Bölüm 12: Çok İşlemcili Sistemler

İşletim Sistemleri

UMA: veri yolu, çapraz çubuk, omega

NUMA ve dizin tabanlı tutarlılık

MESI ve yanlış paylaşım

Çok çekirdek, SMT, GPU

Çok işlemcili işletim sistemi türleri

Senkronizasyon: TSL, bilet, MCS

Çizelgeleme: zaman, alan, çete

Çoklu bilgisayar topolojileri

İleti geçirme ve RPC

DSM ve yük dengeleme

Sanallaştırma ve konteynerler

Dağıtık sistemler ve ara yazılım

Bulut ve kümeler

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_12_Cok_Islemcili_Sistemler.html

GİRİŞ

Öğrenme Çıktıları

Donanım

UMA (veri yolu, çapraz çubuk, omega) ve NUMA mimarilerini karşılaştırmak; dizin tabanlı tutarlılığı izlemek.

Önbellek tutarlılığı

MESI durum geçişlerini bir okuma/yazma dizisi üzerinde uygulamak; yanlış paylaşımı bulup gidermek.

İşletim sistemi ve kilitler

Ayrı işletim sistemi, usta-köle ve SMP modellerini; TSL, bilet ve MCS kilitlerini; dönme ile geçişi değerlendirmek.

Çizelgeleme

Ortak hazır kuyruğunun sınırlarını, alan paylaşımını ve çete çizelgelemeyi örnek üzerinde uygulamak.

Çoklu bilgisayarlar

Topolojilerin çapını hesaplamak; ileti geçirme, RPC, DSM ve yük dengeleme algoritmalarını açıklamak.

Sanallaştırma ve dağıtık sistemler

Hipervizör türlerini, VT-x/EPT ve konteynerleri; ara yazılım türlerini ve bulutu tanımak.

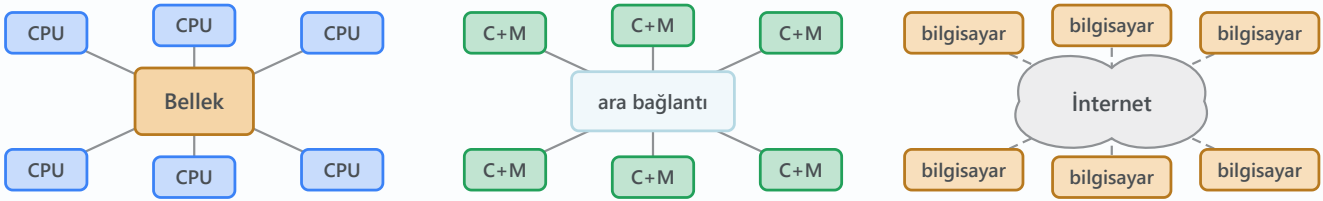
Neden Çok İşlemci?

Simetrik (SMP)
Tüm işlemciler belleğe ve G/Ç'ye eşit hakla erişir; çekirdek kodu her işlemcide çalışabilir.

Asimetrik (AMP)
İşlemciler eşit değildir: biri işletim sistemini yürütür ya da her biri ayrı yazılım çalıştırır.

Küme (cluster)
Ağ ile bağlı bağımsız bilgisayarlar tek sistem gibi çalışır; her düğümün kendi işletim sistemi var.

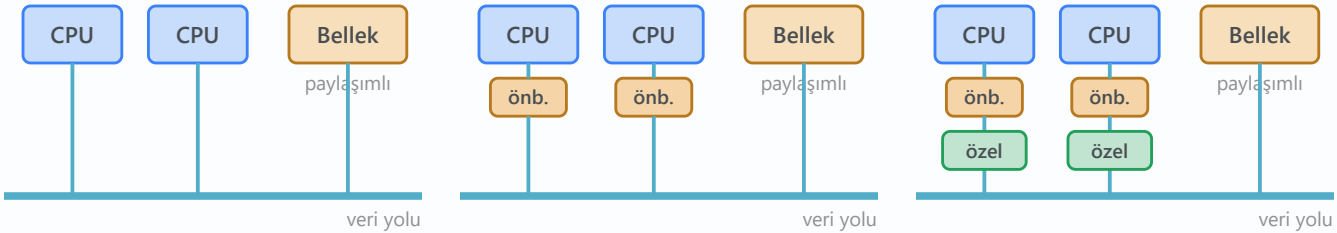
- Saat hızı ≈ 2005'ten beri 3–5 GHz'de takıldı (Dennard ölçeklemesinin sonu); transistör bütçesi artık **daha çok çekirdeğe** gidiyor.
- İşletim sistemi: işlemci tahsisi, senkronizasyon, süreçler arası iletişim ve **paylaşılan verinin tutarlılığı**.



(a) paylaşımlı bellek (b) ileti ileten çoklu bilgisayar (c) geniş alana dağıtılmış sistem

Amdahl: paralel kesim p ise n işlemcide hızlanma 1 / ((1 − p) + p/n). p = 0,95 için 8 işlemcide 5,93; 64 işlemcide 15,42; sınır 20. "İşlemci ekledikçe performans artar" ancak seri kısım küçükse doğrudur.

UMA: Veri Yolu Tabanlı Mimari

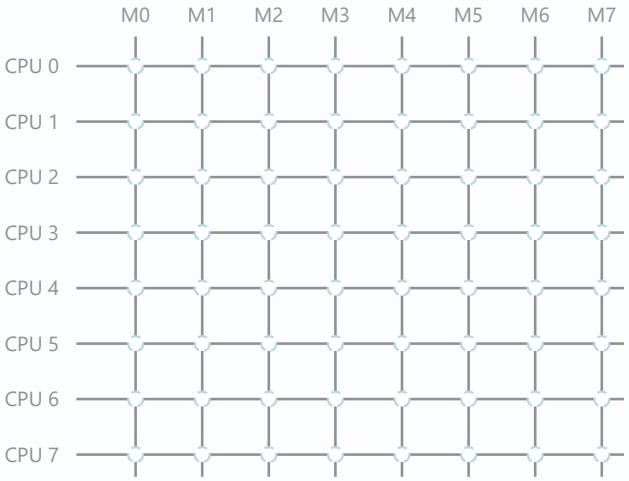


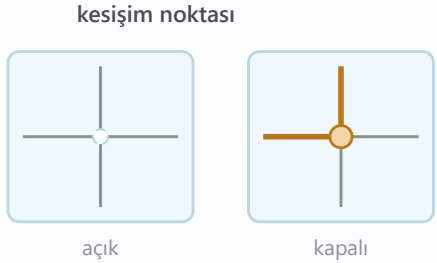
(a) önbellek yok (b) her CPU'da önbellek (c) önbellek + özel bellek

- **UMA** (*Uniform* Memory Access): her CPU her bellek sözcüğüne **aynı sürede** erişir; paralel programlama kolaylaşır.
- (a) Tek veri yolu: CPU sayısı arttıkça yol doyar; 2–4 işlemciden sonra CPU'lar zamanlarının çoğunu beklemekle geçirir.
- (b) Önbellek okumaların çoğunu yerelde karşılar, trafik azalır; ama aynı satırın kopyaları **tutarlı** tutulmalıdır (MESI).
- (c) Özel bellek (private memory): program kodu, sabitler, yığın ve yerel değişkenler; paylaşımlı bellek yalnız paylaşılan değişkenler için.

UMA: *Uniform* (tekdüze) Memory Access. Günümüzde tek bir paylaşılan veri yolu yerine yonga içi halka ya da ızgara ağları kullanılır (Intel mesh, AMD Infinity Fabric).

Çapraz Çubuk (Crossbar) Anahtar





n × n anahtar → n² kesişim

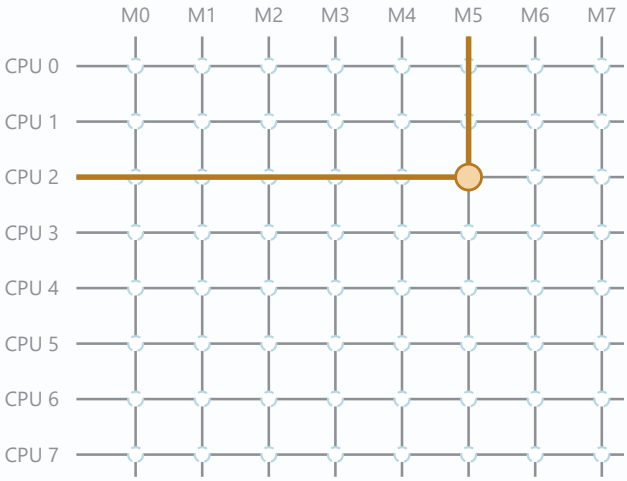
8 → 64 · 64 → 4.096

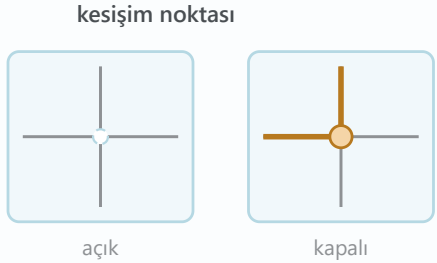
1.024 → 1.048.576

8 CPU (satır) ve 8 bellek modülü (sütun); her kesişimde küçük bir elektronik anahtar var. Hepsi **açık**.

Örnek: eski Sun Fire E25K 18 × 18 çapraz çubuk kullanıyordu; bugün yonga içinde çekirdekler ile L3 dilimleri arasında küçük çapraz çubuklar yaygındır.

Çapraz Çubuk (Crossbar) Anahtar





n × n anahtar → n² kesişim

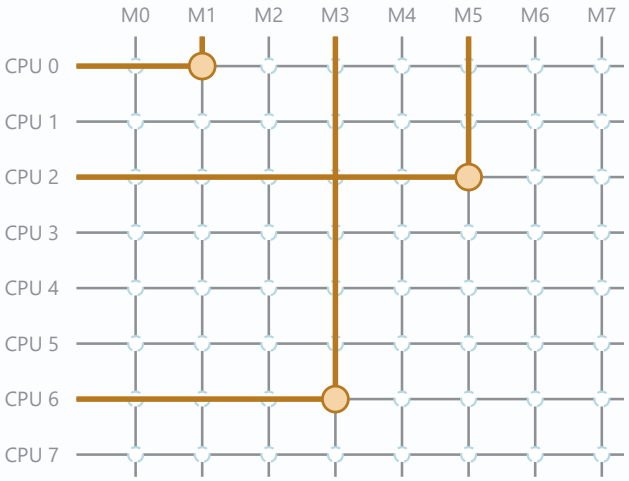
8 → 64 · 64 → 4.096

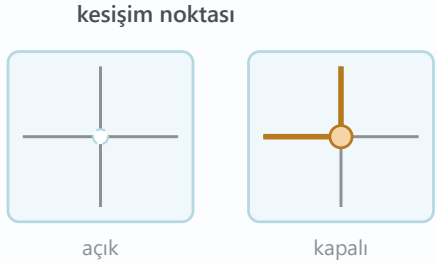
1.024 → 1.048.576

CPU 2, M5'e erişmek ister: (2, 5) kesişim noktası **kapanır**, doğrudan yol kurulur.

Örnek: eski Sun Fire E25K 18 × 18 çapraz çubuk kullanıyordu; bugün yonga içinde çekirdekler ile L3 dilimleri arasında küçük çapraz çubuklar yaygındır.

Çapraz Çubuk (Crossbar) Anahtar





n × n anahtar → n² kesişim

8 → 64 · 64 → 4.096

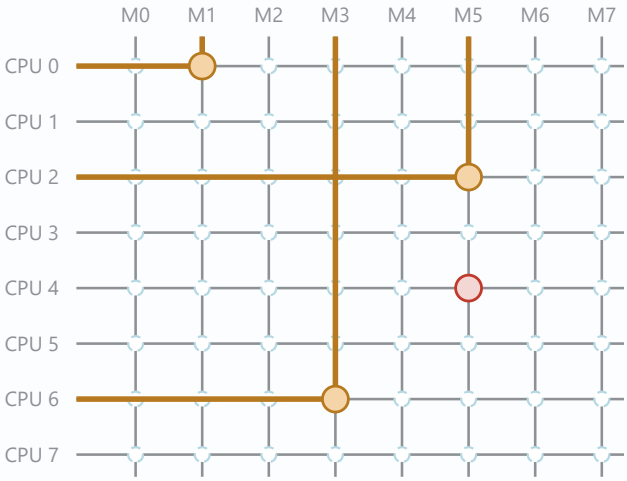
1.024 → 1.048.576

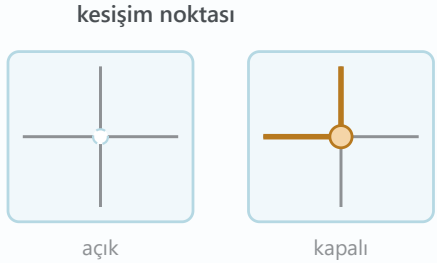
aynı anda 3 aktarım

CPU 0 → M1 ve CPU 6 → M3 aynı anda kurulur. Çapraz çubuk **engellemesiz** (non-blocking): farklı modüllere giden istekler birbirini beklemez.

Örnek: eski Sun Fire E25K 18 × 18 çapraz çubuk kullanıyordu; bugün yonga içinde çekirdekler ile L3 dilimleri arasında küçük çapraz çubuklar yaygındır.

Çapraz Çubuk (Crossbar) Anahtar





n × n anahtar → n² kesişim

8 → 64 · 64 → 4.096

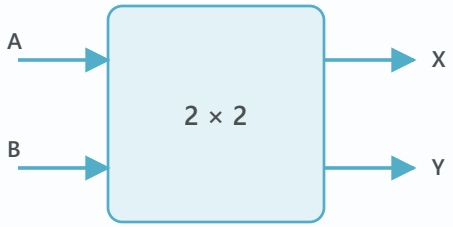
1.024 → 1.048.576

M5 meşgul → CPU 4 bekler

CPU 4 de M5'i ister: ağda yol var ama **modül meşgul**; tek çekişme bellek modülündedir. Bedel: kesişim sayısı n^2 büyür.

Örnek: eski Sun Fire E25K 18 × 18 çapraz çubuk kullanıyordu; bugün yonga içinde çekirdekler ile L3 dilimleri arasında küçük çapraz çubuklar yaygındır.

Omega Ağında Yönlendirme



her giriş her çıkışa bağlanabilir
bit = 0 → üst çıkış (X)
bit = 1 → alt çıkış (Y)

ileti biçimi



Modül: hangi bellek modülü (yönlendirme etiketi)
Adres: modül içindeki sözcük
İşlem kodu: READ / WRITE
Değer: yazılacak sözcük (isteğe bağlı)

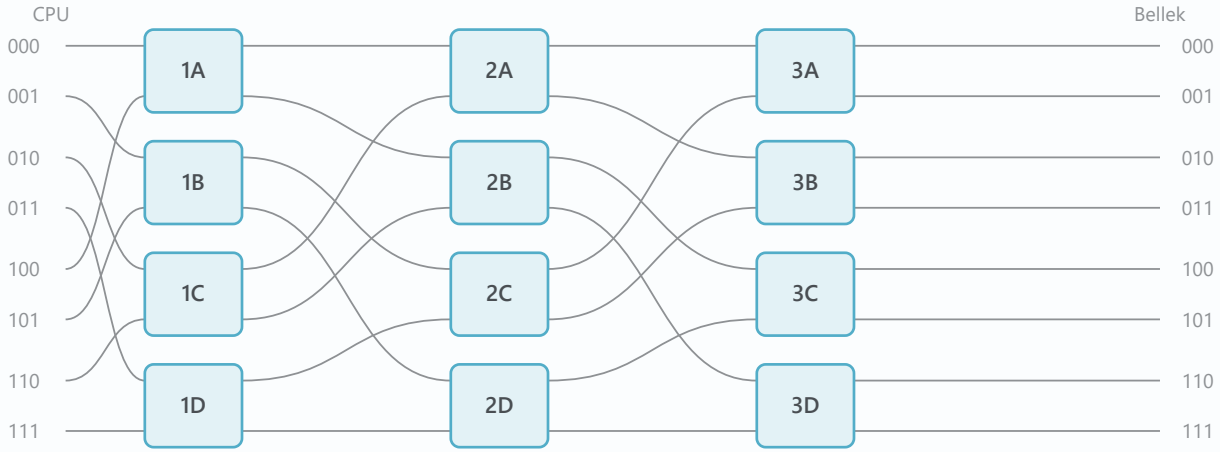
n CPU · n bellek → $\log_2 n$ aşama × n/2 anahtar
n = 8: 3 × 4 = 12 anahtar (çapraz çubukta 64 kesişim)
n = 1.024: 10 × 512 = 5.120 anahtar (1.048.576)

Yapı taşı **2 × 2 anahtar**: iki giriş, iki çıkış. İletinin **Modül** alanı hedef bellek modülünü taşır; anahtarlar yalnız ona bakar.

Adım 1 / 5

Omega ağı “senkron iletişim” sağlamaz; birçok aktarımı *eş zamanlı* taşıyabilir ama çakışmada bekletir; anahtar sayısı $(n/2)\cdot\log_2 n$ olduğu için çapraz çubuktan **çok daha az** donanım ve güç ister.

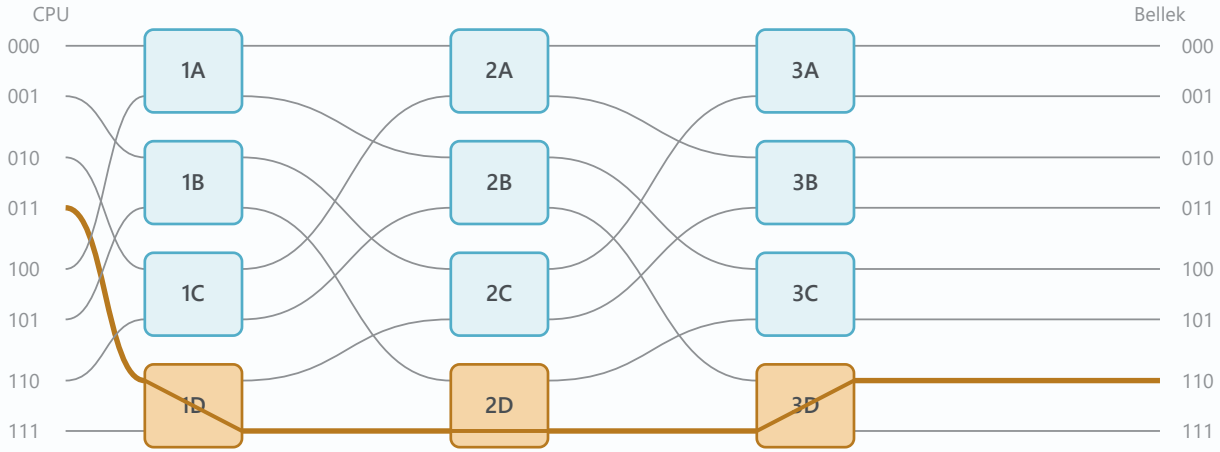
Omega Ağında Yönlendirme



8 CPU, 8 bellek, **3 aşama × 4 anahtar**. Aşamalar arasındaki kablolama **mükemmel karıştırma**dır (adres bitleri bir sola döner).

Omega ağı “senkron iletişim” sağlamaz; birçok aktarımı *eş zamanlı* taşıyabilir ama çakışmada bekletir; anahtar sayısı $(n/2)\cdot\log_2 n$ olduğu için çapraz çubuktan **çok daha az** donanım ve güç ister.

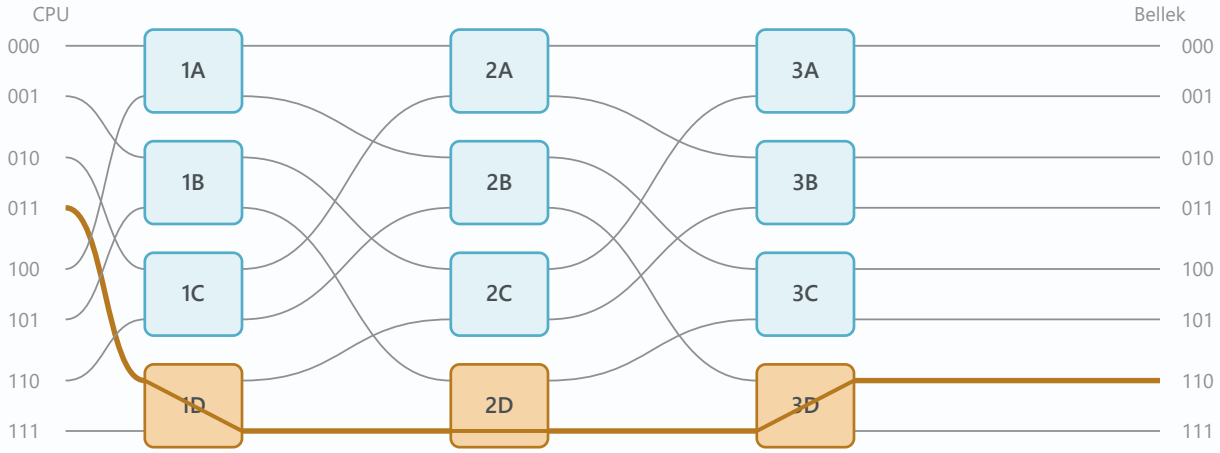
Omega Ağında Yönlendirme



CPU 011, bellek **110**'u okumak ister. 1. aşama (1D) hedefin 1. bitine bakar: **1** → alt çıkış; 2D ikinci bit **1** → alt; 3D üçüncü bit **0** → üst. Varış: 110.

Omega ağı "senkron iletişim" sağlamaz; birçok aktarımı *eş zamanlı* taşıyabilir ama çakışmada bekletir; anahtar sayısı $(n/2)\cdot\log_2 n$ olduğu için çapraz çubuktan **çok daha az** donanım ve güç ister.

Omega Ağında Yönlendirme

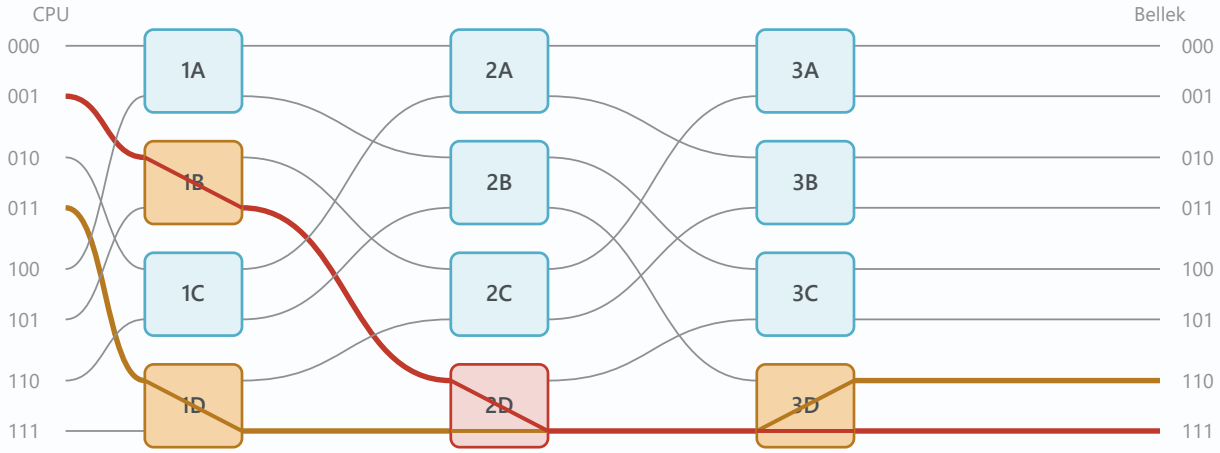


yanıt: Modül alanına kaynak (011) yazılır, yol tersten izlenir

Yanıt aynı ağdan döner: her anahtar geçişte kaynak bitini Modül alanına yazar; bellek bu etiketle yanıtı 011'e gönderir.

Omega ağı "senkron iletişim" sağlamaz; birçok aktarımı *eş zamanlı* taşıyabilir ama çakışmada bekletir; anahtar sayısı $(n/2)\cdot\log_2 n$ olduğu için çapraz çubuktan **çok daha az** donanım ve güç ister.

Omega Ağında Yönlendirme

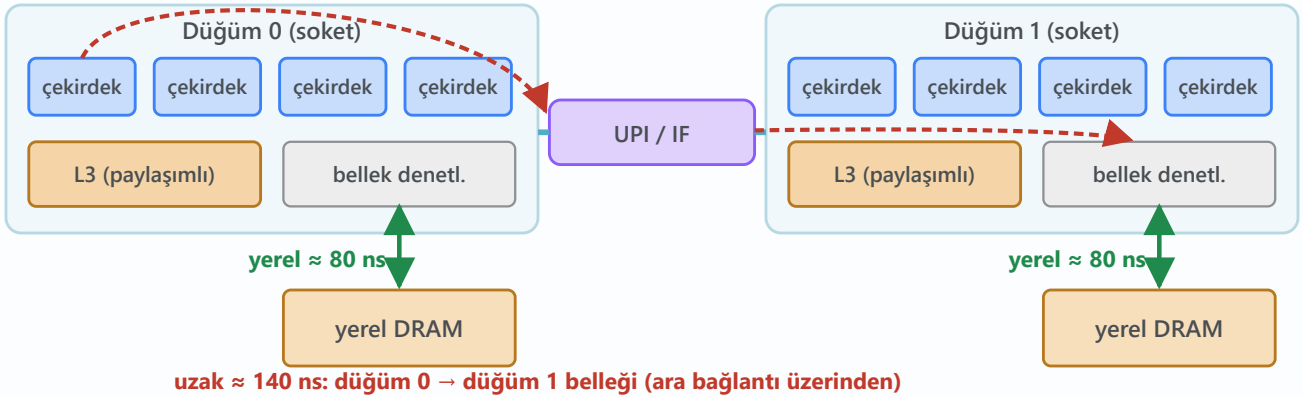


çakışma: 2D'nin alt çıkışını iki istek ister

Aynı anda CPU 001 bellek **111**'e gitmek ister; iki istek de **2D**'nin alt çıkışını kullanır → biri bekler. Omega **engellemeli** (blocking) bir ağdır.

Omega ağı "senkron iletişim" sağlamaz; birçok aktarımı *eş zamanlı* taşıyabilir ama çakışmada bekletir; anahtar sayısı $(n/2)\cdot\log_2 n$ olduğu için çapraz çubuktan **çok daha az** donanım ve güç ister.

NUMA Çok İşlemcileri

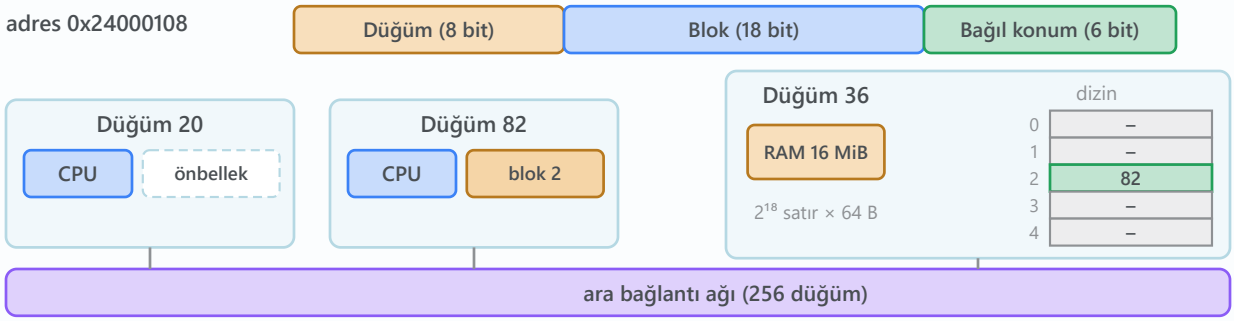


- **NUMA** (*Non-Uniform* Memory Access): tüm CPU'lar **tek adres alanı** görür; uzak belleğe de sıradan LOAD/STORE ile erişilir.
- Uzak erişim yerelden yavaştır. **NC-NUMA**: önbellek tutarlılığı yok; **CC-NUMA**: dizin tabanlı tutarlılık (bugünkü sunucuların hepsi).
- İşletim sistemi: süreci ve sayfalarını **aynı düğümde** tutar (ilk dokunma ilkesi, Linux otomatik NUMA dengeleme, `numactl`).

Ortalama erişim = $y \cdot t_{yerel} + (1 - y) \cdot t_{uzak}$
$y = 0{,}9 \rightarrow 0{,}9 \cdot 80 + 0{,}1 \cdot 140 =$ **86 ns**
$y = 0{,}5 \rightarrow$ **110 ns** (%28 daha yavaş)
gecikmeler örnek değerdir

Terim *Non-Uniform* Memory Access'tir. Tek soketli yongalar da içeride NUMA olabilir (AMD EPYC'de NPS ayarı, Intel SNC).

Dizin Tabanlı CC-NUMA



32 bit adres = 8 bit düğüm + 18 bit blok + 6 bit bağıl konum

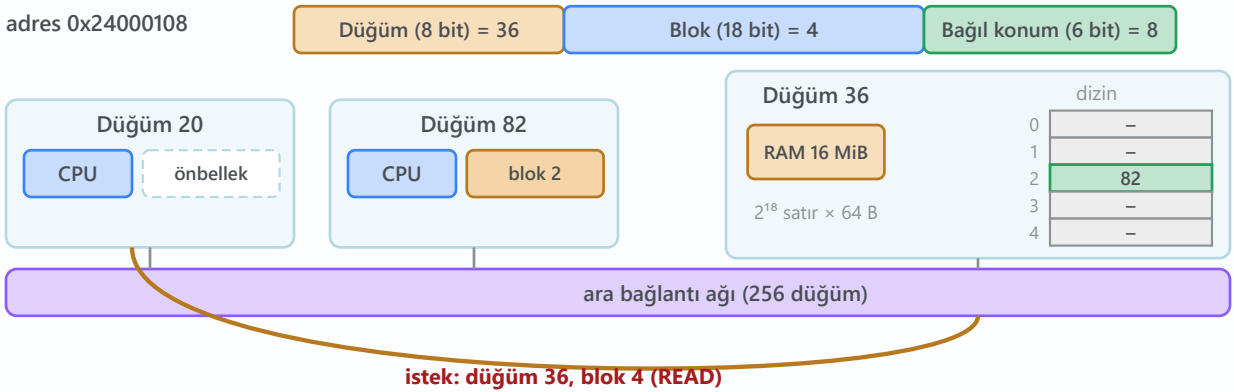
256 düğüm × 16 MiB = 4 GiB · her düğüm kendi bloklarının dizinini tutar

Tanenbaum'un örneği: 256 düğüm, düğüm başına 16 MiB, 64 baytlık satırlar. Her düğüm **kendi** belleğindeki satırların dizinini tutar.

Adım 1 / 6

Bu basit tasarımda bir satır tek düğümde önbelleklenebilir. Gerçek sistemler kopya kümesini bit vektörü ya da sınırlı işaretçi listesiyle tutar; çok çekirdekli yongalarda "snoop filter" aynı işi görür.

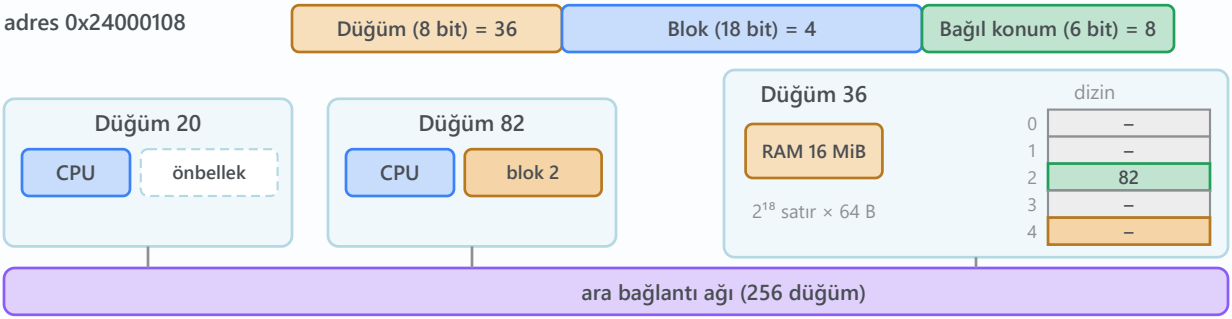
Dizin Tabanlı CC-NUMA



Düğüm 20'deki CPU `LOAD 0x24000108` yürütür: düğüm = 0x24 = **36**, blok = **4**, bağıl konum = **8**. Yerel değil → istek 36'ya gider.

Bu basit tasarımda bir satır tek düğümde önbelleklenebilir. Gerçek sistemler kopya kümesini bit vektörü ya da sınırlı işaretçi listesiyle tutar; çok çekirdekli yongalarda "snoop filter" aynı işi görür.

Dizin Tabanlı CC-NUMA

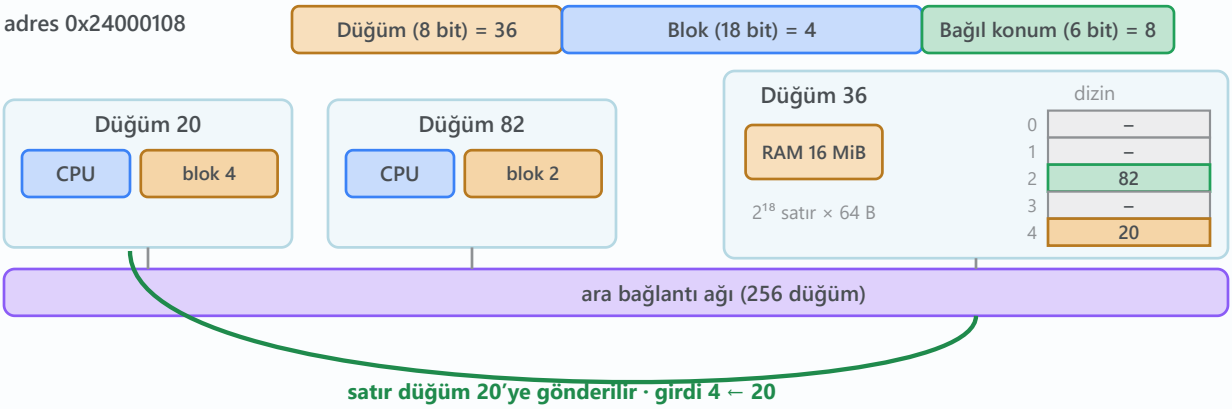


dizin girdisi 4 boş → satır hiçbir önbellekte yok

Düğüm 36 dizinine bakar: girdi 4 boş, satır hiçbir önbellekte değil. Güncel kopya kendi RAM'indedir.

Bu basit tasarımda bir satır tek düğümde önbelleklenebilir. Gerçek sistemler kopya kümesini bit vektörü ya da sınırlı işaretçi listesiyle tutar; çok çekirdekli yongalarda "snoop filter" aynı işi görür.

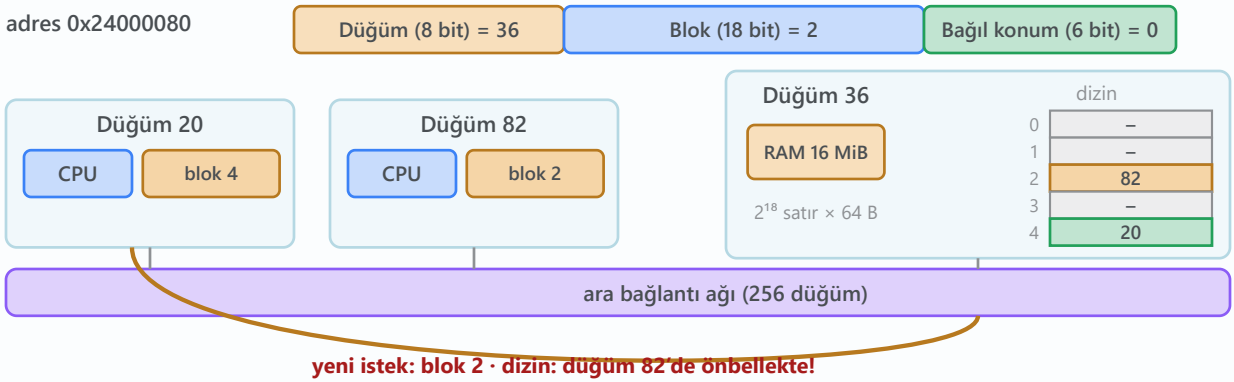
Dizin Tabanlı CC-NUMA



Satır düğüm 20'ye gönderilir ve dizin girdisi 4'e **20** yazılır: "bu satırın kopyası 20'de".

Bu basit tasarımda bir satır tek düğümde önbelleklenebilir. Gerçek sistemler kopya kümesini bit vektörü ya da sınırlı işaretçi listesiyle tutar; çok çekirdekli yongalarda "snoop filter" aynı işi görür.

Dizin Tabanlı CC-NUMA

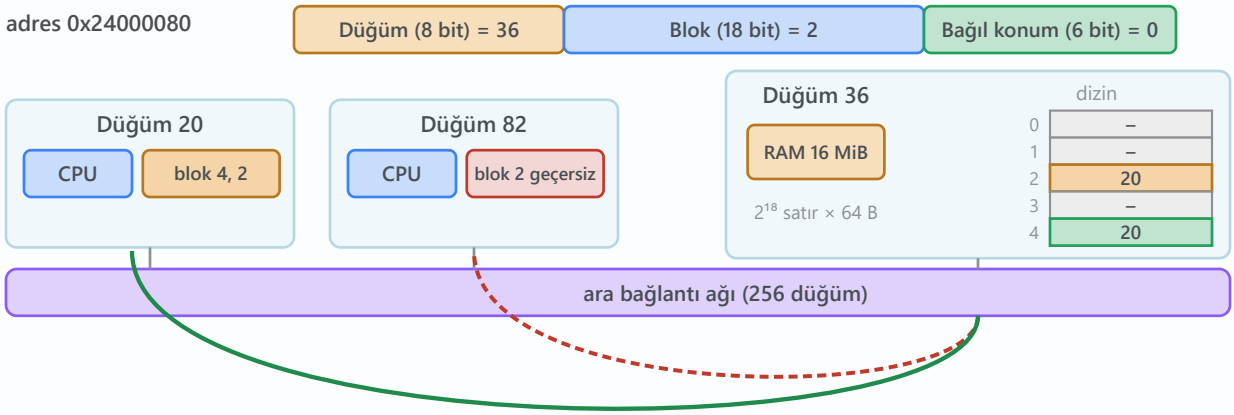


Şimdi düğüm 20, blok **2**'yi ister (0x24000080). Dizin: satır **düğüm 82**'de önbelleklenmiş.

Adım 5 / 6

Bu basit tasarımda bir satır tek düğümde önbelleklenebilir. Gerçek sistemler kopya kümesini bit vektörü ya da sınırlı işaretçi listesiyle tutar; çok çekirdekli yongalarda "snoop filter" aynı işi görür.

Dizin Tabanlı CC-NUMA

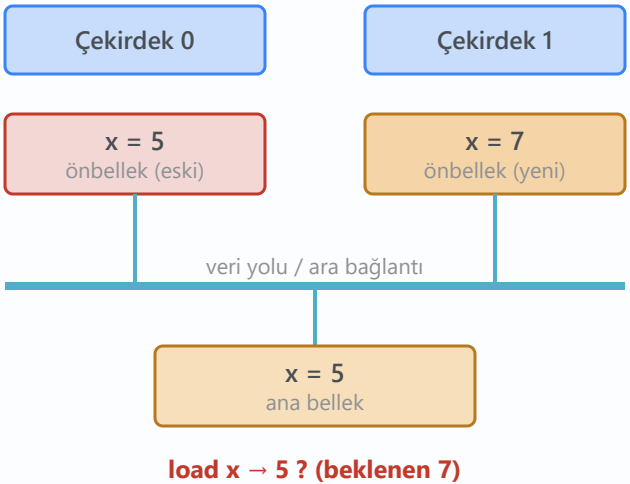


82'ye geçersiz kıl → sonra satırı 20'ye gönder · girdi 2 ← 20

Düğüm 36, 82'ye "geçersiz kıl" (gerekirse geri yaz) gönderir, sonra satırı 20'ye yollar; girdi 2 ← 20. Veri yolu dinlemesi (snooping) yerine **noktadan noktaya** iletiler.

Bu basit tasarımda bir satır tek düğümde önbelleklenebilir. Gerçek sistemler kopya kümesini bit vektörü ya da sınırlı işaretçi listesiyle tutar; çok çekirdekli yongalarda "snoop filter" aynı işi görür.

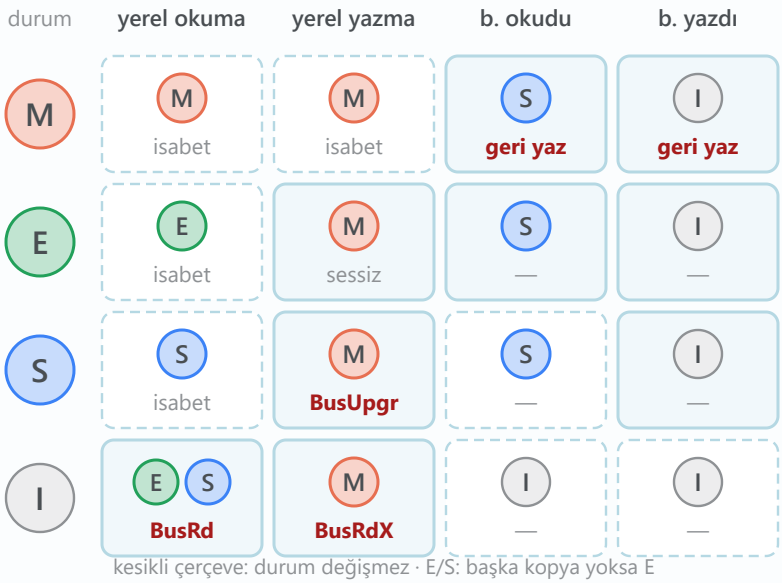
Önbellek Tutarlılığı Sorunu



- Her çekirdeğin özel önbelleği aynı belleğin **kopyalarını** tutar. Biri yazarsa diğerinin kopyası eskir.
- **Tutarlılık** (coherence): aynı adrese yapılan yazmalar tüm çekirdeklerce aynı sırada görülür; okuma en son yazılan değeri döndürür.
- **Dinleme** (snooping): tüm önbellek denetleyicileri ortak yolu dinler; az çekirdekte uygun.
- **Dizin** (directory): kopyaların listesi tutulur, yalnız ilgili düğümlere ileti gider; çok çekirdek ve NUMA için ölçeklenir.
- İki temel politika: **yazınca geçersiz kıl** (write-invalidate, yaygın) ve **yazınca güncelle** (write-update).

Tutarlılık tek bir adresle ilgilidir; farklı adreslerin sırası *bellek tutarlılık modeli*nin (x86-TSO, ARM zayıf model) konusudur; bu yüzden kilitler bellek bariyerleri içerir.

MESI Protokolü



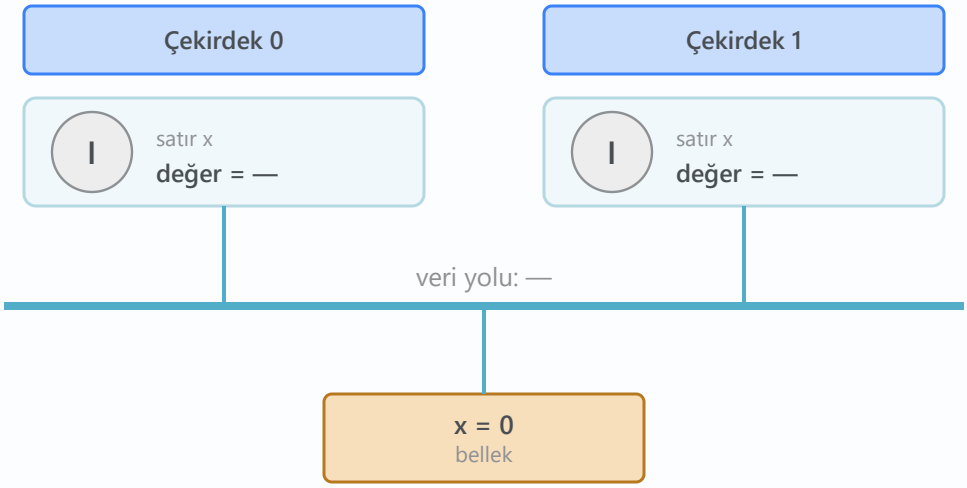
Durum	Anlamı
M Modified	Tek kopya, **kirli**; bellek eski. Atılırken geri yazılır.
E Exclusive	Tek kopya, temiz. Yazma veri yoluna çıkmadan M olur.
S Shared	Birden çok temiz kopya olabilir. Yazmak için önce diğerleri geçersiz kılınır.
I Invalid	Kopya geçersiz; erişim önbellek kaçırmasıdır.

Intel MESIF (F: yanıtı veren kopya), AMD MOESI (O: kirli ama paylaşılan) bu protokolün türevleridir.

MESI Adım Adım

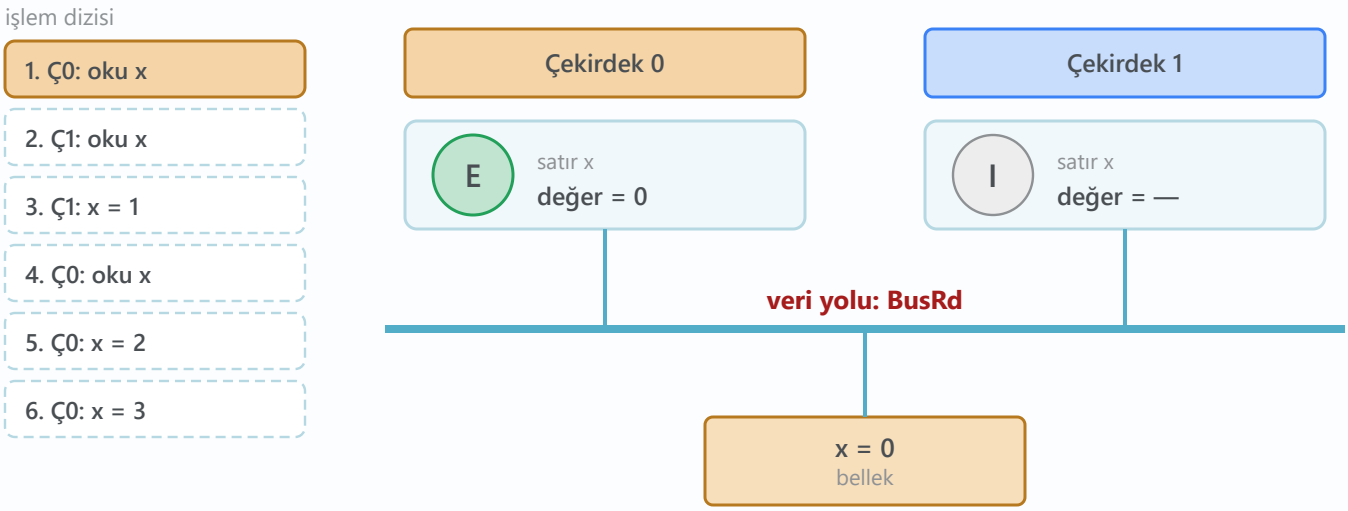
işlem dizisi

1. Ç0: oku x
2. Ç1: oku x
3. Ç1: x = 1
4. Ç0: oku x
5. Ç0: x = 2
6. Ç0: x = 3



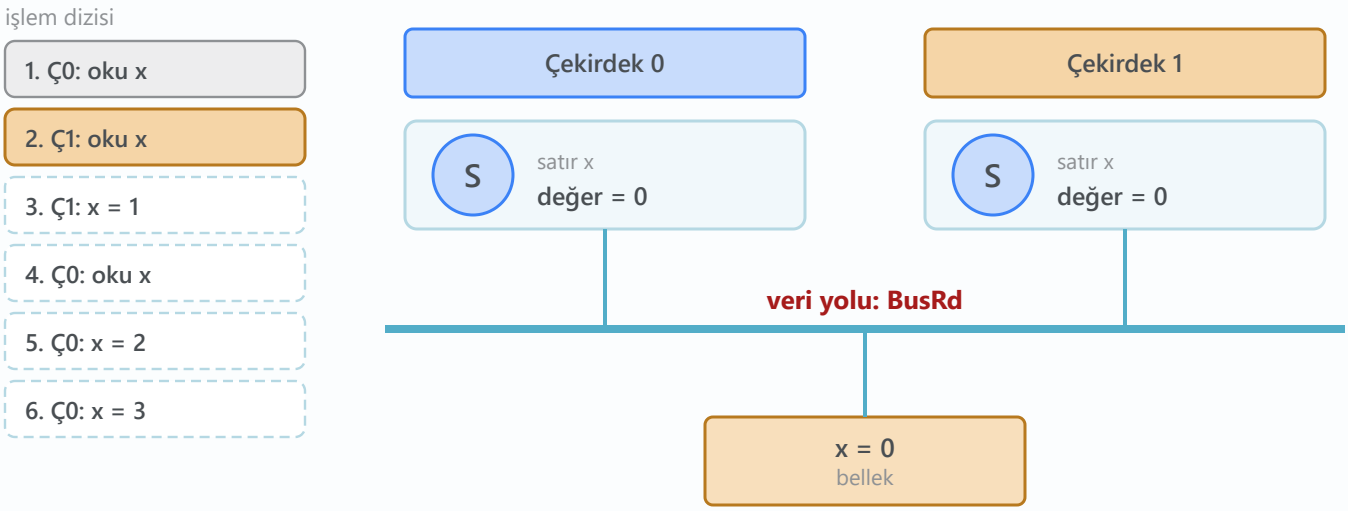
Başlangıç: x yalnız bellekte (0). İki önbellekte de satır **I**.

MESI Adım Adım



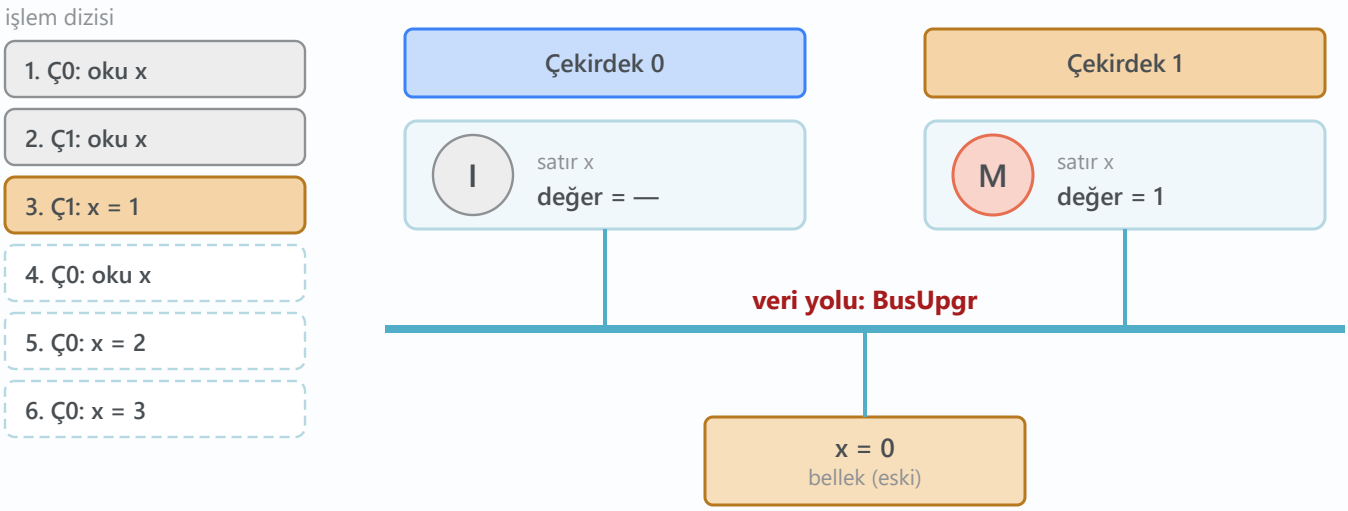
Ç0 okur: BusRd, başka kopya yok → **E** (tek ve temiz).

MESI Adım Adım



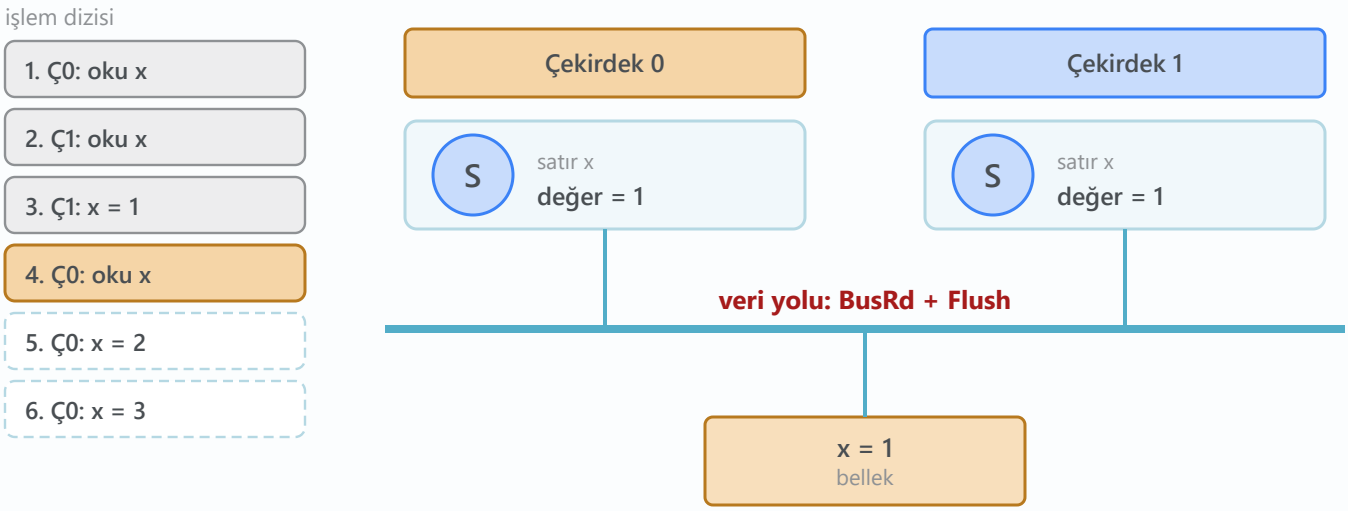
Ç1 okur: Ç0 isteği dinler, “bende var” der → ikisi de **S**. Bellek hâlâ güncel.

MESI Adım Adım



Ç1 yazar: S durumunda yazmak için **BusUpgr** (geçersiz kıl) yayınlanır → Ç0 **I**, Ç1 **M**; bellek artık eski.

MESI Adım Adım

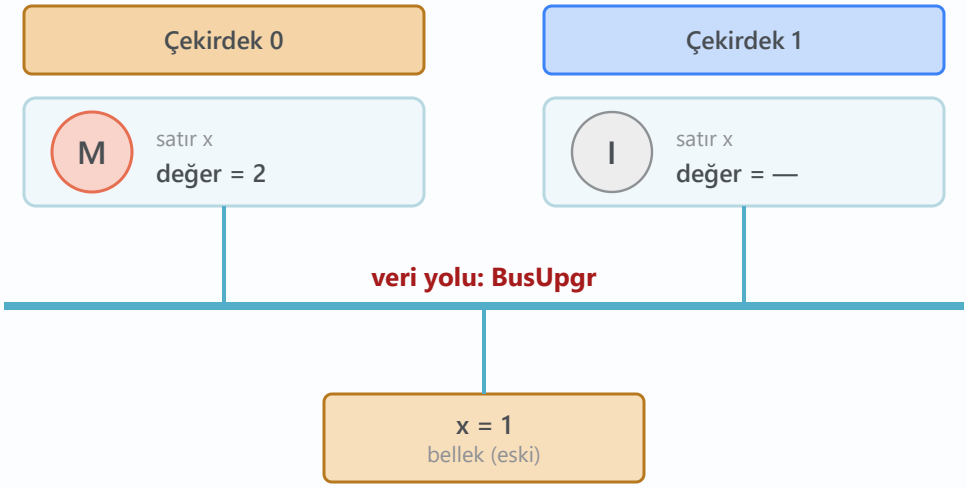


Ç0 okur: kaçırma. Ç1 kirli satırı **Flush** eder (veri Ç0'a ve belleğe), ikisi de **S**, bellek = 1.

MESI Adım Adım

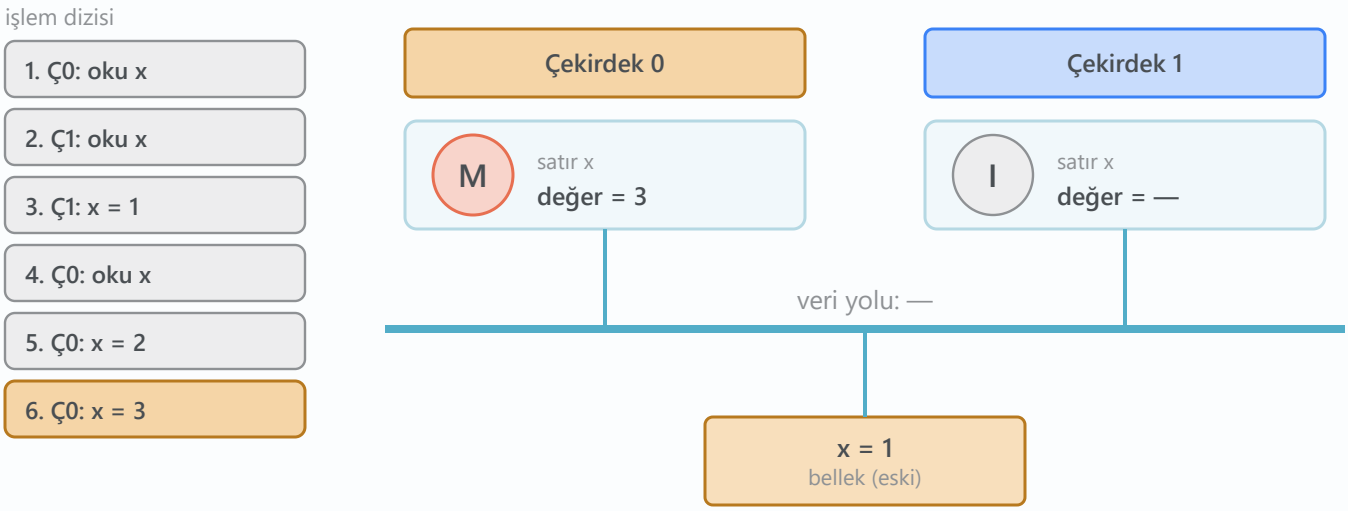
işlem dizisi

1. Ç0: oku x
2. Ç1: oku x
3. Ç1: x = 1
4. Ç0: oku x
5. Ç0: x = 2
6. Ç0: x = 3



Ç0 yazar: yine geçersiz kılma → Ç0 **M**, Ç1 **I**.

MESI Adım Adım



Ç0 tekrar yazar: zaten M, **veri yolu trafiği yok**. Yazma ağırlıklı özel veri bu yüzden ucuzdur.

Yanlış Paylaşım (False Sharing)

```
struct sayaclar {      /* kötü */
  long a;   /* bağıl konum 0 */
  long b;   /* bağıl konum 8 */
};

struct dolgulu {       /* iyi */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* iş parçacığı 0: for(..) s.a++;
   iş parçacığı 1: for(..) s.b++; */
```

- Tutarlılık birimi **satırdır** (64 B), değişken değil.
- Java: `@Contended` ; Linux: `____cacheline_aligned` .
- DSM'de aynı sorun **sayfa** boyutunda olur: bağımsız iki değişken aynı sayfadaysa sayfa makineler arasında gidip gelir.



mantıkta paylaşım yok, donanımda var

`a` ve `b` bağımsız sayaçlar ama **aynı satırda**.

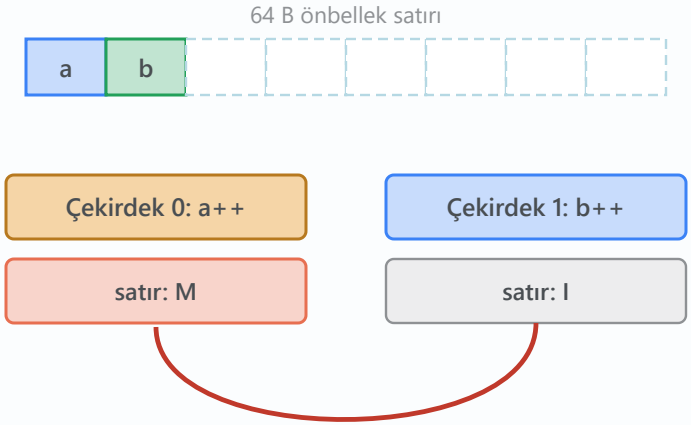
Yanlış Paylaşım (False Sharing)

```
struct sayaclar {      /* kötü */
  long a;   /* bağıl konum 0 */
  long b;   /* bağıl konum 8 */
};

struct dolgulu {       /* iyi */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* iş parçacığı 0: for(..) s.a++;
   iş parçacığı 1: for(..) s.b++; */
```

- Tutarlılık birimi **satırdır** (64 B), değişken değil.
- Java: `@Contended` ; Linux: `____cacheline_aligned` .
- DSM'de aynı sorun **sayfa** boyutunda olur: bağımsız iki değişken aynı sayfadaysa sayfa makineler arasında gidip gelir.



satır karşıya taşınır: ~50–150 ns

Ç0 `a++` : satırı M olarak alır, Ç1'in kopyası geçersiz.

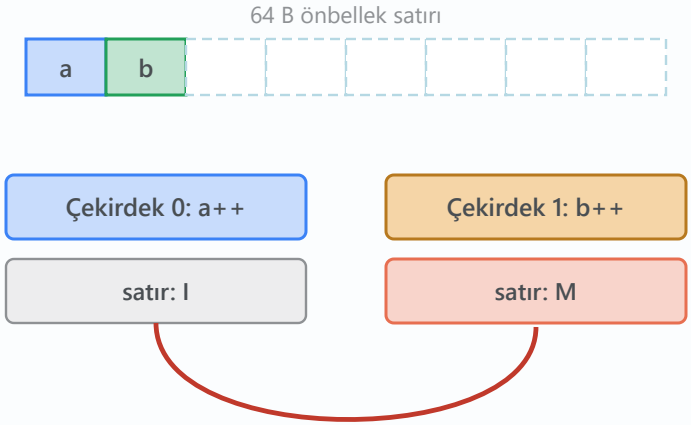
Yanlış Paylaşım (False Sharing)

```
struct sayaclar {      /* kötü */
  long a;   /* bağıl konum 0 */
  long b;   /* bağıl konum 8 */
};

struct dolgulu {       /* iyi */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* iş parçacığı 0: for(..) s.a++;
   iş parçacığı 1: for(..) s.b++; */
```

- Tutarlılık birimi **satırdır** (64 B), değişken değil.
- Java: `@Contended` ; Linux: `____cacheline_aligned` .
- DSM'de aynı sorun **sayfa** boyutunda olur: bağımsız iki değişken aynı sayfadaysa sayfa makineler arasında gidip gelir.



satır karşıya taşınır: ~50–150 ns

Ç1 `b++` : satır Ç1'e taşınır (M), Ç0'daki geçersiz olur.

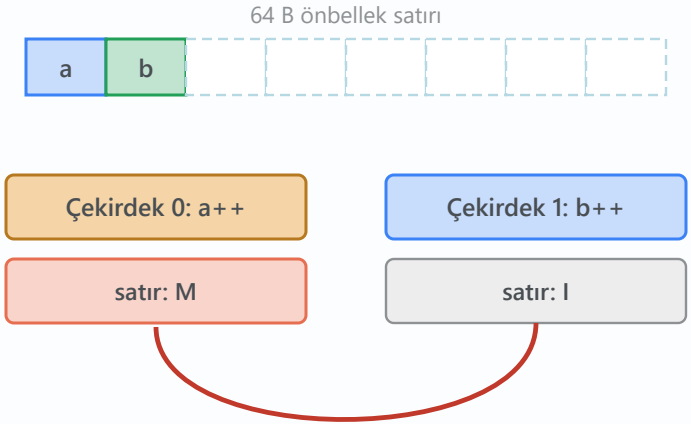
Yanlış Paylaşım (False Sharing)

```
struct sayaclar {      /* kötü */
  long a;   /* bağıl konum 0 */
  long b;   /* bağıl konum 8 */
};

struct dolgulu {       /* iyi */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* iş parçacığı 0: for(..) s.a++;
   iş parçacığı 1: for(..) s.b++; */
```

- Tutarlılık birimi **satırdır** (64 B), değişken değil.
- Java: `@Contended` ; Linux: `____cacheline_aligned` .
- DSM'de aynı sorun **sayfa** boyutunda olur: bağımsız iki değişken aynı sayfadaysa sayfa makineler arasında gidip gelir.



satır karşıya taşınır: ~50–150 ns

Her artırmada satır yeniden taşınır: "ping-pong"; iki iş parçacığı tek iş parçacığından yavaş çalışabilir.

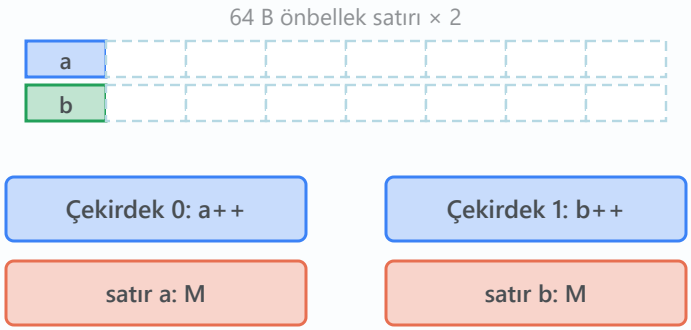
Yanlış Paylaşım (False Sharing)

```
struct sayaclar {      /* kötü */
  long a;   /* bağıl konum 0 */
  long b;   /* bağıl konum 8 */
};

struct dolgulu {       /* iyi */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* iş parçacığı 0: for(..) s.a++;
   iş parçacığı 1: for(..) s.b++; */
```

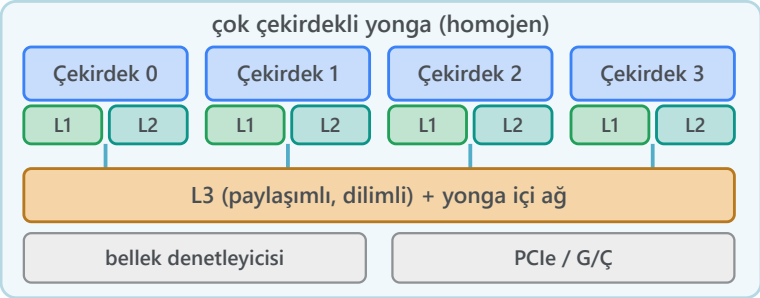
- Tutarlılık birimi **satırdır** (64 B), değişken değil.
- Java: `@Contended` ; Linux: `____cacheline_aligned` .
- DSM'de aynı sorun **sayfa** boyutunda olur: bağımsız iki değişken aynı sayfadaysa sayfa makineler arasında gidip gelir.

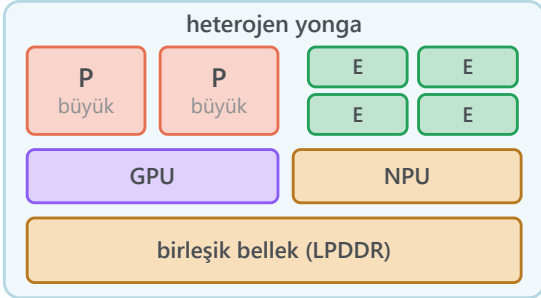


ayrı satırlar: ikisi de M kalır
tutarlılık trafiği yok

Çözüm: hizalama/dolgu ile her sayaç **ayrı satıra** konur. Bellek israfı sayaç başına 56 B, kazanç çoğu zaman kat kat hız.

Çok Çekirdekli ve Heterojen Yongalar





Çok çekirdek (CMP)

L1/L2 özel, L3 paylaşımlı. Sunucu soketi: 192 çekirdek (AMD EPYC 9005), 288 E-çekirdek (Xeon 6900E).

Yongacıklar (chiplet)

Çekirdek grupları ayrı kalıplarda; tek sokette bile L3 ve bellek uzaklığı farklıdır.

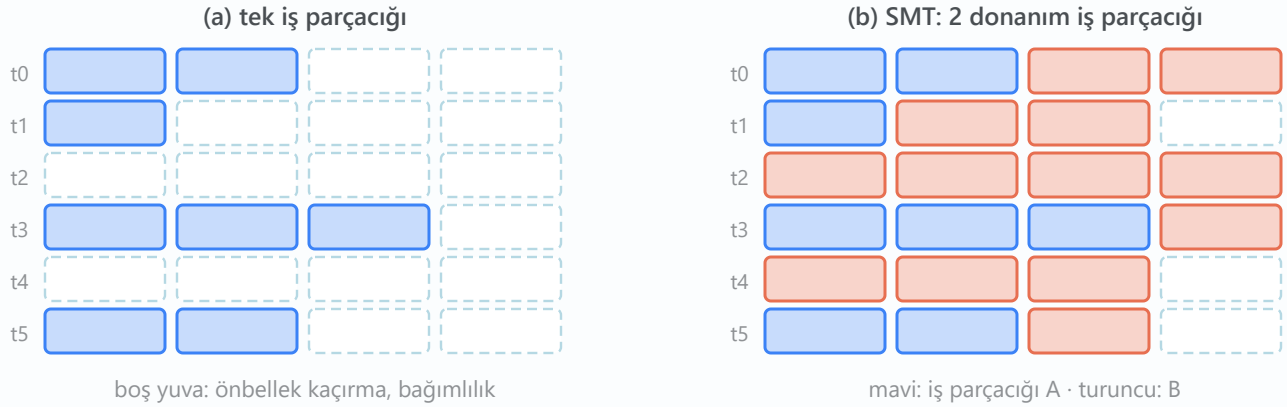
Büyük–küçük çekirdek

big.LITTLE, Apple ve Intel P/E: işletim sistemi işi enerjiye göre yerleştirir (Linux EAS, Thread Director).

Hızlandırıcılar

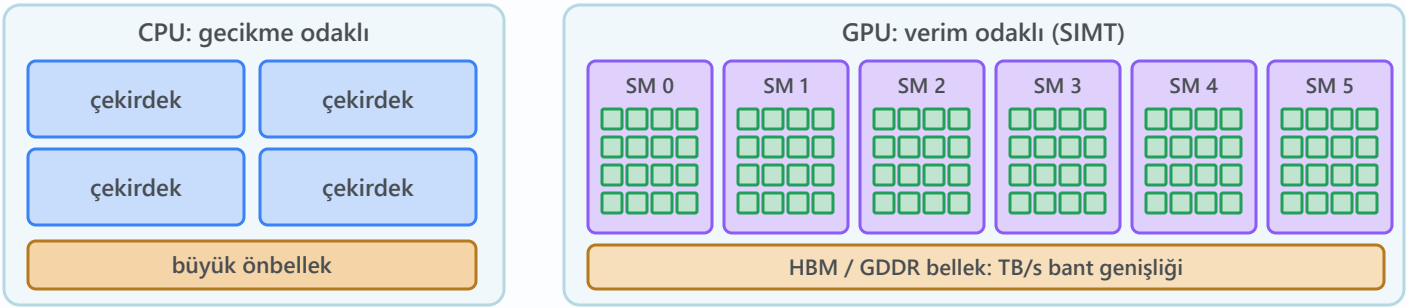
GPU ve NPU aynı yongada, aynı belleği paylaşır; sürücü üzerinden zamanlanır.

Eş Zamanlı Çoklu İş Parçacığı (SMT)



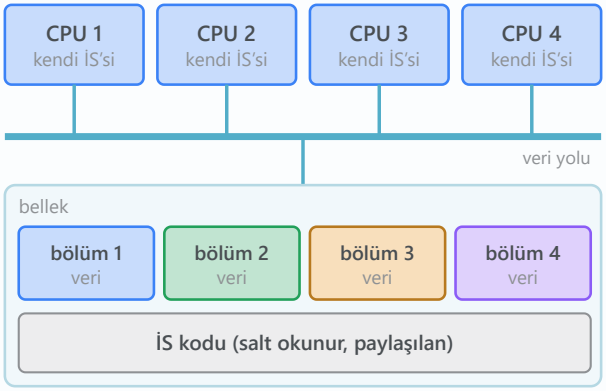
- Bir fiziksel çekirdek **iki mantıksal CPU** gibi görünür (Intel Hyper-Threading, AMD SMT); yazmaç kümeleri ikidir, yürütme birimleri ve L1/L2 ortaktır.
- Kazanç tipik olarak %10–30; iki iş parçacığı aynı birimi isterse hiç yoktur. Çizelgeleyici önce **boş fiziksel çekirdekleri** doldurmalıdır.
- Güvenlik: ortak önbellek yan kanal açar (L1TF, MDS) → Linux *core scheduling* yalnız güvenilen iş parçacıklarını aynı çekirdeğe koyar. Intel Lunar Lake (2024) P-çekirdeklerinde SMT'yi kaldırdı.

GPU ve Heterojen Hesaplama



- GPU binlerce basit şeridi **32'lik gruplar** (warp/wavefront) hâlinde aynı komutla yürütür: veri paralel işler (grafik, yapay zekâ, bilimsel hesap).
- Programlama: CUDA, ROCm/HIP, OpenCL, SYCL; CPU çekirdeği (kernel) başlatır, veri taşır, sonucu bekler.
- İşletim sistemi ve sürücü: GPU belleğini yönetir, bağlamları zaman paylaşımlı çalıştırır, birleşik/paylaşılan sanal bellek (IOMMU, sayfa hatası) sağlar.
- Kümelerde düğüm başına 4–8 GPU, aralarında NVLink; düğümler arası InfiniBand/RoCE ile RDMA.

Her CPU'nun Kendi İşletim Sistemi



- Bellek CPU sayısı kadar bölüme ayrılır; her CPU kendi bölümünde **bağımsız bir işletim sistemi kopyası** çalıştırır. İşletim sistemi kodu paylaşılabilir, verisi ayrıdır.
- Sistem çağrısı yerel CPU'da yakalanır ve işlenir.

Artıları

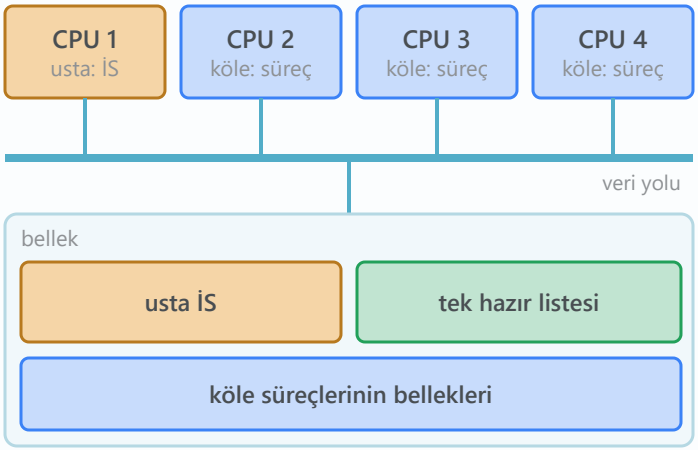
+ Basit: tek işlemcili işletim sistemi neredeyse değişmeden kullanılır.
+ Bölümler arası kilit gerekmez.

Eksileri

– Süreçler paylaşılmaz: bir CPU aşırı yüklüyken diğeri boşta.
– Sayfalar paylaşılmaz; her işletim sistemi kendi tampon önbelleğini tutar → tutarsız disk blokları.

Bugünkü karşılığı: bölümlenmiş (partitioned) sistemler ve hipervizör üzerindeki ayrı sanal makineler; araştırmada Barrelfish "multikernel" her çekirdeği iletiyle haberleşen ayrı bir düğüm gibi ele alır.

Usta-Köle (Master-Slave) Modeli



- İşletim sisteminin tek kopyası ve tabloları **yalnız usta CPU**'dadır; tüm sistem çağrıları ustaya yönlendirilir.
- Köleler ustanın verdiği kullanıcı süreçlerini çalıştırır; tek hazır listesi olduğu için yük dengelidir.

Artıları

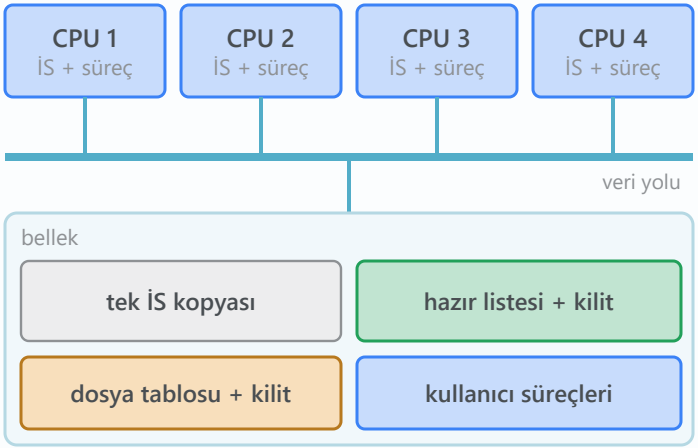
+ Basit, tutarlı: çekirdek verisine tek CPU dokunur.
+ Yük dengeleme kendiliğinden.

Eksileri

– Usta **darboğaz** olur: sistem çağrısı yoğun işlerde birkaç CPU'dan sonra ölçeklenmez.
– Usta çökerse tüm sistem durur.

Asimetrik çok işlemcili (AMP) düzenin klasik biçimi budur. Bugün gömülü yongalarda asimetrik düzen yaşar: Linux çalıştıran Cortex-A çekirdeklerin yanında RTOS çalıştıran bir Cortex-M çekirdeği (OpenAMP, remoteproc).

Simetrik Çok İşlemciler (SMP)



- Bellekte **tek işletim sistemi kopyası**; her CPU çekirdek kodunu çalıştırabilir. Sistem çağrısı onu yapan CPU'da işlenir.
- Sorun: iki CPU aynı anda aynı çekirdek tablosuna dokunabilir → **kilitler**.
- İlk adım tek büyük çekirdek kilidi (Linux BKL, 2.0); 2.6.39'da (2011) tamamen kaldırıldı. Bugün binlerce ince taneli kilit, RCU ve CPU başına veri.

Artıları

+ Darboğaz yok, tüm CPU'lar çekirdeği çalıştırır.
+ Bellek ve süreçler dinamik dengelenir.

Eksileri

– Çekirdek tabloları kritik bölgelere bölünmeli.
– Kilit sırası hatası → kilitlenme.

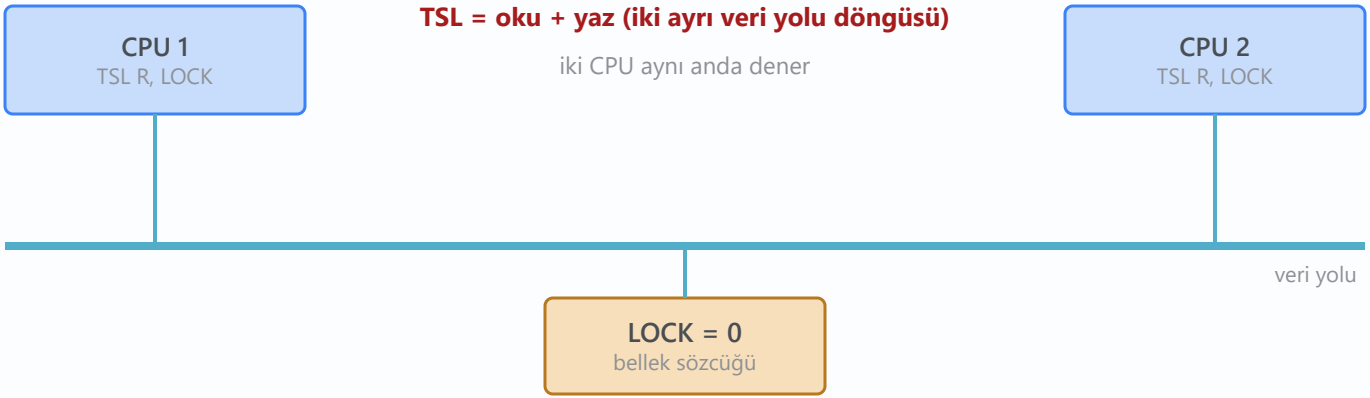
Üç Modelin Karşılaştırması

	Ayrı İS	Usta-köle	SMP
İS kopyası	CPU başına bir	bir (ustada)	bir (herkes çalıştırır)
Sistem çağrısı nerede?	yerel CPU	usta CPU	yerel CPU
Hazır listesi	CPU başına	tek, ustada	tek ya da CPU başına + dengeleme
Yük dengeleme	yok	iyi	iyi
Ölçeklenebilirlik	iyi ama paylaşım yok	usta darboğaz	kilit tasarımına bağlı, en iyi
Karmaşıklık	en düşük	düşük	en yüksek
Günümüz	sanal makineler, bölümleme	gömülü AMP, eski sistemler	Linux, Windows, macOS, BSD



Çok işlemcili işletim sistemi ile çoklu bilgisayar işletim sistemi farklıdır: çok işlemcide tüm CPU'lar **aynı fiziksel belleği** görür; çoklu bilgisayarda her düğümün kendi belleği vardır ve iletişim yalnız **iletiyle** olur.

TSL ve Veri Yolu Kilidi



Tek işlemcide TSL (test and set lock) bölünmez bir komuttur. Çok işlemcide bellek sözcüğünü **okuma** ve **yazma** ayrı veri yolu döngüleridir.

TSL ve Veri Yolu Kilidi



CPU 1 kilidi okur: 0 (serbest).

TSL ve Veri Yolu Kilidi



CPU 1 yazmadan önce CPU 2 de okur: yine 0.

TSL ve Veri Yolu Kilidi



CPU 1 1 yazar, CPU 2 de 1 yazar.

TSL ve Veri Yolu Kilidi



Sonuç: ikisi de “kilit benim” sanır, karşılıklı dışlama bozulur. Kesmeleri kapatmak da işe yaramaz: o yalnız **kendi** CPU’sunu durdurur.

TSL ve Veri Yolu Kilidi



Çözüm: TSL okuma–yazma süresince **veri yolunu kilitler**. Bugün x86 `LOCK` öneki çoğunlukla yalnız **önbellek satırını** M durumunda tutar; ARM `LDXR/STXR` ya da LSE `CAS` kullanır.

Bilet Kilidi (Ticket Lock)

```
1  import java.util.concurrent.atomic.*;
2  public class Bilet {
▶    static AtomicInteger next = new AtomicInteger();
▶    static volatile int serving = 0;
5    static int sayac = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
8      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable is = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); sayac++; unlock(); } };
16     Thread t0 = new Thread(is);
17     Thread t1 = new Thread(is);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("sayac = " + sayac);
20   }
21 }
```



İki sayaç: `next` bilet dağıtır, `serving` sırası geleni gösterir.

Adım 1 / 8

DEĞİŞKENLER

`next = 0` `serving = 0` `T0.my = –` `T1.my = –` `sayac = 0`

ÇIKTI

Bilet Kilidi (Ticket Lock)

```
1  import java.util.concurrent.atomic.*;
2  public class Bilet {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int sayac = 0;
6    static void lock() {
▶      int my = next.getAndIncrement();
8      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable is = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); sayac++; unlock(); } };
16     Thread t0 = new Thread(is);
17     Thread t1 = new Thread(is);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("sayac = " + sayac);
20   }
21 }
```



T0 bilet alır: `getAndIncrement` atomiktir (x86 `LOCK XADD`) → my = 0.

Adım 2 / 8

DEĞİŞKENLER

`next = 1` `serving = 0` `T0.my = 0` `T1.my = –` `sayac = 0`

ÇIKTI

Bilet Kilidi (Ticket Lock)

```
1  import java.util.concurrent.atomic.*;
2  public class Bilet {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int sayac = 0;
6    static void lock() {
▶      int my = next.getAndIncrement();
8      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable is = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); sayac++; unlock(); } };
16     Thread t0 = new Thread(is);
17     Thread t1 = new Thread(is);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("sayac = " + sayac);
20   }
21 }
```



T1 de bilet alır: my = 1. İki CPU aynı anda çağırsa da biletler farklıdır.

Adım 3 / 8

DEĞİŞKENLER

`next = 2` `serving = 0` `T0.my = 0` `T1.my = 1` `sayac = 0`

ÇIKTI

Bilet Kilidi (Ticket Lock)

```
1  import java.util.concurrent.atomic.*;
2  public class Bilet {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int sayac = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
▶      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable is = () -> {
14       for (int i = 0; i < 100_000; i++) {
▶          lock(); sayac++; unlock(); } };
16     Thread t0 = new Thread(is);
17     Thread t1 = new Thread(is);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("sayac = " + sayac);
20   }
21 }
```



T0: serving == 0 → döngüden çıkar, `sayac++` yapar.

Adım 4 / 8

DEĞİŞKENLER

`next = 2` `serving = 0` `T0.my = 0` `T1.my = 1` `sayac = 1`

ÇIKTI

Bilet Kilidi (Ticket Lock)

```
1  import java.util.concurrent.atomic.*;
2  public class Bilet {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int sayac = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
▶      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable is = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); sayac++; unlock(); } };
16     Thread t0 = new Thread(is);
17     Thread t1 = new Thread(is);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("sayac = " + sayac);
20   }
21 }
```



T1 döner: `serving` 'i kendi önbelleğindeki **S** kopyadan okur, veri yolunu meşgul etmez; `onSpinWait` = x86 `PAUSE`.

Adım 5 / 8

DEĞİŞKENLER

`next = 2` `serving = 0` `T0.my = 0` `T1.my = 1` `sayac = 1`

ÇIKTI

Bilet Kilidi (Ticket Lock)

```
1  import java.util.concurrent.atomic.*;
2  public class Bilet {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int sayac = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
8      while (serving != my) Thread.onSpinWait();
9    }
▶    static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable is = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); sayac++; unlock(); } };
16     Thread t0 = new Thread(is);
17     Thread t1 = new Thread(is);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("sayac = " + sayac);
20   }
21 }
```



T0 bırakır: `serving++`. Yazma T1'in kopyasını geçersiz kılar (tek tutarlılık iletisi).

Adım 6 / 8

DEĞİŞKENLER

`next = 2` `serving = 1` `T0.my = 0` `T1.my = 1` `sayac = 1`

ÇIKTI

Bilet Kilidi (Ticket Lock)

```
1  import java.util.concurrent.atomic.*;
2  public class Bilet {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int sayac = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
▶      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable is = () -> {
14       for (int i = 0; i < 100_000; i++) {
▶          lock(); sayac++; unlock(); } };
16     Thread t0 = new Thread(is);
17     Thread t1 = new Thread(is);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("sayac = " + sayac);
20   }
21 }
```



T1 yeni değeri okur: 1 == my → girer. Kilit **adil**dir: kimse aç kalmaz.

Adım 7 / 8

DEĞİŞKENLER

`next = 2` `serving = 1` `T0.my = 0` `T1.my = 1` `sayac = 2`

ÇIKTI

Bilet Kilidi (Ticket Lock)

```
1  import java.util.concurrent.atomic.*;
2  public class Bilet {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int sayac = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
8      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable is = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); sayac++; unlock(); } };
16     Thread t0 = new Thread(is);
17     Thread t1 = new Thread(is);
18     t0.start(); t1.start(); t0.join(); t1.join();
▶      System.out.println("sayac = " + sayac);
20   }
21 }
```



2 × 100.000 artırma, kayıp yok. Kilitsiz `sayac++` ile sonuç çoğu çalıştırmada 200.000'den küçük çıkar.

Adım 8 / 8

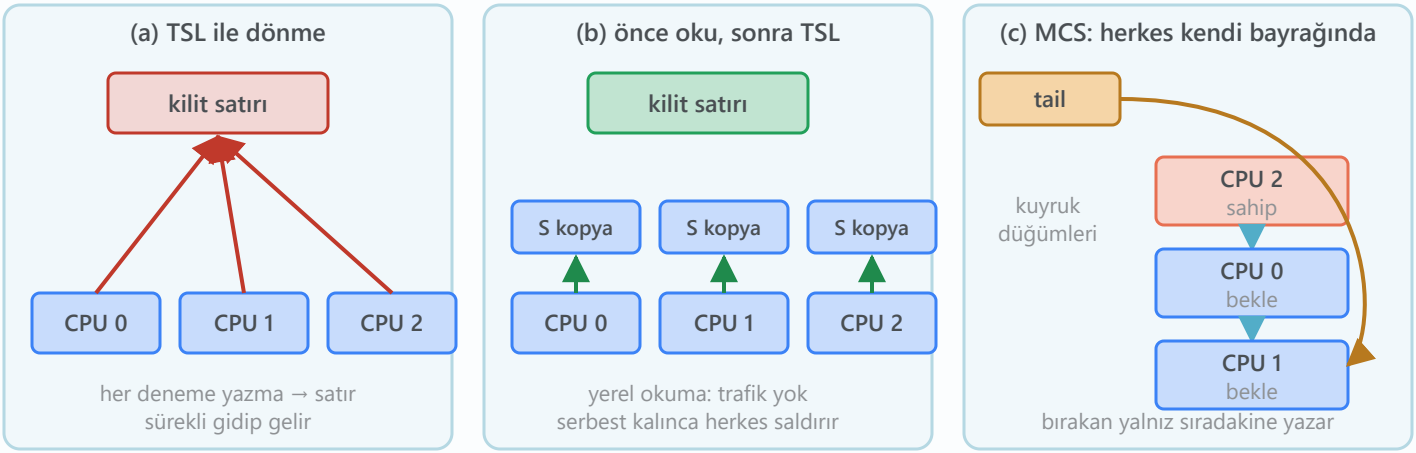
DEĞİŞKENLER

`next = 200000` `serving = 200000` `T0.my = –` `T1.my = –` `sayac = 200000`

ÇIKTI

```
sayac = 200000
```

Spinlock’lar ve Önbellek Trafiği

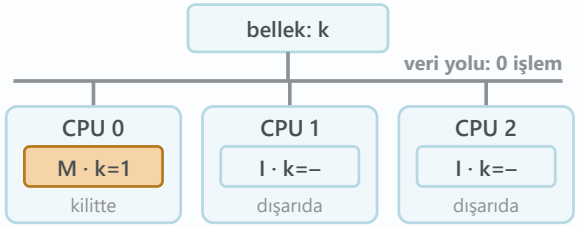


- (a) Saf TSL döngüsü her denemede **yazar**: kilit satırı CPU’lar arasında sürekli taşınır, sahibin işi de yavaşlar.
- (b) **Önce oku, sonra TSL** (TTAS): yerel kopya üzerinde dönülür; serbest görününce TSL denenir. Ek olarak **üstel geri çekilme** (backoff) çakışmayı azaltır.
- (c) **MCS** (Mellor-Crummey & Scott, 1991): her bekleyenin ayrı satırda kendi bayrağı; bırakma tek bir uzak yazmadır. Linux `qspinlock` bu fikre dayanır.

Çoklu kilidin amacı veri tutarsızlığını değil, tutarlılık **trafiğini** (cache thrashing) önlemektir.

TTAS Kilidi: Kod İzleme

```
▶ void kilitle(int *k) {
2   for (;;) {
3     while (*k == 1)   /* yerelde dön */
4       ;
5     if (tsl(k) == 0)  /* serbest göründü */
6       return;         /* kilit alındı */
7   }
8 }
9 void ac(int *k) {
10  *k = 0;
11 }
```



önbellek satırı durumu: M, E, S, I

CPU 0 kilidi tutuyor (k = 1, satır onun önbelleğinde **M**). CPU 1 ve CPU 2 `kilitle` çağırıyor.

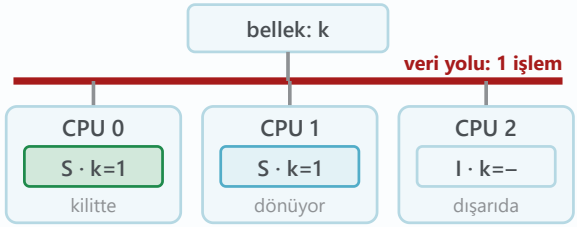
Adım 1 / 10

DEĞİŞKENLER

`k = 1` `CPU1 = I` `CPU2 = I` `veri yolu = 0`

TTAS Kilidi: Kod İzleme

```
1  void kilitle(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* yerelde dön */
4        ;
5      if (tsl(k) == 0)  /* serbest göründü */
6        return;         /* kilit alındı */
7    }
8  }
9  void ac(int *k) {
10   *k = 0;
11 }
```



önbellek satırı durumu: M, E, S, I

CPU 1 `*k`'yı okur: ıska, veri yolundan istenir. CPU 0'ın M kopyası paylaşılır; ikisi de **S**.

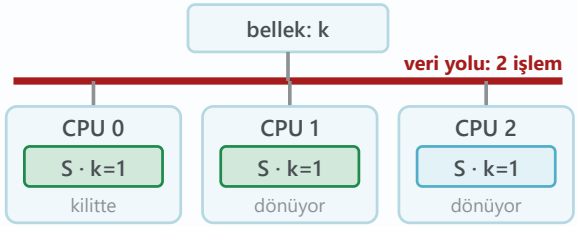
Adım 2 / 10

DEĞİŞKENLER

`k = 1` `CPU1 = S` `CPU2 = I` `veri yolu = 1`

TTAS Kilidi: Kod İzleme

```
1  void kilitle(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* yerelde dön */
4        ;
5      if (tsl(k) == 0)  /* serbest göründü */
6        return;         /* kilit alındı */
7    }
8  }
9  void ac(int *k) {
10   *k = 0;
11 }
```



önbellek satırı durumu: M, E, S, I

CPU 2 de okur: üç kopya S. Buraya kadar 2 veri yolu işlemi.

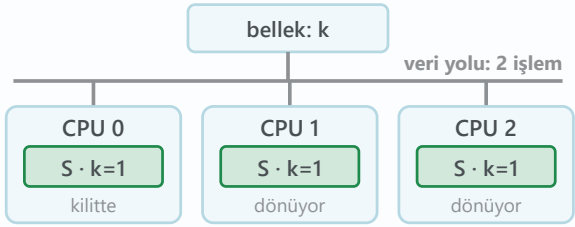
Adım 3 / 10

DEĞİŞKENLER

`k = 1` `CPU1 = S` `CPU2 = S` `veri yolu = 2`

TTAS Kilidi: Kod İzleme

```
1  void kilitle(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* yerelde dön */
▶        ;
5      if (tsl(k) == 0)  /* serbest göründü */
6        return;         /* kilit alındı */
7    }
8  }
9  void ac(int *k) {
10   *k = 0;
11 }
```



önbellek satırı durumu: M, E, S, I

İkisi de k = 1 gördükçe **kendi önbelleğinde** döner: veri yolu sayacı artmaz. Saf TSL döngüsü burada her denemede satırı yazmak için çekerdi.

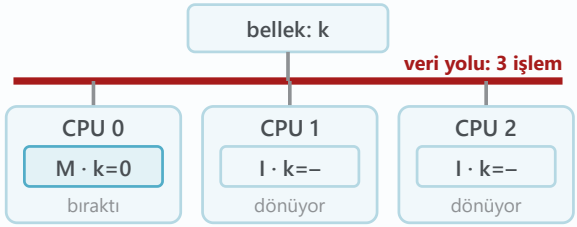
Adım 4 / 10

DEĞİŞKENLER

`k = 1` `CPU1 = S` `CPU2 = S` `veri yolu = 2`

TTAS Kilidi: Kod İzleme

```
1  void kilitle(int *k) {
2    for (;;) {
3      while (*k == 1)   /* yerelde dön */
4        ;
5      if (tsl(k) == 0)  /* serbest göründü */
6        return;         /* kilit alındı */
7    }
8  }
9  void ac(int *k) {
▶    *k = 0;
11 }
```



önbellek satırı durumu: M, E, S, I

CPU 0 `ac` : k = 0 yazmak için diğer kopyaları **geçersiz** kılar (I); satır CPU 0'da M.

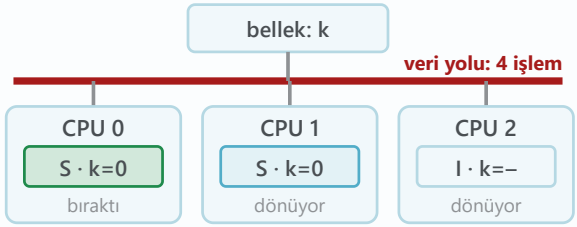
Adım 5 / 10

DEĞİŞKENLER

`k = 0` `CPU1 = I` `CPU2 = I` `veri yolu = 3`

TTAS Kilidi: Kod İzleme

```
1  void kilitle(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* yerelde dön */
4        ;
5      if (tsl(k) == 0)  /* serbest göründü */
6        return;         /* kilit alındı */
7    }
8  }
9  void ac(int *k) {
10   *k = 0;
11 }
```



önbellek satırı durumu: M, E, S, I

CPU 1'in kopyası geçersiz: yeniden okur (ıska), k = 0 görür ve while'dan çıkar.

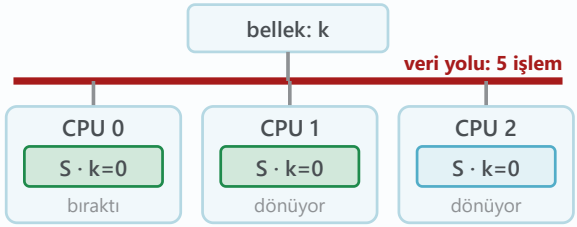
Adım 6 / 10

DEĞİŞKENLER

`k = 0` `CPU1 = S` `CPU2 = I` `veri yolu = 4`

TTAS Kilidi: Kod İzleme

```
1  void kilitle(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* yerelde dön */
4        ;
5      if (tsl(k) == 0)  /* serbest göründü */
6        return;         /* kilit alındı */
7    }
8  }
9  void ac(int *k) {
10   *k = 0;
11 }
```



önbellek satırı durumu: M, E, S, I

CPU 2 de ıska ile k = 0 görür. İkisi de aynı anda "serbest" sanıyor.

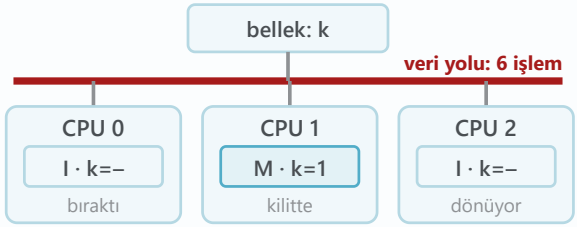
Adım 7 / 10

DEĞİŞKENLER

`k = 0` `CPU1 = S` `CPU2 = S` `veri yolu = 5`

TTAS Kilidi: Kod İzleme

```
1  void kilitle(int *k) {
2    for (;;) {
3      while (*k == 1)   /* yerelde dön */
4        ;
▶      if (tsl(k) == 0)  /* serbest göründü */
▶        return;         /* kilit alındı */
7    }
8  }
9  void ac(int *k) {
10   *k = 0;
11 }
```



önbellek satırı durumu: M, E, S, I

CPU 1 önce TSL yapar: satırı yazmak için alır (M), eski değer 0 → **kilit alındı**.

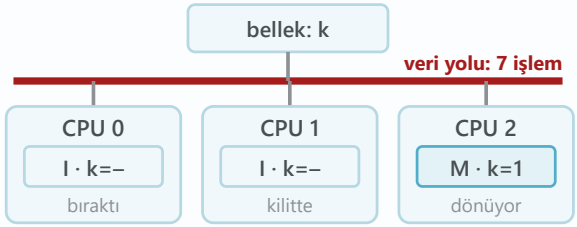
Adım 8 / 10

DEĞİŞKENLER

`k = 1` `CPU1 = M` `CPU2 = I` `veri yolu = 6`

TTAS Kilidi: Kod İzleme

```
1  void kilitle(int *k) {
2    for (;;) {
3      while (*k == 1)   /* yerelde dön */
4        ;
▶      if (tsl(k) == 0)  /* serbest göründü */
6        return;         /* kilit alındı */
7    }
8  }
9  void ac(int *k) {
10   *k = 0;
11 }
```



önbellek satırı durumu: M, E, S, I

CPU 2'nin TSL'i de satırı çeker (CPU 1'in kopyası I) ama eski değer 1: başarısız. Bu yarışın bedeli bir veri yolu işlemi daha.

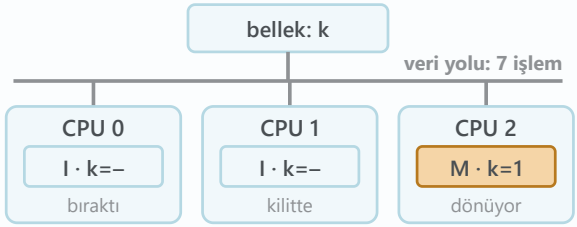
Adım 9 / 10

DEĞİŞKENLER

`k = 1` `CPU1 = I` `CPU2 = M` `veri yolu = 7`

TTAS Kilidi: Kod İzleme

```
1  void kilitle(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* yerelde dön */
▶        ;
5      if (tsl(k) == 0)  /* serbest göründü */
6        return;         /* kilit alındı */
7    }
8  }
9  void ac(int *k) {
10   *k = 0;
11 }
```



önbellek satırı durumu: M, E, S, I

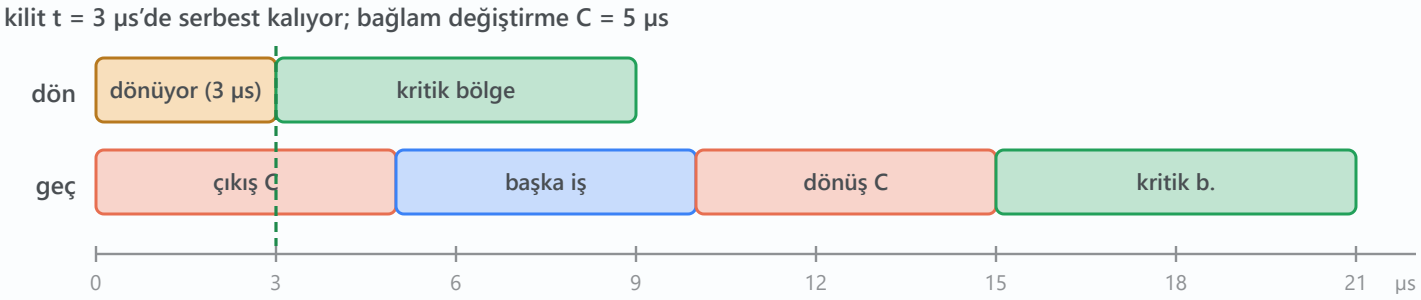
CPU 2 yeniden yerel kopyasında döner. Toplam 7 veri yolu işlemi; bekleme ne kadar uzun sürerse sürsün bu sayı artmaz.

Adım 10 / 10

DEĞİŞKENLER

`k = 1` `CPU1 = I` `CPU2 = M` `veri yolu = 7`

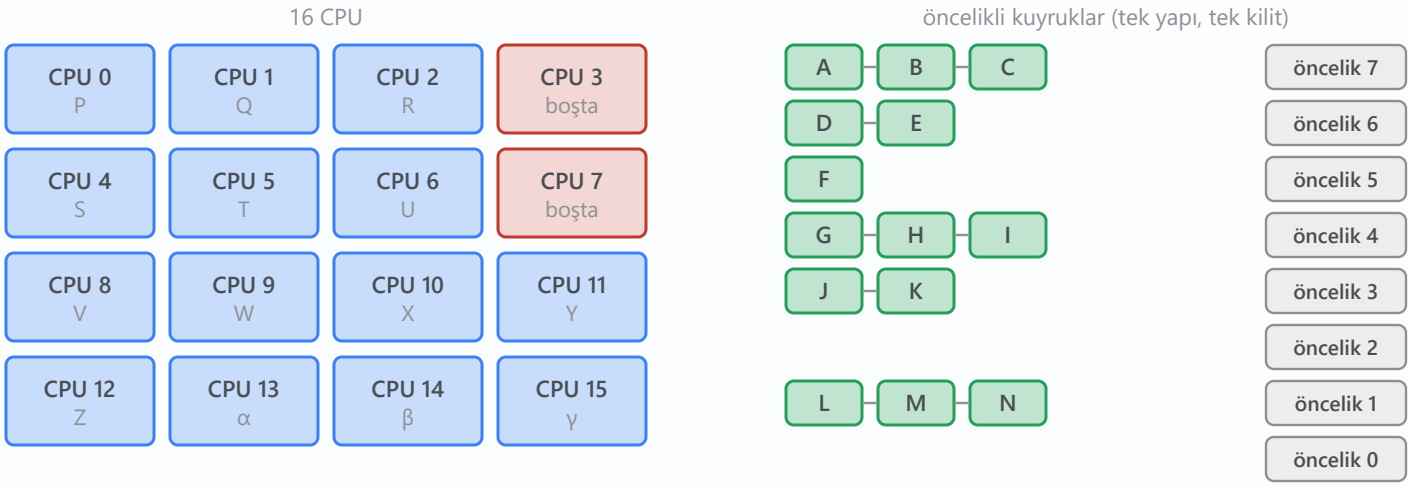
Dönmek mi, Geçiş Yapmak mı?



- **Dönme** (spin): CPU boşa harcanır ama kilit serbest kalır kalmaz devam edilir. Kısa kritik bölgeler için.
- **Geçiş** (switch, block): CPU başka işe verilir; bedeli iki bağlam değiştirme + soğuk önbellek.
- Kural: beklenen bekleme 2C'den kısaysa dönmek, değilse uyumak ucuzdur.

- **Rekabetçi** strateji: geçişin bedeli kadar (2C) dönmek, sonra uyumak → en kötü durumda en iyinin 2 katı maliyet.
- Uygulamada: Linux `futex` + uyarlamalı mutex (sahip çalışıyorsa dön), Windows `CRITICAL_SECTION` dönme sayısı, Java'da kilit şişirme.
- Çekirdekte kesme kapalı bölgelerde yalnız spinlock kullanılabilir.

Zaman Paylaşımı: Ortak Hazır Kuyruğu



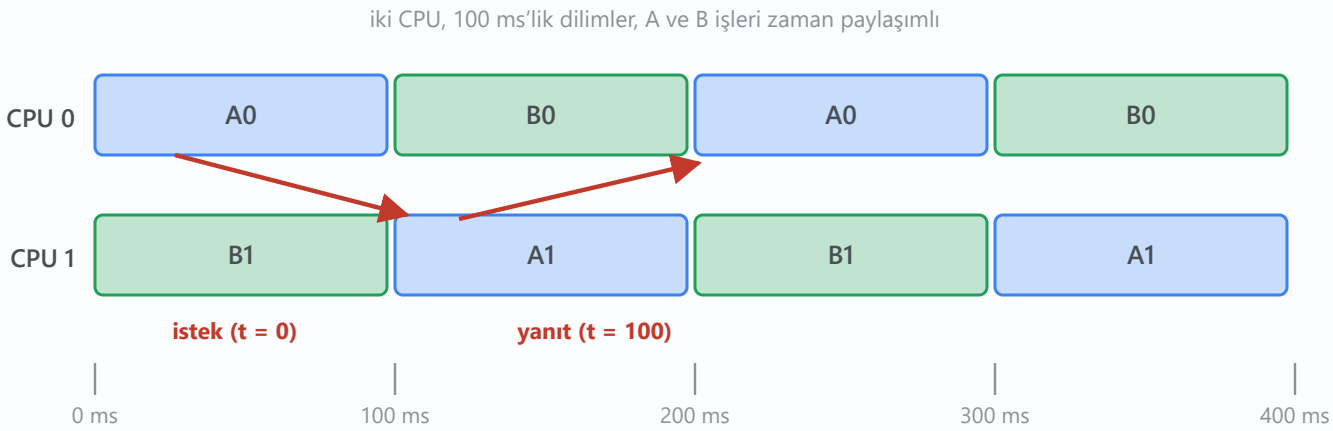
- Bağımsız süreçlerde tüm CPU'lar tek bir öncelikli hazır kuyruğundan iş alır: yük kendiliğinden dengelenir, ama tek kilit darboğaz olur ve süreç soğuk önbellekli bir CPU'ya düşebilir.
- Spinlock tutan iş parçacığının dilimi biterse diğer CPU'lar boşuna döner. **Akıllı çizelgeleme**: kilidi tutan iş parçacığı bir bayrak koyar, çizelgeleyici bayraklı iş parçacığını kesmez, kritik bölgesini bitirmesi için ona biraz ek süre verir.

Alan Paylaşımı (Space Sharing)



- Birbirine bağlı iş parçacıkları **aynı anda** ayrı CPU'larda çalıştırılır: iş başladığında istediği kadar CPU'ya ayrılır, bitene kadar onları tutar.
- CPU başına çok programlama yok → bağlam değiştirme yok. Yetecek CPU yoksa iş, yeterli CPU boşalana kadar bekler (en basit: FCFS).
- Sakınca: iş parçacığı G/Ç için bloke olunca CPU'su **boşa** durur. Bugün HPC kümelerinde iş çizelgeleyicileri (Slurm, PBS) düğümleri böyle bölümler.

Çete (Gang) Çizelgeleme

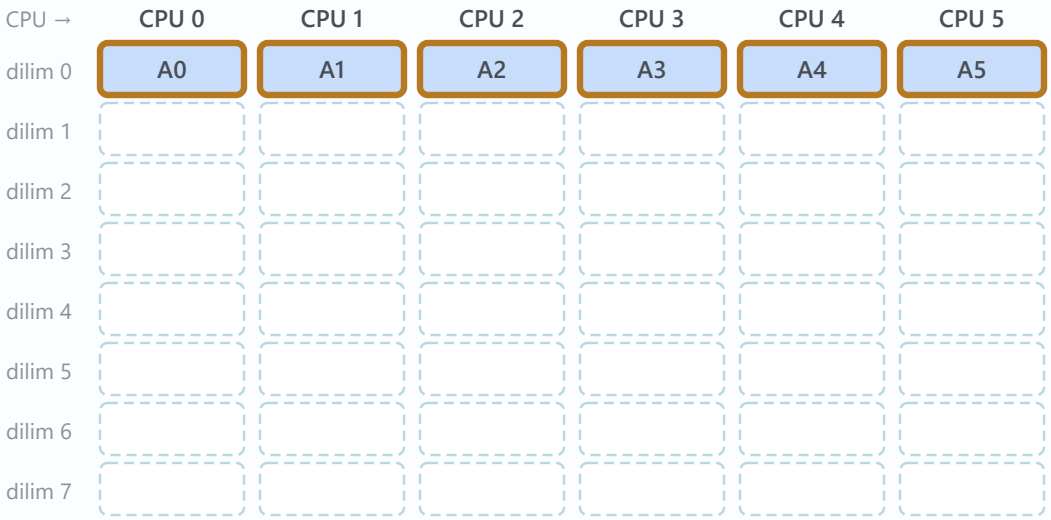


A1 isteği ancak t = 100'de görür, yanıtı A0 t = 200'de alır

birbiriyle konuşan iş parçacıkları farklı zamanlarda çalışırsa her ileti bir dilim bekler

Sorun: A0 ve A1 iletişim kuruyor ama dilimleri kaymış. A0'ın t = 0'daki isteğini A1 ancak 100 ms sonra görür; yanıt da 100 ms bekler.

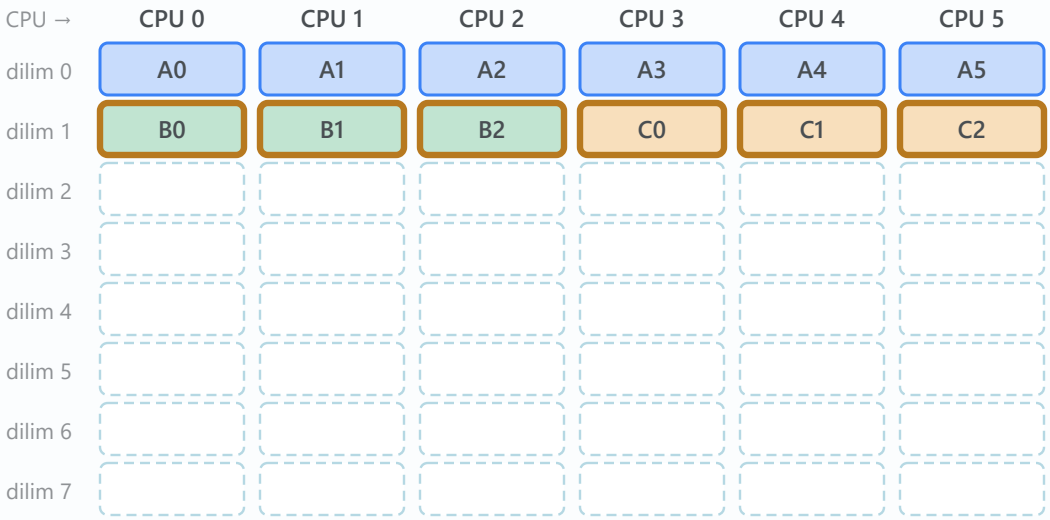
Çete (Gang) Çizelgeleme



A: 6 iş p.
B, C: 3'er
D: 5 · E: 7

Çete çizelgelemede ilgili iş parçacıkları bir **çete**dir. Dilim 0: A'nın 6 iş parçacığının hepsi aynı anda, 6 CPU'da.

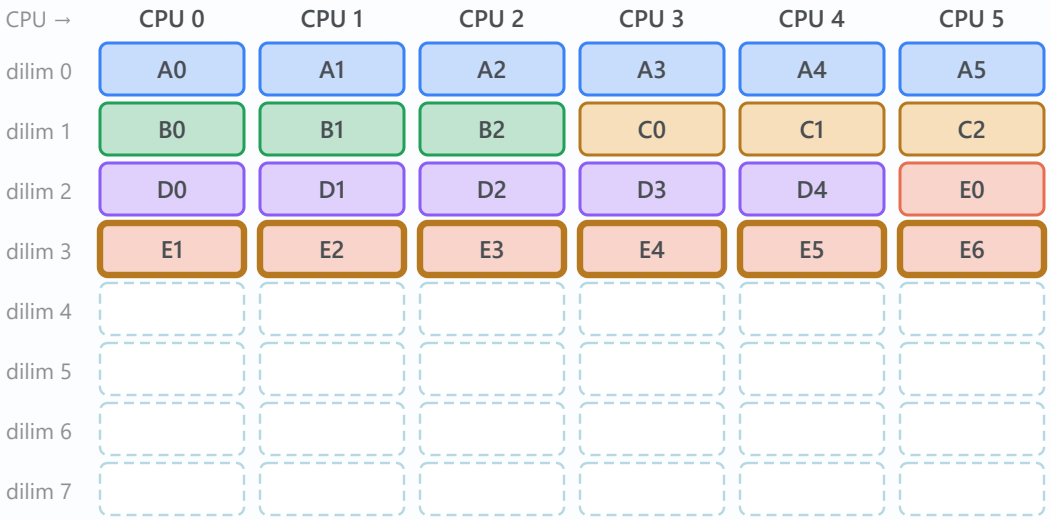
Çete (Gang) Çizelgeleme



A: 6 iş p.
B, C: 3'er
D: 5 · E: 7

Dilim 1: B (3) ve C (3) birlikte; tüm CPU'lar dilimlere **eş zamanlı** başlar ve biter (zaman dilimi senkronize).

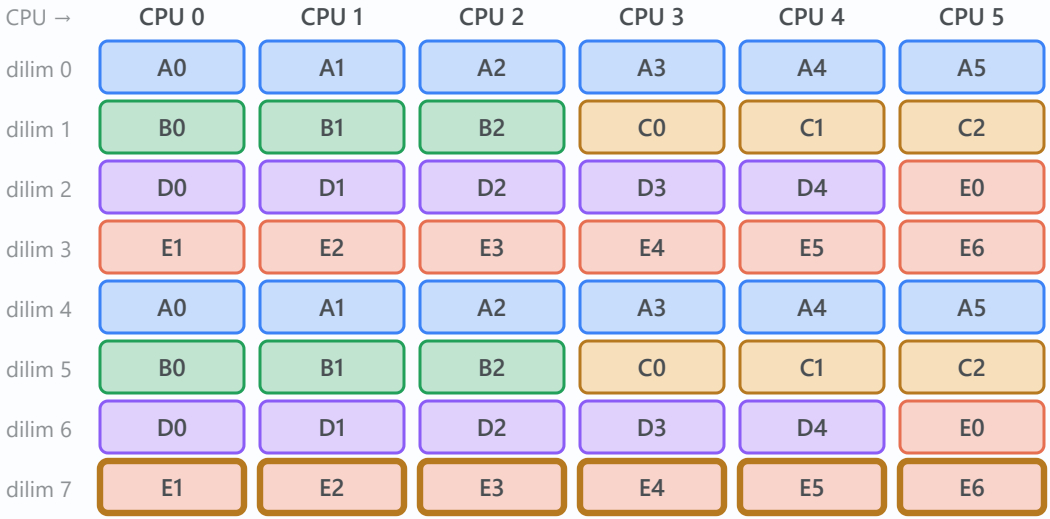
Çete (Gang) Çizelgeleme



A: 6 iş p.
B, C: 3'er
D: 5 · E: 7

Dilim 2–3: D (5) + E0, sonra E'nin kalan 6 iş parçacığı. Bir çetenin tüm üyeleri aynı dilimde koşar; iletiler hemen yanıtlanır.

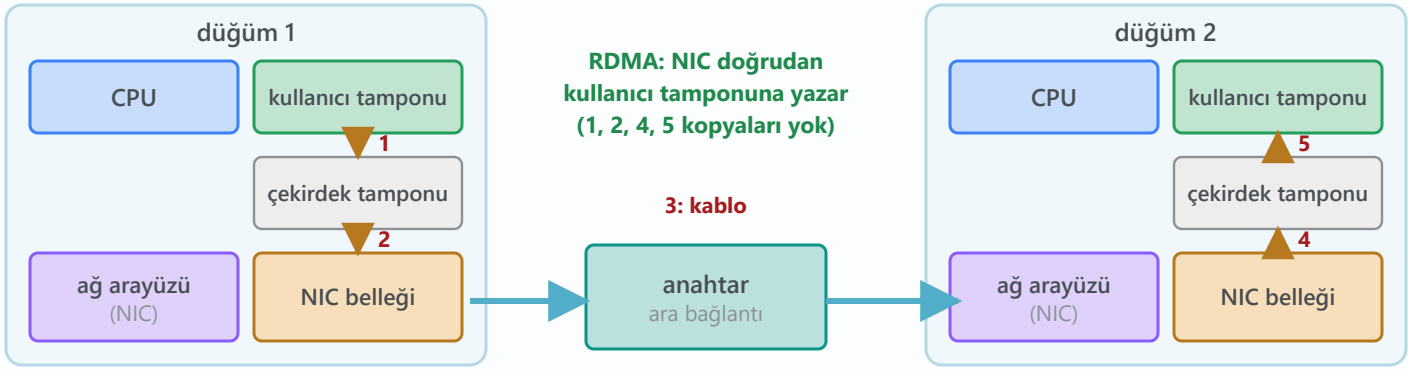
Çete (Gang) Çizelgeleme



A: 6 iş p.
B, C: 3'er
D: 5 · E: 7

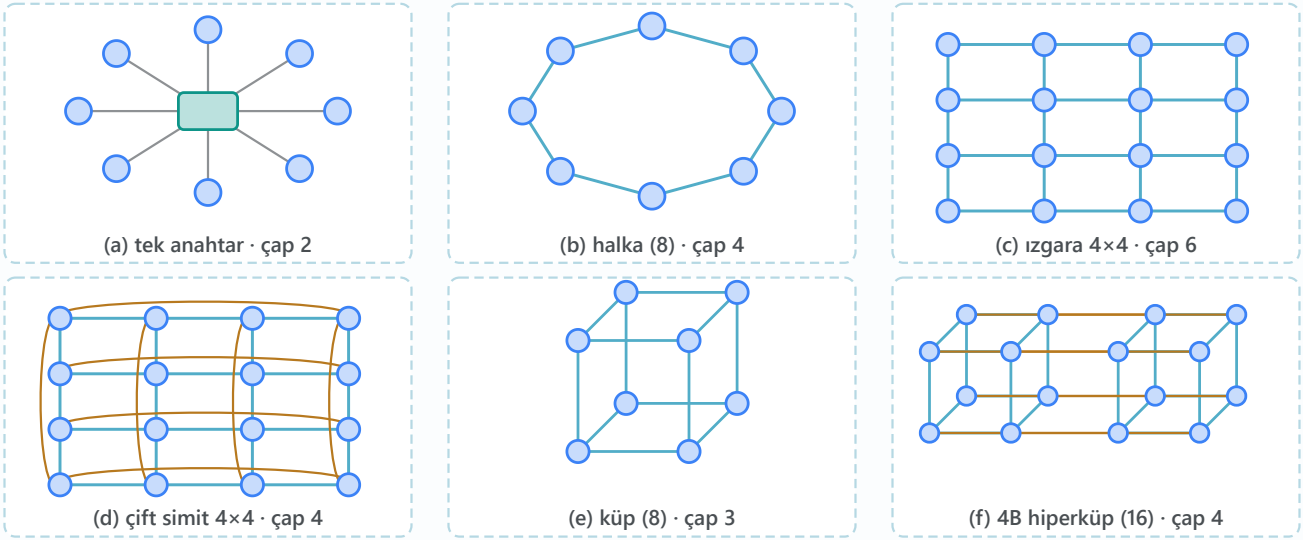
Döngü tekrar eder. İki koşul: çete üyeleri **aynı anda** başlar ve hepsi farklı CPU'lardadır. Günümüzde sanal makinelerde vCPU'ların birlikte çizelgelenmesi (co-scheduling) aynı fikirdir.

Çoklu Bilgisayarlar ve Ağ Arayüzleri



- **Çoklu bilgisayar** (multicomputer, küme): her düğüm CPU + RAM + ağ arayüzü; bellek paylaşılmaz, her düğümde kendi işletim sistemi kopyası çalışır. İletişim yalnız **iletiyle**.
- Bir ileti en kötü 5 kez kopyalanır; her kopya mikrosaniye harcar. Çözüm: NIC belleğini kullanıcı alanına eşle, **DMA** ve **RDMA** (InfiniBand, RoCE): çekirdek hiç devreye girmez.
- Kullanım: büyük ölçekli simülasyon, HPC, yapay zekâ eğitimi; TOP500 sistemleri on binlerce düğümlü çoklu bilgisayarlardır.

Ara Bağlantı Topolojileri



(a) tek anahtar · çap 2

(b) halka (8) · çap 4

(c) ızgara 4×4 · çap 6

(d) çift simit 4×4 · çap 4

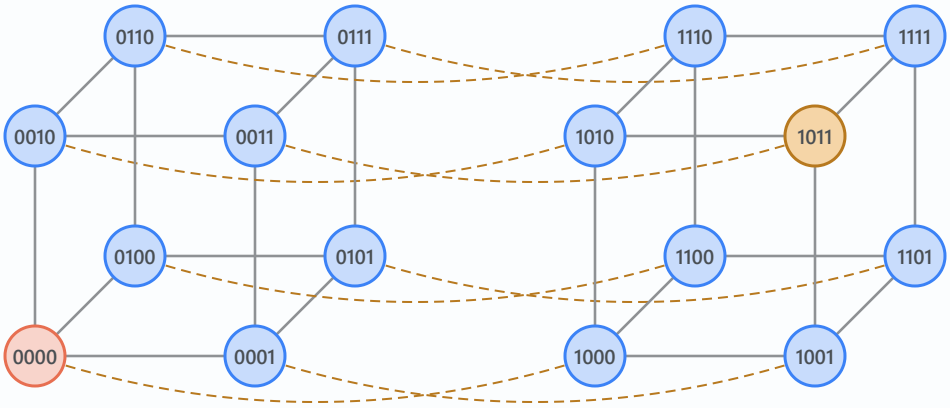
(e) küp (8) · çap 3

(f) 4B hiperküp (16) · çap 4

n = 1.024 düğüm	halka	ızgara 32×32	simit 32×32	hiperküp (10B)
çap (en uzak iki düğüm)	512	62	32	10
bağlantı sayısı	1.024	1.984	2.048	5.120

Çap = en kötü durumda atlama sayısı; hiperküpte $\log_2 n$. Bugünkü süper bilgisayarlar çoğunlukla *fat-tree* ve *dragonfly* topolojisi kullanır; Fugaku 6B simit (Tofu) ağına sahiptir.

Hiperküpte Yönlendirme



hedef 1011

atlama: 0

alt küp: ilk bit 0

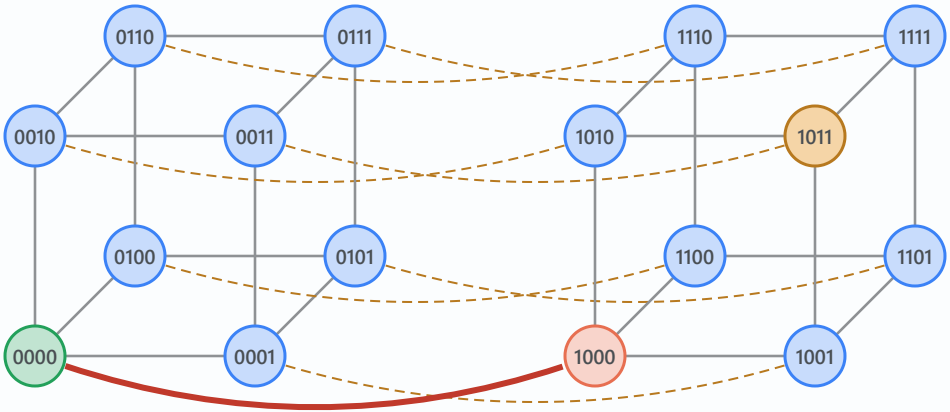
üst küp: ilk bit 1

4B hiperküp: 16 düğüm, her düğüm 4 komşuya bağlı; komşular tek bir bitte farklıdır. 0000'dan 1011'e ileti gidecek.

Adım 1 / 4

Hiperküpte Yönlendirme

hedef 1011
atlama: 1

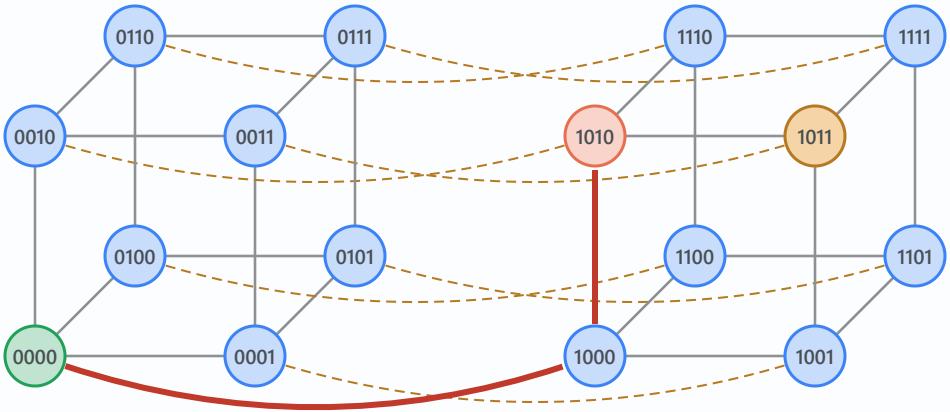


Kaynak ⊕ hedef = 1011: üç bit farklı. Soldan ilk farklı bit düzeltilir → **1000** (alt küpten üst küpe geçiş).

Hiperküpte Yönlendirme

hedef 1011

atlama: 2



alt küp: ilk bit 0

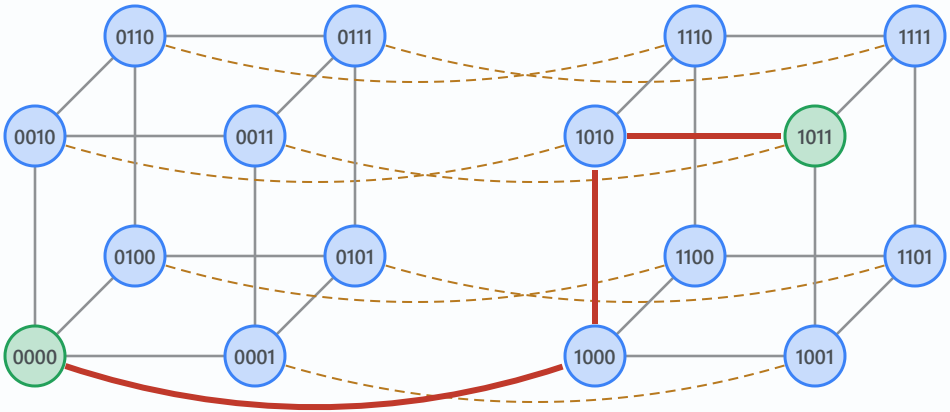
üst küp: ilk bit 1

Sonraki farklı bit: **1010**.

Hiperküpte Yönlendirme

hedef 1011

atlama: 3



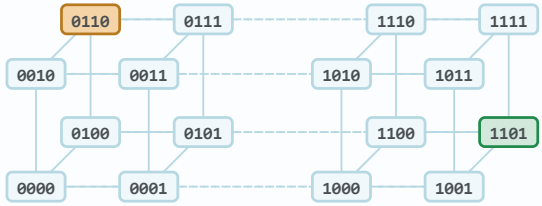
alt küp: ilk bit 0

üst küp: ilk bit 1

Son bit: **1011**. Atlama sayısı = farklı bit sayısı (Hamming uzaklığı) = 3; en kötü durum d = $\log_2 16 = 4$. Kural yerel olduğu için yönlendirme tablosu gerekmez.

E-küp Yönlendirme: Kod İzleme

```
▶ void yonlendir(int src, int dst, int d) {
▶   int u = src;
3   for (int b = d - 1; b >= 0; b--) {
4     int m = 1 << b;
5     if ((u ^ dst) & m) {  /* bit farklı */
6       u ^= m;             /* komşuya geç */
7       printf("%d ", u);
8     }
9   }
10 }
```



u = 0110 · u ⊕ dst = 1011

0110'dan (6) 1101'e (13) ileti. u ⊕ dst = 1011: üç bit farklı, yani en kısa yol 3 atlama. Kesikli kenarlar 4. boyut (iki 3-küp arası).

Adım 1 / 6

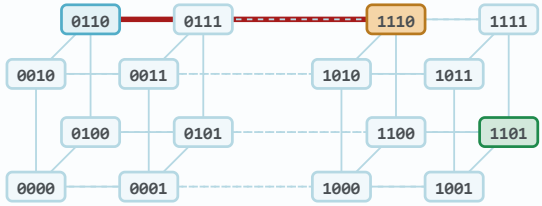
DEĞİŞKENLER

u = **0110** dst = **1101** b = **–** m = **–**

ÇIKTI

E-küp Yönlendirme: Kod İzleme

```
1  void yonlendir(int src, int dst, int d) {
2    int u = src;
▶    for (int b = d - 1; b >= 0; b--) {
▶      int m = 1 << b;
▶      if ((u ^ dst) & m) {  /* bit farklı */
▶        u ^= m;             /* komşuya geç */
▶        printf("%d ", u);
8      }
9    }
10 }
```



u = 1110 · u ⊕ dst = 0011 · bit 3

b = 3: bit farklı → u'nun 3. biti çevrilir, ileti komşu **1110** (14) düğümüne geçer.

Adım 2 / 6

DEĞİŞKENLER

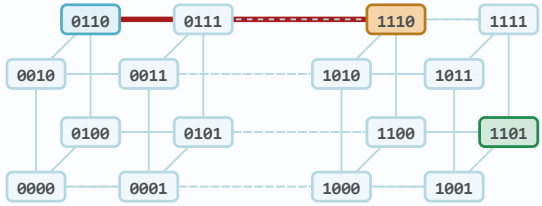
`u = 1110` `dst = 1101` `b = 3` `m = 1000`

ÇIKTI

```
14
```

E-küp Yönlendirme: Kod İzleme

```
1  void yonlendir(int src, int dst, int d) {
2    int u = src;
▶    for (int b = d - 1; b >= 0; b--) {
▶      int m = 1 << b;
▶      if ((u ^ dst) & m) {  /* bit farklı */
6        u ^= m;             /* komşuya geç */
7        printf("%d ", u);
8      }
9    }
10 }
```



u = 1110 · u ⊕ dst = 0011 · bit 2

b = 2: bu bit u ile dst'de aynı (1); atlama yok.

Adım 3 / 6

DEĞİŞKENLER

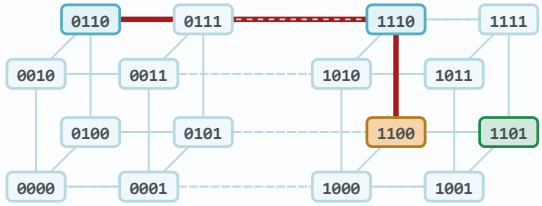
`u = 1110` `dst = 1101` `b = 2` `m = 0100`

ÇIKTI

```
14
```

E-küp Yönlendirme: Kod İzleme

```
1  void yonlendir(int src, int dst, int d) {
2    int u = src;
▶    for (int b = d - 1; b >= 0; b--) {
▶      int m = 1 << b;
▶      if ((u ^ dst) & m) {  /* bit farklı */
▶        u ^= m;             /* komşuya geç */
▶        printf("%d ", u);
8      }
9    }
10 }
```



u = 1100 · u ⊕ dst = 0001 · bit 1

b = 1: bit farklı → u'nun 1. biti çevrilir, ileti komşu **1100** (12) düğümüne geçer.

Adım 4 / 6

DEĞİŞKENLER

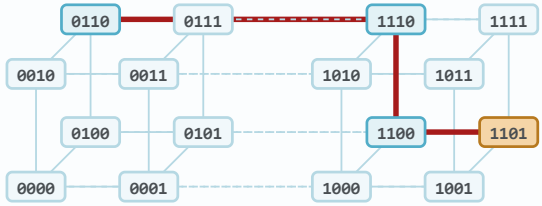
`u = 1100` `dst = 1101` `b = 1` `m = 0010`

ÇIKTI

```
14 12
```

E-küp Yönlendirme: Kod İzleme

```
1  void yonlendir(int src, int dst, int d) {
2    int u = src;
▶    for (int b = d - 1; b >= 0; b--) {
▶      int m = 1 << b;
▶      if ((u ^ dst) & m) {  /* bit farklı */
▶        u ^= m;             /* komşuya geç */
▶        printf("%d ", u);
8      }
9    }
10 }
```



u = 1101 · u ⊕ dst = 0000 · bit 0

b = 0: bit farklı → u'nun 0. biti çevrilir, ileti komşu **1101** (13) düğümüne geçer.

Adım 5 / 6

DEĞİŞKENLER

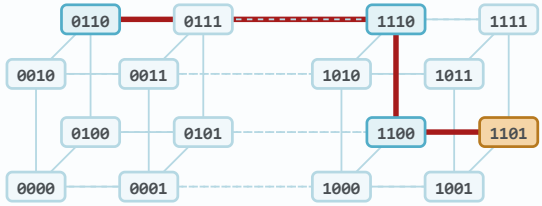
`u = 1101` `dst = 1101` `b = 0` `m = 0001`

ÇIKTI

```
14 12 13
```

E-küp Yönlendirme: Kod İzleme

```
1 void yonlendir(int src, int dst, int d) {
2   int u = src;
3   for (int b = d - 1; b >= 0; b--) {
4     int m = 1 << b;
5     if ((u ^ dst) & m) {  /* bit farklı */
6       u ^= m;             /* komşuya geç */
7       printf("%d ", u);
8     }
9   }
▶ }
```



u = 1101 · u ⊕ dst = 0000

Hedefe 3 atlamada ulaşıldı: 0110 → 1110 → 1100 → 1101. Her düğüm yalnız kendi adresine ve hedefe bakarak karar verir; bitler hep soldan sağa düzeltildiği için yol döngüsüzdür.

Adım 6 / 6

DEĞİŞKENLER

`u = 1101` `dst = 1101` `b = -1` `m = –`

ÇIKTI

```
14 12 13
```

Sakla ve İlet ya da Kesip Geç



sakla ve ilet: paket her anahtarda tamamen alınır, sonra iletilir

gecikme ≈ atlama × (paket / bant genişliği)

t = 0 µs

Örnek: 4 KiB paket, 10 Gb/s bağlantılar; bir bağlantıdan geçmesi $4096 \times 8 / 10^{10} \approx$ **3,28 µs**.

Sakla ve İlet ya da Kesip Geç



sakla ve ilet: paket her anahtarda tamamen alınır, sonra iletilir

gecikme ≈ atlama × (paket / bant genişliği)

t = 3,28 µs

Sakla ve ilet (store-and-forward): anahtar 1 paketin **tamamını** tampona alır, denetler, sonra iletir.

Sakla ve İlet ya da Kesip Geç



sakla ve ilet: paket her anahtarda tamamen alınır, sonra iletilir

gecikme ≈ atlama × (paket / bant genişliği)

t = 9,83 µs

Üç bağlantı → 3 × 3,28 = **9,83 µs**. Gecikme atlama sayısıyla doğrusal büyür.

Sakla ve İlet ya da Kesip Geç



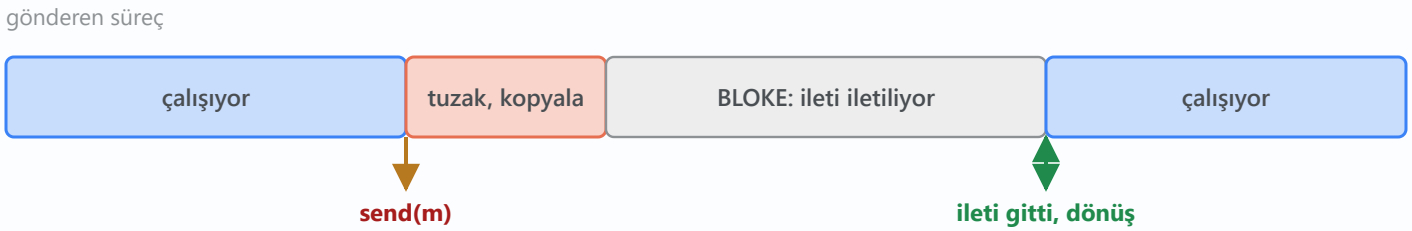
kesip geç / solucan deliği: başlık okunur okunmaz ileri gönderilir

paketin parçaları (flit) aynı anda birden çok bağlantıda

t ≈ 3,38 µs (bir paket süresi + küçük başlık gecikmeleri)

Kesip geç (cut-through) ve solucan deliği (wormhole): başlık gelince yol kurulur, paket boru gibi akar → ≈ **3,38 µs**. HPC ağları ve veri merkezi anahtarları böyle çalışır.

Bloke Eden ve Etmeyen Gönderim



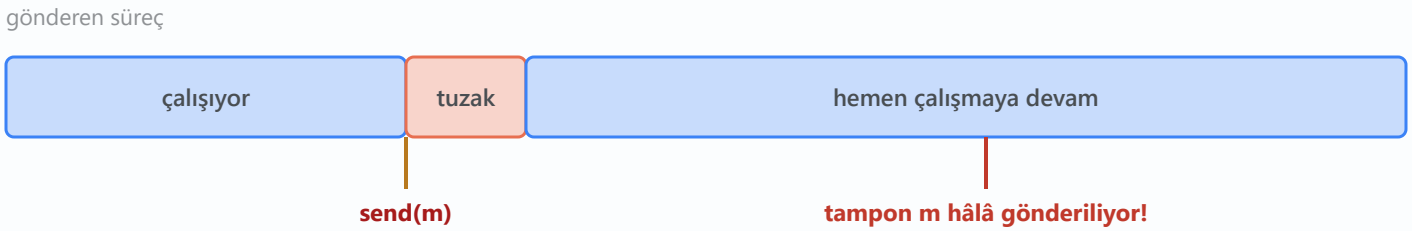
basit ve güvenli; ama CPU iletim boyunca boşta bekler

(a) **Bloke eden** (blocking, senkron) `send` : çağrı ileti iletilene kadar dönmez.

Adım 1 / 3

MPI'de `MPI_Send` / `MPI_Recv` bloke eden, `MPI_Isend` / `MPI_Irecv` + `MPI_Wait` bloke etmeyen çağrılardır; Linux'ta `io_uring` ve sıfır kopya `MSG_ZEROCOPY` aynı soruna modern yanıtlardır.

Bloke Eden ve Etmeyen Gönderim



paralellik var; ama gönderen m'yi iletim bitmeden değiştirirse ne olur?

(b) **Bloke etmeyen** (eş zamansız) `send` : hemen döner; ama tampon iletim bitene kadar kullanılamaz.

Adım 2 / 3

MPI'de `MPI_Send` / `MPI_Recv` bloke eden, `MPI_Isend` / `MPI_Irecv` + `MPI_Wait` bloke etmeyen çağrılardır; Linux'ta `io_uring` ve sıfır kopya `MSG_ZEROCOPY` aynı soruna modern yanıtlardır.

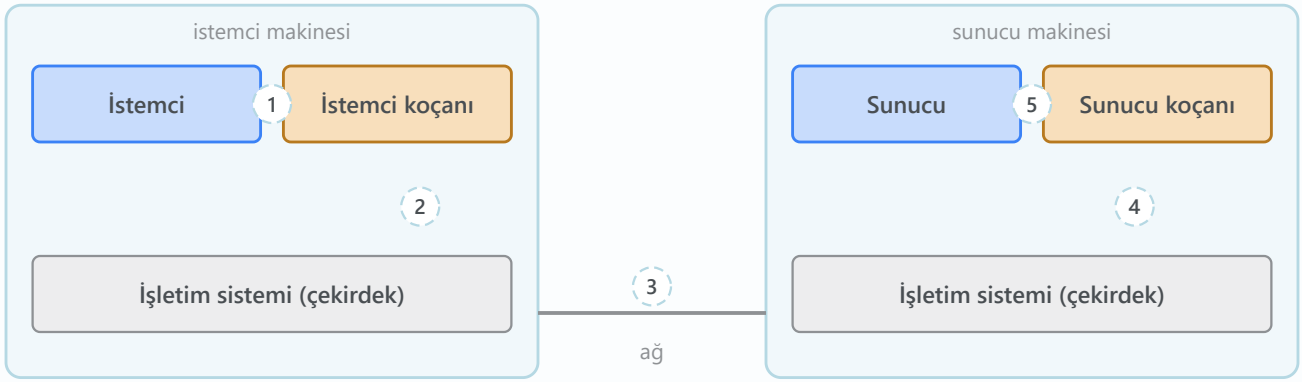
Bloke Eden ve Etmeyen Gönderim

Seçenek	Bedel
Bloke eden gönderim	CPU iletim boyunca boşta
Bloke etmeyen + çekirdeğe kopyala	fazladan kopya için CPU zamanı
Bloke etmeyen + bitince kesme	programlaması zor (yarış durumları)
Yazınca kopyala (CoW)	tampon yazılırsa yine fazladan kopya

Tanenbaum'un dört seçeneği. Alıcı tarafta da benzer: bloke eden `receive`, yoklama (polling) ya da açılan iş parçacığı / etkin ileti (active message).

MPI'de `MPI_Send` / `MPI_Recv` bloke eden, `MPI_Isend` / `MPI_Irecv` + `MPI_Wait` bloke etmeyen çağrılardır; Linux'ta `io_uring` ve sıfır kopya `MSG_ZEROCOPY` aynı soruna modern yanıtlardır.

Uzak Prosedür Çağrısı (RPC)



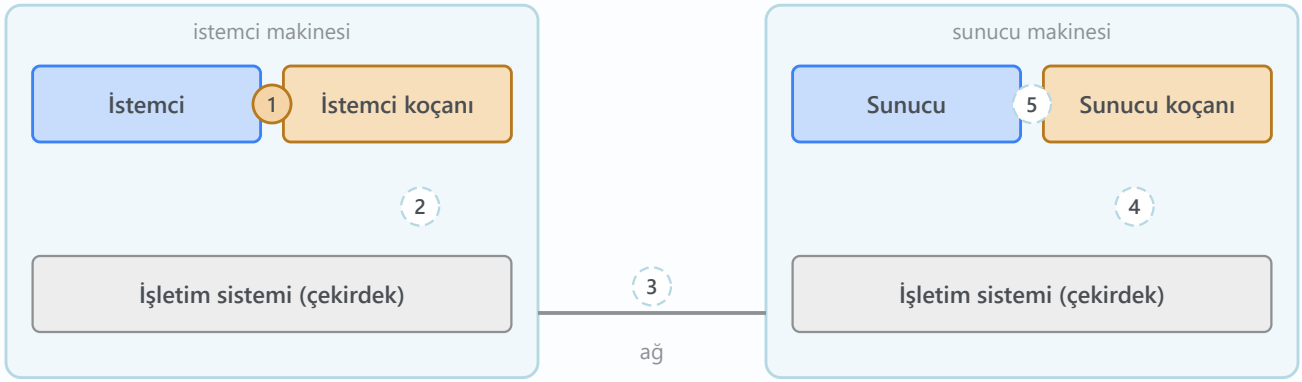
amaç: uzak çağrıyı yerel çağrı gibi göstermek (Birrell ve Nelson, 1984)

İstemci `sonuc = topla(3, 4)` der; sıradan yerel çağrı gibi görünür.

Adım 1 / 6

Zorluklar: işaretçiler (başka adres alanı → kopyala-geri yükle), uzunluğu bilinmeyen diziler, global değişkenler, kısmi hatalar (çağrı en az bir kez mi, en çok bir kez mi?). Koçan (stub) arayüz tanımından otomatik üretilir; bugün gRPC + Protocol Buffers, Thrift, Java RMI.

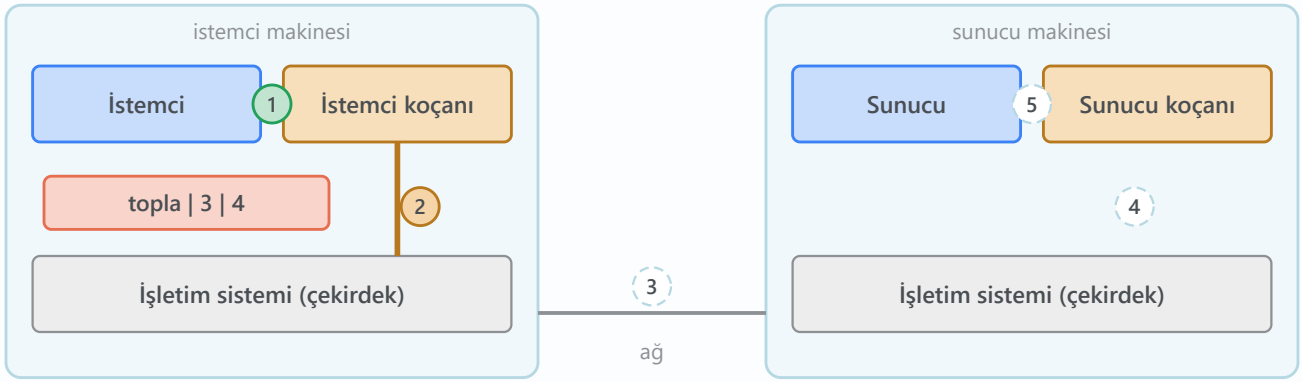
Uzak Prosedür Çağrısı (RPC)



1: İstemci, **istemci koçanını** (stub) yerel bir prosedür gibi çağırır.

Zorluklar: işaretçiler (başka adres alanı → kopyala-geri yükle), uzunluğu bilinmeyen diziler, global değişkenler, kısmi hatalar (çağrı en az bir kez mi, en çok bir kez mi?). Koçan (stub) arayüz tanımından otomatik üretilir; bugün gRPC + Protocol Buffers, Thrift, Java RMI.

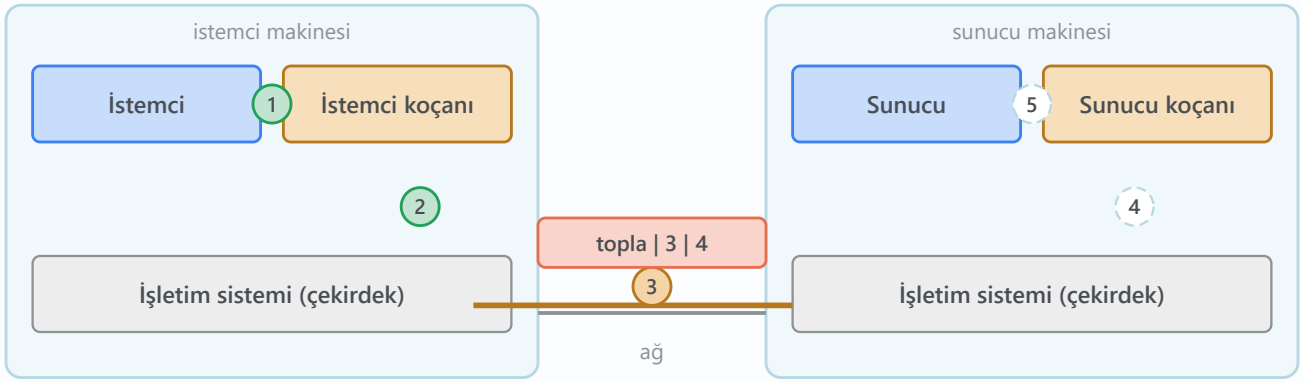
Uzak Prosedür Çağrısı (RPC)



2: Koçan parametreleri iletiye **paketler** (marshalling) ve çekirdeğe sistem çağrısı yapar.

Zorluklar: işaretçiler (başka adres alanı → kopyala-geri yükle), uzunluğu bilinmeyen diziler, global değişkenler, kısmi hatalar (çağrı en az bir kez mi, en çok bir kez mi?). Koçan (stub) arayüz tanımından otomatik üretilir; bugün gRPC + Protocol Buffers, Thrift, Java RMI.

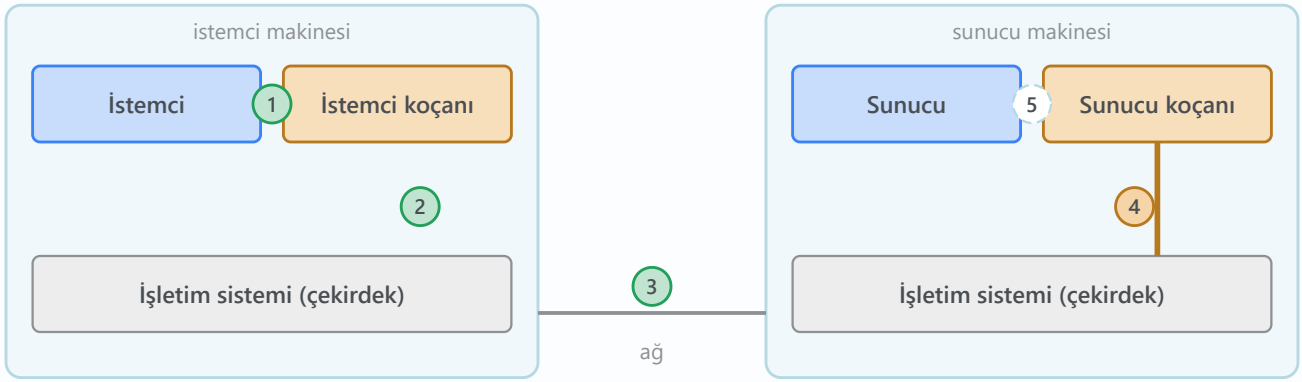
Uzak Prosedür Çağrısı (RPC)



3: Çekirdek iletiyi ağ üzerinden sunucu makinesine gönderir.

Zorluklar: işaretçiler (başka adres alanı → kopyala-geri yükle), uzunluğu bilinmeyen diziler, global değişkenler, kısmi hatalar (çağrı en az bir kez mi, en çok bir kez mi?). Koçan (stub) arayüz tanımından otomatik üretilir; bugün gRPC + Protocol Buffers, Thrift, Java RMI.

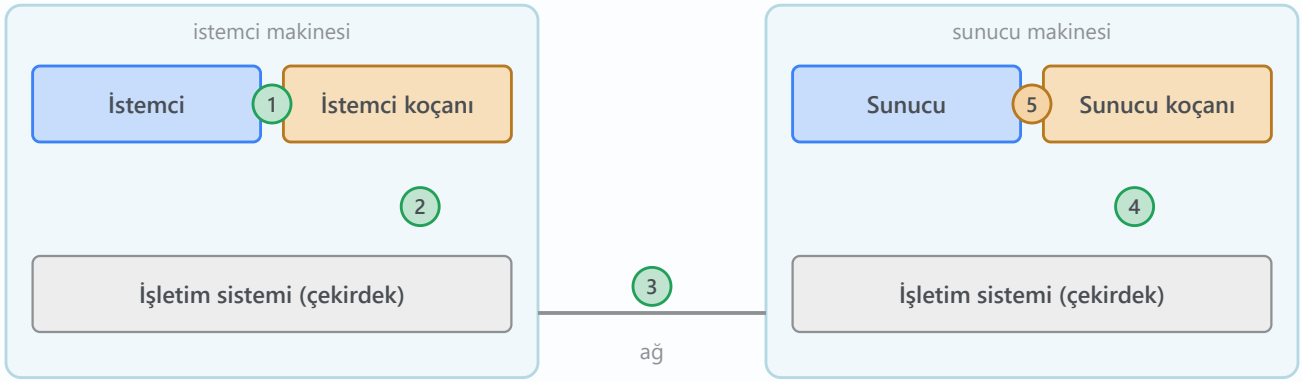
Uzak Prosedür Çağrısı (RPC)



4: Sunucu çekirdeği iletiyi **sunucu koçanına** verir.

Zorluklar: işaretçiler (başka adres alanı → kopyala-geri yükle), uzunluğu bilinmeyen diziler, global değişkenler, kısmi hatalar (çağrı en az bir kez mi, en çok bir kez mi?). Koçan (stub) arayüz tanımından otomatik üretilir; bugün gRPC + Protocol Buffers, Thrift, Java RMI.

Uzak Prosedür Çağrısı (RPC)

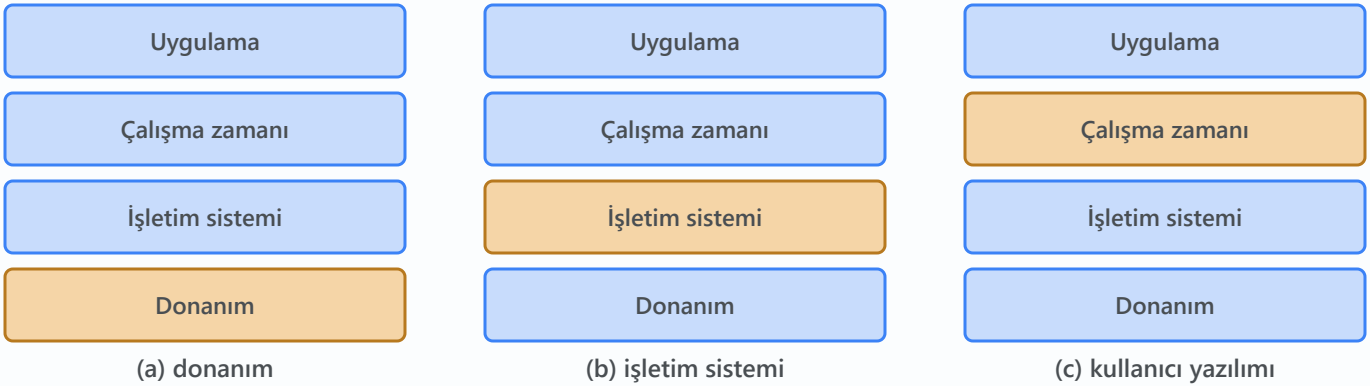


yanıt (7) ters yoldan: sunucu → koçan → çekirdek → ağ → istemci

5: Koçan parametreleri açar ve sunucu prosedürünü yerel olarak çağırır; yanıt aynı yoldan geri döner.

Zorluklar: işaretçiler (başka adres alanı → kopyala-geri yükle), uzunluğu bilinmeyen diziler, global değişkenler, kısmi hatalar (çağrı en az bir kez mi, en çok bir kez mi?). Koçan (stub) arayüz tanımından otomatik üretilir; bugün gRPC + Protocol Buffers, Thrift, Java RMI.

Dağıtık Paylaşımlı Bellek (DSM)



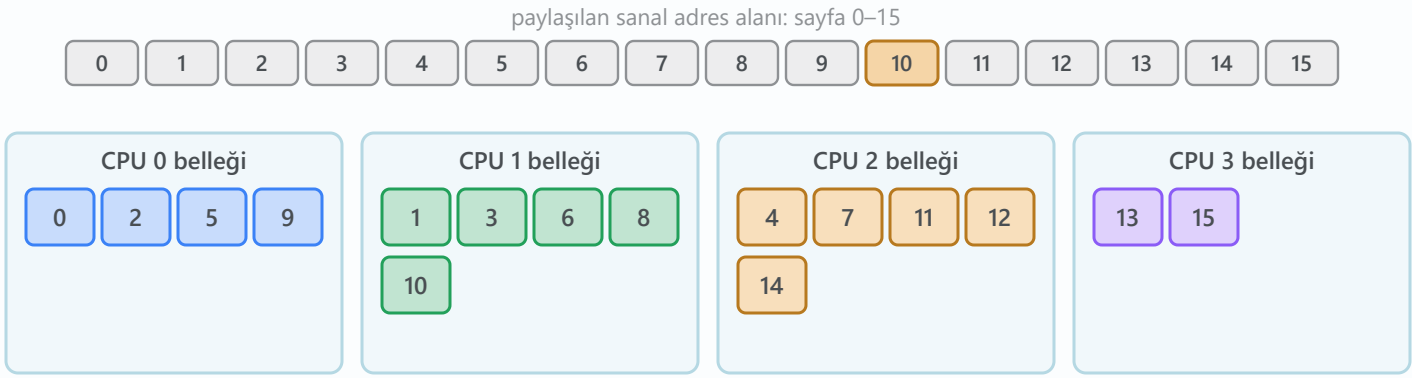
turuncu katman paylaşılan adres alanı yanılsamasını üretir

DSM, belleği paylaşmayan makinelerde paylaşımlı bellek **yanılsaması** verir. Katman donanım (NUMA), işletim sistemi (sayfa tabanlı, Li ve Hudak 1989) ya da kullanıcı kitaplığı olabilir.

Adım 1 / 4

Sayfa (4 KiB) ağda pahalıdır ve kaba tanelidir; bu yüzden DSM yerini ileti iletimine (MPI) ve PGAS dillerine bıraktı. Bugünkü karşılığı CXL ile makineler arası bellek havuzlarıdır.

Dağıtık Paylaşımlı Bellek (DSM)



Sayfalar makinelere dağıtılmıştır. CPU 0 sayfa 10'a erişince MMU sayfa hatası üretir.

Adım 2 / 4

Sayfa (4 KiB) ağda pahalıdır ve kaba tanelidir; bu yüzden DSM yerini ileti iletimine (MPI) ve PGAS dillerine bıraktı. Bugünkü karşılığı CXL ile makineler arası bellek havuzlarıdır.

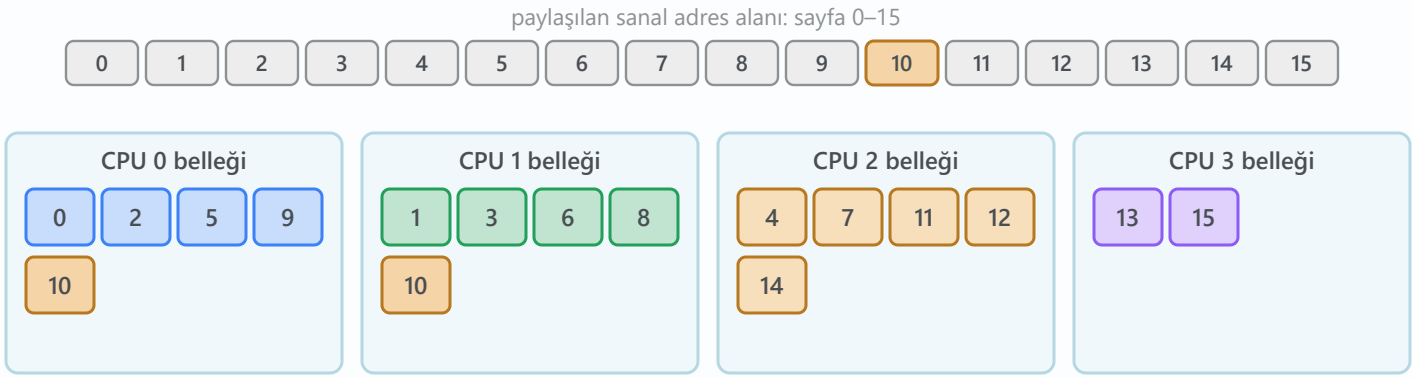
Dağıtık Paylaşımlı Bellek (DSM)



İşletim sistemi sayfanın kimde olduğunu bulur, ister; sayfa CPU 0'a **taşınır** ve komut yeniden yürütülür.

Sayfa (4 KiB) ağda pahalıdır ve kaba tanelidir; bu yüzden DSM yerini ileti iletimine (MPI) ve PGAS dillerine bıraktı. Bugünkü karşılığı CXL ile makineler arası bellek havuzlarıdır.

Dağıtık Paylaşımlı Bellek (DSM)

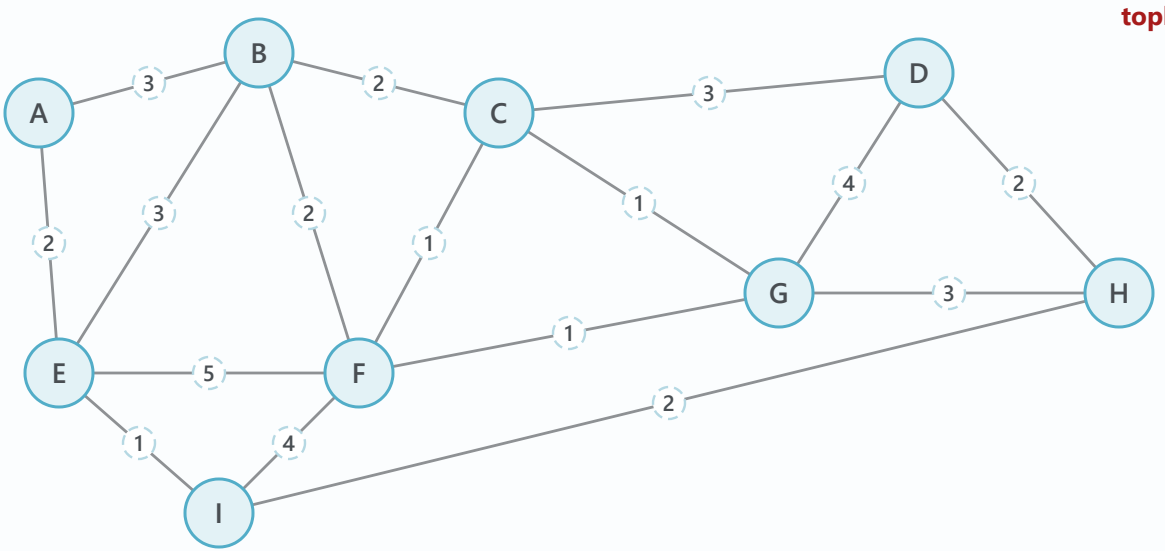


salt okunur sayfa çoğaltılır: CPU 0 ve CPU 1'de kopya; yazınca kopyalar geçersiz kılınır

Sayfa salt okunursa **çoğaltılır** (replication): iki CPU da yerel okur. Yazmada diğer kopyalar geçersiz kılınır; aynı sayfadaki bağımsız değişkenler **yanlış paylaşıma** yol açar.

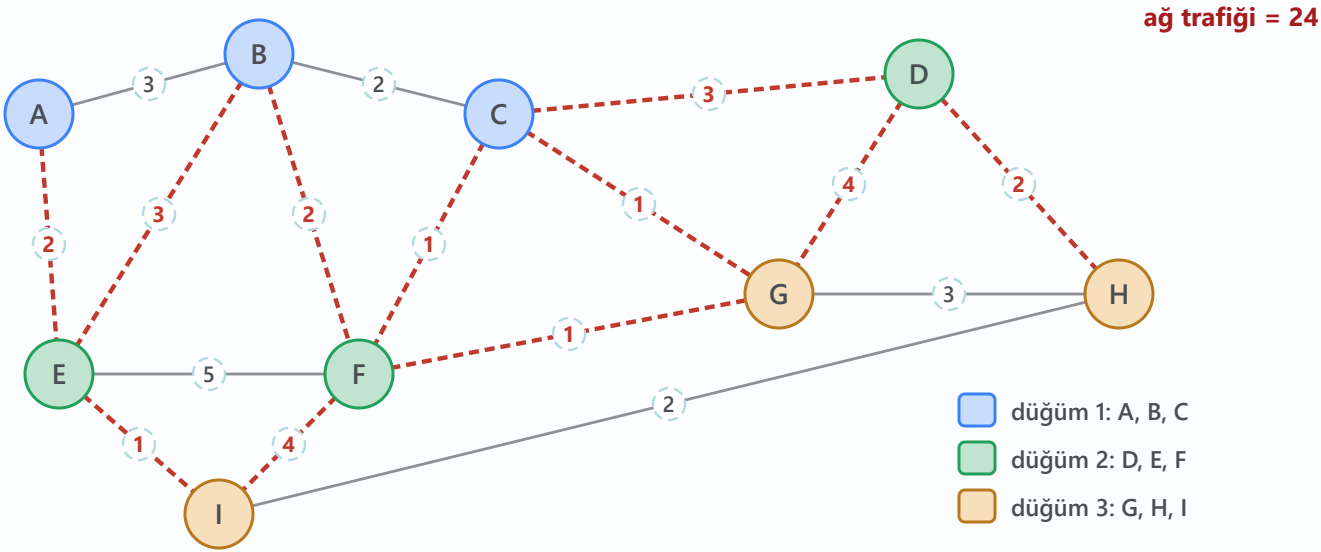
Sayfa (4 KiB) ağda pahalıdır ve kaba tanelidir; bu yüzden DSM yerini ileti iletimine (MPI) ve PGAS dillerine bıraktı. Bugünkü karşılığı CXL ile makineler arası bellek havuzlarıdır.

Çizge Kuramsal Deterministik Atama



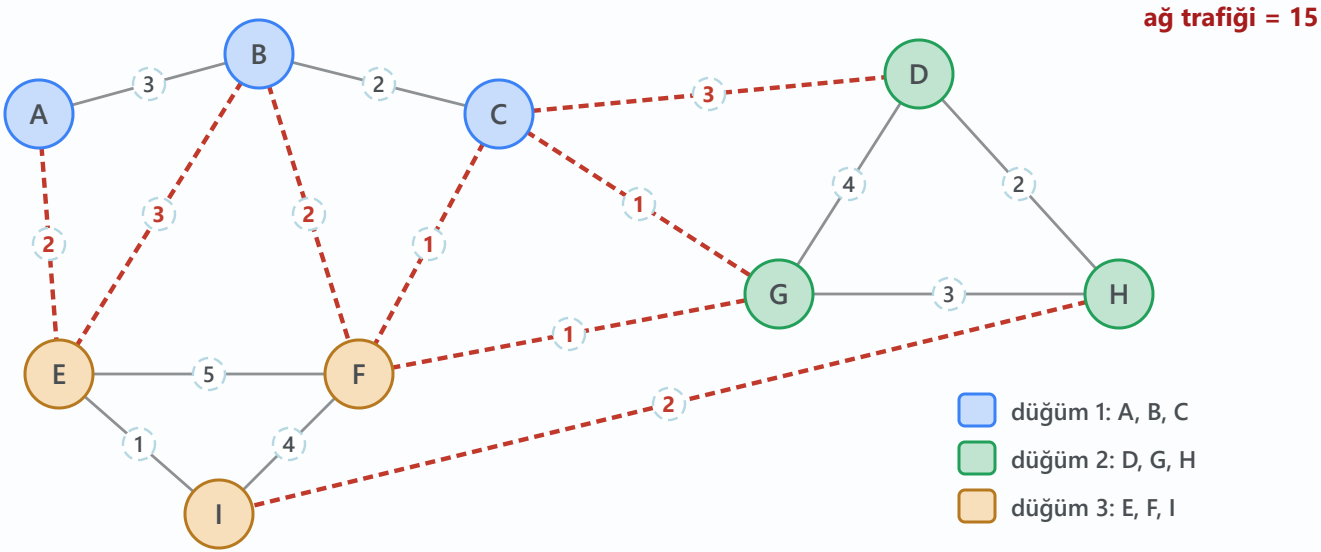
9 süreç (düğüm) ve aralarındaki ileti trafiği (kenar ağırlığı). 3 makineye 3'er süreç atanacak; hedef: **ağ trafiğini** en aza indirmek.

Çizge Kuramsal Deterministik Atama



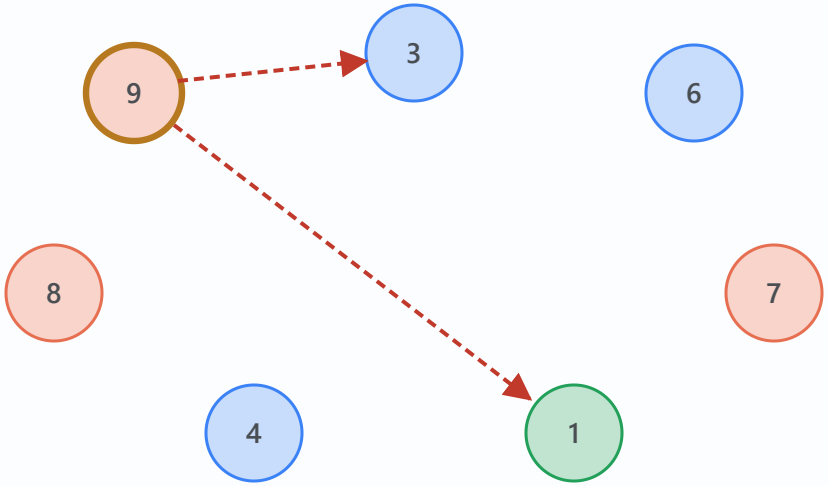
Alfabetik atama: kesilen (makineler arası) kenarların toplamı **24**.

Çizge Kuramsal Deterministik Atama



En iyi bölümleme (tüm 280 dengeli seçenek denendi): **15**. Çok konuşan süreçler aynı makinede; problem genelde NP-zordur, sezgisel yöntemlerle çözülür.

Dağıtık Sezgisel Yük Dengeleme



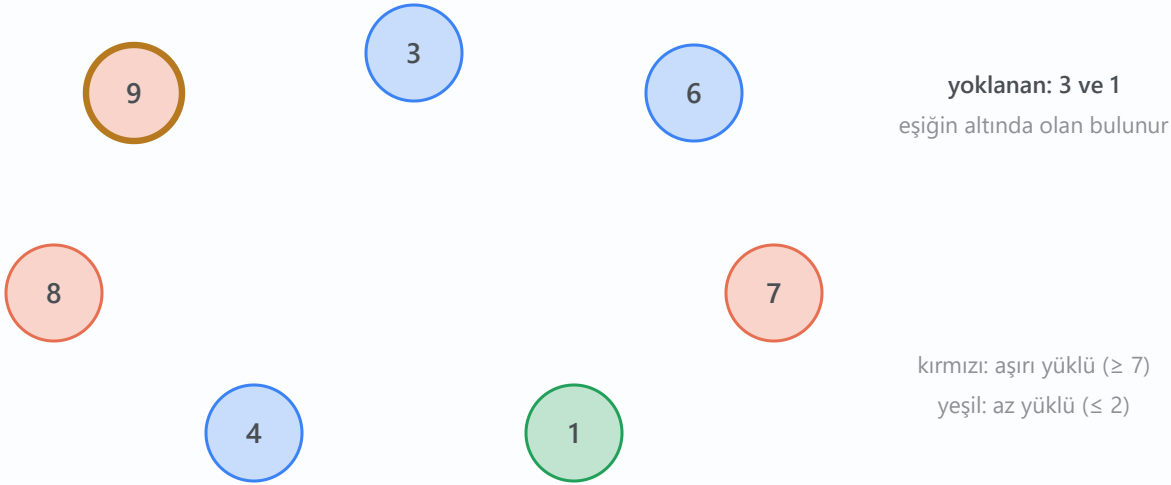
yük ≥ 7: aşırı yüklü

rastgele düğümler yoklanır:
"yükün ne?"

kırmızı: aşırı yüklü (≥ 7)
yeşil: az yüklü (≤ 2)

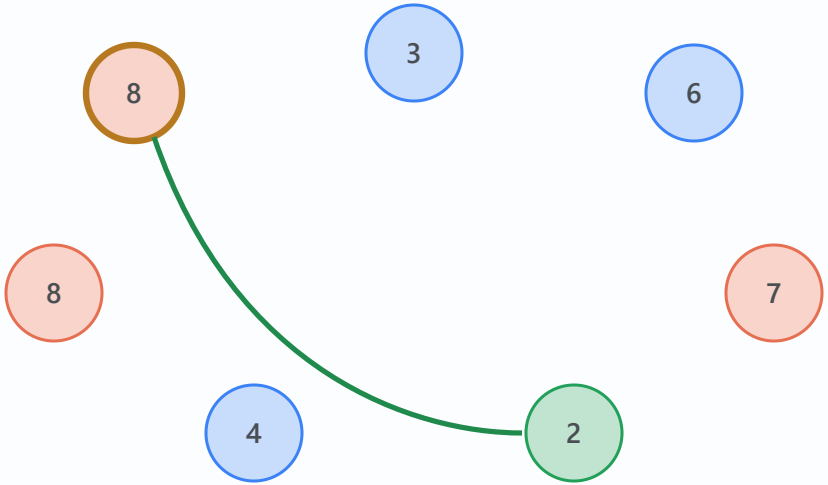
Gönderici başlatmalı: yeni iş gelince yükü eşiği aşan düğüm rastgele düğümleri yoklar (Eager ve diğ., 1986).

Dağıtık Sezgisel Yük Dengeleme



Yoklama sınırlı sayıda yapılır (ör. N = 3); eşiğin altında bir düğüm bulunursa iş ona gönderilir, bulunamazsa yerelde çalışır.

Dağıtık Sezgisel Yük Dengeleme

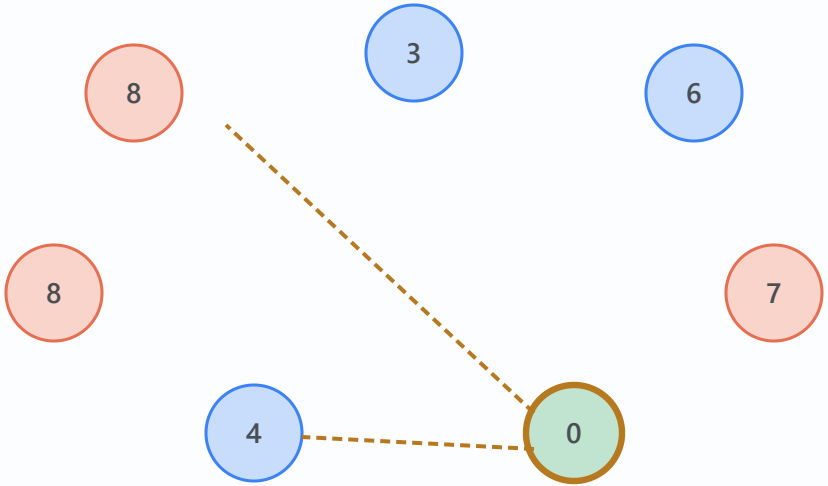


gönderici başlatmalı:
iş yükü 1 olan düğüme gönderildi

kırmızı: aşırı yüklü (≥ 7)
yeşil: az yüklü (≤ 2)

İş taşındı. Sakınca: sistem **tümüyle yüklüyken** yoklamaların hepsi boşa gider ve yükü artırır.

Dağıtık Sezgisel Yük Dengeleme

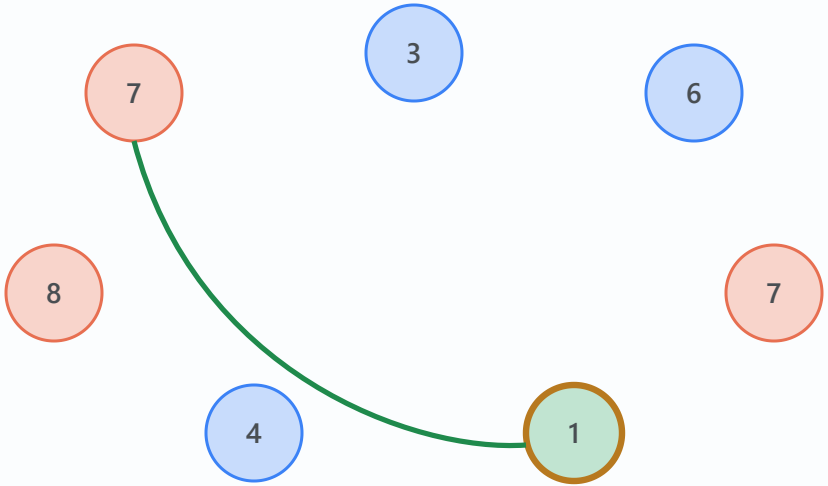


alıcı başlatmalı:
boş kalan düğüm iş arar

kırmızı: aşırı yüklü (≥ 7)
yeşil: az yüklü (≤ 2)

Alıcı başlatmalı: bir işi biten ve boş kalan düğüm kendisi iş arar; yoklama maliyeti boştaki düğüme yüklenir.

Dağıtık Sezgisel Yük Dengeleme



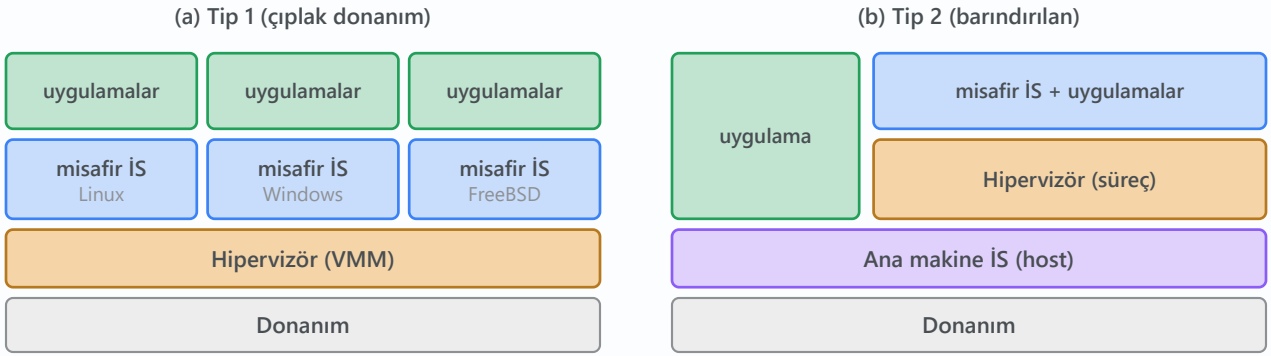
aşırı yüklü düğüm
bir işi boş düğüme verdi

kırmızı: aşırı yüklü (≥ 7)
yeşil: az yüklü (≤ 2)

Yüklü bir düğüm işini devretti. Yük yüksekken iyi çalışır. Bugünkü karşılığı **iş çalma** (work stealing): Cilk, Java ForkJoinPool, Go ve Tokio çalışma zamanları.

SANALLAŞTIRMA

Hipervizörler: Tip 1 ve Tip 2

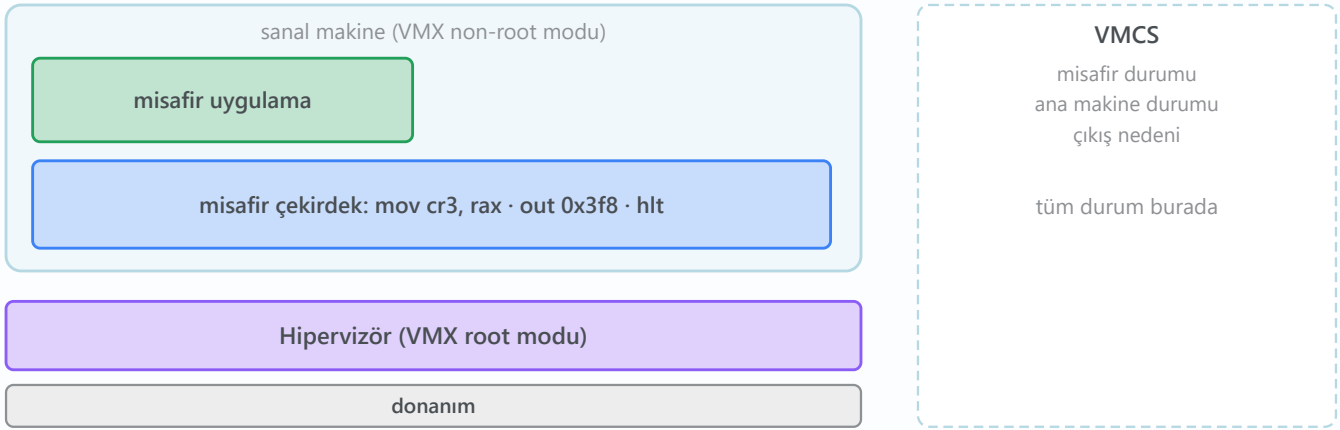


- **Tip 1**: VMware ESXi, Xen (XCP-ng), Microsoft Hyper-V; Linux **KVM** çekirdeği hipervizöre dönüştürür (bulutların çoğu: AWS Nitro, Google). Hyper-V açılınca Windows da bir **kök bölümde** çalışır.
- **Tip 2**: VirtualBox, VMware Workstation/Fusion, Parallels, UTM. **Saf** tip 2 yorumlar ya da ikili çevirir (KVM'siz QEMU); **pratik** tip 2 çekirdek modülüyle VT-x/AMD-V kullanır.

- Popek–Goldberg (1974): **hassas** komutların hepsi ayrıcalıklıysa (kullanıcı modunda tuzağa düşüyorsa) makine sanallaştırılabilir.
- Klasik x86 bu koşulu sağlamıyordu (ör. `POPF` kullanıcı modunda sessizce başarısız olur) → ikili çeviri, sonra VT-x/AMD-V.

Kullanım nedenleri: sunucu birleştirme, çalışan VM'i başka makineye taşıma (canlı göç) ve anlık görüntü; bulutun temeli budur. Tip 2'de G/Ç ana İS'den geçtiği için ek yük daha fazladır.

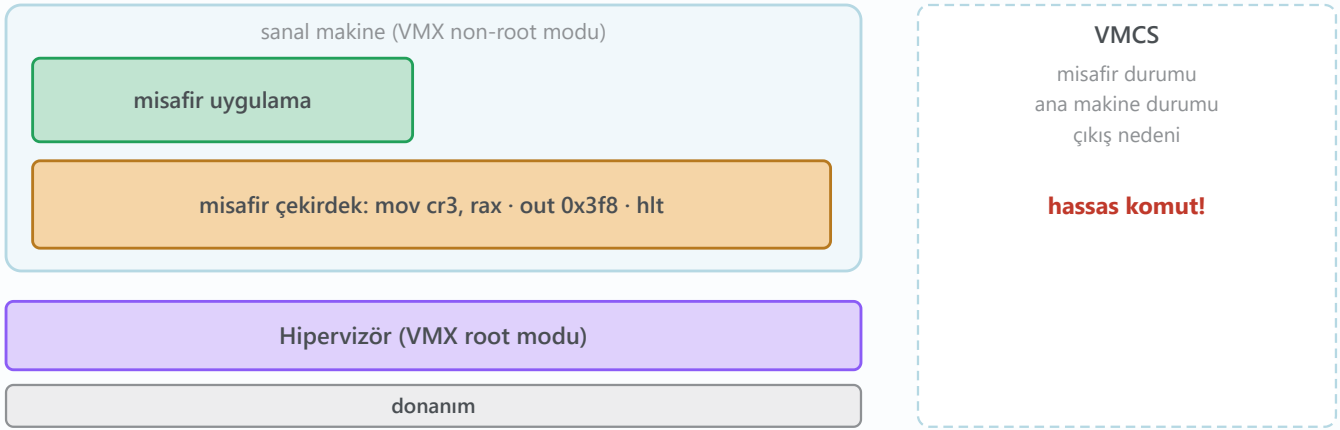
Tuzak ve Benzet: VT-x / AMD-V



Misafir işletim sistemi kendini çekirdek modunda sanır ama gerçek denetim hipervizördedir. Intel VT-x / AMD-V (2005–06) yeni bir mod ekler: **root** (hipervizör) ve **non-root** (misafir).

Donanım desteğinden önce VMware (1999) hassas komutları çalışma anında **ikili çeviri** ile değiştiriyordu. Misafir işletim sisteminin yalnız *hassas* komutları tuzağa düşer, diğerleri doğrudan donanımda çalışır.

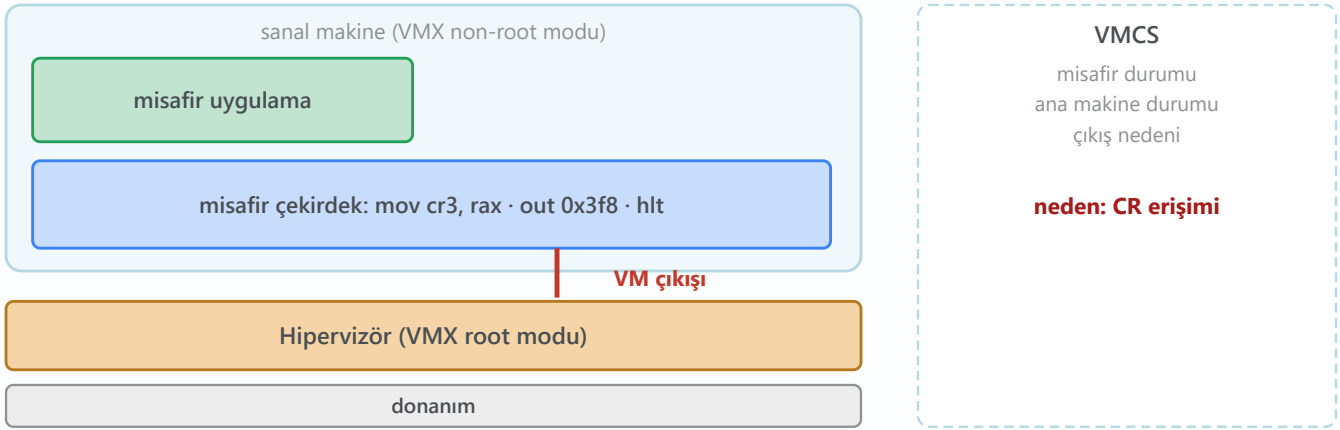
Tuzak ve Benzet: VT-x / AMD-V



Misafir çekirdek hassas bir komut yürütür: sayfa tablosunu değiştirmek (CR3), G/Ç portu, `HLT` …

Donanım desteğinden önce VMware (1999) hassas komutları çalışma anında **ikili çeviri** ile değiştiriyordu. Misafir işletim sisteminin yalnız *hassas* komutları tuzağa düşer, diğerleri doğrudan donanımda çalışır.

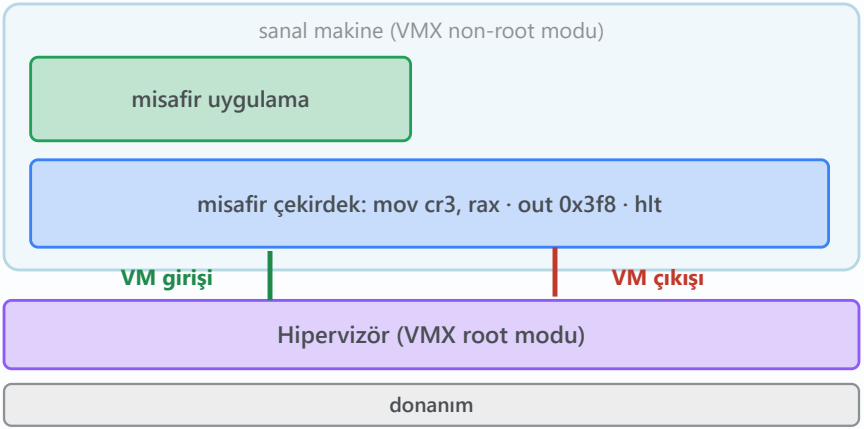
Tuzak ve Benzet: VT-x / AMD-V



Donanım komutu çalıştırmaz: **VM çıkışı** (VM exit) ile hipervizöre tuzak kurar; neden VMCS'ye yazılır.

Donanım desteğinden önce VMware (1999) hassas komutları çalışma anında **ikili çeviri** ile değiştiriyordu. Misafir işletim sisteminin yalnız *hassas* komutları tuzağa düşer, diğerleri doğrudan donanımda çalışır.

Tuzak ve Benzet: VT-x / AMD-V



VMCS

misafir durumu
ana makine durumu
çıkış nedeni

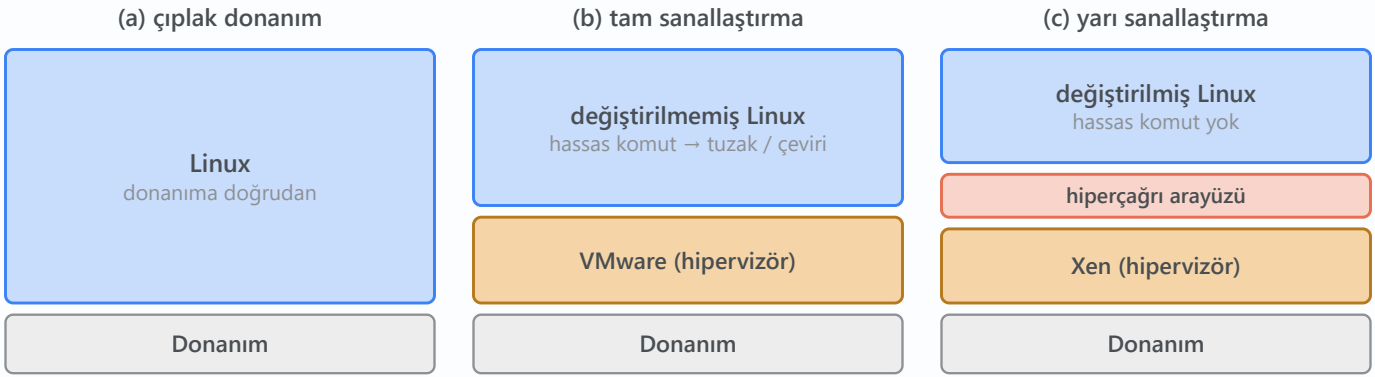
neden: CR erişimi

benzetildi:
gölge/EPT güncellendi

Hipervizör komutun etkisini **benzetir** (emulate), sanal donanım durumunu günceller ve **VM girişi** ile misafire döner. Bir çıkış ~1 µs'dir; sık çıkışlar performansı düşürür.

Donanım desteğinden önce VMware (1999) hassas komutları çalışma anında **ikili çeviri** ile değiştiriyordu. Misafir işletim sisteminin yalnız *hassas* komutları tuzağa düşer, diğerleri doğrudan donanımda çalışır.

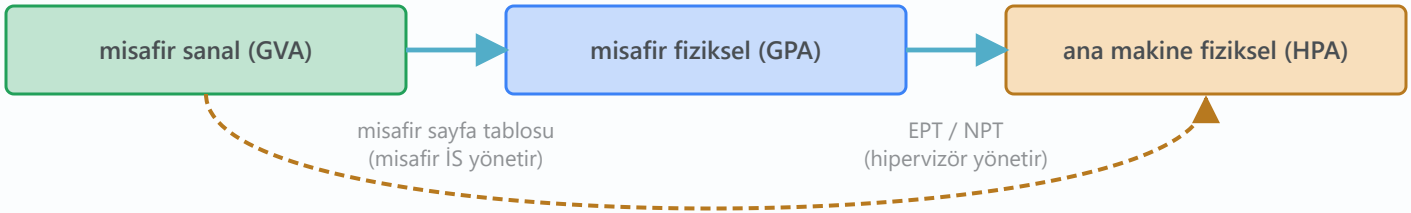
Yarı Sanallaştırma (Paravirtualization)



- Misafir işletim sistemi **değiştirilir**: hassas komutlar yerine hipervizöre doğrudan **hiperçağrı** (hypercall) yapar — sistem çağrısının bir alt katmandaki benzeri.
- Tuzak ve benzetme maliyeti yok, daha hızlı; ama kaynak kodu değiştirilebilen işletim sistemi gerekir (Xen 2003).
- VMI (Virtual Machine Interface), tek çekirdeğin hem donanımda hem farklı hipervizörlerde çalışması için önerildi; 2011'de Linux'tan kaldırıldı. Bugün yarı sanallaştırma **sürücülerde** yaşar: `virtio` disk/ağ, paravirt saat, Hyper-V aydınlatmaları.

Günümüz bulutlarında CPU ve bellek donanım destekli tam sanallaştırılır; G/Ç ise virtio ya da SR-IOV ile doğrudan aygıt ataması kullanır.

Bellek Sanallaştırma: Gölge Tablo ve EPT



gölge sayfa tablosu: GVA → HPA doğrudan (yazılımla güncel tutulur)

TLB kaçırmada 2B tablo yürüyüşü: 4 düzey → $(4+1)^2 - 1 = 24$ bellek erişimi · 5 düzey → 35

- **Gölge sayfa tabloları**: hipervizör, misafirin tablolarını salt okunur yapar; her değişiklikte tuzak kurup GVA → HPA kopyasını günceller. Çok VM çıkışı.
- **EPT** (Intel, 2008) / **NPT** (AMD): donanım iki tabloyu birlikte yürür; misafir kendi tablosunu tuzaksız değiştirir.
- TLB kaçırma pahalılaşır → büyük sayfalar (2 MiB / 1 GiB) ve VM etiketli TLB (VPID/ASID) kullanılır.
- Aşırı tahsis: **balon sürücüsü** misafirden bellek geri alır; **KSM** aynı içerikli sayfaları birleştirir (yazınca kopyala).

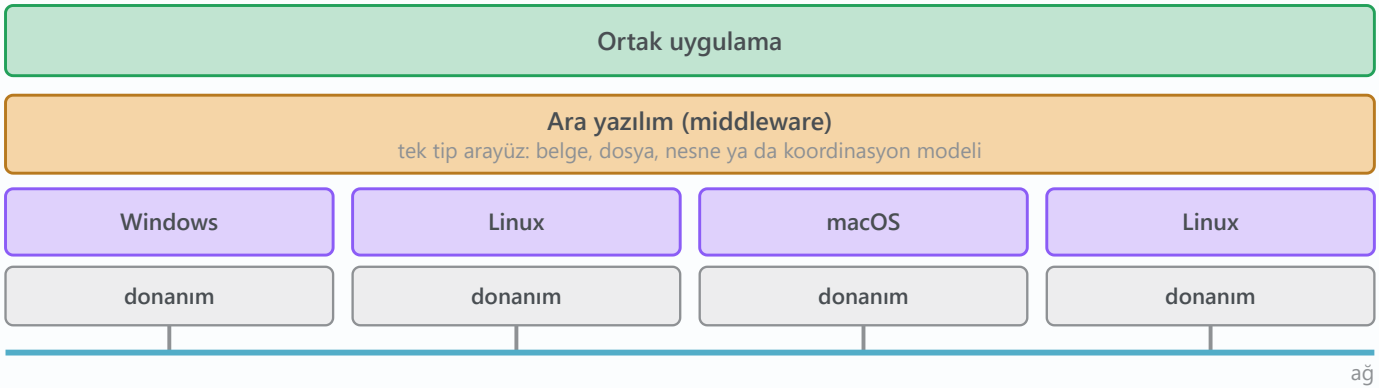
Konteynerler ve Sanal Makineler



	Sanal makine	**Konteyner**
Yalıtım sınırı	donanım düzeyi, ayrı çekirdek	aynı çekirdek: namespace, cgroup, seccomp
Başlama / ek yük	saniyeler, GB bellek	milisaniyeler, MB bellek
Farklı İS	evet (Linux üstünde Windows)	hayır (aynı çekirdek ailesi)
Örnek	KVM, Hyper-V, ESXi	Docker, Podman, Kubernetes pod'ları

Arada: milisaniyelerde açılan Firecracker mikro-VM'leri (AWS Lambda), gVisor ve Kata Containers, konteyner kolaylığını VM yalıtımıyla birleştirir. WSL 2 de hafif bir Hyper-V VM'inde gerçek Linux çekirdeği çalıştırır; Windows'taki Linux konteynerleri bu VM'de koşar.

Dağıtık Sistemler ve Ara Yazılım

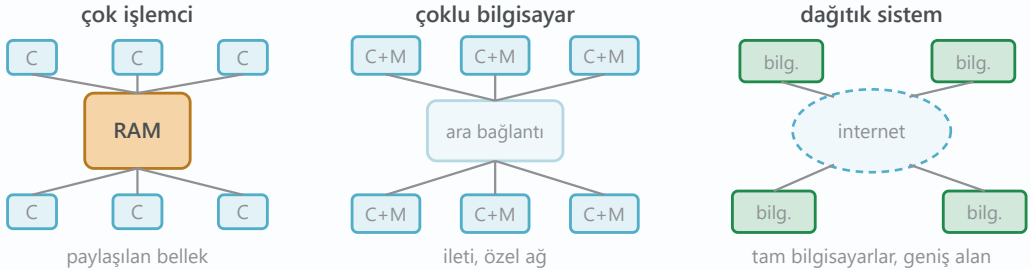


- **Dağıtık sistem**: ağ üzerinden haberleşen özerk bilgisayarlar; her biri kendi işletim sistemini, belleğini ve kaynaklarını yönetir, kullanıcıya tek bir tutarlı sistem gibi görünmeye çalışır.
- Düğümler farklı donanım ve işletim sistemi çalıştırabilir, farklı kurumlara ait olabilir, dünyaya yayılabilir. Ortaklık **ara yazılım** katmanıyla sağlanır.
- Dört paradigma: **belge** (Web), **dosya** (NFS, SMB), **nesne** (CORBA, RMI, gRPC), **koordinasyon** (Linda, yayınla-abone ol).

Gerçek dağıtık işletim sistemleri (Amoeba, Plan 9) araştırmada kaldı; bugün her düğüm kendi işletim sistemini çalıştırır, dağıtımı ara yazılım ve küme yöneticileri (Kubernetes) yapar.

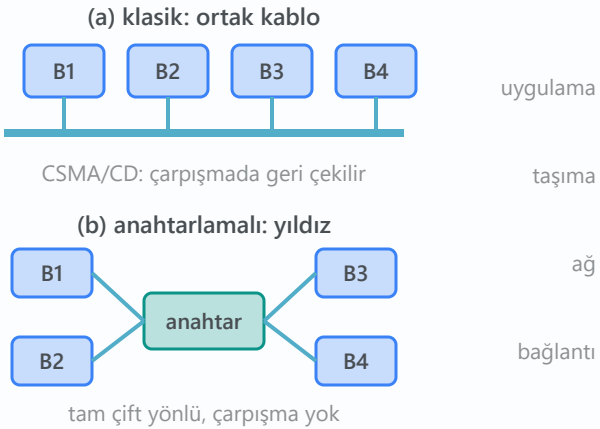
Üç Sistem Türünün Karşılaştırması

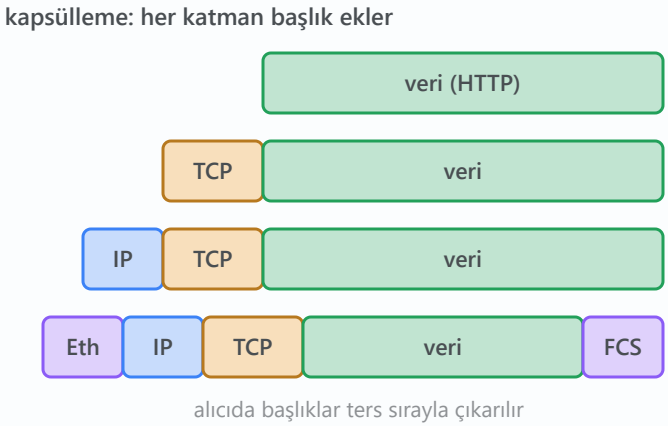
Özellik	Çok işlemci	Çoklu bilgisayar	Dağıtık sistem
Düğüm yapısı	CPU	CPU, RAM, ağ arayüzü	tam bilgisayar
Çevre birimleri	hepsi paylaşılır	paylaşılır (belki disk hariç)	her düğümde tam set
Konum	aynı raf	aynı oda	dünya çapında olabilir
Düğümler arası iletişim	paylaşılan RAM	özel ara bağlantı	geleneksel ağ
İşletim sistemi	bir, paylaşılan	çok, aynı	çok, farklı olabilir
Dosya sistemi	bir, paylaşılan	bir, paylaşılan	her düğümün kendi
Yönetim	tek kurum	tek kurum	birçok kurum



Tanenbaum (MOS 4e, Şekil 8-29). Dağıtık sistemde çevre birimleri paylaşılmaz: her düğüm **kendi tam setine** sahiptir. Çoklu bilgisayarda da disk her zaman paylaşılmaz.

Ağ Donanımı ve Kapsülleme





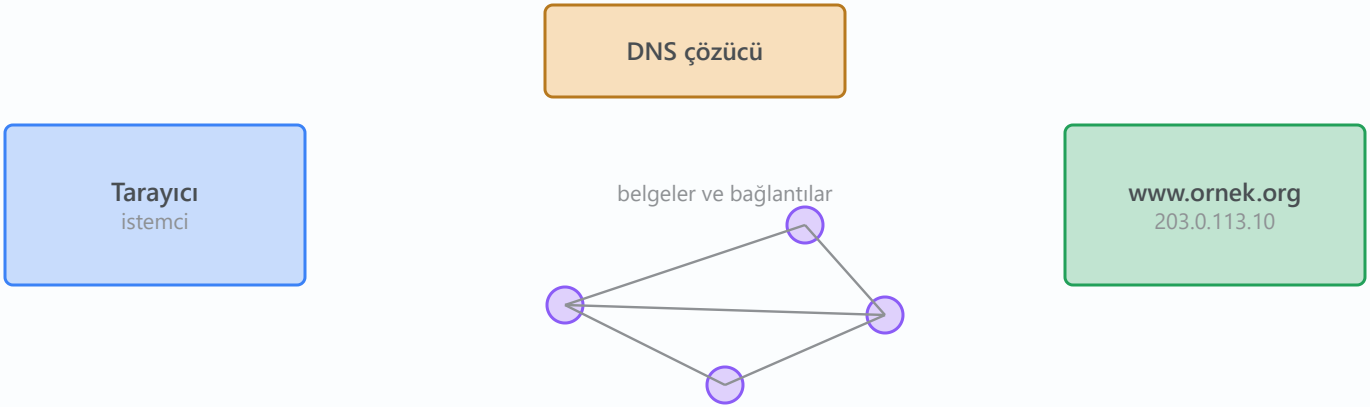
- Klasik Ethernet (1976) tek kabloyu paylaşıyordu; bugün her bilgisayar bir **anahtara** bağlı, bağlantılar 1–400 Gb/s ve tam çift yönlü.
- LAN'lar yönlendiricilerle birbirine, yerel ağlar ISS'lere ve **İnternet omurgasına** (yüksek hızlı fiber hatlar, değişim noktaları) bağlanır.

Ağ Protokolleri ve Hizmet Türleri

Protokol	Katman	Hizmet	Not
IP (v4/v6)	ağ	bağlantısız, güvenilmez	adresleme ve yönlendirme
TCP	taşıma	bağlantı yönelimli, güvenilir bayt akışı	onay, yeniden iletim, akış ve tıkanıklık denetimi
UDP	taşıma	bağlantısız, güvenilmez datagram	sağlama toplamı var; yeniden iletim yok. DNS, oyun, ses
QUIC	taşıma (UDP üstü)	bağlantı yönelimli, güvenilir, şifreli	HTTP/3 (2022) bunu kullanır
DNS	uygulama	ad → IP adresi	hiyerarşik, dağıtık, önbellekli
HTTP/HTTPS	uygulama	Web belgeleri	HTTPS = HTTP + TLS, port 443
SMTP	uygulama	e-posta aktarımı	sunucular arası; okuma IMAP ile
FTP → SFTP	uygulama	dosya aktarımı	FTP şifresizdir; yerini SFTP/HTTPS aldı
SNMP	uygulama	ağ aygıtı izleme	v3 kimlik doğrulama ve şifreleme ekler

Güvenilir hizmet: teslim ve sıra garanti, alındı onayı (ACK). Bağlantı yönelimli: önce kurulum, sonra veri, sonra sonlandırma; bağlantısız hizmet kurulum maliyeti olmadan her paketi bağımsız gönderir. UDP sağlama toplamıyla bozuk paketi **algılar** ama düzeltmez ya da yeniden göndermez.

Belge Tabanlı Ara Yazılım: Web

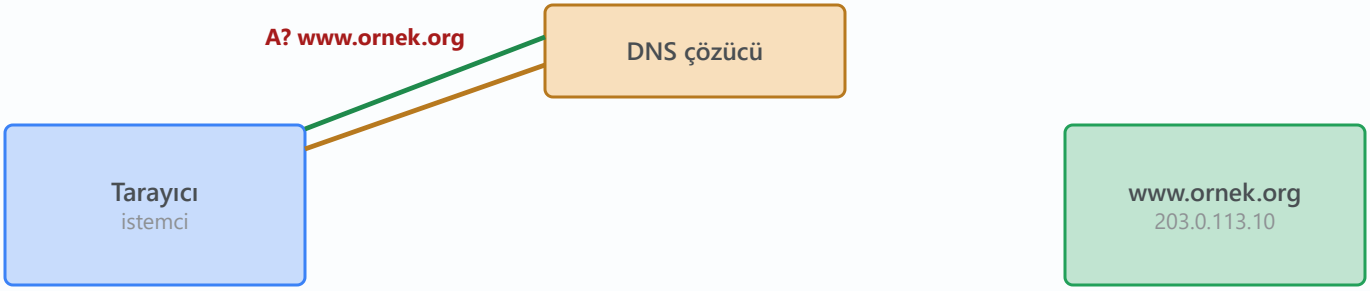


Kullanıcı `www.ornek.org/index.html` adresini yazar. Web: birbirine bağlı belgelerden oluşan dev bir **yönlendirilmiş çizge**.

Adım 1 / 5

Port 80 ve her nesne için yeni TCP bağlantısı açıp kapatma HTTP/1.0 dönemine aitti; bugün varsayılan HTTPS (443), kalıcı bağlantılar ve HTTP/2–3 çoğullamasıdır.

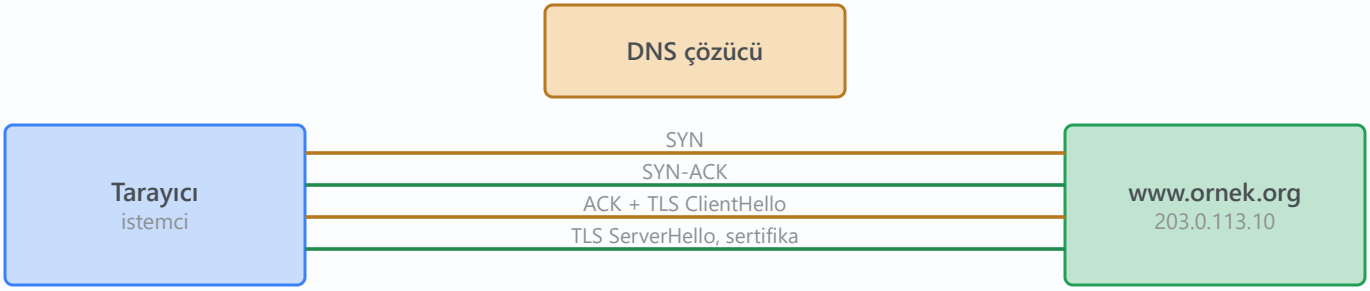
Belge Tabanlı Ara Yazılım: Web



1–2: Tarayıcı (işletim sisteminin çözücüsü üzerinden) **DNS**'e www.ornek.org'un adresini sorar; yanıt: 203.0.113.10 (önbellekte tutulur).

Port 80 ve her nesne için yeni TCP bağlantısı açıp kapatma HTTP/1.0 dönemine aitti; bugün varsayılan HTTPS (443), kalıcı bağlantılar ve HTTP/2–3 çoğullamasıdır.

Belge Tabanlı Ara Yazılım: Web



3: Sunucunun **443** numaralı portuna TCP bağlantısı ve **TLS el sıkışması** (HTTP/3'te ikisi tek adımda, QUIC ile).

Port 80 ve her nesne için yeni TCP bağlantısı açıp kapatma HTTP/1.0 dönemine aitti; bugün varsayılan HTTPS (443), kalıcı bağlantılar ve HTTP/2–3 çoğullamasıdır.

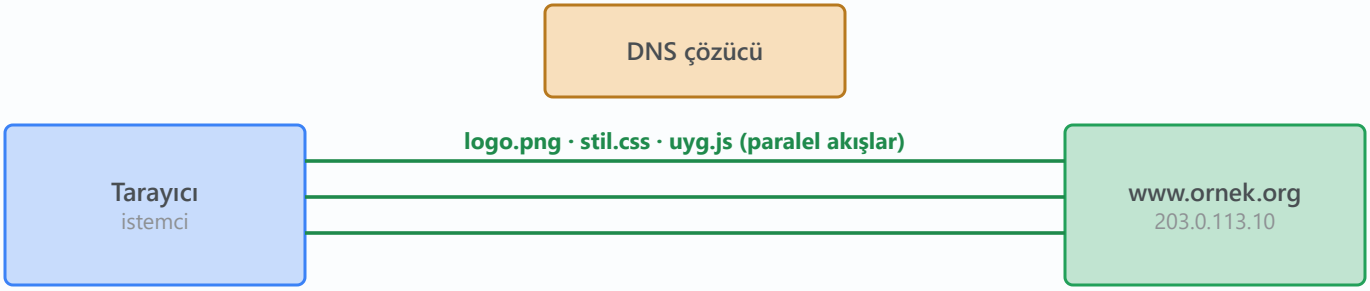
Belge Tabanlı Ara Yazılım: Web



4–5: `GET /index.html` isteği; sunucu HTML'yi döndürür, tarayıcı metni görüntülemeye başlar.

Port 80 ve her nesne için yeni TCP bağlantısı açıp kapatma HTTP/1.0 dönemine aitti; bugün varsayılan HTTPS (443), kalıcı bağlantılar ve HTTP/2–3 çoğullamasıdır.

Belge Tabanlı Ara Yazılım: Web

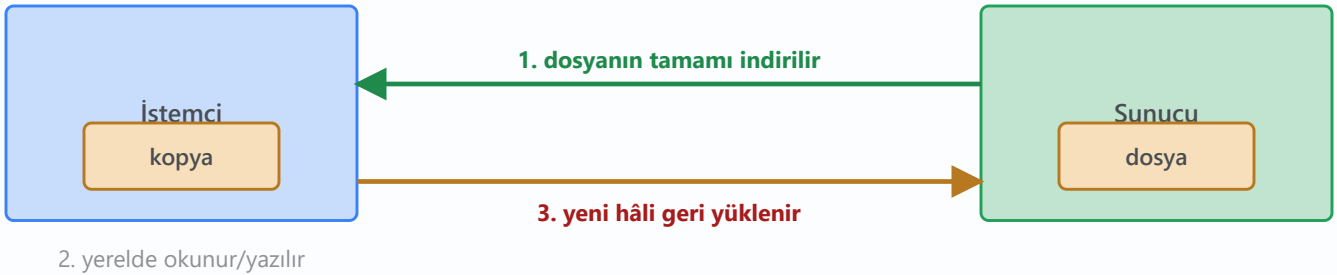


6: HTML'deki resim, CSS ve betikler **aynı bağlantı** üzerinden, HTTP/2–3'te paralel akışlarla alınır; bağlantı bir süre açık kalır.

Adım 5 / 5

Port 80 ve her nesne için yeni TCP bağlantısı açıp kapatma HTTP/1.0 dönemine aitti; bugün varsayılan HTTPS (443), kalıcı bağlantılar ve HTTP/2–3 çoğullamasıdır.

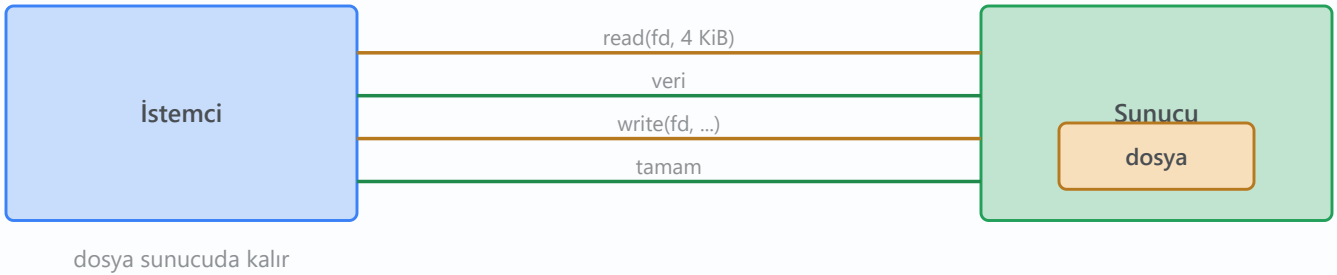
Dosya Sistemi Tabanlı Ara Yazılım



Yükleme/indirme modeli: dosya bütün olarak istemciye kopyalanır, kullanılır, geri gönderilir (FTP benzeri; eski AFS). Basit ama büyük dosyada pahalı.

Adlandırma: `makine:/yol` (konum şeffaf değil) → uzak sistemi yerel ağaca bağlama → tüm makinelerde aynı tek ad alanı (konumdan bağımsızlık). Bugünkü örnekler: NFSv4, SMB 3, nesne depolama (S3).

Dosya Sistemi Tabanlı Ara Yazılım



Uzaktan erişim modeli: dosya sunucuda kalır; istemci `read` / `write` isteklerini ağ üzerinden gönderir (NFS, SMB).

Adlandırma: `makine:/yol` (konum şeffaf değil) → uzak sistemi yerel ağaca bağlama → tüm makinelerde aynı tek ad alanı (konumdan bağımsızlık). Bugünkü örnekler: NFSv4, SMB 3, nesne depolama (S3).

Dosya Sistemi Tabanlı Ara Yazılım

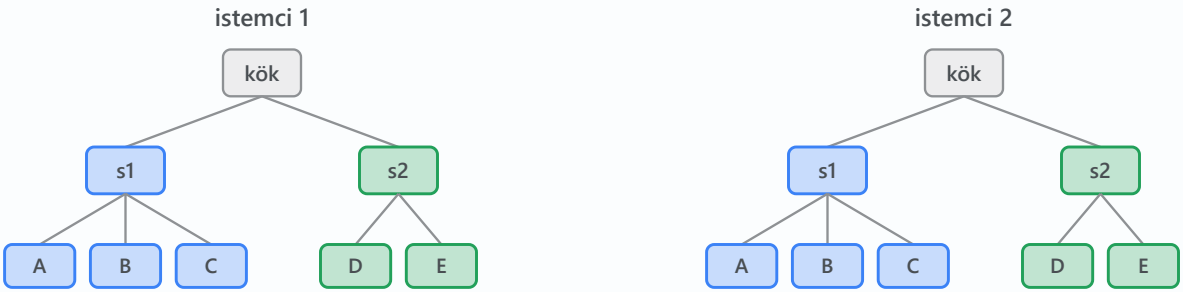


kare: dizin · daire: dosya

İki dosya sunucusu, her biri kendi dizin ağacıyla.

Adlandırma: `makine:/yol` (konum şeffaf değil) → uzak sistemi yerel ağaca bağlama → tüm makinelerde aynı tek ad alanı (konumdan bağımsızlık). Bugünkü örnekler: NFSv4, SMB 3, nesne depolama (S3).

Dosya Sistemi Tabanlı Ara Yazılım



aynı ad alanı: /s1/A her istemcide aynı dosya

Tüm istemciler sunucuları aynı yere bağlar (mount) → **tek, ortak ad alanı**.

Adlandırma: `makine:/yol` (konum şeffaf değil) → uzak sistemi yerel ağaca bağlama → tüm makinelerde aynı tek ad alanı (konumdan bağımsızlık). Bugünkü örnekler: NFSv4, SMB 3, nesne depolama (S3).

Dosya Sistemi Tabanlı Ara Yazılım

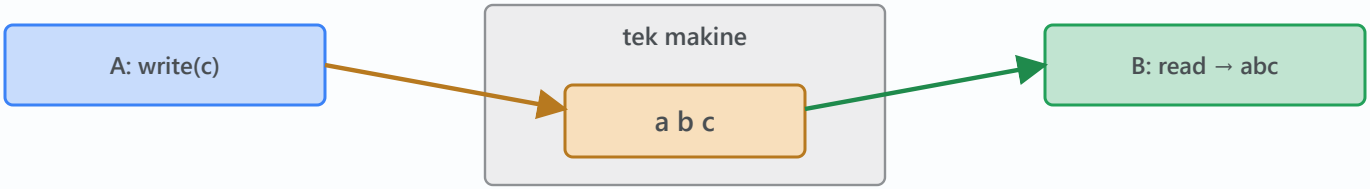


istemci 2'de /s1 yok, /s2/D ise iki istemcide aynı

İstemciler farklı şeyler bağlarsa her biri farklı görünüm görür; aynı yol farklı istemcide farklı dosya olabilir.

Adlandırma: `makine:/yol` (konum şeffaf değil) → uzak sistemi yerel ağaca bağlama → tüm makinelerde aynı tek ad alanı (konumdan bağımsızlık). Bugünkü örnekler: NFSv4, SMB 3, nesne depolama (S3).

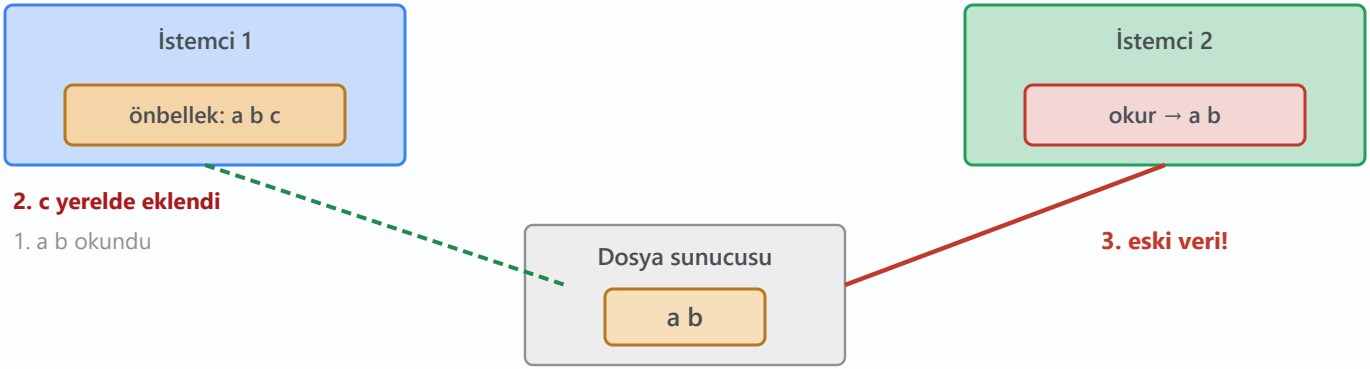
Dosya Paylaşımının Anlamı



her okuma en son yazmayı görür: UNIX semantiği

Tek işlemcide işlemler **sıralıdır**: A “c” ekler, ardından B okur ve “abc” görür (sıralı tutarlılık).

Dosya Paylaşımının Anlamı



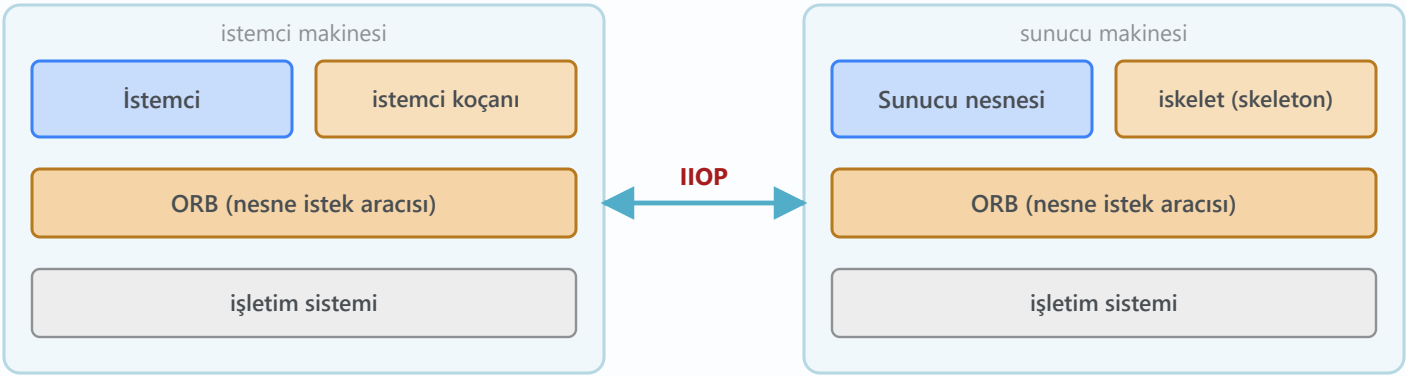
Dağıtık sistemde istemci 1 dosyayı önbelleğinde değiştirir; sunucu bunu henüz bilmez. İstemci 2 sunucudan okur ve **eski** "ab"yi görür.

Dosya Paylaşımının Anlamı

Yaklaşım	Kural	Örnek
UNIX semantiği	her yazma hemen herkese görünür	tek makine; önbelleksiz uzak erişim
Oturum semantiği	değişiklik dosya kapanınca görünür	AFS; NFS kapat-aç tutarlılığı
Değişmez dosyalar	dosya yazılamaz, yenisi oluşturulur	nesne depolama, Git nesneleri
İşlemler (transactions)	hep ya da hiç, sıralanabilir	veritabanları

Çözümler tutarlılık ile performans arasında takas yapar.

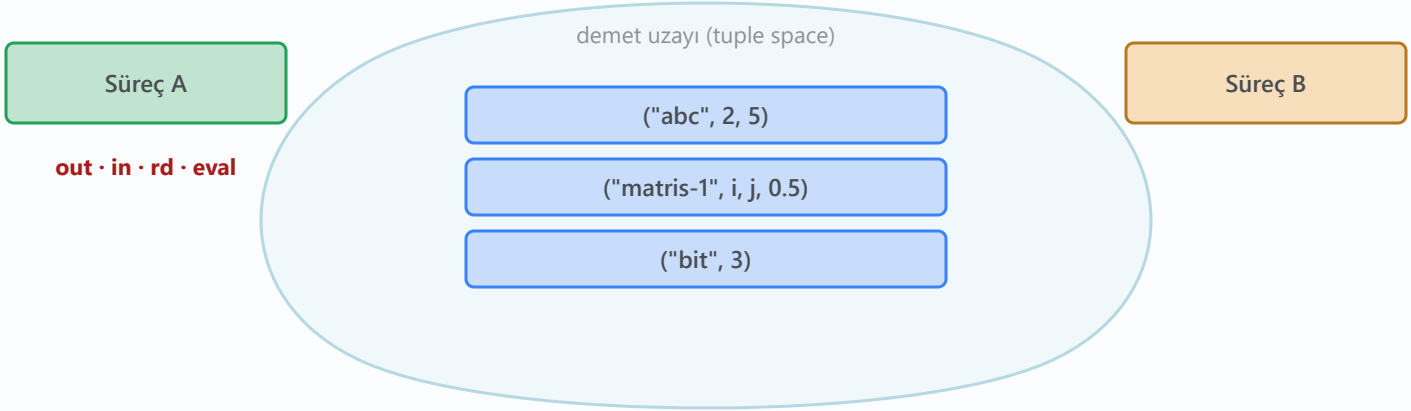
Nesne Tabanlı Ara Yazılım: CORBA



- **IDL**: arayüzler dilden bağımsız tanımlanır; derleyici istemci koçanı ve sunucu iskeletini üretir.
- **ORB**: nesnenin yerini bulur, çağrıyı paketler, IIOP ile iletir: **ağ şeffaflığı**.
- **SII** (statik çağırma): derleme zamanında IDL'den üretilen koçanlarla; hızlı, tür denetimli.
- **DII** (dinamik çağırma): arayüz çalışma anında keşfedilir (arayüz deposu).

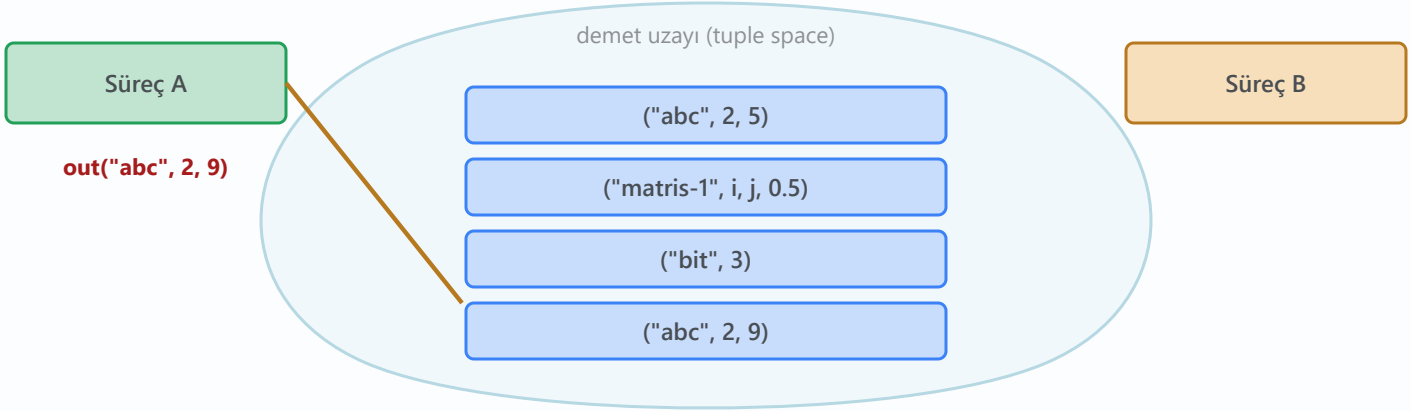
SII ve DII ayrı mekanizmalardır; SII, DII'nin derlenmiş sürümü değildir. CORBA (OMG, 1991) bugün eski sistemlerde yaşar; yeni sistemlerde gRPC/Protobuf, REST ve Java RMI tercih edilir.

Koordinasyon Tabanlı Ara Yazılım: Linda



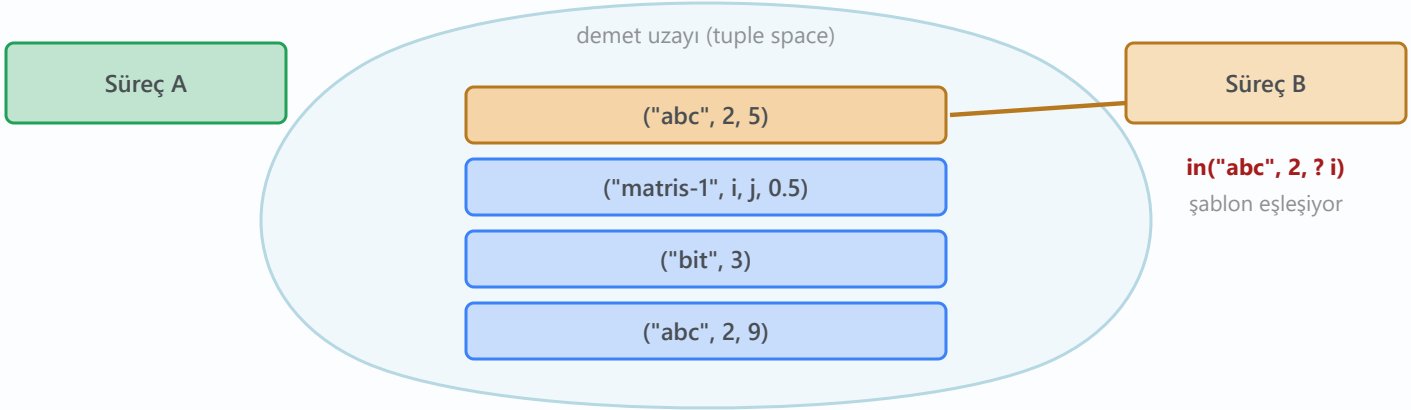
Linda (Gelernter, 1985): süreçler birbirini tanımaz; ortak, kalıcı bir **demet uzayı** üzerinden haberleşir. Demet: türlü alanlardan oluşan kayıt.

Koordinasyon Tabanlı Ara Yazılım: Linda



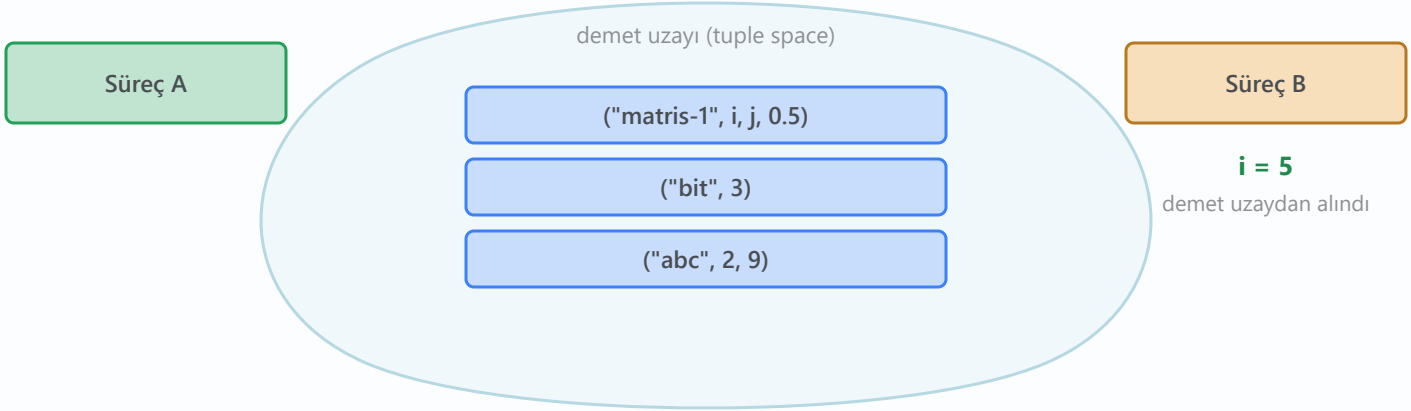
`out` demeti uzaya koyar (bloke etmez).

Koordinasyon Tabanlı Ara Yazılım: Linda



`in` bir **şablonla** arar: alan sayısı aynı, karşılıklı alanlar aynı türde, sabit alanlar eşit olmalı. `? i` biçimsel parametredir. Birden çok eşleşmede biri rastgele seçilir.

Koordinasyon Tabanlı Ara Yazılım: Linda



Eşleşen demet uzaydan **çıkarılır**, `i = 5` atanır. Eşleşme yoksa `in` bloke olur. `rd` okur ama çıkarmaz; `eval` yeni süreç başlatır. Senkronizasyon da böylece kendiliğinden gelir.

Linda in(): Kod İzleme

```
▶ Demet *in_ara(Sablon *s) {
▶   for (Demet *d = uzay; d; d = d->snr) {
3     if (d->n != s->n) continue;
4     int ok = 1;
5     for (int k = 0; ok && k < s->n; k++)
6       if (d->tur[k] != s->tur[k] ||
7           (s->sabit[k] && !esit(d, s, k)))
8         ok = 0;          /* alan uymadı */
9     if (ok) { cikar(d); return d; }
10  }
11  return NULL;           /* bloke ol */
12 }
```

şablon: `"abc"` | `2` | `? int i`

demet uzayı:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d4	"abc"	2	5
d5	"abc"	2	9

Uzayda 5 demet var. Süreç `in("abc", 2, ? int i)` çağırıyor: 3 alanlı, ilk iki alan sabit, üçüncü tam sayı türünden biçimsel parametre.

Adım 1 / 11

DEĞİŞKENLER

`d = d1` `k = –` `ok = –` `i = –`

ÇIKTI

Linda in(): Kod İzleme

```
1  Demet *in_ara(Sablon *s) {
2    for (Demet *d = uzay; d; d = d->snr) {
▶      if (d->n != s->n) continue;
4      int ok = 1;
5      for (int k = 0; ok && k < s->n; k++)
6        if (d->tur[k] != s->tur[k] ||
7            (s->sabit[k] && !esit(d, s, k)))
8          ok = 0;          /* alan uymadı */
9      if (ok) { cikar(d); return d; }
10   }
11   return NULL;           /* bloke ol */
12 }
```

şablon: `"abc"` | `2` | `? int i`

demet uzayı:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d4	"abc"	2	5
d5	"abc"	2	9

d1: 2 alanlı, şablon 3 alanlı → hemen atlanır (`continue`).

Adım 2 / 11

DEĞİŞKENLER

`d = d1` `k = –` `ok = –` `i = –`

ÇIKTI

Linda in(): Kod İzleme

```
1  Demet *in_ara(Sablon *s) {
2    for (Demet *d = uzay; d; d = d->snr) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->tur[k] != s->tur[k] ||
▶            (s->sabit[k] && !esit(d, s, k)))
▶          ok = 0;          /* alan uymadı */
9      if (ok) { cikar(d); return d; }
10   }
11   return NULL;           /* bloke ol */
12 }
```

şablon: `"abc"` | `2` | `? int i`

demet uzayı:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d4	"abc"	2	5
d5	"abc"	2	9

d2, alan 0: "xyz" ≠ "abc" → ok = 0, sonraki demete geçilir.

Adım 3 / 11

DEĞİŞKENLER

`d = d2` `k = 0` `ok = 0` `i = –`

ÇIKTI

Linda in(): Kod İzleme

```
1  Demet *in_ara(Sablon *s) {
2    for (Demet *d = uzay; d; d = d->snr) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->tur[k] != s->tur[k] ||
▶            (s->sabit[k] && !esit(d, s, k)))
▶          ok = 0;          /* alan uymadı */
9      if (ok) { cikar(d); return d; }
10   }
11   return NULL;           /* bloke ol */
12 }
```

şablon: `"abc"` | `2` | `? int i`

demet uzayı:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d4	"abc"	2	5
d5	"abc"	2	9

d3, alan 1: 1 ≠ 2 → ok = 0, sonraki demete geçilir.

Adım 4 / 11

DEĞİŞKENLER

`d = d3` `k = 1` `ok = 0` `i = –`

ÇIKTI

Linda in(): Kod İzleme

```
1  Demet *in_ara(Sablon *s) {
2    for (Demet *d = uzay; d; d = d->snr) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->tur[k] != s->tur[k] ||
▶            (s->sabit[k] && !esit(d, s, k)))
8          ok = 0;          /* alan uymadı */
▶      if (ok) { cikar(d); return d; }
10   }
11   return NULL;           /* bloke ol */
12 }
```

şablon: `"abc"` | `2` | `? int i`

demet uzayı:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d4	"abc"	2	5
d5	"abc"	2	9

d4: alan sayısı ve türler uyuyor, sabitler eşit; biçimsel alan i = **5** alır. Demet uzaydan **çıkarılır**.

Adım 5 / 11

DEĞİŞKENLER

`d = d4` `k = 3` `ok = 1` `i = 5`

ÇIKTI

```
1. in: i = 5
```

Linda in(): Kod İzleme

```
1  Demet *in_ara(Sablon *s) {
2    for (Demet *d = uzay; d; d = d->snr) {
3      if (d->n != s->n) continue;
4      int ok = 1;
5      for (int k = 0; ok && k < s->n; k++)
6        if (d->tur[k] != s->tur[k] ||
7            (s->sabit[k] && !esit(d, s, k)))
8          ok = 0;          /* alan uymadı */
▶      if (ok) { cikar(d); return d; }
10   }
11   return NULL;          /* bloke ol */
12 }
```

şablon: | "abc" | 2 | ? int i

demet uzayı:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d5	"abc"	2	9

d4 artık uzayda yok; d5 de eşleşirdi ama ilk bulunan alındı (Linda hangisinin seçileceğini tanımlamaz). Aynı çağrı yeniden yapılıyor.

Adım 6 / 11

DEĞİŞKENLER

`d = d4` `k = 3` `ok = 1` `i = 5`

ÇIKTI

```
1. in: i = 5
```

Linda in(): Kod İzleme

```
1  Demet *in_ara(Sablon *s) {
2    for (Demet *d = uzay; d; d = d->snr) {
▶      if (d->n != s->n) continue;
4      int ok = 1;
5      for (int k = 0; ok && k < s->n; k++)
6        if (d->tur[k] != s->tur[k] ||
7            (s->sabit[k] && !esit(d, s, k)))
8          ok = 0;          /* alan uymadı */
9      if (ok) { cikar(d); return d; }
10   }
11   return NULL;           /* bloke ol */
12 }
```

şablon: `"abc"` | `2` | `? int i`

demet uzayı:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d5	"abc"	2	9

d1: 2 alanlı, şablon 3 alanlı → hemen atlanır (`continue`).

Adım 7 / 11

DEĞİŞKENLER

`d = d1` `k = –` `ok = –` `i = –`

ÇIKTI

```
1. in: i = 5
```

Linda in(): Kod İzleme

```
1  Demet *in_ara(Sablon *s) {
2    for (Demet *d = uzay; d; d = d->snr) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->tur[k] != s->tur[k] ||
▶            (s->sabit[k] && !esit(d, s, k)))
▶          ok = 0;          /* alan uymadı */
9      if (ok) { cikar(d); return d; }
10   }
11   return NULL;           /* bloke ol */
12 }
```

şablon: "abc" | 2 | ? int i

demet uzayı:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d5	"abc"	2	9

d2, alan 0: "xyz" ≠ "abc" → ok = 0, sonraki demete geçilir.

Adım 8 / 11

DEĞİŞKENLER

d = d2 | k = 0 | ok = 0 | i = –

ÇIKTI

```
1. in: i = 5
```

Linda in(): Kod İzleme

```
1  Demet *in_ara(Sablon *s) {
2    for (Demet *d = uzay; d; d = d->snr) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->tur[k] != s->tur[k] ||
▶            (s->sabit[k] && !esit(d, s, k)))
▶          ok = 0;          /* alan uymadı */
9      if (ok) { cikar(d); return d; }
10   }
11   return NULL;           /* bloke ol */
12 }
```

şablon: `"abc"` | `2` | `? int i`

demet uzayı:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d5	"abc"	2	9

d3, alan 1: 1 ≠ 2 → ok = 0, sonraki demete geçilir.

Adım 9 / 11

DEĞİŞKENLER

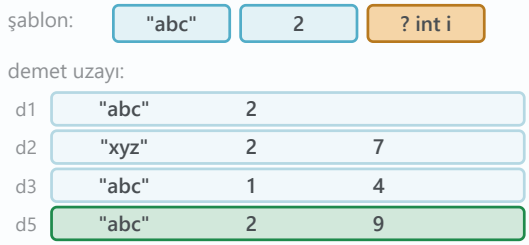
`d = d3` `k = 1` `ok = 0` `i = –`

ÇIKTI

```
1. in: i = 5
```

Linda in(): Kod İzleme

```
1  Demet *in_ara(Sablon *s) {
2    for (Demet *d = uzay; d; d = d->snr) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->tur[k] != s->tur[k] ||
▶            (s->sabit[k] && !esit(d, s, k)))
8          ok = 0;          /* alan uymadı */
▶      if (ok) { cikar(d); return d; }
10   }
11   return NULL;           /* bloke ol */
12 }
```



d5: alan sayısı ve türler uyuyor, sabitler eşit; biçimsel alan i = **9** alır. Demet uzaydan **çıkarılır**.

Adım 10 / 11

DEĞİŞKENLER

`d = d5` `k = 3` `ok = 1` `i = 9`

ÇIKTI

```
1. in: i = 5
2. in: i = 9
```

Linda in(): Kod İzleme

```
1  Demet *in_ara(Sablon *s) {
2    for (Demet *d = uzay; d; d = d->snr) {
3      if (d->n != s->n) continue;
4      int ok = 1;
5      for (int k = 0; ok && k < s->n; k++)
6        if (d->tur[k] != s->tur[k] ||
7            (s->sabit[k] && !esit(d, s, k)))
8          ok = 0;          /* alan uymadı */
9      if (ok) { cikar(d); return d; }
10   }
▶    return NULL;          /* bloke ol */
12 }
```

şablon: | "abc" | 2 | ? int i |

demet uzayı:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4

Üçüncü bir `in("abc", 2, ? int i)` eşleşme bulamaz ve NULL döner: süreç, biri uyan bir demeti `out` ile koyana kadar **bloke** olur.

Adım 11 / 11

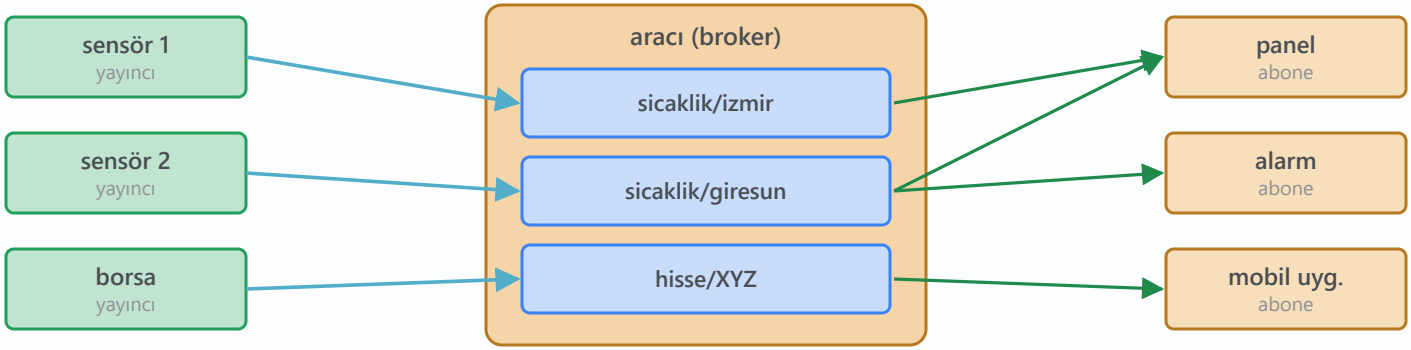
DEĞİŞKENLER

`d = –` `k = –` `ok = –` `i = 9`

ÇIKTI

```
1. in: i = 5
2. in: i = 9
```

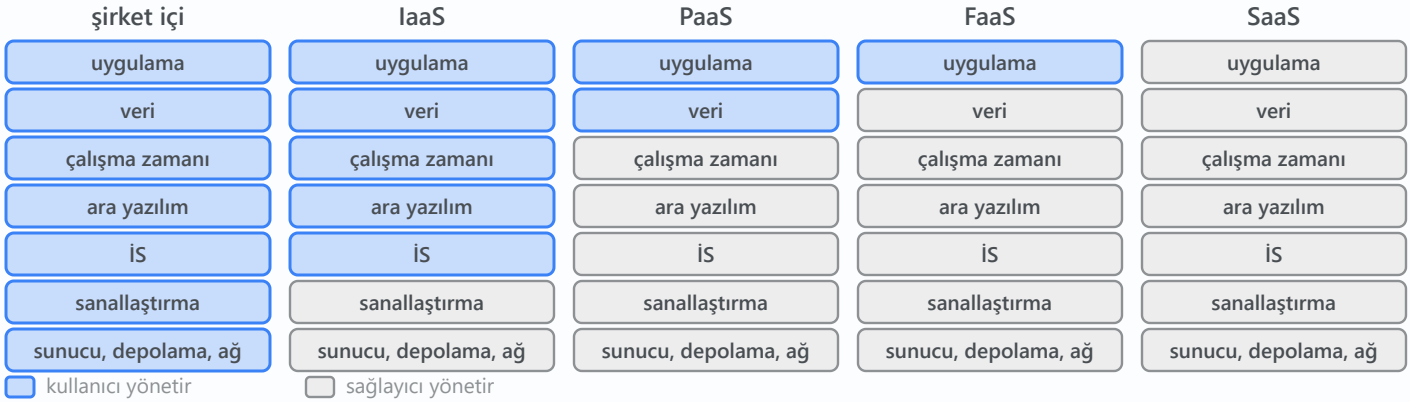
Yayınla-Abone Ol ve Jini



- Yayıncı bir **konuya** yayınlar, abone ilgilendiği konulara abone olur; taraflar birbirinin kimliğini bilmez: **zaman ve uzayda ayrıklık**.
- Abone ve konu dinamik eklenir → ölçeklenebilir, esnek. Teslim garantisi aracının ayarına bağlıdır (en çok bir kez / en az bir kez / kalıcı kuyruk).
- Bugün: MQTT (IoT), Apache Kafka, RabbitMQ, bulut kuyrukları.

- **Jini** (Sun, 1998): Java aygıtları ağda birbirini önceden yapılandırma olmadan **keşfeder**, arama hizmetine kaydolur, Java RMI ile kullanır; kiralama (lease) ile kayıtlar kendiliğinden düşer.
- Apache River adıyla sürdü ama bugün etkin değil; fikirleri servis keşfinde (mDNS/DNS-SD, Consul, Kubernetes servisleri) yaşıyor.

Bulut ve Kümeler



- **Küme**: hızlı ağla bağlı, tek yönetimdeki sunucular (HPC, veri merkezi). **Bulut**: bu kaynakların ağ üzerinden, isteğe bağlı ve kullandıkça öde biçiminde sunulması (AWS, Azure, Google Cloud).
- Altyapı: KVM/Nitro sanallaştırma, konteynerler ve **Kubernetes** (pod'ları düğümlere çizelgeler, çökeni yeniden başlatır: dağıtık işletim sisteminin bugünkü karşılığı).
- Bu bölümün tüm konuları burada birleşir: NUMA sunucular, sanal makineler, RDMA ağları, yük dengeleme, yayınla-abone ol kuyrukları.

Özet

Donanım

UMA (veri yolu, çapraz çubuk n^2, omega $(n/2)\log_2 n$) ve NUMA; tutarlılık için dinleme ya da dizin.

Önbellek

MESI geçişleri; yazmalar geçersiz kılma üretir; yanlış paylaşım satır düzeyinde olur, dolgu ile çözülür.

İşletim sistemi ve kilitler

Ayrı işletim sistemi, usta-köle, SMP; TSL için veri yolu/satır kilidi, TTAS, bilet ve MCS; kısa beklemede dönme, uzunda uyuma.

Çizelgeleme

Ortak kuyrukta tek kilit ve soğuk önbellek; birbirine bağlı iş parçacıkları için alan paylaşımı ve çete çizelgeleme.

Çoklu bilgisayar

Topoloji çapı, sakla-ilet ve kesip geç, bloke eden/etmeyen gönderim, RPC, DSM, yük dengeleme.

Sanallaştırma ve dağıtık

Tip 1/2, VT-x, EPT, konteyner; ara yazılım: belge, dosya, nesne, koordinasyon; bulut.

ÖZET

Kontrol Soruları

1. 64 × 64 çapraz çubukta kaç kesişim, 64 girişli omega ağında kaç 2 × 2 anahtar var?
2. Omega ağında CPU 011 → bellek 110 isteği hangi anahtar ve çıkışlardan geçer?
3. Yerel 80 ns, uzak 140 ns, erişimlerin %75'i yerel: ortalama erişim süresi?
4. MESI'de E ve S durumundaki bir satıra yazmanın veri yolu açısından farkı nedir?
5. İki iş parçacığının ayrı sayaçları neden birbirini yavaşlatabilir? Çözüm?
6. Bilet kilidi saf TSL döngüsünden hangi iki açıdan üstündür?
7. Çete çizelgelemenin iki koşulu nedir, hangi sorunu çözer?
8. 256 düğümlü hiperküpün çapı nedir? 16 × 16 simidin çapı nedir?
9. 4 düzeyli tablolar ve EPT ile TLB kaçırması en kötü kaç bellek erişimi ister?
10. `in("abc", ? x, 5)` (x int), `("abc", 2, 5)` ve `("abc", 2.0, 5)` demetlerinden hangisiyle eşleşir?

▶ **Cevaplar**

Kaynaklar

- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 8: Multiple Processor Systems; Bölüm 7: Virtualization and the Cloud.
- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 5.5 (Multi-Processor Scheduling), 18 (Virtual Machines), 19 (Networks and Distributed Systems).
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Multiprocessor Scheduling", "Locks", "Distributed Systems", "Sun's NFS", "AFS", Ek: "Virtual Machine Monitors".
- J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6. baskı, Morgan Kaufmann, 2017 — Bölüm 5: Thread-Level Parallelism (tutarlılık, MESI, dizin protokolleri).
- J. M. Mellor-Crummey, M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", ACM TOCS, 1991; A. D. Birrell, B. J. Nelson, "Implementing Remote Procedure Calls", ACM TOCS, 1984; G. J. Popek, R. P. Goldberg, CACM, 1974.
- Linux çekirdek belgeleri: "CFS/EEVDF Scheduler", "Scheduler Domains", "NUMA Memory Policy", "Core Scheduling"; Intel SDM Cilt 3 (VMX, EPT).

Son

Bölüm 12: Çok İşlemcili Sistemler

Sonraki bölüm: Güvenlik · Terim sözlüğü