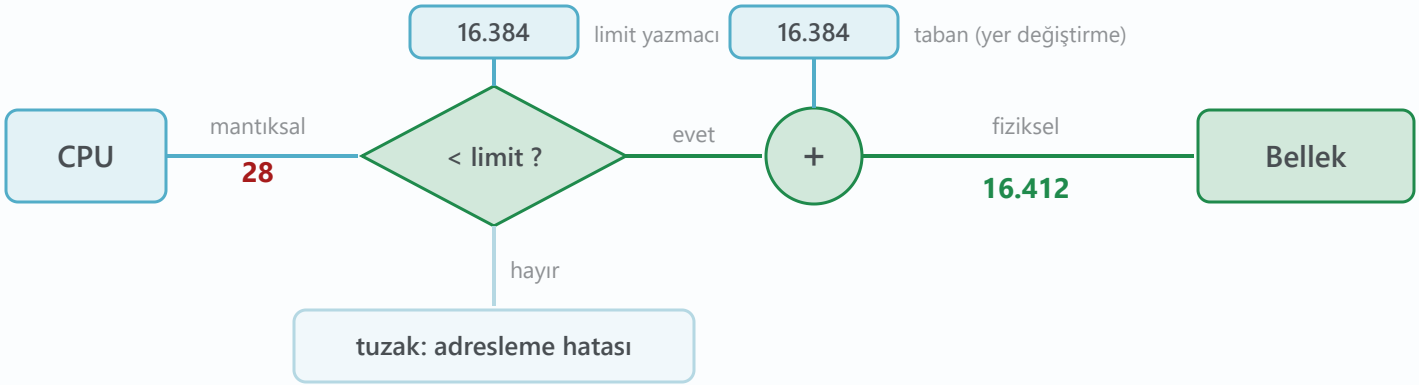


3.500 ≥ limit → toplama yapılmadan **tuzak**. Karşılaştırma mantıksal adres üzerinden olduğu için taban değeri ne olursa olsun aynıdır.

Adım 3 / 4

Silberschatz'ın *yer değiştirme yazmacı* Tanenbaum'un *taban yazmacıdır*. Her erişimde bir toplama ve bir karşılaştırma gerekir; toplama, elde yayılımı yüzünden özel devre olmadan yavaştır.

MMU ile Dinamik Yer Değiştirme



Önceki örnekteki B programı: taban = limit = 16.384. `JMP 28` değişmeden kalır, MMU 28 + 16.384 = **16.412** üretir. Statik yer değiştirmeye gerek kalmaz.

Adım 4 / 4

Silberschatz'ın *yer değiştirme yazmacı* Tanenbaum'un *taban yazmacıdır*. Her erişimde bir toplama ve bir karşılaştırma gerekir; toplama, elde yayılımı yüzünden özel devre olmadan yavaştır.

Dinamik Yükleme

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

int main(void) {
  void *h = dlopen("libm.so.6", RTLD_LAZY);
  if (!h) { puts(dlerror()); return 1; }
  double (*kok)(double) =
      (double (*)(double)) dlsym(h, "sqrt");
  printf("%.3f\n", kok(2.0));
  dlclose(h);
  return 0;
}
```



Program başladı: adres alanında yalnız `main`, `libc`, heap ve yığın (stack) var; **libm yüklü değil**.

Adım 1 / 5

DEĞİŞKENLER

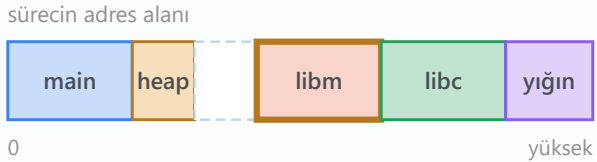
`h = –` `kok = –`

ÇIKTI

Rutin çağrılana kadar belleğe alınmaz; kullanılmayan kod hiç yüklenmez. Donanım desteği gerekmez, program böyle tasarlanır; işletim sistemi kütüphane sağlar (`dlopen`, Windows'ta `LoadLibrary` / `GetProcAddress`). glibc 2.34'ten beri `-ldl` gerekmez.

Dinamik Yükleme

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3
4  int main(void) {
▶    void *h = dlopen("libm.so.6", RTLD_LAZY);
6    if (!h) { puts(dlerror()); return 1; }
7    double (*kok)(double) =
8        (double (*)(double)) dlsym(h, "sqrt");
9    printf("%.3f\n", kok(2.0));
10   dlclose(h);
11   return 0;
12 }
```



`dlopen` : dinamik bağlayıcı (`ld.so`) dosyayı bulur, `mmap` ile adres alanına eşler. Yükleme **şimdi**, programın isteğiyle olur.

Adım 2 / 5

DEĞİŞKENLER

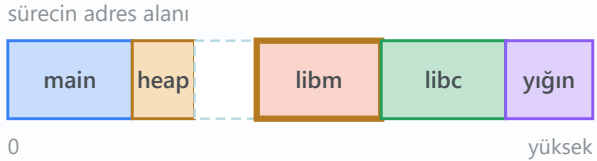
`h = tanıtıcı` `kok = –`

ÇIKTI

Rutin çağrılana kadar belleğe alınmaz; kullanılmayan kod hiç yüklenmez. Donanım desteği gerekmez, program böyle tasarlanır; işletim sistemi kütüphane sağlar (`dlopen` , Windows'ta `LoadLibrary` / `GetProcAddress`). glibc 2.34'ten beri `-ldl` gerekmez.

Dinamik Yükleme

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3
4  int main(void) {
5    void *h = dlopen("libm.so.6", RTLD_LAZY);
6    if (!h) { puts(dlerror()); return 1; }
▶    double (*kok)(double) =
▶        (double (*)(double)) dlsym(h, "sqrt");
9    printf("%.3f\n", kok(2.0));
10   dlclose(h);
11   return 0;
12 }
```



`dlsym` sembol tablosunda `sqrt`'u arar ve o sürecin adres alanındaki adresini döndürür.

Adım 3 / 5

DEĞİŞKENLER

`h = tanıtıcı` `kok = &sqrt`

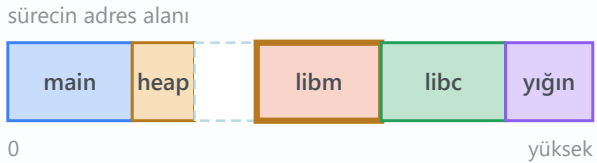
ÇIKTI

Rutin çağrılana kadar belleğe alınmaz; kullanılmayan kod hiç yüklenmez. Donanım desteği gerekmez, program böyle tasarlanır; işletim sistemi kütüphane sağlar (`dlopen`, Windows'ta `LoadLibrary` / `GetProcAddress`). glibc 2.34'ten beri `-ldl` gerekmez.

Dinamik Yükleme

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

int main(void) {
  void *h = dlopen("libm.so.6", RTLD_LAZY);
  if (!h) { puts(dlerror()); return 1; }
  double (*kok)(double) =
      (double (*)(double)) dlsym(h, "sqrt");
  printf("%.3f\n", kok(2.0));
  dlclose(h);
  return 0;
}
```



Fonksiyon işaretçisiyle çağrı yapılır; çıktı `1.414`.

Adım 4 / 5

DEĞİŞKENLER

`h = tanıtıcı` `kok = &sqrt`

ÇIKTI

```
1.414
```

Rutin çağrılana kadar belleğe alınmaz; kullanılmayan kod hiç yüklenmez. Donanım desteği gerekmez, program böyle tasarlanır; işletim sistemi kütüphane sağlar (`dlopen`, Windows'ta `LoadLibrary` / `GetProcAddress`). glibc 2.34'ten beri `-ldl` gerekmez.

Dinamik Yükleme

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3
4  int main(void) {
5    void *h = dlopen("libm.so.6", RTLD_LAZY);
6    if (!h) { puts(dlerror()); return 1; }
7    double (*kok)(double) =
8        (double (*)(double)) dlsym(h, "sqrt");
9    printf("%.3f\n", kok(2.0));
▶    dlclose(h);
11   return 0;
12 }
```



`dlclose` : referans sayısı 0'a inince kütüphane adres alanından çıkarılır.

Adım 5 / 5

DEĞİŞKENLER

`h = kapalı` `kok = –`

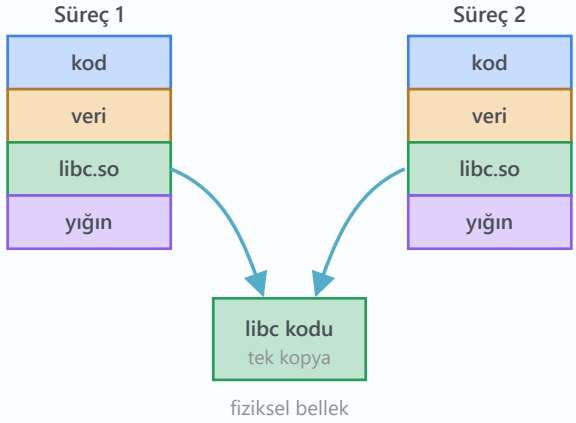
ÇIKTI

```
1.414
```

Rutin çağrılana kadar belleğe alınmaz; kullanılmayan kod hiç yüklenmez. Donanım desteği gerekmez, program böyle tasarlanır; işletim sistemi kütüphane sağlar (`dlopen` , Windows'ta `LoadLibrary` / `GetProcAddress`). glibc 2.34'ten beri `-ldl` gerekmez.

Statik ve Dinamik Bağlama

	Statik bağlama	Dinamik bağlama
Kim, ne zaman?	Bağlayıcı (`ld`), derlemeden sonra	Dinamik bağlayıcı (`ld.so`), yüklemede/ çalışırken
Dosya	`.a` , `.lib`	`.so` , `.dll` , `.dylib`
Bellek	Her programda ayrı kopya	Kod sayfaları bir kez bellekte, paylaşılır
Güncelleme	Yeniden bağlamak gerekir	Kütüphaneyi değiştirmek yeter (sürüm: *soname*)
Risk	Büyük dosya, güvenlik yaması gecikir	Sürüm uyuşmazlığı (“DLL cehennemi”)



- Statik: kütüphane rutinleri ikili dosyaya kopyalanır (`ld *.o -lc -lm`).
- Dinamik: dosyada yalnızca bir **taslak (stub)** kalır; adres yürütme zamanında çözülür.

Tembel Bağlama: PLT ve GOT



Derleyici `printf`'in adresini bilmez; çağrı küçük bir **PLT** (procedure linkage table) taslağına gider. Kitaplardaki *taslak (stub)* budur.

Kitaplardaki "taslak kendini rutinin adresiyle değiştirir" anlatımı kavramsaldır; ELF sistemlerinde değişen kod değil, GOT girdisidir. Bunu işletim sistemi değil kullanıcı alanındaki dinamik bağlayıcı yapar.

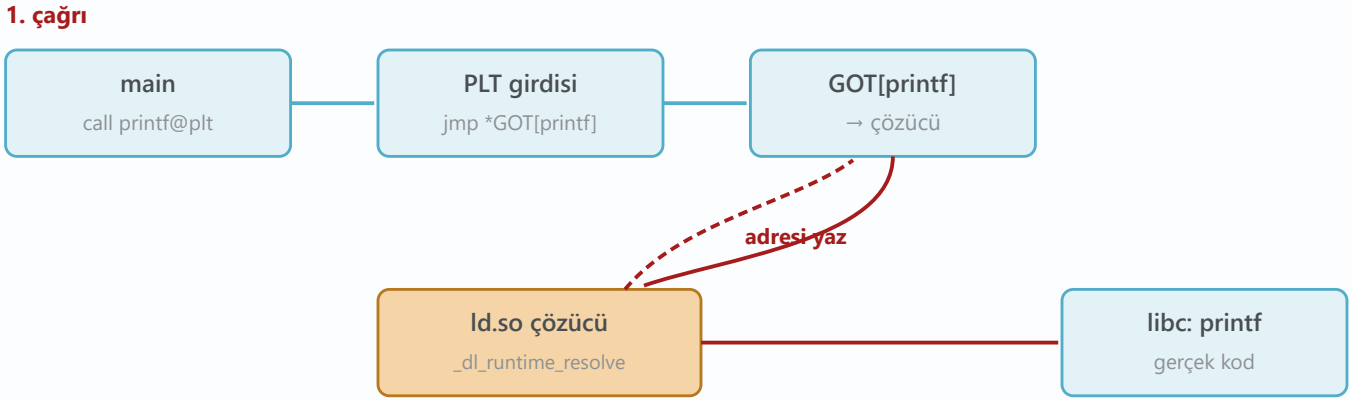
Tembel Bağlama: PLT ve GOT



PLT, **GOT** (global offset table) içindeki adrese dolaylı atlar. Başlangıçta GOT girdisi çözücüyü gösterir.

Kitaplardaki “taslak kendini rutinin adresiyle değiştirir” anlatımı kavramsaldır; ELF sistemlerinde değişen kod değil, GOT girdisidir. Bunu işletim sistemi değil kullanıcı alanındaki dinamik bağlayıcı yapar.

Tembel Bağlama: PLT ve GOT



`ld.so` çözücüsü sembolü yüklü kütüphanelerde arar, bulduğu adresi **GOT'a yazar**.

Kitaplardaki "taslak kendini rutinin adresiyle değiştirir" anlatımı kavramsaldır; ELF sistemlerinde değişen kod değil, GOT girdisidir. Bunu işletim sistemi değil kullanıcı alanındaki dinamik bağlayıcı yapar.

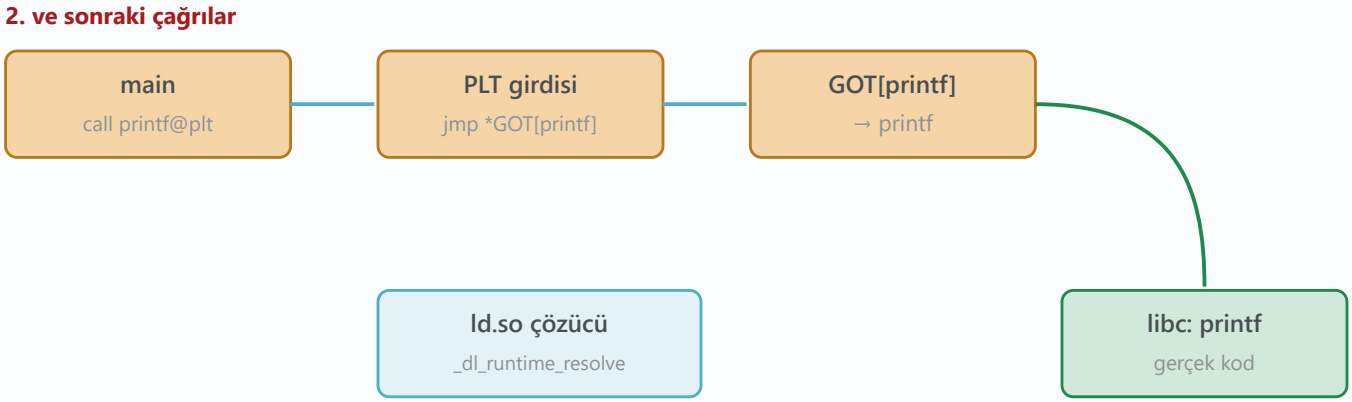
Tembel Bağlama: PLT ve GOT



Çözücü `printf`'e atlar. Kod sayfaları değişmedi; yalnızca sürece özel GOT sayfası değişti, bu yüzden kod paylaşılabilir kalır.

Kitaplardaki "taslak kendini rutinin adresiyle değiştirir" anlatımı kavramsaldır; ELF sistemlerinde değişen kod değil, GOT girdisidir. Bunu işletim sistemi değil kullanıcı alanındaki dinamik bağlayıcı yapar.

Tembel Bağlama: PLT ve GOT



Sonraki çağrılar PLT → GOT → `printf` : yalnızca bir dolaylı atlama. Güvenlik için birçok dağıtım `-z now` (tam RELRO) ile tüm sembolleri yüklemede çözüp GOT'u salt okunur yapar.

Kitaplardaki "taslak kendini rutinin adresiyle değiştirir" anlatımı kavramsaldır; ELF sistemlerinde değişen kod değil, GOT girdisidir. Bunu işletim sistemi değil kullanıcı alanındaki dinamik bağlayıcı yapar.

Kütüphane Oluşturma: Statik ve Paylaşımlı

Statik kütüphane

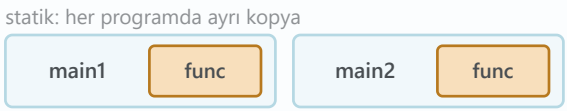
```
$ gcc -c func.c -o func.o
$ ar rcs libfunc.a func.o
$ gcc main.c -o main -static -L. -lfunc
$ ./main
```

- `ar` nesne dosyalarını arşivler; bağlayıcı yalnızca gereken nesneleri kopyalar.
- `-static` libc'nin de statik sürümünü (`libc.a`) ister.

Paylaşımlı kütüphane

```
$ gcc -fPIC -c func.c -o func.o
$ gcc -shared -o libfunc.so func.o
$ gcc main.c -o main -L. -lfunc
$ export LD_LIBRARY_PATH=$(pwd)
$ ./main
$ ldd ./main   (libfunc.so => ./libfunc.so)
```

- `-fPIC` : **konumdan bağımsız kod**; mutlak adres yerine PC'ye göreli adresleme, veriye GOT üzerinden erişim.
- Böylece aynı fiziksel sayfa her süreçte **farklı sanal adrese** eşlenebilir (ASLR).
- `LD_LIBRARY_PATH` yerine kalıcı çözüm: `-Wl,-rpath,'$ORIGIN'` ya da `ldconfig` .

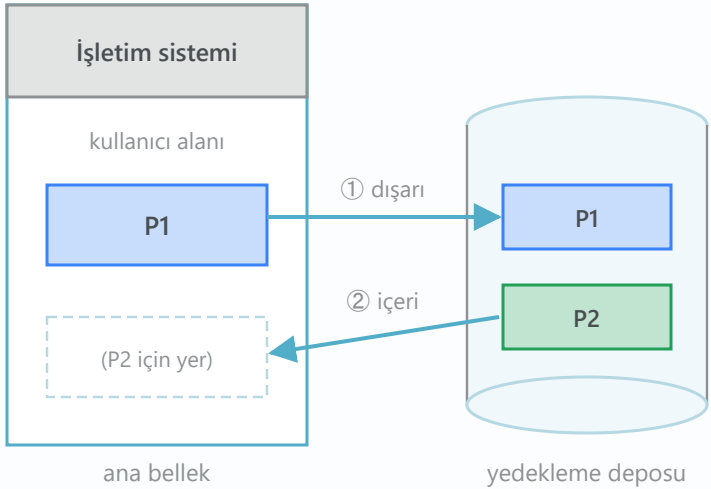




Modern dağıtımlar programları da konumdan bağımsız (PIE) derler; böylece programın kendisi de ASLR ile rastgele adrese yüklenir.

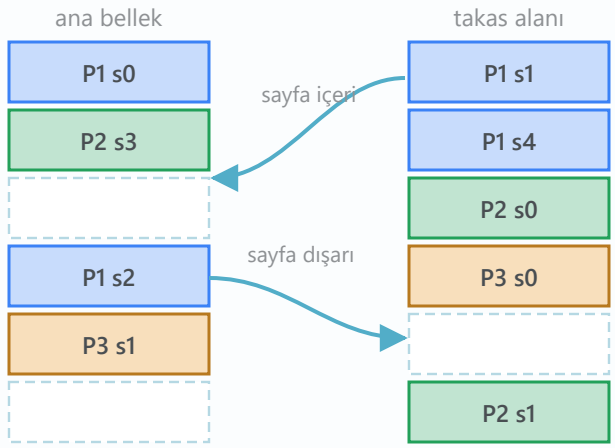
Takas (Swapping)

- **Standart takas**: bir sürecin *tamamı* geçici olarak yedekleme deposuna (disk) yazılır, sonra geri getirilir.
- Toplam bellek ihtiyacı fiziksel belleği aşabilir; çoklu programlama derecesi artar.
- Geri gelen süreç başka adrese yerleşebilir → **yürütme zamanı bağlama** gerekir.
- Bedel: 100 MB süreç, 50 MB/s disk → 2 s dışarı + 2 s içeri = **4 s** (Silberschatz).
- Bekleyen G/Ç'si olan süreç takas edilmez (DMA eski adrese yazar) ya da G/Ç çekirdek tamponlarından yapılır.



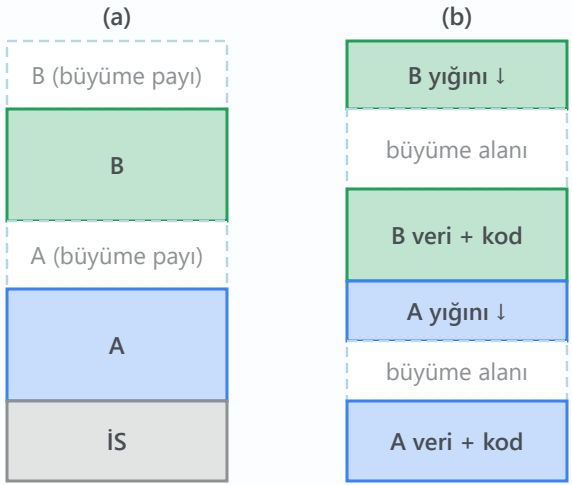
Sayfa Takası ve Modern Sistemler

- Bütün süreci taşımak bugün kabul edilemez: 1 GiB süreç, 3 GB/s NVMe → gidiş-dönüş ≈ **0,72 s**.
- Linux, Windows, macOS **sayfa** takas eder: yalnızca uzun süredir kullanılmayan (soğuk) sayfalar dışarı gider, 4 KiB'lık bir sayfa onlarca µs'de gelir.
- Diske gitmeden önce **sıkıştırma**: Linux zswap ve zram, Windows bellek sıkıştırma, macOS sıkıştırıcı.
- Mobil: flash ömrü ve enerji yüzünden geleneksel takas yok ya da sınırlı; Android zram + `lmkd`, iOS sıkıştırma + uygulama sonlandırma (jetsam).



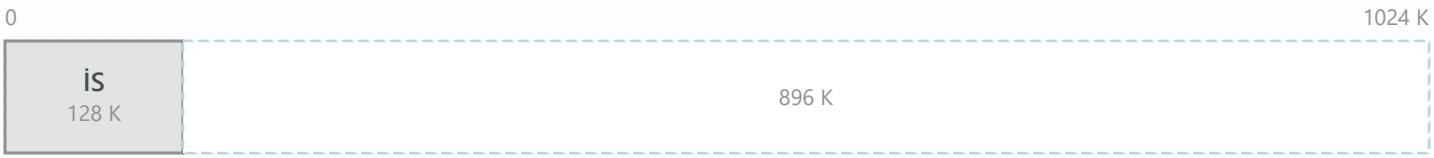
Bitişik Bellek Tahsisi

- Bellek iki bölgeye ayrılır: **işletim sistemi** ve **kullanıcı süreçleri**. Linux ve Windows çekirdeği yüksek adreslere koyar.
- Her süreç tek, **bitişik** bir bölgede; koruma taban-limit ile.
- **Sabit bölümleme** (IBM OS/360 MFT): bölüm sayısı = çoklu programlama derecesi; küçük süreç büyük bölümde → *iç parçalanma*.
- **Değişken bölümleme** (MVT): bölüm süreç kadar; boş bloklara **delik (hole)** denir.
- Süreçler büyür: veri kesimi (`malloc`) ve yığın için **pay bırakılır**. Takastan dönen süreç payıyla birlikte değil, kullandığı kadar getirilir.



BİTİŞİK TAHSİS · ADIM ADIM

Değişken Bölümleme



delikler: 896 K @ 128

Bellekte yalnız işletim sistemi var.

Adım 1 / 8

- İşletim sistemi tahsisli bölümleri ve delikleri bir tabloda (ya da listede) tutar.
- Süreç gelince yeterince büyük bir delik bulunur, artan kısım delik olarak kalır; süreç bitince alanı deliğe döner ve komşularla birleşir.

Değişken Bölümleme



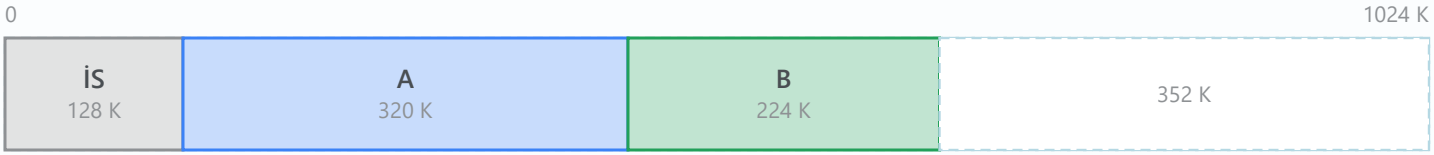
delikler: 576 K @ 448

A (320 K) gelir; ilk uygun boşluğa yerleşir.

Adım 2 / 8

- İşletim sistemi tahsisli bölümleri ve delikleri bir tabloda (ya da listede) tutar.
- Süreç gelince yeterince büyük bir delik bulunur, artan kısım delik olarak kalır; süreç bitince alanı deliğe döner ve komşularla birleşir.

Değişken Bölümleme



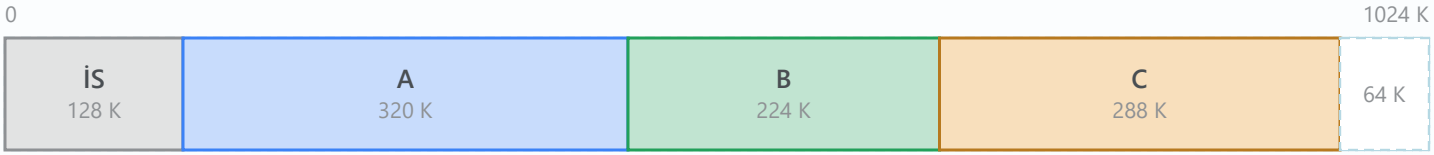
delikler: 352 K @ 672

B (224 K) gelir; ilk uygun boşluğa yerleşir.

Adım 3 / 8

- İşletim sistemi tahsisli bölümleri ve delikleri bir tabloda (ya da listede) tutar.
- Süreç gelince yeterince büyük bir delik bulunur, artan kısım delik olarak kalır; süreç bitince alanı deliğe döner ve komşularla birleşir.

Değişken Bölümleme



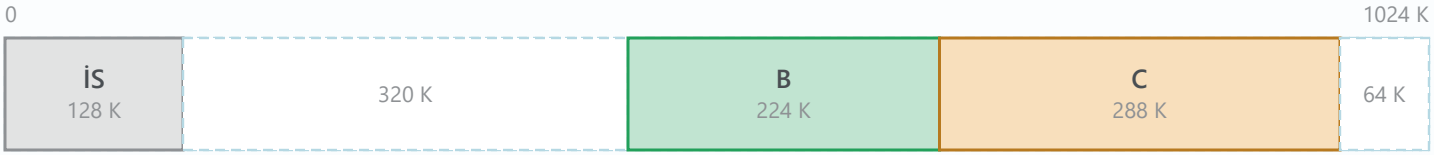
delikler: 64 K @ 960

C (288 K) gelir; ilk uygun boşluğa yerleşir.

Adım 4 / 8

- İşletim sistemi tahsisli bölümleri ve delikleri bir tabloda (ya da listede) tutar.
- Süreç gelince yeterince büyük bir delik bulunur, artan kısım delik olarak kalır; süreç bitince alanı deliğe döner ve komşularla birleşir.

Değişken Bölümleme

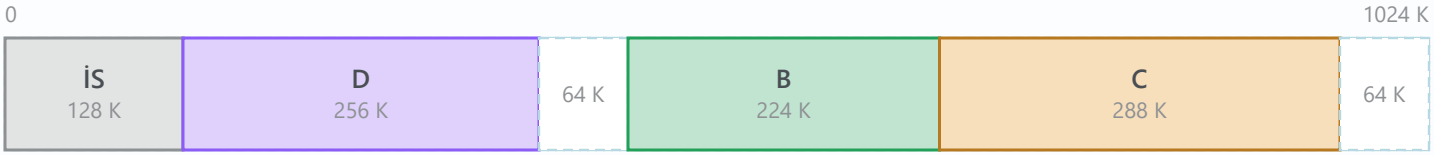


delikler: 320 K @ 128, 64 K @ 960

A sonlanır: 128–448 arası **320 K** delik olur.

- İşletim sistemi tahsisli bölümleri ve delikleri bir tabloda (ya da listede) tutar.
- Süreç gelince yeterince büyük bir delik bulunur, artan kısım delik olarak kalır; süreç bitince alanı deliğe döner ve komşularla birleşir.

Değişken Bölümleme

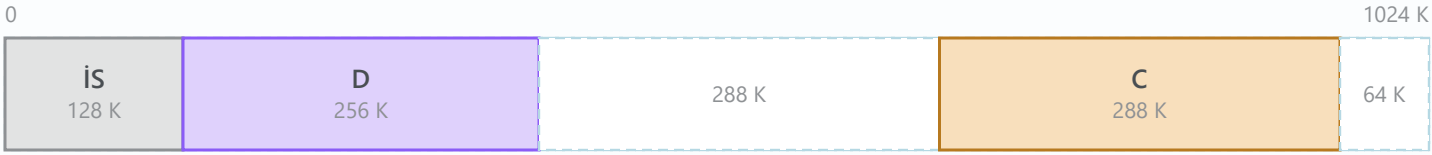


delikler: 64 K @ 384, 64 K @ 960

D (256 K) A’nın deliğine girer; geriye **64 K**’lık küçük bir delik kalır.

- İşletim sistemi tahsisli bölümleri ve delikleri bir tabloda (ya da listede) tutar.
- Süreç gelince yeterince büyük bir delik bulunur, artan kısım delik olarak kalır; süreç bitince alanı deliğe döner ve komşularla birleşir.

Değişken Bölümleme



delikler: 288 K @ 384, 64 K @ 960

B sonlanır: alanı solundaki 64 K delikle **birleşir** → 288 K. Bitişik delikler hemen birleştirilir.

- İşletim sistemi tahsisli bölümleri ve delikleri bir tabloda (ya da listede) tutar.
- Süreç gelince yeterince büyük bir delik bulunur, artan kısım delik olarak kalır; süreç bitince alanı deliğe döner ve komşularla birleşir.

Değişken Bölümleme



delikler: 96 K @ 576, 64 K @ 960

E (192 K) 288 K deliğe girer. Toplam boş 160 K ama **iki parçaya** dağılmış: dış parçalanma büyüyor.

Adım 8 / 8

- İşletim sistemi tahsisli bölümleri ve delikleri bir tabloda (ya da listede) tutar.
- Süreç gelince yeterince büyük bir delik bulunur, artan kısım delik olarak kalır; süreç bitince alanı deliğe döner ve komşularla birleşir.

BİTİŞİK TAHSİS

Yerleştirme Algoritmaları

İlk uyan (first fit)
Liste baştan taranır, ilk yeterli delikte durulur. Hızlı; başta küçük delikler birikir.

Sonraki uyan (next fit)
Arama son kalınan yerden sürer; ilk uyandan biraz *kötü* çıkar (Bays, 1977).

En iyi uyan (best fit)
Bütün liste taranır, ihtiyaca en yakın delik seçilir. Yavaş; işe yaramaz **çok küçük** delikler bırakır.

En kötü uyan (worst fit)
Kalan parça işe yarasın diye en büyük delik seçilir. Benzetimlerde iyi sonuç vermez.

Hızlı uyan (quick fit)
Sık istenen boyutlar için **ayrı listeler** (4 K, 8 K, 12 K …). Bulmak hızlı, birleştirmek pahalı.

Ortak iyileştirme
Süreçler ve delikler için ayrı listeler. Bugünkü `malloc` gerçeklemeleri (jemalloc, tcmalloc) boyut sınıfları kullanır.

Benzetimlere göre (Silberschatz, Knuth) zaman ve bellek kullanımında ilk uyan ve en iyi uyan, en kötü uyandan iyidir; ilk uyan genellikle daha hızlıdır.

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
4      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
6        en = i;
7    if (en != -1) delik[en] -= istek;
8    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 210 K

120	450	230	300	180
[0]	[1]	[2]	[3]	[4]

Delik listesi 120, 450, 230, 300, 180 K. İlk istek **210 K**; `en = −1` (aday yok).

Adım 1 / 12

DEĞİŞKENLER

`istek = 210` `i = –` `en = -1`

`delik = [120 450 230 300 180]`

ÇIKTI

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
▶      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
6        en = i;
7    if (en != -1) delik[en] -= istek;
8    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 210 K

120	450	230	300	180
[0]	[1]	[2]	[3]	[4]

[0] = 120 < 210: sığmaz, atla.

Adım 2 / 12

DEĞİŞKENLER

`istek = 210` `i = 0` `en = -1`

`delik = [120 450 230 300 180]`

ÇIKTI

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
▶      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
▶        en = i;
7    if (en != -1) delik[en] -= istek;
8    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 210 K

120	450	230	300	180
[0]	[1]	[2]	[3]	[4]

[1] = 450 ≥ 210 ve ilk aday → `en = 1`.

Adım 3 / 12

DEĞİŞKENLER

`istek = 210` `i = 1` `en = 1`

`delik = [120 450 230 300 180]`

ÇIKTI

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
▶      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
▶        en = i;
7    if (en != -1) delik[en] -= istek;
8    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 210 K

120	450	230	300	180
[0]	[1]	[2]	[3]	[4]

[2] = 230 ≥ 210 ve 450'den küçük → yeni en iyi `en = 2`.

Adım 4 / 12

DEĞİŞKENLER

`istek = 210` `i = 2` `en = 2`

`delik = [120 450 230 300 180]`

ÇIKTI

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
▶      if (delik[i] >= istek &&
▶          (en == -1 || delik[i] < delik[en]))
6        en = i;
7    if (en != -1) delik[en] -= istek;
8    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 210 K

120	450	230	300	180
[0]	[1]	[2]	[3]	[4]

[3] = 300 sığar ama mevcut en iyi 230 daha küçük: değişmez.

Adım 5 / 12

DEĞİŞKENLER

`istek = 210` `i = 3` `en = 2`

`delik = [120 450 230 300 180]`

ÇIKTI

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
▶      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
6        en = i;
7    if (en != -1) delik[en] -= istek;
8    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 210 K

120	450	230	300	180
[0]	[1]	[2]	[3]	[4]

[4] = 180 < 210: sığmaz, atla.

Adım 6 / 12

DEĞİŞKENLER

`istek = 210` `i = 4` `en = 2`

`delik = [120 450 230 300 180]`

ÇIKTI

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
4      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
6        en = i;
▶    if (en != -1) delik[en] -= istek;
▶    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 210 K



Seçilen [2] deliğinden 210 K ayrılır; geriye **20 K**'lık küçük bir delik kalır. Dönüş: 2. İlk uyan aynı istek için 450'yi bölerdi.

Adım 7 / 12

DEĞİŞKENLER

`istek = 210` `en = 2`

`delik = [120 450 20 300 180]`

ÇIKTI

```
en_iyi(210) = 2
```

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
▶    int en = -1;
3    for (int i = 0; i < n; i++)
4      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
6        en = i;
7    if (en != -1) delik[en] -= istek;
8    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 160 K

120	450	20	300	180
[0]	[1]	[2]	[3]	[4]

İkinci istek **160 K**; delik [2] artık 20 K. Tarama yine baştan, tüm listeyi gezer.

Adım 8 / 12

DEĞİŞKENLER

`istek = 160` `i = –` `en = -1`

`delik = [120 450 20 300 180]`

ÇIKTI

```
en_iyi(210) = 2
```

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
▶      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
▶        en = i;
7    if (en != -1) delik[en] -= istek;
8    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 160 K

120	450	20	300	180
[0]	[1]	[2]	[3]	[4]

[0] = 120 sığmaz; [1] = 450 ≥ 160 ve ilk aday → `en = 1`.

Adım 9 / 12

DEĞİŞKENLER

`istek = 160` `i = 1` `en = 1`

`delik = [120 450 20 300 180]`

ÇIKTI

```
en_iyi(210) = 2
```

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
▶      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
▶        en = i;
7    if (en != -1) delik[en] -= istek;
8    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 160 K

120	450	20	300	180
[0]	[1]	[2]	[3]	[4]

[2] = 20 sığmaz; [3] = 300 ≥ 160 ve 450'den küçük → yeni en iyi `en = 3`.

Adım 10 / 12

DEĞİŞKENLER

`istek = 160` `i = 3` `en = 3`

`delik = [120 450 20 300 180]`

ÇIKTI

```
en_iyi(210) = 2
```

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
▶      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
▶        en = i;
7    if (en != -1) delik[en] -= istek;
8    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 160 K

120	450	20	300	180
[0]	[1]	[2]	[3]	[4]

[4] = 180 ≥ 160 ve 300'den küçük → yeni en iyi `en = 4`.

Adım 11 / 12

DEĞİŞKENLER

`istek = 160` `i = 4` `en = 4`

`delik = [120 450 20 300 180]`

ÇIKTI

```
en_iyi(210) = 2
```

En İyi Uyan: Kod İzleme

```
1  int en_iyi(int delik[], int n, int istek) {
2    int en = -1;
3    for (int i = 0; i < n; i++)
4      if (delik[i] >= istek &&
5          (en == -1 || delik[i] < delik[en]))
6        en = i;
▶    if (en != -1) delik[en] -= istek;
▶    return en;
9  }
10 /* delik = 120, 450, 230, 300, 180 (K) */
11 /* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

istek = 160 K



Seçilen [4] deliğinden 160 K ayrılır; geriye **20 K**'lık küçük bir delik kalır. Dönüş: 4. En iyi uyan iki kez işe yaramaz 20 K'lık delik bıraktı.

Adım 12 / 12

DEĞİŞKENLER

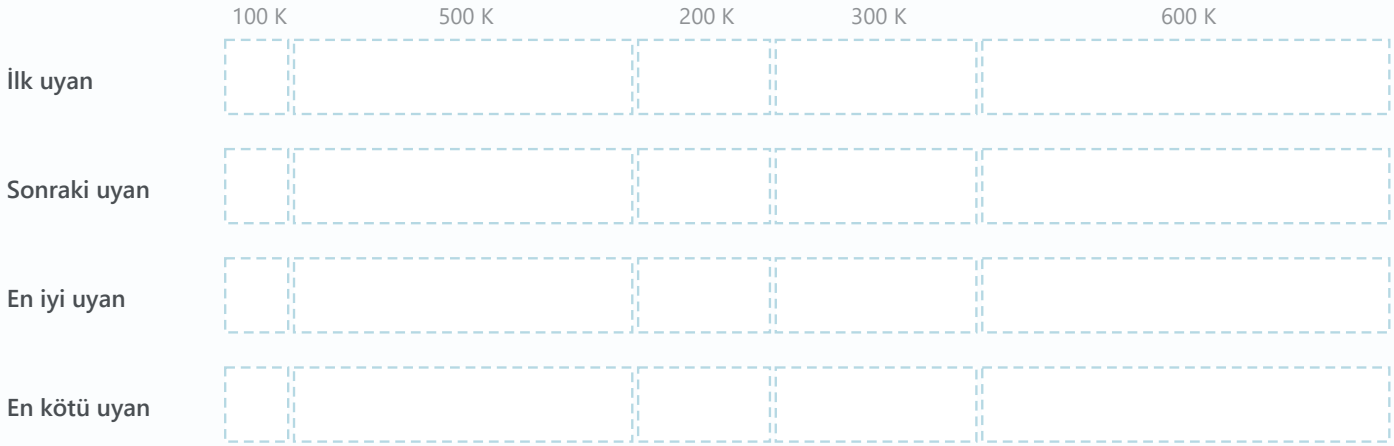
`istek = 160` `en = 4`

`delik = [120 450 20 300 20]`

ÇIKTI

```
en_iyi(210) = 2
en_iyi(160) = 4
```

Yerleştirme: Örnek

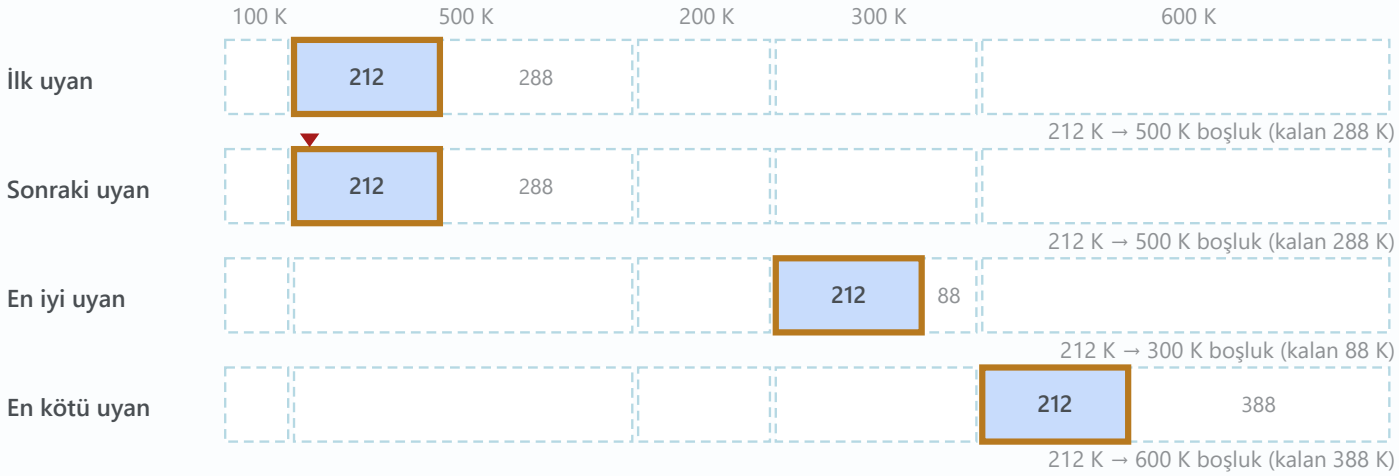


Delikler sırasıyla **100, 500, 200, 300, 600 K**. İstekler sırayla: **212, 417, 112, 426 K**.

Adım 1 / 5

Etkileşimli simülatör: Bellek Tahsisi Simülatörü: ilk/sonraki/en iyi/en kötü uyan farklı isteklerle çalıştırılır, parçalanma ölçülür.

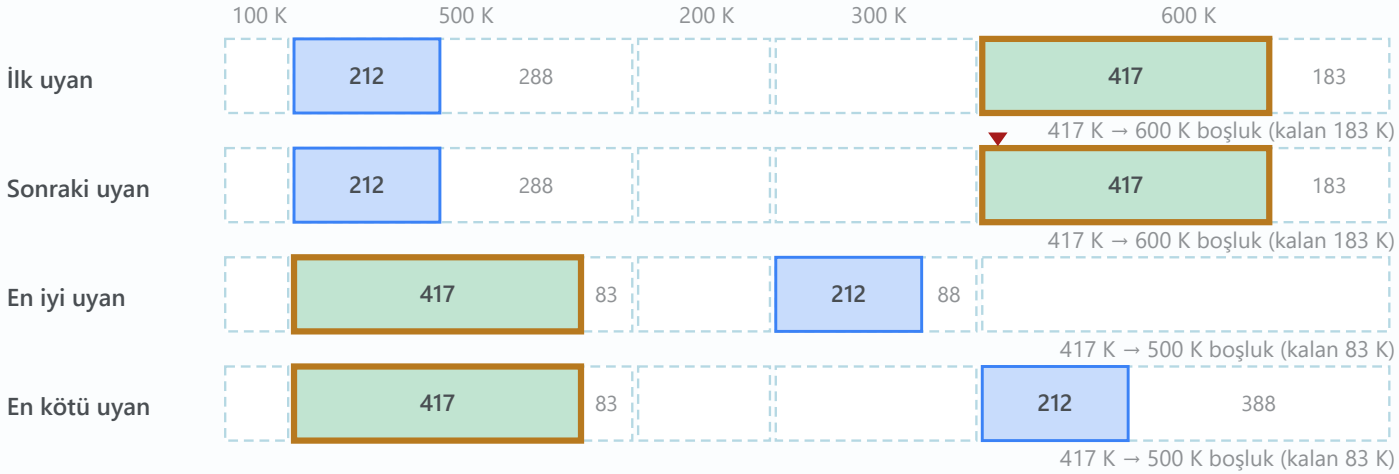
Yerleştirme: Örnek



212 K: ilk/sonraki uyan 500'e, en iyi uyan 300'e (kalan 88), en kötü uyan 600'e koyar.

Etkileşimli simülatör: Bellek Tahsisi Simülatörü: ilk/sonraki/en iyi/en kötü uyan farklı isteklerle çalıştırılır, parçalanma ölçülür.

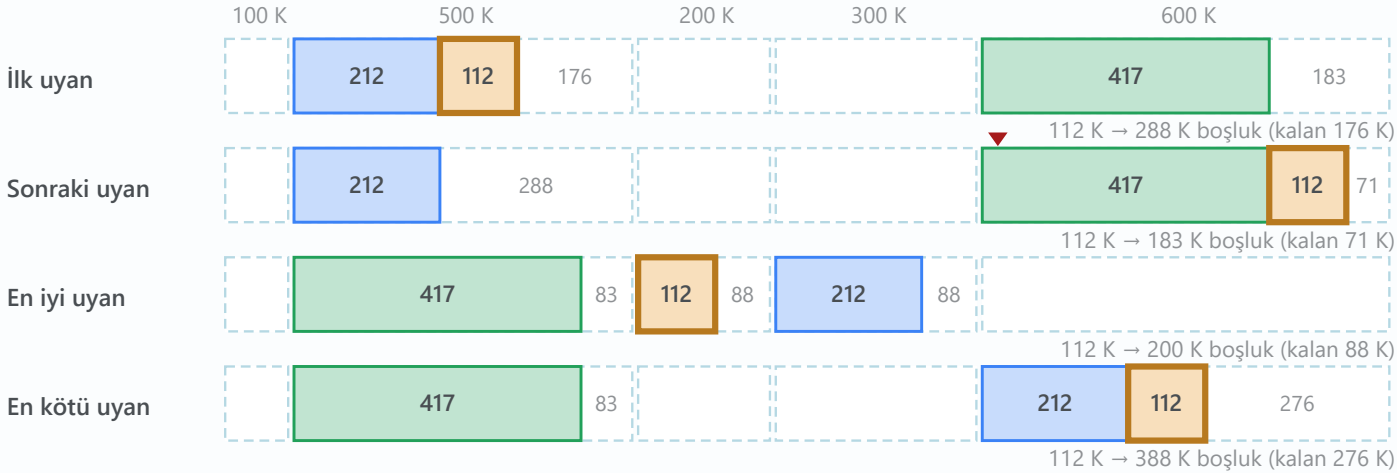
Yerleştirme: Örnek



417 K: ilk ve sonraki uyan 600'e; en iyi ve en kötü uyan 500'e (kalan 83).

Etkileşimli simülatör: Bellek Tahsisi Simülatörü: ilk/sonraki/en iyi/en kötü uyan farklı isteklerle çalıştırılır, parçalanma ölçülür.

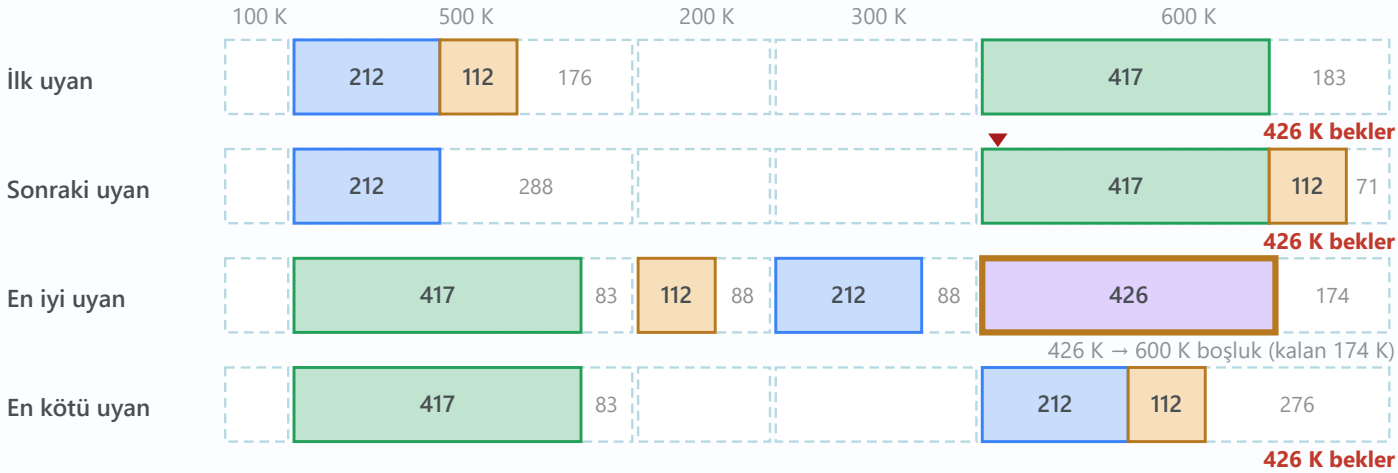
Yerleştirme: Örnek



112 K: ilk uyan 288'lik artığa; sonraki uyan son kaldığı 183'e (kalan 71); en iyi uyan 200'e; en kötü uyan 388'e.

Etkileşimli simülatör: Bellek Tahsisi Simülatörü: ilk/sonraki/en iyi/en kötü uyan farklı isteklerle çalıştırılır, parçalanma ölçülür.

Yerleştirme: Örnek

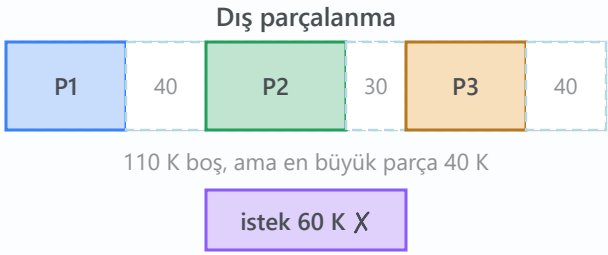


426 K: yalnız **en iyi uyan** yerleştirebilir (600 K hâlâ bütün). Diğerlerinde toplam boş alan yeter ama tek parça yok: **dış parçalanma**.

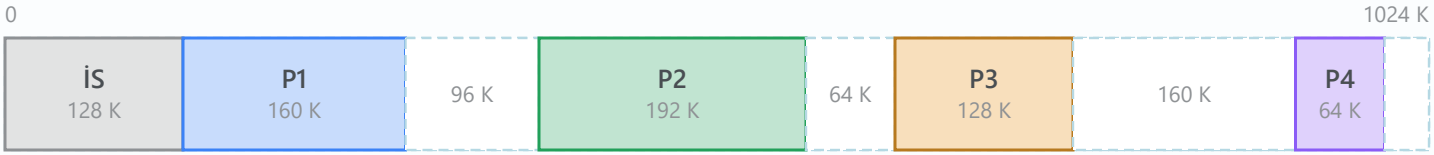
Etkileşimli simülatör: Bellek Tahsisi Simülatörü: ilk/sonraki/en iyi/en kötü uyan farklı isteklerle çalıştırılır, parçalanma ölçülür.

İç ve Dış Parçalanma

- **Dış parçalanma**: isteği karşılayacak toplam boş bellek var, ama **bitişik değil**. Değişken bölümlemenin sorunudur.
- **%50 kuralı** (Knuth): ilk uyan ile N tahsisli blok varken yaklaşık 0,5 N blok parçalanmaya gider → belleğin **üçte biri** kullanılamaz.
- **İç parçalanma**: verilen blok istenenden büyüktür; fark bloğun *içinde* boşa gider. Sabit bölümleme, sayfalama ve buddy'de görülür.
- Çözümler: **sıkıştırma** (dış), sürecin adres alanının **bitişik olmamasına** izin vermek (sayfalama, kesimleme).



Sıkıştırma (Compaction)

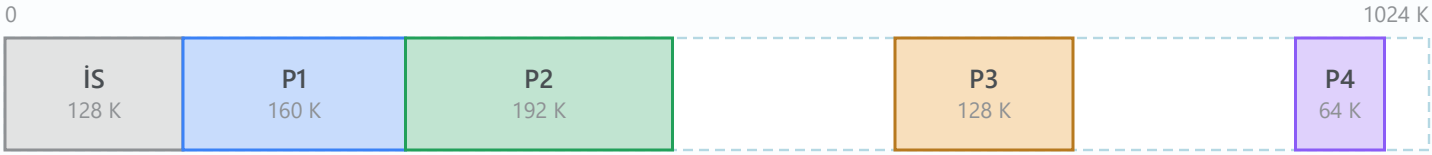


Başlangıç: dört süreç arasında 96 + 64 + 160 + 32 = **352 K** boş alan, ama en büyük delik 160 K.

Adım 1 / 4

- **Sıkıştırma (compaction)**: süreçleri bir uca kaydırıp delikleri tek bloğa toplamak. Yalnızca **yürütme zamanı bağlamada** (taban yazmacı) mümkün.
- Pahalı: Tanenbaum'a göre 16 GB bellek, 8 ns'de 8 bayt kopyalayarak ≈ **16 s**; bugünkü ≈20 GB/s kopyalama hızıyla bile ≈ 0,9 s ve bu sırada süreçler durur.
- Veri sıkıştırma (*compression*) ile karıştırılmamalı: amaç veriyi küçültmek değil, blokları *bitiştirmektir*.
- Bugün: Linux, büyük sayfa için bitişik fiziksel çerçeve üretmek amacıyla **sayfa göçü** yapar (`kcompactd`).

Sıkıştırma (Compaction)

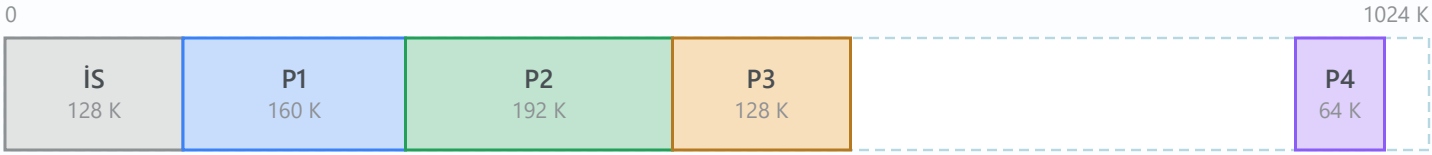


P2 aşağı kaydırılır: 384 → 288. Taşınan sürecin taban yazmacı güncellenir.

Adım 2 / 4

- **Sıkıştırma (compaction)**: süreçleri bir uca kaydırıp delikleri tek bloğa toplamak. Yalnızca **yürütme zamanı bağlamada** (taban yazmacı) mümkün.
- Pahalı: Tanenbaum'a göre 16 GB bellek, 8 ns'de 8 bayt kopyalayarak ≈ **16 s**; bugünkü ≈20 GB/s kopyalama hızıyla bile ≈ 0,9 s ve bu sırada süreçler durur.
- Veri sıkıştırma (*compression*) ile karıştırılmamalı: amaç veriyi küçültmek değil, blokları *bitiştirmektir*.
- Bugün: Linux, büyük sayfa için bitişik fiziksel çerçeve üretmek amacıyla **sayfa göçü** yapar (`kcompactd`).

Sıkıştırma (Compaction)

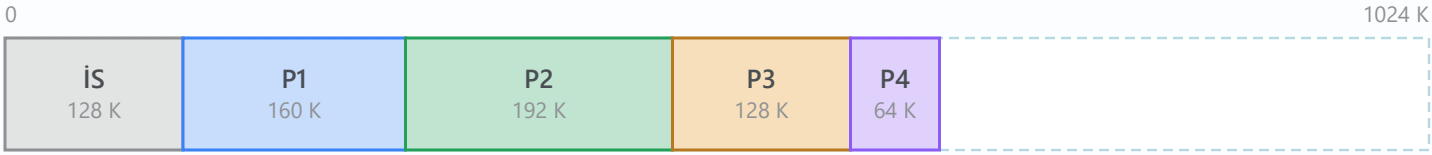


P3: 640 → 480.

Adım 3 / 4

- **Sıkıştırma (compaction)**: süreçleri bir uca kaydırıp delikleri tek bloğa toplamak. Yalnızca **yürütme zamanı bağlamada** (taban yazmacı) mümkün.
- Pahalı: Tanenbaum'a göre 16 GB bellek, 8 ns'de 8 bayt kopyalayarak ≈ **16 s**; bugünkü ≈20 GB/s kopyalama hızıyla bile ≈ 0,9 s ve bu sırada süreçler durur.
- Veri sıkıştırma (*compression*) ile karıştırılmamalı: amaç veriyi küçültmek değil, blokları *bitiştirmektir*.
- Bugün: Linux, büyük sayfa için bitişik fiziksel çerçeve üretmek amacıyla **sayfa göçü** yapar (`kcompactd`).

Sıkıştırma (Compaction)

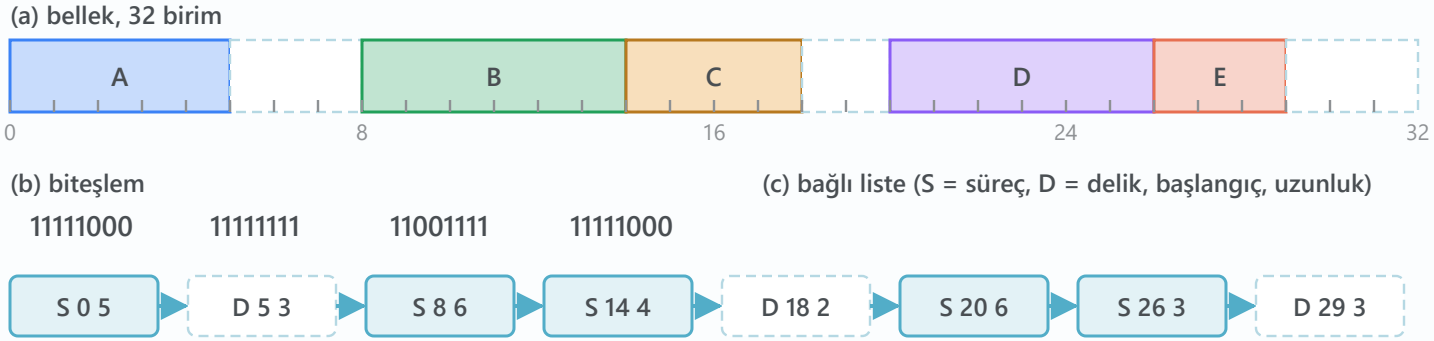


P4: 928 → 608. Artık tek bir **352 K**'lık delik var. Toplam 384 K kopyalandı.

Adım 4 / 4

- **Sıkıştırma (compaction)**: süreçleri bir uca kaydırıp delikleri tek bloğa toplamak. Yalnızca **yürütme zamanı bağlamada** (taban yazmacı) mümkün.
- Pahalı: Tanenbaum'a göre 16 GB bellek, 8 ns'de 8 bayt kopyalayarak ≈ **16 s**; bugünkü ≈20 GB/s kopyalama hızıyla bile ≈ 0,9 s ve bu sırada süreçler durur.
- Veri sıkıştırma (*compression*) ile karıştırılmamalı: amaç veriyi küçültmek değil, blokları *bitiştirmektir*.
- Bugün: Linux, büyük sayfa için bitişik fiziksel çerçeve üretmek amacıyla **sayfa göçü** yapar (`kcompactd`).

Boş Bellek Takibi: Biteşlem ve Bağlı Liste



	Biteşlem	**Bağlı liste**
Yapı	Her tahsis birimine 1 bit (0 boş, 1 dolu)	Her süreç/delik için düğüm (S/D, başlangıç, uzunluk)
Boyut	Sabit: birim 4 B ise belleğin 1/33'ü; 16 GiB, 4 KiB birim → 512 KiB	Süreç + delik sayısıyla orantılı
Arama	k ardışık 0 bulmak **yavaş**	Delik listesi gezilir; adrese göre sıralı tutulursa birleştirme kolay

Biteşlemde Boş Alan Arama: Kod İzleme

```
1  /* k ardışık boş (0) birim ara */
2  int bul(const char bit[], int n, int k) {
▶    int bas = 0, uz = 0;
4    for (int i = 0; i < n; i++) {
5      if (bit[i] == 0) {
6        if (uz == 0) bas = i;  /* yeni koşu */
7        if (++uz == k) return bas;
8      } else
9        uz = 0;                /* koşu bozuldu */
10   }
11   return -1;
12 }
```



turuncu: süren boş koşu · yeşil: bulunan alan

Bellek 10 birim; dolu birimler 1. **k = 3** birimlik bir süreç için ardışık üç 0 aranıyor.

Adım 1 / 10

DEĞİŞKENLER

`k = 3` `i = –` `bas = 0` `uz = 0`

ÇIKTI

Biteşlemde Boş Alan Arama: Kod İzleme

```
1  /* k ardışık boş (0) birim ara */
2  int bul(const char bit[], int n, int k) {
3    int bas = 0, uz = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (uz == 0) bas = i;  /* yeni koşu */
7        if (++uz == k) return bas;
8      } else
▶        uz = 0;                /* koşu bozuldu */
10   }
11   return -1;
12 }
```



turuncu: süren boş koşu · yeşil: bulunan alan

bit[0] = 1: dolu birim, uz = 0 kalır.

Adım 2 / 10

DEĞİŞKENLER

`k = 3` `i = 0` `bas = 0` `uz = 0`

ÇIKTI

Biteşlemde Boş Alan Arama: Kod İzleme

```
1  /* k ardışık boş (0) birim ara */
2  int bul(const char bit[], int n, int k) {
3    int bas = 0, uz = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (uz == 0) bas = i;  /* yeni koşu */
7        if (++uz == k) return bas;
8      } else
▶        uz = 0;                /* koşu bozuldu */
10   }
11   return -1;
12 }
```



turuncu: süren boş koşu · yeşil: bulunan alan

bit[1] = 1: dolu birim, uz = 0 kalır.

Adım 3 / 10

DEĞİŞKENLER

`k = 3` `i = 1` `bas = 0` `uz = 0`

ÇIKTI

Biteşlemde Boş Alan Arama: Kod İzleme

```
1  /* k ardışık boş (0) birim ara */
2  int bul(const char bit[], int n, int k) {
3    int bas = 0, uz = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
▶        if (uz == 0) bas = i;  /* yeni koşu */
▶        if (++uz == k) return bas;
8      } else
9        uz = 0;                /* koşu bozuldu */
10   }
11   return -1;
12 }
```



turuncu: süren boş koşu · yeşil: bulunan alan

bit[2] = 0 ve önceki koşu yok → yeni koşu `bas = 2`, uz = 1.

Adım 4 / 10

DEĞİŞKENLER

`k = 3` `i = 2` `bas = 2` `uz = 1`

ÇIKTI

Biteşlemde Boş Alan Arama: Kod İzleme

```
1  /* k ardışık boş (0) birim ara */
2  int bul(const char bit[], int n, int k) {
3    int bas = 0, uz = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (uz == 0) bas = i;  /* yeni koşu */
▶        if (++uz == k) return bas;
8      } else
9        uz = 0;                /* koşu bozuldu */
10   }
11   return -1;
12 }
```



turuncu: süren boş koşu · yeşil: bulunan alan

bit[3] = 0 → koşu uzar: uz = 2; henüz k'ye ulaşmadı.

Adım 5 / 10

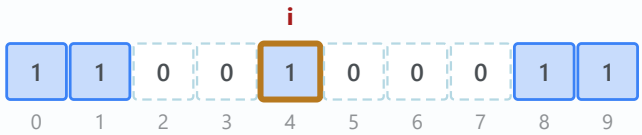
DEĞİŞKENLER

`k = 3` `i = 3` `bas = 2` `uz = 2`

ÇIKTI

Biteşlemde Boş Alan Arama: Kod İzleme

```
1  /* k ardışık boş (0) birim ara */
2  int bul(const char bit[], int n, int k) {
3    int bas = 0, uz = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (uz == 0) bas = i;  /* yeni koşu */
7        if (++uz == k) return bas;
8      } else
▶        uz = 0;                /* koşu bozuldu */
10   }
11   return -1;
12 }
```



turuncu: süren boş koşu · yeşil: bulunan alan

bit[4] = 1: 2 birimlik koşu yetmeden bozuldu → **uz = 0**. 2–3 arasındaki delik bu süreç için küçük.

Adım 6 / 10

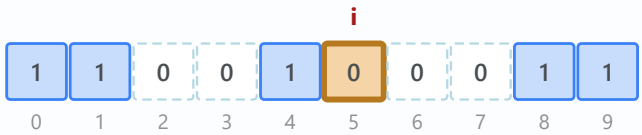
DEĞİŞKENLER

`k = 3` `i = 4` `bas = 2` `uz = 0`

ÇIKTI

Biteşlemde Boş Alan Arama: Kod İzleme

```
1  /* k ardışık boş (0) birim ara */
2  int bul(const char bit[], int n, int k) {
3    int bas = 0, uz = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
▶        if (uz == 0) bas = i;  /* yeni koşu */
▶        if (++uz == k) return bas;
8      } else
9        uz = 0;                /* koşu bozuldu */
10   }
11   return -1;
12 }
```



turuncu: süren boş koşu · yeşil: bulunan alan

bit[5] = 0 ve önceki koşu yok → yeni koşu `bas = 5`, uz = 1.

Adım 7 / 10

DEĞİŞKENLER

`k = 3` `i = 5` `bas = 5` `uz = 1`

ÇIKTI

Biteşlemde Boş Alan Arama: Kod İzleme

```
1  /* k ardışık boş (0) birim ara */
2  int bul(const char bit[], int n, int k) {
3    int bas = 0, uz = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (uz == 0) bas = i;  /* yeni koşu */
▶        if (++uz == k) return bas;
8      } else
9        uz = 0;                /* koşu bozuldu */
10   }
11   return -1;
12 }
```



turuncu: süren boş koşu · yeşil: bulunan alan

bit[6] = 0 → koşu uzar: uz = 2; henüz k'ye ulaşmadı.

Adım 8 / 10

DEĞİŞKENLER

`k = 3` `i = 6` `bas = 5` `uz = 2`

ÇIKTI

Biteşlemde Boş Alan Arama: Kod İzleme

```
1  /* k ardışık boş (0) birim ara */
2  int bul(const char bit[], int n, int k) {
3    int bas = 0, uz = 0;
4    for (int i = 0; i < n; i++) {
▶      if (bit[i] == 0) {
6        if (uz == 0) bas = i;  /* yeni koşu */
▶        if (++uz == k) return bas;
8      } else
9        uz = 0;                /* koşu bozuldu */
10   }
11   return -1;
12 }
```



turuncu: süren boş koşu · yeşil: bulunan alan

bit[7] = 0 → uz = **3 = k**: koşu yeterli, başlangıç **5** döndürülür.

Adım 9 / 10

DEĞİŞKENLER

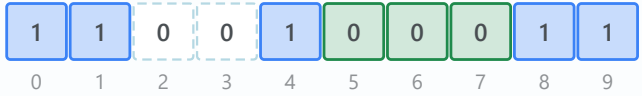
`k = 3` `i = 7` `bas = 5` `uz = 3`

ÇIKTI

```
bul = 5
```

Biteşlemde Boş Alan Arama: Kod İzleme

```
1  /* k ardışık boş (0) birim ara */
2  int bul(const char bit[], int n, int k) {
3    int bas = 0, uz = 0;
4    for (int i = 0; i < n; i++) {
5      if (bit[i] == 0) {
6        if (uz == 0) bas = i;  /* yeni koşu */
▶        if (++uz == k) return bas;
8      } else
9        uz = 0;                /* koşu bozuldu */
10   }
11   return -1;
12 }
```



turuncu: süren boş koşu · yeşil: bulunan alan

Ayırıcı 5, 6, 7. bitleri 1 yapar. Arama dizinin başından doğrusal ilerler; büyük biteşlemde bu yüzden **yavaştır**.

Adım 10 / 10

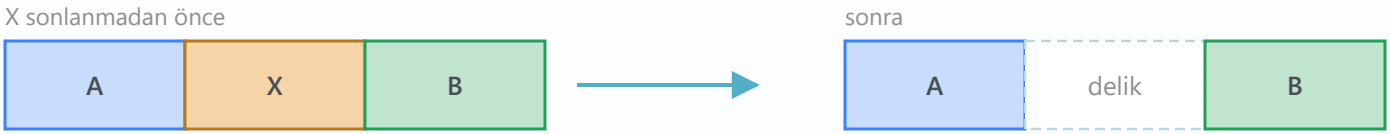
DEĞİŞKENLER

`k = 3` `bas = 5` `uz = 3`

ÇIKTI

```
bul = 5
```

Bağlı Listede Serbest Bırakma



Komşuların ikisi de süreç: X'in düğümü **D**'ye çevrilir, liste uzunluğu değişmez.

Adım 1 / 4

- Liste **adrese göre sıralı** tutulursa birleştirme komşulara bakarak sabit zamanda yapılır.
- Delikleri ayrı bir listede, **boyuta göre** sıralı tutmak en iyi uyanı hızlandırır (ilk uyan = en iyi uyan olur) ama serbest bırakmayı yavaşlatır.
- Delik listesi deliklerin *kendi içinde* saklanabilir: her deliğin ilk sözcükleri boyut ve sonraki delik işaretçisidir.

Bağlı Listede Serbest Bırakma



Sağda delik: X ile sağdaki delik **birleşir**, bir düğüm silinir.

- Liste **adrese göre sıralı** tutulursa birleştirme komşulara bakarak sabit zamanda yapılır.
- Delikleri ayrı bir listede, **boyuta göre** sıralı tutmak en iyi uyanı hızlandırır (ilk uyan = en iyi uyan olur) ama serbest bırakmayı yavaşlatır.
- Delik listesi deliklerin *kendi içinde* saklanabilir: her deliğin ilk sözcükleri boyut ve sonraki delik işaretçisidir.

Bağlı Listede Serbest Bırakma



Solda delik: soldaki delik X kadar uzar, X'in düğümü silinir.

- Liste **adrese göre sıralı** tutulursa birleştirme komşulara bakarak sabit zamanda yapılır.
- Delikleri ayrı bir listede, **boyuta göre** sıralı tutmak en iyi uyanı hızlandırır (ilk uyan = en iyi uyan olur) ama serbest bırakmayı yavaşlatır.
- Delik listesi deliklerin *kendi içinde* saklanabilir: her deliğin ilk sözcükleri boyut ve sonraki delik işaretçisidir.

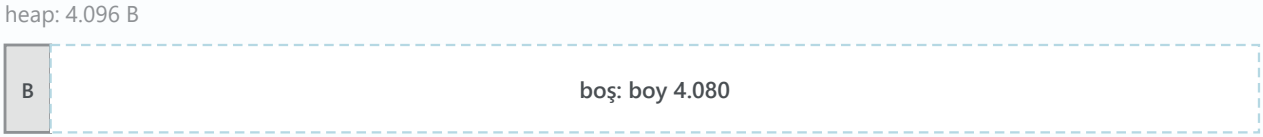
Bağlı Listede Serbest Bırakma



İki yanda delik: üç düğüm **tek deliğe** iner, iki düğüm silinir. Önceki düğümü bulmak için **çift bağlı liste** kullanmak kolaylık sağlar.

- Liste **adrese göre sıralı** tutulursa birleştirme komşulara bakarak sabit zamanda yapılır.
- Delikleri ayrı bir listede, **boyuta göre** sıralı tutmak en iyi uyanı hızlandırır (ilk uyan = en iyi uyan olur) ama serbest bırakmayı yavaşlatır.
- Delik listesi deliklerin *kendi içinde* saklanabilir: her deliğin ilk sözcükleri boyut ve sonraki delik işaretçisidir.

malloc İçinde: Başlık ve Gömülü Boş Liste



Başlangıç: tek boş parça. Boş listenin düğümü parçanın **kendi içinde** durur; ayrı bir veri yapısı için `malloc` çağrılamaz.

Sihirli sayı

Başlıktaki sabit desen bozulmuşsa (taşma, çift `free`, yanlış işaretçi) kütüphane bunu fark edip programı durdurabilir.

Gerçek istek

`malloc(n)` en az n + başlık bayt tüketir; hizalama için yukarı yuvarlanır (glibc: 16 B).

Sistem çağrısı mı?

Hayır, kütüphane işlevidir. Yer bitince `brk` / `sbrk` ile heap'i büyütür; büyük istekleri (glibc'de ≥ 128 KiB) doğrudan `mmap` ile alır.

malloc İçinde: Başlık ve Gömülü Boş Liste



B = 16 B başlık: kullanımdaysa {boy, sihirli sayı}, boşsa {boy, sonraki}

boş liste: **head → [3.864] → NULL**

malloc(200) 216 B kullanır: 4.080 − 216 = 3.864; p1 başlığın hemen arkasını gösterir

`p1 = malloc(200)` : ilk uyan parça **bölünür**. Başlığa boy ve sihirli sayı yazılır.

Sihirli sayı

Başlıktaki sabit desen bozulmuşsa (taşma, çift `free`, yanlış işaretçi) kütüphane bunu fark edip programı durdurabilir.

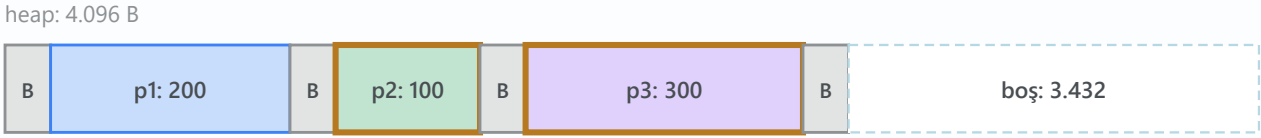
Gerçek istek

`malloc(n)` en az n + başlık bayt tüketir; hizalama için yukarı yuvarlanır (glibc: 16 B).

Sistem çağrısı mı?

Hayır, kütüphane işlevidir. Yer bitince `brk` / `sbrk` ile heap'i büyütür; büyük istekleri (glibc'de ≥ 128 KiB) doğrudan `mmap` ile alır.

malloc İçinde: Başlık ve Gömülü Boş Liste



B = 16 B başlık: kullanımdaysa {boy, sihirli sayı}, boşsa {boy, sonraki}

boş liste: **head → [3.432] → NULL**

3.864 − (100 + 16) − (300 + 16) = 3.432

`p2 = malloc(100)`, `p3 = malloc(300)`: aynı bölme tekrarlanır.

Adım 3 / 5

Sihirli sayı

Başlıktaki sabit desen bozulmuşsa (taşma, çift `free`, yanlış işaretçi) kütüphane bunu fark edip programı durdurabilir.

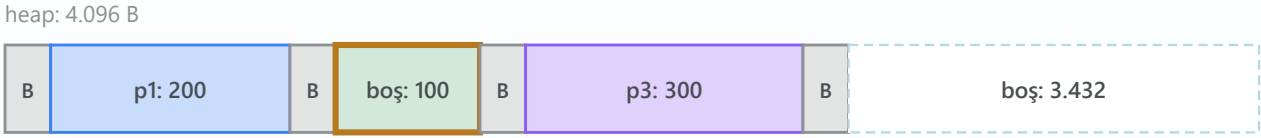
Gerçek istek

`malloc(n)` en az n + başlık bayt tüketir; hizalama için yukarı yuvarlanır (glibc: 16 B).

Sistem çağrısı mı?

Hayır, kütüphane işlevidir. Yer bitince `brk` / `sbrk` ile heap'i büyütür; büyük istekleri (glibc'de ≥ 128 KiB) doğrudan `mmap` ile alır.

malloc İçinde: Başlık ve Gömülü Boş Liste



`free(p2)` boyut almaz: boy başlıktan okunur. Artık 150 B'lık istek 100'lük parçaya sığmaz: **dış parçalanma**.

Sihirli sayı

Başlıktaki sabit desen bozulmuşsa (taşma, çift `free`, yanlış işaretçi) kütüphane bunu fark edip programı durdurabilir.

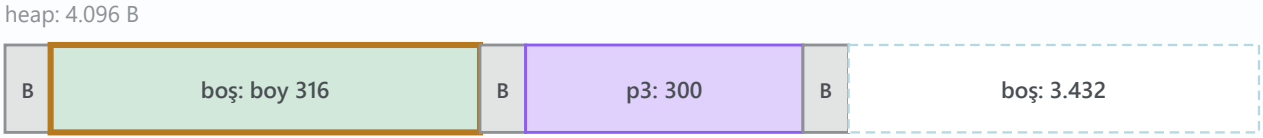
Gerçek istek

`malloc(n)` en az n + başlık bayt tüketir; hizalama için yukarı yuvarlanır (glibc: 16 B).

Sistem çağrısı mı?

Hayır, kütüphane işlevidir. Yer bitince `brk` / `sbrk` ile heap'i büyütür; büyük istekleri (glibc'de ≥ 128 KiB) doğrudan `mmap` ile alır.

malloc İçinde: Başlık ve Gömülü Boş Liste



`free(p1)` : adrese göre komşu iki boş parça **birleştirilir**; liste kısalır.

Sihirli sayı

Başlıktaki sabit desen bozulmuşsa (taşma, çift `free`, yanlış işaretçi) kütüphane bunu fark edip programı durdurabilir.

Gerçek istek

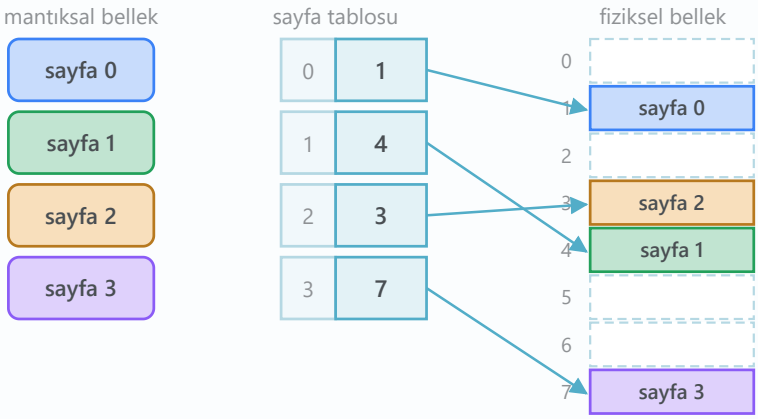
`malloc(n)` en az n + başlık bayt tüketir; hizalama için yukarı yuvarlanır (glibc: 16 B).

Sistem çağrısı mı?

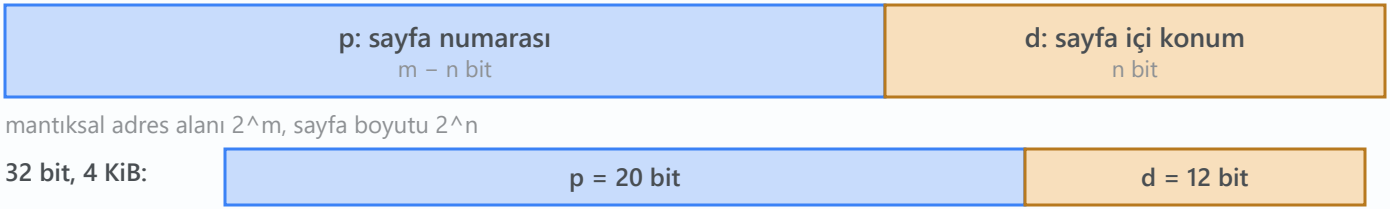
Hayır, kütüphane işlevidir. Yer bitince `brk` / `sbrk` ile heap'i büyütür; büyük istekleri (glibc'de ≥ 128 KiB) doğrudan `mmap` ile alır.

Sayfalama: Temel Fikir

- Fiziksel bellek sabit boyutlu **çerçevelere** (frame), mantıksal bellek aynı boyutta **sayfalara** (page) bölünür.
- Sürecin fiziksel alanı **bitişik olmak zorunda değil** → dış parçalanma yok; yalnızca son sayfada iç parçalanma.
- N sayfalık süreç için N boş çerçeve yeter; işletim sistemi boş çerçeveleri izler.
- **Sayfa tablosu** sayfa numarasını çerçeve numarasına eşler; her sürecin kendi tablosu vardır.
- Yedekleme deposu da aynı boyutlu bloklara bölünür.



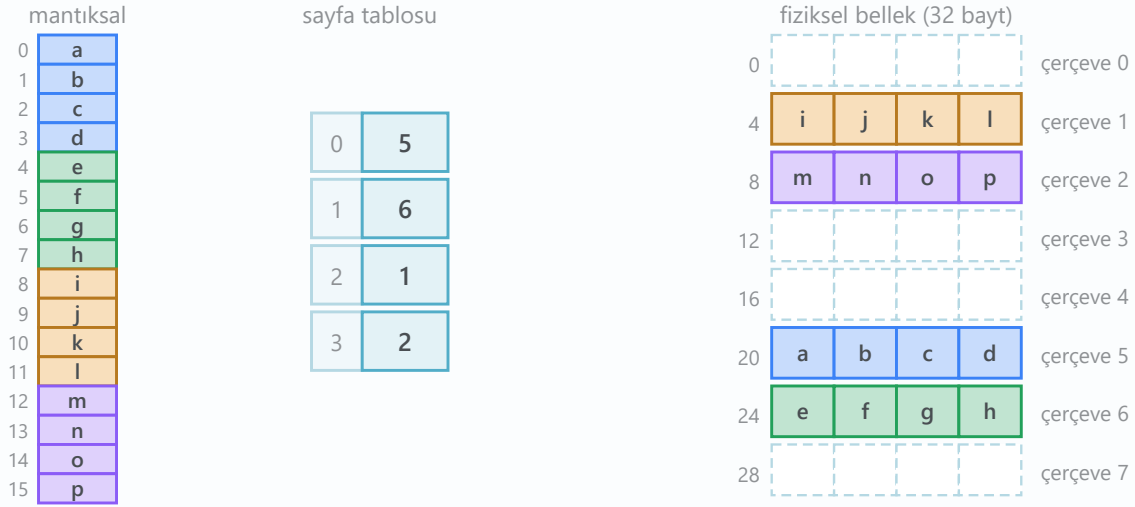
Adresin Bölünmesi: (p, d)



Örnek	Hesap	Sonuç
32 bit adres, 4 KiB sayfa	4096 = 2^{12} → d 12 bit, p = 32 − 12 = 20 bit	2^{20} = 1.048.576 sayfa
`0x12345678`	p = `0x12345678 >> 12`, d = `0x12345678 & 0xFFF`	p = `0x12345` (74.565), d = `0x678` (1656)
16 bit adres, 4 KiB (Tanenbaum)	8196 = 2·4096 + 4	p = 2, d = 4
48 bit (x86-64), 4 KiB	p = 48 − 12	36 bit sayfa numarası

Sayfa boyutu 2'nin kuvveti olduğu için bölme ve kalan birer bit kaydırma ve maskeleme işlemidir; donanım bunu bedavaya yapar. Fiziksel adres = çerçeve × sayfa boyutu + d.

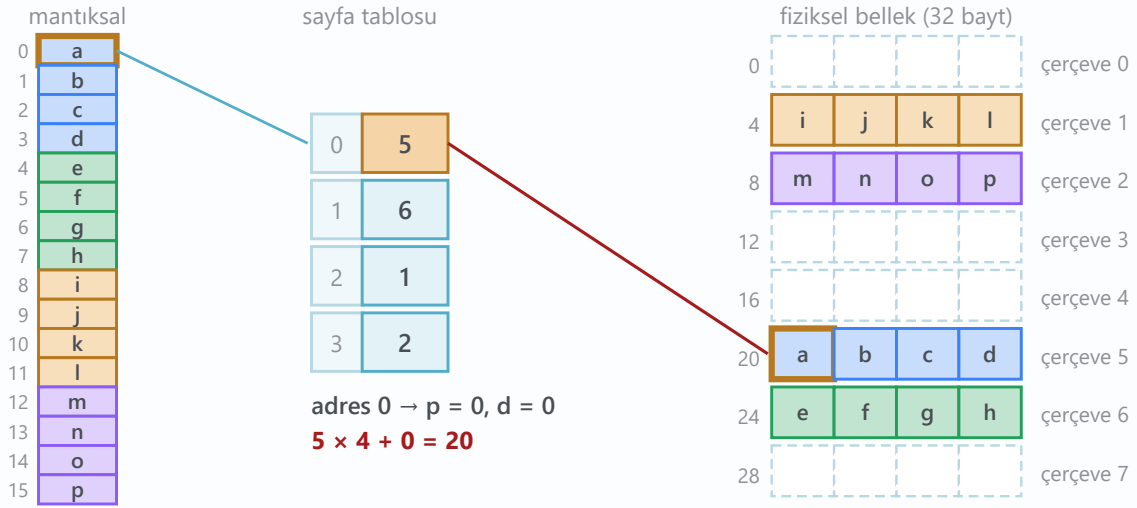
Sayfa Tablosu ile Çeviri



Sayfa boyutu **4 bayt**, fiziksel bellek **32 bayt** (8 çerçeve). 16 baytlık mantıksal bellek a…p; sayfa tablosu 0→5, 1→6, 2→1, 3→2.

Silberschatz Şekil 9.10'daki örnek. Sayfa tablosu girişinin indisi sayfa numarasıdır; sayfa numarası tabloda saklanmaz.

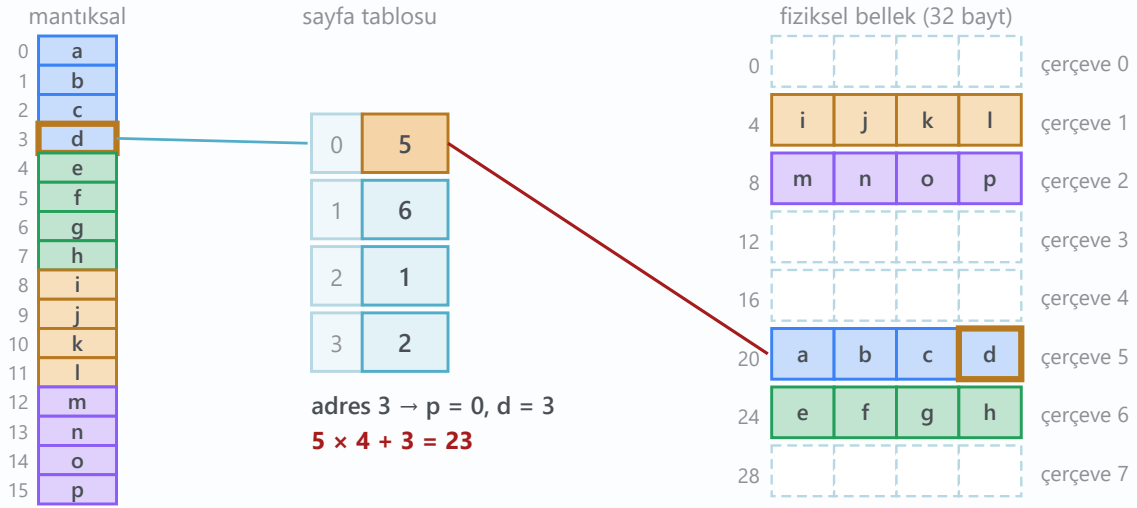
Sayfa Tablosu ile Çeviri



Mantıksal **0** (`a`): sayfa 0, konum 0 → çerçeve 5 → fiziksel **20**.

Silberschatz Şekil 9.10'daki örnek. Sayfa tablosu girişinin indisi sayfa numarasıdır; sayfa numarası tabloda saklanmaz.

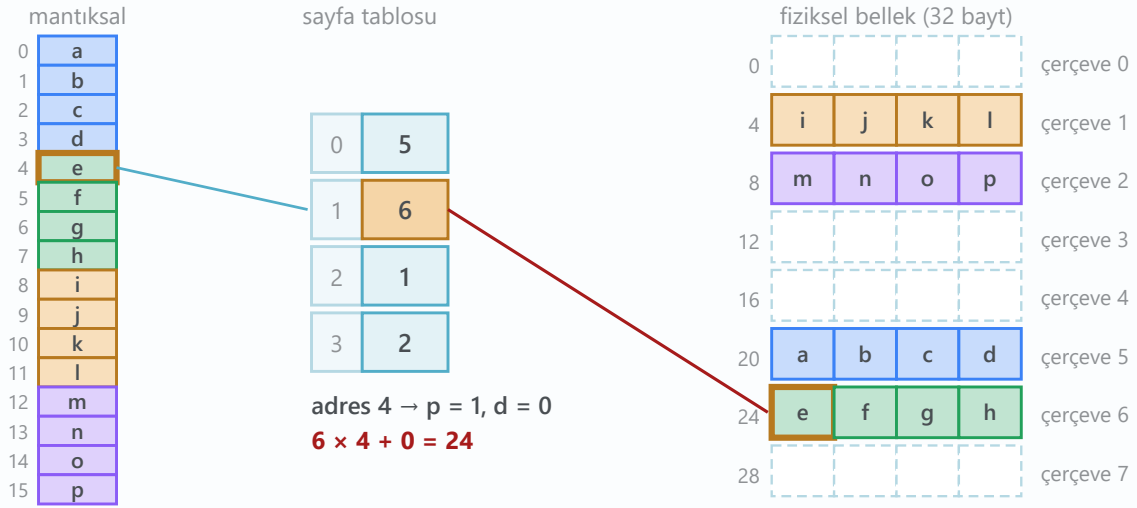
Sayfa Tablosu ile Çeviri



Mantıksal **3** (`d`): sayfa 0, konum 3 → çerçeve 5 → fiziksel **23**.

Silberschatz Şekil 9.10'daki örnek. Sayfa tablosu girişinin indisi sayfa numarasıdır; sayfa numarası tabloda saklanmaz.

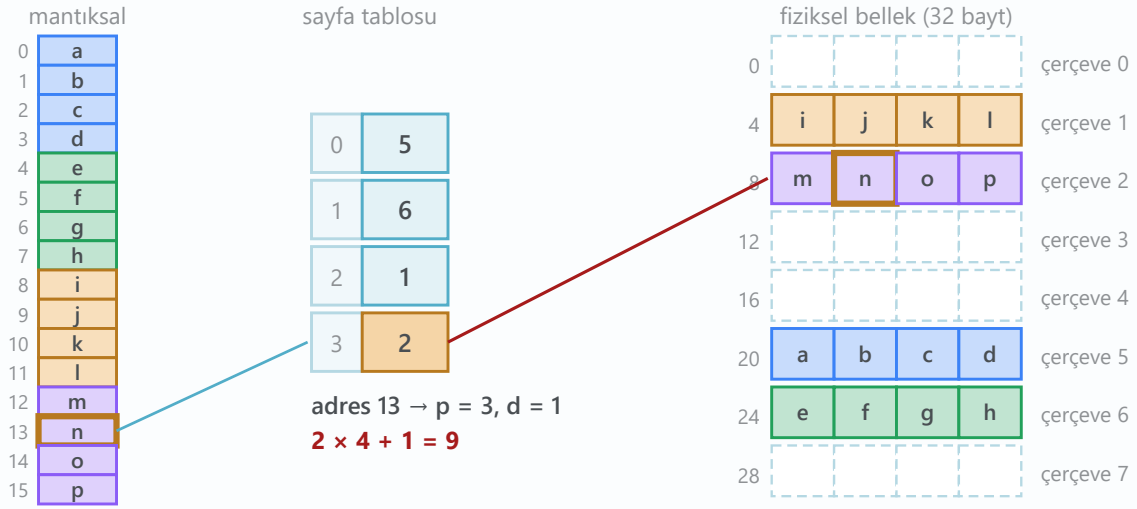
Sayfa Tablosu ile Çeviri



Mantıksal **4** (e): sayfa 1, konum 0 → çerçeve 6 → fiziksel **24**.

Silberschatz Şekil 9.10'daki örnek. Sayfa tablosu girişinin indisi sayfa numarasıdır; sayfa numarası tabloda saklanmaz.

Sayfa Tablosu ile Çeviri

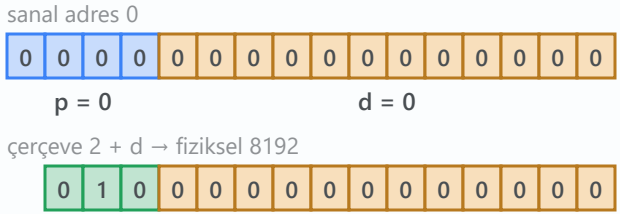


Mantıksal **13** (`n`): sayfa 3, konum 1 → çerçeve 2 → fiziksel **9**. Sürecin sırası fiziksel bellekte korunmaz; yine de program ardışık görür.

Silberschatz Şekil 9.10'daki örnek. Sayfa tablosu girişinin indisi sayfa numarasıdır; sayfa numarası tabloda saklanmaz.

MMU İçinde Çeviri

```
1  /* 16 bit sanal adres, 4 KiB sayfa
   */
▶  MOV REG, 0
3  MOV REG, 8192
4  MOV REG, 20500
5  MOV REG, 8196
6  MOV REG, 32780
7
8  /* sayfa tablosu (sayfa → çerçeve)
   */
9  /* 0→2 1→1 2→6 3→0 4→4 5→3 */
10 /* 9→5 11→7, diğerleri: yok */
```



Sayfa 0 → çerçeve 2: **8192**. Üst 4 bit sayfa numarası, alt 12 bit aynen kopyalanır.

Adım 1 / 5

DEĞİŞKENLER

`sanal = 0` `p = 0` `d = 0` `çerçeve = 2` `fiziksel = 8192`

64 KiB sanal adres alanı (16 sayfa), 32 KiB fiziksel bellek (8 çerçeve → çerçeve numarası 3 bit).

MMU İçinde Çeviri

```
1  /* 16 bit sanal adres, 4 KiB sayfa
   */
2  MOV REG, 0
▶  MOV REG, 8192
4  MOV REG, 20500
5  MOV REG, 8196
6  MOV REG, 32780
7
8  /* sayfa tablosu (sayfa → çerçeve)
   */
9  /* 0→2 1→1 2→6 3→0 4→4 5→3 */
10 /* 9→5 11→7, diğerleri: yok */
```



8192 = sayfa 2, konum 0 → çerçeve 6: **24576**.

Adım 2 / 5

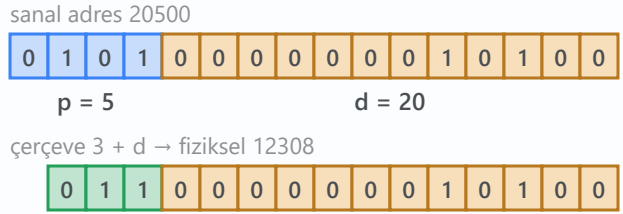
DEĞİŞKENLER

`sanal = 8192` `p = 2` `d = 0` `çerçeve = 6` `fiziksel = 24576`

64 KiB sanal adres alanı (16 sayfa), 32 KiB fiziksel bellek (8 çerçeve → çerçeve numarası 3 bit).

MMU İçinde Çeviri

```
1  /* 16 bit sanal adres, 4 KiB sayfa
   */
2  MOV REG, 0
3  MOV REG, 8192
▶  MOV REG, 20500
5  MOV REG, 8196
6  MOV REG, 32780
7
8  /* sayfa tablosu (sayfa → çerçeve)
   */
9  /* 0→2 1→1 2→6 3→0 4→4 5→3 */
10 /* 9→5 11→7, diğerleri: yok */
```



20500 = 5·4096 + 20 → çerçeve 3: 12288 + 20 = **12308**.

Adım 3 / 5

DEĞİŞKENLER

`sanal = 20500` `p = 5` `d = 20` `çerçeve = 3` `fiziksel = 12308`

64 KiB sanal adres alanı (16 sayfa), 32 KiB fiziksel bellek (8 çerçeve → çerçeve numarası 3 bit).

MMU İçinde Çeviri

```
1  /* 16 bit sanal adres, 4 KiB sayfa
   */
2  MOV REG, 0
3  MOV REG, 8192
4  MOV REG, 20500
▶  MOV REG, 8196
6  MOV REG, 32780
7
8  /* sayfa tablosu (sayfa → çerçeve)
   */
9  /* 0→2 1→1 2→6 3→0 4→4 5→3 */
10 /* 9→5 11→7, diğerleri: yok */
```



8196 = 2·4096 + 4 → 6·4096 + 4 = **24580** (Tanenbaum Şekil 3-10).

Adım 4 / 5

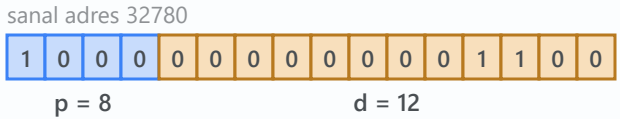
DEĞİŞKENLER

`sanal = 8196` `p = 2` `d = 4` `çerçeve = 6` `fiziksel = 24580`

64 KiB sanal adres alanı (16 sayfa), 32 KiB fiziksel bellek (8 çerçeve → çerçeve numarası 3 bit).

MMU İçinde Çeviri

```
1  /* 16 bit sanal adres, 4 KiB sayfa
   */
2  MOV REG, 0
3  MOV REG, 8192
4  MOV REG, 20500
5  MOV REG, 8196
▶  MOV REG, 32780
7
8  /* sayfa tablosu (sayfa → çerçeve)
   */
9  /* 0→2 1→1 2→6 3→0 4→4 5→3 */
10 /* 9→5 11→7, diğerleri: yok */
```



sayfa 8 yok → sayfa hatası (tuzak)

32780 = 8·4096 + 12; sayfa 8 eşlenmemiş (mevcut/yok biti 0) → **sayfa hatası**; işletim sistemi devreye girer.

Adım 5 / 5

DEĞİŞKENLER

`sanal = 32780` `p = 8` `d = 12` `çerçeve = –` `fiziksel = –`

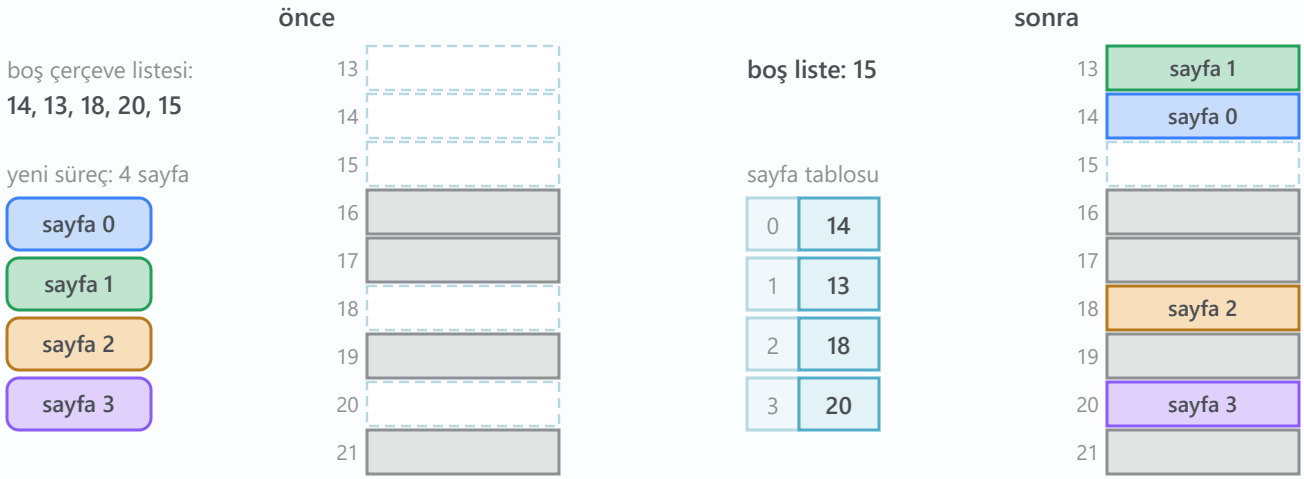
64 KiB sanal adres alanı (16 sayfa), 32 KiB fiziksel bellek (8 çerçeve → çerçeve numarası 3 bit).

Sayfa Boyutu ve İç Parçalanma

- Örnek: sayfa 2.048 B, süreç 72.766 B → 35 sayfa + 1.086 B → **36 çerçeve**, son sayfada 962 B iç parçalanma.
- En kötü durumda iç parçalanma 1 sayfa − 1 bayt, ortalama yarım sayfa.
- Küçük sayfa: az israf ama büyük sayfa tablosu, çok TLB ıskası. Büyük sayfa: verimli disk G/Ç, az tablo girdisi.
- Ek yük = $s \cdot e / p + p / 2$ (s süreç boyutu, e girdi boyutu). Türevi sıfır: p = √(2se); s = 1 MiB, e = 8 B → **p = 4096**.
- Bugün: x86-64 4 KiB (+ 2 MiB, 1 GiB büyük sayfa), ARM64 4/16/64 KiB, Apple Silicon 16 KiB.



Boş Çerçeveler



- İşletim sistemi bir **çerçeve tablosu** tutar: her fiziksel çerçeve için boş mu, hangi sürecin hangi sayfası.
- Kullanıcı süreci adres verdiğinde (ör. `read` tamponu) çekirdek bunu sürecin sayfa tablosuyla fiziksel adrese çevirir.
- Bağlam değiştirmede dağıtıcı yeni sürecin sayfa tablosunu etkinleştirir (x86'da `CR3`): **bağlam değiştirme süresi uzar**.

SAYFALAMA

Sayfa Tablosu Girdisi

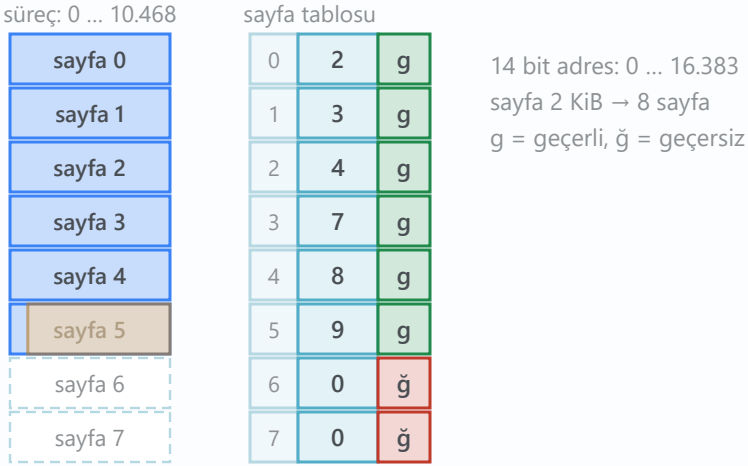
x86-64 sayfa tablosu girdisi (64 bit, 4 KiB sayfa)



Bit	Anlamı (Tanenbaum adı)	Kim yazar?
P	Mevcut/yok (present/absent): 0 ise erişim sayfa hatası üretir	İS
R/W, U/S, NX	Koruma: yazılabilir mi, kullanıcı modu erişebilir mi, çalıştırılabilir mi	İS
A	Erişildi (referenced): okuma ya da yazmada 1 olur; sayfa yer değiştirmede kullanılır	donanım (İS sıfırlar)
D	Değiştirildi (modified, kirli): yazmada 1; kirli sayfa çıkarılırken diske yazılır	donanım
PCD, PWT, PAT	Önbellek devre dışı / yazma politikası: bellek eşlemeli G/Ç için	İS
G	Global: bağlam değiştirmede TLB'den silinmez (çekirdek sayfaları)	İS

Disk adresi PTE'de tutulmaz: sayfa bellekte değilse girdinin geri kalan bitleri işletim sistemine aittir (Linux takas girdisini buraya kodlar).

Geçerli / Geçersiz Biti



Süreç **0–10.468** arasını kullanır: sayfa 0–5 geçerli (çerçeve 2, 3, 4, 7, 8, 9), sayfa 6 ve 7 **geçersiz**.

Adım 1 / 5

Tablonun tamamını tutmak yerine **sayfa tablosu uzunluk yazmacı** (PTLR) da kullanılabilir: uzunluğu aşan sayfa numarası tuzak üretir (Silberschatz Şekil 9.13).

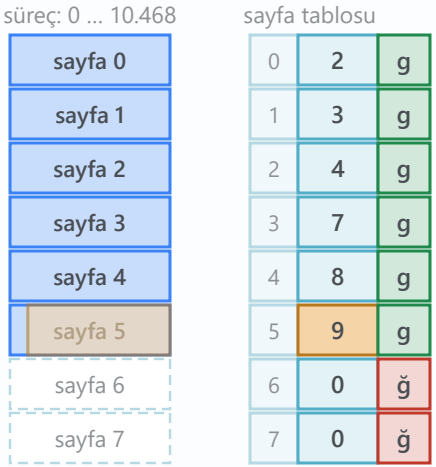
Geçerli / Geçersiz Biti



3.000: sayfa 1, konum 952 → çerçeve 3 → **7.096**.

Tablonun tamamını tutmak yerine **sayfa tablosu uzunluk yazmacı** (PTLR) da kullanılabilir: uzunluğu aşan sayfa numarası tuzak üretir (Silberschatz Şekil 9.13).

Geçerli / Geçersiz Biti



14 bit adres: 0 … 16.383
sayfa 2 KiB → 8 sayfa
g = geçerli, ğ = geçersiz

adres 10.468

= 5 × 2048 + 228 → sayfa 5

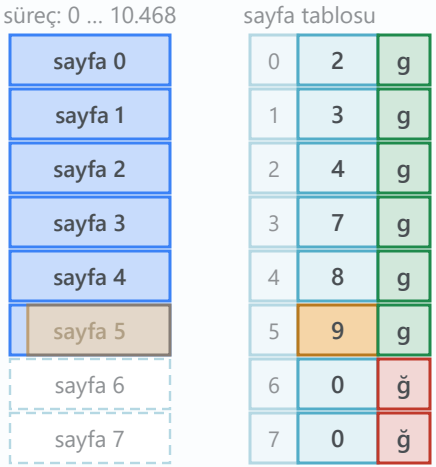
geçerli → çerçeve 9

fiziksel = 9 × 2048 + 228 = 18.660

10.468: sayfa 5'in içinde → **18.660**. Sürecin son adresi.

Tablonun tamamını tutmak yerine **sayfa tablosu uzunluk yazmacı** (PTLR) da kullanılabilir: uzunluğu aşan sayfa numarası tuzak üretir (Silberschatz Şekil 9.13).

Geçerli / Geçersiz Biti



14 bit adres: 0 … 16.383
sayfa 2 KiB → 8 sayfa
g = geçerli, ğ = geçersiz

adres 12.000

= 5 × 2048 + 1760 → sayfa 5

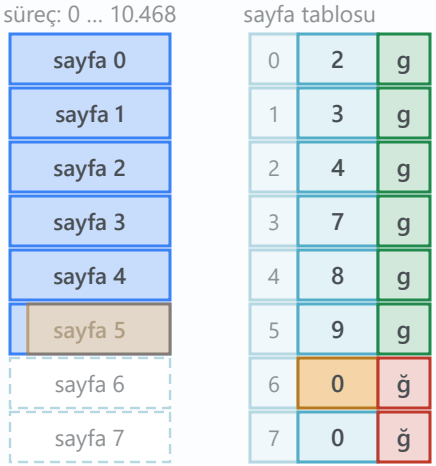
geçerli → çerçeve 9

fiziksel = 9 × 2048 + 1760 = 20.192

12.000 program dışında ama sayfa 5 içinde (10.240–12.287) → donanım **yakalamaz**: 20.192. Sayfa ayrıntı düzeyinin bedeli, iç parçalanma.

Tablonun tamamını tutmak yerine **sayfa tablosu uzunluk yazmacı** (PTLR) da kullanılabilir: uzunluğu aşan sayfa numarası tuzak üretir (Silberschatz Şekil 9.13).

Geçerli / Geçersiz Biti



14 bit adres: 0 … 16.383
sayfa 2 KiB → 8 sayfa
g = geçerli, ğ = geçersiz

adres 12.500

= 6 × 2048 + 212 → sayfa 6

geçersiz bit → tuzak (koruma hatası)

12.500: sayfa 6, geçersiz → **tuzak**. İşletim sistemi bunu geçersiz erişim (SIGSEGV) mi yoksa henüz yüklenmemiş sayfa mı diye ayırır.

Tablonun tamamını tutmak yerine **sayfa tablosu uzunluk yazmacı** (PTLR) da kullanılabilir: uzunluğu aşan sayfa numarası tuzak üretir (Silberschatz Şekil 9.13).

SAYFALAMA

Koruma Bitleri

Okuma / yazma (R/W)
Kod sayfaları salt okunur eşlenir; koda yazma girişimi koruma hatası üretir.

Kullanıcı / denetçi (U/S)
Çekirdek sayfaları kullanıcı modunda erişilemez. Aynı tabloda hem çekirdek hem süreç eşlemeleri durabilir.

Çalıştırma yasağı (NX / XD)
Veri, yığın ve heap sayfalarında kod çalıştırılamaz. İşletim sistemleri **W^X** uygular: bir sayfa ya yazılabilir ya çalıştırılabilir.

SMEP / SMAP
Çekirdek modundayken kullanıcı sayfalarındaki kodu çalıştırmayı (SMEP) ve izinsiz veriye erişmeyi (SMAP) engeller.

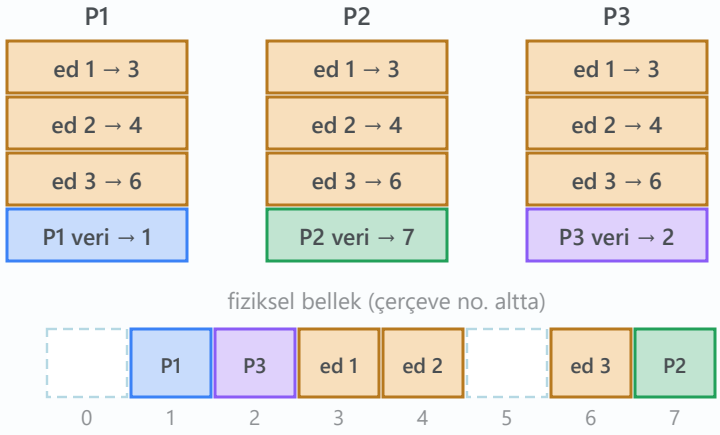
Koruma anahtarları (PKU)
PTE'deki 4 bitlik anahtar ve kullanıcı yazmacı (PKRU) ile sistem çağrısı olmadan sayfa grubunun iznini değiştirmek.

Geçerli / geçersiz
Erişim izni en kaba düzeyde: sayfa adres alanında hiç yoksa her erişim tuzaktır.

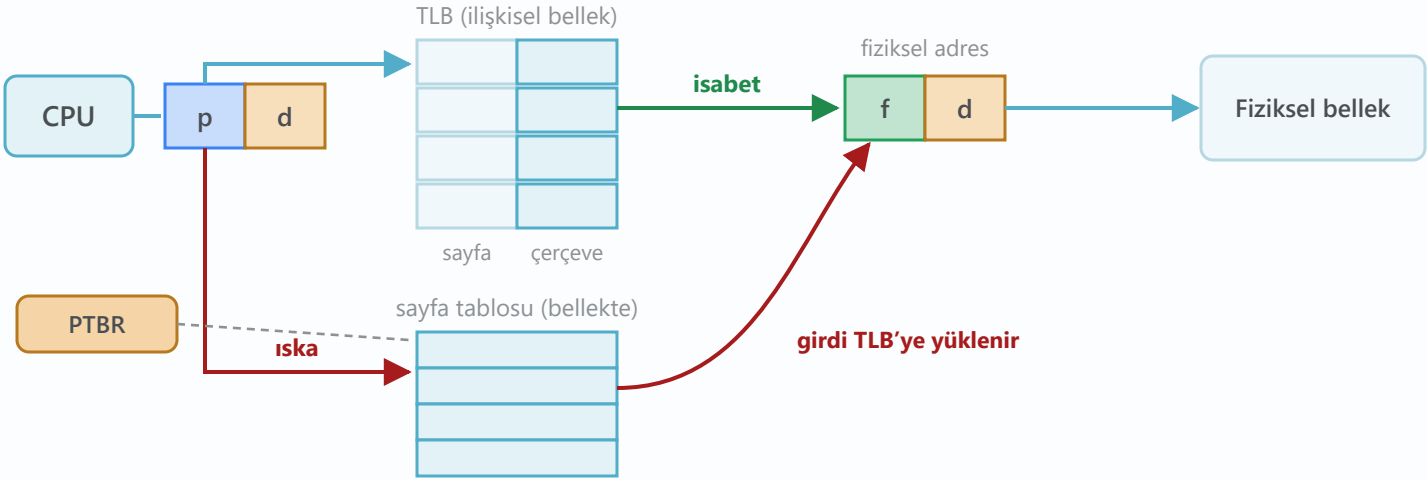
Koruma sayfa ayrıntı düzeyindedir: izin her PTE'de ayrı tutulur ve her erişimde TLB girdisiyle birlikte denetlenir.

Paylaşımlı Sayfalar

- **Yeniden girilebilir (reentrant)**, kendini değiştirmeyen kod paylaşılabilir: üç kullanıcı aynı editörü çalıştırırsa kod çerçeveleri bir kez bellekte olur; yalnızca veri sayfaları ayrıdır.
- Paylaşımlı kütüphaneler (libc) bu şekilde paylaşılır. Sayfa sahibi süreç bitince çerçeve hemen boşaltılamaz: **referans sayısı** tutulur.
- `fork()` : ebeveyn ve çocuk tüm sayfaları **yazınca kopyala** (COW) ile paylaşır; sayfalar salt okunur işaretlenir, ilk yazmada kopya çıkarılır.
- **Bellek eşlemeli dosya** (`mmap`) ve paylaşımlı bellek (`shm_open`): iki süreç aynı çerçeveleri görür, kopyalama olmadan iletişim kurar.



Sayfa Tablosu Nerede? TLB



- Tablo yazmaçlarda tutulamayacak kadar büyüktür; bellekte durur, başlangıcını **PTBR** (x86: `CR3`, ARM: `TTBR0/1`) gösterir. Ama her erişim **iki bellek erişimi** olur.
- **TLB** (translation look-aside buffer): MMU içindeki küçük, hızlı, ilişkisel (paralel aranan) önbellek; tipik 64 girdi L1 + 1–3 bin girdi L2.
- Girdiler **ASID**/PCID ile etiketlenirse bağlam değiştirmede TLB boşaltılmak zorunda kalmaz.

TLB · ADIM ADIM

TLB İsabet ve Iska

erişim dizisi (16 bit, 4 KiB sayfa)

0x0010	0x2004	0x0020	0x5010	0x9000	0x2008	0xB004	0x0030
p = 0	p = 2	p = 0	p = 5	p = 9	p = 2	p = 11	p = 0

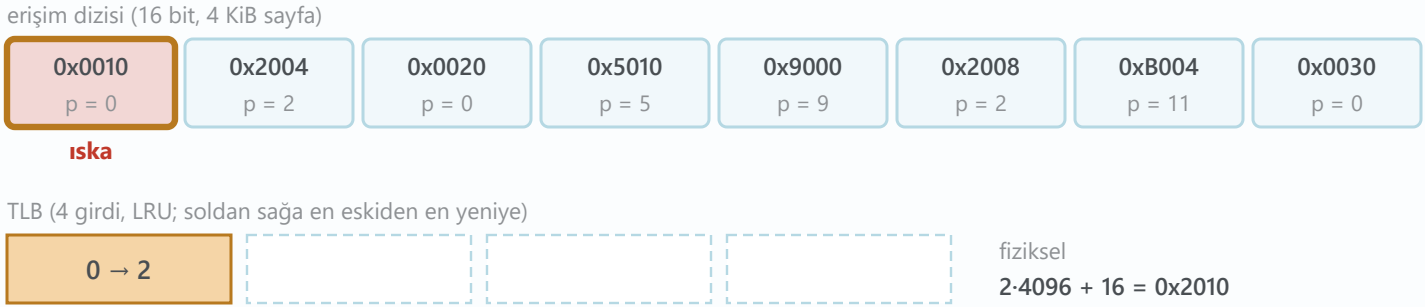
TLB (4 girdi, LRU; soldan sağa en eskiden en yeniye)

Sayfa tablosu MMU örneğindeki Tanenbaum tablosu (0→2, 2→6, 5→3, 9→5, 11→7). TLB başlangıçta boş.

Adım 1 / 9

Etkileşimli simülatör: Sayfalama ve Adres Çevirisi Simülatörü: farklı adreslerle TLB isabet/ıska ve sayfa hatası.

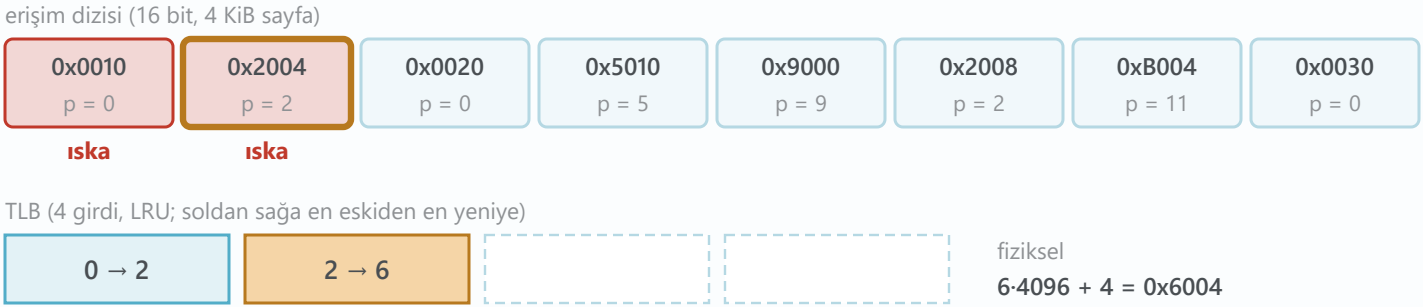
TLB İsabet ve Iska



0x0010: sayfa 0 yok → **ıska**; sayfa tablosundan okunur, TLB'ye yazılır.

Etkileşimli simülatör: Sayfalama ve Adres Çevirisi Simülatörü: farklı adreslerle TLB isabet/ıska ve sayfa hatası.

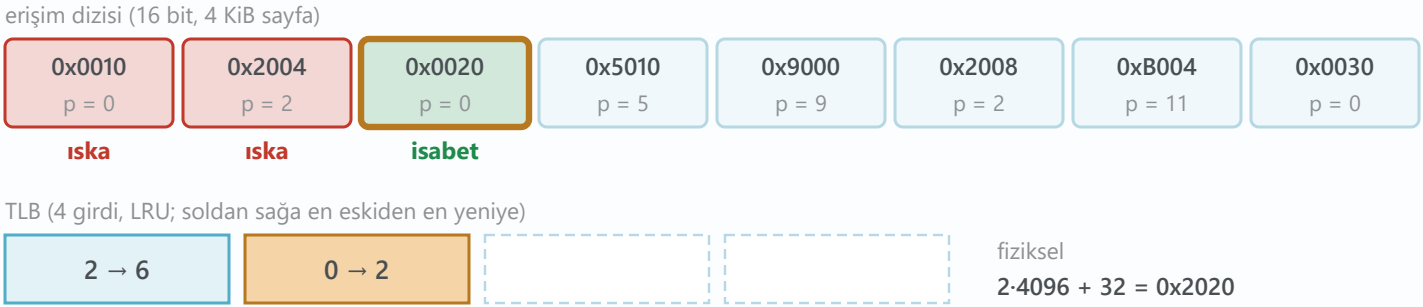
TLB İsabet ve Iska



0x2004: sayfa 2 yok → **ıska**; sayfa tablosundan okunur, TLB'ye yazılır.

Etkileşimli simülatör: Sayfalama ve Adres Çevirisi Simülatörü: farklı adreslerle TLB isabet/ıska ve sayfa hatası.

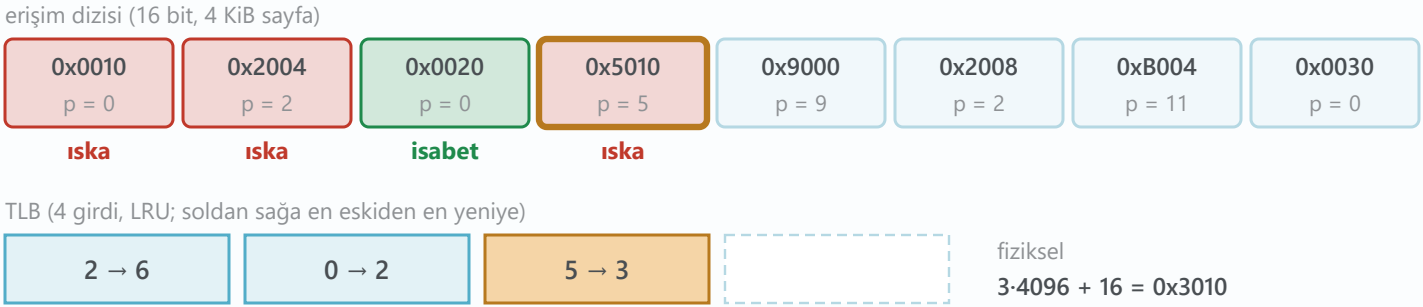
TLB İsabet ve Iska



0x0020: sayfa 0 TLB'de → **isabet**, sayfa tablosuna gidilmez.

Etkileşimli simülatör: Sayfalama ve Adres Çevirisi Simülatörü: farklı adreslerle TLB isabet/ıska ve sayfa hatası.

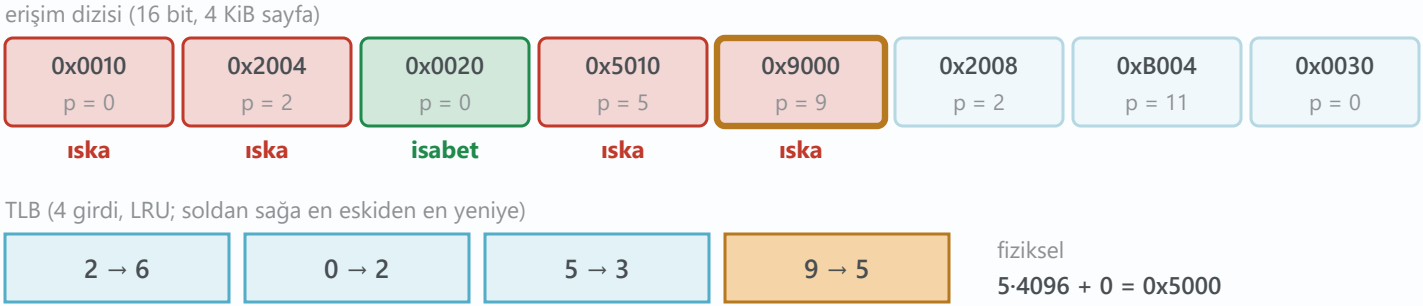
TLB İsabet ve Iska



0x5010: sayfa 5 yok → **ıska**; sayfa tablosundan okunur, TLB'ye yazılır.

Etkileşimli simülatör: Sayfalama ve Adres Çevirisi Simülatörü: farklı adreslerle TLB isabet/ıska ve sayfa hatası.

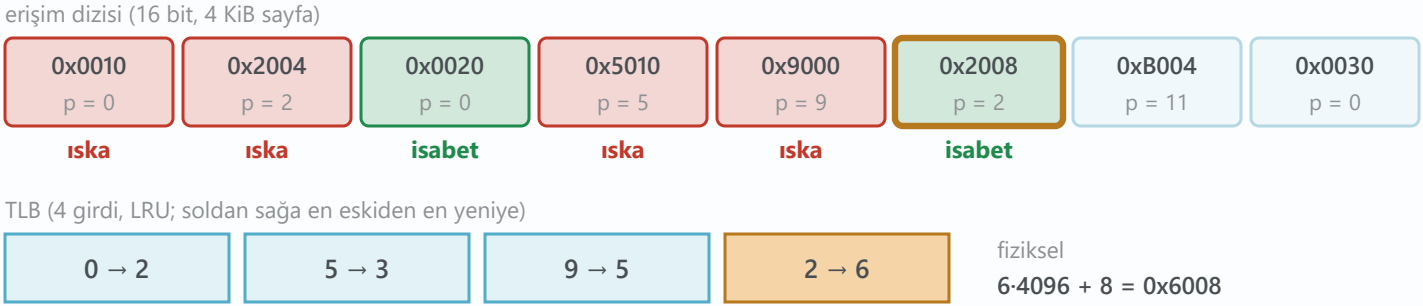
TLB İsabet ve Iska



0x9000: sayfa 9 yok → **ıska**; sayfa tablosundan okunur, TLB'ye yazılır.

Etkileşimli simülatör: Sayfalama ve Adres Çevirisi Simülatörü: farklı adreslerle TLB isabet/ıska ve sayfa hatası.

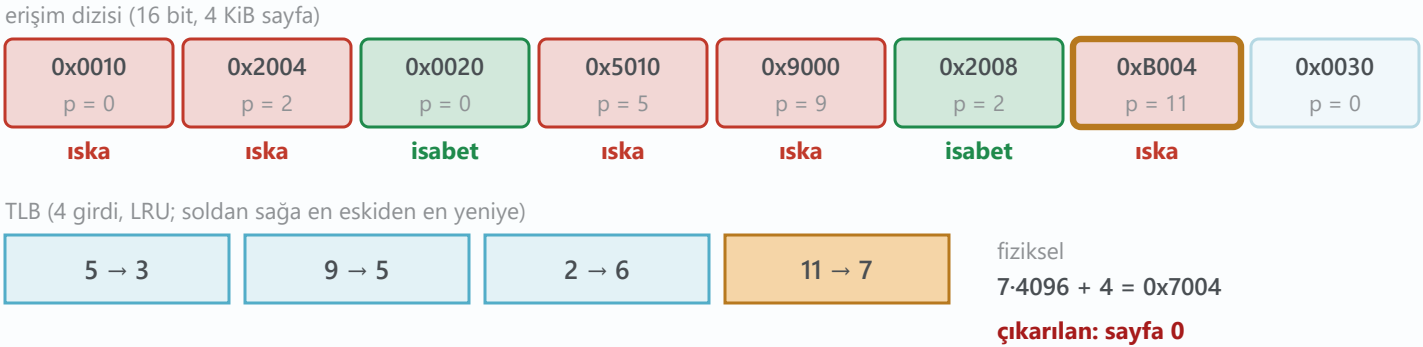
TLB İsabet ve Iska



0x2008: sayfa 2 TLB'de → **isabet**, sayfa tablosuna gidilmez.

Etkileşimli simülatör: Sayfalama ve Adres Çevirisi Simülatörü: farklı adreslerle TLB isabet/ıska ve sayfa hatası.

TLB İsabet ve Iska



0xB004: sayfa 11 yok → **ıska**; sayfa tablosundan okunur, TLB'ye yazılır; dolu olduğu için en uzun süredir kullanılmayan sayfa 0 çıkarılır.

Etkileşimli simülatör: Sayfalama ve Adres Çevirisi Simülatörü: farklı adreslerle TLB isabet/ıska ve sayfa hatası.

TLB İsabet ve Iska

erişim dizisi (16 bit, 4 KiB sayfa)

0x0010	0x2004	0x0020	0x5010	0x9000	0x2008	0xB004	0x0030
p = 0	p = 2	p = 0	p = 5	p = 9	p = 2	p = 11	p = 0
ıska	ıska	isabet	ıska	ıska	isabet	ıska	ıska

TLB (4 girdi, LRU; soldan sağa en eskiden en yeniye)

9 → 5	2 → 6	11 → 7	0 → 2

fiziksel
2·4096 + 48 = 0x2030
çıkarılan: sayfa 5

0x0030: sayfa 0 yok → **ıska**; sayfa tablosundan okunur, TLB'ye yazılır; dolu olduğu için en uzun süredir kullanılmayan sayfa 5 çıkarılır. Toplam: 2 isabet / 8 erişim. Gerçek programlarda yerellik sayesinde oran %99'u aşar.

Etkileşimli simülatör: Sayfalama ve Adres Çevirisi Simülatörü: farklı adreslerle TLB isabet/ıska ve sayfa hatası.

TLB · ADIM ADIM HESAP

Etkin Erişim Süresi

TLB arama süresi **ε = 20 ns**, bellek erişimi **t = 100 ns**, isabet oranı **α**.
EES = (t + ε)·α + (2t + ε)·(1 − α) = 2t + ε − t·α

Formül: isabette bir bellek erişimi, ıskada iki (tablo + veri).

Etkin Erişim Süresi

TLB arama süresi **ε = 20 ns**, bellek erişimi **t = 100 ns**, isabet oranı **α**.

EES = (t + ε)·α + (2t + ε)·(1 − α) = 2t + ε − t·α

Formül: isabette bir bellek erişimi, ıskada iki (tablo + veri).

İsabet: 20 + 100 = **120 ns**; ıska: 20 + 100 + 100 = **220 ns**

İki durumun süresi.

Etkin Erişim Süresi

TLB arama süresi **ε = 20 ns**, bellek erişimi **t = 100 ns**, isabet oranı **α**.
EES = (t + ε)·α + (2t + ε)·(1 − α) = 2t + ε − t·α

Formül: isabette bir bellek erişimi, ıskada iki (tablo + veri).

İsabet: 20 + 100 = **120 ns**; ıska: 20 + 100 + 100 = **220 ns**

İki durumun süresi.

α = %80: 120 × 0,80 + 220 × 0,20 = **140 ns** (%40 yavaşlama)

%80 isabette ortalama 140 ns: TLB'siz 200 ns'ye göre iyi, ama hâlâ bellekten %40 yavaş.

Etkin Erişim Süresi

TLB arama süresi **ε = 20 ns**, bellek erişimi **t = 100 ns**, isabet oranı **α**.
EES = (t + ε)·α + (2t + ε)·(1 − α) = 2t + ε − t·α

Formül: isabette bir bellek erişimi, ıskada iki (tablo + veri).

İsabet: 20 + 100 = **120 ns**; ıska: 20 + 100 + 100 = **220 ns**

İki durumun süresi.

α = %80: 120 × 0,80 + 220 × 0,20 = **140 ns** (%40 yavaşlama)

%80 isabette ortalama 140 ns: TLB'siz 200 ns'ye göre iyi, ama hâlâ bellekten %40 yavaş.

α = %98: 120 × 0,98 + 220 × 0,02 = **122 ns**

%98 isabette 122 ns. Gerçek iş yüklerinde oran genellikle %99'un üzerindedir.

Etkin Erişim Süresi

TLB arama süresi **ε = 20 ns**, bellek erişimi **t = 100 ns**, isabet oranı **α**.
EES = (t + ε)·α + (2t + ε)·(1 − α) = 2t + ε − t·α

Formül: isabette bir bellek erişimi, ıskada iki (tablo + veri).

İsabet: 20 + 100 = **120 ns**; ıska: 20 + 100 + 100 = **220 ns**

İki durumun süresi.

α = %80: 120 × 0,80 + 220 × 0,20 = **140 ns** (%40 yavaşlama)

%80 isabette ortalama 140 ns: TLB'siz 200 ns'ye göre iyi, ama hâlâ bellekten %40 yavaş.

α = %98: 120 × 0,98 + 220 × 0,02 = **122 ns**

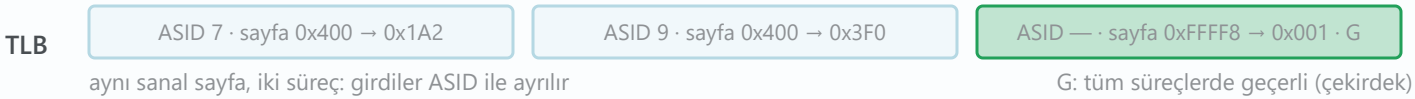
%98 isabette 122 ns. Gerçek iş yüklerinde oran genellikle %99'un üzerindedir.

Dört düzeyli tablo (x86-64): ıska = 20 + 4·100 + 100 = **520 ns**; α = %98 → 128 ns, α = %99 → **124 ns**

Çok düzeyli tabloda ıska çok daha pahalıdır; bu yüzden donanım ara düzeyleri de önbelleğe alır (sayfa yürüme önbelleği) ve TLB isabet oranı daha da önemlidir.

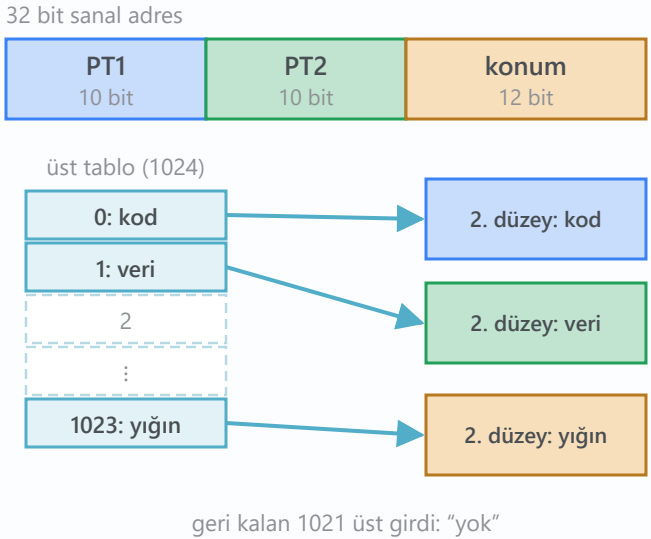
TLB Yönetimi

Konu	Açıklama
Donanım yönetimli	x86-64, ARMv8, RISC-V: ıskada MMU sayfa tablosunu kendisi yürür (page walk); işletim sistemi yalnızca tabloyu kurar.
Yazılım yönetimli	MIPS, SPARC, Alpha gibi eski RISC'lerde ıska işletim sistemine tuzak olur; çekirdek girdiyi bulup TLB'ye yazar. Bu tüm RISC'ler için geçerli değildir: ARMv8 ve RISC-V donanım yönetimlidir.
Yumuşak / sert ıska	Yumuşak ıska: sayfa bellekte, yalnızca TLB'de yok (ns). Sert ıska: sayfa diskte, G/Ç gerekir (µs–ms).
ASID / PCID	Girdi süreç kimliğiyle etiketlenir; bağlam değiştirmede TLB temizlenmez. Meltdown (2018) sonrası KPTI'nin maliyetini PCID azaltır.
Global girdiler	Çekirdek eşlemeleri G biti ile işaretlenir, tüm süreçlerde geçerli kalır.
TLB vurusu (shootdown)	Çok çekirdekte bir eşleme değişince diğer çekirdeklerin TLB'leri kesmeler (IPI) ile geçersizlenir; yeni işlemcilerde yayın komutları var (AMD `INVLPGB`).

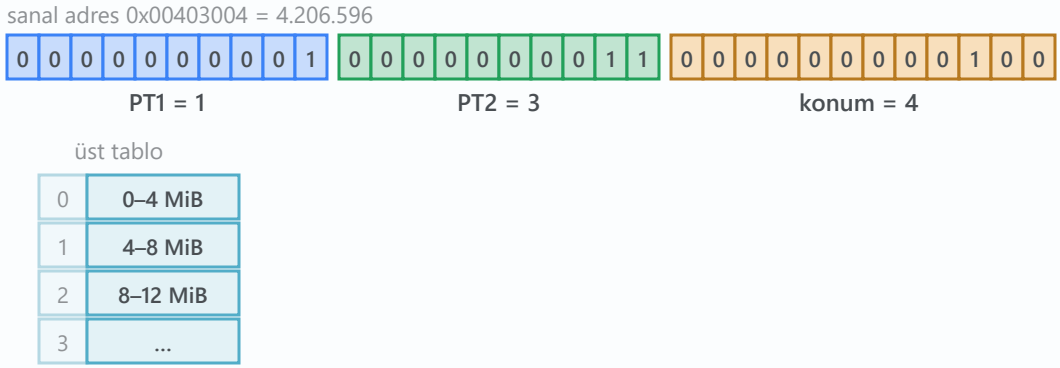


Hiyerarşik (İki Düzeyli) Sayfa Tablosu

- Tek düzeyli tablo, 32 bit ve 4 KiB sayfada 2^{20} girdi × 4 B = **4 MiB**; *her süreç* için ve bitişik olmalı.
- 64 bitte tek düzey: 2^{52} × 8 B = **32 PiB** (≈ 33,6 milyon GiB). 48 bit kullanılsa bile 2^{36} × 8 B = 512 GiB.
- Çözüm: **tabloyu sayfalamak**. 32 bit x86: 10 bit üst tablo (sayfa dizini), 10 bit ikinci düzey, 12 bit konum. Bölünen şey PTE değil, **sanal adrestir**.
- Yalnız kullanılan bölgelerin ikinci düzey tablosu var: 4 MiB kod + 4 MiB veri + 4 MiB yığın kullanan süreç → 1 + 3 tablo = **16 KiB** (4 MiB yerine).



İki Düzeyli Çeviri Örneği

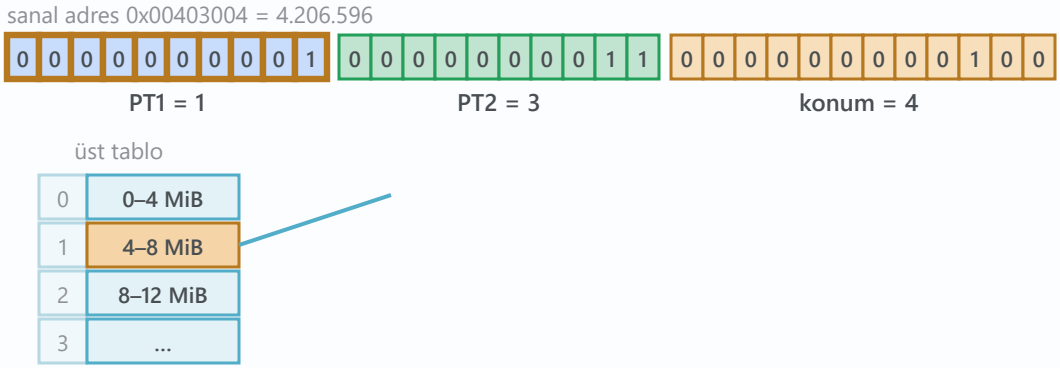


Sanal adres **0x00403004** (4.206.596): 4 MiB ile 8 MiB arasında, veri bölgesinde (Tanenbaum Şekil 3-13).

Adım 1 / 4

Örnekteki çerçeve numaraları açıklama amaçlıdır. 32 bit x86'da üst tablonun adresi `CR3` 'tedir; her iki düzeydeki tablo tam bir 4 KiB sayfa doldurur (1024 × 4 B).

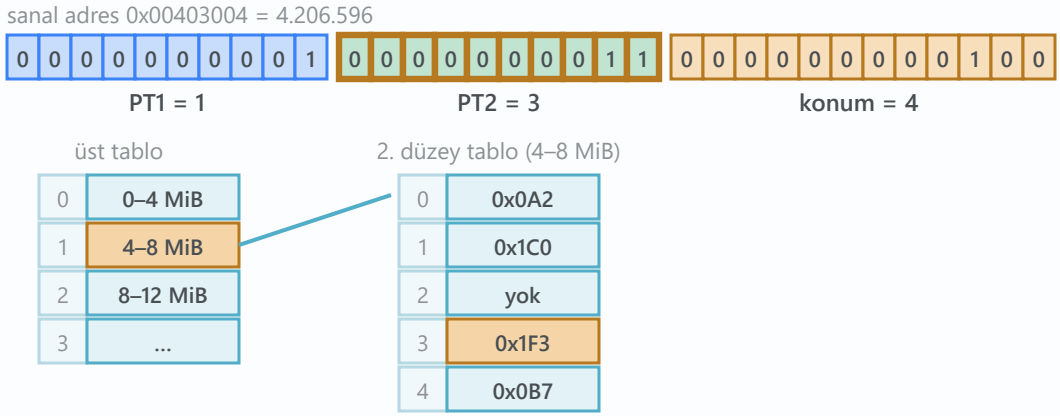
İki Düzeyli Çeviri Örneği



Üst 10 bit **PT1 = 1**: üst tablonun 1. girdisi 4–8 MiB bölgesinin ikinci düzey tablosunu gösterir. Her girdi 4 MiB'ı kapsar.

Örnekteki çerçeve numaraları açıklama amaçlıdır. 32 bit x86'da üst tablonun adresi `CR3` 'tedir; her iki düzeydeki tablo tam bir 4 KiB sayfa doldurur (1024 × 4 B).

İki Düzeyli Çeviri Örneği

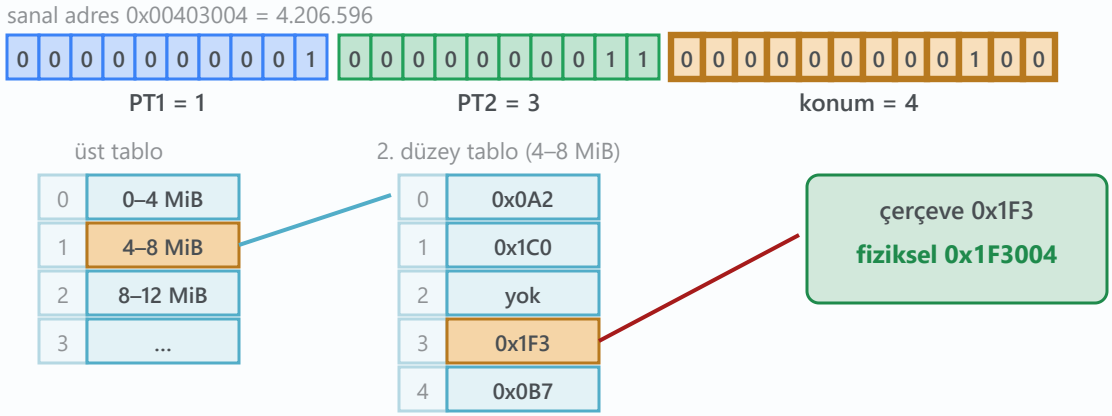


Sonraki 10 bit **PT2 = 3**: o tablonun 3. girdisine bakılır (12.288–16.383 bayt aralığındaki sayfa). Mevcut biti 0 olsaydı sayfa hatası.

Adım 3 / 4

Örnekteki çerçeve numaraları açıklama amaçlıdır. 32 bit x86'da üst tablonun adresi `CR3` 'tedir; her iki düzeydeki tablo tam bir 4 KiB sayfa doldurur (1024 × 4 B).

İki Düzeyli Çeviri Örneği



Girdi çerçeve numarasını verir (örnekte 0x1F3); **konum = 4** eklenir → fiziksel **0x1F3004**. Iskada 2 tablo erişimi + 1 veri erişimi.

Adım 4 / 4

Örnekteki çerçeve numaraları açıklama amaçlıdır. 32 bit x86'da üst tablonun adresi `CR3` 'tedir; her iki düzeydeki tablo tam bir 4 KiB sayfa doldurur (1024 × 4 B).

İki Düzeyli Çeviri: Kod İzleme

```
1  /* 32 bit: 10 + 10 + 12 bit */
2  uint32_t cevir(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
6    pte_t e1 = ust[p1];
7    if (!e1.p) sayfa_hatasi(va);
8    pte_t e2 = e1.tablo[p2];
9    if (!e2.p) sayfa_hatasi(va);
10   return (e2.cerceve << 12) | d;
11 }
```

va = 0x00801A3C

0000000010	0000000001	101000111100
p1	p2	d

Sanal adres 0x00801A3C (8 MiB'ın biraz üstü). Üst tabloda yalnız girdi 2 geçerli; onun tablosunda girdi 1 → çerçeve 0x2F.

Adım 1 / 9

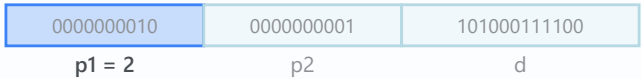
DEĞİŞKENLER

va = **0x00801A3C**

İki Düzeyli Çeviri: Kod İzleme

```
1  /* 32 bit: 10 + 10 + 12 bit */
2  uint32_t cevir(uint32_t va) {
▶    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
6    pte_t e1 = ust[p1];
7    if (!e1.p) sayfa_hatasi(va);
8    pte_t e2 = e1.tablo[p2];
9    if (!e2.p) sayfa_hatasi(va);
10   return (e2.cerceve << 12) | d;
11 }
```

va = 0x00801A3C



Üst 10 bit: 0x00801A3C >> 22 = **2**.

Adım 2 / 9

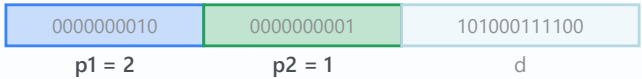
DEĞİŞKENLER

va = **0x00801A3C** p1 = **2**

İki Düzeyli Çeviri: Kod İzleme

```
1  /* 32 bit: 10 + 10 + 12 bit */
2  uint32_t cevir(uint32_t va) {
3    uint32_t p1 = va >> 22;
▶    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
6    pte_t e1 = ust[p1];
7    if (!e1.p) sayfa_hatasi(va);
8    pte_t e2 = e1.tablo[p2];
9    if (!e2.p) sayfa_hatasi(va);
10   return (e2.cerceve << 12) | d;
11 }
```

va = 0x00801A3C



Orta 10 bit: (va >> 12) & 0x3FF = 0x801 & 0x3FF = **1**.

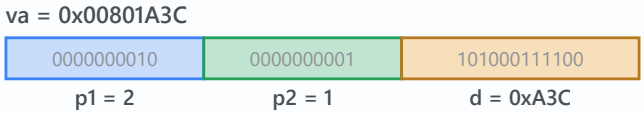
Adım 3 / 9

DEĞİŞKENLER

va = **0x00801A3C** | p1 = **2** | p2 = **1**

İki Düzeyli Çeviri: Kod İzleme

```
1  /* 32 bit: 10 + 10 + 12 bit */
2  uint32_t cevir(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
▶    uint32_t d  = va & 0xFFF;
6    pte_t e1 = ust[p1];
7    if (!e1.p) sayfa_hatasi(va);
8    pte_t e2 = e1.tablo[p2];
9    if (!e2.p) sayfa_hatasi(va);
10   return (e2.cerceve << 12) | d;
11 }
```



Alt 12 bit konum: **0xA3C**; çeviride hiç değişmez.

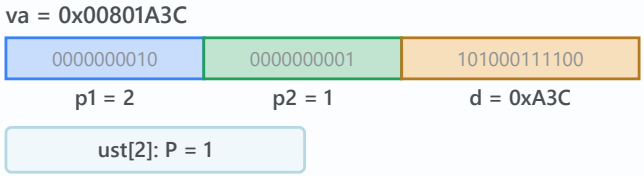
Adım 4 / 9

DEĞİŞKENLER

`va = 0x00801A3C` `p1 = 2` `p2 = 1` `d = 0xA3C`

İki Düzeyli Çeviri: Kod İzleme

```
1  /* 32 bit: 10 + 10 + 12 bit */
2  uint32_t cevir(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
▶    pte_t e1 = ust[p1];
▶    if (!e1.p) sayfa_hatasi(va);
8    pte_t e2 = e1.tablo[p2];
9    if (!e2.p) sayfa_hatasi(va);
10   return (e2.cerceve << 12) | d;
11 }
```



Üst tablonun 2. girdisi geçerli (P = 1): ikinci düzey tablonun çerçevesini verir. Birinci bellek erişimi.

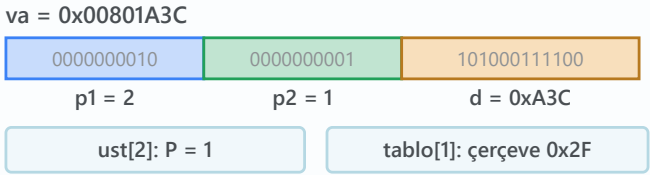
Adım 5 / 9

DEĞİŞKENLER

`va = 0x00801A3C` `p1 = 2` `p2 = 1` `d = 0xA3C` `e1 = P=1`

İki Düzeyli Çeviri: Kod İzleme

```
1  /* 32 bit: 10 + 10 + 12 bit */
2  uint32_t cevir(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
6    pte_t e1 = ust[p1];
7    if (!e1.p) sayfa_hatasi(va);
▶    pte_t e2 = e1.tablo[p2];
▶    if (!e2.p) sayfa_hatasi(va);
10   return (e2.cerceve << 12) | d;
11 }
```



İkinci düzey tablonun 1. girdisi: çerçeve **0x2F**, P = 1. İkinci bellek erişimi.

Adım 6 / 9

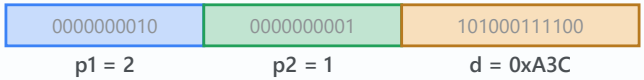
DEĞİŞKENLER

`va = 0x00801A3C` `p1 = 2` `p2 = 1` `d = 0xA3C` `cerceve = 0x2F`

İki Düzeyli Çeviri: Kod İzleme

```
1  /* 32 bit: 10 + 10 + 12 bit */
2  uint32_t cevir(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
6    pte_t e1 = ust[p1];
7    if (!e1.p) sayfa_hatasi(va);
8    pte_t e2 = e1.tablo[p2];
9    if (!e2.p) sayfa_hatasi(va);
▶    return (e2.cerceve << 12) | d;
11 }
```

va = 0x00801A3C



ust[2]: P = 1

tablo[1]: çerçeve 0x2F

fiziksel = 0x0002FA3C

0x2F << 12 = 0x2F000; konumla birleşince fiziksel **0x0002FA3C**. Üçüncü erişim veridir.

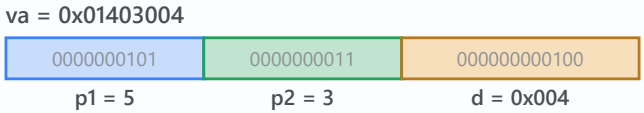
Adım 7 / 9

DEĞİŞKENLER

`va = 0x00801A3C` `cerceve = 0x2F` `pa = 0x0002FA3C`

İki Düzeyli Çeviri: Kod İzleme

```
1  /* 32 bit: 10 + 10 + 12 bit */
2  uint32_t cevir(uint32_t va) {
▶    uint32_t p1 = va >> 22;
▶    uint32_t p2 = (va >> 12) & 0x3FF;
▶    uint32_t d  = va & 0xFFF;
6    pte_t e1 = ust[p1];
7    if (!e1.p) sayfa_hatasi(va);
8    pte_t e2 = e1.tablo[p2];
9    if (!e2.p) sayfa_hatasi(va);
10   return (e2.cerceve << 12) | d;
11 }
```



İkinci adres 0x01403004: p1 = **5**, p2 = 3, konum = 0x004.

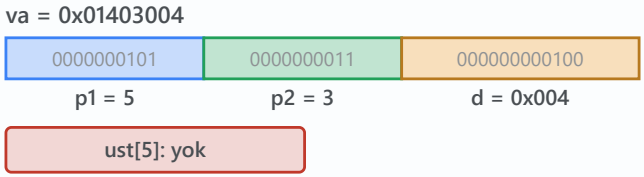
Adım 8 / 9

DEĞİŞKENLER

va = **0x01403004** p1 = **5** p2 = **3** d = **0x004**

İki Düzeyli Çeviri: Kod İzleme

```
1  /* 32 bit: 10 + 10 + 12 bit */
2  uint32_t cevir(uint32_t va) {
3    uint32_t p1 = va >> 22;
4    uint32_t p2 = (va >> 12) & 0x3FF;
5    uint32_t d  = va & 0xFFF;
▶    pte_t e1 = ust[p1];
▶    if (!e1.p) sayfa_hatasi(va);
8    pte_t e2 = e1.tablo[p2];
9    if (!e2.p) sayfa_hatasi(va);
10   return (e2.cerceve << 12) | d;
11 }
```



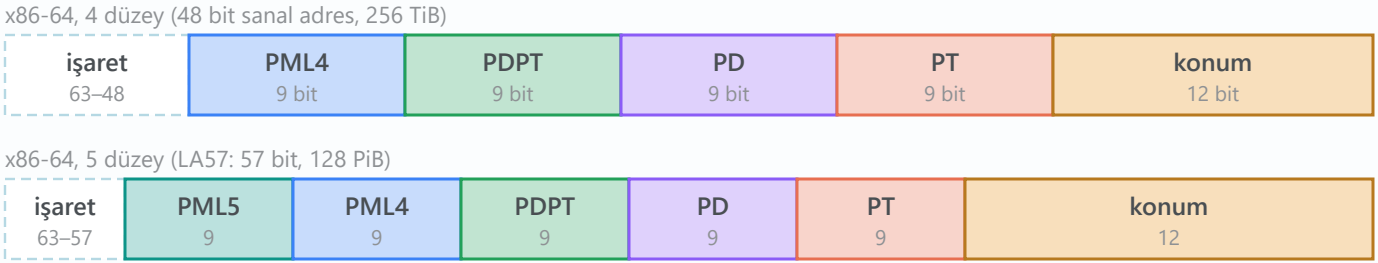
Üst tablonun 5. girdisi geçersiz (P = 0): 20–24 MiB bölgesi için ikinci düzey tablo **hiç yok**. MMU tuzak üretir; tablo için bellek harcanmamıştır.

Adım 9 / 9

DEĞİŞKENLER

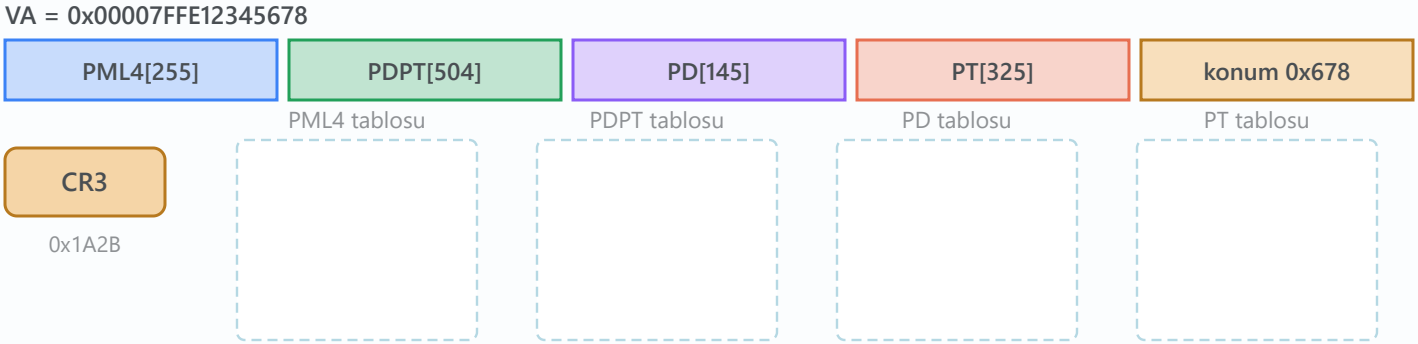
`va = 0x01403004` `p1 = 5` `e1 = P=0` `sonuc = sayfa hatası`

x86-64: Dört ve Beş Düzeyli Sayfalama



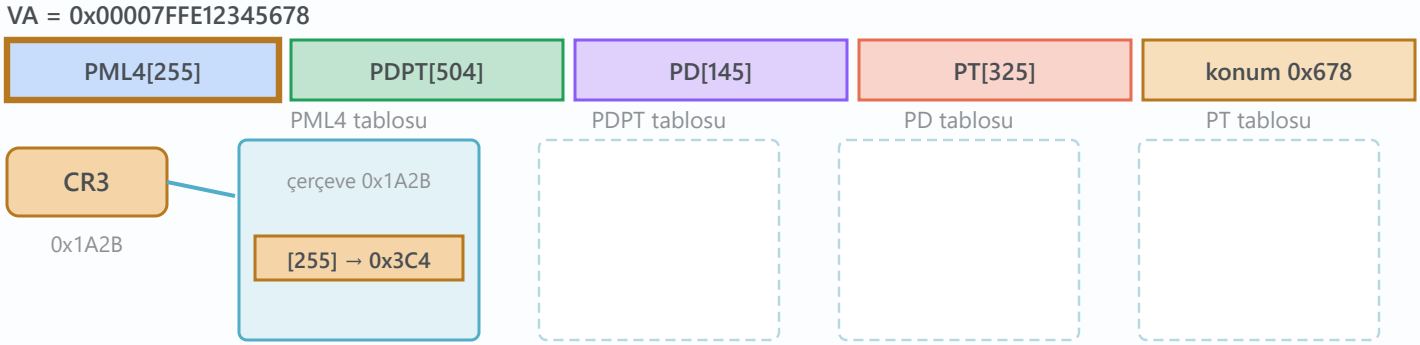
- Her tablo 512 girdi × 8 B = **4 KiB**, yani tam bir sayfa; 9 bit indis. 4 × 9 + 12 = **48 bit**.
- Üst 16 bit, bit 47'nin kopyası olmalıdır (**kanonik adres**): alt yarı kullanıcıya, üst yarı çekirdeğe.
- 5 düzey (Intel Ice Lake sunucu, AMD Zen 4 ve sonrası) 57 bit sanal adres verir; Linux 4.14'ten beri destekler.
- Hiyerarşik tablo 64 bitte de çalışır: bütün 64 bit adres kullanılmadığı için bugünkü x86-64 ve ARMv8 tam da hiyerarşik tablo kullanır.
- Bir orta düzey girdisinde `PS` biti 1 ise yürüyüş orada biter: PD düzeyinde **2 MiB**, PDPT düzeyinde **1 GiB** büyük sayfa.

Dört Düzeyli Yürüyüş



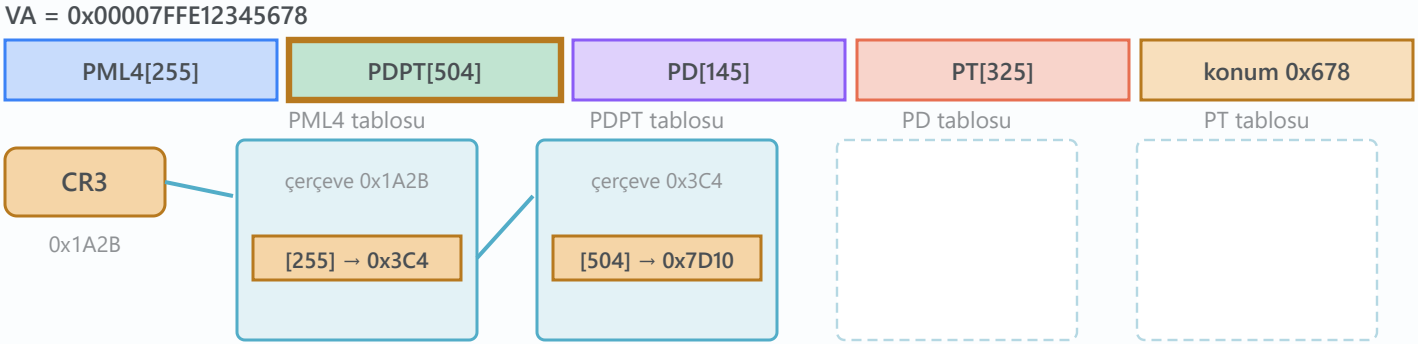
Kullanıcı yığınına yakın bir adres. Bölünmesi (9+9+9+9+12): PML4 = 255, PDPT = 504, PD = 145, PT = 325, konum = 0x678.

Dört Düzeyli Yürüyüş



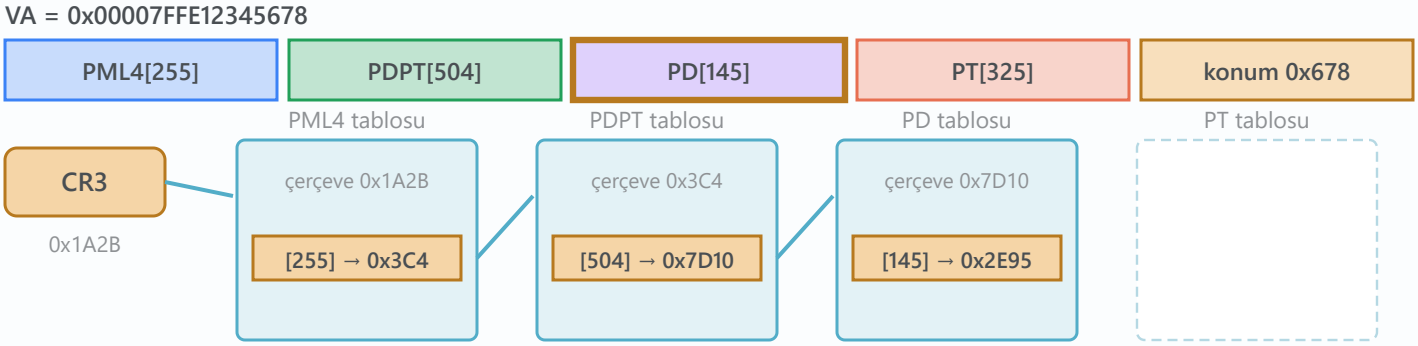
CR3 PML4 tablosunun çerçevesini verir; girdi 255 okunur → PDPT tablosunun çerçevesi.

Dört Düzeyli Yürüyüş



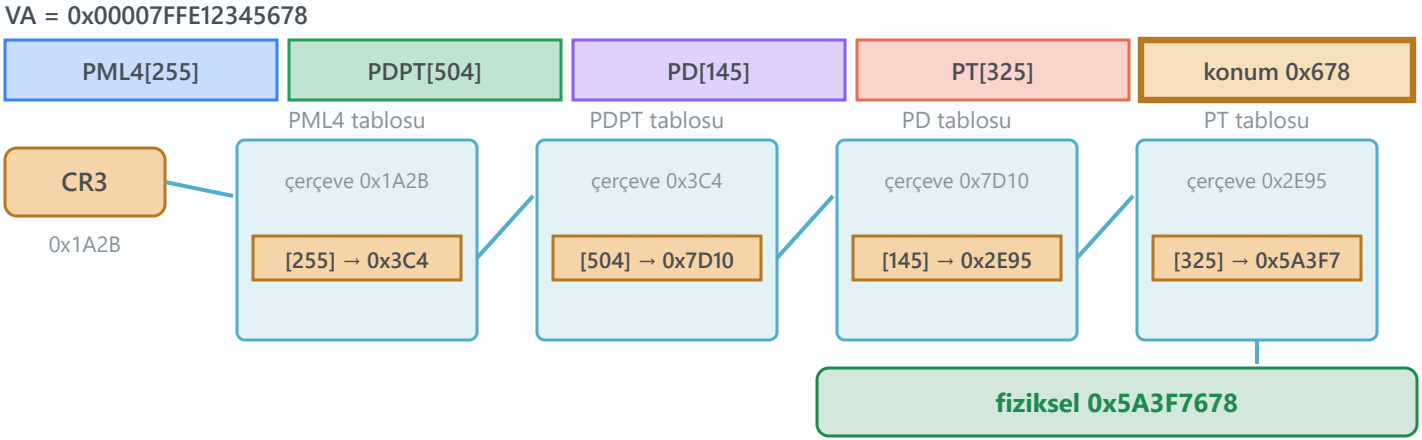
PDPT[504] → sayfa dizini (PD) çerçevesi. Burada `PS` = 1 olsaydı 1 GiB'lık sayfa bulunmuş olurdu.

Dört Düzeyli Yürüyüş



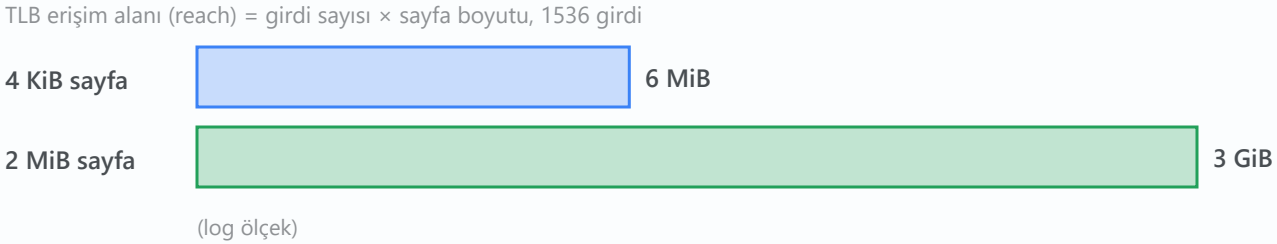
PD[145] → son düzey tablo. `PS` = 1 olsaydı 2 MiB'lık sayfa.

Dört Düzeyli Yürüyüş



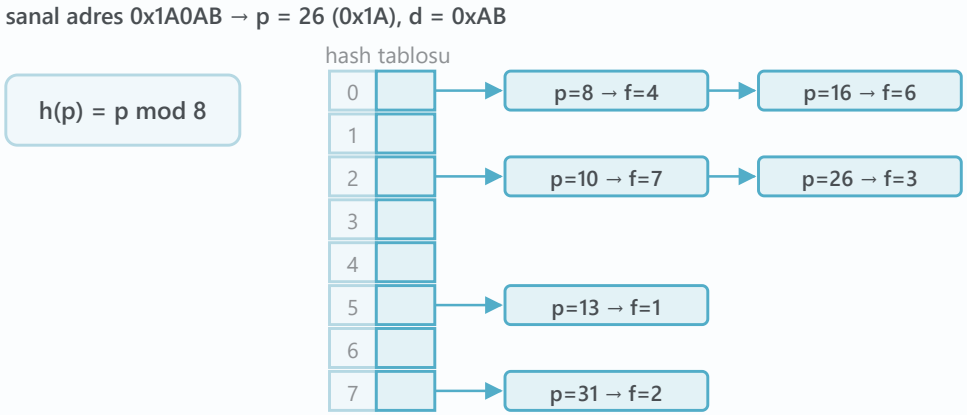
PT[325] çerçeve 0x5A3F7'i verir; konum eklenir. TLB ıskasında **4 bellek erişimi** + veri erişimi; çerçeve değerleri örnektir.

Büyük (Huge) Sayfalar



- Büyük çalışma kümeli programlar (veritabanı, sanal makine, yapay zekâ) 4 KiB sayfalarla sürekli TLB ıskası yaşar.
- 1 GiB'ı 4 KiB ile eşlemek 262.144 PTE = 2 MiB tablo ister; 2 MiB sayfalarla yalnızca 512 PD girdisi (4 KiB).
- Linux: **hugetlbfs** (önceden ayrılan 2 MiB / 1 GiB), **şeffaf büyük sayfalar** (THP, çekirdek kendiliğinden birleştirir), son sürümlerde mTHP (16–64 KiB ara boyutlar). Windows: *large pages*.
- Bedel: iç parçalanma ve 2 MiB bitişik fiziksel çerçeve bulma zorunluluğu (buddy ve sıkıştırma burada geri döner).
- ARMv8'de *contiguous* biti: 16 komşu 4 KiB girdi tek TLB girdisiyle (64 KiB) tutulabilir.

Hash Sayfa Tablosu



Her kova, aynı hash değerine düşen **(sanal sayfa, çerçeve, sonraki)** girdilerinin zinciridir. Örnekte 8 kova, h(p) = p mod 8.

Adım 1 / 4

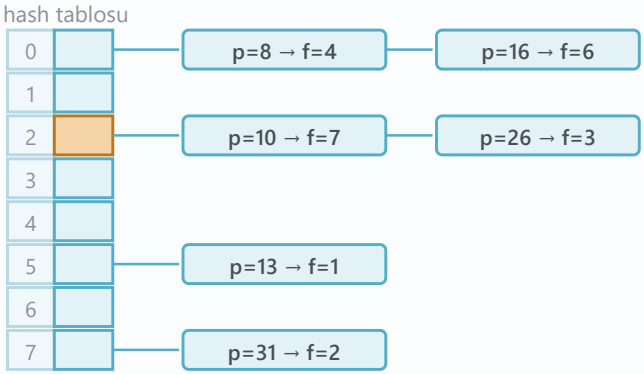
- 32 bitten büyük adres alanlarında kullanılır; tablo boyutu kullanılan sayfa sayısıyla orantılıdır, adres alanıyla değil.
- **Kümelenmiş** varyant: her girdi birden çok (ör. 16) ardışık sayfayı tutar; seyrek 64 bit uzaylar için uygundur.

Hash Sayfa Tablosu

sanal adres 0x1A0AB → p = 26 (0x1A), d = 0xAB

h(p) = p mod 8

h(26) = 2



Sayfa numarası 26 hash'lenir: kova **2**.

Adım 2 / 4

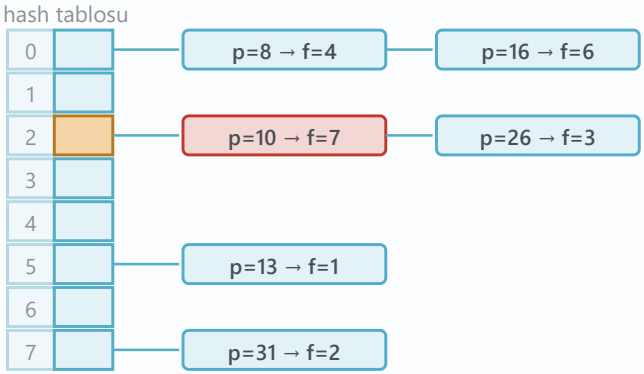
- 32 bitten büyük adres alanlarında kullanılır; tablo boyutu kullanılan sayfa sayısıyla orantılıdır, adres alanıyla değil.
- **Kümelenmiş** varyant: her girdi birden çok (ör. 16) ardışık sayfayı tutar; seyrek 64 bit uzaylar için uygundur.

Hash Sayfa Tablosu

sanal adres 0x1A0AB → p = 26 (0x1A), d = 0xAB

h(p) = p mod 8

h(26) = 2



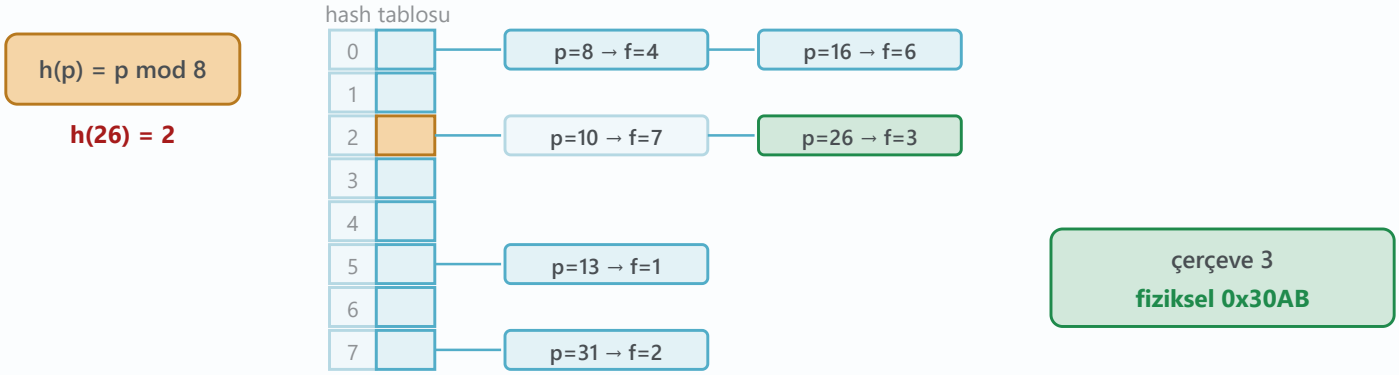
Zincirin ilk girdisi p = 10: eşleşmez, sonrakine geçilir.

Adım 3 / 4

- 32 bitten büyük adres alanlarında kullanılır; tablo boyutu kullanılan sayfa sayısıyla orantılıdır, adres alanıyla değil.
- **Kümelenmiş** varyant: her girdi birden çok (ör. 16) ardışık sayfayı tutar; seyrek 64 bit uzaylar için uygundur.

Hash Sayfa Tablosu

sanal adres 0x1A0AB → p = 26 (0x1A), d = 0xAB



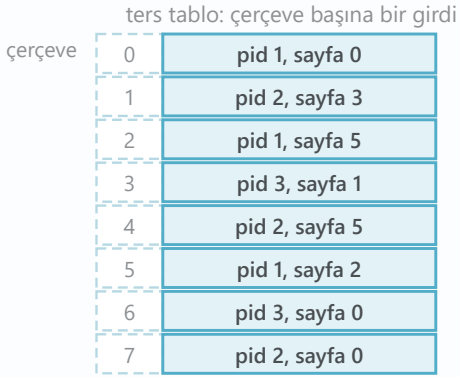
p = 26 eşleşir → çerçeve **3**; fiziksel = 3 × 4096 + 0xAB = **0x30AB**.

Adım 4 / 4

- 32 bitten büyük adres alanlarında kullanılır; tablo boyutu kullanılan sayfa sayısıyla orantılıdır, adres alanıyla değil.
- **Kümelenmiş** varyant: her girdi birden çok (ör. 16) ardışık sayfayı tutar; seyrek 64 bit uzaylar için uygundur.

Ters (Inverted) Sayfa Tablosu

mantıksal adres: (pid = 2, p = 5, d = 0x123)



Tabloda **süreç başına değil, fiziksel çerçeve başına** bir girdi vardır: o çerçevede hangi sürecin hangi sayfası.

Adım 1 / 6

- Tablo boyutu fiziksel bellekle orantılı ve sistemde **tek**: 16 GiB RAM, 4 KiB çerçeve → 4.194.304 girdi; 8 B/girdi ile 32 MiB.
- Dezavantaj: paylaşımlı bellek zor (çerçeve başına tek sanal sayfa); sayfa hatası işlemek için ayrıca süreç başına dış tablo gerekir. Örnekler: PowerPC/IBM POWER (hash'li ters tablo), IA-64.

Ters (Inverted) Sayfa Tablosu

mantıksal adres: (pid = 2, p = 5, d = 0x123)



Girdi 0: (pid 1, sayfa 0) ≠ (2, 5), sonraki.

Adım 2 / 6

- Tablo boyutu fiziksel bellekle orantılı ve sistemde **tek**: 16 GiB RAM, 4 KiB çerçeve → 4.194.304 girdi; 8 B/girdi ile 32 MiB.
- Dezavantaj: paylaşımlı bellek zor (çerçeve başına tek sanal sayfa); sayfa hatası işlemek için ayrıca süreç başına dış tablo gerekir. Örnekler: PowerPC/IBM POWER (hash'li ters tablo), IA-64.

Ters (Inverted) Sayfa Tablosu

mantıksal adres: (pid = 2, p = 5, d = 0x123)



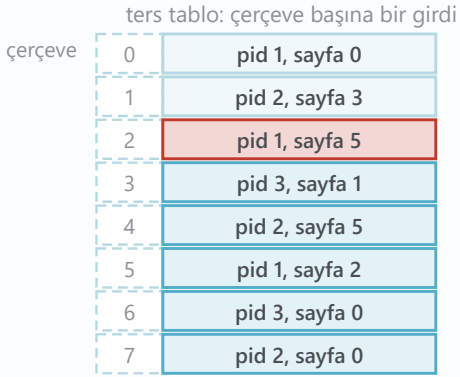
Girdi 1: (pid 2, sayfa 3) ≠ (2, 5), sonraki.

Adım 3 / 6

- Tablo boyutu fiziksel bellekle orantılı ve sistemde **tek**: 16 GiB RAM, 4 KiB çerçeve → 4.194.304 girdi; 8 B/girdi ile 32 MiB.
- Dezavantaj: paylaşımlı bellek zor (çerçeve başına tek sanal sayfa); sayfa hatası işlemek için ayrıca süreç başına dış tablo gerekir. Örnekler: PowerPC/IBM POWER (hash'li ters tablo), IA-64.

Ters (Inverted) Sayfa Tablosu

mantıksal adres: (pid = 2, p = 5, d = 0x123)

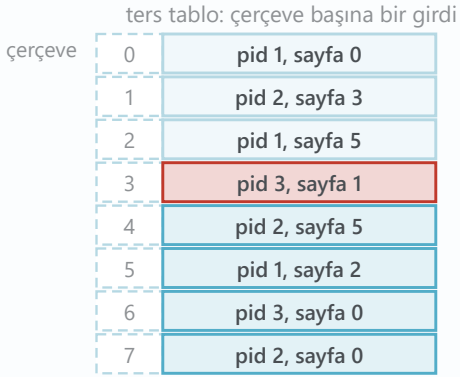


Girdi 2: (pid 1, sayfa 5) ≠ (2, 5), sonraki.

- Tablo boyutu fiziksel bellekle orantılı ve sistemde **tek**: 16 GiB RAM, 4 KiB çerçeve → 4.194.304 girdi; 8 B/girdi ile 32 MiB.
- Dezavantaj: paylaşımlı bellek zor (çerçeve başına tek sanal sayfa); sayfa hatası işlemek için ayrıca süreç başına dış tablo gerekir. Örnekler: PowerPC/IBM POWER (hash'li ters tablo), IA-64.

Ters (Inverted) Sayfa Tablosu

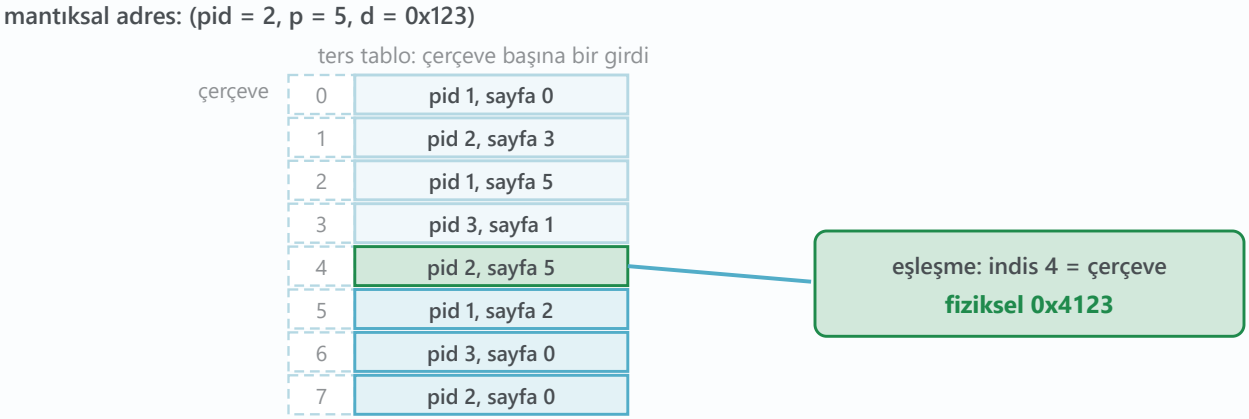
mantıksal adres: (pid = 2, p = 5, d = 0x123)



Girdi 3: (pid 3, sayfa 1) ≠ (2, 5), sonraki.

- Tablo boyutu fiziksel bellekle orantılı ve sistemde **tek**: 16 GiB RAM, 4 KiB çerçeve → 4.194.304 girdi; 8 B/girdi ile 32 MiB.
- Dezavantaj: paylaşımlı bellek zor (çerçeve başına tek sanal sayfa); sayfa hatası işlemek için ayrıca süreç başına dış tablo gerekir. Örnekler: PowerPC/IBM POWER (hash'li ters tablo), IA-64.

Ters (Inverted) Sayfa Tablosu

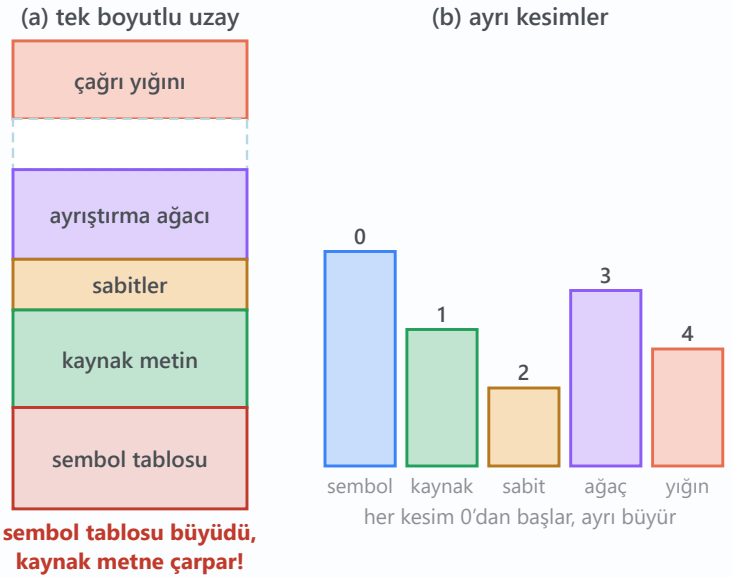


Girdi 4 eşleşir: **indisin kendisi çerçeve numarasıdır** → 4 × 4096 + 0x123 = **0x4123**. Doğrusal arama pahalı olduğundan gerçek sistemler hash tablosu + TLB kullanır.

- Tablo boyutu fiziksel bellekle orantılı ve sistemde **tek**: 16 GiB RAM, 4 KiB çerçeve → 4.194.304 girdi; 8 B/girdi ile 32 MiB.
- Dezavantaj: paylaşımlı bellek zor (çerçeve başına tek sanal sayfa); sayfa hatası işlemek için ayrıca süreç başına dış tablo gerekir. Örnekler: PowerPC/IBM POWER (hash'li ters tablo), IA-64.

Kesimleme: Neden?

- Derleyici derleme sırasında birçok tablo kurar: kaynak metin, **sembol tablosu**, sabitler, ayrıştırma ağacı, çağrı yığını. Hepsi büyüyüp küçülür.
- Tek boyutlu adres alanında bir tablo dolunca komşusu boş olsa bile **çarpışır**.
- **Kesimleme**: birbirinden bağımsız, değişken uzunluklu birden çok adres alanı. Adres = **(kesim no, kesim içi konum)**.
- Programcının bakışına uyar: ana program, işlevler, yığın, diziler, sembol tablosu ayrı birimlerdir.



Kesim Tablosu ile Çeviri

kesim tablosu

	limit	taban
0	1.000	1.400
1	400	6.300
2	400	4.300
3	1.100	3.200
4	1.000	4.700



Her girdi bir kesimin **taban** (fiziksel başlangıç) ve **limit** (uzunluk) değerini tutar; tablonun yeri ve boyu STBR ve STLR yazmaçlarındadır.

Adım 1 / 5

Silberschatz'ın beş kesimli örneği. Kesimleme, her kesim için ayrı bir taban-limit çiftidir.

Kesim Tablosu ile Çeviri

kesim tablosu

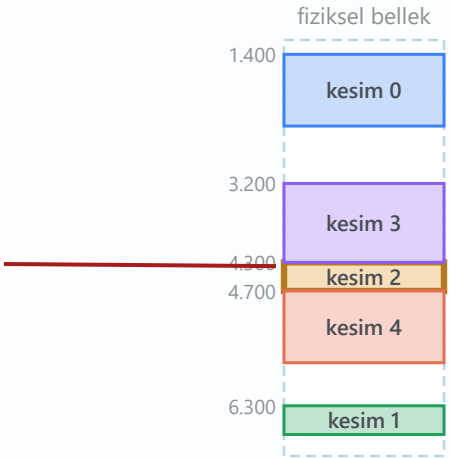
	limit	taban
0	1.000	1.400
1	400	6.300
2	400	4.300
3	1.100	3.200
4	1.000	4.700

(s, d) = (2, 53)

d < limit ? 53 < 400

evet → taban + d

4.300 + 53 = 4.353



(2, 53): 53 < 400 → 4.300 + 53 = **4.353**.

Silberschatz'ın beş kesimli örneği. Kesimleme, her kesim için ayrı bir taban-limit çiftidir.

Kesim Tablosu ile Çeviri

kesim tablosu

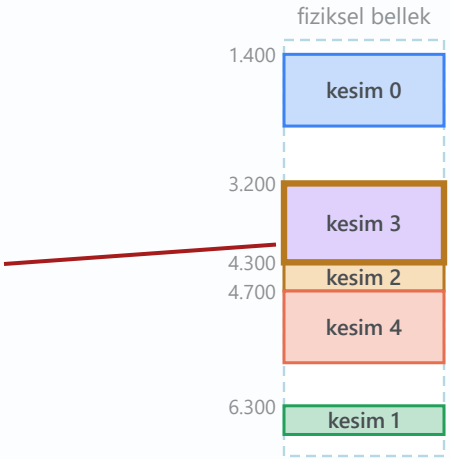
	limit	taban
0	1.000	1.400
1	400	6.300
2	400	4.300
3	1.100	3.200
4	1.000	4.700

(s, d) = (3, 852)

d < limit ? 852 < 1.100

evet → taban + d

3.200 + 852 = 4.052



(3, 852): 852 < 1.100 → 3.200 + 852 = **4.052**.

Silberschatz'ın beş kesimli örneği. Kesimleme, her kesim için ayrı bir taban-limit çiftidir.

Kesim Tablosu ile Çeviri

kesim tablosu

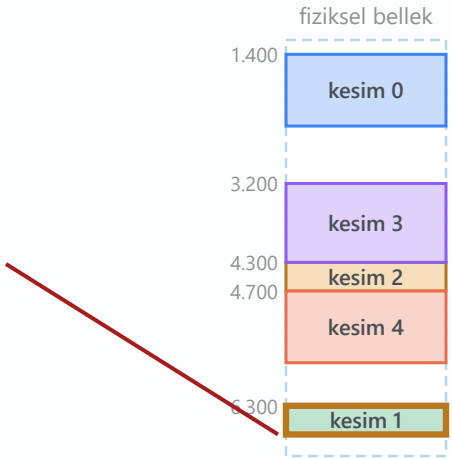
	limit	taban
0	1.000	1.400
1	400	6.300
2	400	4.300
3	1.100	3.200
4	1.000	4.700

(s, d) = (1, 399)

d < limit ? 399 < 400

evet → taban + d

6.300 + 399 = 6.699



(1, 399): 399 < 400 → 6.300 + 399 = **6.699**.

Silberschatz'ın beş kesimli örneği. Kesimleme, her kesim için ayrı bir taban-limit çiftidir.

Kesim Tablosu ile Çeviri

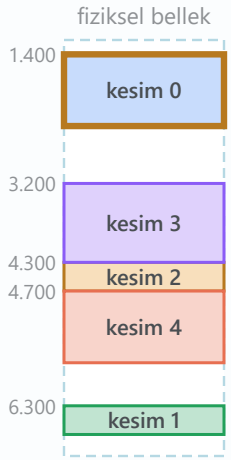
kesim tablosu

	limit	taban
0	1.000	1.400
1	400	6.300
2	400	4.300
3	1.100	3.200
4	1.000	4.700

(s, d) = (0, 1.222)

d < limit ? 1.222 < 1.000

hayır → tuzak (adresleme hatası)



(0, 1.222): kesim 0'ın boyu yalnızca 1.000 bayt → **tuzak**. Denetim olmasaydı 2.622 adresine, yani kesimin dışına erişilirdi.

Silberschatz'ın beş kesimli örneği. Kesimleme, her kesim için ayrı bir taban-limit çiftidir.

Kesimi Seçen Bitler, Aşağı Büyüyen Yığın

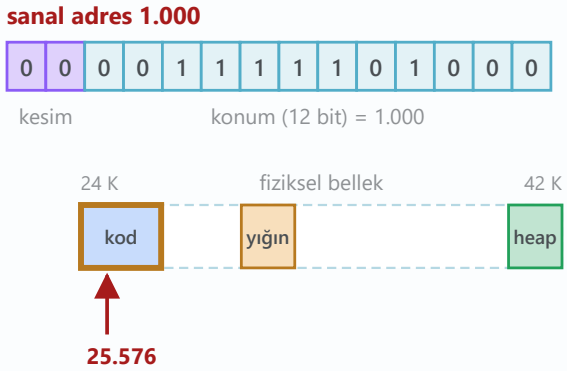
kesim	taban	boy	yön	koruma
00 kod	24.576	3.072	+	r-x
01 heap	40.960	2.048	+	rw-
11 yığın	32.768	2.048	–	rw-

kesim 00 → kod

konum < boy ? 1.000 < 3.072

evet → taban + konum

24.576 + 1.000 = 25.576



1.000: üst bitler **00** → kod. 1.000 < 3.072 → 24.576 + 1.000 = **25.576**.

Adım 1 / 4

Örnekte sanal adres 14 bittir (16 KiB). Üst iki bit en çok dört kesim, kalan 12 bit kesim başına en çok 4 KiB verir. Kesim yazmacındaki **yön** biti yığını, **koruma** bitleri paylaşımı mümkün kılar: `r-x` kod kesimi birçok süreçte aynı çerçevelere eşlenebilir, ona yazma girişimi koruma hatasıdır.

KESİMLEME · ADIM ADIM

Kesimi Seçen Bitler, Aşağı Büyüyen Yığın

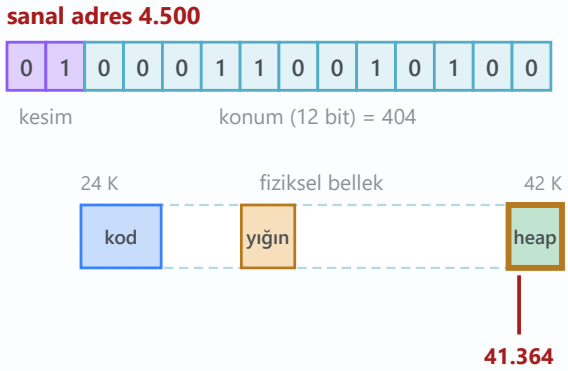
kesim	taban	boy	yön	koruma
00 kod	24.576	3.072	+	r-x
01 heap	40.960	2.048	+	rw-
11 yığın	32.768	2.048	–	rw-

kesim 01 → heap

konum < boy ? 404 < 2.048

evet → taban + konum

40.960 + 404 = 41.364



4.500: **01** → heap. Konum 4.500 − 4.096 = 404 → 40.960 + 404 = **41.364**. Sanal adres + taban (45.460) yanlış olurdu.

Adım 2 / 4

Örnekte sanal adres 14 bittir (16 KiB). Üst iki bit en çok dört kesim, kalan 12 bit kesim başına en çok 4 KiB verir. Kesim yazmacındaki **yön** biti yığını, **koruma** bitleri paylaşımı mümkün kılar: `r-x` kod kesimi birçok süreçte aynı çerçevelere eşlenebilir, ona yazma girişimi koruma hatasıdır.

KESİMLEME · ADIM ADIM

Kesimi Seçen Bitler, Aşağı Büyüyen Yığın

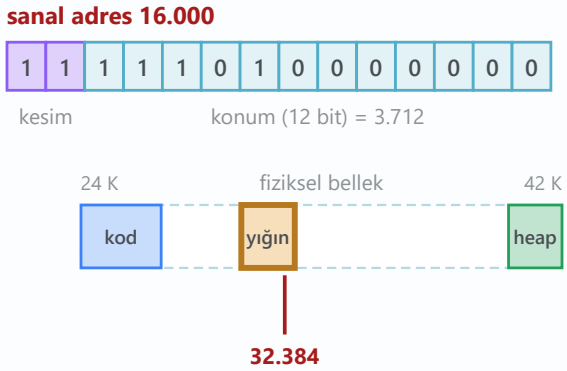
kesim	taban	boy	yön	koruma
00 kod	24.576	3.072	+	r-x
01 heap	40.960	2.048	+	rw-
11 yığın	32.768	2.048	–	rw-

kesim 11 → yığın

aşağı büyür: negatif konum = 3.712 − 4.096 = −384

|−384| ≤ 2.048 → geçerli

32.768 − 384 = 32.384



16.000: **11** → yığın. Yığın aşağı büyüdüğü için konumdan en büyük kesim boyu (4.096) çıkarılır: −384 → **32.384**.

Adım 3 / 4

Örnekte sanal adres 14 bittir (16 KiB). Üst iki bit en çok dört kesim, kalan 12 bit kesim başına en çok 4 KiB verir. Kesim yazmacındaki **yön** biti yığını, **koruma** bitleri paylaşımı mümkün kılar: `r-x` kod kesimi birçok süreçte aynı çerçevelere eşlenebilir, ona yazma girişimi koruma hatasıdır.

KESİMLEME · ADIM ADIM

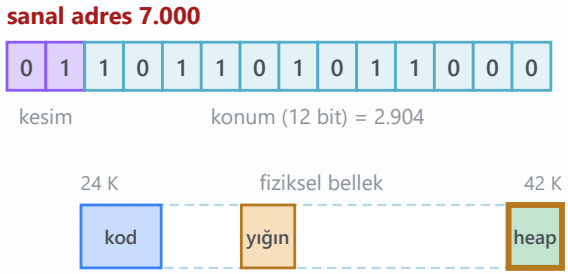
Kesimi Seçen Bitler, Aşağı Büyüyen Yığın

kesim	taban	boy	yön	koruma
00 kod	24.576	3.072	+	r-x
01 heap	40.960	2.048	+	rw-
11 yığın	32.768	2.048	–	rw-

kesim 01 → heap

konum < boy ? 2.904 < 2.048

hayır → tuzak (koruma hatası)



7.000: **01** → heap ama konum 2.904 ≥ 2.048: donanım **tuzak** üretir, İS süreci `SIGSEGV` ile sonlandırır.

Adım 4 / 4

Örnekte sanal adres 14 bittir (16 KiB). Üst iki bit en çok dört kesim, kalan 12 bit kesim başına en çok 4 KiB verir. Kesim yazmacındaki **yön** biti yığını, **koruma** bitleri paylaşımı mümkün kılar: `r-x` kod kesimi birçok süreçte aynı çerçevelere eşlenebilir, ona yazma girişimi koruma hatasıdır.

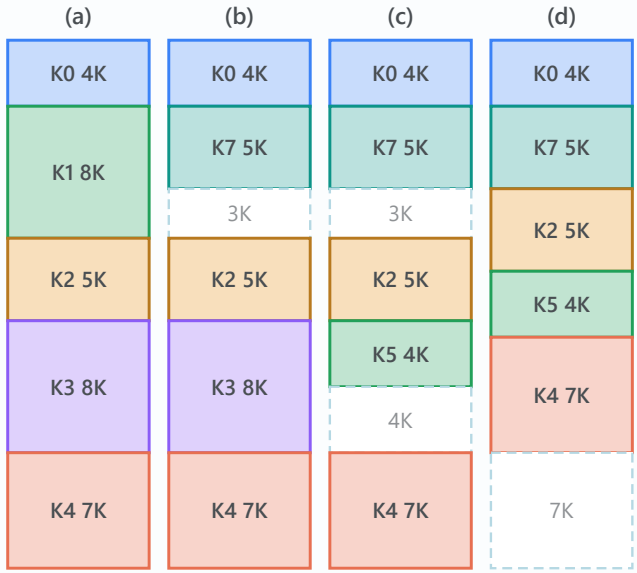
Kesimlemenin Artı ve Eksileri

Avantajlar

- Büyüyen/küçülen tablolar birbirini etkilemez.
- Kesimler ayrı derlenebilir; bir işlev değişince diğerlerinin adresi değişmez.
- Kütüphane bir kesime konup **paylaşılabilir**.
- Kesim başına koruma: kod `x`, sabitler `r`, veri `rw`.

Dezavantajlar

- Değişken uzunluk → **dış parçalanma** (dama tahtası), sıkıştırma.
- Kesim bütünüyle bellekte olmalı; çözüm kesimleri sayfalamak (MULTICS, Intel 386).

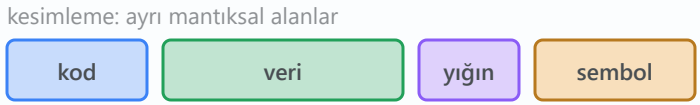


(b)–(c) dış parçalanma (dama tahtası), (d) sıkıştırma

Sayfalama ve Kesimleme Karşılaştırması

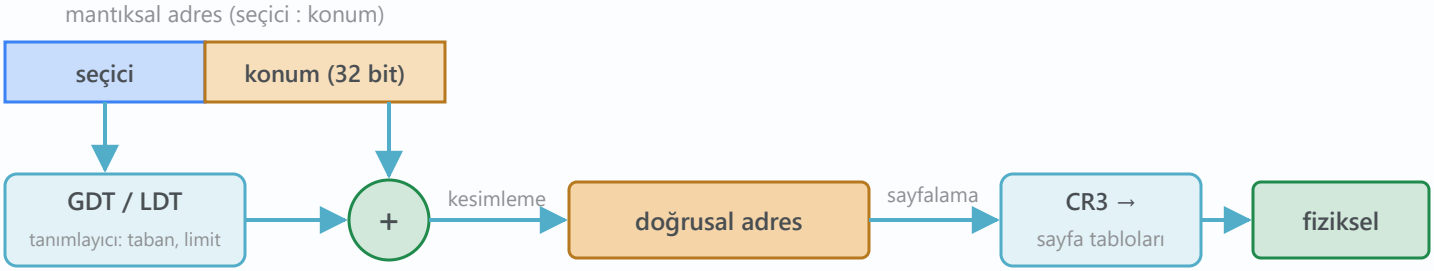
Durum	Sayfalama	Kesimleme
Programcı tekniğin farkında olmalı mı?	Hayır	Evet
Kaç doğrusal adres alanı var?	1	Çok
Toplam adres alanı fiziksel belleği aşabilir mi?	Evet	Evet
Kod ve veri ayrılıp ayrı korunabilir mi?	Sayfa bitleriyle evet*	Evet
Boyu değişen tablolar kolayca yerleşir mi?	Hayır	Evet
Kullanıcılar arasında işlev paylaşımı kolay mı?	Paylaşımlı sayfalarla*	Evet
Parçalanma	İç (son sayfa)	Dış
Neden icat edildi?	Fiziksel bellekten büyük doğrusal adres alanı	Program ve veriyi bağımsız mantıksal uzaylara ayırmak; paylaşım, koruma





Tanenbaum Şekil 3-32'den uyarlandı. * Tanenbaum burada sayfalama için "hayır" der; modern R/W/NX bitleri ve paylaşımlı sayfalar aynı işi sayfa düzeyinde yapar.

x86: Kesimleme + Sayfalama ve Düz Model



- 8086 gerçek mod: fiziksel = kesim × 16 + konum (20 bit, 1 MiB); koruma yok.
- 80386 korumalı mod: 16 bitlik seçici (13 bit indis, tablo seçimi, RPL) GDT/LDT'den 8 baytlık tanımlayıcı seçer; taban + konum = **doğrusal adres**, sonra iki düzeyli sayfalama. MULTICS fikrinin donanımdaki karşılığı.
- Linux ve Windows 32 bitte bile tüm kesimlerin tabanını 0, limitini 4 GiB yaparak **düz model** kullandı: kesimleme fiilen kapalı, koruma sayfalarla.
- **x86-64 uzun mod**: CS, DS, ES, SS tabanı donanımca 0 sayılır, limit denetlenmez. Yalnızca **FS ve GS** tabanı kullanılır: kullanıcıda iş parçacığı yerel depolama (TLS), çekirdekte çekirdek başına veri.

GERÇEK MİMARİLER

ARMv8-A ve RISC-V Notu

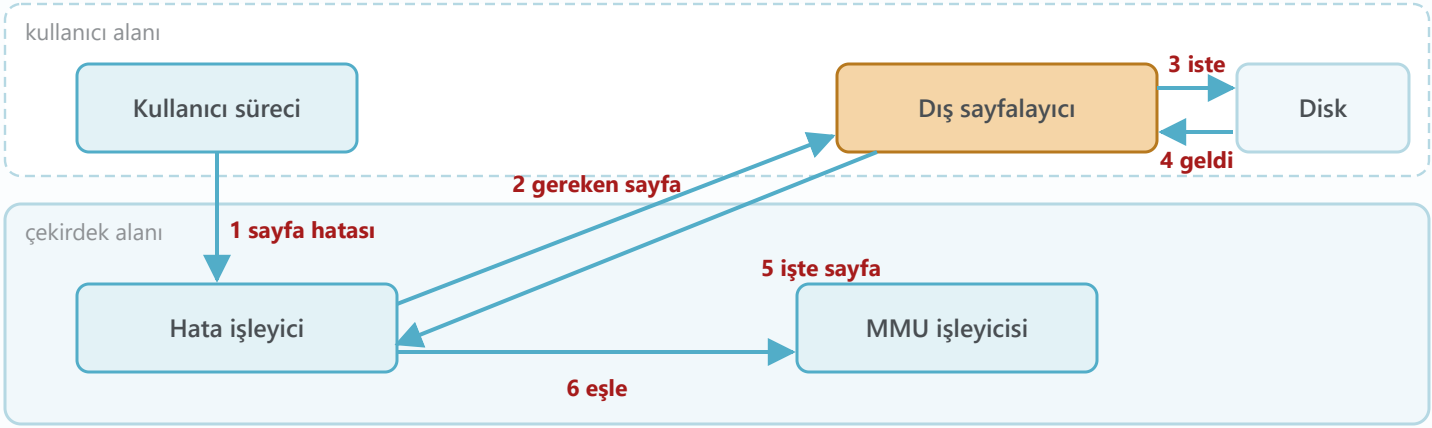
	ARMv8-A (AArch64)	RISC-V
Sayfa (granül)	4, 16 veya 64 KiB; Apple Silicon ve bazı Android 15+ aygıtlar 16 KiB	4 KiB (+ 2 MiB, 1 GiB, 512 GiB büyük sayfa)
Sanal adres	48 bit (4 KiB ile 4 düzey, L0–L3); LVA/LPA2 ile 52 bit	Sv39 (3 düzey), Sv48 (4), Sv57 (5)
Tablo tabanı	`TTBR0_EL1` alt yarı (kullanıcı), `TTBR1_EL1` üst yarı (çekirdek)	`satp` (mod + ASID + kök çerçeve)
Süreç etiketi	ASID 8/16 bit	ASID (16 bite kadar)
Koruma	AP bitleri, UXN/PXN (çalıştırma yasağı), PAN (≈ SMAP)	R/W/X/U bitleri, SUM
TLB	Donanım yürüyüşü; A/D bitleri isteğe bağlı donanımla (v8.1); contiguous biti	Donanım yürüyüşü (çoğu gerçekleme); `sfence.vma`



Sv39: 3 düzey × 512 girdi; satp kök tablonun çerçevesini ve ASID'i tutar

İki mimaride de kesimleme yoktur: düz sanal adres, çok düzeyli sayfa tablosu, TLB ve ASID. x86-64 ile aynı fikirler, farklı adlar.

Politika ve Mekanizma Ayrımı



- İşletim sistemi sayfalamaya dört anda katılır: süreç oluşturma (tablo kurulur), bağlam değiştirme (tablo etkinleşir, TLB), sayfa hatası ve süreç sonu (çerçeveler ve takas alanı bırakılır).
- **Mekanizma** çekirdekte: makineye bağlı *MMU işleyicisi* ve makineden bağımsız *sayfa hatası işleyicisi*. **İlke** (hangi sayfa ne zaman nereye) kullanıcı alanındaki *dış sayfalayıcıda* (Mach).
- Esneklik ve modülerlik kazanılır; bedeli kullanıcı/çekirdek sınırını birçok kez geçmektir.
- Bugünkü karşılığı: Linux `userfaultfd` (sanal makinelerin canlı göçü, CRIU), mikro çekirdeklerde (seL4, Fuchsia) kullanıcı alanı sayfalayıcıları.

ÖZET

Özet

Teknik	Fikir	Artı	Eksi
Taban-limit	Tek bitişik bölge + MMU	Basit, hızlı	Dış parçalanma, paylaşım zor
Değişken bölümleme	İlk/en iyi uyan ile delik seçimi	İç parçalanma yok	%50 kuralı, sıkıştırma
Sayfalama	Sabit sayfa ↔ çerçeve, sayfa tablosu	Dış parçalanma yok, paylaşım, koruma	Tablo boyutu, çift erişim
TLB	Çeviri önbelleği, ASID	EES ≈ bellek süresi	Bağlam değiştirme, vurusu
Hiyerarşik tablo	Tabloyu sayfala (x86-64: 4/5 düzey)	Seyrek uzay ucuz	Iskada çok erişim
Hash / ters	Kullanılan sayfa / çerçeve başına girdi	Küçük tablo	Arama, paylaşım zor
Kesimleme	(kesim, konum), kesim tablosu	Mantıksal bölme, koruma	Dış parçalanma; x86-64'te düz model

Bugünkü genel amaçlı sistemler: düz sanal adres alanı + çok düzeyli sayfalama + TLB + büyük sayfalar; fiziksel çerçeveler için buddy, çekirdek nesneleri için slab (SLUB).

Alıştırmalar

1. `for (i = 0; i < 1000; i++) a[i] = 0;` döngüsü aşağıdaki dört komuta derlenir. Doğrusal sayfa tablosu var, TLB yok: toplam kaç bellek erişimi olur? TLB olsaydı? (`a` sanal 40.000'de başlar, `int` 4 B, sayfa 4 KiB, kod tek sayfada.)

```
dongu: movl $0,(%edi,%eax,4)  /* a[i] = 0 */
       incl %eax              /* i++      */
       cmpl $1000,%eax
       jne  dongu
```

2. "Kesimi Seçen Bitler" slaytındaki kesim yazmaçlarıyla sanal 13.000 ve 14.500 adresleri nereye gider?
3. 16 B başlıklı ayırıcıda 4.096 B heap'ten üç kez `malloc(1000)` yapılır. Boş parçanın boyu kaçtır? Dördüncü `malloc(1040)` karşılanabilir mi?
4. `free(p + 4)` çağrılırsa ne olur?

▶ **Cevaplar**

ÖZET

Kontrol Soruları

1. Taban = 300.040, limit = 120.900 iken 420.940 adresine erişilirse ne olur?
2. Delikler 100, 500, 200, 300, 600 K; istekler 212, 417, 112, 426 K. Hangi algoritma hepsini yerleştirir?
3. 32 bit adres ve 4 KiB sayfada `0x12345678` için p ve d nedir?
4. Tanenbaum tablosunda (5 → 3) sanal 20.500 hangi fiziksel adrese çevrilir?
5. ε = 20 ns, t = 100 ns, α = %90 için tek düzeyli tabloda EES nedir?
6. Hiyerarşik sayfa tablosu neden 64 bit sistemlerde de kullanılabilir? x86-64 kaç düzey kullanır?
7. x86-64 uzun modda kesimlemeden geriye ne kalmıştır?
8. 4 kümeli, 64 B satırlı doğrudan eşlemeli boş önbellekte 0, 128, 256, 0 adreslerine erişiliyor. Kaç isabet olur?

▶ **Cevaplar**

Kaynaklar

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 9: Main Memory.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 3: Memory Management (§3.1–3.3, 3.5–3.7).
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Address Spaces", "Address Translation", "Segmentation", "Free-Space Management", "Paging", "TLBs", "Smaller Tables".
- D. E. Knuth, *The Art of Computer Programming*, Cilt 1, §2.5 Dynamic Storage Allocation (%50 kuralı).
- Intel 64 and IA-32 Architectures Software Developer's Manual, Cilt 3A, Bölüm 4–5 (Paging, Protection); Arm Architecture Reference Manual for A-profile (VMSAv8-64); RISC-V Privileged Specification (Sv39/48/57).
- U. Drepper, "How To Write Shared Libraries", 2011; Linux çekirdek belgeleri: `mm/`, "Transparent Hugepage Support", `userfaultfd`.

Son

Bölüm 8: Bellek Yönetimi

Sonraki bölüm: Sanal Bellek · Terim sözlüğü