Bölüm 4: İş Parçacıkları

İşletim Sistemleri

Süreç ve iş parçacığı | Paylaşılan ve özel durum | Çok iş parçacıklı sunucu

Eş zamanlılık ve paralellik | Amdahl yasası | Kullanıcı ve çekirdek iş parçacıkları

1:1, N:1, M:N modelleri | Pthreads ve Java | Havuzlar ve fork-join | OpenMP ve GCD

Yarış durumu | TLS, iptal, sinyaller

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_04_Is_Parcaciklari.html

GİRİŞ

Öğrenme Çıktıları

Kavram

İş parçacığını süreçten ayırmak; hangi durumun paylaşıldığını, hangisinin iş parçacığına özel olduğunu söylemek.

Performans

Eş zamanlılık ile paralelliği ayırt etmek ve Amdahl yasasıyla hızlanma sınırını hesaplamak.

Modeller

Kullanıcı ve çekirdek düzeyi iş parçacıklarını, N:1, 1:1 ve M:N eşleme modellerini karşılaştırmak.

Programlama

Pthreads ve Java ile iş parçacığı oluşturmak, beklemek; havuz, fork-join, OpenMP ve GCD gibi örtük yaklaşımları tanımak.

Sorunlar

Yarış durumu, yeniden girilemeyen kod, TLS, iptal, sinyaller ve `fork()` davranışını açıklamak.

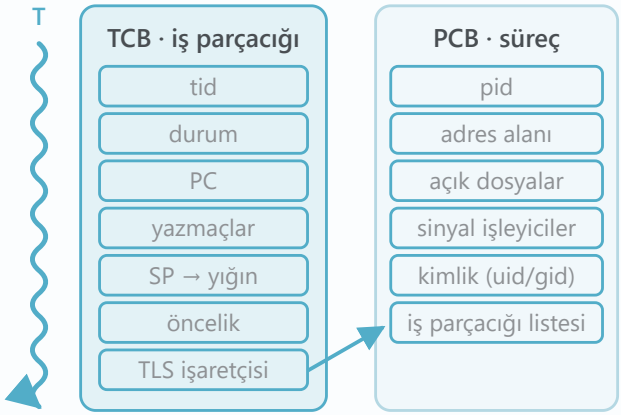
Gerçek sistemler

Linux `clone()`, Java 21 sanal iş parçacıkları ve Go gibi güncel gerçeklemeleri yorumlamak.

TEMEL KAVRAMLAR

İş Parçacığı Nedir?

- **Süreç (process):** yürütülmekte olan bir program; adres alanı, açık dosyalar gibi **kaynakların sahibidir**.
- **İş parçacığı (thread):** süreç içindeki bir **yürütme akışı**; CPU kullanımının ve çizelgelemenin temel birimidir.
- Bir süreçte bir ya da daha çok iş parçacığı bulunur; hepsi **aynı adres alanını** paylaşır.
- Her iş parçacığının kendine ait **PC, yazmaç kümesi, yığını ve durumu** vardır; bunlar TCB'de (thread control block) tutulur.
- Tanenbaum'un ayrımı: süreç *kaynakları gruplar*, iş parçacığı *yürütülür*.

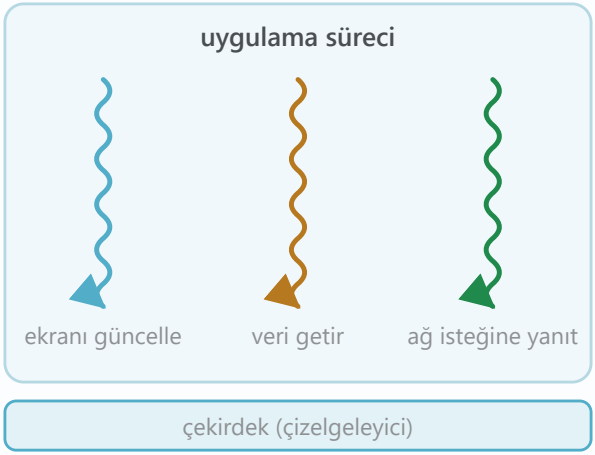


çok sayıda TCB → bir PCB
(Linux'ta ikisi de task_struct)

Eski adı **hafif süreç** (lightweight process, LWP): oluşturmak ve aralarında geçiş yapmak sürece göre ucuzdur, çünkü adres alanı kopyalanmaz.

Neden Çok İş Parçacığı?

- Modern uygulamaların çoğu **çok iş parçacıklıdır**: tarayıcı, IDE, oyun motoru, veritabanı, web sunucusu.
- Farklı görevler ayrı iş parçacıklarında yürür:
  - ekranı güncelleme, veri getirme, ağ isteğine yanıt verme, yazım denetimi…
- Bir görev beklerken (disk, ağ) diğerleri **ilerlemeye devam eder**.
- Süreç oluşturmak pahalıdır; iş parçacığı oluşturmak çok daha ucuzdur.
- Sıralı kod gibi yazılan akışlar programı **basitleştirir**.
- İşletim sistemi çekirdeği de çok iş parçacıklıdır (Linux: `kthreadd`, `kworker`, `ksoftirqd`).

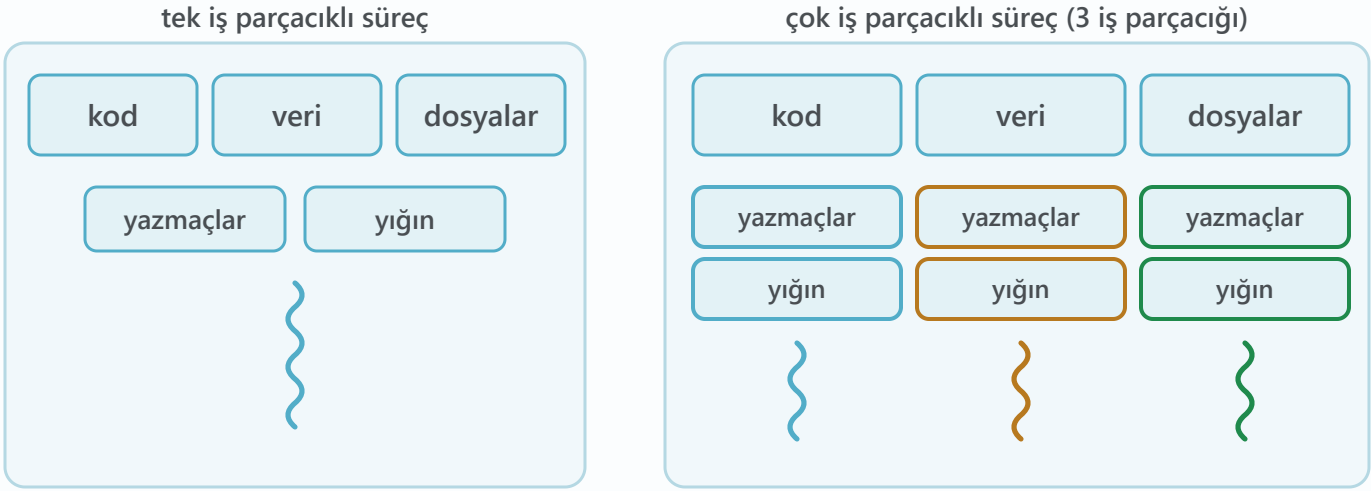


Tek ve Çok İş Parçacıklı Süreç



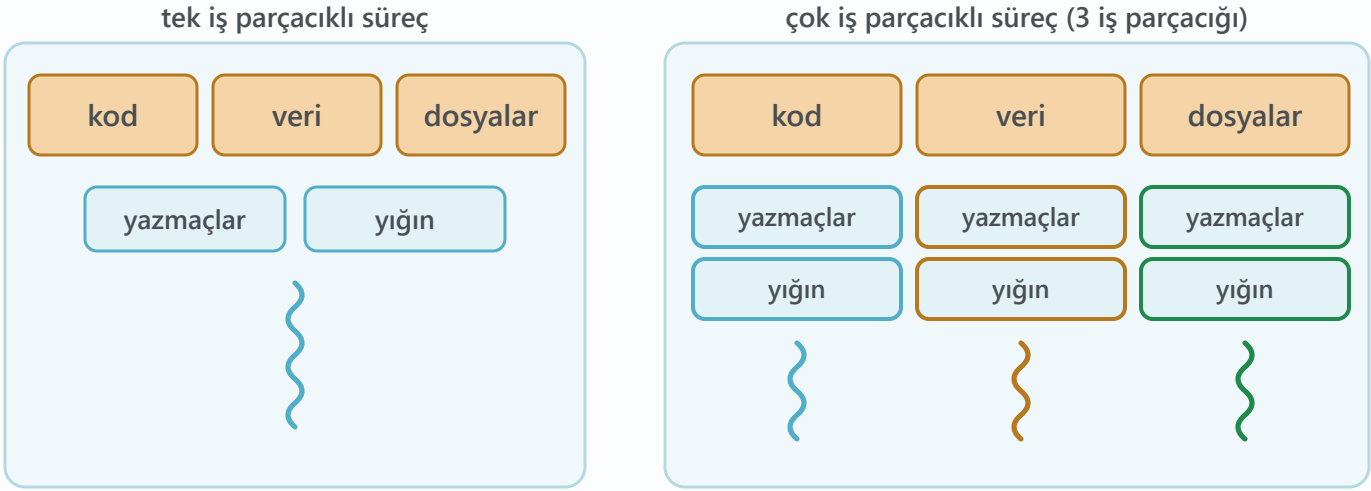
Tek iş parçacıklı süreç: bir yürütme akışı, bir PC, bir yazmaç kümesi ve bir yığın.

Tek ve Çok İş Parçacıklı Süreç



Çok iş parçacıklı süreç: kod, veri ve dosyalar **bir kez** bulunur; her akışın kendi yazmaçları ve yığını vardır.

Tek ve Çok İş Parçacıklı Süreç



üst sıra: paylaşılan · alt sıra: iş parçacığına özel

üç akış aynı kodu ve global veriyi görür

Turuncu satır **paylaşılan** kaynaklar: bir iş parçacığının global değişkene yazdığını diğerleri hemen görür.

TEMEL KAVRAMLAR

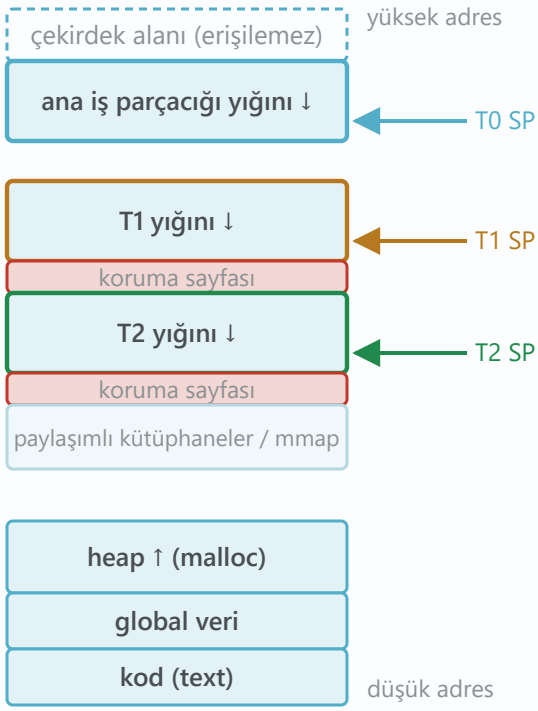
Paylaşılan ve Özel Öğeler

Süreç başına (tüm iş parçacıkları paylaşır)
Adres alanı: kod, global değişkenler, heap
Açık dosyalar, soketler
Çocuk süreçler
Bekleyen alarmlar (zamanlayıcılar)
Sinyal işleyiciler (signal handlers)
Kimlik ve hesap (accounting) bilgisi

İş parçacığı başına (özel)
Program sayacı (PC)
Yazmaçlar (register)
Yığın (stack) ve yığın işaretçisi
Durum: çalışıyor, hazır, bekliyor, sonlandı
İş parçacığı kimliği, sinyal maskesi
İş parçacığına yerel depolama (TLS), `errno`

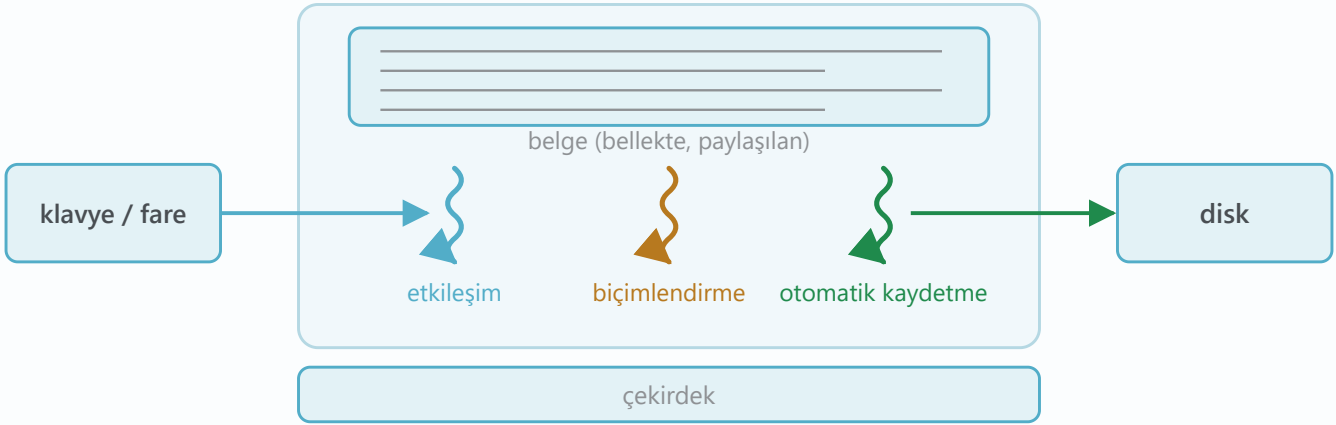
Yığın **özeldir ama korunmaz**: hepsi aynı adres alanında olduğundan hatalı bir işaretçi başka bir iş parçacığının yığınını bozabilir. Paylaşılan veriye erişim senkronizasyon gerektirir.

Her İş Parçacığının Kendi Yığını



- Fonksiyon çağrıları ve yerel değişkenler her akış için ayrı tutulmalıdır: bu yüzden **her iş parçacığının kendi yığını** vardır.
- Linux/glibc'de yeni bir iş parçacığının yığını `mmap` ile ayrılır; varsayılan boyut genellikle `ulimit -s` değeri (çoğunlukla 8 MiB) kadardır.
- Yığının ucundaki **koruma sayfası** (guard page) erişilemez; taşan yığın `SIGSEGV` üretir, komşu yığını sessizce bozmaz.
- Boyut ayarı: `pthread_attr_setstacksize`, Java'da `-Xss`.
- Java sanal iş parçacıklarının yığın çerçeveleri heap'te tutulur; bu sayede milyonlarca sanal iş parçacığı mümkündür.

Örnek: Üç İş Parçacıklı Kelime İşlemci



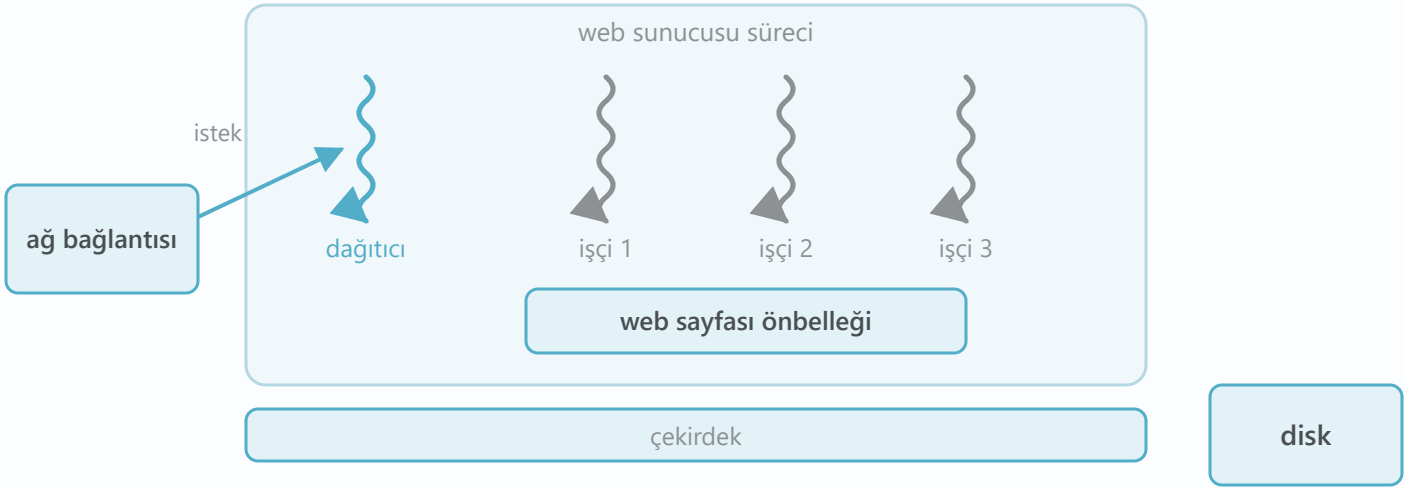
Etkileşim
Tuş ve fare olaylarını hemen işler; arayüz donmaz.

Biçimlendirme
Bir satır silinince yüzlerce sayfayı arka planda yeniden düzenler.

Otomatik kaydetme
Belgeyi belirli aralıklarla diske yazar; G/Ç beklerken kullanıcıyı bekletmez.

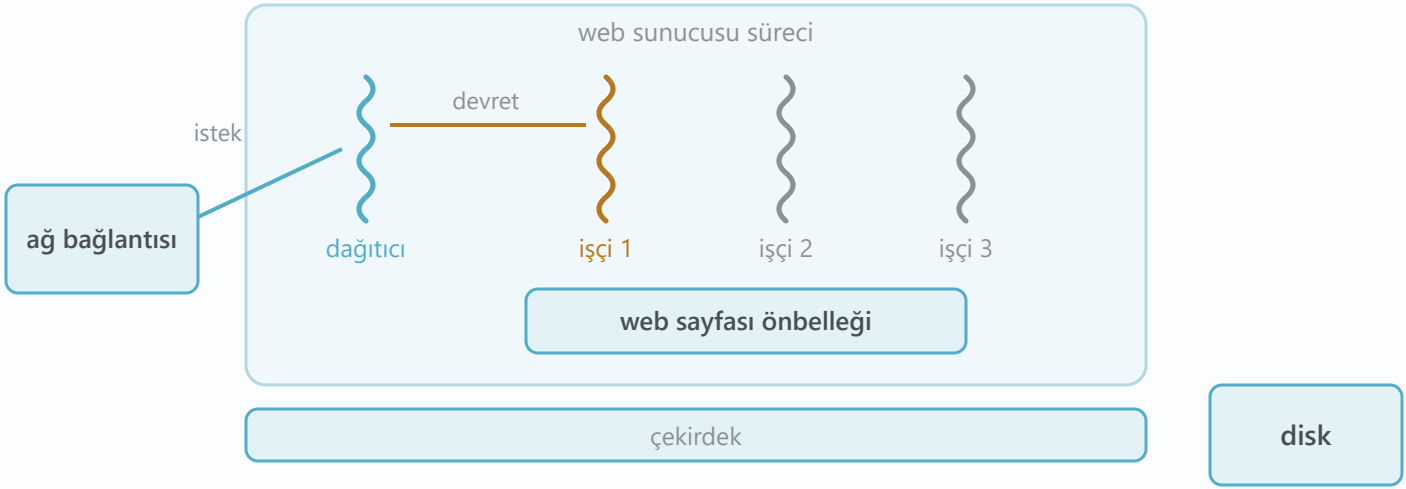
Üç ayrı süreç bu işi yapamaz: üçü de **aynı belge** üzerinde çalışmalıdır. Ortak adres alanı burada temel avantajdır.

Çok İş Parçacıklı Web Sunucusu



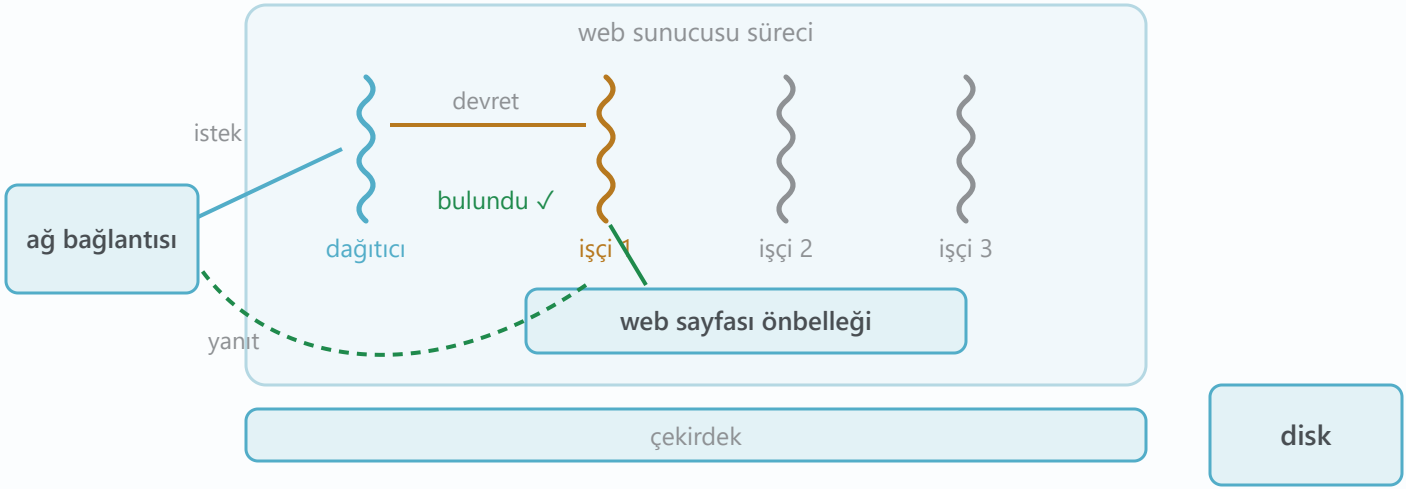
Bir istek gelir: **dağıtıcı** (dispatcher) iş parçacığı isteği ağdan okur.

Çok İş Parçacıklı Web Sunucusu



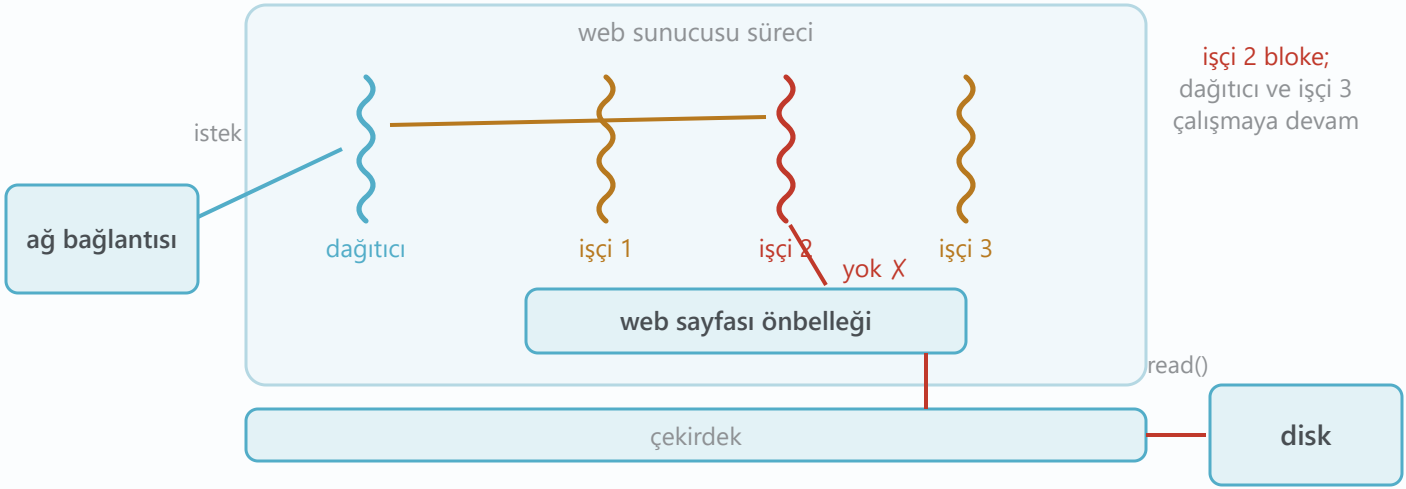
Dağıtıcı boştaki bir **işçi** (worker) iş parçacığını seçer, isteği ona devreder ve uyandırır.

Çok İş Parçacıklı Web Sunucusu



İşçi 1 sayfayı **önbellekte bulur** ve yanıtı gönderir; disk erişimi gerekmez.

Çok İş Parçacıklı Web Sunucusu



Yeni istek işçi 2'ye gider; sayfa önbellekte yoksa işçi 2 diskte **bloke olur**. Bu sırada dağıtıcı ve işçi 3 başka istekleri işler.

Dağıtıcı ve İşçi Kodu

(a) dağıtıcı iş parçacığı

```
while (TRUE) {
  get_next_request(&buf);   /* ağdan oku */
  handoff_work(&buf);       /* işçiye ver */
}
```

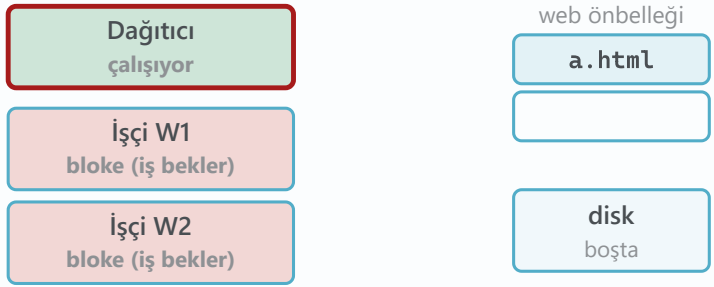
(b) işçi iş parçacığı

```
while (TRUE) {
  wait_for_work(&buf);
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page))
    read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

- Her iki kod da **sıralı** ve okunaklıdır: bloke eden `read` çağrısı yalnızca o işçiyi durdurur.
- Tek iş parçacıklı sunucuda önbellekte olmayan sayfa için tüm süreç bloke olur; disk beklenirken CPU boşta kalır.
- Çok iş parçacıklı sunucuda iş bir işçiye aktarılır, sunucu yeni istekleri kabul etmeye devam eder.

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
▶  while (TRUE) {
▶    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



Sunucu sürecinde üç iş parçacığı var: dağıtıcı ağı dinliyor, iki işçi `wait_for_work`'te bloke. Önbellekte yalnızca a.html var.

Adım 1 / 11

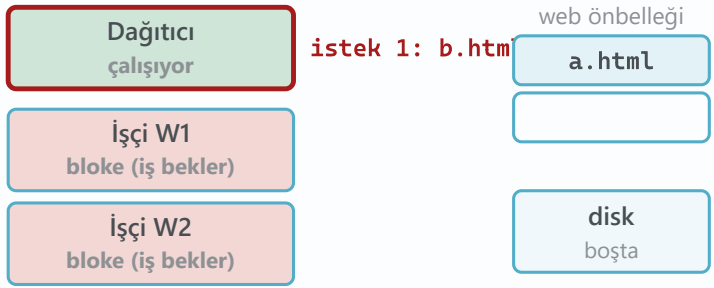
DEĞİŞKENLER

`buf = ?`

ÇIKTI

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
2  while (TRUE) {
▶    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



İstek 1 gelir: **b.html**.

Adım 2 / 11

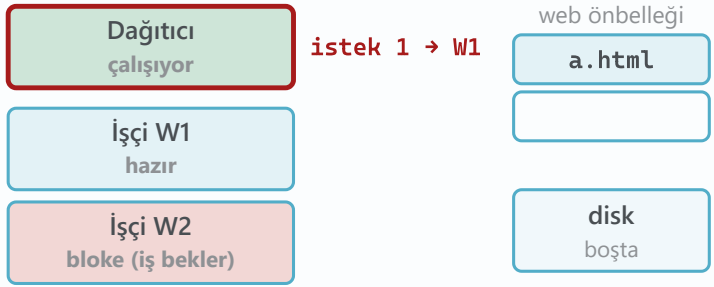
DEĞİŞKENLER

`buf = b.html`

ÇIKTI

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



`handoff_work` isteği boştaki işçiye, W1'e verir ve onu uyandırır. Dağıtıcı hemen ağa döner.

Adım 3 / 11

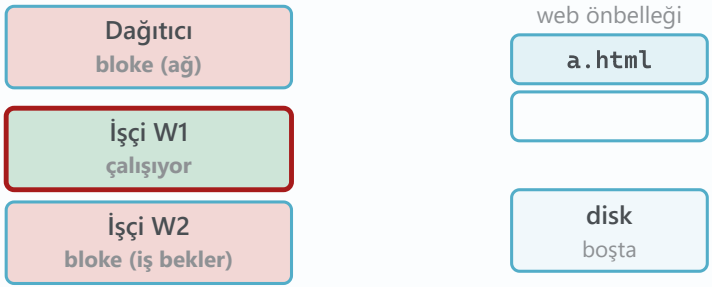
DEĞİŞKENLER

`buf = b.html`

ÇIKTI

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
▶    wait_for_work(&buf);
▶    look_for_page_in_cache(&buf, &page);
▶    if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



W1 sayfayı önbellekte arar: **yok** (ıska).

Adım 4 / 11

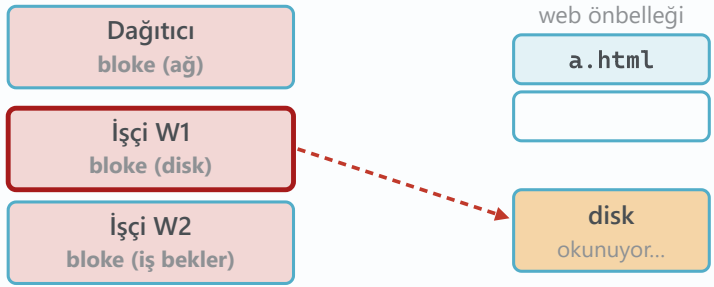
DEĞİŞKENLER

`W1.buf = b.html` `W1.page = yok`

ÇIKTI

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
▶      read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



W1 diskten okumak için **bloke olur**. Yalnızca W1 durur; aynı süreçteki diğer iş parçacıkları çalışmaya devam eder.

Adım 5 / 11

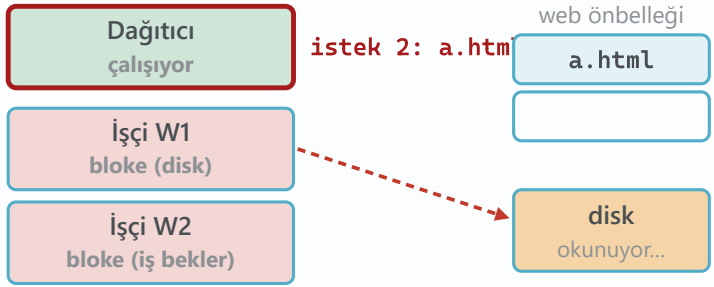
DEĞİŞKENLER

`W1.buf = b.html` `W1.page = yok`

ÇIKTI

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
2  while (TRUE) {
▶    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



Disk beklenirken dağıtıcı istek 2'yi alır: **a.html**.

Adım 6 / 11

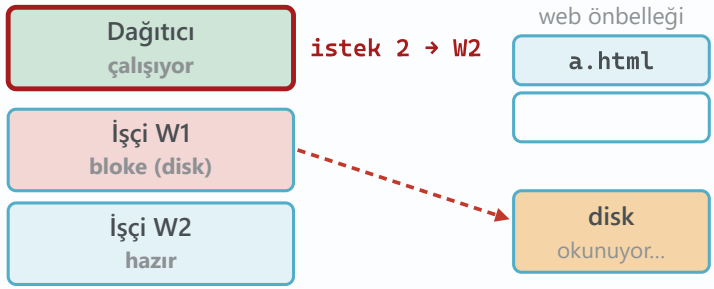
DEĞİŞKENLER

`buf = a.html`

ÇIKTI

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



W1 meşgul, istek 2 W2'ye verilir.

Adım 7 / 11

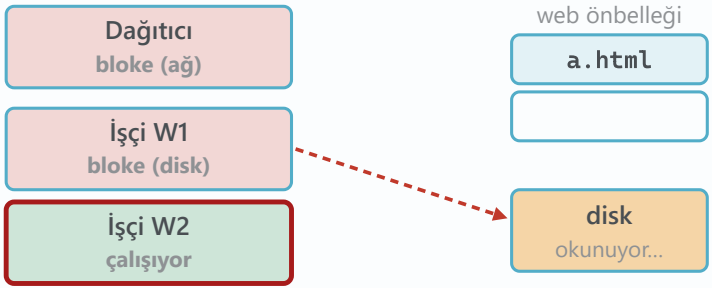
DEĞİŞKENLER

`buf = a.html`

ÇIKTI

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
▶    wait_for_work(&buf);
▶    look_for_page_in_cache(&buf, &page);
▶    if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



W2 a.html'i önbellekte **bulur**: diske gitmeye gerek yok.

Adım 8 / 11

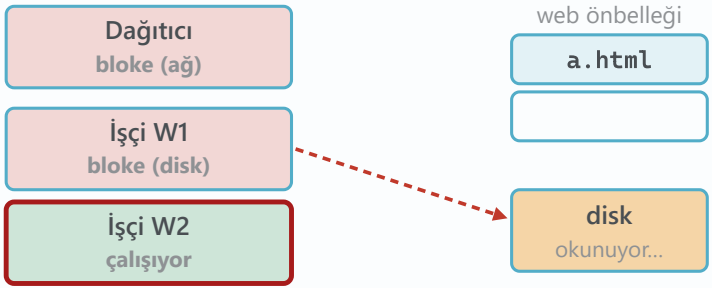
DEĞİŞKENLER

`W2.buf = a.html` `W2.page = a.html`

ÇIKTI

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
▶  return_page(&page);
13 }
```



W2 sayfayı döndürür. **İstek 2, istek 1'den önce** yanıtlandı.

Adım 9 / 11

DEĞİŞKENLER

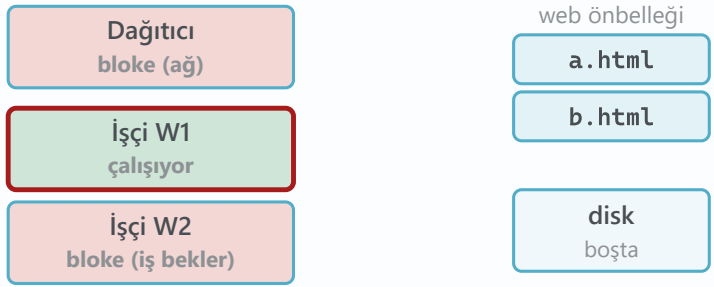
`W2.buf` = **`a.html`** `W2.page` = **`a.html`**

ÇIKTI

```
a.html → istemci 2
```

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
▶      read_page_from_disk(&buf, &page);
▶    return_page(&page);
13 }
```



Disk okuması bitti: W1 uyanır, sayfa önbelleğe de konur ve istemci 1'e gönderilir.

Adım 10 / 11

DEĞİŞKENLER

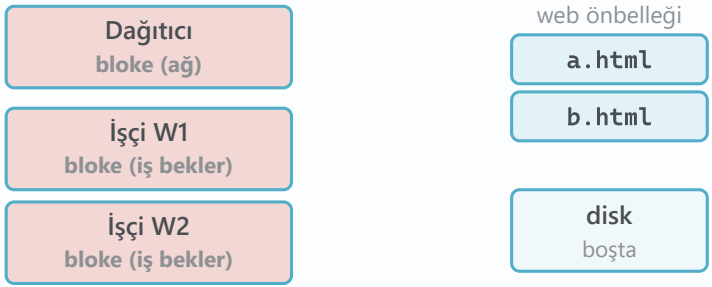
`W1.buf = b.html` `W1.page = b.html`

ÇIKTI

```
a.html → istemci 2
b.html → istemci 1
```

Kod İzleme: Çok İş Parçacıklı Web Sunucusu

```
1  /* dağıtıcı iş parçacığı */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* işçi iş parçacığı */
7  while (TRUE) {
▶    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



İki işçi de yeniden iş bekliyor. Tek iş parçacıklı sunucuda istek 2, disk okuması bitene kadar **bekleyecekti**; burada her kod parçası yine sıralı ve okunaklı.

Adım 11 / 11

DEĞİŞKENLER

`buf = –`

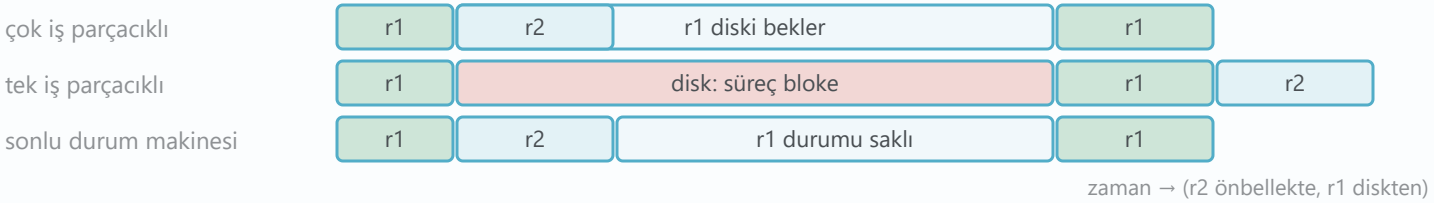
ÇIKTI

```
a.html → istemci 2
b.html → istemci 1
```

Sunucu Tasarımının Üç Yolu

Model	Paralellik / örtüşme	Sistem çağrıları	Programlama
Çok iş parçacıklı süreç	var	bloke eden	sıralı kod, kolay
Tek iş parçacıklı süreç	yok	bloke eden	kolay ama yavaş
Sonlu durum makinesi (olay güdümlü)	var (G/Ç örtüşür)	bloke etmeyen + bildirim / kesme	zor: her isteğin durumu elle saklanır

Bugün: olay güdümlü model nginx, Node.js, Redis gibi sistemlerde yaygındır. Linux'ta `epoll` ve `io_uring`, BSD/macOS'ta `kqueue` kullanılır. Çoğu sunucu iki yaklaşımı birleştirir: **çekirdek başına bir olay döngüsü** + bloke eden işler için iş parçacığı havuzu.



Kaynak: Tanenbaum & Bos, Modern Operating Systems, Şekil 2-10'daki karşılaştırma.

TEMEL KAVRAMLAR

Çok İş Parçacığının Faydaları

Duyarlılık (responsiveness)

Bir kısım bloke olsa ya da uzun bir işlem yapsa bile program yanıt vermeye devam eder; özellikle kullanıcı arayüzleri için önemlidir.

Kaynak paylaşımı

İş parçacıkları sürecin belleğini ve dosyalarını kendiliğinden paylaşır; paylaşımlı bellek ya da ileti geçirme kurmaktan kolaydır.

Ekonomi

Oluşturmak ve aralarında geçiş yapmak süreçlere göre ucuzdur: adres alanı kopyalanmaz, sayfa tabloları değişmez.

Ölçeklenebilirlik

Çok iş parçacıklı tek bir süreç birden çok çekirdekte **aynı anda** çalışabilir; tek iş parçacıklı süreç yalnızca bir çekirdek kullanır.

Sık aktarılan “oluşturma 30 kat, bağlam değiştirme 5 kat hızlı” rakamları eski bir **Solaris** ölçümüdür. Linux’ta süreç de iş parçacığı da `clone()` ile oluşturulduğundan fark daha küçüktür; asıl kazanç bellek eşlemesinin kopyalanmaması ve TLB’nin korunmasıdır.

Bağlam Değiştirme Maliyeti



- Kullanıcı düzeyi iş parçacıklarında geçiş çekirdeğe girmeden (tuzak yok) bir kütüphane fonksiyonuyla yapılır: daha da ucuzdur.
- Doğrudan maliyet mikrosaniyeler mertebesindedir; dolaylı maliyet (soğuk önbellek, TLB ıskaları) çoğu zaman daha büyüktür.

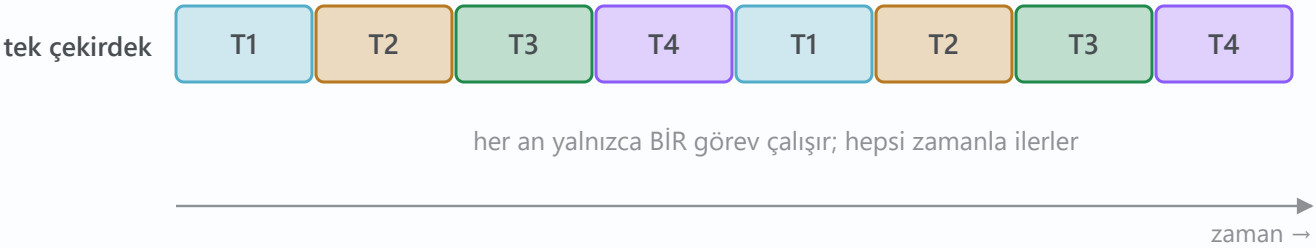
Eş Zamanlılık ve Paralellik

Eş zamanlılık (concurrency)

Birden çok görev *aynı zaman aralığında* ilerler; tek çekirdekte iç içe geçerek bile olur.

Paralellik (parallellism)

Birden çok görev *aynı anda* yürütülür; en az iki çekirdek (ya da işlem birimi) gerekir.



Tek çekirdekte zaman paylaşımı: **eş zamanlı ama paralel değil**.

Eş Zamanlılık ve Paralellik

Eş zamanlılık (concurrency)
Birden çok görev *aynı zaman aralığında* ilerler; tek çekirdekte iç içe geçerek bile olur.

Paralellik (parallelism)
Birden çok görev *aynı anda* yürütülür; en az iki çekirdek (ya da işlem birimi) gerekir.



İki çekirdekte T1/T3 ve T2/T4 **aynı anda** çalışır: paralel yürütme.

Eş Zamanlılık ve Paralellik

Eş zamanlılık (concurrency)

Birden çok görev *aynı zaman aralığında* ilerler; tek çekirdekte iç içe geçerek bile olur.

Paralellik (parallelism)

Birden çok görev *aynı anda* yürütülür; en az iki çekirdek (ya da işlem birimi) gerekir.



Paralellik eş zamanlılığı gerektirir; tersi doğru değildir. Aynı program iki donanımda farklı davranır.

Çok Çekirdekli Programlamanın Zorlukları

Görevleri belirleme
Uygulamada birbirinden bağımsız, paralel yürüyebilecek etkinlikleri bulmak.

Denge (balance)
Görevler benzer büyüklükte olmalı; küçük bir görev için çekirdek ayırmak kazandırmaz.

Veri bölme
Görevlerin eriştiği veri de çekirdeklere bölünmeli.

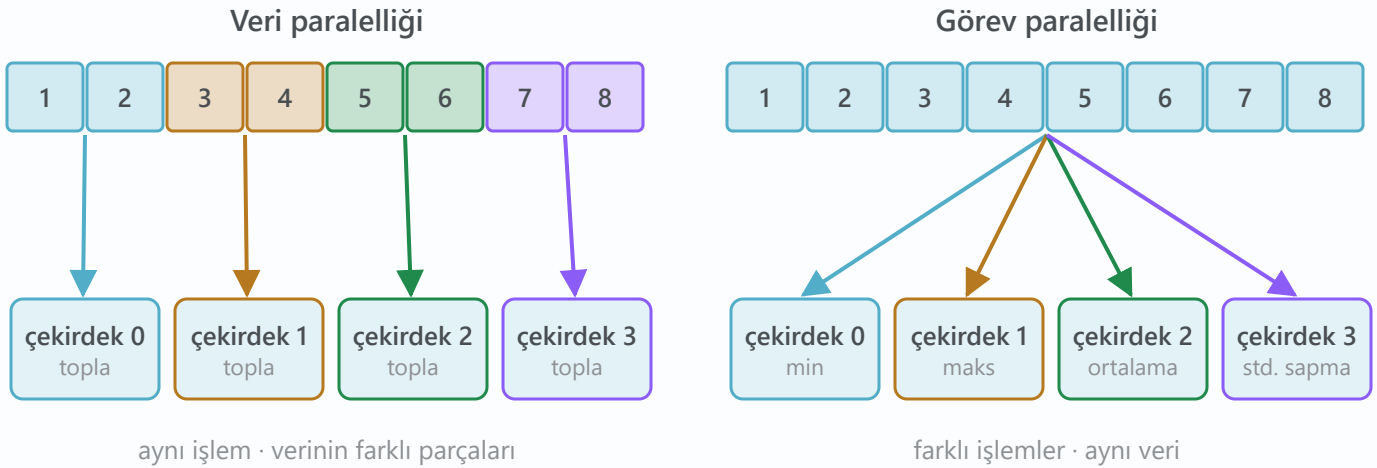
Veri bağımlılığı
Bir görev diğerinin sonucunu bekliyorsa erişim senkronize edilmeli.

Test ve hata ayıklama
Yürütme sırası her çalıştırmada değişebilir; hatalar zor tekrarlanır.

Bellek etkileri
Aynı önbellek satırına yazan çekirdekler birbirini yavaşlatır (yanlış paylaşım, *false sharing*).

Araçlar: ThreadSanitizer (`-fsanitize=thread`), Helgrind, Java Flight Recorder; ama hiçbir araç tüm yarışları yakalamaz. Tasarım aşamasında paylaşımı azaltmak en etkili yöntemdir.

Veri ve Görev Paralelliği

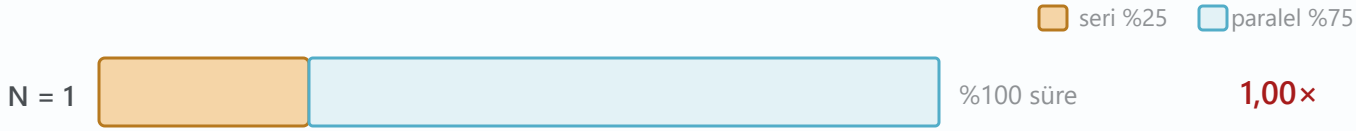


- **Veri paralelliği:** `N` öğeli diziyi toplamak için her çekirdek `N/4` öğe toplar; sonunda kısmi toplamlar birleştirilir (GPU'lar bu modelin uç örneğidir).
- **Görev paralelliği:** iş, bağımsız görevlere bölünür; görevler aynı ya da farklı veriyi kullanabilir.
- Gerçek uygulamalar çoğunlukla ikisini birlikte kullanır.

Amdahl Yasası

$$hızlanma \le 1 / (S + (1 - S) / N)$$

S: seri (paralelleştirilemeyen) oran · N: çekirdek sayısı · $N \to \infty$ iken sınır **$1 / S$**



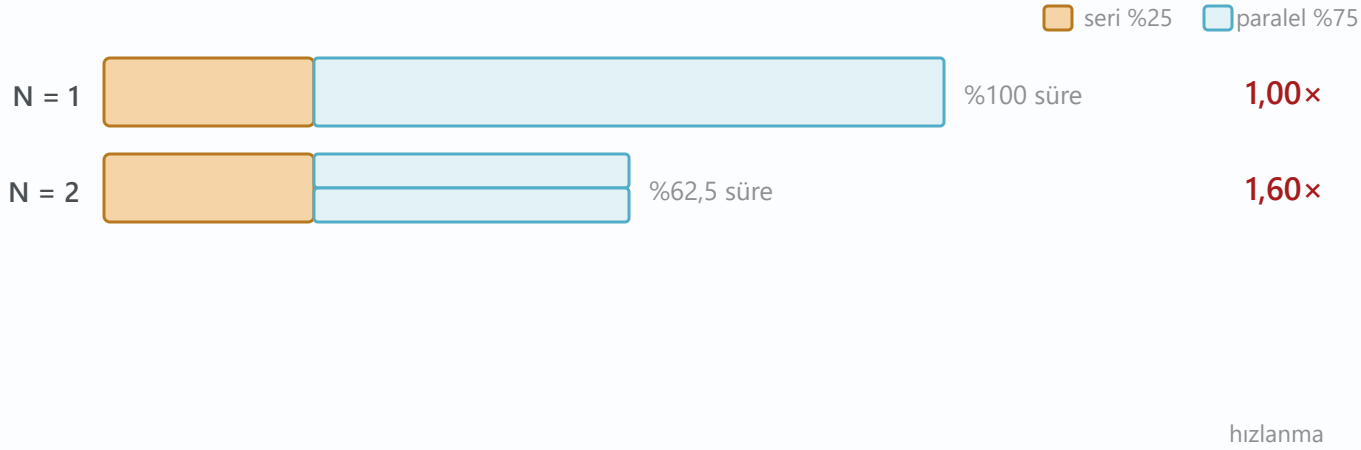
hızlanma

Örnek: **S = 0,25**. Tek çekirdekte süre %100, hızlanma 1×.

Amdahl Yasası

$$hızlanma \le 1 / (S + (1 - S) / N)$$

S: seri (paralelleştirilemeyen) oran · N: çekirdek sayısı · $N \to \infty$ iken sınır **$1 / S$**

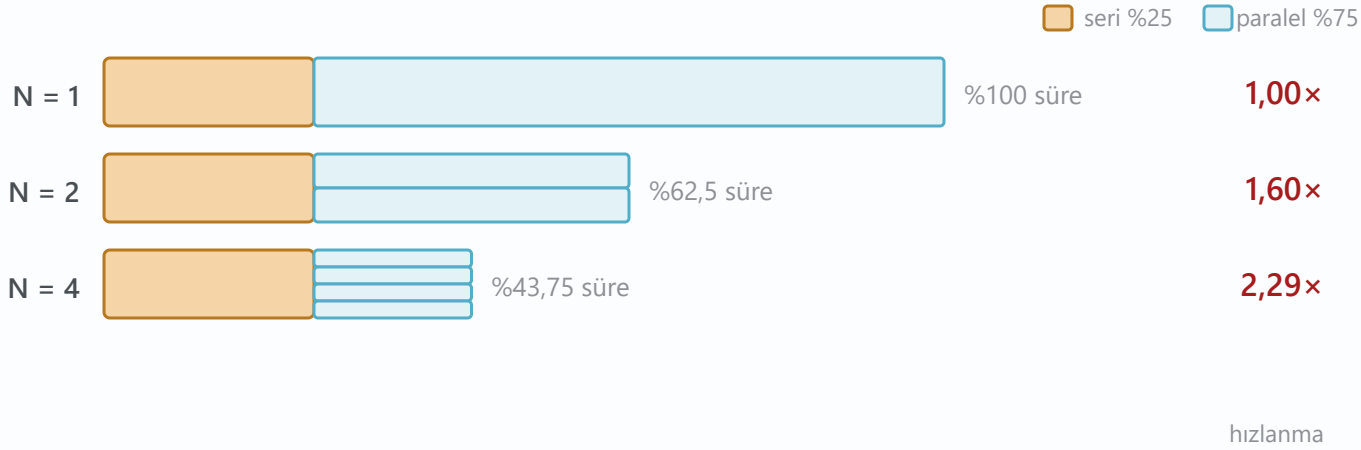


N = 2: 0,25 + 0,75/2 = 0,625 → hızlanma 1 / 0,625 = **1,60×** (2× değil).

Amdahl Yasası

$$hızlanma \le 1 / (S + (1 - S) / N)$$

S: seri (paralelleştirilemeyen) oran · N: çekirdek sayısı · $N \to \infty$ iken sınır **$1 / S$**

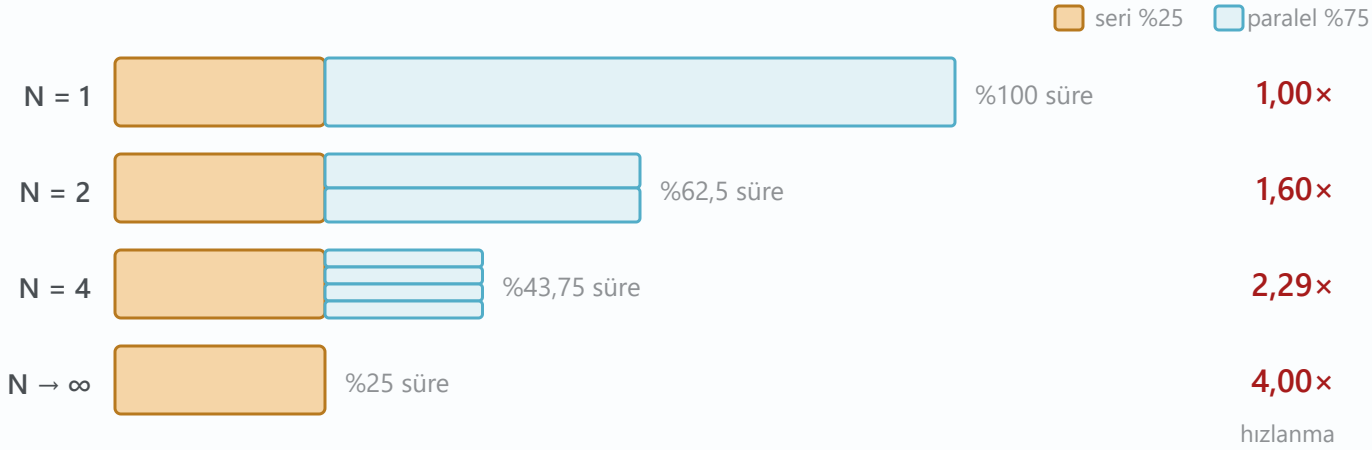


N = 4: 0,25 + 0,75/4 = 0,4375 → hızlanma **2,29×**.

Amdahl Yasası

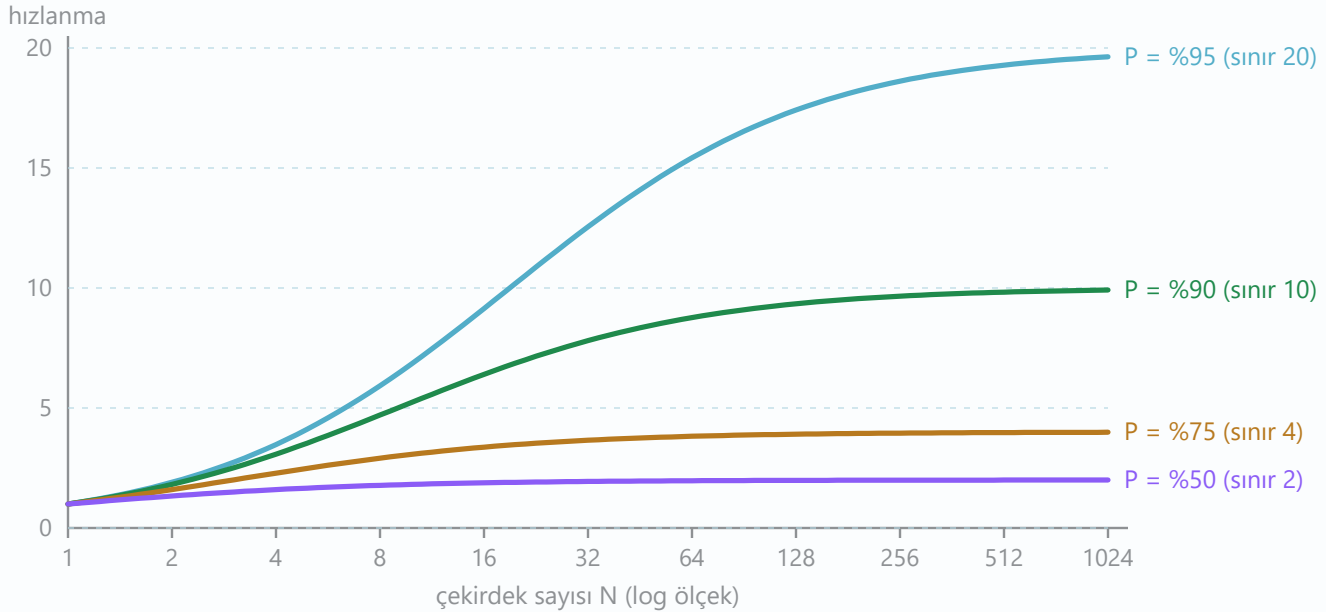
$$hızlanma \le 1 / (S + (1 - S) / N)$$

S: seri (paralelleştirilemeyen) oran · N: çekirdek sayısı · $N \to \infty$ iken sınır **$1 / S$**



N → ∞: paralel kısım sıfıra yaklaşır, seri %25 kalır → en fazla **4×**.

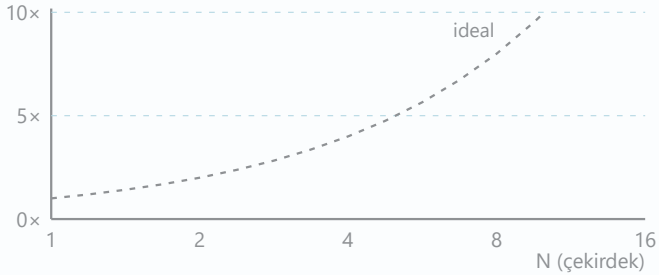
Amdahl Eğrileri



- Programın **%95'i** paralelse teorik hızlanma sınırı 1 / 0,05 = **20×**; 1024 çekirdekte bile yaklaşık 19,6×.
- Gerçekte senkronizasyon ve iletişim yükü nedeniyle hızlanma daha da düşüktür. **Gustafson yasası:** problem boyutu çekirdekle büyürse ölçeklenmiş hızlanma S + (1 − S)·N olur.

Kod İzleme: Amdahl Hızlanması

```
▶ double S = 0.1;            /* seri oran */
▶ int N = 1;                 /* çekirdek */
3 while (N <= 16) {
4   double h = 1 / (S + (1 - S) / N);
5   printf("N=%2d  %.2fx\n", N, h);
6   N = N * 2;
7 }
8 printf("sınır %.0fx\n", 1 / S);
```



Programın **%10**'u seri, %90'ı paralelleştirilebilir. Tek çekirdekte süre 1 birim.

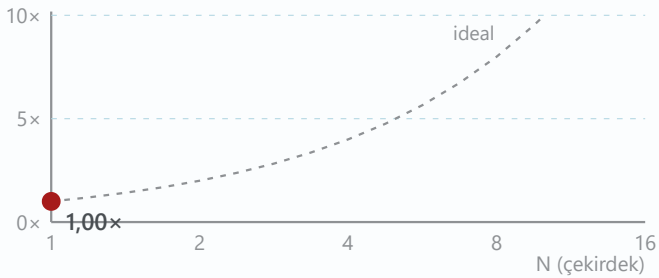
Adım 1 / 9

DEĞİŞKENLER

`S = 0.1` `N = 1`

Kod İzleme: Amdahl Hızlanması

```
1 double S = 0.1;            /* seri oran */
2 int N = 1;                 /* çekirdek */
▶ while (N <= 16) {
▶   double h = 1 / (S + (1 - S) / N);
5   printf("N=%2d  %.2fx\n", N, h);
6   N = N * 2;
7 }
8 printf("sınır %.0fx\n", 1 / S);
```



N = 1: 0,1 + 0,9 / 1 = 1 → **h = 1**.

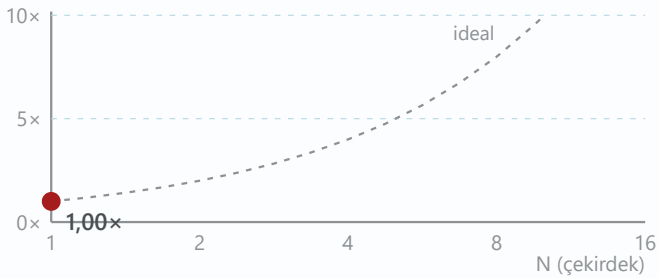
Adım 2 / 9

DEĞİŞKENLER

`S = 0.1` `N = 1` `h = 1`

Kod İzleme: Amdahl Hızlanması

```
1 double S = 0.1;            /* seri oran */
2 int N = 1;                 /* çekirdek */
3 while (N <= 16) {
4   double h = 1 / (S + (1 - S) / N);
▶   printf("N=%2d  %.2fx\n", N, h);
▶   N = N * 2;
7 }
8 printf("sınır %.0fx\n", 1 / S);
```



Sonuç yazılır ve çekirdek sayısı ikiye katlanır: N = 2.

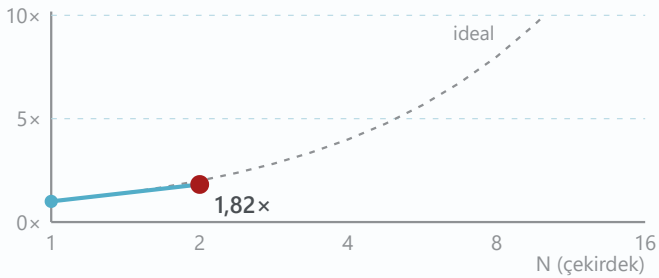
Adım 3 / 9

DEĞİŞKENLER

`S = 0.1` `N = 2` `h = 1`

Kod İzleme: Amdahl Hızlanması

```
1 double S = 0.1;            /* seri oran */
2 int N = 1;                 /* çekirdek */
▶ while (N <= 16) {
▶   double h = 1 / (S + (1 - S) / N);
▶   printf("N=%2d  %.2fx\n", N, h);
▶   N = N * 2;
7 }
8 printf("sınır %.0fx\n", 1 / S);
```



N = 2: 0,1 + 0,9 / 2 = 0,55 → h = **1,82×**.

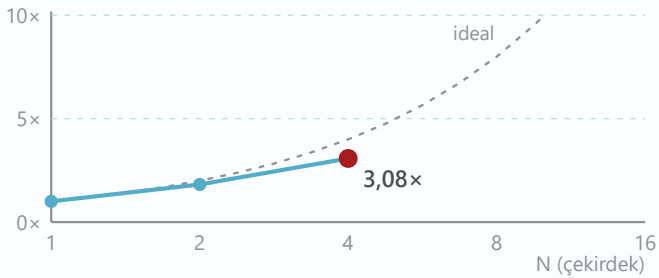
Adım 4 / 9

DEĞİŞKENLER

`S = 0.1` `N = 2` `h = 1.818`

Kod İzleme: Amdahl Hızlanması

```
1 double S = 0.1;            /* seri oran */
2 int N = 1;                 /* çekirdek */
▶ while (N <= 16) {
▶   double h = 1 / (S + (1 - S) / N);
▶   printf("N=%2d  %.2fx\n", N, h);
▶   N = N * 2;
7 }
8 printf("sınır %.0fx\n", 1 / S);
```



N = 4: 0,1 + 0,9 / 4 = 0,325 → h = **3,08×**.

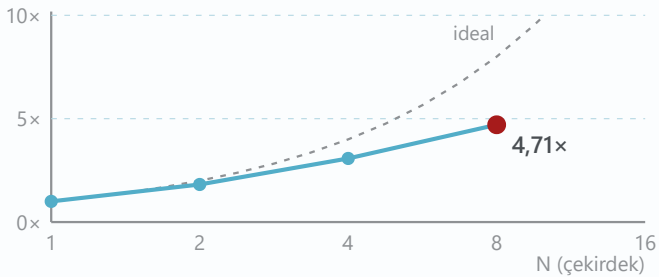
Adım 5 / 9

DEĞİŞKENLER

`S = 0.1` `N = 4` `h = 3.077`

Kod İzleme: Amdahl Hızlanması

```
1 double S = 0.1;            /* seri oran */
2 int N = 1;                 /* çekirdek */
▶ while (N <= 16) {
▶   double h = 1 / (S + (1 - S) / N);
▶   printf("N=%2d  %.2fx\n", N, h);
▶   N = N * 2;
7 }
8 printf("sınır %.0fx\n", 1 / S);
```



N = 8: 0,1 + 0,9 / 8 = 0,2125 → h = **4,71×**. Çekirdek iki katına çıktı, hızlanma %53 arttı.

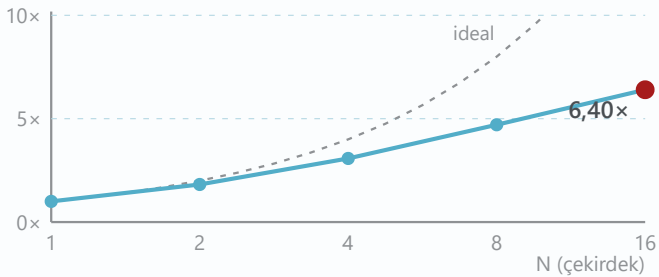
Adım 6 / 9

DEĞİŞKENLER

`S = 0.1` `N = 8` `h = 4.706`

Kod İzleme: Amdahl Hızlanması

```
1 double S = 0.1;            /* seri oran */
2 int N = 1;                 /* çekirdek */
▶ while (N <= 16) {
▶   double h = 1 / (S + (1 - S) / N);
▶   printf("N=%2d  %.2fx\n", N, h);
▶   N = N * 2;
7 }
8 printf("sınır %.0fx\n", 1 / S);
```



N = 16: 0,1 + 0,9 / 16 = 0,15625 → h = **6,40×**.
Çekirdek iki katına çıktı, hızlanma %36 arttı.

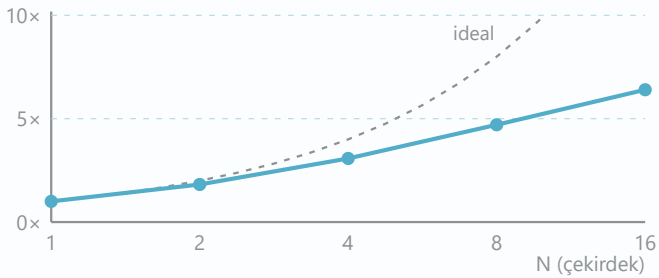
Adım 7 / 9

DEĞİŞKENLER

`S = 0.1` `N = 16` `h = 6.4`

Kod İzleme: Amdahl Hızlanması

```
1 double S = 0.1;            /* seri oran */
2 int N = 1;                 /* çekirdek */
▶ while (N <= 16) {
4   double h = 1 / (S + (1 - S) / N);
5   printf("N=%2d  %.2fx\n", N, h);
6   N = N * 2;
7 }
8 printf("sınır %.0fx\n", 1 / S);
```



N = 32 > 16: döngü biter.

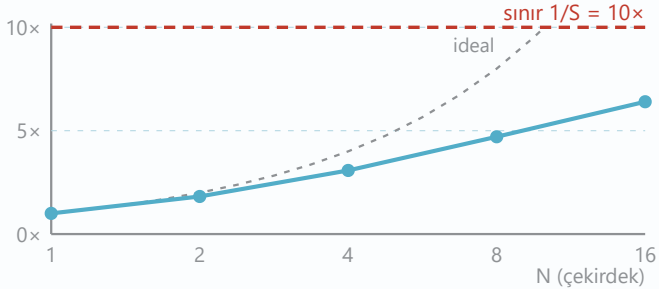
Adım 8 / 9

DEĞİŞKENLER

`S = 0.1` `N = 32`

Kod İzleme: Amdahl Hızlanması

```
1 double S = 0.1;            /* seri oran */
2 int N = 1;                 /* çekirdek */
3 while (N <= 16) {
4   double h = 1 / (S + (1 - S) / N);
5   printf("N=%2d  %.2fx\n", N, h);
6   N = N * 2;
7 }
▶ printf("sınır %.0fx\n", 1 / S);
```



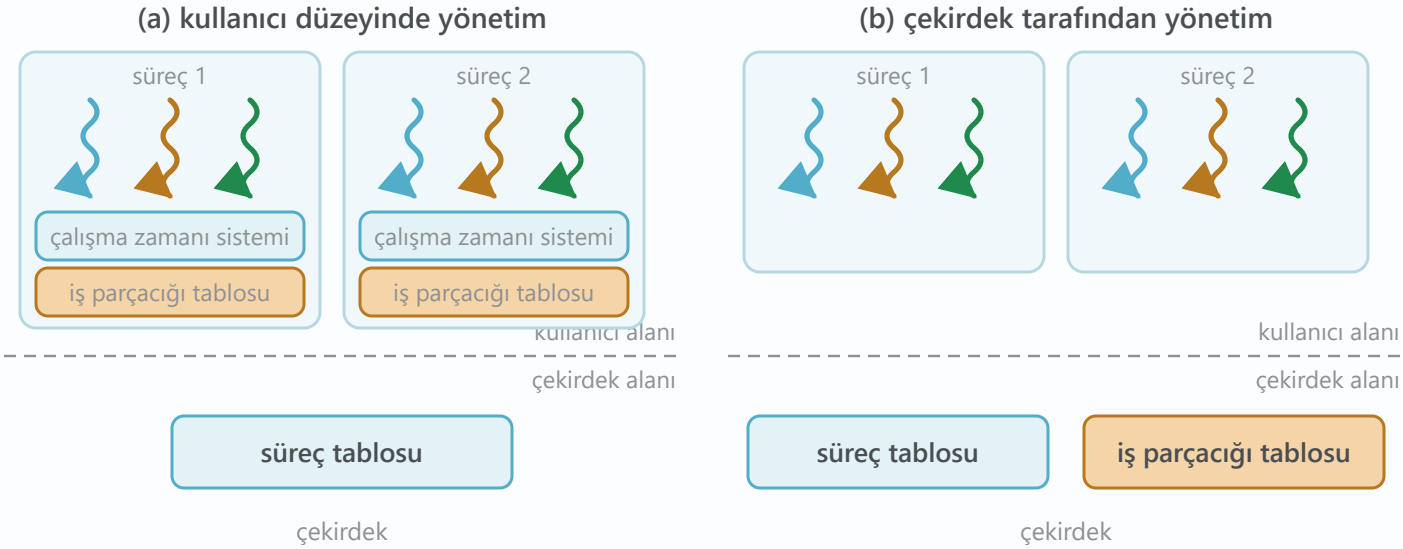
N → ∞ iken paralel kısım sıfıra iner, seri %10 kalır: hızlanma **10×**'u asla geçemez. 16 çekirdekte bile 6,4×.

Adım 9 / 9

DEĞİŞKENLER

`S = 0.1` `N = 32`

Kullanıcı ve Çekirdek Düzeyi İş Parçacıkları



- **Kullanıcı düzeyi:** iş parçacıkları kütüphanede (çalışma zamanı sistemi) yönetilir; çekirdek her süreci tek akışlı sanır.
- **Çekirdek düzeyi:** çekirdek her iş parçacığını tanır ve kendisi çizelgeler. Linux, Windows, macOS böyledir.

Kullanıcı Düzeyi İş Parçacıkları

Avantajlar

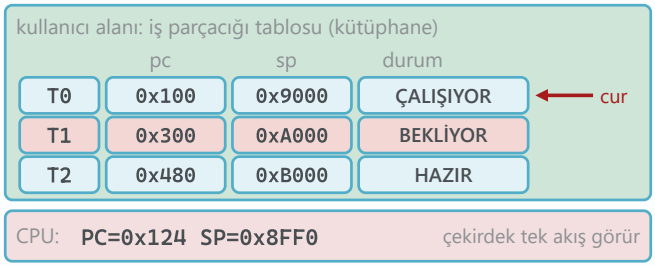
+ İş parçacığı desteği olmayan bir çekirdekte bile kütüphane olarak gerçekleştirilebilir.
+ Her süreç kendi **iş parçacığı tablosunu** tutar: PC, yazmaçlar, SP, durum.
+ Geçiş çok hızlıdır: **tuzak (trap) yok**, çekirdeğe giriş yok, bellek eşlemesi değişmez, TLB boşaltılmaz.
+ Her süreç kendine özgü bir çizelgeleme algoritması kullanabilir.

Dezavantajlar

– Bloke eden bir sistem çağrısı (ör. `read`) **tüm süreci**, dolayısıyla tüm iş parçacıklarını durdurur.
– Çözümler zarif değil: çağrıyı `select` ile önceden sınayan sarmalayıcılar (jacket), bloke etmeyen çağrılar.
– Sayfa hatası da tüm süreci bloke eder.
– Saat kesmesi kütüphaneye gelmez: iş parçacığı **gönüllü bırakmazsa** (`yield`) diğerleri hiç çalışmaz.
– Tek süreç birden çok çekirdeği kullanamaz.

Kod İzleme: Kullanıcı Düzeyinde `thread_yield`

```
1  enum { HAZIR, CALISIYOR, BEKLIYOR };
2  struct tcb { int pc, sp, durum; } T[3];
▶  int cur = 0;                /* çalışan */
4  void thread_yield(void) {   /* kütüphanede */
5    T[cur].pc = PC;   T[cur].sp = SP;
6    T[cur].durum = HAZIR;
7    do cur = (cur + 1) % 3;
8    while (T[cur].durum != HAZIR);
9    T[cur].durum = CALISIYOR;
10   PC = T[cur].pc;   SP = T[cur].sp;
11 }
```



Üç iş parçacığı aynı süreçte. T0 çalışıyor (CPU'da PC = 0x124); T1 kütüphanedeki bir kilidi **bekliyor**; T2 hazır. T0 gönüllü olarak `thread_yield()` çağırır.

Adım 1 / 10

DEĞİŞKENLER

`cur = 0` `PC = 0x124` `SP = 0x8FF0`

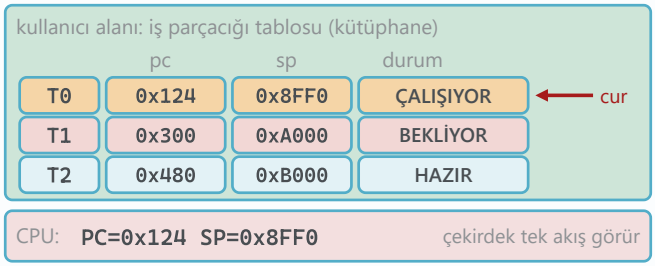
ÇAĞRI YIĞINI

`thread_yield()`

`T0`

Kod İzleme: Kullanıcı Düzeyinde `thread_yield`

```
1  enum { HAZIR, CALISIYOR, BEKLIYOR };
2  struct tcb { int pc, sp, durum; } T[3];
3  int cur = 0;                /* çalışan */
4  void thread_yield(void) {   /* kütüphanede */
▶    T[cur].pc = PC;  T[cur].sp = SP;
6    T[cur].durum = HAZIR;
7    do cur = (cur + 1) % 3;
8    while (T[cur].durum != HAZIR);
9    T[cur].durum = CALISIYOR;
10   PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



T0'ın PC ve SP değerleri **tabloya** kaydedilir. Bu sıradan bir fonksiyon çağrısıdır: tuzak yok, çekirdeğe giriş yok.

Adım 2 / 10

DEĞİŞKENLER

`cur = 0` `PC = 0x124` `SP = 0x8FF0`

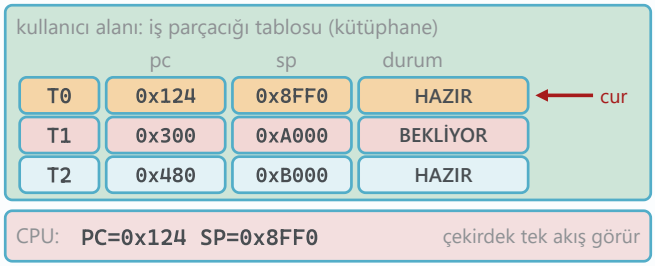
ÇAĞRI YIĞINI

`thread_yield()`

`T0`

Kod İzleme: Kullanıcı Düzeyinde `thread_yield`

```
1  enum { HAZIR, CALISIYOR, BEKLIYOR };
2  struct tcb { int pc, sp, durum; } T[3];
3  int cur = 0;                /* çalışan */
4  void thread_yield(void) {   /* kütüphanede */
5    T[cur].pc = PC;  T[cur].sp = SP;
▶    T[cur].durum = HAZIR;
7    do cur = (cur + 1) % 3;
8    while (T[cur].durum != HAZIR);
9    T[cur].durum = CALISIYOR;
10   PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



T0 hazır.

Adım 3 / 10

DEĞİŞKENLER

`cur = 0` `PC = 0x124` `SP = 0x8FF0`

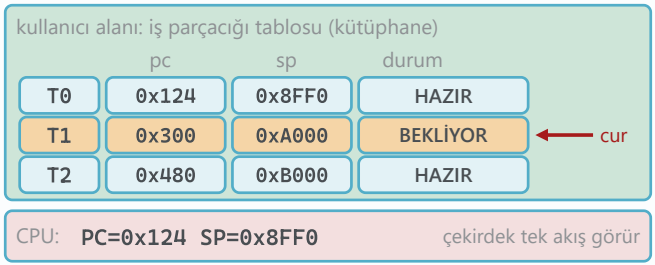
ÇAĞRI YIĞINI

`thread_yield()`

`T0`

Kod İzleme: Kullanıcı Düzeyinde `thread_yield`

```
1  enum { HAZIR, CALISIYOR, BEKLIYOR };
2  struct tcb { int pc, sp, durum; } T[3];
3  int cur = 0;                /* çalışan */
4  void thread_yield(void) {   /* kütüphanede */
5    T[cur].pc = PC;  T[cur].sp = SP;
6    T[cur].durum = HAZIR;
▶    do cur = (cur + 1) % 3;
8    while (T[cur].durum != HAZIR);
9    T[cur].durum = CALISIYOR;
10   PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



cur = 1.

Adım 4 / 10

DEĞİŞKENLER

`cur = 1` `PC = 0x124` `SP = 0x8FF0`

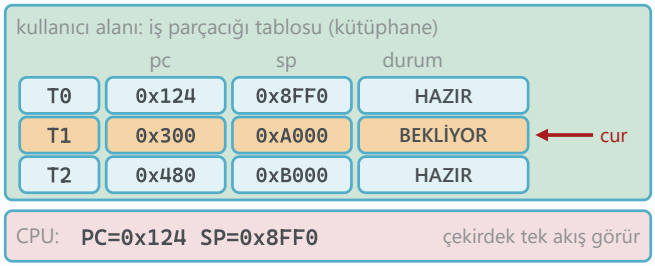
ÇAĞRI YIĞINI

`thread_yield()`

`T0`

Kod İzleme: Kullanıcı Düzeyinde `thread_yield`

```
1  enum { HAZIR, CALISIYOR, BEKLIYOR };
2  struct tcb { int pc, sp, durum; } T[3];
3  int cur = 0;                /* çalışan */
4  void thread_yield(void) {   /* kütüphanede */
5    T[cur].pc = PC;  T[cur].sp = SP;
6    T[cur].durum = HAZIR;
7    do cur = (cur + 1) % 3;
▶    while (T[cur].durum != HAZIR);
9    T[cur].durum = CALISIYOR;
10   PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



T1 BEKLİYOR, atlanır: döngü sürer.

Adım 5 / 10

DEĞİŞKENLER

`cur = 1` `PC = 0x124` `SP = 0x8FF0`

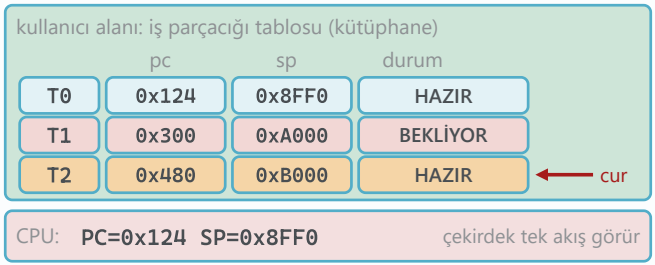
ÇAĞRI YIĞINI

`thread_yield()`

`T0`

Kod İzleme: Kullanıcı Düzeyinde `thread_yield`

```
1  enum { HAZIR, CALISIYOR, BEKLIYOR };
2  struct tcb { int pc, sp, durum; } T[3];
3  int cur = 0;                /* çalışan */
4  void thread_yield(void) {   /* kütüphanede */
5    T[cur].pc = PC;  T[cur].sp = SP;
6    T[cur].durum = HAZIR;
▶    do cur = (cur + 1) % 3;
8    while (T[cur].durum != HAZIR);
9    T[cur].durum = CALISIYOR;
10   PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



cur = 2.

Adım 6 / 10

DEĞİŞKENLER

`cur = 2` `PC = 0x124` `SP = 0x8FF0`

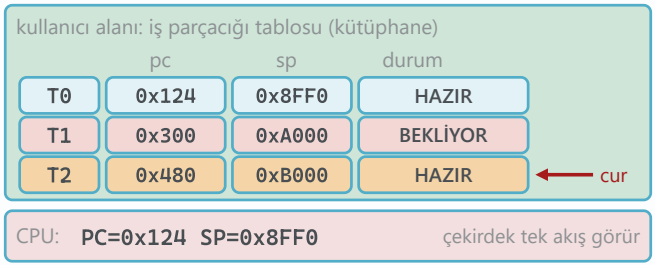
ÇAĞRI YIĞINI

`thread_yield()`

`T0`

Kod İzleme: Kullanıcı Düzeyinde `thread_yield`

```
1  enum { HAZIR, CALISIYOR, BEKLIYOR };
2  struct tcb { int pc, sp, durum; } T[3];
3  int cur = 0;                /* çalışan */
4  void thread_yield(void) {   /* kütüphanede */
5    T[cur].pc = PC;  T[cur].sp = SP;
6    T[cur].durum = HAZIR;
7    do cur = (cur + 1) % 3;
▶    while (T[cur].durum != HAZIR);
9    T[cur].durum = CALISIYOR;
10   PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



T2 HAZIR: arama biter.

Adım 7 / 10

DEĞİŞKENLER

`cur = 2` `PC = 0x124` `SP = 0x8FF0`

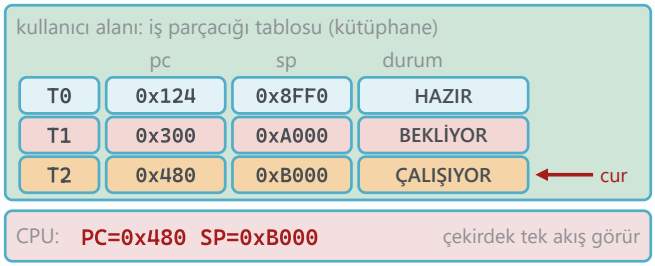
ÇAĞRI YIĞINI

`thread_yield()`

`T0`

Kod İzleme: Kullanıcı Düzeyinde `thread_yield`

```
1  enum { HAZIR, CALISIYOR, BEKLIYOR };
2  struct tcb { int pc, sp, durum; } T[3];
3  int cur = 0;                /* çalışan */
4  void thread_yield(void) {   /* kütüphanede */
5    T[cur].pc = PC;  T[cur].sp = SP;
6    T[cur].durum = HAZIR;
7    do cur = (cur + 1) % 3;
8    while (T[cur].durum != HAZIR);
▶    T[cur].durum = CALISIYOR;
▶    PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



T2 çalışıyor olarak işaretlenir ve yazmaçları **yüklenir**: PC = 0x480. Dönüş komutu artık T2'nin yığınından döner; geçiş birkaç komutla bitti.

Adım 8 / 10

DEĞİŞKENLER

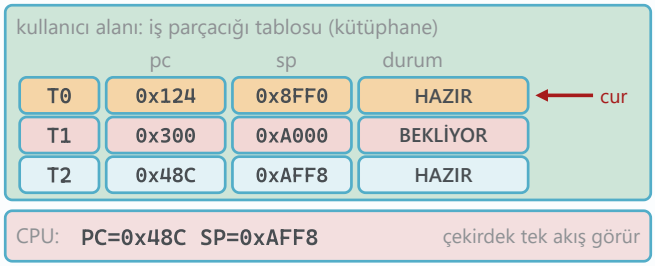
`cur = 2` `PC = 0x480` `SP = 0xB000`

ÇAĞRI YIĞINI

T2

Kod İzleme: Kullanıcı Düzeyinde `thread_yield`

```
1  enum { HAZIR, CALISIYOR, BEKLIYOR };
2  struct tcb { int pc, sp, durum; } T[3];
3  int cur = 0;                /* çalışan */
4  void thread_yield(void) {   /* kütüphanede */
▶    T[cur].pc = PC;  T[cur].sp = SP;
▶    T[cur].durum = HAZIR;
▶    do cur = (cur + 1) % 3;
▶    while (T[cur].durum != HAZIR);
9    T[cur].durum = CALISIYOR;
10   PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



T2 biraz çalıştıktan sonra (PC = 0x48C) o da `thread_yield()` çağırır: durumu kaydedilir, cur = 0 ve T0 hazır.

Adım 9 / 10

DEĞİŞKENLER

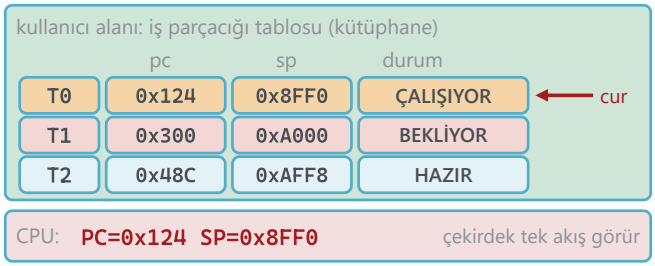
`cur = 0` `PC = 0x48C` `SP = 0xAFF8`

ÇAĞRI YIĞINI

- `thread_yield()`
- T2

Kod İzleme: Kullanıcı Düzeyinde `thread_yield`

```
1  enum { HAZIR, CALISIYOR, BEKLIYOR };
2  struct tcb { int pc, sp, durum; } T[3];
3  int cur = 0;                /* çalışan */
4  void thread_yield(void) {   /* kütüphanede */
5    T[cur].pc = PC;  T[cur].sp = SP;
6    T[cur].durum = HAZIR;
7    do cur = (cur + 1) % 3;
8    while (T[cur].durum != HAZIR);
▶    T[cur].durum = CALISIYOR;
▶    PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



T0 kaldığı yerden, PC = 0x124'ten sürer. T2 hiç bırakmasaydı T0 sonsuza dek bekleyecekti: kütüphaneye **saat kesmesi gelmez**.

Adım 10 / 10

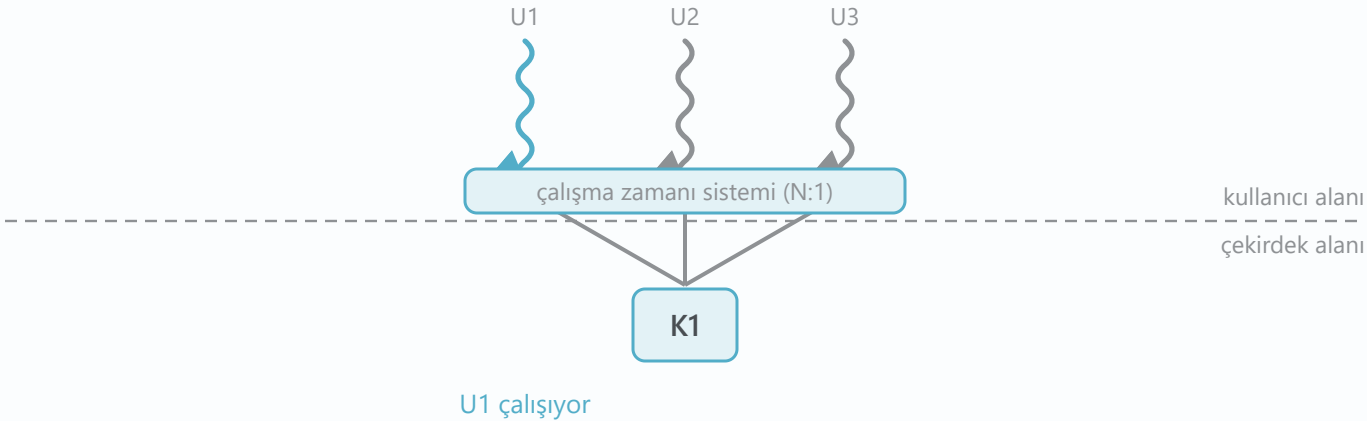
DEĞİŞKENLER

`cur = 0` `PC = 0x124` `SP = 0x8FF0`

ÇAĞRI YIĞINI

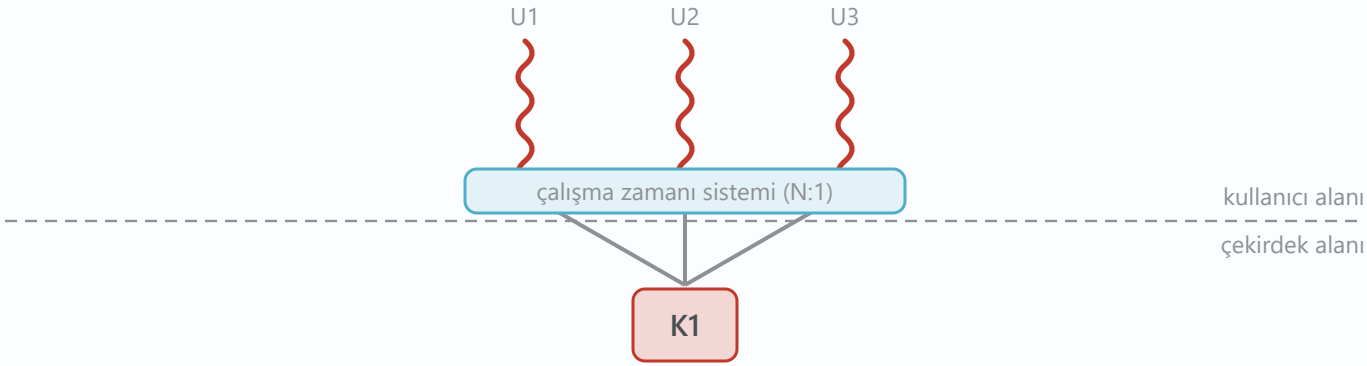
`T0`

Bloke Eden Çağrı: N:1 ve 1:1



Çoktan bire (N:1): üç kullanıcı iş parçacığı tek bir çekirdek iş parçacığı (K1) üzerinde; çizelgelemeyi kütüphane yapar.

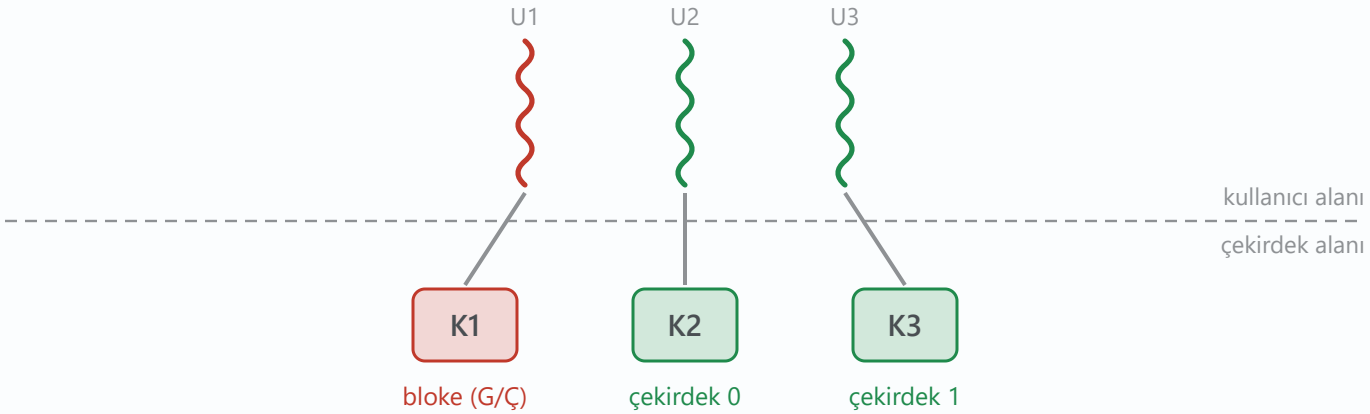
Bloke Eden Çağrı: N:1 ve 1:1



U1 read() çağırdı → K1 bloke → U2 ve U3 de bekler

U1 bloke eden `read()` çağırır: çekirdek yalnızca K1'i görür ve onu bloke eder → **tüm süreç durur**.

Bloke Eden Çağrı: N:1 ve 1:1



Bire bir (1:1): U1'in bloke olması yalnızca K1'i etkiler; U2 ve U3 iki çekirdekte **paralel** çalışır.

Çekirdek Düzeyi İş Parçacıkları

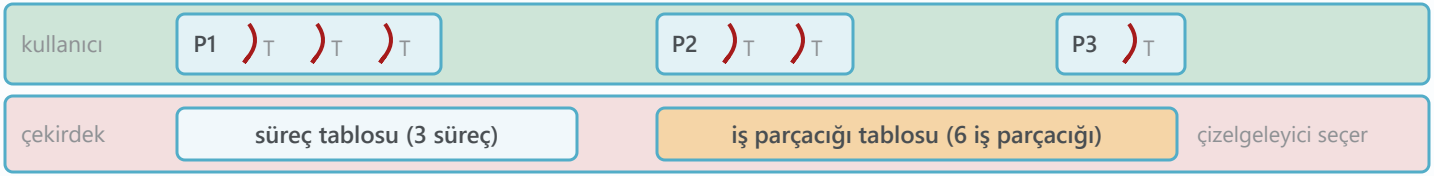
- Çekirdek, kullanıcı düzeyindekiyle aynı bilgileri (yazmaçlar, durum…) içeren bir **iş parçacığı tablosu** tutar.
- Oluşturma ve sonlandırma **sistem çağrısıdır**: kullanıcı düzeyine göre daha pahalı.
- Bir iş parçacığı bloke olursa çekirdek başka birini seçer; seçilenin **aynı süreçten olması gerekmez**.
- Sayfa hatası yalnızca hatayı yapan iş parçacığını bekletir.
- Bir sürecin iş parçacıkları farklı çekirdeklerde **gerçekten paralel** çalışır.
- Her iş parçacığı çekirdek belleğinde yer kaplar (TCB, çekirdek yığını): sayısı sınırsız değildir.

Bugün

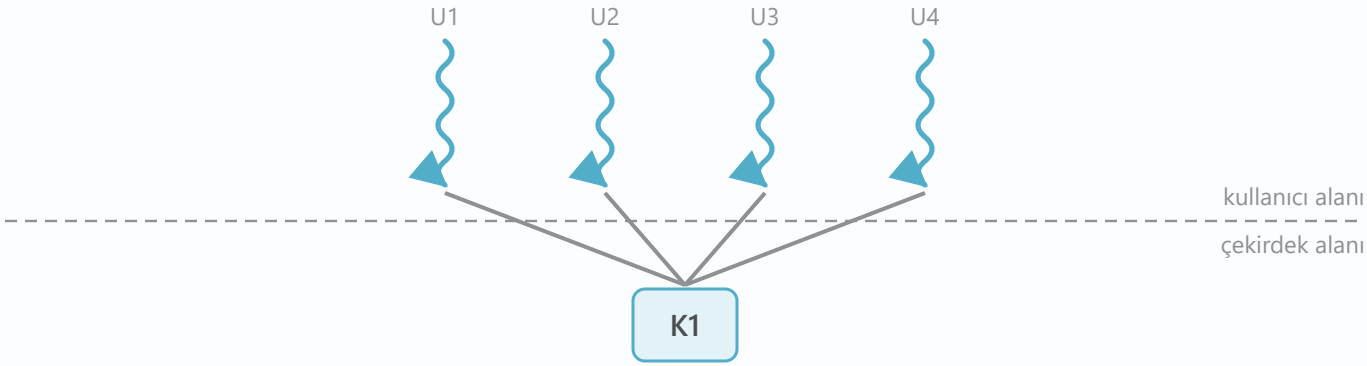
Linux (NPTL), Windows ve macOS **1:1** model kullanır. Oluşturma maliyeti azaldığından on binlerce çekirdek iş parçacığı sorun değildir.

Linux'ta sınırlar: `/proc/sys/kernel/threads-max`, `ulimit -u`, cgroup `pids.max`.

Gözlemlemek için: `ps -eLf`, `top -H`, `/proc/<pid>/task/`.

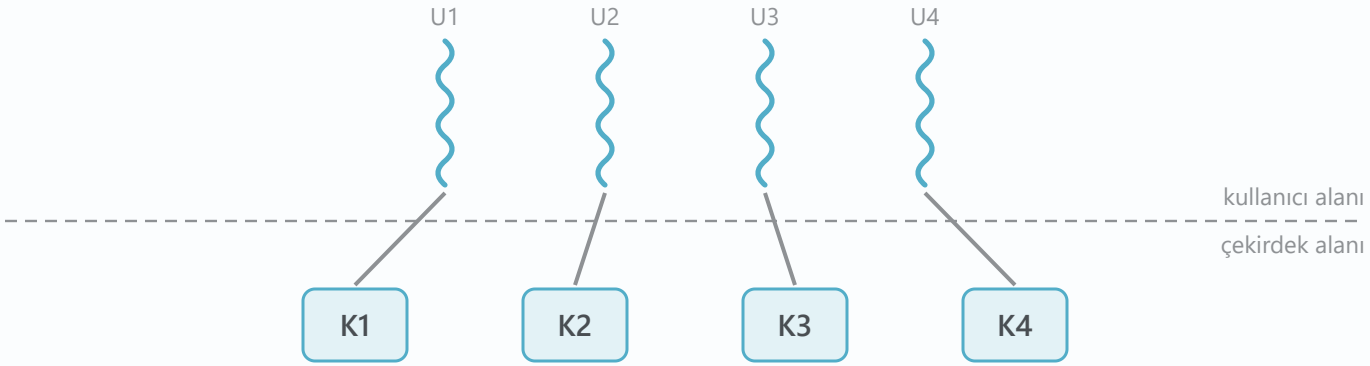


Eşleme Modelleri



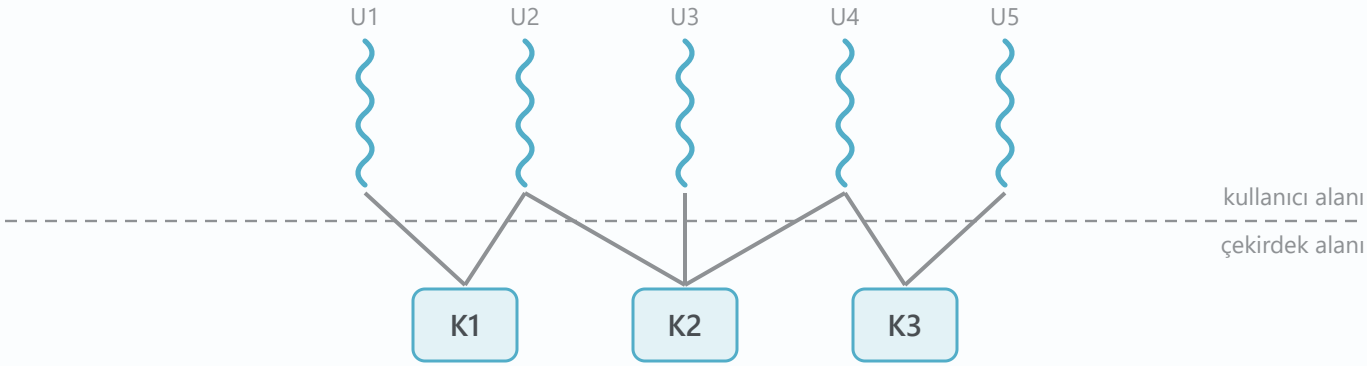
Çoktan bire (N:1): hızlı geçiş, ama bir bloke tüm süreci durdurur ve paralellik yoktur. Örnek: Solaris *green threads*, GNU Portable Threads (tarihî).

Eşleme Modelleri



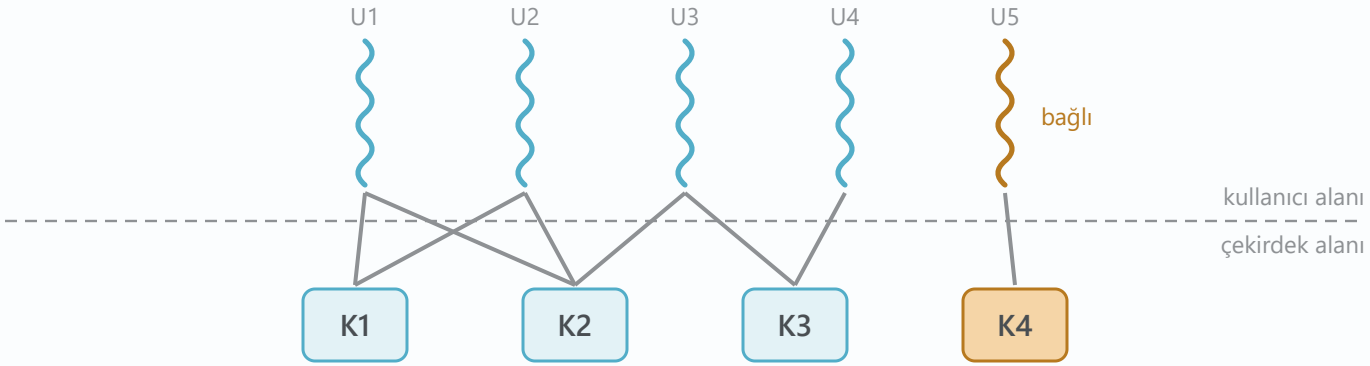
Bire bir (1:1): gerçek paralellik ve bağımsız bloke olma; her iş parçacığı çekirdek kaynağı harcar. Linux, Windows, macOS.

Eşleme Modelleri



Çoktan çoğa (M:N): çok sayıda kullanıcı iş parçacığı, daha az sayıda çekirdek iş parçacığına çoğullanır; gerçeklemesi zordur.

Eşleme Modelleri



İki düzeyli / hibrit: M:N'e ek olarak bir kullanıcı iş parçacığı tek bir çekirdek iş parçacığına **bağlanabilir** (bound).

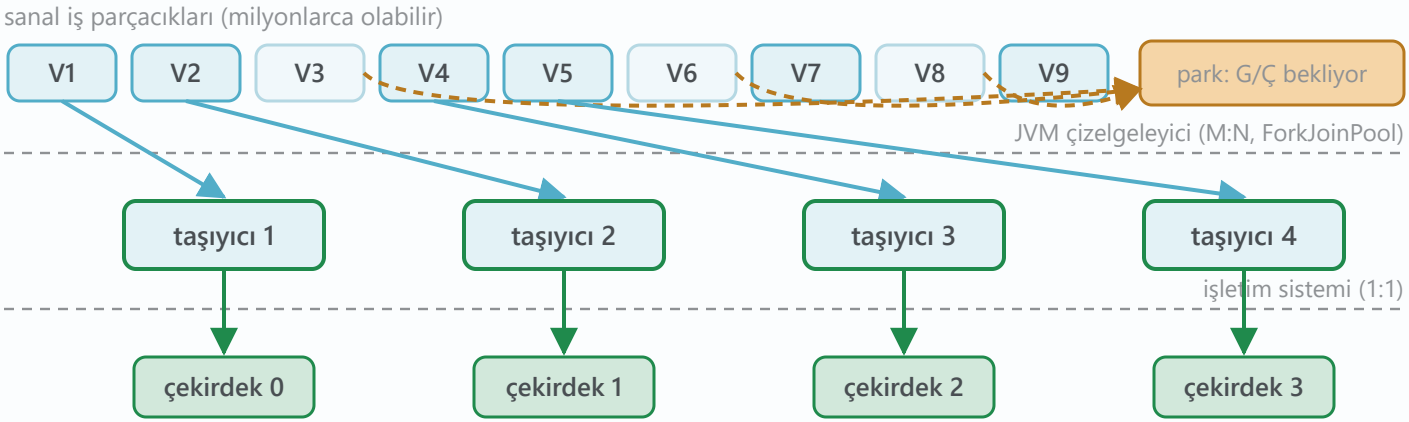
Modellerin Karşılaştırılması

Ölçüt	N:1	1:1	M:N
Geçiş maliyeti	çok düşük	orta (çekirdek)	düşük
Bloke eden çağrı	tüm süreç durur	yalnız o iş parçacığı	yalnız o iş parçacığı*
Çok çekirdek	kullanılamaz	kullanılır	kullanılır
İş parçacığı sayısı	çok fazla	çekirdek belleğiyle sınırlı	çok fazla
Gerçekleme	kolay (kütüphane)	kolay	zor (iki çizelgeleyici)

* Çalışma zamanı, bloke olan çağrıyı fark edip başka çekirdek iş parçacığına geçebiliyorsa.

İşletim sistemi düzeyinde M:N **terk edildi**: Solaris 9 ve FreeBSD 7 1:1'e geçti. Ancak M:N **dil çalışma zamanlarında** geri döndü: Go goroutine'leri, Erlang süreçleri, Java 21 sanal iş parçacıkları.

Güncel M:N: Sanal İş Parçacıkları



V3, V6, V8 bloke değil, park edildi: taşıyıcıları başka sanal iş parçacıklarını çalıştırır

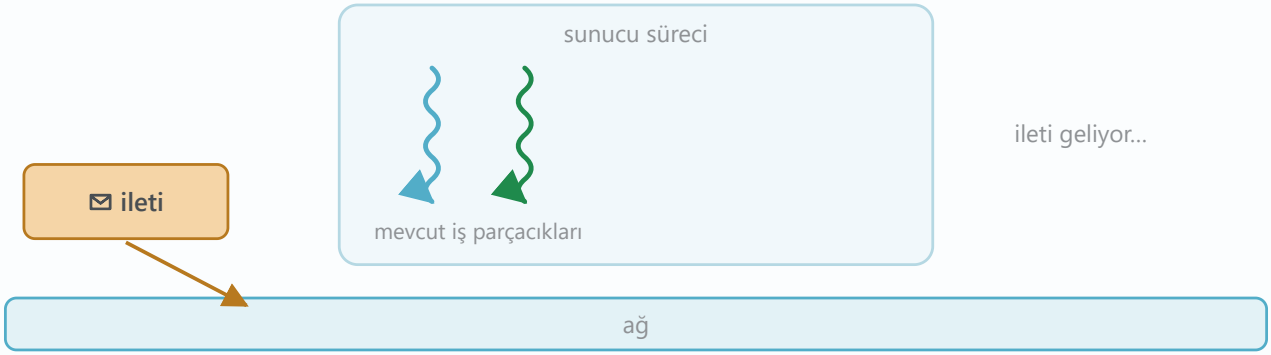
Java 21+ (Loom)

`Thread.ofVirtual().start(r)`; bloke eden G/Ç çağrısında JVM sanal iş parçacığını taşıyıcıdan indirir (unmount).

Go

G (goroutine) · M (İS iş parçacığı) · P (mantıksal işlemci, `GOMAXPROCS`); boştaki P iş çalar.

Açılır (Pop-up) İş Parçacıkları



(a) İleti gelmeden önce: süreçte bu iletiyi bekleyen bir iş parçacığı yoktur.

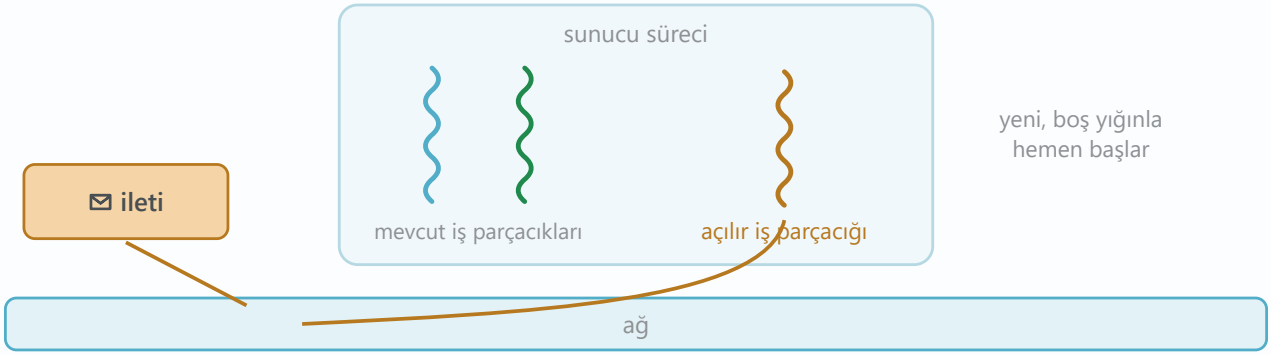
Adım 1 / 2

Alternatif: `receive` çağrısında bloke olmuş bir iş parçacığı; onun durumunun (yazmaçlar, yığın) her iletide geri yüklenmesi gerekir.

Açılır iş parçacığı yenidir, geri yüklenecek geçmişi yoktur: **gecikme düşüktür**. Çekirdek içinde de kullanılabilir (kesme işleme).

```
/* mesaj gelince */
pthread_t t;
pthread_create(&t, NULL, isle, msg);
pthread_detach(t); /* join yok */
```

Açılır (Pop-up) İş Parçacıkları



(b) İleti gelince sistem onu işleyecek **yeni bir iş parçacığı** oluşturur.

Alternatif: `receive` çağrısında bloke olmuş bir iş parçacığı; onun durumunun (yazmaçlar, yığın) her iletide geri yüklenmesi gerekir.

Açılır iş parçacığı yenidir, geri yüklenecek geçmişi yoktur: **gecikme düşüktür**. Çekirdek içinde de kullanılabilir (kesme işleme).

```
/* mesaj gelince */
pthread_t t;
pthread_create(&t, NULL, isle, msg);
pthread_detach(t); /* join yok */
```

İş Parçacığı Kütüphaneleri

POSIX Pthreads

Linux, macOS, BSD; C API'si. Oluşturma, bekleme, mutex, koşul değişkeni.

Windows API

`CreateThread`, `WaitForSingleObject`; çekirdek düzeyi, 1:1.

Java

`Thread`, `Runnable`, `ExecutorService`; JVM, işletim sistemi iş parçacıklarını (ve Java 21'den beri sanal iş parçacıklarını) kullanır.

C ve C++ standardı

C11 `<threads.h>`; C++11 `std::thread`, C++20 `std::jthread` (eski Boost.Thread'in yerini aldı).

Python

`threading`; GIL yüzünden CPU işinde paralellik yok. 3.13'te deneysel, 3.14'te desteklenen GIL'siz derleme.

Go

`go f()` ile goroutine; M:N çalışma zamanı, kanallarla iletişim.

Eş zamansız iş parçacığı: ebeveyn çocuğu başlatıp kendi işine devam eder (ör. sunucu). **Senkron**: ebeveyn çocuklarının bitmesini bekler (`join`) ve sonuçlarını birleştirir: fork-join.

POSIX Pthreads

- IEEE 1003.1c (POSIX.1c) standardı: iş parçacığı oluşturma ve senkronizasyon **API'si**.
- Bir **tanımlamadır**, gerçekleme değil: kütüphanenin nasıl davranacağını söyler; kullanıcı ya da çekirdek düzeyinde gerçekleştirilebilir.
- Linux'ta glibc NPTL (1:1), macOS ve BSD'lerde de çekirdek iş parçacıkları üzerinde çalışır.
- Derleme: `gcc prog.c -pthread`.

Çağrı	Açıklama
`pthread_create`	Yeni iş parçacığı oluşturur, verilen fonksiyonu çalıştırır.
`pthread_exit`	Çağıran iş parçacığını sonlandırır.
`pthread_join`	Belirli bir iş parçacığının bitmesini bekler.
`pthread_detach`	Kaynakları bitince kendiliğinden geri verilir; join edilmez.
`pthread_self`	Kendi kimliğini döndürür.
`sched_yield`	CPU'yu bırakır. (`pthread_yield` standart dışıdır, glibc 2.34'te kullanımdan kaldırıldı.)
`pthread_attr_init` / `_destroy`	Öznitelik yapısını (yığın boyutu, ayrık durum…) ilklendirir / kaldırır.

Pthreads: Oluştur ve Bekle

```c
int toplam;           /* paylaşılan */
void *runner(void *param) {
  int ust = atoi(param);
  toplam = 0;
  for (int i = 1; i <= ust; i++)
    toplam += i;
  pthread_exit(0);
}
int main(void) {
  pthread_t tid;
  pthread_create(&tid, NULL, runner, "5");
  pthread_join(tid, NULL);
  printf("toplam = %d\n", toplam);
  return 0;
}
```

main — main çalışıyor

runner — henüz yok

Süreç tek iş parçacığıyla (main) başlar; global `toplam` 0'dır.

Adım 1 / 7

DEĞİŞKENLER

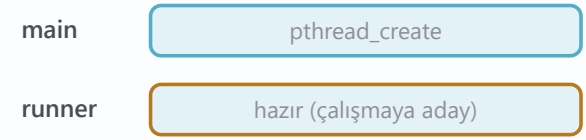
`toplam = 0`

ÇIKTI

Silberschatz Şekil 4.11'den uyarlanmıştır; `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` satırları gösterilmedi.

Pthreads: Oluştur ve Bekle

```
1  int toplam;           /* paylaşılan */
2  void *runner(void *param) {
3    int ust = atoi(param);
4    toplam = 0;
5    for (int i = 1; i <= ust; i++)
6      toplam += i;
7    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
▶    pthread_create(&tid, NULL, runner, "5");
12   pthread_join(tid, NULL);
13   printf("toplam = %d\n", toplam);
14   return 0;
15 }
```



`pthread_create` yeni iş parçacığını oluşturur; `runner("5")` çalışacaktır.

Adım 2 / 7

DEĞİŞKENLER

`toplam = 0` `tid = T1`

ÇIKTI

Silberschatz Şekil 4.11'den uyarlanmıştır; `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` satırları gösterilmedi.

Pthreads: Oluştur ve Bekle

```
1  int toplam;           /* paylaşılan */
2  void *runner(void *param) {
3    int ust = atoi(param);
4    toplam = 0;
5    for (int i = 1; i <= ust; i++)
6      toplam += i;
7    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
11   pthread_create(&tid, NULL, runner, "5");
▶    pthread_join(tid, NULL);
13   printf("toplam = %d\n", toplam);
14   return 0;
15 }
```

main — pthread_join → bekliyor

runner — çalışıyor

main `join`'de bloke olur. runner'ın ne zaman başlayacağını çizelgeleyici belirler.

Adım 3 / 7

DEĞİŞKENLER

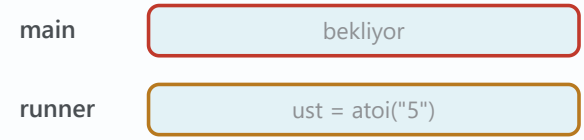
`toplam = 0` `tid = T1`

ÇIKTI

Silberschatz Şekil 4.11'den uyarlanmıştır; `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` satırları gösterilmedi.

Pthreads: Oluştur ve Bekle

```
1  int toplam;           /* paylaşılan */
2  void *runner(void *param) {
▶    int ust = atoi(param);
4    toplam = 0;
5    for (int i = 1; i <= ust; i++)
6      toplam += i;
7    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
11   pthread_create(&tid, NULL, runner, "5");
12   pthread_join(tid, NULL);
13   printf("toplam = %d\n", toplam);
14   return 0;
15 }
```



runner kendi yığınında yerel `ust` değişkenini oluşturur.

Adım 4 / 7

DEĞİŞKENLER

`toplam = 0` `tid = T1` `ust = 5`

ÇIKTI

Silberschatz Şekil 4.11'den uyarlanmıştır; `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` satırları gösterilmedi.

Pthreads: Oluştur ve Bekle

```
1  int toplam;           /* paylaşılan */
2  void *runner(void *param) {
3    int ust = atoi(param);
4    toplam = 0;
▶    for (int i = 1; i <= ust; i++)
▶      toplam += i;
7    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
11   pthread_create(&tid, NULL, runner, "5");
12   pthread_join(tid, NULL);
13   printf("toplam = %d\n", toplam);
14   return 0;
15 }
```

main — bekliyor

runner — döngü: 1+2+3+4+5

Döngü paylaşılan `toplam`'ı günceller: 15.

Adım 5 / 7

DEĞİŞKENLER

`toplam = 15` `tid = T1` `ust = 5`

ÇIKTI

Silberschatz Şekil 4.11'den uyarlanmıştır; `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` satırları gösterilmedi.

Pthreads: Oluştur ve Bekle

```
1  int toplam;           /* paylaşılan */
2  void *runner(void *param) {
3    int ust = atoi(param);
4    toplam = 0;
5    for (int i = 1; i <= ust; i++)
6      toplam += i;
▶    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
11   pthread_create(&tid, NULL, runner, "5");
12   pthread_join(tid, NULL);
13   printf("toplam = %d\n", toplam);
14   return 0;
15 }
```

main — uyanıyor

runner — pthread_exit(0) → bitti

runner sonlanır; `join` döner ve main uyanır.

Adım 6 / 7

DEĞİŞKENLER

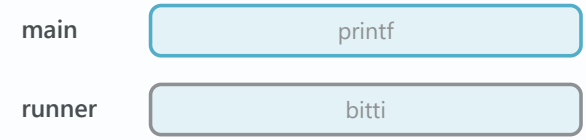
`toplam = 15` `tid = T1`

ÇIKTI

Silberschatz Şekil 4.11'den uyarlanmıştır; `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` satırları gösterilmedi.

Pthreads: Oluştur ve Bekle

```
1  int toplam;           /* paylaşılan */
2  void *runner(void *param) {
3    int ust = atoi(param);
4    toplam = 0;
5    for (int i = 1; i <= ust; i++)
6      toplam += i;
7    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
11   pthread_create(&tid, NULL, runner, "5");
12   pthread_join(tid, NULL);
▶    printf("toplam = %d\n", toplam);
14   return 0;
15 }
```



join sayesinde çıktı her çalıştırmada aynıdır: **toplam = 15**.

Adım 7 / 7

DEĞİŞKENLER

`toplam = 15` `tid = T1`

ÇIKTI

```
toplam = 15
```

Silberschatz Şekil 4.11'den uyarlanmıştır; `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` satırları gösterilmedi.

Birden Çok İş Parçacığı

```c
#define N 4
long kismi[N];           /* her biri kendi hücresine */

void *topla(void *arg) {
  long k = (long)arg;    /* 0 .. N-1 */
  for (long i = k*250 + 1; i <= (k+1)*250; i++)
    kismi[k] += i;
  return NULL;
}
int main(void) {
  pthread_t t[N];
  for (long k = 0; k < N; k++)
    pthread_create(&t[k], NULL, topla, (void *)k);
  long toplam = 0;
  for (int k = 0; k < N; k++) {
    pthread_join(t[k], NULL);
    toplam += kismi[k];
  }
  printf("%ld\n", toplam);  /* 500500 */
}
```

- 1..1000 toplamı dört iş parçacığına bölünür: her biri 250 sayı toplar (**veri paralelliği**).
- Her iş parçacığı **yalnızca kendi** `kismi[k]` hücresine yazar: yarış yoktur.
- main hepsini `join` ile bekler ve kısmi toplamları birleştirir: sonuç her zaman **500500**.
- Argüman olarak `&k` geçmek hatalı olurdu: döngü değişkeni değişmeye devam eder.
- Komşu hücreler aynı önbellek satırındadır: yoğun yazmada *false sharing* olur; yerel değişkende toplayıp sonda yazmak daha iyidir.

Argüman Geçirme ve Sonuç Alma

```
1  typedef struct { int alt, ust; } aralik_t;
2  typedef struct { long toplam; int adet; } sonuc_t;
3
4  void *topla(void *arg) {
5    aralik_t *a = arg;          /* void* → tür */
6    sonuc_t *s = malloc(sizeof *s); /* heap'te */
7    s->toplam = 0; s->adet = 0;
8    for (int i = a->alt; i <= a->ust; i++) {
9      s->toplam += i; s->adet++;
10   }
11   return s;         /* = pthread_exit(s) */
12 }
13 int main(void) {
▶  14   aralik_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, topla, &a);
17   pthread_join(t, &ret);          /* ret = s */
18   sonuc_t *s = ret;
19   printf("%d sayı, toplam %ld\n", s->adet, s->toplam);
20   free(s);
21   return 0;
22 }
```



Argüman main'in yığınındaki `a` yapısıdır. main T'yi join edene kadar bu çerçeve yaşar, bu yüzden adresini vermek güvenlidir.

Adım 1 / 7

DEĞİŞKENLER

`a.alt = 1` `a.ust = 100`

ÇIKTI

Argüman Geçirme ve Sonuç Alma

```c
typedef struct { int alt, ust; } aralik_t;
typedef struct { long toplam; int adet; } sonuc_t;

void *topla(void *arg) {
  aralik_t *a = arg;          /* void* → tür */
  sonuc_t *s = malloc(sizeof *s); /* heap'te */
  s->toplam = 0; s->adet = 0;
  for (int i = a->alt; i <= a->ust; i++) {
    s->toplam += i; s->adet++;
  }
  return s;          /* = pthread_exit(s) */
}
int main(void) {
  aralik_t a = { 1, 100 };
  pthread_t t;  void *ret;
  pthread_create(&t, NULL, topla, &a);
  pthread_join(t, &ret);          /* ret = s */
  sonuc_t *s = ret;
  printf("%d sayı, toplam %ld\n", s->adet, s->toplam);
  free(s);
  return 0;
}
```



Tek bir `void *` parametre vardır: birden çok değer bir yapıya konup **adresi** geçirilir.

Adım 2 / 7

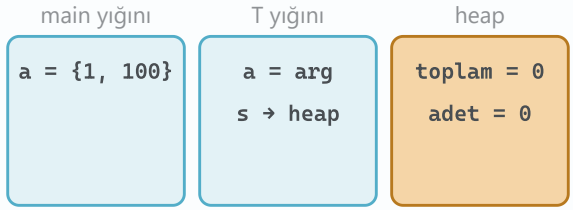
DEĞİŞKENLER

`a.alt = 1` `a.ust = 100` `t = T1`

ÇIKTI

Argüman Geçirme ve Sonuç Alma

```
1  typedef struct { int alt, ust; } aralik_t;
2  typedef struct { long toplam; int adet; } sonuc_t;
3
4  void *topla(void *arg) {
▶    aralik_t *a = arg;          /* void* → tür */
▶    sonuc_t *s = malloc(sizeof *s); /* heap'te */
7    s->toplam = 0; s->adet = 0;
8    for (int i = a->alt; i <= a->ust; i++) {
9      s->toplam += i; s->adet++;
10   }
11   return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   aralik_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, topla, &a);
17   pthread_join(t, &ret);          /* ret = s */
18   sonuc_t *s = ret;
19   printf("%d sayı, toplam %ld\n", s->adet, s->toplam);
20   free(s);
21   return 0;
22 }
```



T, `void *` 'ı gerçek türüne çevirir. Sonuç **heap'te** ayrılır: T'nin kendi yığını, T bitince yok olacaktır.

Adım 3 / 7

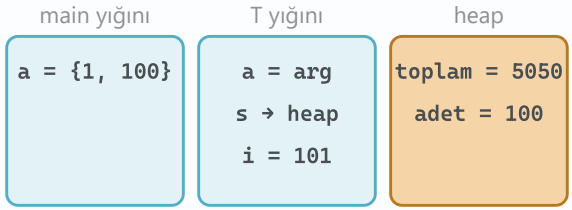
DEĞİŞKENLER

`a.alt = 1` `a.ust = 100` `t = T1` `s = heap'te`

ÇIKTI

Argüman Geçirme ve Sonuç Alma

```
1  typedef struct { int alt, ust; } aralik_t;
2  typedef struct { long toplam; int adet; } sonuc_t;
3
4  void *topla(void *arg) {
5    aralik_t *a = arg;          /* void* → tür */
6    sonuc_t *s = malloc(sizeof *s); /* heap'te */
7    s->toplam = 0; s->adet = 0;
▶    for (int i = a->alt; i <= a->ust; i++) {
▶      s->toplam += i; s->adet++;
10   }
11   return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   aralik_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, topla, &a);
17   pthread_join(t, &ret);          /* ret = s */
18   sonuc_t *s = ret;
19   printf("%d sayı, toplam %ld\n", s->adet, s->toplam);
20   free(s);
21   return 0;
22 }
```



Döngü 1..100'ü toplar: heap'teki yapıda toplam = 5050, adet = 100.

Adım 4 / 7

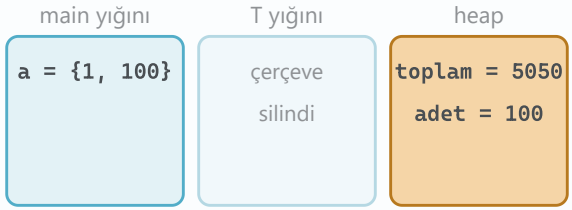
DEĞİŞKENLER

`a.alt = 1` `a.ust = 100` `t = T1` `s.toplam = 5050` `s.adet = 100`

ÇIKTI

Argüman Geçirme ve Sonuç Alma

```
1  typedef struct { int alt, ust; } aralik_t;
2  typedef struct { long toplam; int adet; } sonuc_t;
3
4  void *topla(void *arg) {
5    aralik_t *a = arg;          /* void* → tür */
6    sonuc_t *s = malloc(sizeof *s); /* heap'te */
7    s->toplam = 0; s->adet = 0;
8    for (int i = a->alt; i <= a->ust; i++) {
9      s->toplam += i; s->adet++;
10   }
▶    return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   aralik_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, topla, &a);
17   pthread_join(t, &ret);          /* ret = s */
18   sonuc_t *s = ret;
19   printf("%d sayı, toplam %ld\n", s->adet, s->toplam);
20   free(s);
21   return 0;
22 }
```



`return s` : işaretçi join'i bekleyene teslim edilir. T'nin yerel değişkenleri artık geçersizdir, heap'teki nesne ise **yaşar**.

Adım 5 / 7

DEĞİŞKENLER

`t = T1` `s.toplam = 5050` `s.adet = 100`

ÇIKTI

Argüman Geçirme ve Sonuç Alma

```
1  typedef struct { int alt, ust; } aralik_t;
2  typedef struct { long toplam; int adet; } sonuc_t;
3
4  void *topla(void *arg) {
5    aralik_t *a = arg;          /* void* → tür */
6    sonuc_t *s = malloc(sizeof *s); /* heap'te */
7    s->toplam = 0; s->adet = 0;
8    for (int i = a->alt; i <= a->ust; i++) {
9      s->toplam += i; s->adet++;
10   }
11   return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   aralik_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, topla, &a);
▶    pthread_join(t, &ret);          /* ret = s */
▶    sonuc_t *s = ret;
19   printf("%d sayı, toplam %ld\n", s->adet, s->toplam);
20   free(s);
21   return 0;
22 }
```



`pthread_join(t, &ret)` dönüş değerini `void **` üzerinden `ret` 'e yazar; tür dönüşümü main'de yapılır.

Adım 6 / 7

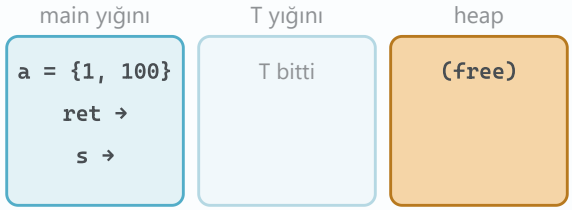
DEĞİŞKENLER

`ret = s` `s.toplam = 5050` `s.adet = 100`

ÇIKTI

Argüman Geçirme ve Sonuç Alma

```
1  typedef struct { int alt, ust; } aralik_t;
2  typedef struct { long toplam; int adet; } sonuc_t;
3
4  void *topla(void *arg) {
5    aralik_t *a = arg;          /* void* → tür */
6    sonuc_t *s = malloc(sizeof *s); /* heap'te */
7    s->toplam = 0; s->adet = 0;
8    for (int i = a->alt; i <= a->ust; i++) {
9      s->toplam += i; s->adet++;
10   }
11   return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   aralik_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, topla, &a);
17   pthread_join(t, &ret);          /* ret = s */
18   sonuc_t *s = ret;
▶    printf("%d sayı, toplam %ld\n", s->adet, s->toplam);
▶    free(s);
21   return 0;
22 }
```



Sonuç yazdırılır ve `free(s)` çağrılır: nesnenin sahipliği join ile main'e geçmişti.

Adım 7 / 7

DEĞİŞKENLER

`ret = s` `s.toplam = 5050` `s.adet = 100`

ÇIKTI

```
100 sayı, toplam 5050
```

Yaygın Hata: Yığındaki Sonucu Döndürmek

Hatalı

```
void *topla(void *arg) {
  sonuc_t s = { 0, 0 };  /* T'nin yığınında */
  /* ... s doldurulur ... */
  return &s;             /* HATA */
}
```

Doğru

```
void *topla(void *arg) {
  sonuc_t *s = malloc(sizeof *s);
  /* ... s doldurulur ... */
  return s;    /* heap'te: yaşar */
}
```



T çalışıyor: s, topla'nın yığın çerçevesinde

Hatalı sürümde sonuç, T'nin yığınındaki yerel `s` değişkenidir ve `&s` döndürülür.

Adım 1 / 3

Aynı kural argüman için de geçerlidir: iş parçacığına verilen adres, iş parçacığı onu kullandığı sürece yaşamalıdır. gcc çoğu durumda uyarır (`-Wreturn-local-addr`); bu uyarı hata gibi ele alınmalıdır.

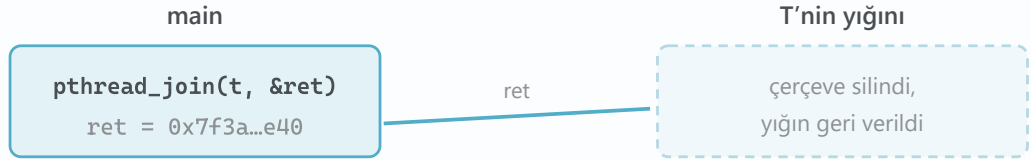
Yaygın Hata: Yığındaki Sonucu Döndürmek

Hatalı

```
void *topla(void *arg) {
  sonuc_t s = { 0, 0 };  /* T'nin yığınında */
  /* ... s doldurulur ... */
  return &s;             /* HATA */
}
```

Doğru

```
void *topla(void *arg) {
  sonuc_t *s = malloc(sizeof *s);
  /* ... s doldurulur ... */
  return s;    /* heap'te: yaşar */
}
```



T döndü: adres hâlâ main'de, ama gösterdiği yer artık kimsenin değil

Fonksiyon dönünce çerçeve silinir; iş parçacığı bitince yığınının tamamı geri verilir ya da başka bir iş parçacığına verilebilir.

Adım 2 / 3

Aynı kural argüman için de geçerlidir: iş parçacığına verilen adres, iş parçacığı onu kullandığı sürece yaşamalıdır. gcc çoğu durumda uyarır (`-Wreturn-local-addr`); bu uyarı hata gibi ele alınmalıdır.

Yaygın Hata: Yığındaki Sonucu Döndürmek

Hatalı

```
void *topla(void *arg) {
  sonuc_t s = { 0, 0 };  /* T'nin yığınında */
  /* ... s doldurulur ... */
  return &s;             /* HATA */
}
```

Doğru

```
void *topla(void *arg) {
  sonuc_t *s = malloc(sizeof *s);
  /* ... s doldurulur ... */
  return s;    /* heap'te: yaşar */
}
```



main okur: tanımsız davranış

main bu adresi okursa sonuç **tanımsızdır**: çöp değer, SIGSEGV ya da en kötüsü bazen doğru görünen bir değer. Sağdaki sürüm heap'i kullanır ve her zaman doğrudur.

Adım 3 / 3

Aynı kural argüman için de geçerlidir: iş parçacığına verilen adres, iş parçacığı onu kullandığı sürece yaşamalıdır. gcc çoğu durumda uyarır (`-Wreturn-local-addr`); bu uyarı hata gibi ele alınmalıdır.

Java: Thread, start ve join

```
1  public class Topla {
2    static int toplam = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
▶      Thread t = new Thread(() -> {
▶        for (int i = 1; i <= 5; i++) toplam += i;
▶      });
8      t.start();
9      System.out.println("main: bekliyor");
10     t.join();
11     System.out.println("toplam = " + toplam);
12   }
13 }
```

main	RUNNABLE
t	NEW (oluşturuldu, başlamadı)

`new Thread(lambda)` yalnızca nesneyi oluşturur; durum **NEW**.

Adım 1 / 6

DEĞİŞKENLER

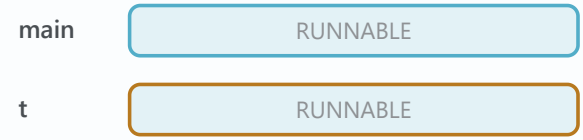
`toplam = 0` `t = NEW`

ÇIKTI

JDK 21 ile derlenip çalıştırıldı; çıktı her seferinde aynıdır.

Java: Thread, start ve join

```
1  public class Topla {
2    static int toplam = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
5      Thread t = new Thread(() -> {
6        for (int i = 1; i <= 5; i++) toplam += i;
7      });
▶      t.start();
9      System.out.println("main: bekliyor");
10     t.join();
11     System.out.println("toplam = " + toplam);
12   }
13 }
```



`start()` yeni işletim sistemi iş parçacığını başlatır (`run()` çağırmak başlatmazdı!).

Adım 2 / 6

DEĞİŞKENLER

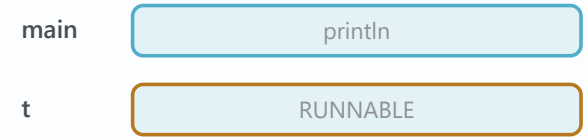
`toplam = 0` `t = RUNNABLE`

ÇIKTI

JDK 21 ile derlenip çalıştırıldı; çıktı her seferinde aynıdır.

Java: Thread, start ve join

```
1  public class Topla {
2    static int toplam = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
5      Thread t = new Thread(() -> {
6        for (int i = 1; i <= 5; i++) toplam += i;
7      });
8      t.start();
▶      System.out.println("main: bekliyor");
10     t.join();
11     System.out.println("toplam = " + toplam);
12   }
13 }
```



main yazmaya devam eder; t ile aynı anda çalışabilir.

Adım 3 / 6

DEĞİŞKENLER

`toplam = 0` `t = RUNNABLE`

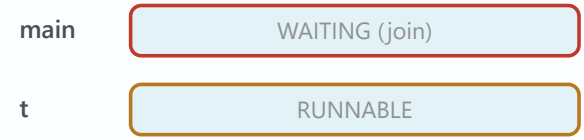
ÇIKTI

```
main: bekliyor
```

JDK 21 ile derlenip çalıştırıldı; çıktı her seferinde aynıdır.

Java: Thread, start ve join

```
1  public class Topla {
2    static int toplam = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
5      Thread t = new Thread(() -> {
6        for (int i = 1; i <= 5; i++) toplam += i;
7      });
8      t.start();
9      System.out.println("main: bekliyor");
▶      t.join();
11     System.out.println("toplam = " + toplam);
12   }
13 }
```



`join()` : main, t bitene kadar **WAITING** durumunda.

Adım 4 / 6

DEĞİŞKENLER

`toplam = 0` `t = RUNNABLE`

ÇIKTI

```
main: bekliyor
```

JDK 21 ile derlenip çalıştırıldı; çıktı her seferinde aynıdır.

Java: Thread, start ve join

```
1  public class Topla {
2    static int toplam = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
5      Thread t = new Thread(() -> {
▶        for (int i = 1; i <= 5; i++) toplam += i;
7      });
8      t.start();
9      System.out.println("main: bekliyor");
10     t.join();
11     System.out.println("toplam = " + toplam);
12   }
13 }
```

main	uyanıyor
t	TERMINATED

t döngüyü bitirir: toplam = 15, durum **TERMINATED**.

Adım 5 / 6

DEĞİŞKENLER

`toplam = 15` `t = TERMINATED`

ÇIKTI

```
main: bekliyor
```

JDK 21 ile derlenip çalıştırıldı; çıktı her seferinde aynıdır.

Java: Thread, start ve join

```
1  public class Topla {
2    static int toplam = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
5      Thread t = new Thread(() -> {
6        for (int i = 1; i <= 5; i++) toplam += i;
7      });
8      t.start();
9      System.out.println("main: bekliyor");
10     t.join();
▶      System.out.println("toplam = " + toplam);
12   }
13 }
```



join, t'nin yazdıklarının main tarafından görülmesini de garanti eder (happens-before).

Adım 6 / 6

DEĞİŞKENLER

`toplam = 15` `t = TERMINATED`

ÇIKTI

```
main: bekliyor
toplam = 15
```

JDK 21 ile derlenip çalıştırıldı; çıktı her seferinde aynıdır.

Java 21: Sanal İş Parçacıkları

```java
import java.time.Duration;
import java.util.concurrent.*;
import java.util.stream.IntStream;

public class Sanal {
  public static void main(String[] args) {
    try (var ex = Executors.newVirtualThreadPerTaskExecutor()) {
      IntStream.range(0, 10_000).forEach(i ->
        ex.submit(() -> {
          Thread.sleep(Duration.ofSeconds(1));
          return i;
        }));
    }  // close(): tüm görevleri bekler
    System.out.println("bitti");
  }
}
```

- 10.000 görevin her biri kendi **sanal iş parçacığında** 1 sn uyur.
- Ölçümümüzde (JDK 21) tümü yaklaşık **2 saniyede** bitti: birkaç taşıyıcı iş parçacığı yetti.
- 10.000 platform iş parçacığı ise 10.000 çekirdek iş parçacığı ve yığını demektir.
- Uygun alan: çok sayıda **G/Ç bekleyen** görev (sunucular). CPU yoğun işte kazanç yoktur.
- Klasik oluşturma da çalışır: `Thread.ofVirtual().start(r)`, `Thread.ofPlatform()`.

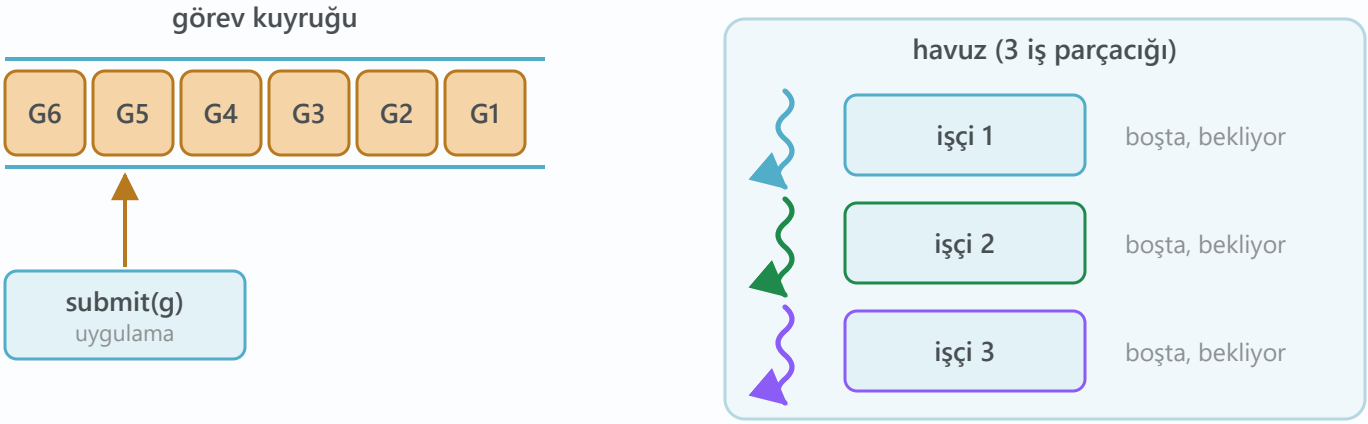
Örtük İş Parçacığı (Implicit Threading)

- İş parçacığı sayısı yüzlere çıkınca doğruluğu elle yönetmek zorlaşır.
- Çözüm: programcı **görevleri** (paralel çalışabilecek işleri) belirtir; iş parçacıklarını **derleyici ve çalışma zamanı kütüphaneleri** oluşturur ve yönetir.

Yaklaşım	Nerede	Fikir
İş parçacığı havuzu	Java `ExecutorService`, Windows thread pool	Hazır iş parçacıkları kuyruktaki görevleri alır.
Fork-join	Java `ForkJoinPool`, paralel akışlar	Görev böl, paralel çöz, sonuçları birleştir.
OpenMP	C, C++, Fortran	Derleyici yönergeleri ile paralel bölgeler.
Grand Central Dispatch	macOS, iOS (libdispatch)	Bloklar dağıtım kuyruklarına konur.
oneTBB (eski adı Intel TBB)	C++ şablon kütüphanesi	`parallel_for`, görev çizelgeleme, iş çalma.

ÖRTÜK İŞ PARÇACIĞI

İş Parçacığı Havuzu

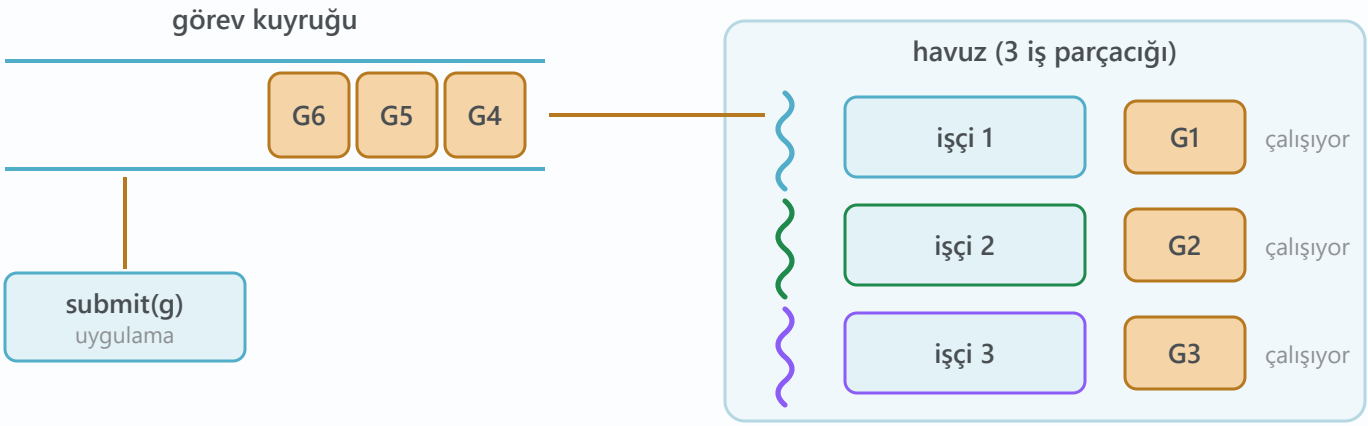


Havuz başlangıçta **sabit sayıda** iş parçacığı oluşturur; uygulama görevleri kuyruğa koyar.

Adım 1 / 4

Faydalar: daha hızlı hizmet, sistemi aşırı iş parçacığından koruma, görevin nasıl çalıştırılacağını (gecikmeli, periyodik) görevden ayırma.

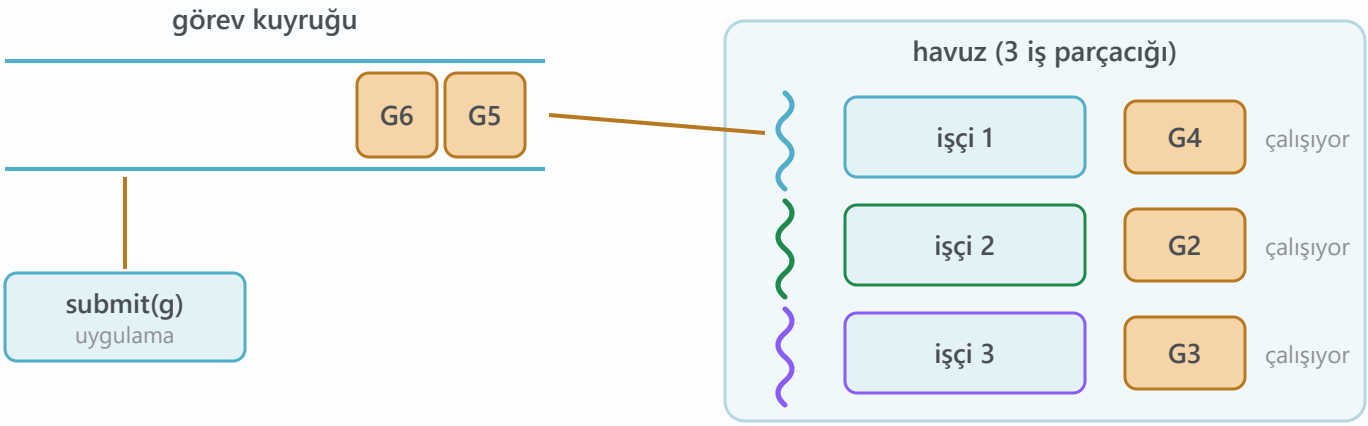
İş Parçacığı Havuzu



Boştaki işçiler kuyruktan görev alır: **yeni iş parçacığı oluşturma maliyeti yok**.

Faydalar: daha hızlı hizmet, sistemi aşırı iş parçacığından koruma, görevin nasıl çalıştırılacağını (gecikmeli, periyodik) görevden ayırma.

İş Parçacığı Havuzu

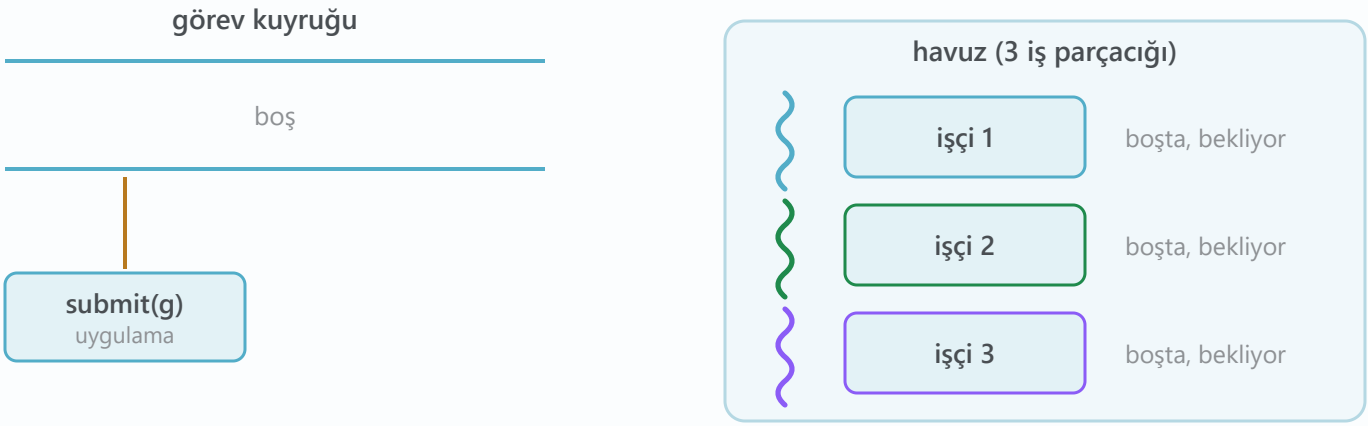


İşçi 1, G1'i bitirince yok olmaz, hemen sıradaki G4'ü alır.

Faydalar: daha hızlı hizmet, sistemi aşırı iş parçacığından koruma, görevin nasıl çalıştırılacağını (gecikmeli, periyodik) görevden ayırma.

ÖRTÜK İŞ PARÇACIĞI

İş Parçacığı Havuzu



Kuyruk boşalınca işçiler bir sonraki göreve kadar bekler. Eş zamanlı iş parçacığı sayısı havuz boyutuyla **sınırlıdır**.

Faydalar: daha hızlı hizmet, sistemi aşırı iş parçacığından koruma, görevin nasıl çalıştırılacağını (gecikmeli, periyodik) görevden ayırma.

ÖRTÜK İŞ PARÇACIĞI

Java ExecutorService

```
ExecutorService havuz = Executors.newFixedThreadPool(2);
List<Future<Integer>> sonuclar = new ArrayList<>();
for (int g = 1; g <= 4; g++) {
  final int n = g;
  sonuclar.add(havuz.submit(() -> n * n));
}
int toplam = 0;
for (Future<Integer> f : sonuclar)
  toplam += f.get();      // sonucu bekler
System.out.println("toplam = " + toplam);
havuz.shutdown();
```

ÇIKTI (JDK 21)

```
toplam = 30
```

- Dört görev (1^2, 2^2, 3^2, 4^2) iki iş parçacıklı havuza verilir.
- `submit` bir `Future` döndürür; `get()` sonuç hazır olana dek bekler.
- Havuz boyutu: CPU yoğun işte ≈ çekirdek sayısı (`availableProcessors()`); G/Ç yoğun işte daha fazla.
- Diğer fabrikalar: `newCachedThreadPool`, `newScheduledThreadPool`, `newVirtualThreadPerTaskExecutor`.

ÖRTÜK İŞ PARÇACIĞI

Fork-Join

görev

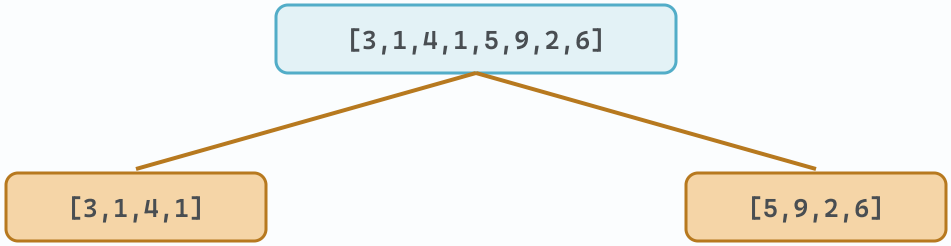
`[3,1,4,1,5,9,2,6]`

Büyük görev: 8 sayının toplamı. Eşikten büyükse **böl**.

Adım 1 / 5

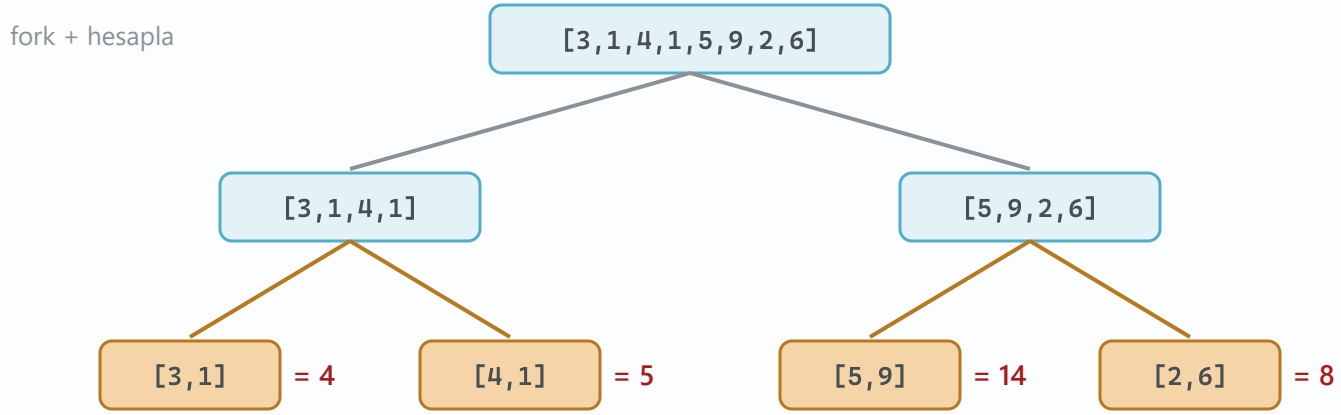
Fork-Join

fork



fork: iki alt görev oluşturulur; havuzdaki farklı iş parçacıklarında çalışabilirler.

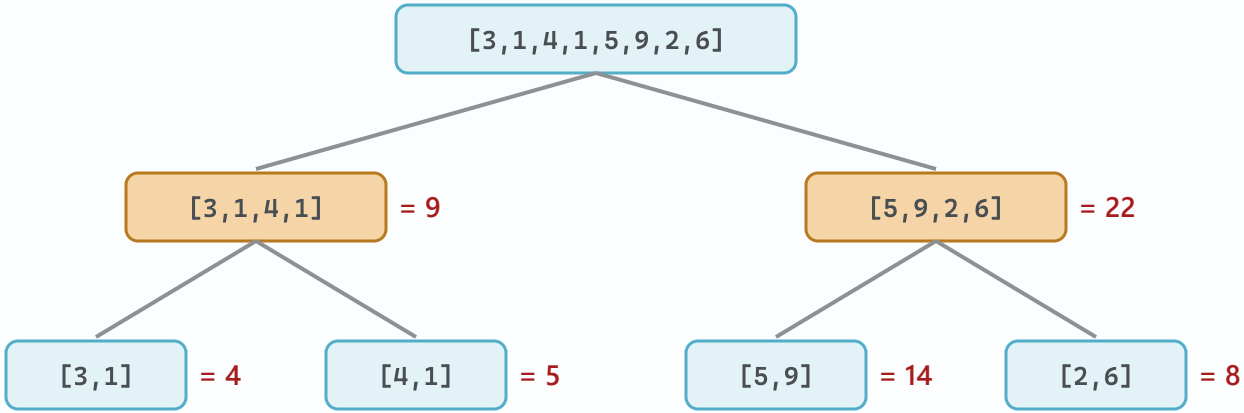
Fork-Join



Alt görevler yine bölünür; eşiğe (2 öğe) inen yapraklar doğrudan hesaplanır: 4, 5, 14, 8.

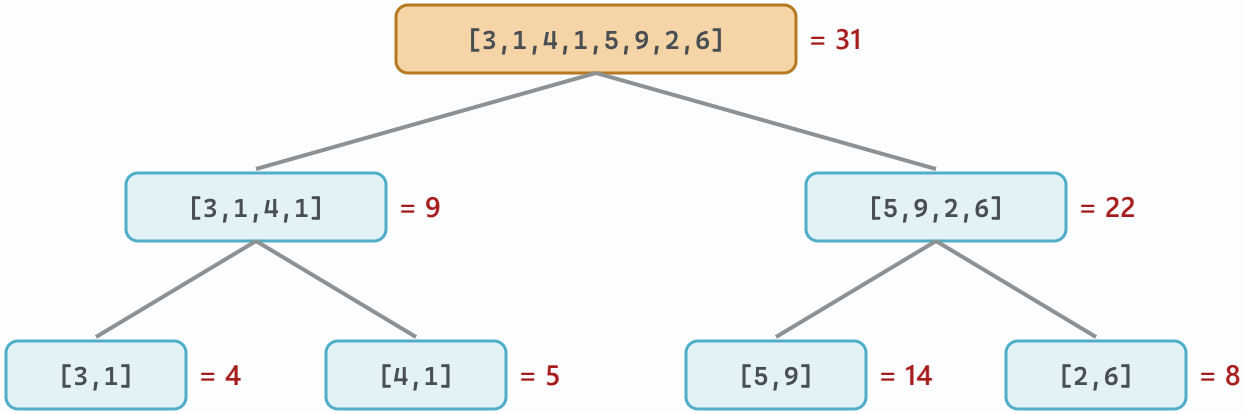
Fork-Join

join



join: ebeveyn çocuklarını bekler ve birleştirir: 4+5 = 9, 14+8 = 22.

Fork-Join

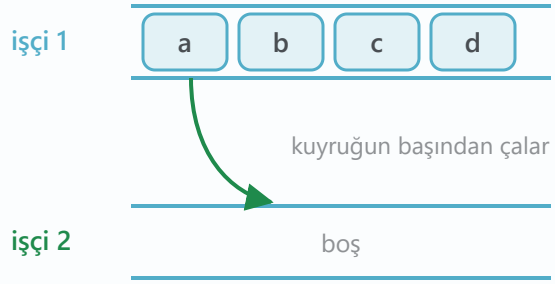


Kök sonuç: 9 + 22 = **31**. Senkron iş parçacığı kullanımının tipik örneği.

ÖRTÜK İŞ PARÇACIĞI

Java ForkJoinPool ve İş Çalma

```
class Toplam extends RecursiveTask<Long> {
  protected Long compute() {
    if (son - bas <= ESIK) {      // küçükse doğrudan
      long s = 0;
      for (int i = bas; i < son; i++) s += a[i];
      return s;
    }
    int orta = (bas + son) / 2;
    Toplam sol = new Toplam(a, bas, orta);
    Toplam sag = new Toplam(a, orta, son);
    sol.fork();                   // eş zamansız başlat
    long sagSonuc = sag.compute();
    return sol.join() + sagSonuc;
  }
}
long s = ForkJoinPool.commonPool()
           .invoke(new Toplam(a, 0, a.length));
```

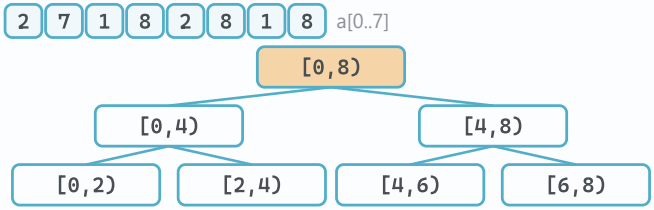


sahip kendi ucundan (sondan) alır

1..100 000 dizisi için çıktı (JDK 21): `toplam = 5000050000`. Boşta kalan işçi başkasının kuyruğundan görev **çalar** (work stealing); yük kendiliğinden dengelenir.

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
▶    if (son - bas <= ESIK) {
3      long s = 0;
4      for (int i = bas; i < son; i++)
5        s += a[i];
6      return s;
7    }
8    int orta = (bas + son) / 2;
9    Toplam sol = new Toplam(a, bas, orta);
10   Toplam sag = new Toplam(a, orta, son);
11   sol.fork();                    // kuyruğa
12   long sagSonuc = sag.compute();
13   return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): boş

Kök görev 8 öğeyi topluyor. 8 − 0 = 8 > ESIK, doğrudan hesaplanmaz.

Adım 1 / 11

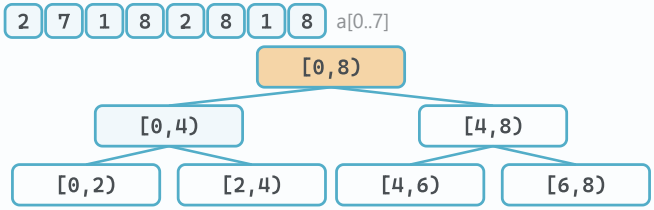
DEĞİŞKENLER

`bas = 0` `son = 8`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))` ; çıktı (JDK 21): `toplam = 37` .

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
2    if (son - bas <= ESIK) {
3      long s = 0;
4      for (int i = bas; i < son; i++)
5        s += a[i];
6      return s;
7    }
▶    int orta = (bas + son) / 2;
▶    Toplam sol = new Toplam(a, bas, orta);
▶    Toplam sag = new Toplam(a, orta, son);
▶    sol.fork();                    // kuyruğa
12   long sagSonuc = sag.compute();
13   return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): [0,4)

Görev ikiye bölünür. `sol.fork()` [0,4)'ü işçinin **kuyruğuna** koyar; hemen çalışmaz.

Adım 2 / 11

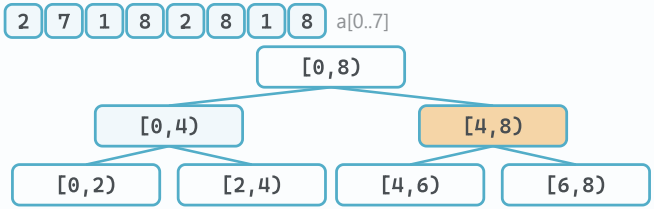
DEĞİŞKENLER

`bas = 0` `son = 8` `orta = 4`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))` ; çıktı (JDK 21): `toplam = 37` .

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
2    if (son - bas <= ESIK) {
3      long s = 0;
4      for (int i = bas; i < son; i++)
5        s += a[i];
6      return s;
7    }
8    int orta = (bas + son) / 2;
9    Toplam sol = new Toplam(a, bas, orta);
10   Toplam sag = new Toplam(a, orta, son);
11   sol.fork();                    // kuyruğa
▶    long sagSonuc = sag.compute();
13   return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): [0,4)

Sağ yarı aynı iş parçacığında, doğrudan `compute()` ile çalışır.

Adım 3 / 11

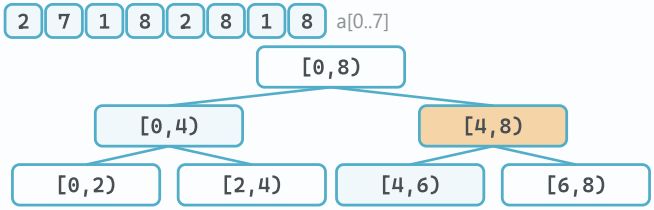
DEĞİŞKENLER

`bas = 4` `son = 8`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))`; çıktı (JDK 21): `toplam = 37`.

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
2    if (son - bas <= ESIK) {
3      long s = 0;
4      for (int i = bas; i < son; i++)
5        s += a[i];
6      return s;
7    }
▶    int orta = (bas + son) / 2;
▶    Toplam sol = new Toplam(a, bas, orta);
▶    Toplam sag = new Toplam(a, orta, son);
▶    sol.fork();                    // kuyruğa
12   long sagSonuc = sag.compute();
13   return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): [0,4) [4,6)

[4,8) de bölünür: [4,6) kuyruğa.

Adım 4 / 11

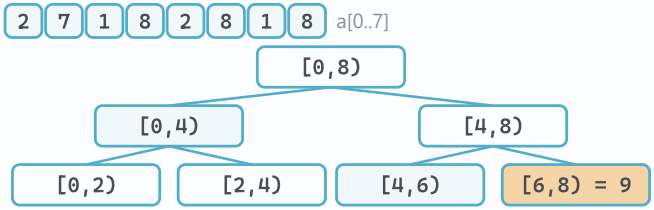
DEĞİŞKENLER

`bas = 4` `son = 8` `orta = 6`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))` ; çıktı (JDK 21): `toplam = 37` .

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
▶    if (son - bas <= ESIK) {
▶      long s = 0;
▶      for (int i = bas; i < son; i++)
▶        s += a[i];
▶      return s;
7    }
8    int orta = (bas + son) / 2;
9    Toplam sol = new Toplam(a, bas, orta);
10   Toplam sag = new Toplam(a, orta, son);
11   sol.fork();                    // kuyruğa
12   long sagSonuc = sag.compute();
13   return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): [0,4) [4,6)

[6,8): 2 öğe, eşikte. Doğrudan toplanır: 1 + 8 = **9**.

Adım 5 / 11

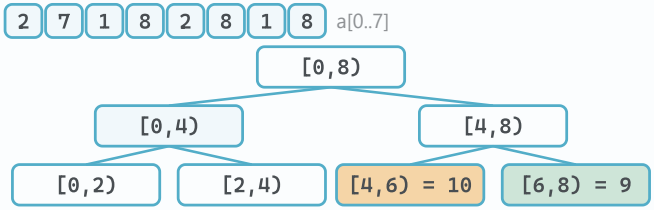
DEĞİŞKENLER

`bas = 6` `son = 8` `s = 9`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))` ; çıktı (JDK 21): `toplam = 37` .

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
2    if (son - bas <= ESIK) {
3      long s = 0;
4      for (int i = bas; i < son; i++)
5        s += a[i];
6      return s;
7    }
8    int orta = (bas + son) / 2;
9    Toplam sol = new Toplam(a, bas, orta);
10   Toplam sag = new Toplam(a, orta, son);
11   sol.fork();                    // kuyruğa
12   long sagSonuc = sag.compute();
▶    return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): [0,4)

`sol.join()` : [4,6) hâlâ kuyruğun ucunda, kimse çalmamış. İşçi onu çıkarıp **kendisi** çalıştırır: 2 + 8 = 10.

Adım 6 / 11

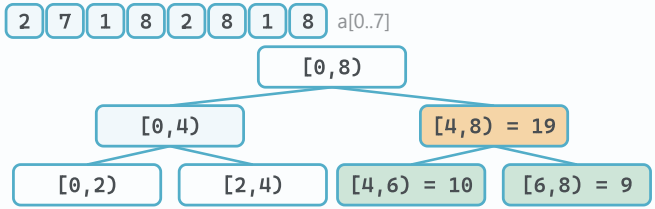
DEĞİŞKENLER

`bas = 4` `son = 6` `s = 10`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))` ; çıktı (JDK 21): `toplam = 37` .

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
2    if (son - bas <= ESIK) {
3      long s = 0;
4      for (int i = bas; i < son; i++)
5        s += a[i];
6      return s;
7    }
8    int orta = (bas + son) / 2;
9    Toplam sol = new Toplam(a, bas, orta);
10   Toplam sag = new Toplam(a, orta, son);
11   sol.fork();                    // kuyruğa
12   long sagSonuc = sag.compute();
▶    return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): [0,4)

[4,8) = 10 + 9 = **19**, köke döner.

Adım 7 / 11

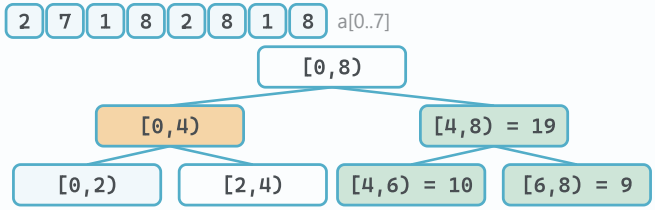
DEĞİŞKENLER

`bas = 4` `son = 8` `sagSonuc = 9`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))`; çıktı (JDK 21): `toplam = 37`.

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
2    if (son - bas <= ESIK) {
3      long s = 0;
4      for (int i = bas; i < son; i++)
5        s += a[i];
6      return s;
7    }
▶    int orta = (bas + son) / 2;
▶    Toplam sol = new Toplam(a, bas, orta);
▶    Toplam sag = new Toplam(a, orta, son);
▶    sol.fork();                    // kuyruğa
12   long sagSonuc = sag.compute();
▶    return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): [0,2)

Kök `sol.join()` çağırır: [0,4) kuyruktan çıkar ve çalışır; o da bölünür, [0,2) kuyruğa.

Adım 8 / 11

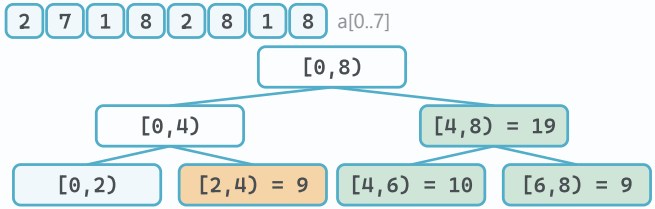
DEĞİŞKENLER

`bas = 0` `son = 4` `orta = 2`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))` ; çıktı (JDK 21): `toplam = 37` .

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
▶    if (son - bas <= ESIK) {
▶      long s = 0;
▶      for (int i = bas; i < son; i++)
▶        s += a[i];
▶      return s;
7    }
8    int orta = (bas + son) / 2;
9    Toplam sol = new Toplam(a, bas, orta);
10   Toplam sag = new Toplam(a, orta, son);
11   sol.fork();                    // kuyruğa
▶    long sagSonuc = sag.compute();
13   return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): [0,2)

[2,4) yaprak: 1 + 8 = **9**.

Adım 9 / 11

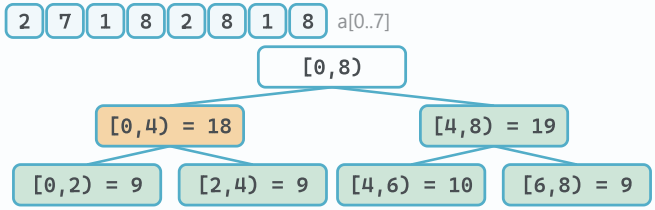
DEĞİŞKENLER

`bas = 2` `son = 4` `s = 9`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))` ; çıktı (JDK 21): `toplam = 37` .

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
2    if (son - bas <= ESIK) {
3      long s = 0;
4      for (int i = bas; i < son; i++)
5        s += a[i];
6      return s;
7    }
8    int orta = (bas + son) / 2;
9    Toplam sol = new Toplam(a, bas, orta);
10   Toplam sag = new Toplam(a, orta, son);
11   sol.fork();                    // kuyruğa
12   long sagSonuc = sag.compute();
▶    return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): boş

join [0,2) yaprağı çalıştırır: 2 + 7 = 9; [0,4) = 9 + 9 = **18**.

Adım 10 / 11

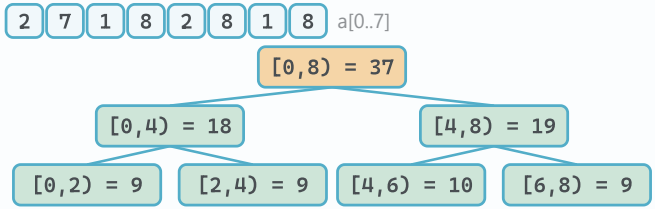
DEĞİŞKENLER

`bas = 0` `son = 4` `sagSonuc = 9`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))`; çıktı (JDK 21): `toplam = 37`.

Kod İzleme: Fork-Join ile Toplam

```
1  protected Long compute() {     // ESIK = 2
2    if (son - bas <= ESIK) {
3      long s = 0;
4      for (int i = bas; i < son; i++)
5        s += a[i];
6      return s;
7    }
8    int orta = (bas + son) / 2;
9    Toplam sol = new Toplam(a, bas, orta);
10   Toplam sag = new Toplam(a, orta, son);
11   sol.fork();                    // kuyruğa
12   long sagSonuc = sag.compute();
▶    return sol.join() + sagSonuc;
14 }
```



iş kuyruğu (deque): boş

Kök: 18 + 19 = **37**. Havuzda boşta işçi olsaydı kuyruktaki [0,4) gibi görevleri **çalabilirdi**; sıra değişir, sonuç değişmez.

Adım 11 / 11

DEĞİŞKENLER

`bas = 0` `son = 8` `sagSonuc = 19`

a = {2, 7, 1, 8, 2, 8, 1, 8}, tek işçili havuz: `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))` ; çıktı (JDK 21): `toplam = 37` .

ÖRTÜK İŞ PARÇACIĞI

OpenMP

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

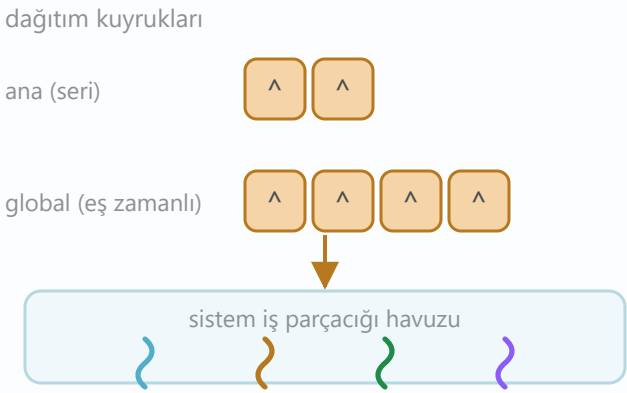
int main(void) {
  #pragma omp parallel
  printf("merhaba, ben %d\n", omp_get_thread_num());

  double a[1000], b[1000], c[1000], s = 0;
  /* ... a ve b doldurulur ... */
  #pragma omp parallel for reduction(+:s)
  for (int i = 0; i < 1000; i++) {
    c[i] = a[i] + b[i];
    s += c[i];
  }
  return 0;
}
```

- C, C++ ve Fortran için **derleyici yönergeleri** (compiler directives) ve kütüphane; paylaşımlı bellekli sistemler için.
- `#pragma omp parallel` : **paralel bölge**; varsayılan olarak çekirdek sayısı kadar iş parçacığı (`OMP_NUM_THREADS` ya da `num_threads(n)` ile ayarlanır).
- `parallel for` : döngü yinelemelerini iş parçacıklarına böler; `reduction` kısmi toplamları güvenle birleştirir.
- Derleme: `gcc -fopenmp` . “merhaba” satırlarının sırası her çalıştırmada değişebilir.

Grand Central Dispatch ve oneTBB

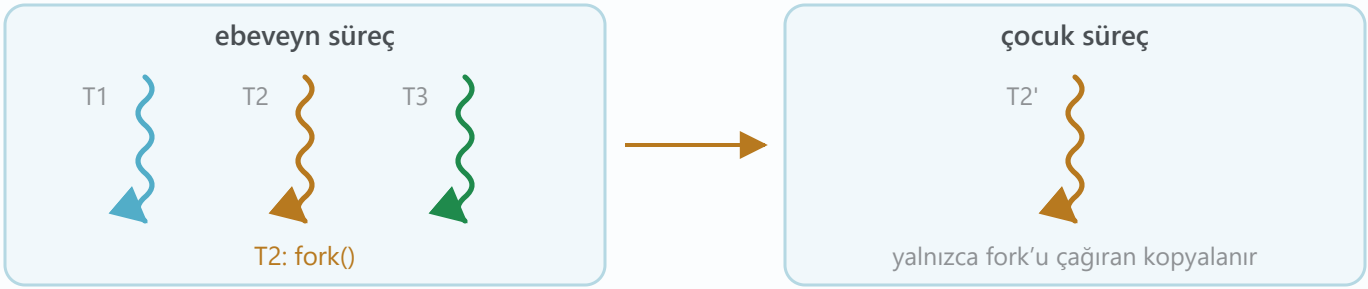
- Apple'ın macOS/iOS için çalışma zamanı kütüphanesi ve C/Objective-C dil eklentisi (**bloklar**): `^{ printf("Ben bir bloğum"); }`
- Bloklar **dağıtım kuyruklarına** konur; seri kuyruk sırayla, eş zamanlı kuyruk paralel çalıştırır. QoS sınıfları önceliği belirler.
- Kuyruktan çıkan blok havuzdaki boş bir iş parçacığına atanır; iş parçacığı yönetimi tamamen GCD'dedir.




```
dispatch_queue_t q = dispatch_get_global_queue(
                        QOS_CLASS_USER_INITIATED, 0);
dispatch_async(q, ^{ puts("Ben bir bloğum"); });
```

oneTBB (Intel Threading Building Blocks'un açık kaynak devamı): `tbb::parallel_for(0, n, [&](int i) { f(v[i]); });` Görevleri kendi çizelgeleyicisiyle iş çalma kullanarak iş parçacıklarına dağıtır. Swift'te GCD'nin yerini büyük ölçüde `async/await` almıştır.

fork() ve exec() İş Parçacıklarıyla



T1'in tuttuğu bir kilit çocukta sonsuza dek kilitli kalabilir!

- Bazı UNIX'ler iki `fork` sürümü sunmuştu (tüm iş parçacıklarını ya da yalnızca çağıranı kopyalayan). **POSIX:** yalnızca çağıran iş parçacığı kopyalanır.
- `fork` 'tan hemen sonra `exec` çağrılacaksa bu doğrudur: yeni program zaten tüm adres alanını değiştirir.
- Çocukta `exec` 'e kadar yalnızca async-signal-safe fonksiyonlar güvenlidir; `pthread_atfork` kilitleri düzenlemeye yarar. Çoğu zaman `posix_spawn` daha iyi seçimdir.

Sinyal İşleme

Sinyal: sürece bir olayın olduğunu bildirir. Çok iş parçacıklı süreçte soru: **hangi iş parçacığına** teslim edilmeli?

Seçenek	Örnek
Sinyalin ilgili olduğu iş parçacığına	senkron: `SIGSEGV`, `SIGFPE`
Süreçteki her iş parçacığına	bazı sonlandırma durumları
Sinyali bloke etmeyen herhangi birine	POSIX süreç sinyali: `SIGINT`
Tüm sinyalleri alan özel bir iş parçacığına	`sigwait` döngüsü

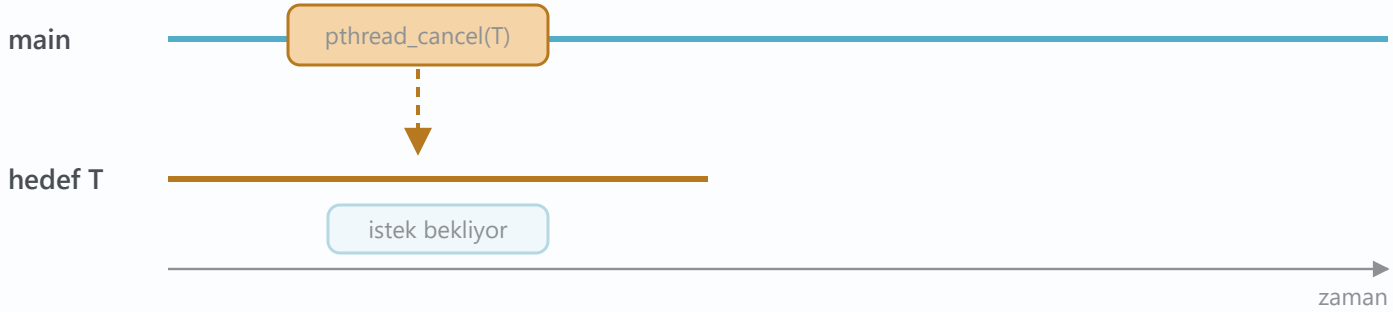
- Sinyal **işleyicileri** süreç geneli, sinyal **maskesi** iş parçacığına özeldir: `pthread_sigmask`.
- Belirli bir iş parçacığına: `pthread_kill(tid, sig)`.
- İyi uygulama: tüm iş parçacıklarında engelle, bir iş parçacığı `sigwait` ile beklesin.
- Kullanıcı düzeyi iş parçacıklarında çekirdek doğru iş parçacığını **adresleyemez**; yalnızca süreci tanır.
- Windows'ta sinyal yok; benzeri: APC (asynchronous procedure call).



İş Parçacığı İptali

Eş zamansız iptal
Hedef *hemen* sonlandırılır. Kilit tutarken ya da veriyi yarım güncellemişken ölebilir: tehlikeli.

Ertelenmiş iptal (varsayılan)
Hedef iptal isteğini belirli *iptal noktalarında* denetler ve düzenli biçimde sonlanır.



main `pthread_cancel(T)` çağırır; ertelenmiş modda bu yalnızca bir **istektir**, T çalışmaya devam eder.

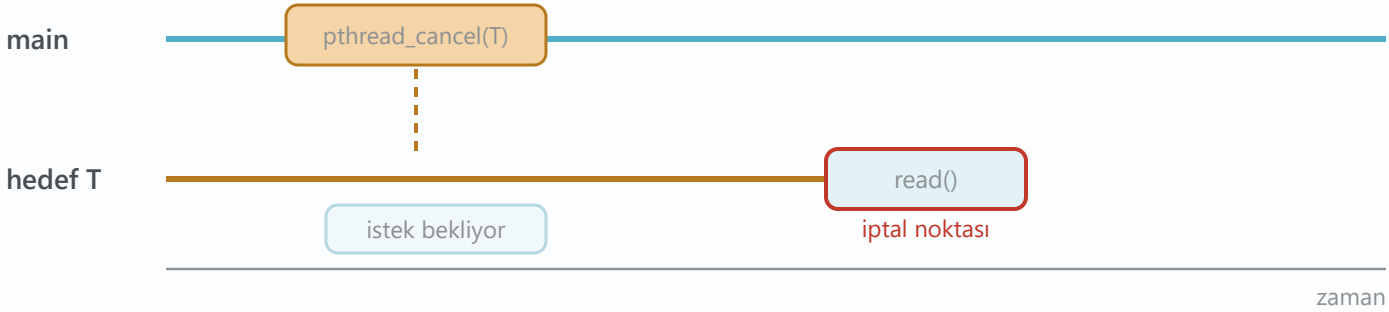
İş Parçacığı İptali

Eş zamansız iptal

Hedef *hemen* sonlandırılır. Kilit tutarken ya da veriyi yarım güncellemişken ölebilir: tehlikeli.

Ertelenmiş iptal (varsayılan)

Hedef iptal isteğini belirli *iptal noktalarında* denetler ve düzenli biçimde sonlanır.



T bir **iptal noktasına** gelir: `read`, `sleep`, `pthread_cond_wait`, `pthread_testcancel()` ...

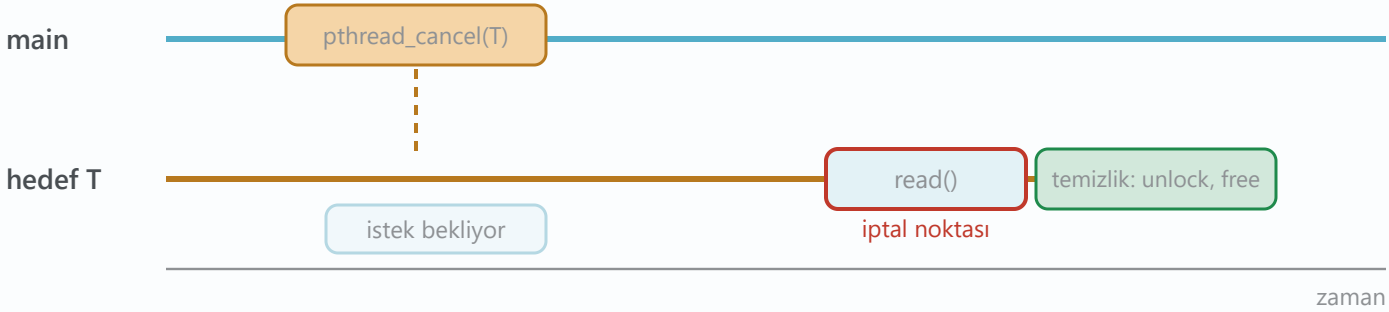
İş Parçacığı İptali

Eş zamansız iptal

Hedef *hemen* sonlandırılır. Kilit tutarken ya da veriyi yarım güncellemişken ölebilir: tehlikeli.

Ertelenmiş iptal (varsayılan)

Hedef iptal isteğini belirli *iptal noktalarında* denetler ve düzenli biçimde sonlanır.



`pthread_cleanup_push` ile kaydedilen **temizlik işleyicileri** çalışır: kilitler açılır, bellek geri verilir.

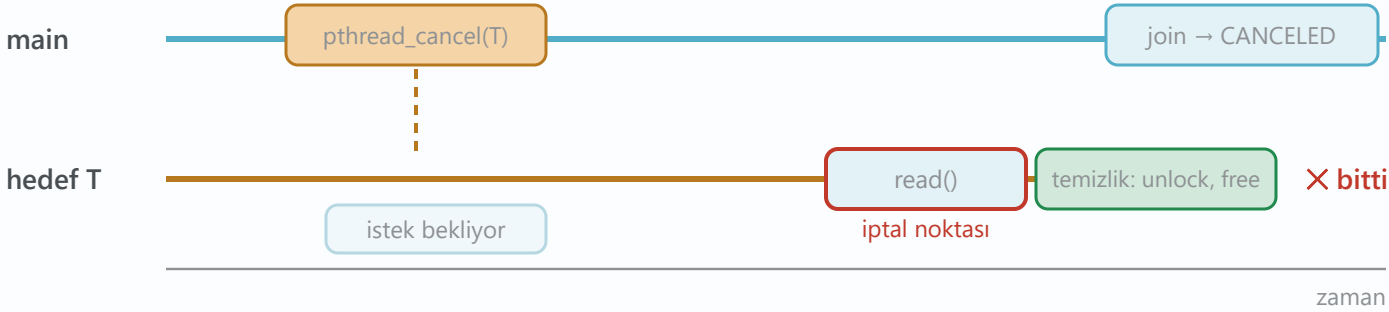
İş Parçacığı İptali

Eş zamansız iptal

Hedef *hemen* sonlandırılır. Kilit tutarken ya da veriyi yarım güncellemişken ölebilir: tehlikeli.

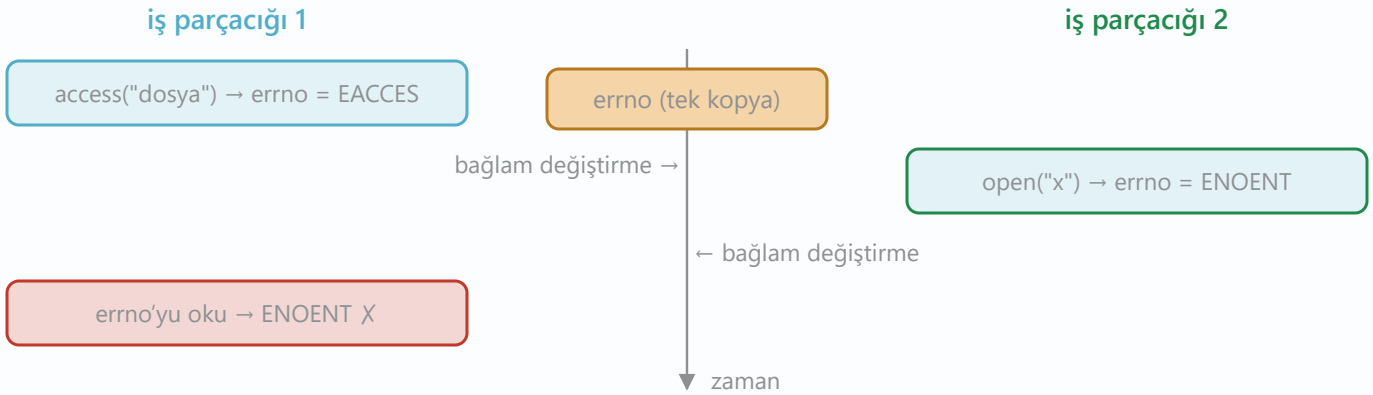
Ertelenmiş iptal (varsayılan)

Hedef iptal isteğini belirli *iptal noktalarında* denetler ve düzenli biçimde sonlanır.



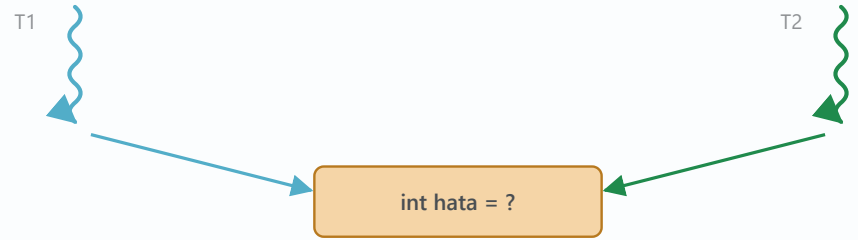
T sonlanır; `pthread_join` `PTHREAD_CANCELED` döndürür. Java'da karşılığı işbirlikçi `interrupt()` 'tır; `Thread.stop()` JDK 20'den beri çalışmaz.

Global Değişken Çakışması



- Tek iş parçacıklı programlar için tasarlanmış **global** değişkenler (UNIX `errno`) iş parçacıkları arasında çakışır.
- Çözüm: değişkeni yasaklamak, ya da her iş parçacığına **kendi kopyasını** vermek. Bugünkü C kütüphanelerinde `errno` iş parçacığına yereldir.

İş Parçacığına Yerel Depolama (TLS)



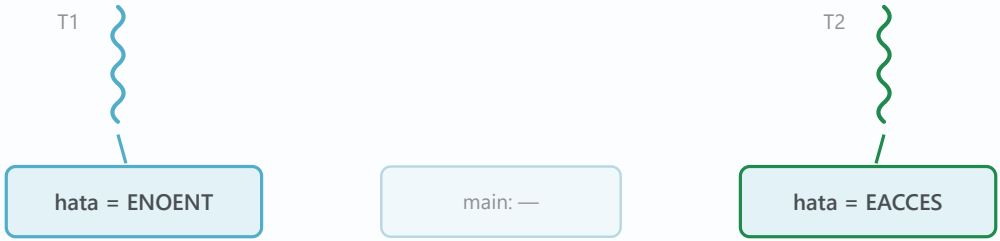
Düz global: `int hata;` iki iş parçacığı **aynı** değişkeni okur ve yazar.

Adım 1 / 3

- Yerel değişkenden farkı: fonksiyon çağrıları **boyunca** yaşar; global gibi ama iş parçacığına özel.
- C11 `_Thread_local`, C23/C++ `thread_local`, GCC `__thread`; Pthreads: `pthread_key_create`, `pthread_setspecific`.
- Örnek kullanım: `errno`, işlem kimliği, rastgele sayı üreteci.

```
static ThreadLocal<Integer> no =
  ThreadLocal.withInitial(() -> 0);
```

İş Parçacığına Yerel Depolama (TLS)



`_Thread_local int hata;` (C11) her iş parçacığına **ayrı kopya** verir; T1'in yazdığını T2 görmez.

Adım 2 / 3

- Yerel değişkenden farkı: fonksiyon çağrıları **boyunca** yaşar; global gibi ama iş parçacığına özel.
- C11 `_Thread_local`, C23/C++ `thread_local`, GCC `__thread`; Pthreads: `pthread_key_create`, `pthread_setspecific`.
- Örnek kullanım: `errno`, işlem kimliği, rastgele sayı üreteci.

```
static ThreadLocal<Integer> no =
  ThreadLocal.withInitial(() -> 0);
```

İş Parçacığına Yerel Depolama (TLS)



Java `ThreadLocal` çıktısı (JDK 21): `A: 1`, `B: 2`, `main: 0`; main başlangıç değerini görür.

Adım 3 / 3

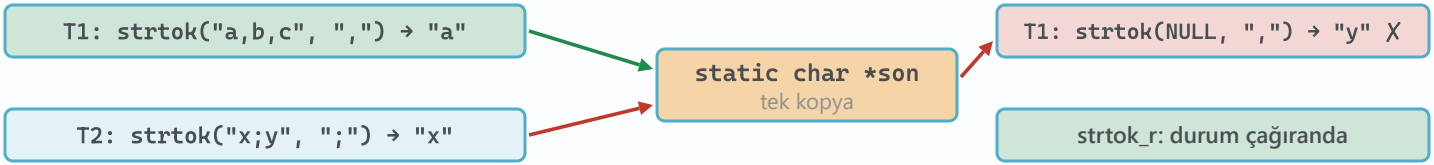
- Yerel değişkenden farkı: fonksiyon çağrıları **boyunca** yaşar; global gibi ama iş parçacığına özel.
- C11 `_Thread_local`, C23/C++ `thread_local`, GCC `__thread`; Pthreads: `pthread_key_create`, `pthread_setspecific`.
- Örnek kullanım: `errno`, işlem kimliği, rastgele sayı üreteci.

```
static ThreadLocal<Integer> no =
  ThreadLocal.withInitial(() -> 0);
```

Yeniden Girilir ve İş Parçacığı Güvenli Kod

- **Yeniden girilemez** (non-reentrant) kütüphane yordamları iç durumlarını statik değişkenlerde saklar.
- Örnek: bir iş parçacığı iletiyi ortak ara belleğe koyar, bağlam değişir, başka iş parçacığı aynı belleğe yazar → **ilk ileti kaybolur**.
- `malloc` serbest bellek listesini güncellerken listeyi geçici olarak tutarsız bırakır; araya giren iş parçacığı geçersiz işaretçi alabilir. Bugünkü `malloc`'lar iç kilit ve iş parçacığı başına arenalar kullanır.
- **İş parçacığı güvenli**: birden çok iş parçacığından aynı anda çağrılabilir (kilitle ya da durumsuz).
  Yeniden girilir: paylaşılan durum kullanmaz.

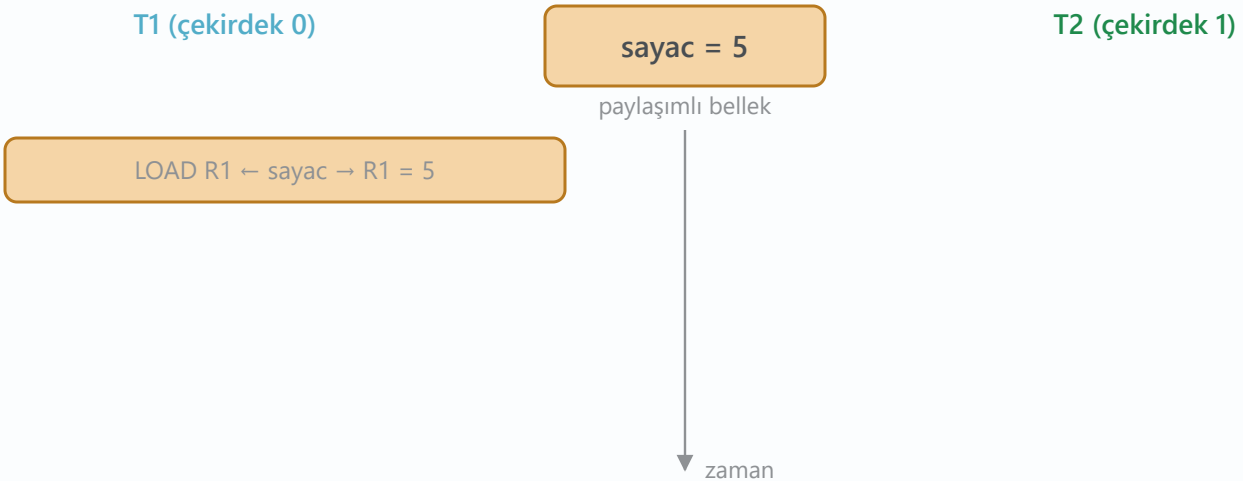
Güvensiz	Güvenli sürüm
`strtok`	`strtok_r`
`localtime`	`localtime_r`
`rand`	`rand_r` / yerel üreteç
`asctime`, `ctime`	`strftime`
Java `SimpleDateFormat`	`DateTimeFormatter`



Yarış Durumu: sayac++

```
LOAD  R, sayac
ADD   R, 1
STORE sayac, R
```

`sayac++` tek satırdır ama makinede **üç adımdan oluşur** (oku, artır, yaz). Başlangıçta `sayac = 5` ; iki iş parçacığı birer kez artırıyor. Beklenen sonuç: **7**.

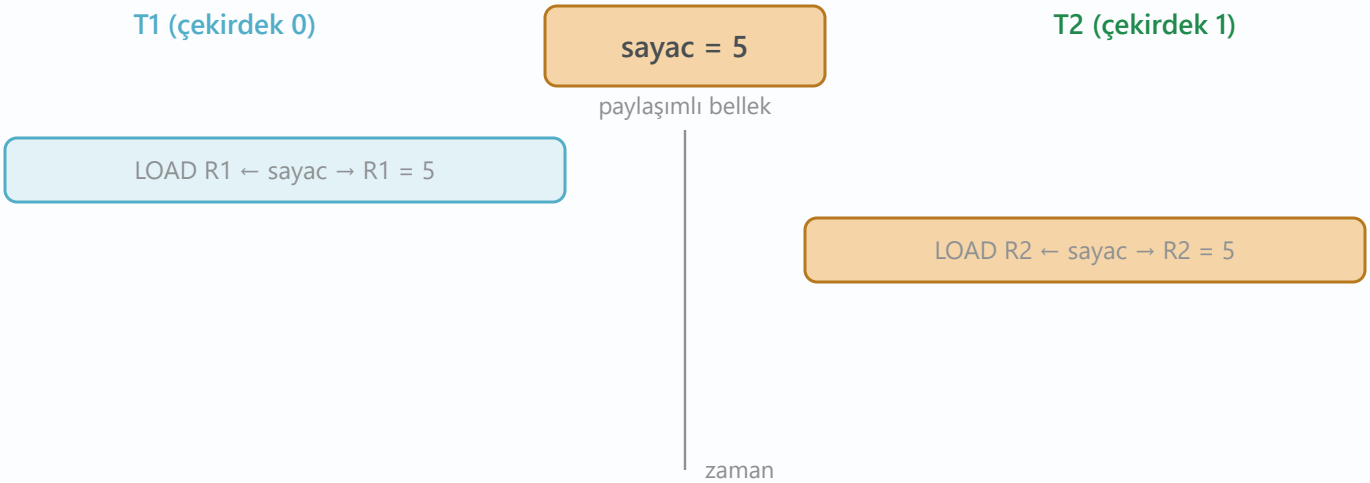


T1 değeri yazmacına okur: R1 = 5.

Yarış Durumu: sayac++

```
LOAD  R, sayac
ADD   R, 1
STORE sayac, R
```

`sayac++` tek satırdır ama makinede **üç adımdan oluşur** (oku, artır, yaz). Başlangıçta `sayac = 5` ; iki iş parçacığı birer kez artırıyor. Beklenen sonuç: **7**.

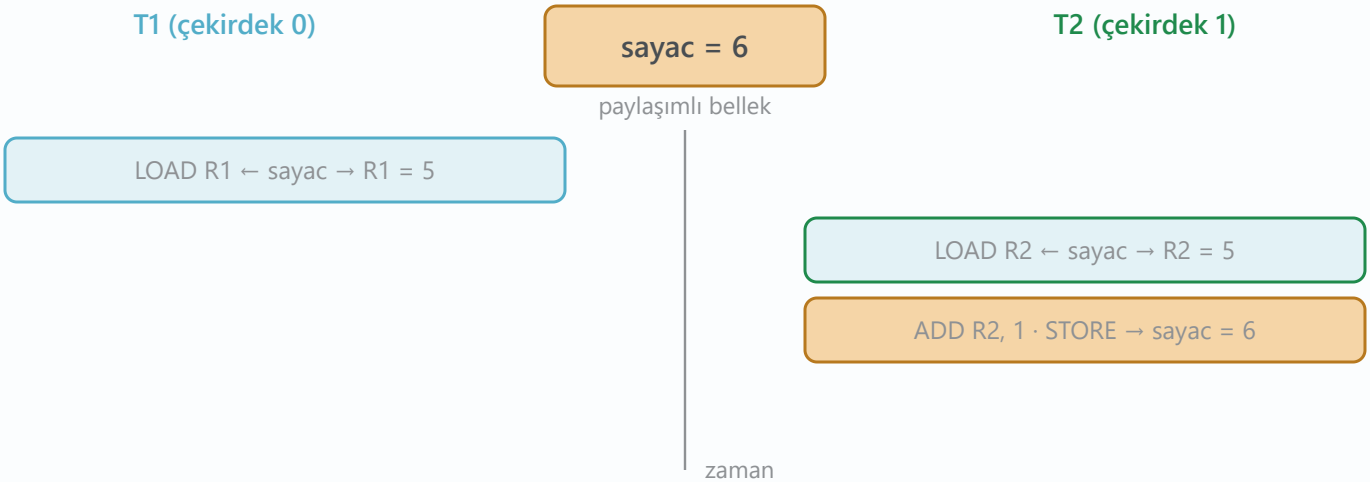


Araya T2 girer (ya da diğer çekirdekte aynı anda çalışır) ve o da **5** okur.

Yarış Durumu: sayac++

```
LOAD  R, sayac
ADD   R, 1
STORE sayac, R
```

`sayac++` tek satırdır ama makinede **üç adımdan oluşur** (oku, artır, yaz). Başlangıçta `sayac = 5`; iki iş parçacığı birer kez artırıyor. Beklenen sonuç: **7**.

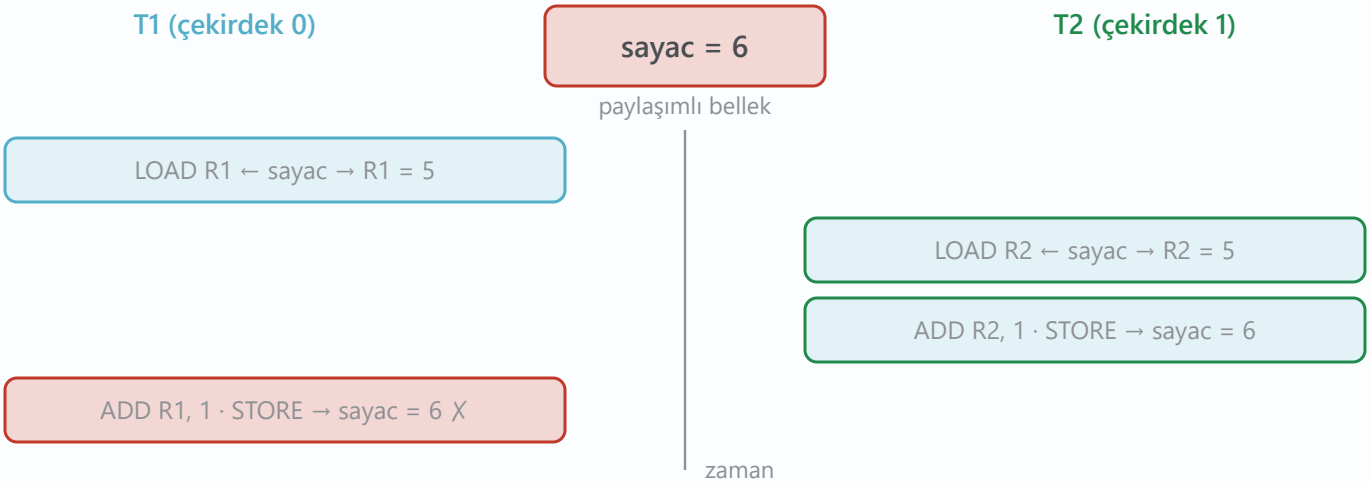


T2 artırıp yazar: sayac = 6.

Yarış Durumu: sayac++

```
LOAD  R, sayac
ADD   R, 1
STORE sayac, R
```

`sayac++` tek satırdır ama makinede **üç adımdan oluşur** (oku, artır, yaz). Başlangıçta `sayac = 5` ; iki iş parçacığı birer kez artırıyor. Beklenen sonuç: **7**.



T1 eski değerinden devam eder: 5 + 1 = 6 yazar. T2'nin artışı **kayboldu** (lost update).

Yarış Durumu: sayac++

```
LOAD  R, sayac
ADD   R, 1
STORE sayac, R
```

`sayac++` tek satırdır ama makinede **üç adımdan oluşur** (oku, artır, yaz). Başlangıçta `sayac = 5`; iki iş parçacığı birer kez artırıyor. Beklenen sonuç: **7**.

T1 (çekirdek 0)

T2 (çekirdek 1)

sayac = 7

paylaşımlı bellek

lock · LOAD 5 · ADD · STORE 6 · unlock

lock · LOAD 6 · ADD · STORE 7 · unlock

kritik bölge sırayla yürür: sonuç 7 ✓

C: pthread_mutex_lock / atomic_fetch_add · Java: synchronized / AtomicInteger

Çözüm: oku-artır-yaz **atomik** olmalı: kilit ya da atomik işlem.

Yarışı Gözlemlemek (Java)

```java
public class Yaris {
  static int sayac = 0;
  public static void main(String[] a) throws Exception {
    Runnable is = () -> {
      for (int i = 0; i < 1_000_000; i++) sayac++;
    };
    Thread t1 = new Thread(is), t2 = new Thread(is);
    t1.start(); t2.start();
    t1.join();  t2.join();
    System.out.println("sayac = " + sayac);
  }
}
```

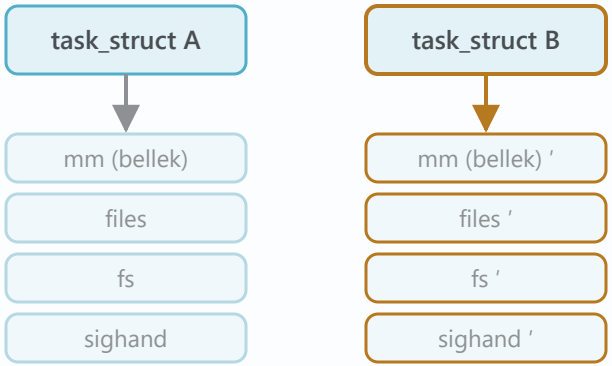
ÜÇ AYRI ÇALIŞTIRMA (JDK 21)

```
sayac = 1456465
sayac = 1544210
sayac = 1910165
```

- Beklenen 2.000.000; her çalıştırmada **farklı ve daha küçük** bir sonuç çıkar.
- Sonuç **belirlenimci değildir**: çizelgelemeye, çekirdek sayısına ve JIT'e bağlıdır. Bu yüzden çıktı adım adım izlenemez.
- Düzeltmeler: `synchronized` blok, `AtomicInteger.incrementAndGet()`, `LongAdder`.
- En iyisi: her iş parçacığı yerel değişkende saysın, sonda birleştirilsin (fork-join fikri).

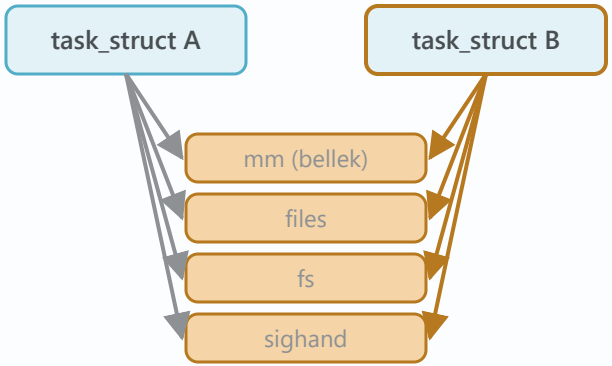
Linux'ta İş Parçacıkları: clone()

fork(): kopya



- Linux "süreç" ve "iş parçacığı" ayırmaz: hepsi **görev** (task, `task_struct`).
- Ne paylaşılacağını `clone()` bayrakları belirler; hiç bayrak yoksa `fork` gibidir.

clone(CLONE_VM | CLONE_FILES | ...): paylaşım



- `pthread_create` ≈ `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD | …)`.
- `CLONE_THREAD`: aynı iş parçacığı grubu; `getpid()` herkes için aynı (TGID), `gettid()` farklı.

ÖZET

Avantajlar ve Dezavantajlar

Avantajlar

+ **Duyarlılık:** biri bloke olunca diğeri kullanıcı G/Ç'sini işler.
+ **Kaynak paylaşımı:** bellek, açık dosyalar, soketler.
+ **Hız:** oluşturma ve geçiş süreçlerden ucuz.
+ **Donanım paralelliği:** tek süreç çok çekirdeği kullanır.
+ Bloke eden çağrılarla bile paralellik (web sunucusu), kolay programlama modeli.

Dezavantajlar

– **Senkronizasyon:** paylaşılan veriye erişim denetlenmeli; yarış ve kilitlenme riski.
– **Bağımlılık:** aynı süreçteki iş parçacıkları bağımsız değildir; biri çökerse süreç çöker.
– **Bellek koruması yok:** bir iş parçacığı başkasının yığınının üzerine yazabilir.
– Hata ayıklama ve test zor; hatalar zamanlamaya bağlı.

Yalıtım gerekiyorsa (tarayıcı sekmeleri, güvenlik sınırları) **süreç**, sıkı veri paylaşımı ve düşük maliyet gerekiyorsa **iş parçacığı** seçilir. Chrome her ikisini birlikte kullanır.

ÖZET

Kontrol Soruları (1/2)

1. Yazmaç değerleri, heap, global değişkenler ve yığından hangileri bir sürecin iş parçacıkları arasında paylaşılır?
2. Tek çekirdekli bir sistemde eş zamanlılık var mıdır? Paralellik var mıdır?
3. Programın %80'i paralelleştirilebiliyor. 8 çekirdekte hızlanma ve teorik üst sınır nedir?
4. Çoktan bire modelde bir iş parçacığı `read()` ile bloke olursa ne olur? Neden çok çekirdekten yararlanamaz?

▶ **Cevaplar**

ÖZET

Kontrol Soruları (2/2)

5. İki iş parçacığı başlangıçta 0 olan `sayac`'ı birer kez artırıyor. Olası son değerler nelerdir?
6. İş parçacığı havuzunun yeni iş parçacığı oluşturmaya göre iki avantajını yazın.
7. Ertelenmiş ve eş zamansız iptal arasındaki fark nedir? Neden ertelenmiş tercih edilir?
8. POSIX'te üç iş parçacıklı bir süreçte `fork()` sonrası çocukta kaç iş parçacığı vardır?

▶ **Cevaplar**

Kaynaklar

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 4: Threads & Concurrency.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 2.2: Threads.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Concurrency: An Introduction", "Thread API".
- POSIX.1-2017 (IEEE Std 1003.1), `pthreads(7)` ve `clone(2)` Linux kılavuz sayfaları.
- JEP 444: Virtual Threads (Java 21); OpenMP 5.2 belirtimi.

Son

Bölüm 4: İş Parçacıkları

Sonraki bölüm: CPU Çizelgeleme · Terim sözlüğü