Bölüm 5: CPU Çizelgeleme

İşletim Sistemleri

CPU–G/Ç döngüsü | Dağıtıcı ve bağlam değiştirme | Çizelgeleme ölçütleri

FCFS ve konvoy etkisi | SJF, SRTF, üstel ortalama | Öncelik ve yaşlandırma

Round Robin ve zaman dilimi | MLQ ve MLFQ | Piyango, stride, adil paylaşım

Çok işlemcili çizelgeleme | Gerçek zamanlı: RM ve EDF | Linux EEVDF, Windows

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_05_CPU_Cizelgeleme.html

GİRİŞ

Öğrenme Çıktıları

Kavramlar

CPU–G/Ç patlama döngüsünü, çizelgeleme karar noktalarını, kesmeli ve kesmesiz çizelgelemeyi ve dağıtıcının görevini açıklamak.

Ölçütler

Bekleme, dönüş ve tepki sürelerini Gantt şemasından hesaplamak; verim, CPU kullanımı ve adalet hedeflerini tartmak.

Algoritmalar

FCFS, SJF, SRTF, öncelik, RR, MLQ ve MLFQ algoritmalarını aynı süreç kümesi üzerinde adım adım uygulamak ve karşılaştırmak.

Tahmin ve adalet

Sonraki CPU patlamasını üstel ortalamayla tahmin etmek; yaşlandırma, piyango, stride ve adil paylaşımla açlığı önlemek.

Çok işlemci ve gerçek zaman

Yük dengeleme ve işlemci yakınlığını açıklamak; RM ve EDF ile çizelgelenebilirliği kullanım sınırıyla sınamak.

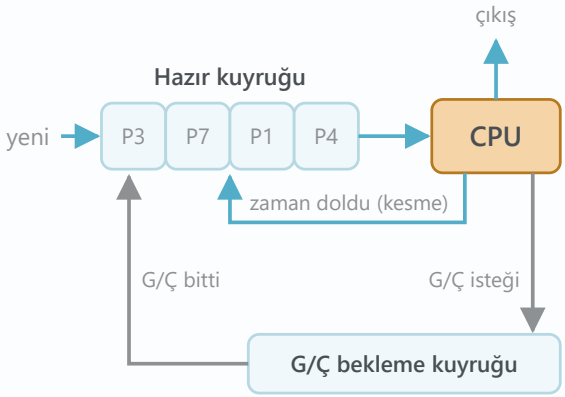
Gerçek sistemler

Linux CFS ve onun yerini alan EEVDF ile Windows öncelik sınıflarını ve dinamik öncelik artırmalarını yorumlamak.

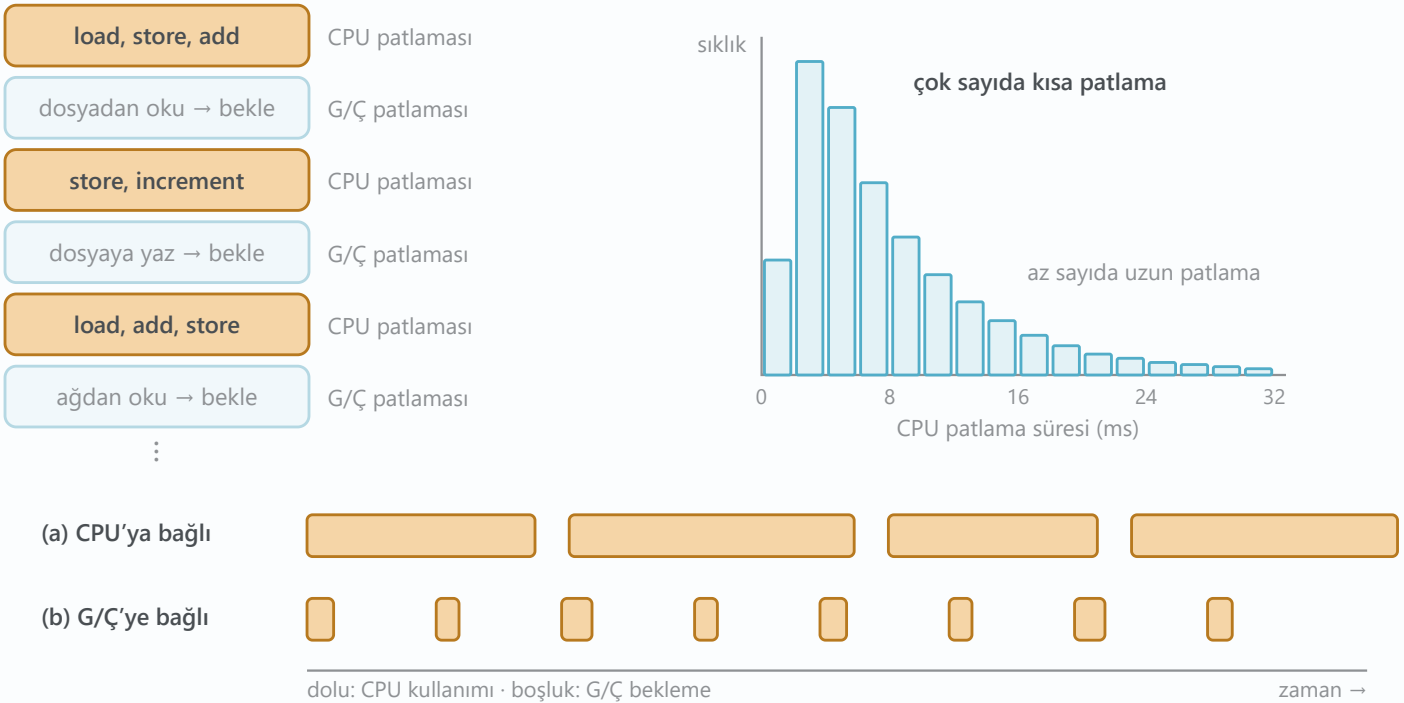
TEMEL KAVRAMLAR

Çizelgeleme Nedir?

- Çok programlı sistemde birçok süreç bellekte hazır bekler; CPU boşaldığında hangisinin çalışacağına **kısa vadeli çizelgeleyici** (CPU scheduler) karar verir.
- Amaç: CPU'yu boş bırakmamak ve seçilen hedefleri (tepki, verim, adalet, son tarih) iyileştirmek.
- **Hazır kuyruğu:** çalışmaya hazır süreçler. **Bekleme kuyrukları:** G/Ç ya da bir olayı bekleyenler. **Çalışan:** her çekirdekte en fazla bir tane.
- Hazır kuyruğu FIFO olmak zorunda değildir: öncelikli kuyruk, ağaç ya da sırasız liste olabilir.
- Modern çekirdekler aslında **çekirdek iş parçacıklarını** çizelgeler; "süreç çizelgeleme" deyimi bu anlamda kullanılır.



CPU–G/Ç Patlama Döngüsü

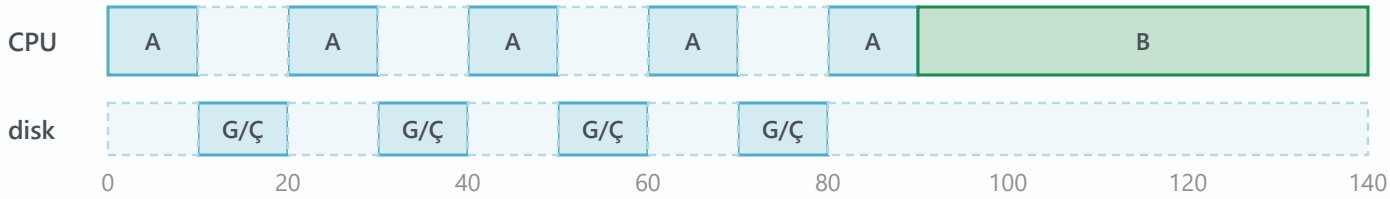


Belirleyici olan G/Ç beklemesinin uzunluğu değil, **CPU patlamasının kısalığıdır**. Hazır olan G/Ç'ye bağlı süreç hemen çalıştırılırsa diski ve ağı meşgul tutar.

G/Ç ile Örtüşme: Patlamaları Ayrı İş Saymak

A ve B 50'şer ms CPU ister; A 10 ms çalışıp 10 ms disk bekler, B G/Ç yapmaz.

Örtüşmesiz: A'nın 50 ms'lik işi tek parça sayılır, B ancak A bitince başlar



A biter 90, B biter 140 → CPU 140 ms'nin 100 ms'sinde dolu: %71

A tek 50 ms'lik iş sayılırsa A diskteyken CPU **boşta**: B ancak A bitince başlar.

Adım 1 / 2

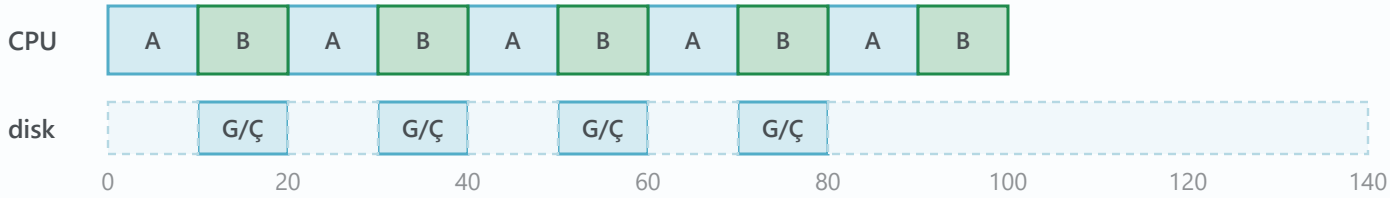
Çizelge	A bitiş	B bitiş	Ort. dönüş	CPU kullanımı
Örtüşmesiz	90	140	(90 + 140) / 2 = 115 ms	100 / 140 ≈ %71
Örtüşmeli	90	100	(90 + 100) / 2 = 95 ms	100 / 100 = %100

Patlama sürelerini bilmeyen gerçek sistemde bu davranışı MLFQ öğrenir.

G/Ç ile Örtüşme: Patlamaları Ayrı İş Saymak

A ve B 50'şer ms CPU ister; A 10 ms çalışıp 10 ms disk bekler, B G/Ç yapmaz.

Örtüşmeli: A'nın her 10 ms'lik patlaması ayrı, kısa bir iş; A diskteyken B çalışır



A biter 90, B biter 100 → CPU 100 ms boyunca hep dolu: %100

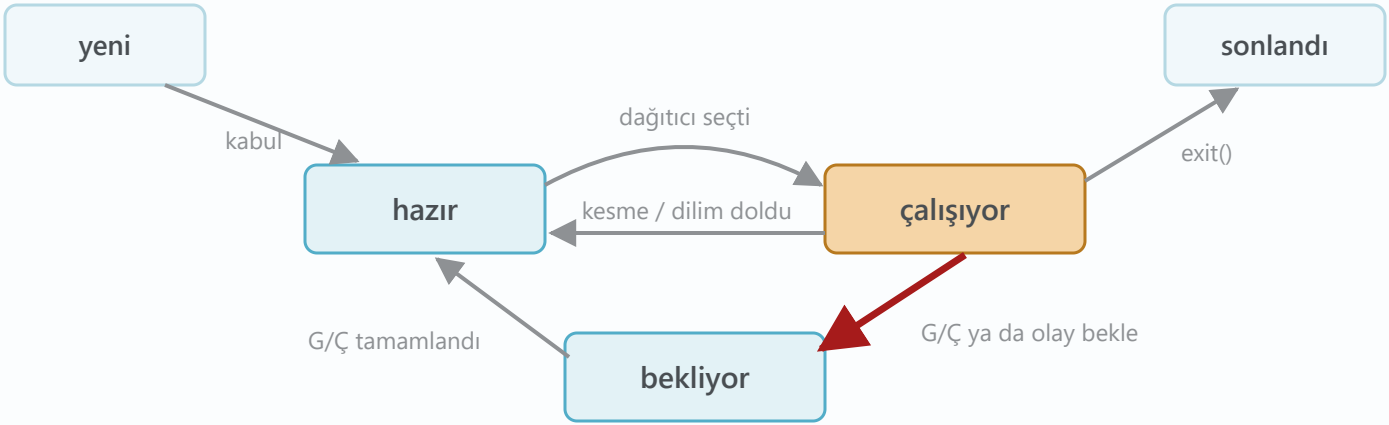
Her patlama ayrı iş: A diskteyken B çalışır; dönen kısa A, B'yi keser (STCF).

Adım 2 / 2

Çizelge	A bitiş	B bitiş	Ort. dönüş	CPU kullanımı
Örtüşmesiz	90	140	(90 + 140) / 2 = 115 ms	100 / 140 ≈ %71
Örtüşmeli	90	100	(90 + 100) / 2 = 95 ms	100 / 100 = %100

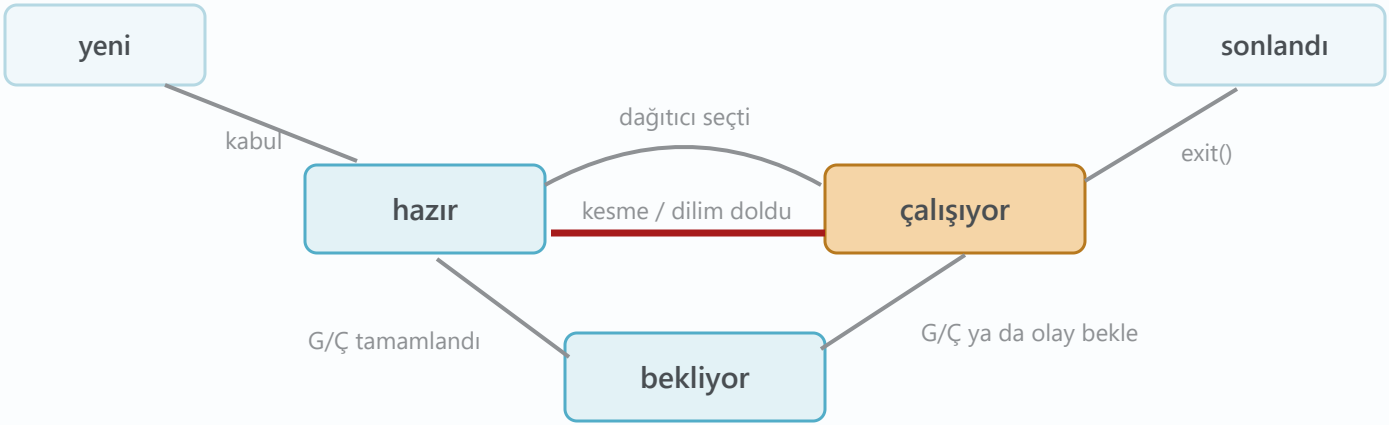
Patlama sürelerini bilmeyen gerçek sistemde bu davranışı MLFQ öğrenir.

Çizelgeleme Kararları Ne Zaman Alınır?



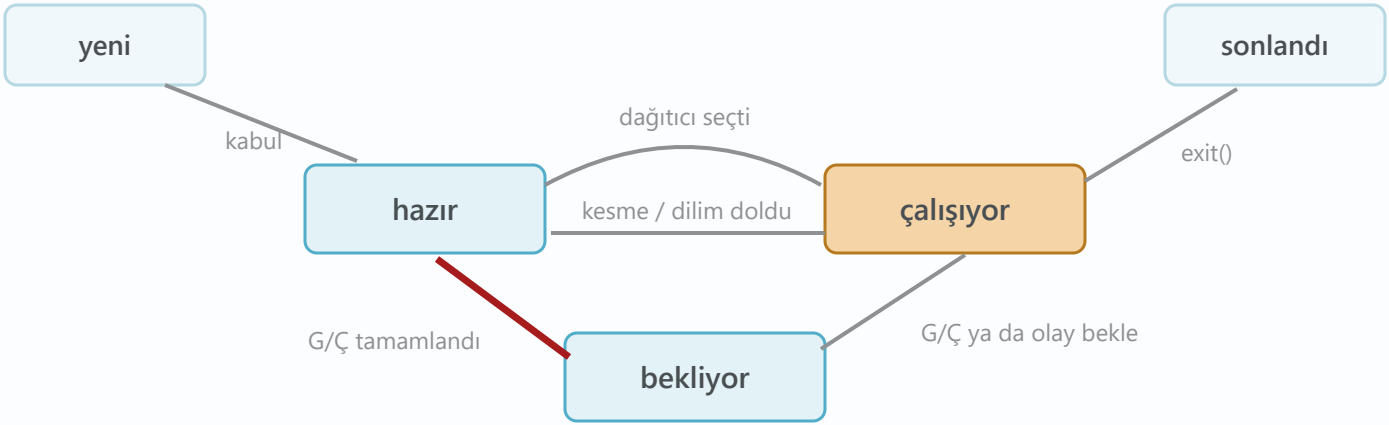
1. Çalışıyor → bekliyor: süreç G/Ç ister ya da `wait()` çağırır. CPU boşalır, seçim **zorunludur**.

Çizelgeleme Kararları Ne Zaman Alınır?



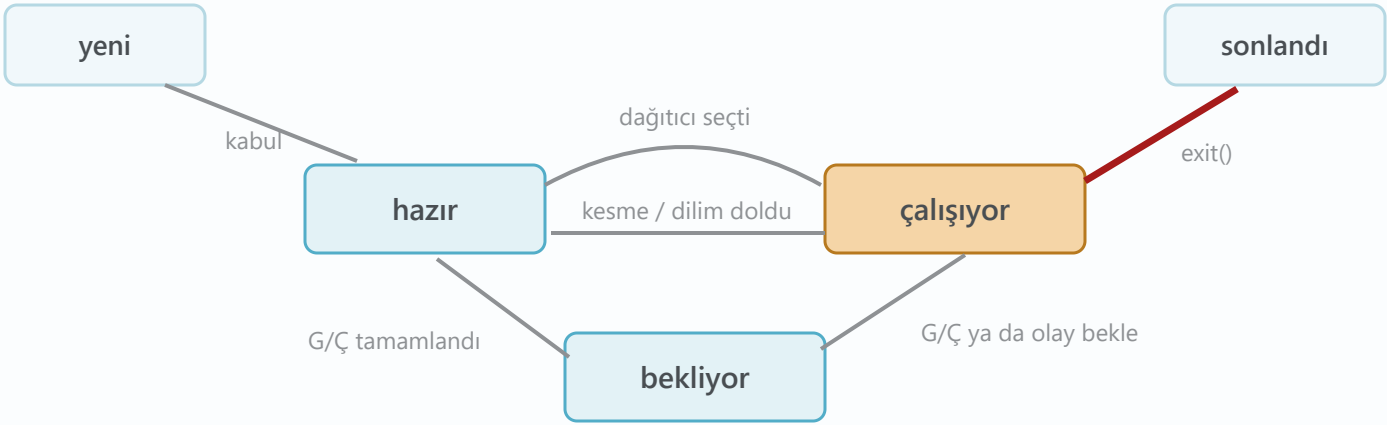
2. Çalışıyor → hazır: saat kesmesi geldi, zaman dilimi doldu. Çalışanı kesmek **isteğe bağlıdır**.

Çizelgeleme Kararları Ne Zaman Alınır?



3. Bekliyor → hazır: G/Ç tamamlandı; uyanan süreç çalışandan daha önemliyse onu kesebilir.

Çizelgeleme Kararları Ne Zaman Alınır?



4. Sonlanma: süreç `exit()` ile biter; CPU boşalır, seçim **zorunludur**.

Çizelgeleme Kararları Ne Zaman Alınır?



Yalnızca 1 ve 4'te karar veren çizelgeleyici **kesmesiz** (işbirlikçi); 2 ve 3'te de karar veren **kesmeli**dir.

Kesmeli ve Kesmesiz Çizelgeleme

Kesmesiz (non-preemptive)

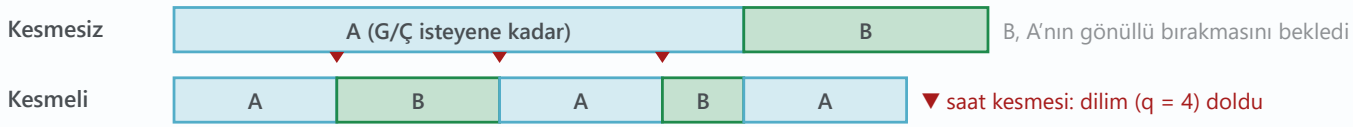
Seçilen süreç bloke olana, sonlanana ya da CPU'yu gönüllü bırakana (`yield`) kadar çalışır. Saat kesmesi gerekmez; geçiş azdır.

Örnek: Windows 3.x ve klasik Mac OS (9 ve öncesi) işbirlikçi çoklu görevdi; tek bir hatalı program tüm sistemi kilitleyebiliyordu.

Kesmeli (preemptive)

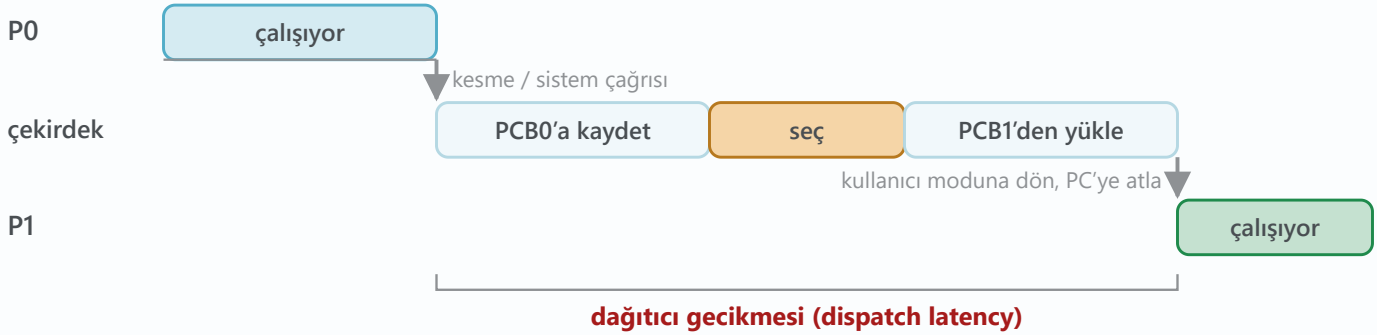
Süreç, zaman dilimi dolunca ya da daha önemli bir süreç hazır olunca askıya alınır. Saat kesmesi şarttır; tüm genel amaçlı modern işletim sistemleri böyledir.

Bedeli: paylaşılan veride **yarış durumu** riski ve daha çok bağlam değiştirme.



- **Çekirdek içi kesme:** Linux 2.6'dan beri çekirdek kodu da kesilebilir; kesme yalnızca kilit tutulmadığında yapılır (`preempt_count = 0`).
- **PREEMPT_RT** yamaları 20 yıllık geliştirmeden sonra Linux 6.12 (2024) ile ana çekirdeğe girdi: kesme işleyicileri iş parçacığı olarak çalışır, gecikme mikrosaniyeler düzeyine iner.
- Toplu işlerde kesmesiz ya da uzun dilimli kesmeli algoritmalar yeterlidir; etkileşimli ve sunucu sistemlerinde kesme zorunludur.

Görev Dağıtıcı ve Bağlam Değiştirme



- **Dağıtıcı** (dispatcher), çizelgeleyicinin seçtiği sürece CPU'yu verir: bağlam değiştirir, kullanıcı moduna geçer, programdaki doğru adrese atlar.
- Her geçişte çalıştığı için **hızlı** olmalıdır; doğrudan maliyet bugün birkaç mikrosaniyedir.
- Linux'ta ölçüm: `vmstat 1` çıktısındaki `cs` sütunu; `/proc/<pid>/status` içindeki `voluntary_ctxt_switches` ve `nonvoluntary_ctxt_switches`.

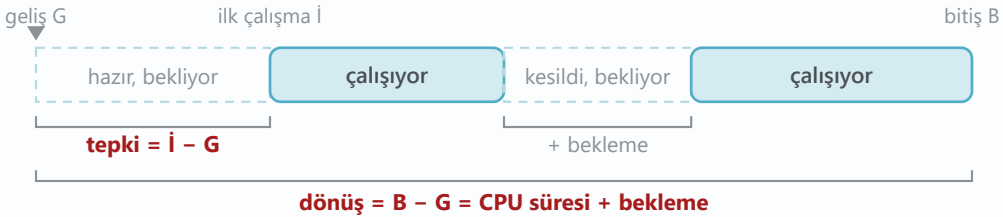
Bileşen	Süreçler arası geçişte
Yazmaçlar, PC, SP	kaydedilir / yüklenir
Bellek haritası	sayfa tablosu tabanı değişir (x86: CR3)
TLB	PCID/ASID varsa boşaltılmaz
Önbellekler	boşaltılmaz ama **soğuk** kalır

Ortama Göre Çizelgeleme Hedefleri

Ortam	Öncelikli hedefler	Uygun yaklaşım
Tüm sistemler	**Adalet** (benzer süreçlere benzer pay), **politika** uygulama (ör. güvenlik süreçleri önce), **denge** (CPU ve G/Ç aygıtları birlikte meşgul)	—
Toplu (batch)	**Verim** (iş/saat), kısa **dönüş süresi**, yüksek **CPU kullanımı**	Kesmesiz ya da uzun dilimli kesmeli
Etkileşimli	Kısa **tepki süresi**, **orantılılık** (kullanıcının beklentisine uygun gecikme)	Kesmeli; kısa zaman dilimi
Gerçek zamanlı	**Son tarihlere** uymak (veri kaybı yok), **öngörülebilirlik** (ses/görüntü kalitesi düşmez)	Öncelik / son tarih tabanlı; görevler kısa çalışıp bloke olur

Toplu işler bugün de yaygındır: bordro ve bankacılık gece işleri, bilimsel hesap ve yapay zekâ eğitimi (HPC kuyruk yöneticileri, ör. Slurm), CI derlemeleri. Hedefler çoğu zaman çelişir: tepki süresini kısaltan sık geçişler verimi düşürür.

Çizelgeleme Ölçütleri



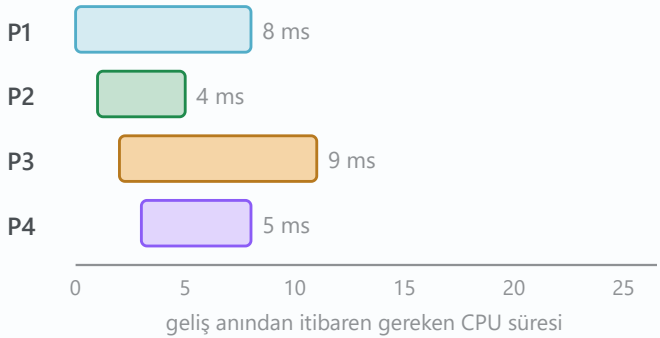
Ölçüt	Tanım	Hedef
CPU kullanımı	CPU'nun meşgul olduğu zaman oranı (gerçek sistemlerde %40–%90)	en büyük
Verim (throughput)	Birim zamanda tamamlanan süreç sayısı	en büyük
Dönüş süresi (turnaround)	Sisteme girişten tamamlanmaya kadar geçen süre: B − G	en küçük
Bekleme süresi (waiting)	Hazır kuyruğunda geçen toplam süre (G/Ç yoksa: dönüş − CPU süresi)	en küçük
Tepki süresi (response)	Girişten ilk kez CPU'ya çıkmaya (ilk tepkiye) kadar geçen süre: İ − G	en küçük

Çizelgeleyici yalnızca hazır kuyruğundaki bekleme süresini etkiler; CPU ve G/Ç süresi programa aittir. Etkileşimli sistemde ortalamadan çok tepki süresinin **varyansını** küçültmek önemlidir. Ağırlıklı dönüş = dönüş / CPU süresi.

Ortak Süreç Kümesi

Süreç	Geliş	CPU süresi	Öncelik
P1	0	8	2
P2	1	4	3
P3	2	9	1
P4	3	5	4

- Küçük sayı = **yüksek öncelik**; eşitlikte önce gelen kazanır.
- RR'de aynı anda gelen yeni süreç, dilimi biten süreçten **önce** kuyruğa girer.
- Süreler ms; bağlam değiştirme süresi 0 kabul edilir; G/Ç yok.



FCFS, SJF, SRTF, öncelik ve RR bu dört süreçle adım adım çalıştırılır; böylece sonuçlar doğrudan karşılaştırılabilir. Farklı süreç kümeleri: CPU Çizelgeleme Simülatörü; Gantt şeması ve ölçütler anında hesaplanır.

FCFS Adım Adım

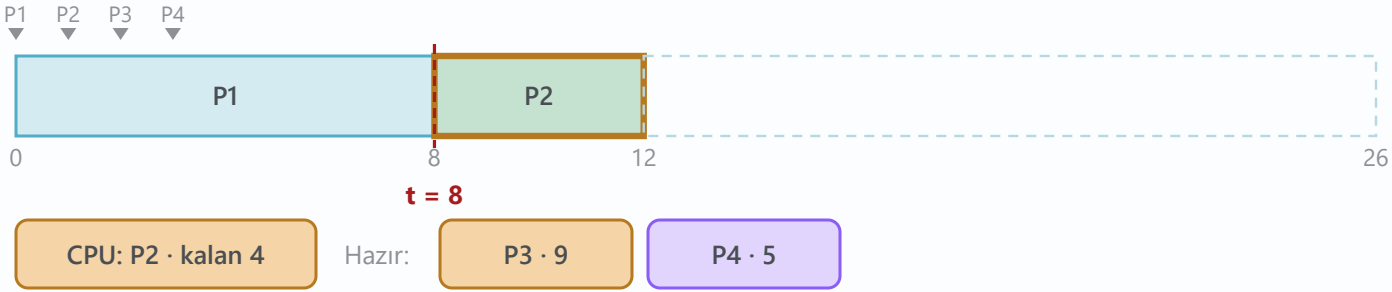
İlk gelen ilk hizmet alır (first-come, first-served): hazır kuyruğu FIFO; CPU'yu alan süreç bitene ya da bloke olana kadar bırakmaz. Gerçeklemesi en basit algoritmadır.



t = 0: kuyrukta yalnızca **P1** var; CPU'yu alır ve 8 ms boyunca bırakmaz (kesmesiz).

FCFS Adım Adım

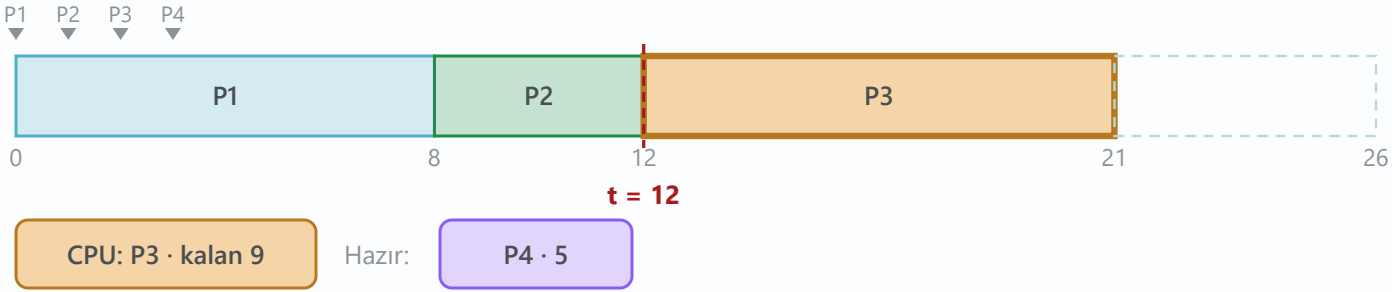
İlk gelen ilk hizmet alır (first-come, first-served): hazır kuyruğu FIFO; CPU'yu alan süreç bitene ya da bloke olana kadar bırakmaz. Gerçeklemesi en basit algoritmadır.



t = 8: P1 bitti. Kuyruk geliş sırasıyla ilerler → baştaki **P2** çalışır (7 ms bekledi).

FCFS Adım Adım

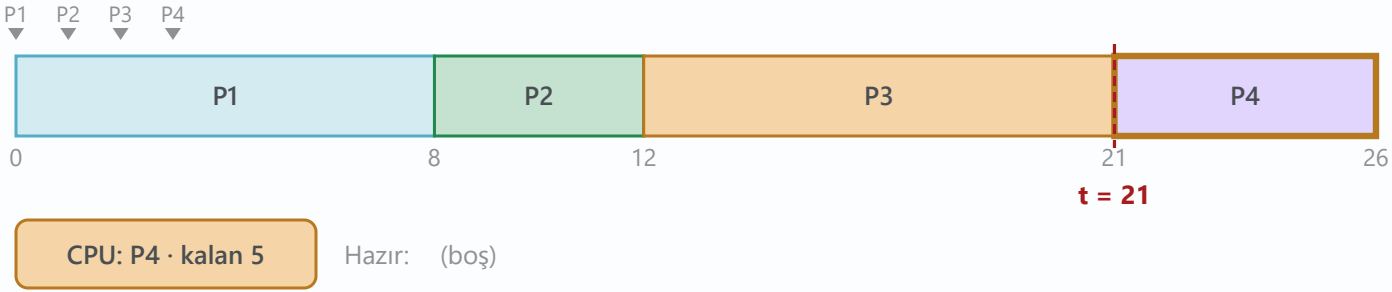
İlk gelen ilk hizmet alır (first-come, first-served): hazır kuyruğu FIFO; CPU'yu alan süreç bitene ya da bloke olana kadar bırakmaz. Gerçeklemesi en basit algoritmadır.



t = 12: P2 bitti. Kuyruk geliş sırasıyla ilerler → baştaki **P3** çalışır (10 ms bekledi).

FCFS Adım Adım

İlk gelen ilk hizmet alır (first-come, first-served): hazır kuyruğu FIFO; CPU'yu alan süreç bitene ya da bloke olana kadar bırakmaz. Gerçeklemesi en basit algoritmadır.



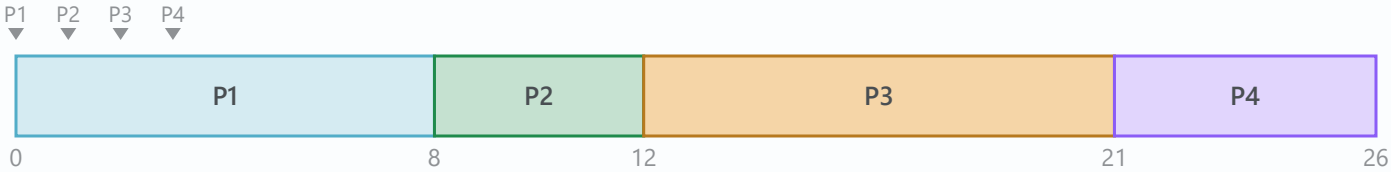
CPU: P4 · kalan 5

Hazır: (boş)

t = 21: P3 bitti. Kuyruk geliş sırasıyla ilerler → baştaki **P4** çalışır (18 ms bekledi).

FCFS Adım Adım

İlk gelen ilk hizmet alır (first-come, first-served): hazır kuyruğu FIFO; CPU'yu alan süreç bitene ya da bloke olana kadar bırakmaz. Gerçeklemesi en basit algoritmadır.

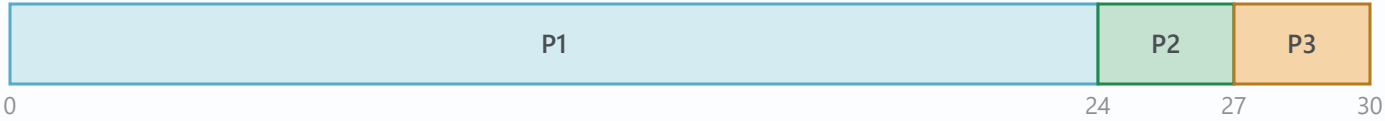


Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	0	8	8	0	0
P2	1	4	8	12	11	7	7
P3	2	9	12	21	19	10	10
P4	3	5	21	26	23	18	18
Ort.					**15,25**	**8,75**	**8,75**

Ortalama bekleme **8,75**, dönüş **15,25**, tepki **8,75** ms · 3 bağlam değiştirme.

FCFS: Konvoy Etkisi

Sıra: P1 → P2 → P3 (hepsi t = 0'da hazır)



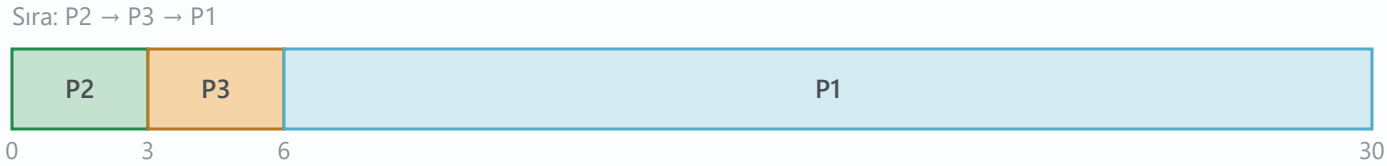
Bekleme: P1 = 0, P2 = 24, P3 = 27 → ortalama (0 + 24 + 27) / 3 = 17 ms

Uzun iş önde: kısa P2 ve P3, P1'in arkasında bekler. Ortalama bekleme **17 ms**.

Adım 1 / 3

CPU'yu sonlanana kadar tutan **CPU'ya bağlı** süreçtir; G/Ç'ye bağlı süreçler kısa patlamalardan sonra CPU'yu bırakır. FCFS kesmesizdir; açlık yoktur ama etkileşimli sistemler için uygun değildir.

FCFS: Konvoy Etkisi



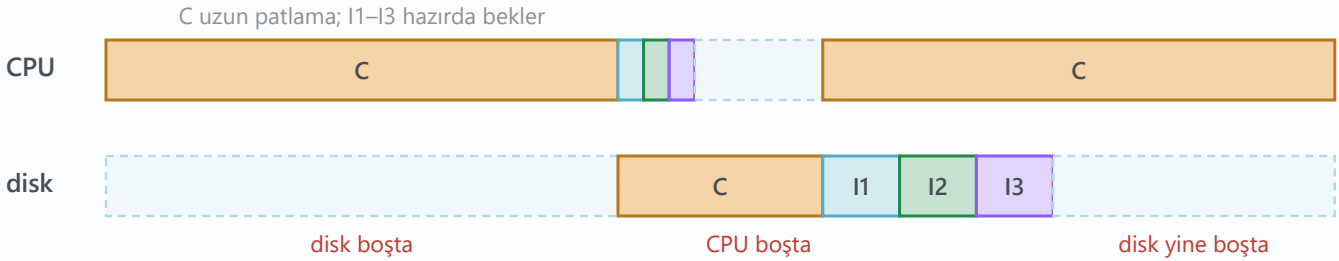
Bekleme: P1 = 6, P2 = 0, P3 = 3 → ortalama (6 + 0 + 3) / 3 = 3 ms

Aynı işler, farklı geliş sırası: ortalama **3 ms**. FCFS'de sonuç geliş sırasına aşırı duyarlıdır.

Adım 2 / 3

CPU'yu sonlanana kadar tutan **CPU'ya bağlı** süreçtir; G/Ç'ye bağlı süreçler kısa patlamalardan sonra CPU'yu bırakır. FCFS kesmesizdir; açlık yoktur ama etkileşimli sistemler için uygun değildir.

FCFS: Konvoy Etkisi



Konvoy etkisi: CPU'ya bağlı C CPU'yu bırakmaz; G/Ç'ye bağlı süreçler arkasında birikir ve aygıtlar sırayla boşta kalır.

Adım 3 / 3

CPU'yu sonlanana kadar tutan **CPU'ya bağlı** süreçtir; G/Ç'ye bağlı süreçler kısa patlamalardan sonra CPU'yu bırakır. FCFS kesmesizdir; açlık yoktur ama etkileşimli sistemler için uygun değildir.

FCFS Bekleme Süresini Hesaplamak

```
1  /* p[] geliş sırasına göre sıralı */
2  void fcfs(Surec p[], int n) {
▶    int t = 0, top = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].gelis) t = p[i].gelis;
6      int bekleme = t - p[i].gelis;
7      t += p[i].sure;          /* bitiş */
8      top += bekleme;
9      printf("%s: %d\n", p[i].ad, bekleme);
10   }
11   printf("ort = %.2f\n", (double) top / n);
12 }
```



Saat `t` ve toplam bekleme `top` sıfırlanır. Süreçler geliş sırasındadır.

Adım 1 / 10

DEĞİŞKENLER

`i = –` `t = 0` `top = 0`

ÇIKTI

FCFS Bekleme Süresini Hesaplamak

```
1  /* p[] geliş sırasına göre sıralı */
2  void fcfs(Surec p[], int n) {
3    int t = 0, top = 0;
4    for (int i = 0; i < n; i++) {
▶      if (t < p[i].gelis) t = p[i].gelis;
▶      int bekleme = t - p[i].gelis;
7      t += p[i].sure;          /* bitiş */
8      top += bekleme;
9      printf("%s: %d\n", p[i].ad, bekleme);
10   }
11   printf("ort = %.2f\n", (double) top / n);
12 }
```



P1 0'da gelmişti; CPU 0'da boşaldı → bekleme = 0 − 0 = **0**.

Adım 2 / 10

DEĞİŞKENLER

`i = 0` `t = 0` `gelis = 0` `bekleme = 0` `top = 0`

ÇIKTI

FCFS Bekleme Süresini Hesaplamak

```
1  /* p[] geliş sırasına göre sıralı */
2  void fcfs(Surec p[], int n) {
3    int t = 0, top = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].gelis) t = p[i].gelis;
6      int bekleme = t - p[i].gelis;
▶      t += p[i].sure;          /* bitiş */
▶      top += bekleme;
▶      printf("%s: %d\n", p[i].ad, bekleme);
10   }
11   printf("ort = %.2f\n", (double) top / n);
12 }
```



P1 8 ms çalışır, saat **8** olur; toplam bekleme 0.

Adım 3 / 10

DEĞİŞKENLER

`i = 0` `t = 8` `gelis = 0` `bekleme = 0` `top = 0`

ÇIKTI

```
P1: 0
```

FCFS Bekleme Süresini Hesaplamak

```
1  /* p[] geliş sırasına göre sıralı */
2  void fcfs(Surec p[], int n) {
3    int t = 0, top = 0;
4    for (int i = 0; i < n; i++) {
▶      if (t < p[i].gelis) t = p[i].gelis;
▶      int bekleme = t - p[i].gelis;
7      t += p[i].sure;          /* bitiş */
8      top += bekleme;
9      printf("%s: %d\n", p[i].ad, bekleme);
10   }
11   printf("ort = %.2f\n", (double) top / n);
12 }
```



P2 1'de gelmişti; CPU 8'de boşaldı → bekleme = 8 − 1 = **7**.

Adım 4 / 10

DEĞİŞKENLER

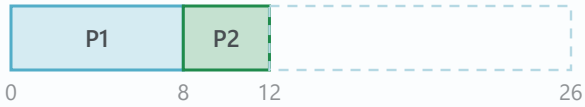
`i = 1` `t = 8` `gelis = 1` `bekleme = 7` `top = 0`

ÇIKTI

```
P1: 0
```

FCFS Bekleme Süresini Hesaplamak

```
1  /* p[] geliş sırasına göre sıralı */
2  void fcfs(Surec p[], int n) {
3    int t = 0, top = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].gelis) t = p[i].gelis;
6      int bekleme = t - p[i].gelis;
▶      t += p[i].sure;          /* bitiş */
▶      top += bekleme;
▶      printf("%s: %d\n", p[i].ad, bekleme);
10   }
11   printf("ort = %.2f\n", (double) top / n);
12 }
```



P2 4 ms çalışır, saat **12** olur; toplam bekleme 7.

Adım 5 / 10

DEĞİŞKENLER

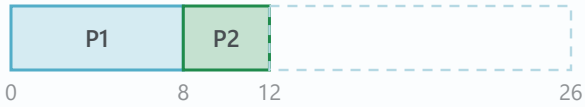
`i = 1` `t = 12` `gelis = 1` `bekleme = 7` `top = 7`

ÇIKTI

```
P1: 0
P2: 7
```

FCFS Bekleme Süresini Hesaplamak

```
1  /* p[] geliş sırasına göre sıralı */
2  void fcfs(Surec p[], int n) {
3    int t = 0, top = 0;
4    for (int i = 0; i < n; i++) {
▶      if (t < p[i].gelis) t = p[i].gelis;
▶      int bekleme = t - p[i].gelis;
7      t += p[i].sure;          /* bitiş */
8      top += bekleme;
9      printf("%s: %d\n", p[i].ad, bekleme);
10   }
11   printf("ort = %.2f\n", (double) top / n);
12 }
```



P3 2'de gelmişti; CPU 12'de boşaldı → bekleme = 12 − 2 = **10**.

Adım 6 / 10

DEĞİŞKENLER

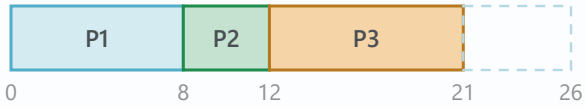
`i = 2` `t = 12` `gelis = 2` `bekleme = 10` `top = 7`

ÇIKTI

```
P1: 0
P2: 7
```

FCFS Bekleme Süresini Hesaplamak

```
1  /* p[] geliş sırasına göre sıralı */
2  void fcfs(Surec p[], int n) {
3    int t = 0, top = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].gelis) t = p[i].gelis;
6      int bekleme = t - p[i].gelis;
▶      t += p[i].sure;          /* bitiş */
▶      top += bekleme;
▶      printf("%s: %d\n", p[i].ad, bekleme);
10   }
11   printf("ort = %.2f\n", (double) top / n);
12 }
```



P3 9 ms çalışır, saat **21** olur; toplam bekleme 17.

Adım 7 / 10

DEĞİŞKENLER

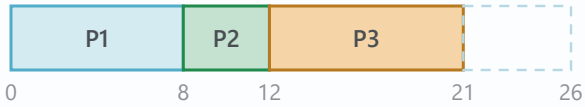
`i = 2` `t = 21` `gelis = 2` `bekleme = 10` `top = 17`

ÇIKTI

```
P1: 0
P2: 7
P3: 10
```

FCFS Bekleme Süresini Hesaplamak

```
1  /* p[] geliş sırasına göre sıralı */
2  void fcfs(Surec p[], int n) {
3    int t = 0, top = 0;
4    for (int i = 0; i < n; i++) {
▶      if (t < p[i].gelis) t = p[i].gelis;
▶      int bekleme = t - p[i].gelis;
7      t += p[i].sure;          /* bitiş */
8      top += bekleme;
9      printf("%s: %d\n", p[i].ad, bekleme);
10   }
11   printf("ort = %.2f\n", (double) top / n);
12 }
```



P4 3'te gelmişti; CPU 21'de boşaldı → bekleme = 21 − 3 = **18**.

Adım 8 / 10

DEĞİŞKENLER

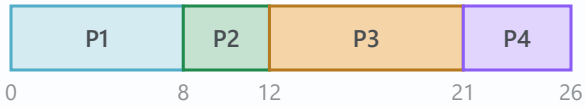
`i = 3` `t = 21` `gelis = 3` `bekleme = 18` `top = 17`

ÇIKTI

```
P1: 0
P2: 7
P3: 10
```

FCFS Bekleme Süresini Hesaplamak

```
1  /* p[] geliş sırasına göre sıralı */
2  void fcfs(Surec p[], int n) {
3    int t = 0, top = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].gelis) t = p[i].gelis;
6      int bekleme = t - p[i].gelis;
▶      t += p[i].sure;          /* bitiş */
▶      top += bekleme;
▶      printf("%s: %d\n", p[i].ad, bekleme);
10   }
11   printf("ort = %.2f\n", (double) top / n);
12 }
```



P4 5 ms çalışır, saat **26** olur; toplam bekleme 35.

Adım 9 / 10

DEĞİŞKENLER

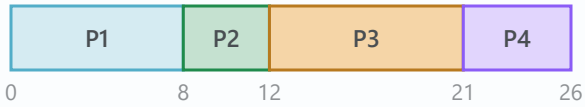
`i = 3` `t = 26` `gelis = 3` `bekleme = 18` `top = 35`

ÇIKTI

```
P1: 0
P2: 7
P3: 10
P4: 18
```

FCFS Bekleme Süresini Hesaplamak

```
1  /* p[] geliş sırasına göre sıralı */
2  void fcfs(Surec p[], int n) {
3    int t = 0, top = 0;
4    for (int i = 0; i < n; i++) {
5      if (t < p[i].gelis) t = p[i].gelis;
6      int bekleme = t - p[i].gelis;
7      t += p[i].sure;          /* bitiş */
8      top += bekleme;
9      printf("%s: %d\n", p[i].ad, bekleme);
10   }
▶    printf("ort = %.2f\n", (double) top / n);
12 }
```



Ortalama bekleme = 35 / 4 = **8,75 ms**, FCFS Gantt şemasıyla aynı.

Adım 10 / 10

DEĞİŞKENLER

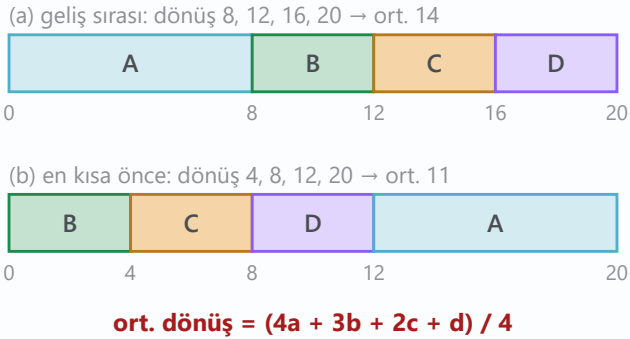
`i = 4` `t = 26` `top = 35`

ÇIKTI

```
P1: 0
P2: 7
P3: 10
P4: 18
ort = 8.75
```

SJF: En Kısa İş Önce

- **En kısa iş önce** (shortest-job-first): hazır süreçlerden **sonraki CPU patlaması** en kısa olan seçilir; eşitlikte FCFS.
- Tüm işler aynı anda hazırsa **ortalama bekleme süresini en aza indirdiği kanıtlanabilir**: ilk işin süresi herkesin beklemesine eklenir, o yüzden en kısası öne alınmalıdır (sağdaki formülde a'nın katsayısı en büyük).
- Gelişler farklı zamanlardaysa kesmesiz SJF optimal değildir; optimal olan kesmeli sürümü **SRTF**'dir.
- Sorun: sonraki patlama süresi **bilinmez**. Toplu sistemde kullanıcı tahmini (ör. Slurm `--time`), kısa vadeli çizelgelemede geçmişten tahmin.
- Uzun işler sürekli gelen kısa işler yüzünden **açlık** çekebilir.

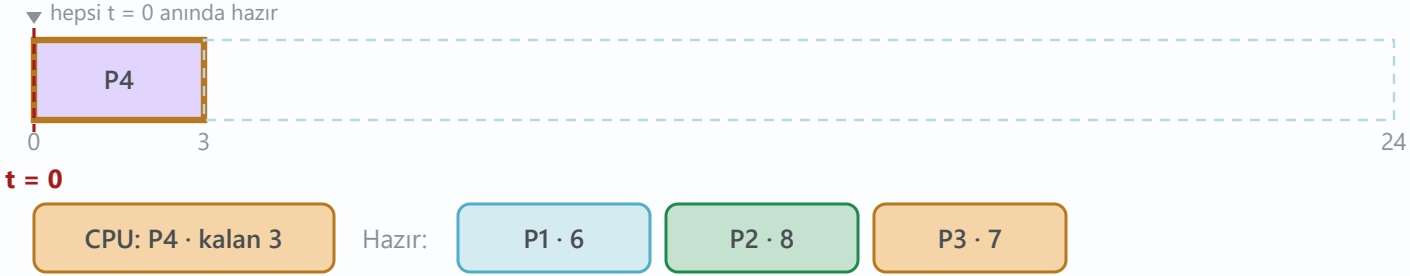


ort. dönüş = (4a + 3b + 2c + d) / 4

SJF, işlerin hepsi aynı anda hazırken **ortalama bekleme ve dönüş süresini** en aza indirir; CPU kullanımını artırmayı hedeflemez, CPU her algoritmada iş olduğu sürece meşguldür.

SJF: Hepsi Aynı Anda Hazır

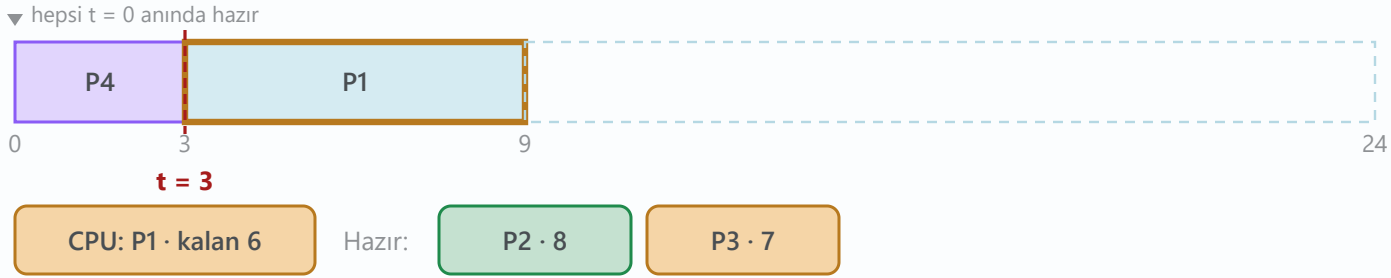
Klasik örnek: dört süreç **t = 0**'da hazır; CPU süreleri P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Aynı küme FCFS ile çalışsaydı ortalama bekleme (0 + 6 + 14 + 21) / 4 = **10,25 ms** olurdu.



t = 0: hepsi hazır; CPU süresi en kısa olan **P4** (3) seçilir.

SJF: Hepsi Aynı Anda Hazır

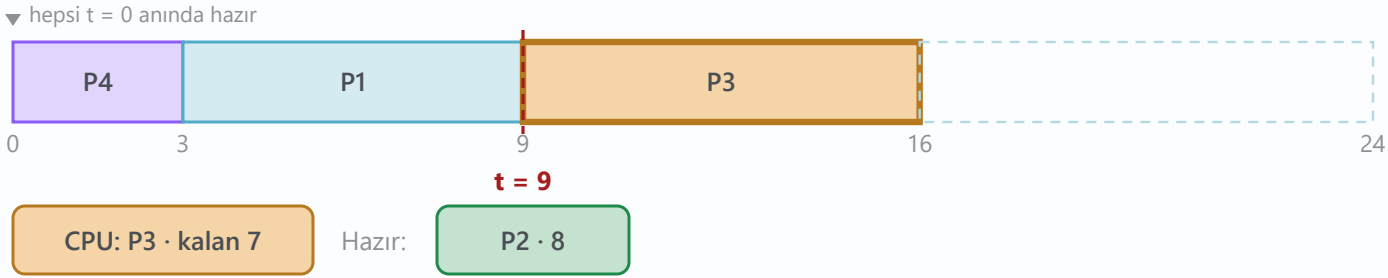
Klasik örnek: dört süreç **t = 0**'da hazır; CPU süreleri P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Aynı küme FCFS ile çalışsaydı ortalama bekleme (0 + 6 + 14 + 21) / 4 = **10,25 ms** olurdu.



t = 3: P4 bitti. Hazır: P1 (6), P2 (8), P3 (7) → en kısa **P1** (6).

SJF: Hepsi Aynı Anda Hazır

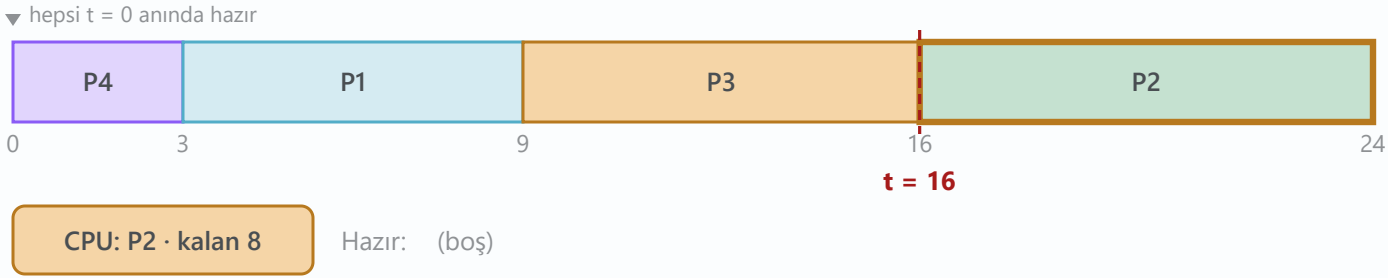
Klasik örnek: dört süreç **t = 0**'da hazır; CPU süreleri P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Aynı küme FCFS ile çalışsaydı ortalama bekleme (0 + 6 + 14 + 21) / 4 = **10,25 ms** olurdu.



t = 9: P1 bitti. Hazır: P3 (7), P2 (8) → en kısa **P3** (7).

SJF: Hepsi Aynı Anda Hazır

Klasik örnek: dört süreç **t = 0**'da hazır; CPU süreleri P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Aynı küme FCFS ile çalışsaydı ortalama bekleme (0 + 6 + 14 + 21) / 4 = **10,25 ms** olurdu.



t = 16: P3 bitti. Hazır: P2 (8) → en kısa **P2** (8).

SJF: Hepsi Aynı Anda Hazır

Klasik örnek: dört süreç **t = 0**'da hazır; CPU süreleri P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. Aynı küme FCFS ile çalışsaydı ortalama bekleme (0 + 6 + 14 + 21) / 4 = **10,25 ms** olurdu.



Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	6	3	9	9	3	3
P2	0	8	16	24	24	16	16
P3	0	7	9	16	16	9	9
P4	0	3	0	3	3	0	0
Ort.					**13**	**7**	**7**

Ortalama bekleme **7**, dönüş **13**, tepki **7** ms · 3 bağlam değiştirme.

SJF Ortak Kümede

Ortak kümede kesmesiz SJF: t = 0'da yalnızca P1 vardır ve kesilemez; sonraki her seçimde hazır süreçlerin **en kısası** alınır.



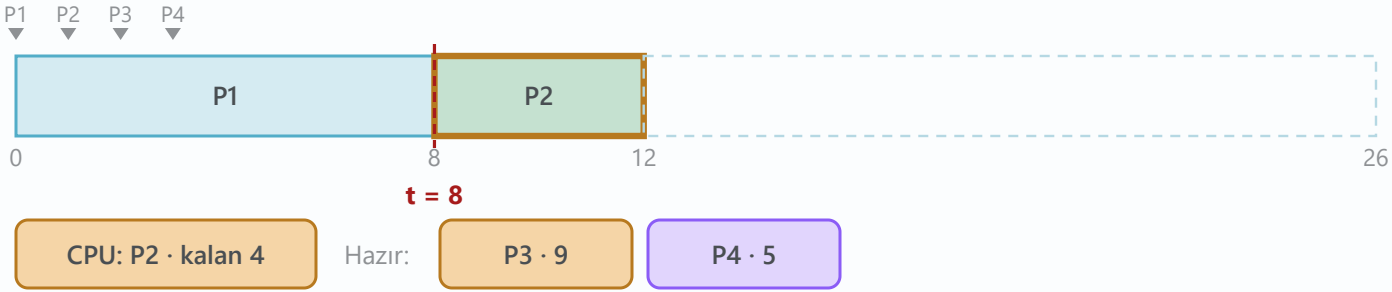
t = 0

CPU: P1 · kalan 8 Hazır: (boş)

t = 0: yalnızca **P1** hazır. SJF kesmesiz: sonradan gelen kısa işler P1'i durduramaz.

SJF Ortak Kümede

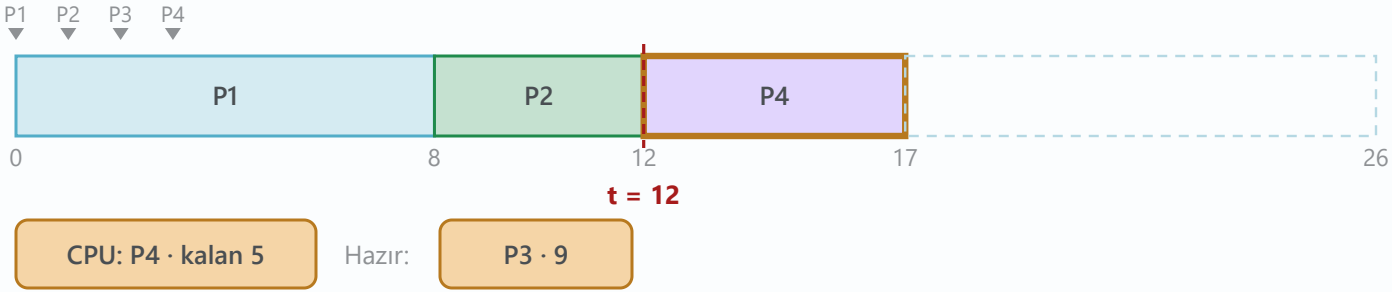
Ortak kümede kesmesiz SJF: t = 0'da yalnızca P1 vardır ve kesilemez; sonraki her seçimde hazır süreçlerin **en kısası** alınır.



t = 8: P1 bitti. Hazır: P2 (4), P3 (9), P4 (5) → en kısa **P2** (4).

SJF Ortak Kümede

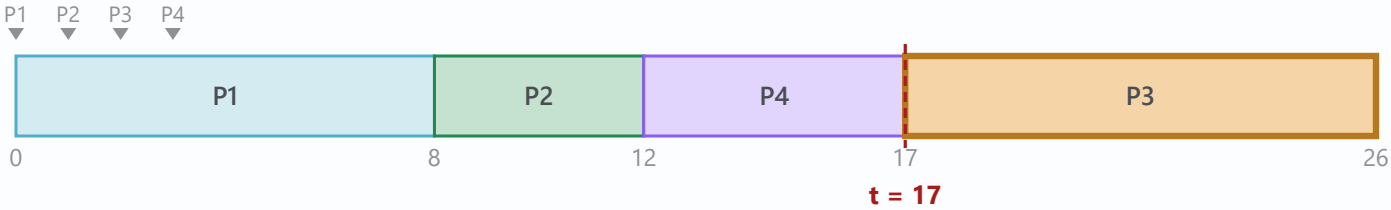
Ortak kümede kesmesiz SJF: t = 0'da yalnızca P1 vardır ve kesilemez; sonraki her seçimde hazır süreçlerin **en kısası** alınır.



t = 12: P2 bitti. Hazır: P3 (9), P4 (5) → en kısa **P4** (5).

SJF Ortak Kümede

Ortak kümede kesmesiz SJF: t = 0'da yalnızca P1 vardır ve kesilemez; sonraki her seçimde hazır süreçlerin **en kısası** alınır.



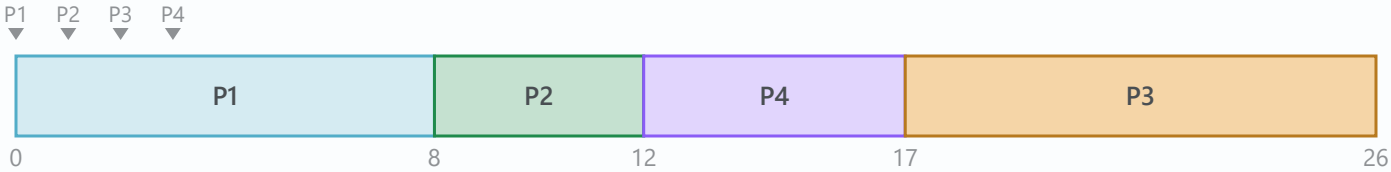
CPU: P3 · kalan 9

Hazır: (boş)

t = 17: P4 bitti. Hazır: P3 (9) → en kısa **P3** (9).

SJF Ortak Kümede

Ortak kümede kesmesiz SJF: t = 0'da yalnızca P1 vardır ve kesilemez; sonraki her seçimde hazır süreçlerin **en kısası** alınır.



Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	0	8	8	0	0
P2	1	4	8	12	11	7	7
P3	2	9	17	26	24	15	15
P4	3	5	12	17	14	9	9
Ort.					**14,25**	**7,75**	**7,75**

Ortalama bekleme **7,75**, dönüş **14,25**, tepki **7,75** ms · 3 bağlam değiştirme.

FCFS ve SJF Karşılaştırması

		A	B	C	D	E	Ort.
	Geliş	0	1	2	3	4	
	CPU	4	3	5	2	4	
SJF	Bitiş	4	9	18	6	13	
	Dönüş	4	8	16	3	9	**8**
	Ağırlıklı	1	2,67	3,2	1,5	2,25	**2,12**
FCFS	Bitiş	4	7	12	14	18	
	Dönüş	4	6	10	11	14	**9**
	Ağırlıklı	1	2	2	5,5	3,5	**2,8**

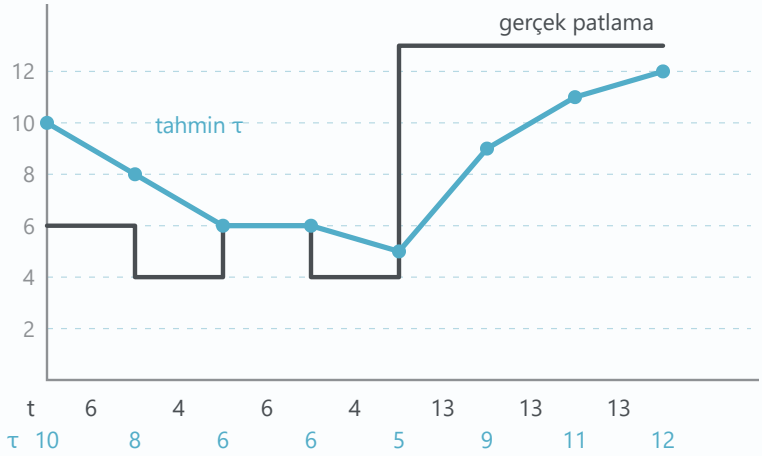
Ağırlıklı (normalleştirilmiş) dönüş = dönüş / CPU süresi; 1 ideal. FCFS kısa işleri cezalandırır: A(0, 1), B(1, 100), C(2, 1), D(3, 100) kümesinde 1 ms'lik C, B'nin arkasında 99 ms bekler, ağırlıklı dönüşü **100** olur (D için 1,99).

C için ağırlıklı dönüş 16 / 5 = **3,2**; SJF ortalaması 2,12'dir.

Sonraki CPU Patlamasını Tahmin: Üstel Ortalama

$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n, \quad 0 \le \alpha \le 1$

- t_n: n. patlamanın gerçek süresi; τ_n: onun için yapılmış tahmin; τ_0: sabit (ör. sistem ortalaması).
- $\alpha = 0$ → son ölçüm yok sayılır, tahmin hiç değişmez; $\alpha = 1$ → yalnızca son patlama sayılır. Yaygın seçim $\alpha = ½$.
- Açılım: $\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \ldots + (1-\alpha)^{n+1}\tau_0$: eski ölçümlerin ağırlığı üstel azalır.
- Tanenbaum'un "yaşlandırma" (aging) tahmini aynı fikir: T_0, $T_0/2 + T_1/2$, $T_0/4 + T_1/4 + T_2/2$, …
- Örnek ($\alpha = ½$, $\tau_0 = 10$): $t_0 = 6$ → $\tau_1 = ½\cdot 6 + ½\cdot 10 = \mathbf{8}$.



Üstel Ortalamayı Hesaplamak

```
▶ double tau = 10.0;       /* τ0 */
▶ const double a = 0.5;  /* α */
▶ int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
5   printf("%.1f %d\n", tau, t[n]);
6   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("sonraki: %.1f\n", tau);
```



İlk tahmin τ_0 = 10, α = 0,5. Henüz hiç patlama ölçülmedi.

Adım 1 / 9

DEĞİŞKENLER

`tau = 10.0` `a = 0.5`

ÇIKTI

Üstel Ortalamayı Hesaplamak

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("sonraki: %.1f\n", tau);
```



n = 0: gerçek 6, tahmin 10 idi → τ = 0,5·6 + 0,5·10 = **8**.

Adım 2 / 9

DEĞİŞKENLER

`n = 0` `t[n] = 6` `tau = 8.0`

ÇIKTI

```
10.0 6
```

Üstel Ortalamayı Hesaplamak

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("sonraki: %.1f\n", tau);
```



n = 1: gerçek 4, tahmin 8 idi → τ = 0,5·4 + 0,5·8 = **6**.

Adım 3 / 9

DEĞİŞKENLER

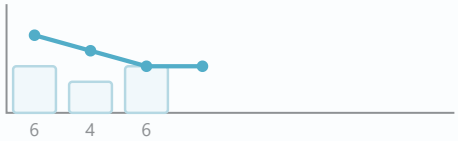
`n = 1` `t[n] = 4` `tau = 6.0`

ÇIKTI

```
10.0 6
8.0 4
```

Üstel Ortalamayı Hesaplamak

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("sonraki: %.1f\n", tau);
```



n = 2: gerçek 6, tahmin 6 idi → τ = 0,5·6 + 0,5·6 = **6**.

Adım 4 / 9

DEĞİŞKENLER

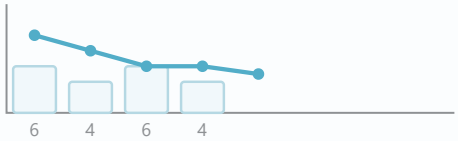
`n = 2` `t[n] = 6` `tau = 6.0`

ÇIKTI

```
10.0 6
8.0 4
6.0 6
```

Üstel Ortalamayı Hesaplamak

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("sonraki: %.1f\n", tau);
```



n = 3: gerçek 4, tahmin 6 idi → τ = 0,5·4 + 0,5·6 = **5**.

Adım 5 / 9

DEĞİŞKENLER

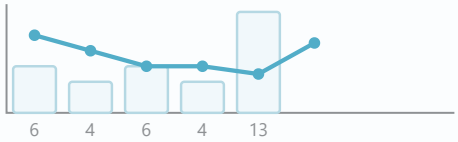
`n = 3` `t[n] = 4` `tau = 5.0`

ÇIKTI

```
10.0 6
8.0 4
6.0 6
6.0 4
```

Üstel Ortalamayı Hesaplamak

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("sonraki: %.1f\n", tau);
```



n = 4: gerçek 13, tahmin 5 idi → τ = 0,5·13 + 0,5·5 = **9**.

Adım 6 / 9

DEĞİŞKENLER

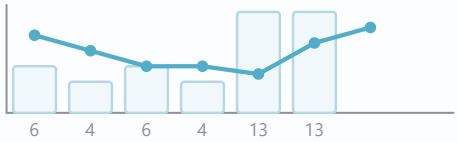
`n = 4` `t[n] = 13` `tau = 9.0`

ÇIKTI

```
10.0 6
8.0 4
6.0 6
6.0 4
5.0 13
```

Üstel Ortalamayı Hesaplamak

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("sonraki: %.1f\n", tau);
```



n = 5: gerçek 13, tahmin 9 idi → τ = 0,5·13 + 0,5·9 = **11**.

Adım 7 / 9

DEĞİŞKENLER

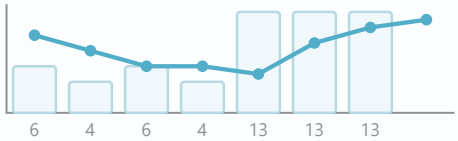
`n = 5` `t[n] = 13` `tau = 11.0`

ÇIKTI

```
10.0 6
8.0 4
6.0 6
6.0 4
5.0 13
9.0 13
```

Üstel Ortalamayı Hesaplamak

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
▶   printf("%.1f %d\n", tau, t[n]);
▶   tau = a * t[n] + (1 - a) * tau;
7 }
8 printf("sonraki: %.1f\n", tau);
```



n = 6: gerçek 13, tahmin 11 idi → τ = 0,5·13 + 0,5·11 = **12**.

Adım 8 / 9

DEĞİŞKENLER

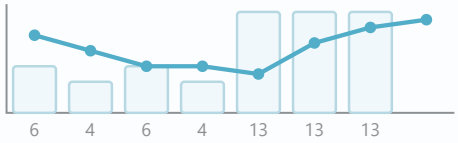
`n = 6` `t[n] = 13` `tau = 12.0`

ÇIKTI

```
10.0 6
8.0 4
6.0 6
6.0 4
5.0 13
9.0 13
11.0 13
```

Üstel Ortalamayı Hesaplamak

```
1 double tau = 10.0;       /* τ0 */
2 const double a = 0.5;  /* α */
3 int t[] = { 6, 4, 6, 4, 13, 13, 13 };
4 for (int n = 0; n < 7; n++) {
5   printf("%.1f %d\n", tau, t[n]);
6   tau = a * t[n] + (1 - a) * tau;
7 }
▶ printf("sonraki: %.1f\n", tau);
```



Patlamalar 13'e sıçradı; tahmin 5 → 9 → 11 → **12** ile birkaç adımda yetişir.

Adım 9 / 9

DEĞİŞKENLER

`n = 7` `tau = 12.0`

ÇIKTI

```
10.0 6
8.0 4
6.0 6
6.0 4
5.0 13
9.0 13
11.0 13
sonraki: 12.0
```

SRTF Adım Adım

En kısa kalan süre önce (SRTF / SRTN): SJF'nin kesmeli sürümü. Yeni süreç gelince süresi, çalışanın **kalan** süresiyle karşılaştırılır.



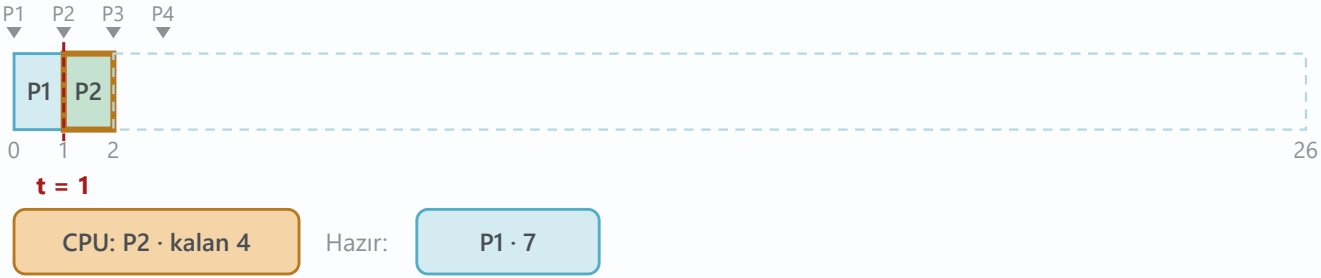
t = 0: yalnızca **P1** var ve çalışmaya başlar.

Adım 1 / 8

P1 t = 0'da başlar, 1'de kesilir, 10'da kaldığı yerden devam eder. Ortalama bekleme 6,5 ms'dir.

SRTF Adım Adım

En kısa kalan süre önce (SRTF / SRTN): SJF'nin kesmeli sürümü. Yeni süreç gelince süresi, çalışanın **kalan** süresiyle karşılaştırılır.

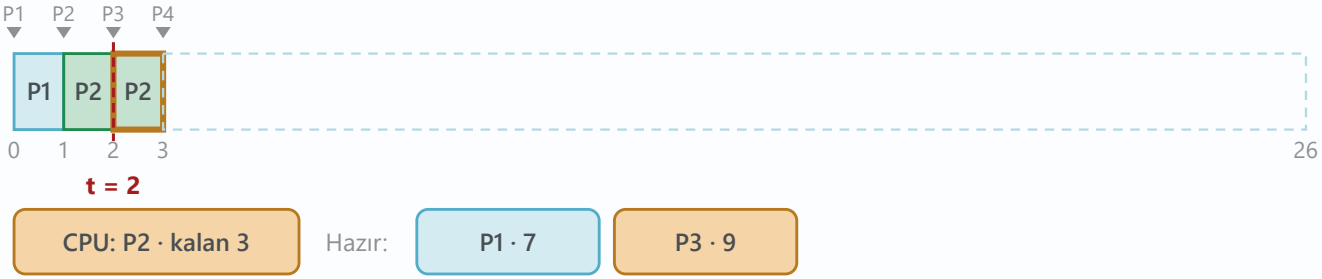


t = 1: **P2** gelir (4); çalışan P1 için kalan 7. 4 < 7 → P1 kesilir, **P2** çalışır.

P1 t = 0'da başlar, 1'de kesilir, 10'da kaldığı yerden devam eder. Ortalama bekleme 6,5 ms'dir.

SRTF Adım Adım

En kısa kalan süre önce (SRTF / SRTN): SJF'nin kesmeli sürümü. Yeni süreç gelince süresi, çalışanın **kalan** süresiyle karşılaştırılır.

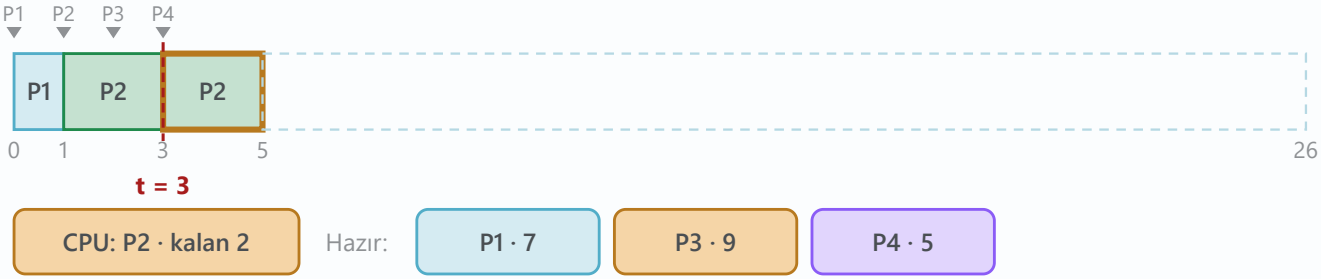


t = 2: **P3** gelir (9). Çalışan P2 için kalan 3 ≤ 9 → kesme yok.

P1 t = 0'da başlar, 1'de kesilir, 10'da kaldığı yerden devam eder. Ortalama bekleme 6,5 ms'dir.

SRTF Adım Adım

En kısa kalan süre önce (SRTF / SRTN): SJF'nin kesmeli sürümü. Yeni süreç gelince süresi, çalışanın **kalan** süresiyle karşılaştırılır.

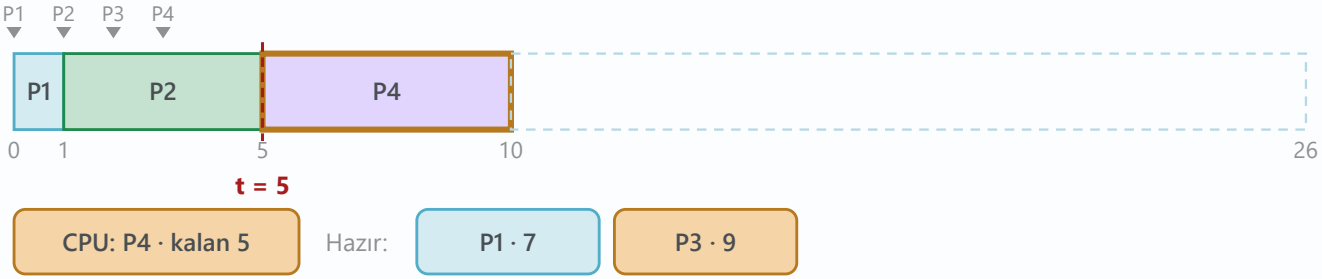


t = 3: **P4** gelir (5). Çalışan P2 için kalan 2 ≤ 5 → kesme yok.

P1 t = 0'da başlar, 1'de kesilir, 10'da kaldığı yerden devam eder. Ortalama bekleme 6,5 ms'dir.

SRTF Adım Adım

En kısa kalan süre önce (SRTF / SRTN): SJF'nin kesmeli sürümü. Yeni süreç gelince süresi, çalışanın **kalan** süresiyle karşılaştırılır.

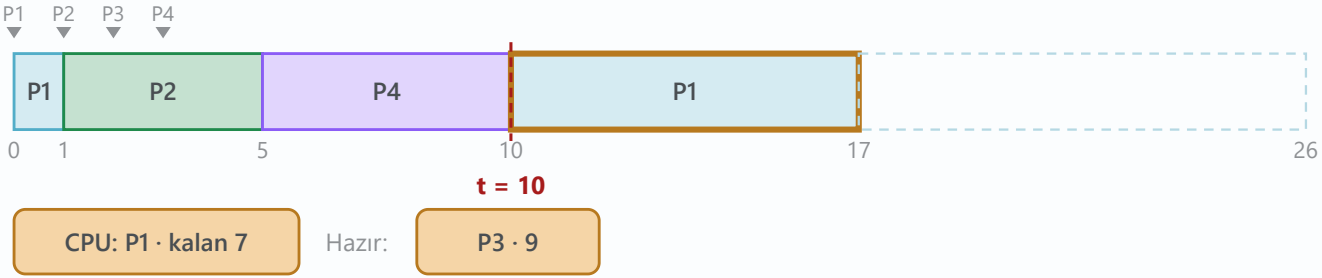


t = 5: P2 bitti. Kalan süreler: P1 7, P3 9, P4 5 → en kısa **P4** (5).

P1 t = 0'da başlar, 1'de kesilir, 10'da kaldığı yerden devam eder. Ortalama bekleme 6,5 ms'dir.

SRTF Adım Adım

En kısa kalan süre önce (SRTF / SRTN): SJF'nin kesmeli sürümü. Yeni süreç gelince süresi, çalışanın **kalan** süresiyle karşılaştırılır.

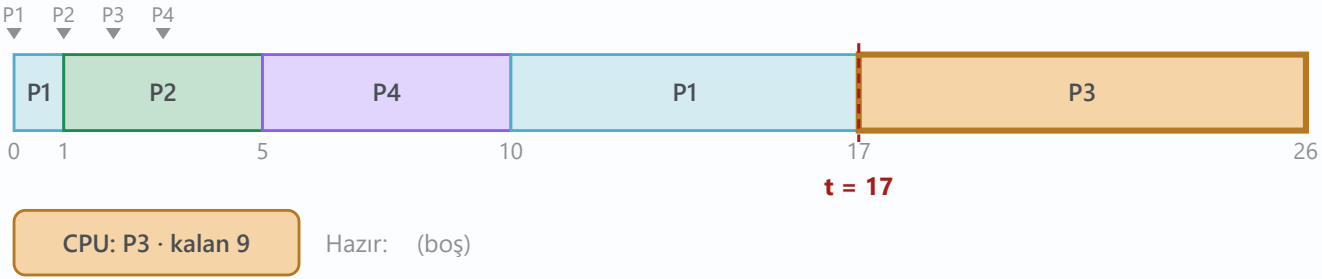


t = 10: P4 bitti. Kalan süreler: P1 7, P3 9 → en kısa **P1** (7).

P1 t = 0'da başlar, 1'de kesilir, 10'da kaldığı yerden devam eder. Ortalama bekleme 6,5 ms'dir.

SRTF Adım Adım

En kısa kalan süre önce (SRTF / SRTN): SJF'nin kesmeli sürümü. Yeni süreç gelince süresi, çalışanın **kalan** süresiyle karşılaştırılır.



CPU: P3 · kalan 9

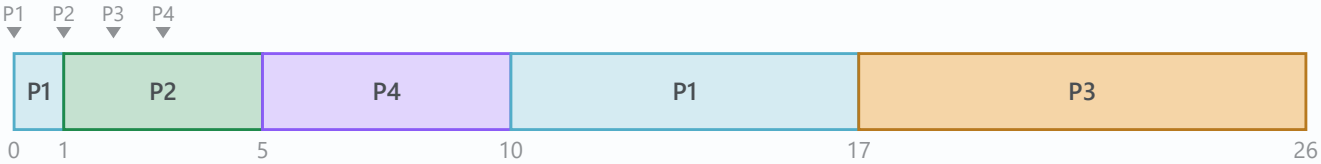
Hazır: (boş)

t = 17: P1 bitti. Kalan süreler: P3 9 → en kısa **P3** (9).

P1 t = 0'da başlar, 1'de kesilir, 10'da kaldığı yerden devam eder. Ortalama bekleme 6,5 ms'dir.

SRTF Adım Adım

En kısa kalan süre önce (SRTF / SRTN): SJF'nin kesmeli sürümü. Yeni süreç gelince süresi, çalışanın **kalan** süresiyle karşılaştırılır.



Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	0	17	17	9	0
P2	1	4	1	5	4	0	0
P3	2	9	17	26	24	15	15
P4	3	5	5	10	7	2	2
Ort.					**13**	**6,5**	**4,25**

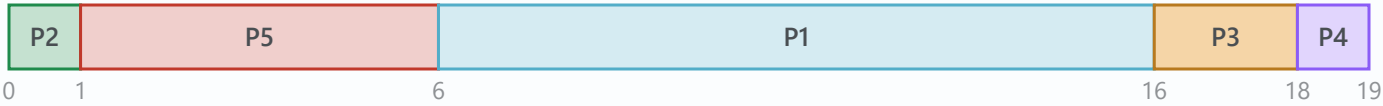
Ortalama bekleme **6,5**, dönüş **13**, tepki **4,25** ms · 4 bağlam değiştirme.

P1 t = 0'da başlar, 1'de kesilir, 10'da kaldığı yerden devam eder. Ortalama bekleme 6,5 ms'dir.

Öncelik Çizelgeleme

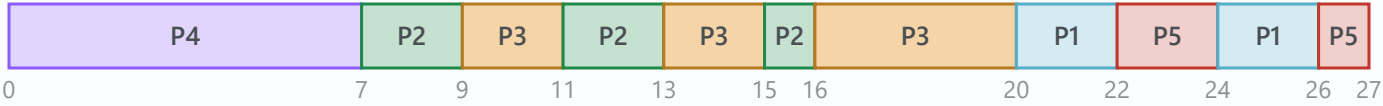
- Her sürece bir **öncelik** sayısı verilir; en yüksek öncelikli hazır süreç çalışır. Bu derste küçük sayı = yüksek öncelik (Linux ve Windows'ta yön farklıdır).
- Öncelik **iç** (bellek, açık dosya, ölçülen G/Ç/CPU oranı) ya da **dış** (önem, ücret, politika) etkenlerden gelir. SJF, önceliği "tahmini sonraki patlama" olan özel bir durumdur; tüm öncelikler eşitse algoritma FCFS'ye döner.
- Kesmeli ya da kesmesiz olabilir; aynı öncelik düzeyindekiler arasında RR uygulanabilir (b).
- Yan etkisi **öncelik terslemesi**dir: yüksek öncelikli iş, düşük öncelikli işin tuttuğu kilidi beklerken orta öncelikli işler araya girer; çözümü öncelik kalıtımıdır.

(a) Kesmesiz öncelik · P1 (10, ö 3), P2 (1, ö 1), P3 (2, ö 4), P4 (1, ö 5), P5 (5, ö 2), hepsi t = 0



Bekleme: P1 6, P2 0, P3 16, P4 18, P5 1 → ortalama 41 / 5 = 8,2 ms

(b) Öncelik + RR (q = 2) · P1 (4, ö 3), P2 (5, ö 2), P3 (8, ö 2), P4 (7, ö 1), P5 (3, ö 3)



Bekleme: P1 22, P2 11, P3 12, P4 0, P5 24 → ortalama 13,8 ms

Kesmeli Öncelik Ortak Kümede

Kesmeli öncelik, ortak küme (P1 ö 2, P2 ö 3, P3 ö 1, P4 ö 4). Yeni gelen süreç çalışandan **daha yüksek öncelikliyse** onu keser.



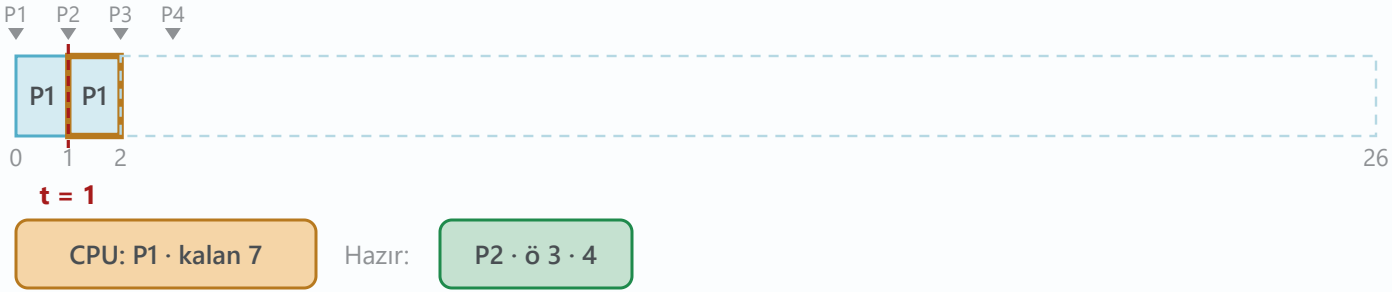
t = 0

CPU: P1 · kalan 8 Hazır: (boş)

t = 0: yalnızca **P1** var ve çalışmaya başlar.

Kesmeli Öncelik Ortak Kümede

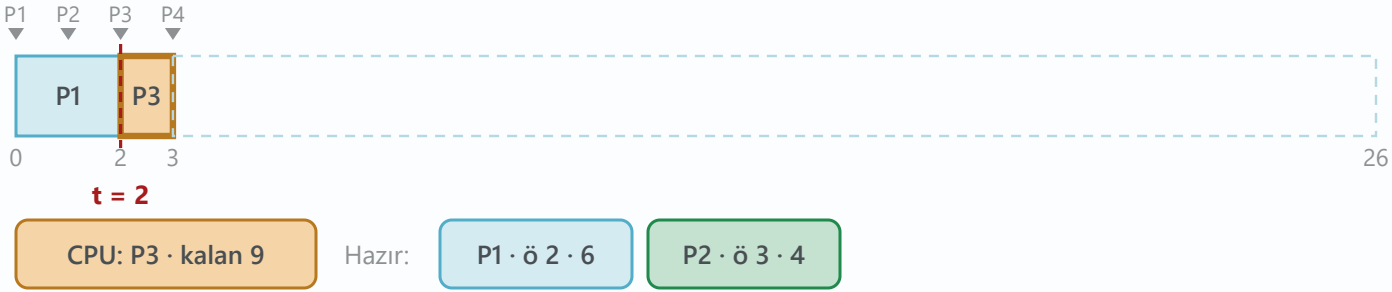
Kesmeli öncelik, ortak küme (P1 ö 2, P2 ö 3, P3 ö 1, P4 ö 4). Yeni gelen süreç çalışandan **daha yüksek öncelikliyse** onu keser.



t = 1: **P2** gelir (öncelik 3). Çalışan P1'in önceliği (2) daha yüksek → kesme yok.

Kesmeli Öncelik Ortak Kümede

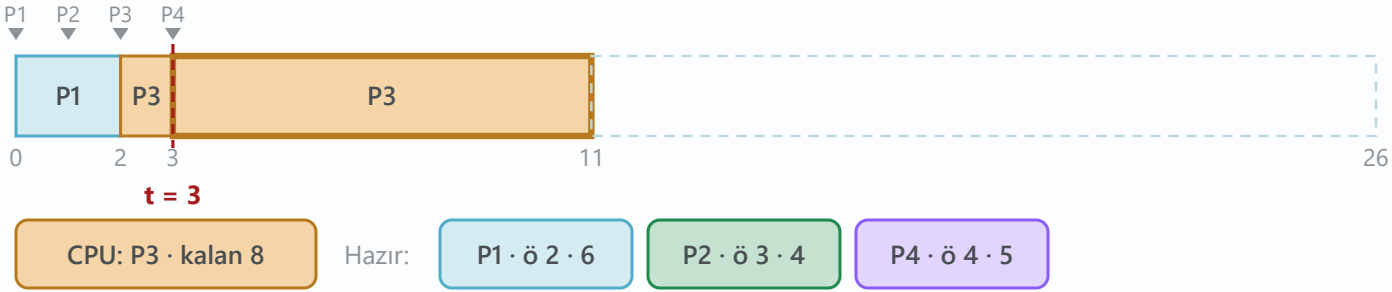
Kesmeli öncelik, ortak küme (P1 ö 2, P2 ö 3, P3 ö 1, P4 ö 4). Yeni gelen süreç çalışandan **daha yüksek öncelikliyse** onu keser.



t = 2: **P3** gelir (öncelik 1); çalışan P1'in önceliği (2) daha düşük → kesilir, **P3** çalışır.

Kesmeli Öncelik Ortak Kümede

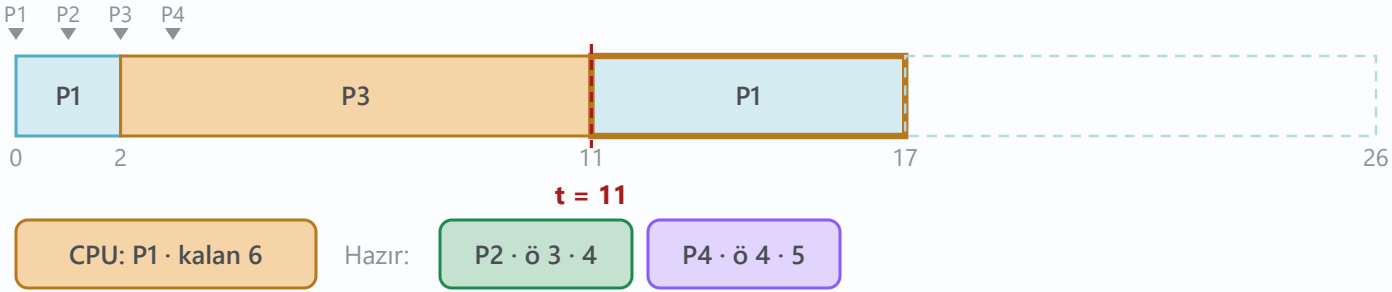
Kesmeli öncelik, ortak küme (P1 ö 2, P2 ö 3, P3 ö 1, P4 ö 4). Yeni gelen süreç çalışandan **daha yüksek öncelikliyse** onu keser.



t = 3: **P4** gelir (öncelik 4). Çalışan P3'ün önceliği (1) daha yüksek → kesme yok.

Kesmeli Öncelik Ortak Kümede

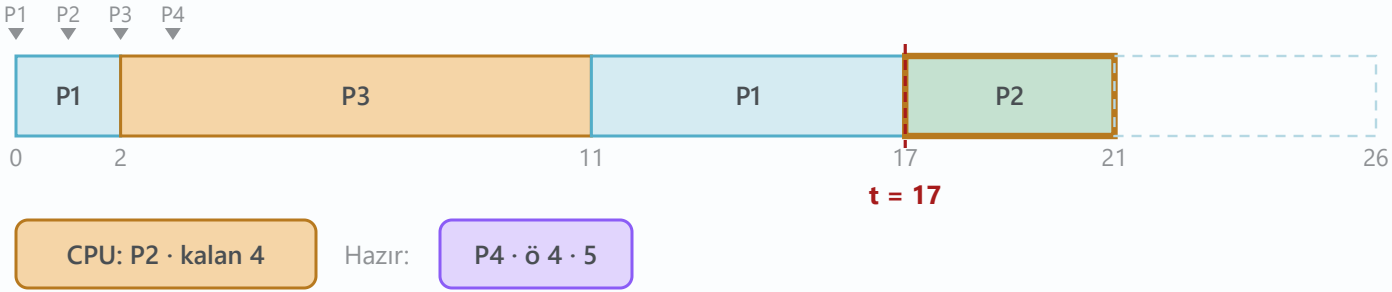
Kesmeli öncelik, ortak küme (P1 ö 2, P2 ö 3, P3 ö 1, P4 ö 4). Yeni gelen süreç çalışandan **daha yüksek öncelikliyse** onu keser.



t = 11: P3 bitti. Hazır: P1 (ö 2), P2 (ö 3), P4 (ö 4) → en yüksek öncelikli **P1**.

Kesmeli Öncelik Ortak Kümede

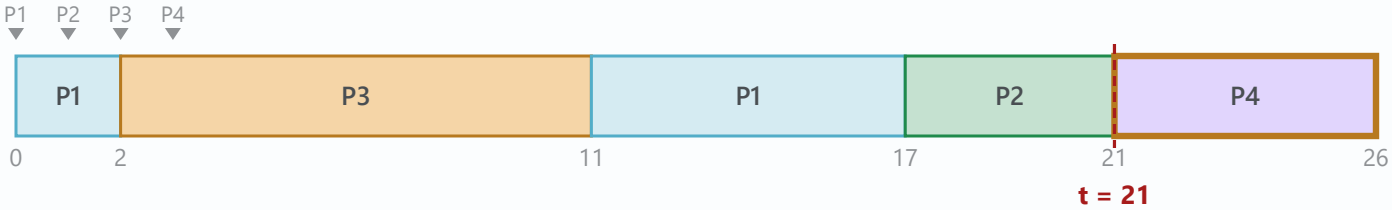
Kesmeli öncelik, ortak küme (P1 ö 2, P2 ö 3, P3 ö 1, P4 ö 4). Yeni gelen süreç çalışandan **daha yüksek öncelikliyse** onu keser.



t = 17: P1 bitti. Hazır: P2 (ö 3), P4 (ö 4) → en yüksek öncelikli **P2**.

Kesmeli Öncelik Ortak Kümede

Kesmeli öncelik, ortak küme (P1 ö 2, P2 ö 3, P3 ö 1, P4 ö 4). Yeni gelen süreç çalışandan **daha yüksek öncelikliyse** onu keser.



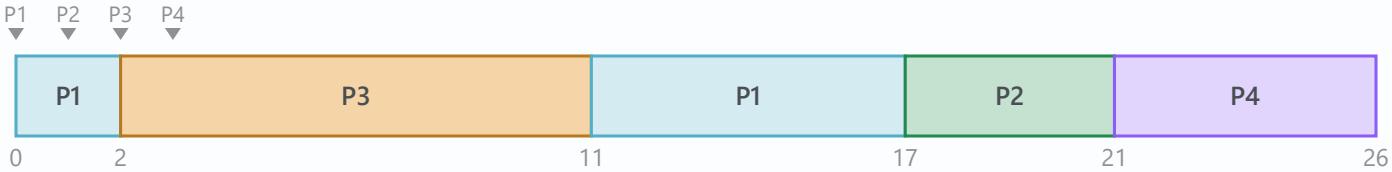
CPU: P4 · kalan 5

Hazır: (boş)

t = 21: P2 bitti. Hazır: P4 (ö 4) → en yüksek öncelikli **P4**.

Kesmeli Öncelik Ortak Kümede

Kesmeli öncelik, ortak küme (P1 ö 2, P2 ö 3, P3 ö 1, P4 ö 4). Yeni gelen süreç çalışandan **daha yüksek öncelikliyse** onu keser.

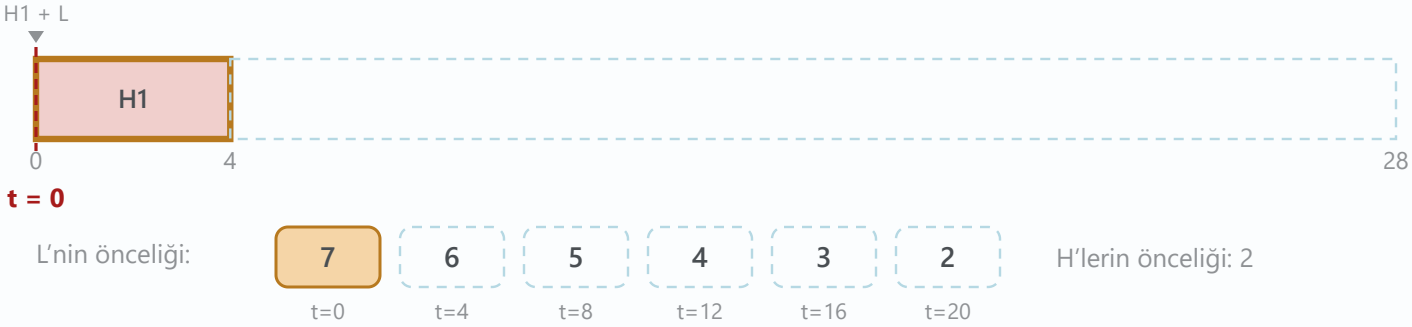


Süreç	Geliş	CPU	Öncelik	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	2	0	17	17	9	0
P2	1	4	3	17	21	20	16	16
P3	2	9	1	2	11	9	0	0
P4	3	5	4	21	26	23	18	18
Ort.						**17,25**	**10,75**	**8,5**

Ortalama bekleme **10,75**, dönüş **17,25**, tepki **8,5** ms · 4 bağlam değiştirme.

Açlık ve Yaşlandırma

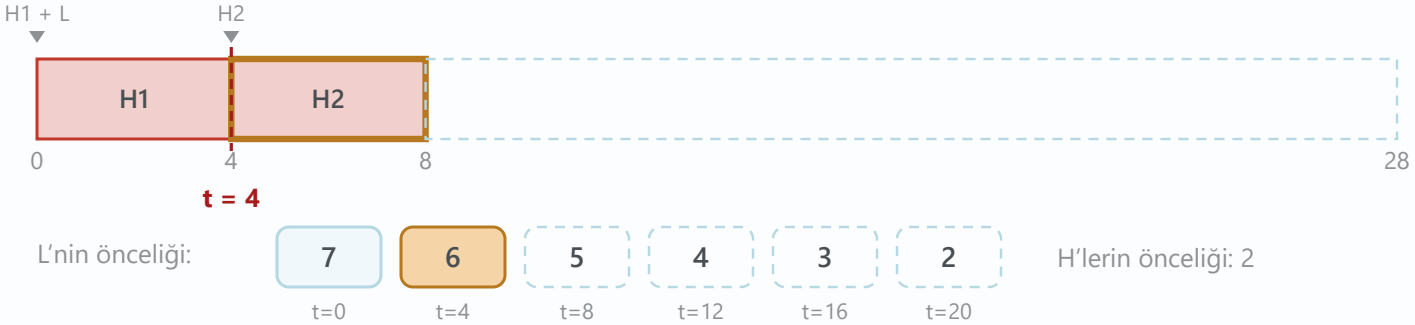
Açlık (starvation): yüksek öncelikli işler sürekli gelirse düşük öncelikli iş hiç çalışamaz. **Yaşlandırma** (aging): beklerken önceliği adım adım artırılır. Kural: hazırda geçen her 4 ms için öncelik sayısı 1 azalır. L (3 ms, ö 7) t = 0'da gelir; her 4 ms'de bir yeni H (4 ms, ö 2) gelir. Kesmesiz öncelik.



t = 0: L'nin önceliği 7, yeni gelen H1'inki 2 → **H1** çalışır. L 4 ms daha bekleyince önceliği 6 olur.

Açlık ve Yaşlandırma

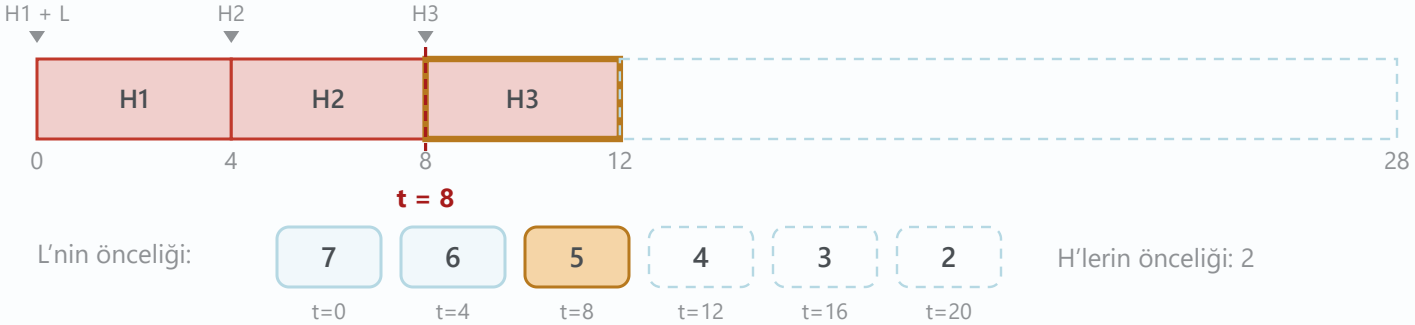
Açlık (starvation): yüksek öncelikli işler sürekli gelirse düşük öncelikli iş hiç çalışamaz. **Yaşlandırma** (aging): beklerken önceliği adım adım artırılır. Kural: hazırda geçen her 4 ms için öncelik sayısı 1 azalır. L (3 ms, ö 7) t = 0'da gelir; her 4 ms'de bir yeni H (4 ms, ö 2) gelir. Kesmesiz öncelik.



t = 4: L'nin önceliği 6, yeni gelen H2'ninki 2 → **H2** çalışır. L 4 ms daha bekleyince önceliği 5 olur.

Açlık ve Yaşlandırma

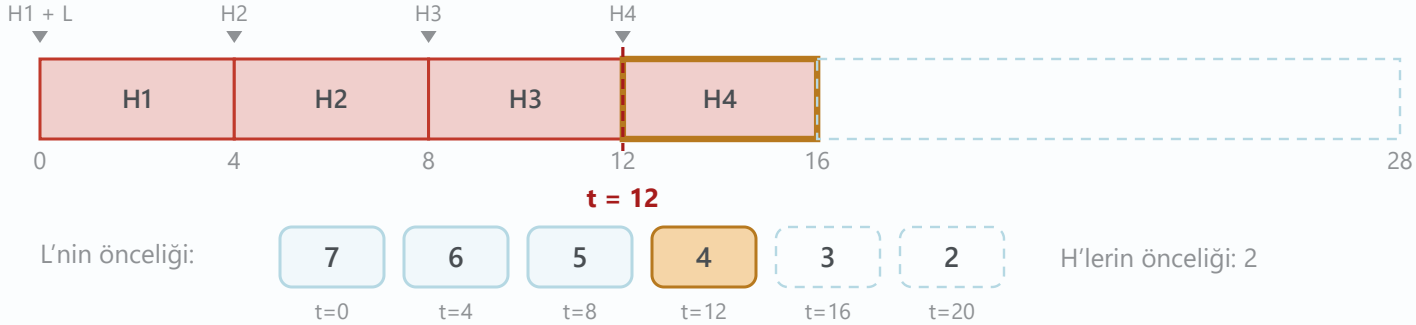
Açlık (starvation): yüksek öncelikli işler sürekli gelirse düşük öncelikli iş hiç çalışamaz. **Yaşlandırma** (aging): beklerken önceliği adım adım artırılır. Kural: hazırda geçen her 4 ms için öncelik sayısı 1 azalır. L (3 ms, ö 7) t = 0'da gelir; her 4 ms'de bir yeni H (4 ms, ö 2) gelir. Kesmesiz öncelik.



t = 8: L'nin önceliği 5, yeni gelen H3'ünki 2 → **H3** çalışır. L 4 ms daha bekleyince önceliği 4 olur.

Açlık ve Yaşlandırma

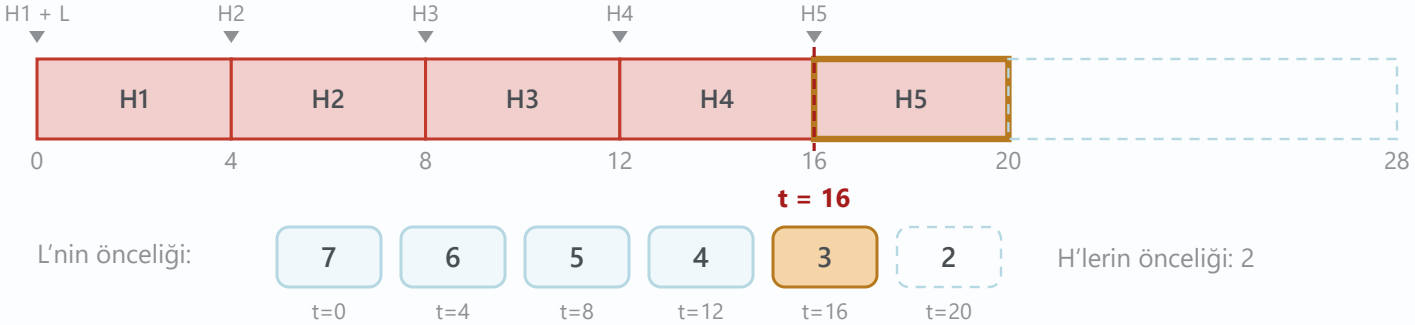
Açlık (starvation): yüksek öncelikli işler sürekli gelirse düşük öncelikli iş hiç çalışamaz. **Yaşlandırma** (aging): beklerken önceliği adım adım artırılır. Kural: hazırda geçen her 4 ms için öncelik sayısı 1 azalır. L (3 ms, ö 7) t = 0'da gelir; her 4 ms'de bir yeni H (4 ms, ö 2) gelir. Kesmesiz öncelik.



t = 12: L'nin önceliği 4, yeni gelen H4'ünki 2 → **H4** çalışır. L 4 ms daha bekleyince önceliği 3 olur.

Açlık ve Yaşlandırma

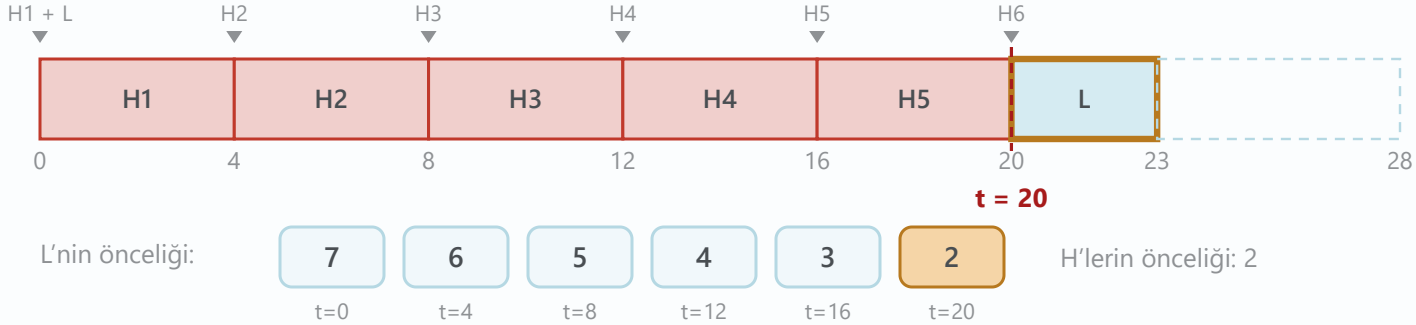
Açlık (starvation): yüksek öncelikli işler sürekli gelirse düşük öncelikli iş hiç çalışamaz. **Yaşlandırma** (aging): beklerken önceliği adım adım artırılır. Kural: hazırda geçen her 4 ms için öncelik sayısı 1 azalır. L (3 ms, ö 7) t = 0'da gelir; her 4 ms'de bir yeni H (4 ms, ö 2) gelir. Kesmesiz öncelik.



t = 16: L'nin önceliği 3, yeni gelen H5'inki 2 → **H5** çalışır. L 4 ms daha bekleyince önceliği 2 olur.

Açlık ve Yaşlandırma

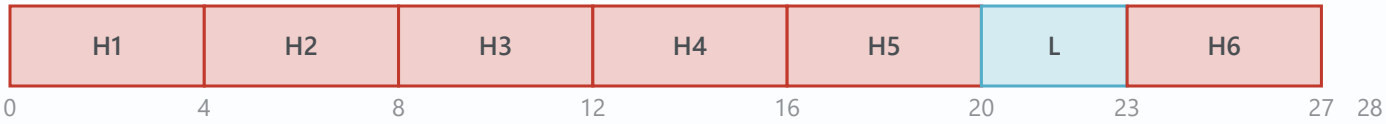
Açlık (starvation): yüksek öncelikli işler sürekli gelirse düşük öncelikli iş hiç çalışamaz. **Yaşlandırma** (aging): beklerken önceliği adım adım artırılır. Kural: hazırda geçen her 4 ms için öncelik sayısı 1 azalır. L (3 ms, ö 7) t = 0'da gelir; her 4 ms'de bir yeni H (4 ms, ö 2) gelir. Kesmesiz öncelik.



t = 20: L'nin önceliği **2**, H6 ile eşit; eşitlikte önce gelen kazanır → **L** nihayet çalışır (20 ms bekledi).

Açlık ve Yaşlandırma

Açlık (starvation): yüksek öncelikli işler sürekli gelirse düşük öncelikli iş hiç çalışamaz. **Yaşlandırma** (aging): beklerken önceliği adım adım artırılır. Kural: hazırda geçen her 4 ms için öncelik sayısı 1 azalır. L (3 ms, ö 7) t = 0'da gelir; her 4 ms'de bir yeni H (4 ms, ö 2) gelir. Kesmesiz öncelik.

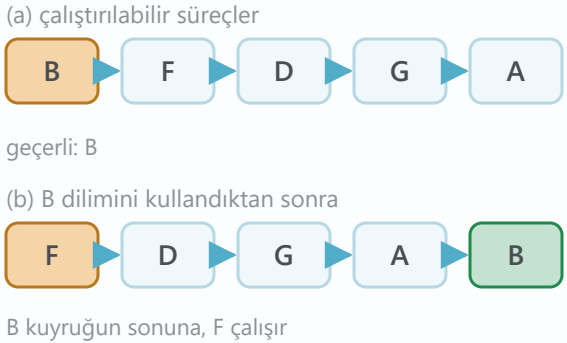


Yaşlandırma olmasaydı: H akışı sürdükçe L'nin önceliği 7'de kalır ve L hiç çalışamaz (açlık).

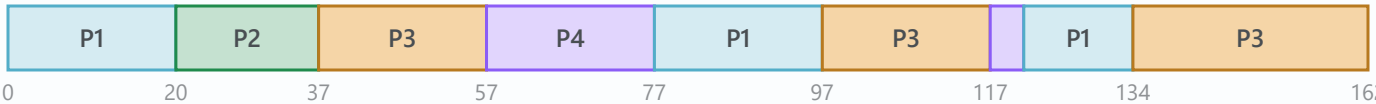
Yaşlandırma sınırsız beklemeyi **sınırlı** hale getirir: en kötü durumda L, öncelik farkı × 4 ms = 20 ms bekler.

Round Robin (RR)

- **Round Robin** (döner sıralı): FCFS + kesme. Her süreç en fazla bir **zaman dilimi** (time quantum, q) çalışır, sonra kuyruğun sonuna gider.
- n süreç ve q dilimde hiçbir süreç (n − 1) · q'dan uzun beklemez: **tepki süresi sınırlıdır**, açlık yoktur.
- Dönüş süresi genellikle SJF'den kötü, tepki süresi daha iyidir.
- G/Ç'ye bağlı süreç dilimin küçük kısmını kullanıp kuyruğun sonuna döner; CPU'ya bağlılar daha çok pay alır (bu yüzden MLFQ ve sanal RR gibi iyileştirmeler var).



Örnek: q = 20 · P1 53, P2 17, P3 68, P4 24 (hepsi t = 0)



Bekleme: P1 81, P2 20, P3 94, P4 97 → ort. 73 ms · tepki ort. 28,5 ms (SJF'de bekleme ort. 38)

RR Adım Adım: q = 2

Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 0

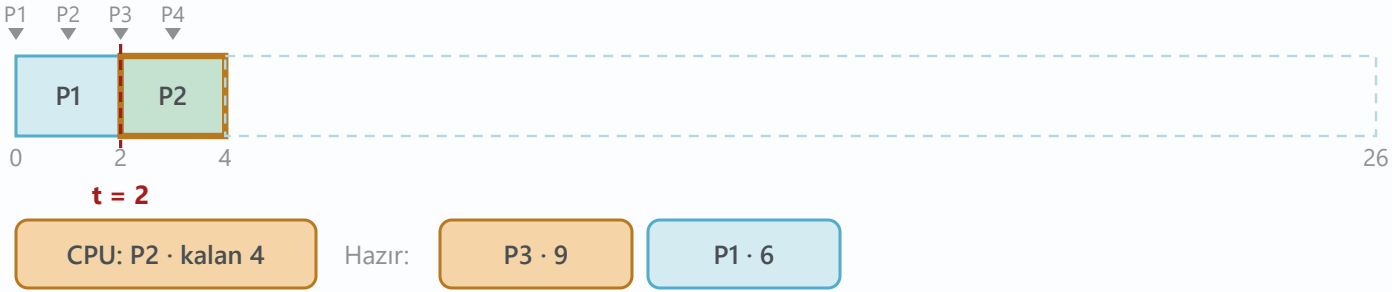
CPU: P1 · kalan 8

Hazır: (boş)

t = 0: yalnızca **P1** var; en fazla q = 2 ms çalışır, sonra sırayı bırakır.

RR Adım Adım: q = 2

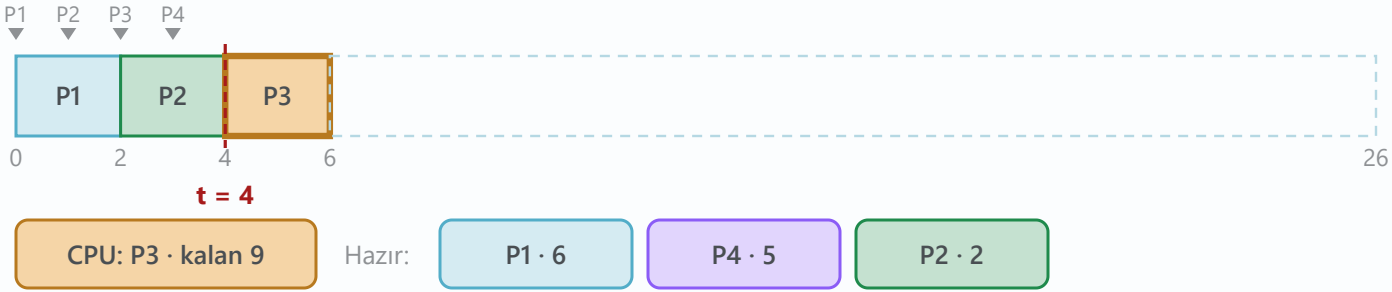
Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 2: P1 dilimini doldurdu (kalan 6) ve kuyruğun sonuna gider. Arada gelen P2, P3 kuyruğa ondan önce girdi. Sıradaki: **P2**.

RR Adım Adım: q = 2

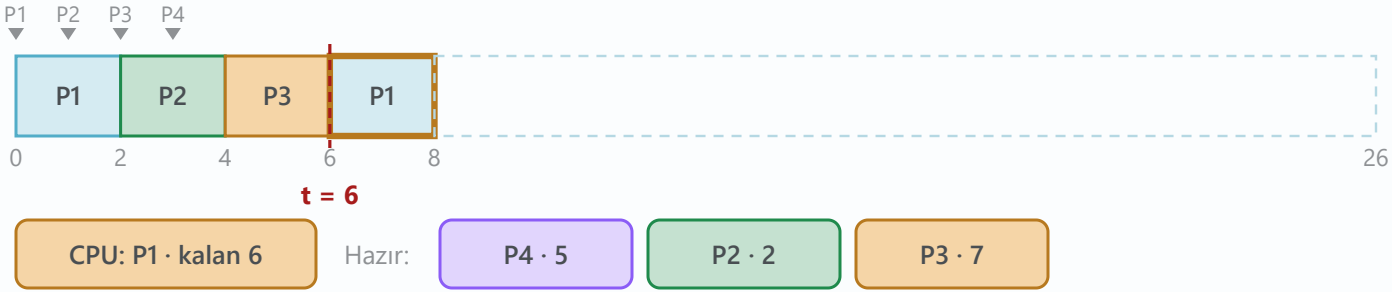
Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 4: P2 dilimini doldurdu (kalan 2) ve kuyruğun sonuna gider. Arada gelen P4 kuyruğa ondan önce girdi. Sıradaki: **P3**.

RR Adım Adım: q = 2

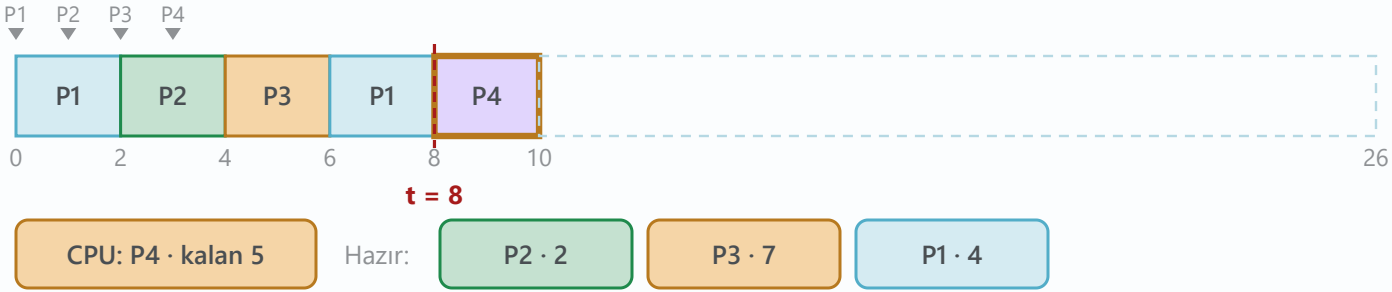
Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 6: P3 dilimini doldurdu (kalan 7) ve kuyruğun sonuna gider. Sıradaki: **P1**.

RR Adım Adım: q = 2

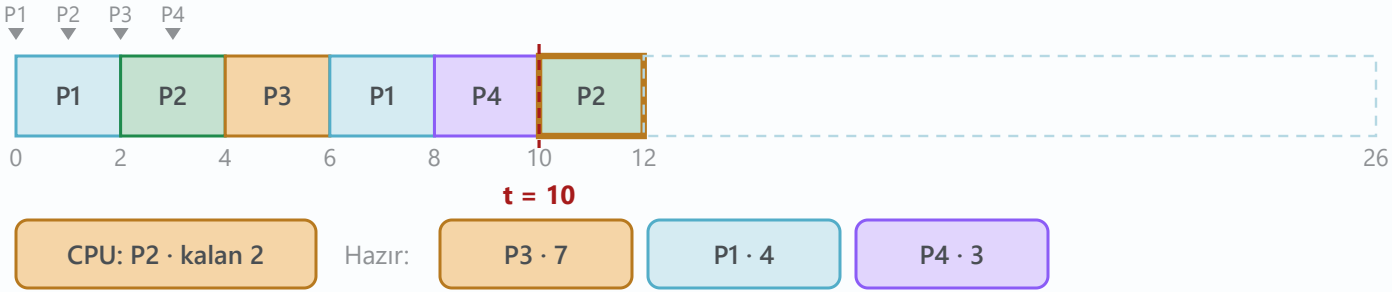
Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 8: P1 dilimini doldurdu (kalan 4) ve kuyruğun sonuna gider. Sıradaki: **P4**.

RR Adım Adım: q = 2

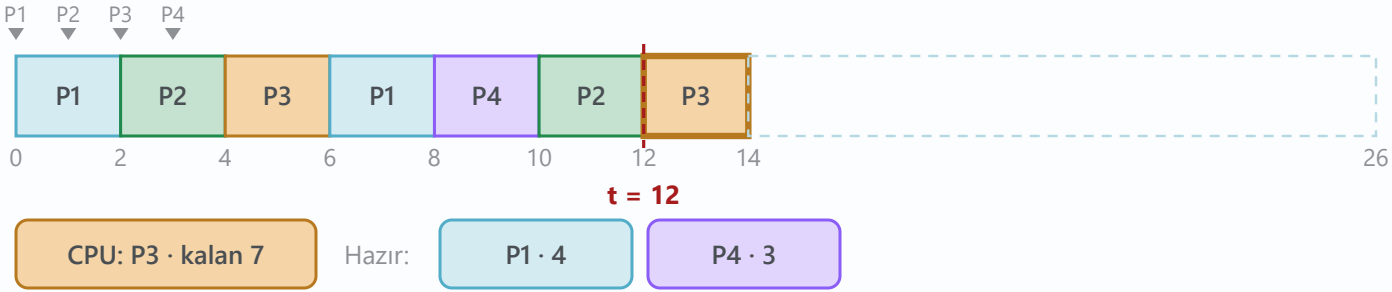
Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 10: P4 dilimini doldurdu (kalan 3) ve kuyruğun sonuna gider. Sıradaki: **P2**.

RR Adım Adım: q = 2

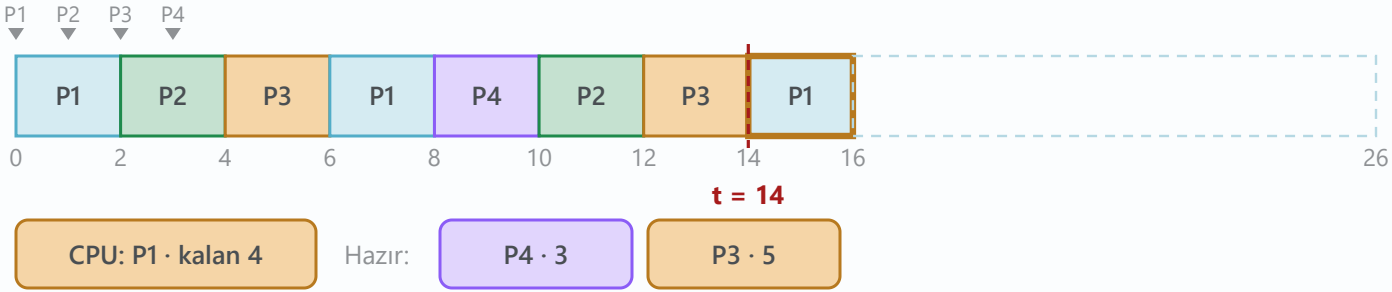
Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 12: P2 işini dilim dolmadan bitirdi. Sıradaki: **P3**.

RR Adım Adım: q = 2

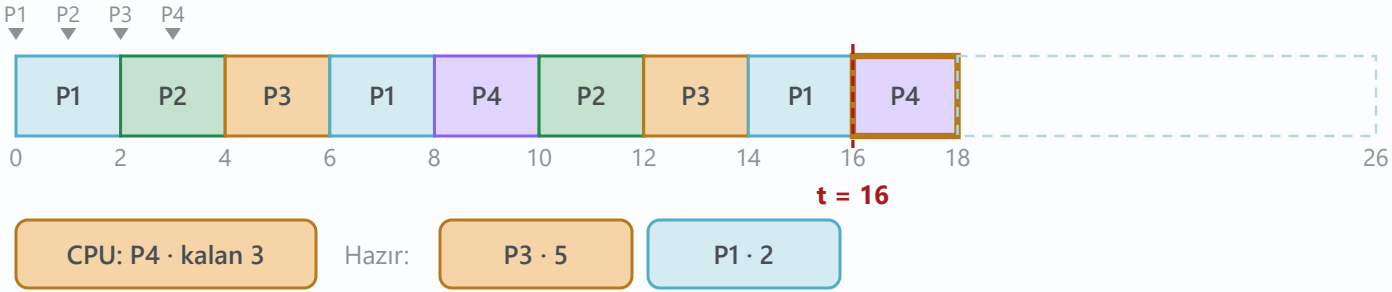
Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 14: P3 dilimini doldurdu (kalan 5) ve kuyruğun sonuna gider. Sıradaki: **P1**.

RR Adım Adım: q = 2

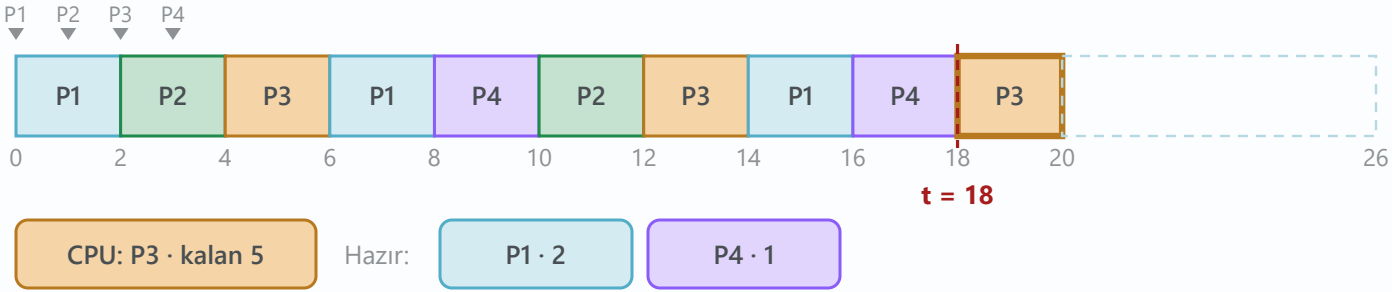
Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 16: P1 dilimini doldurdu (kalan 2) ve kuyruğun sonuna gider. Sıradaki: **P4**.

RR Adım Adım: q = 2

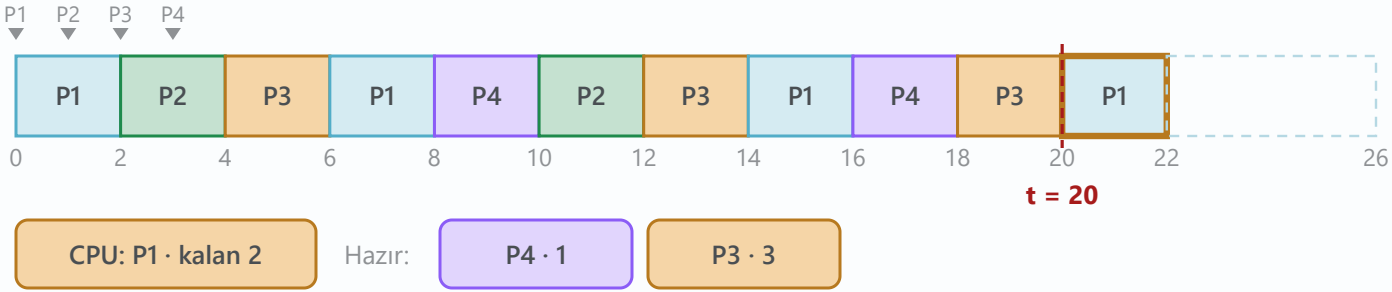
Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 18: P4 dilimini doldurdu (kalan 1) ve kuyruğun sonuna gider. Sıradaki: **P3**.

RR Adım Adım: q = 2

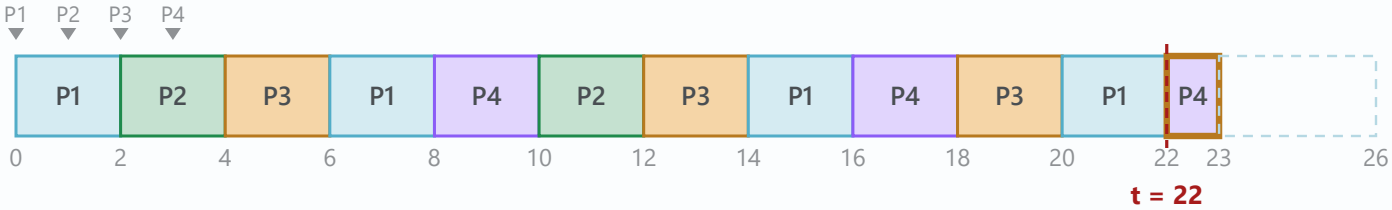
Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



t = 20: P3 dilimini doldurdu (kalan 3) ve kuyruğun sonuna gider. Sıradaki: **P1**.

RR Adım Adım: q = 2

Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.

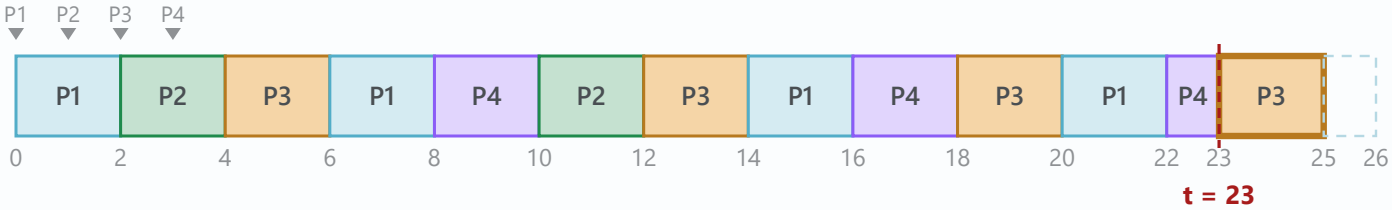


CPU: P4 · kalan 1 | Hazır: P3 · 3

t = 22: P1 işini dilim dolmadan bitirdi. Sıradaki: **P4**.

RR Adım Adım: q = 2

Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



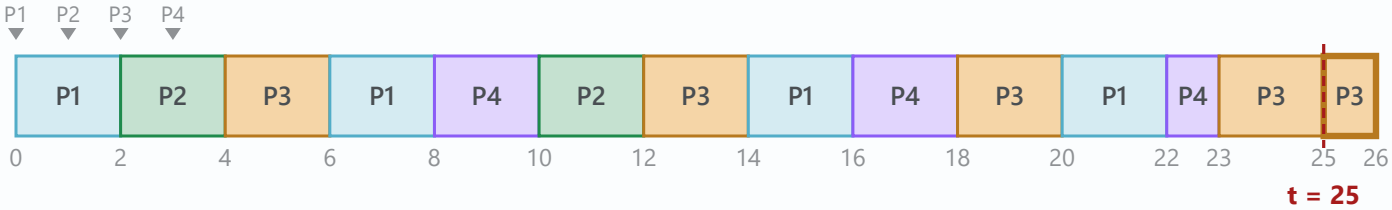
CPU: P3 · kalan 3

Hazır: (boş)

t = 23: P4 işini dilim dolmadan bitirdi. Sıradaki: **P3**.

RR Adım Adım: q = 2

Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



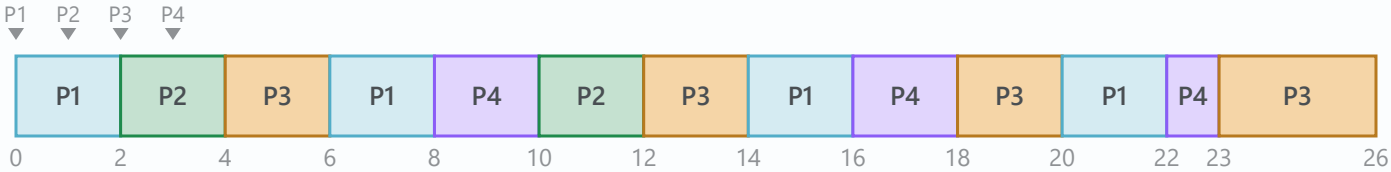
CPU: P3 · kalan 1

Hazır: (boş)

t = 25: P3 dilimini doldurdu (kalan 1) ama kuyrukta başka süreç yok → yeni dilim alır, bağlam değiştirme olmaz.

RR Adım Adım: q = 2

Ortak küme, **q = 2 ms**. Küçük dilim: süreçler sık sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.



Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	0	22	22	14	0
P2	1	4	2	12	11	7	1
P3	2	9	4	26	24	15	2
P4	3	5	8	23	20	15	5
Ort.					**19,25**	**12,75**	**2**

Ortalama bekleme **12,75**, dönüş **19,25**, tepki **2** ms · 12 bağlam değiştirme.

RR Adım Adım: q = 5

Ortak küme, **q = 5 ms**. Daha büyük dilim: geçiş sayısı azalır, ama tepki süresi uzar.



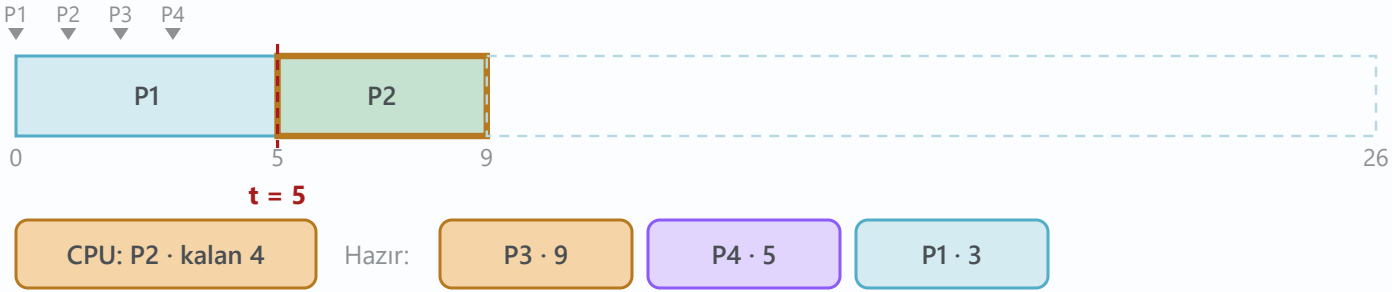
t = 0

CPU: P1 · kalan 8 Hazır: (boş)

t = 0: yalnızca **P1** var; en fazla q = 5 ms çalışır, sonra sırayı bırakır.

RR Adım Adım: q = 5

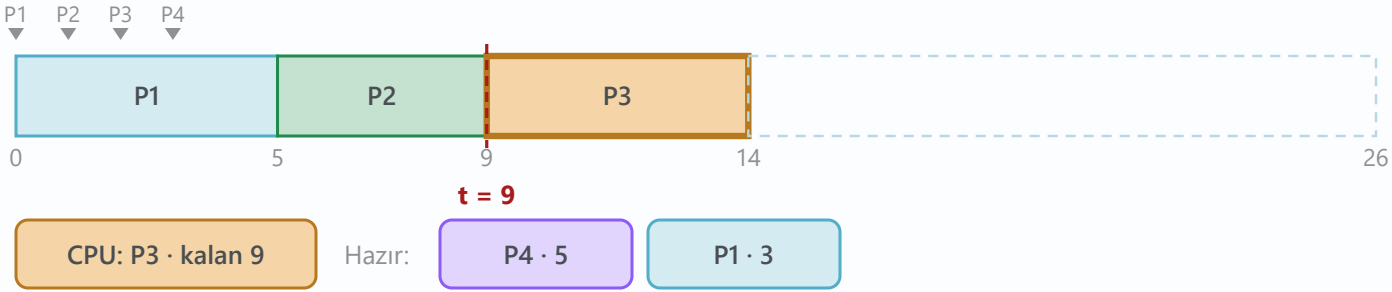
Ortak küme, **q = 5 ms**. Daha büyük dilim: geçiş sayısı azalır, ama tepki süresi uzar.



t = 5: P1 dilimini doldurdu (kalan 3) ve kuyruğun sonuna gider. Arada gelen P2, P3, P4 kuyruğa ondan önce girdi. Sıradaki: **P2**.

RR Adım Adım: q = 5

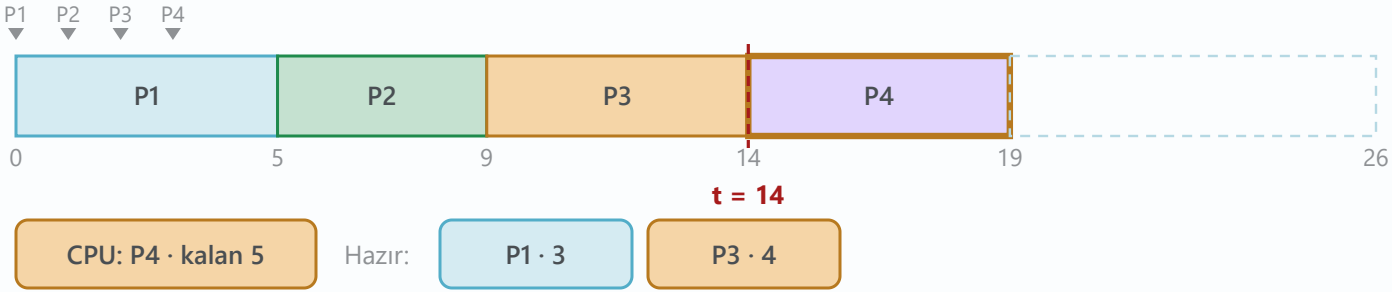
Ortak küme, **q = 5 ms**. Daha büyük dilim: geçiş sayısı azalır, ama tepki süresi uzar.



t = 9: P2 işini dilim dolmadan bitirdi. Sıradaki: **P3**.

RR Adım Adım: q = 5

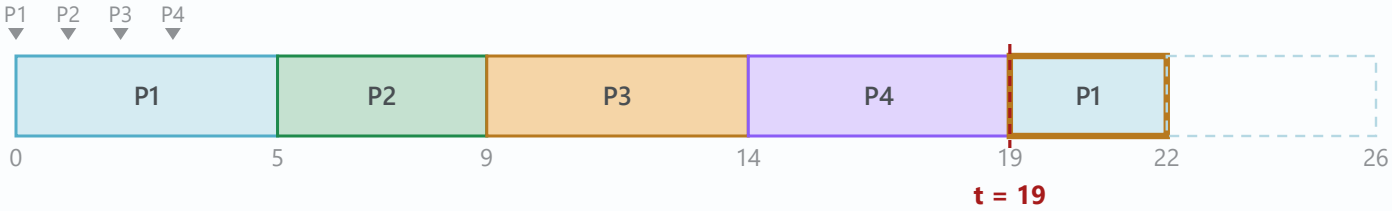
Ortak küme, **q = 5 ms**. Daha büyük dilim: geçiş sayısı azalır, ama tepki süresi uzar.



t = 14: P3 dilimini doldurdu (kalan 4) ve kuyruğun sonuna gider. Sıradaki: **P4**.

RR Adım Adım: q = 5

Ortak küme, **q = 5 ms**. Daha büyük dilim: geçiş sayısı azalır, ama tepki süresi uzar.

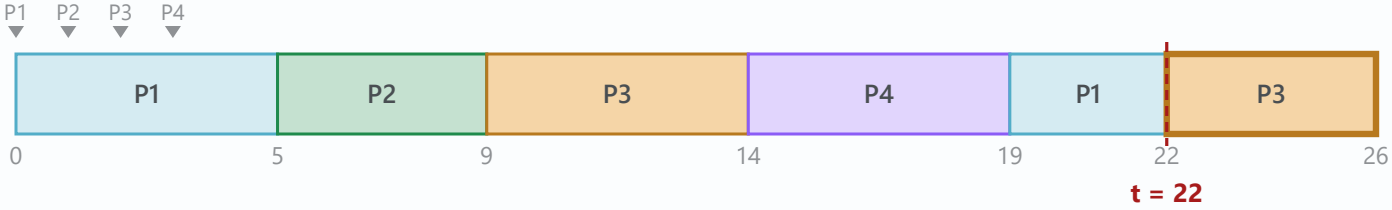


CPU: P1 · kalan 3 | Hazır: P3 · 4

t = 19: P4 işini dilim dolmadan bitirdi. Sıradaki: **P1**.

RR Adım Adım: q = 5

Ortak küme, **q = 5 ms**. Daha büyük dilim: geçiş sayısı azalır, ama tepki süresi uzar.



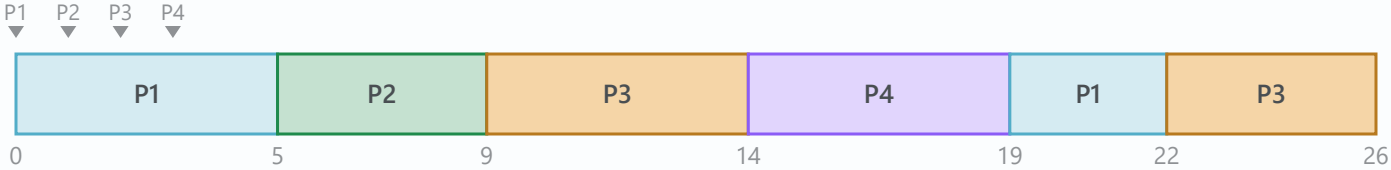
CPU: P3 · kalan 4

Hazır: (boş)

t = 22: P1 işini dilim dolmadan bitirdi. Sıradaki: **P3**.

RR Adım Adım: q = 5

Ortak küme, **q = 5 ms**. Daha büyük dilim: geçiş sayısı azalır, ama tepki süresi uzar.



Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	0	22	22	14	0
P2	1	4	5	9	8	4	4
P3	2	9	9	26	24	15	7
P4	3	5	14	19	16	11	11
Ort.					**17,5**	**11**	**5,5**

Ortalama bekleme **11**, dönüş **17,5**, tepki **5,5** ms · 5 bağlam değiştirme.

Round Robin: Kod İzleme

```
1  /* p[] geliş sırasına göre sıralı */
2  void rr(Surec p[], int n, int q) {
▶    int t = 0, k = 0;      /* k: sıradaki */
▶    gelenleri_ekle(p, n, &k, t);
5    while (!bos() || k < n) {
6      if (bos()) {          /* CPU boşta */
7        t = p[k].gelis;     /* sonraki geliş */
8        gelenleri_ekle(p, n, &k, t);
9      }
10     Surec *s = cikar();  /* baştaki */
11     int dilim = min(q, s->kalan);
12     t += dilim; s->kalan -= dilim;
13     gelenleri_ekle(p, n, &k, t); /* önce */
14     if (s->kalan > 0) ekle(s);   /* sonra */
15     else printf("%s %d\n", s->ad, t);
16   }
17 }
```



CPU: boşta Hazır: P1·8

t = 0: yalnızca P1 gelmiştir; kuyruğa girer. Dilim `q = 4` ms.

Adım 1 / 10

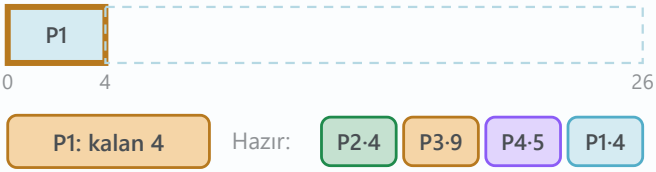
DEĞİŞKENLER

`q = 4` `t = 0` `kuyruk = [P1]`

ÇIKTI

Round Robin: Kod İzleme

```
1  /* p[] geliş sırasına göre sıralı */
2  void rr(Surec p[], int n, int q) {
3    int t = 0, k = 0;      /* k: sıradaki */
4    gelenleri_ekle(p, n, &k, t);
5    while (!bos() || k < n) {
6      if (bos()) {          /* CPU boşta */
7        t = p[k].gelis;     /* sonraki geliş */
8        gelenleri_ekle(p, n, &k, t);
9      }
10     Surec *s = cikar();  /* baştaki */
▶      int dilim = min(q, s->kalan);
▶      t += dilim; s->kalan -= dilim;
13     gelenleri_ekle(p, n, &k, t); /* önce */
▶      if (s->kalan > 0) ekle(s);   /* sonra */
15     else printf("%s %d\n", s->ad, t);
16   }
17 }
```



t = 0–4: P1 kuyruktan çıkar, 4 ms çalışır (kalan 4). Bu sırada P2, P3, P4 geldi; önce onlar, sonra P1 kuyruğa girer.

Adım 2 / 10

DEĞİŞKENLER

`t = 4` `s = P1` `dilim = 4` `s->kalan = 4`

`kuyruk = [P2, P3, P4, P1]`

ÇIKTI

Round Robin: Kod İzleme

```
1  /* p[] geliş sırasına göre sıralı */
2  void rr(Surec p[], int n, int q) {
3    int t = 0, k = 0;      /* k: sıradaki */
4    gelenleri_ekle(p, n, &k, t);
5    while (!bos() || k < n) {
6      if (bos()) {          /* CPU boşta */
7        t = p[k].gelis;     /* sonraki geliş */
8        gelenleri_ekle(p, n, &k, t);
9      }
10     Surec *s = cikar();  /* baştaki */
▶      int dilim = min(q, s->kalan);
▶      t += dilim; s->kalan -= dilim;
13     gelenleri_ekle(p, n, &k, t); /* önce */
14     if (s->kalan > 0) ekle(s);   /* sonra */
▶      else printf("%s %d\n", s->ad, t);
16   }
17 }
```



t = 8: P2 son 4 ms'sini çalıştı ve bitti; çıktıya bitiş anı yazılır.

Adım 3 / 10

DEĞİŞKENLER

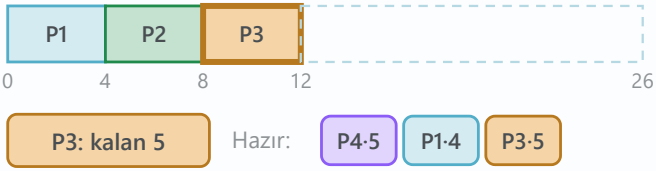
`t = 8` `s = P2` `dilim = 4` `s->kalan = 0` `kuyruk = [P3, P4, P1]`

ÇIKTI

```
P2 8
```

Round Robin: Kod İzleme

```
1  /* p[] geliş sırasına göre sıralı */
2  void rr(Surec p[], int n, int q) {
3    int t = 0, k = 0;      /* k: sıradaki */
4    gelenleri_ekle(p, n, &k, t);
5    while (!bos() || k < n) {
6      if (bos()) {          /* CPU boşta */
7        t = p[k].gelis;     /* sonraki geliş */
8        gelenleri_ekle(p, n, &k, t);
9      }
10     Surec *s = cikar();  /* baştaki */
▶      int dilim = min(q, s->kalan);
▶      t += dilim; s->kalan -= dilim;
13     gelenleri_ekle(p, n, &k, t); /* önce */
▶      if (s->kalan > 0) ekle(s);   /* sonra */
15     else printf("%s %d\n", s->ad, t);
16   }
17 }
```



t = 12: P3 dilimini doldurdu (kalan 5) ve kuyruğun sonuna döner.

Adım 4 / 10

DEĞİŞKENLER

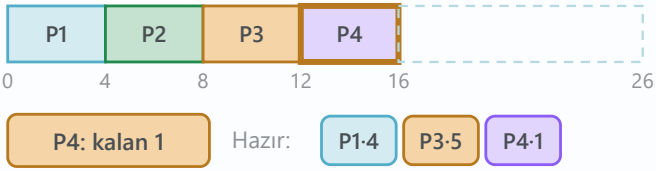
`t = 12` `s = P3` `dilim = 4` `s->kalan = 5` `kuyruk = [P4, P1, P3]`

ÇIKTI

```
P2 8
```

Round Robin: Kod İzleme

```
1  /* p[] geliş sırasına göre sıralı */
2  void rr(Surec p[], int n, int q) {
3    int t = 0, k = 0;      /* k: sıradaki */
4    gelenleri_ekle(p, n, &k, t);
5    while (!bos() || k < n) {
6      if (bos()) {          /* CPU boşta */
7        t = p[k].gelis;     /* sonraki geliş */
8        gelenleri_ekle(p, n, &k, t);
9      }
10     Surec *s = cikar();  /* baştaki */
▶      int dilim = min(q, s->kalan);
▶      t += dilim; s->kalan -= dilim;
13     gelenleri_ekle(p, n, &k, t); /* önce */
▶      if (s->kalan > 0) ekle(s);   /* sonra */
15     else printf("%s %d\n", s->ad, t);
16   }
17 }
```



t = 16: P4 dilimini doldurdu (kalan 1) ve kuyruğun sonuna döner.

Adım 5 / 10

DEĞİŞKENLER

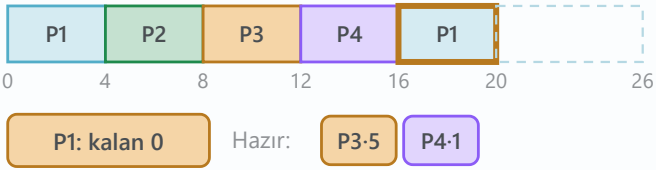
`t = 16` `s = P4` `dilim = 4` `s->kalan = 1` `kuyruk = [P1, P3, P4]`

ÇIKTI

```
P2 8
```

Round Robin: Kod İzleme

```
1  /* p[] geliş sırasına göre sıralı */
2  void rr(Surec p[], int n, int q) {
3    int t = 0, k = 0;      /* k: sıradaki */
4    gelenleri_ekle(p, n, &k, t);
5    while (!bos() || k < n) {
6      if (bos()) {          /* CPU boşta */
7        t = p[k].gelis;     /* sonraki geliş */
8        gelenleri_ekle(p, n, &k, t);
9      }
10     Surec *s = cikar();  /* baştaki */
▶      int dilim = min(q, s->kalan);
▶      t += dilim; s->kalan -= dilim;
13     gelenleri_ekle(p, n, &k, t); /* önce */
14     if (s->kalan > 0) ekle(s);   /* sonra */
▶      else printf("%s %d\n", s->ad, t);
16   }
17 }
```



t = 20: P1 son 4 ms'sini çalıştı ve bitti; çıktıya bitiş anı yazılır.

Adım 6 / 10

DEĞİŞKENLER

`t = 20` `s = P1` `dilim = 4` `s->kalan = 0` `kuyruk = [P3, P4]`

ÇIKTI

```
P2 8
P1 20
```

Round Robin: Kod İzleme

```
1  /* p[] geliş sırasına göre sıralı */
2  void rr(Surec p[], int n, int q) {
3    int t = 0, k = 0;      /* k: sıradaki */
4    gelenleri_ekle(p, n, &k, t);
5    while (!bos() || k < n) {
6      if (bos()) {          /* CPU boşta */
7        t = p[k].gelis;     /* sonraki geliş */
8        gelenleri_ekle(p, n, &k, t);
9      }
10     Surec *s = cikar();  /* baştaki */
▶      int dilim = min(q, s->kalan);
▶      t += dilim; s->kalan -= dilim;
13     gelenleri_ekle(p, n, &k, t); /* önce */
▶      if (s->kalan > 0) ekle(s);   /* sonra */
15     else printf("%s %d\n", s->ad, t);
16   }
17 }
```



t = 24: P3 dilimini doldurdu (kalan 1) ve kuyruğun sonuna döner.

Adım 7 / 10

DEĞİŞKENLER

`t = 24` `s = P3` `dilim = 4`

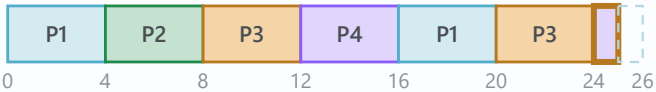
`s->kalan = 1` `kuyruk = [P4, P3]`

ÇIKTI

```
P2 8
P1 20
```

Round Robin: Kod İzleme

```
1  /* p[] geliş sırasına göre sıralı */
2  void rr(Surec p[], int n, int q) {
3    int t = 0, k = 0;      /* k: sıradaki */
4    gelenleri_ekle(p, n, &k, t);
5    while (!bos() || k < n) {
6      if (bos()) {          /* CPU boşta */
7        t = p[k].gelis;     /* sonraki geliş */
8        gelenleri_ekle(p, n, &k, t);
9      }
10     Surec *s = cikar();  /* baştaki */
▶      int dilim = min(q, s->kalan);
▶      t += dilim; s->kalan -= dilim;
13     gelenleri_ekle(p, n, &k, t); /* önce */
14     if (s->kalan > 0) ekle(s);   /* sonra */
▶      else printf("%s %d\n", s->ad, t);
16   }
17 }
```



P4: kalan 0 Hazır: P3·1

t = 25: P4 kalan 1 ms'yi dilim dolmadan bitirdi (`dilim = 1`); kuyruğa dönmez.

Adım 8 / 10

DEĞİŞKENLER

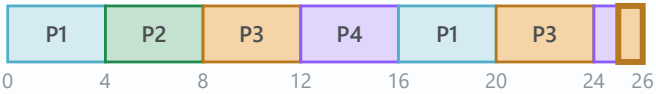
`t = 25` `s = P4` `dilim = 1` `s->kalan = 0` `kuyruk = [P3]`

ÇIKTI

```
P2 8
P1 20
P4 25
```

Round Robin: Kod İzleme

```
1  /* p[] geliş sırasına göre sıralı */
2  void rr(Surec p[], int n, int q) {
3    int t = 0, k = 0;      /* k: sıradaki */
4    gelenleri_ekle(p, n, &k, t);
5    while (!bos() || k < n) {
6      if (bos()) {          /* CPU boşta */
7        t = p[k].gelis;     /* sonraki geliş */
8        gelenleri_ekle(p, n, &k, t);
9      }
10     Surec *s = cikar();  /* baştaki */
▶      int dilim = min(q, s->kalan);
▶      t += dilim; s->kalan -= dilim;
13     gelenleri_ekle(p, n, &k, t); /* önce */
14     if (s->kalan > 0) ekle(s);   /* sonra */
▶      else printf("%s %d\n", s->ad, t);
16   }
17 }
```



P3: kalan 0 Hazır: (boş)

t = 26: P3 kalan 1 ms'yi dilim dolmadan bitirdi (`dilim = 1`); kuyruğa dönmez.

Adım 9 / 10

DEĞİŞKENLER

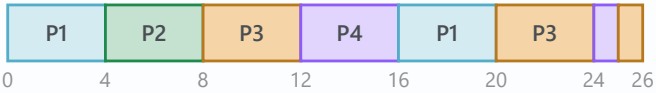
`t = 26` `s = P3` `dilim = 1` `s->kalan = 0` `kuyruk = []`

ÇIKTI

```
P2 8
P1 20
P4 25
P3 26
```

Round Robin: Kod İzleme

```
1  /* p[] geliş sırasına göre sıralı */
2  void rr(Surec p[], int n, int q) {
3    int t = 0, k = 0;      /* k: sıradaki */
4    gelenleri_ekle(p, n, &k, t);
▶    while (!bos() || k < n) {
6      if (bos()) {          /* CPU boşta */
7        t = p[k].gelis;     /* sonraki geliş */
8        gelenleri_ekle(p, n, &k, t);
9      }
10     Surec *s = cikar();  /* baştaki */
11     int dilim = min(q, s->kalan);
12     t += dilim; s->kalan -= dilim;
13     gelenleri_ekle(p, n, &k, t); /* önce */
14     if (s->kalan > 0) ekle(s);   /* sonra */
15     else printf("%s %d\n", s->ad, t);
16   }
17 }
```



Bekleme: P1 12 · P2 3 · P3 15 · P4 17

ortalama bekleme 11,75 · dönüş 18,25 · tepki 4,5 ms

Kuyruk boş ve gelecek süreç yok (`k = n`), döngü biter: 8 dağıtım, **7 bağlam değiştirme**. Yeni gelenlerin kesilen süreçten önce eklenmesi 13. ve 14. satırların sırasından gelir. 6–9. satırlar bu örnekte hiç çalışmaz.

Adım 10 / 10

DEĞİŞKENLER

`t = 26` `kuyruk = []`

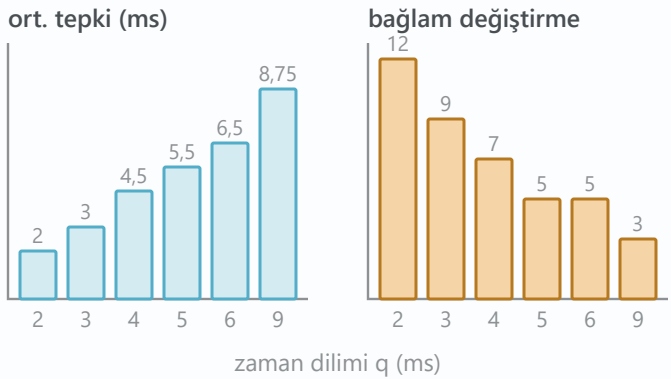
ÇIKTI

```
P2 8
P1 20
P4 25
P3 26
```

Zaman Dilimi Seçimi: Ödünleşim

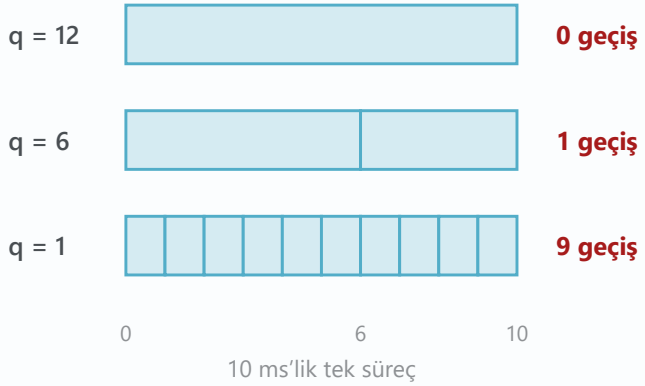
Zaman dilimi	Bekleme	Dönüş	Tepki	Geçiş
q = 2	12,75	19,25	2	12
q = 3	13,5	20	3	9
q = 4	11,75	18,25	4,5	7
q = 5	11	17,5	5,5	5
q = 6	12	18,5	6,5	5
q ≥ 9 (FCFS)	8,75	15,25	8,75	3

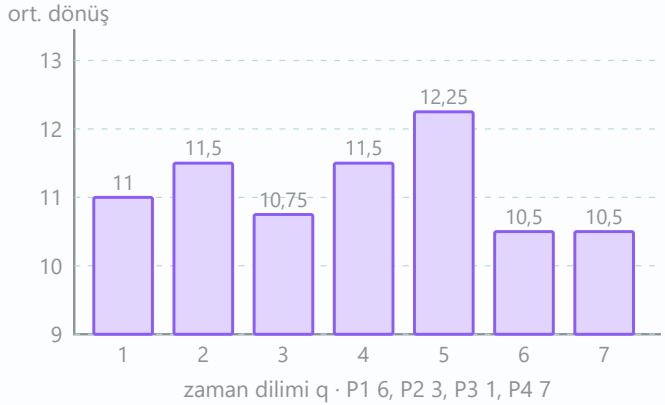
Ortalamalar ms cinsinden; geçiş = bağlam değiştirme sayısı.



- q küçüldükçe **tepki** iyileşir (q = 2'de 2 ms) ama **bağlam değiştirme** artar (12 geçiş); her geçiş gerçek sistemde CPU zamanı yer.
- Ortalama bekleme ve dönüş, q ile **tekdüze değişmez** (q = 3, q = 2'den kötü). q ≥ 9 (en uzun patlama) olunca RR tam olarak FCFS olur.

Zaman Dilimi ve Bağlam Değiştirme





- Kayıp oranı = s / (q + s): q = 4 ms, geçiş s = 1 ms ise CPU'nun **%20**'si boşa gider; q = 100 ms'de ≈ %1 ama sistem ağır yanıt verir.
- Bugün geçiş birkaç **µs**, tipik dilim 1–100 ms (Linux EEVDF temel dilimi 0,75 ms × (1 + $\log_2$ CPU), 8+ çekirdekte 3 ms; Windows istemcide ≈ 2 saat tıkı). Kural: CPU patlamalarının ≈ %80'i q'dan kısa olmalı.
- Dönüş süresi q ile tekdüze iyileşmez (sağda). q = 4 ile P1 24, P2 3, P3 3 kümesinde ortalama bekleme 17 / 3 ≈ 5,67 ms (FCFS'de 17).

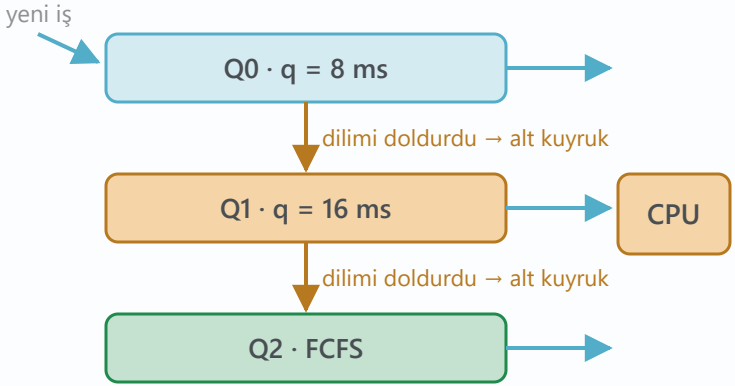
Çok Düzeyli Kuyruk (MLQ)

- Hazır kuyruğu **sınıflara** bölünür; her süreç türüne göre (ör. ön plan / arka plan) **kalıcı olarak** bir kuyruğa atanır.
- Her kuyruğun kendi algoritması vardır: ön plan için RR (kısa q), arka plan için FCFS.
- Kuyruklar arası çizelgeleme: **sabit öncelik** (üst kuyruk boşalmadan alt kuyruk çalışmaz → açlık) ya da **zaman paylaşımı** (ör. CPU'nun %80'i ön plana, %20'si arka plana).
- Yükü düşüktür ama esnek değildir: süreç kuyruk değiştiremez.



Sabit atamalı yapı MLQ'dur; süreçlerin davranışına göre kuyruk değiştirdiği yapı **geri beslemeli** MLFQ'dur.

Çok Düzeyli Geri Beslemeli Kuyruk (MLFQ)



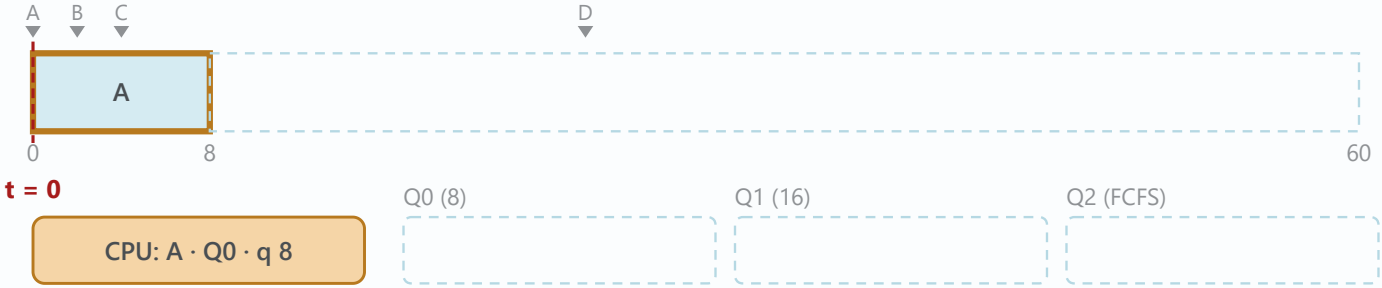
Q0 boşsa Q1, Q0 ve Q1 boşsa Q2 çalışır

1. Öncelik(A) > Öncelik(B) → A çalışır; eşitse RR.
2. Yeni iş **en üst** kuyruğa girer (önce kısa olduğu varsayılır).
3. Bir düzeydeki **hakkını** (allotment) bitiren iş, kaç kez CPU'yu bırakmış olursa olsun bir alt düzeye iner.
4. Her S sürede bir tüm işler en üst kuyruğa taşınır (**öncelik yükseltme**): açlığı önler, davranışı değişen işe yeni şans verir.

Parametreler: kuyruk sayısı, her kuyruğun algoritması ve dilimi, alt/üst kuyruğa geçiş kuralı, yeni işin giriş kuyruğu. Kurallar OSTEP (Bölüm 8) düzenlemesidir; kısa işler üstte biter, uzun CPU işleri aşağı süzülür.

MLFQ Adım Adım

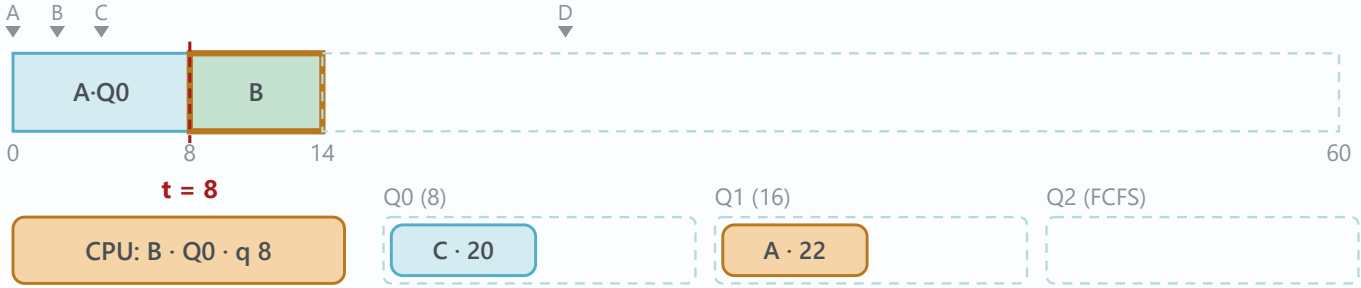
Üç kuyruk: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). İşler: A (0, 30), B (2, 6), C (4, 20), D (25, 4). Kesilen iş kuyruğunun başına döner ve dilimini tamamlar.



t = 0: A gelir ve Q0'a girer; 8 ms'lik dilimiyle çalışır. B (t = 2) ve C (t = 4) Q0'a eklenir.

MLFQ Adım Adım

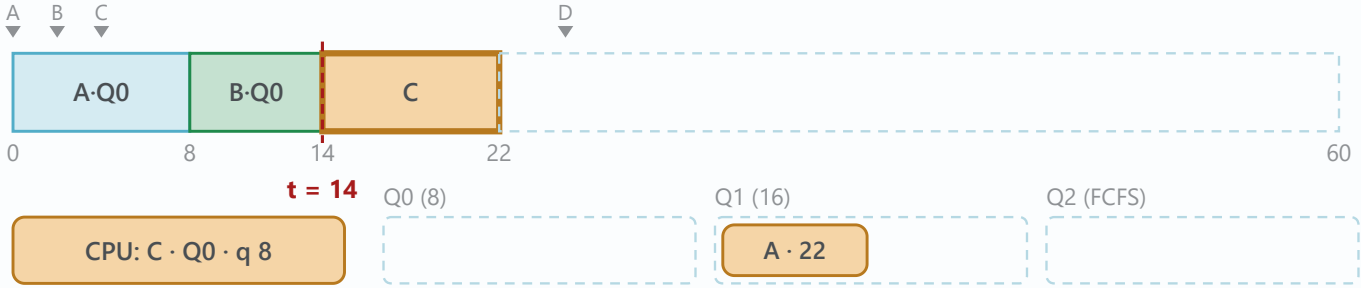
Üç kuyruk: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). İşler: A (0, 30), B (2, 6), C (4, 20), D (25, 4). Kesilen iş kuyruğunun başına döner ve dilimini tamamlar.



t = 8: A dilimini doldurdu → **Q1**'e iner. Q0'da sıradaki B çalışır.

MLFQ Adım Adım

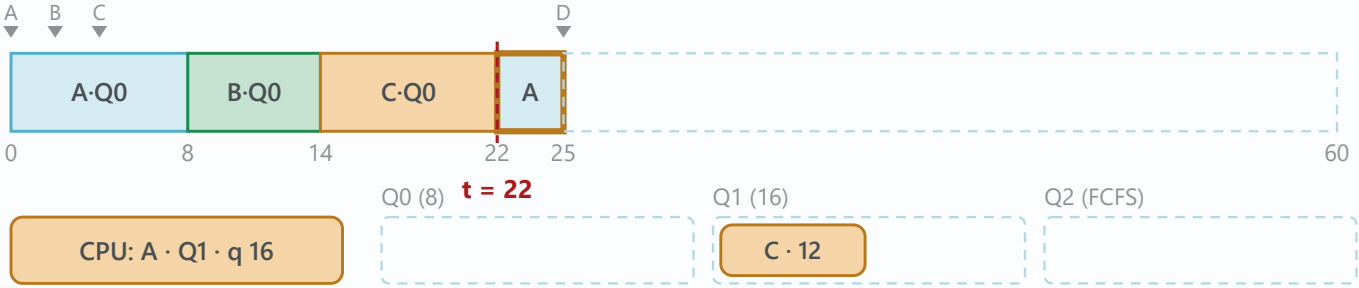
Üç kuyruk: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). İşler: A (0, 30), B (2, 6), C (4, 20), D (25, 4). Kesilen iş kuyruğunun başına döner ve dilimini tamamlar.



t = 14: B 6 ms'de bitti (dilim dolmadan, Q0'da). Şimdi C, Q0'da 8 ms çalışır.

MLFQ Adım Adım

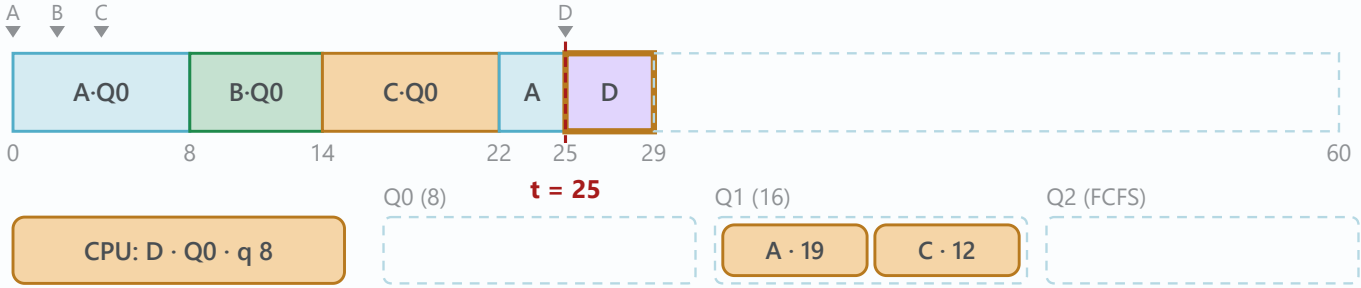
Üç kuyruk: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). İşler: A (0, 30), B (2, 6), C (4, 20), D (25, 4). Kesilen iş kuyruğunun başına döner ve dilimini tamamlar.



t = 22: C de Q1'e iner. Q0 boş → Q1'in başındaki A 16 ms'lik dilimle çalışır.

MLFQ Adım Adım

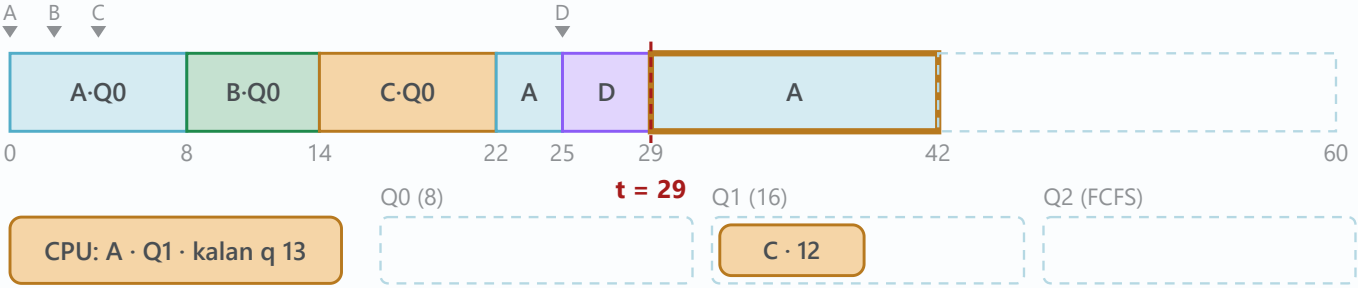
Üç kuyruk: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). İşler: A (0, 30), B (2, 6), C (4, 20), D (25, 4). Kesilen iş kuyruğunun başına döner ve dilimini tamamlar.



t = 25: D gelir ve Q0'a girer: üst kuyruk olduğu için **A'yı keser**. A, Q1'in başına döner (13 ms dilimi kaldı).

MLFQ Adım Adım

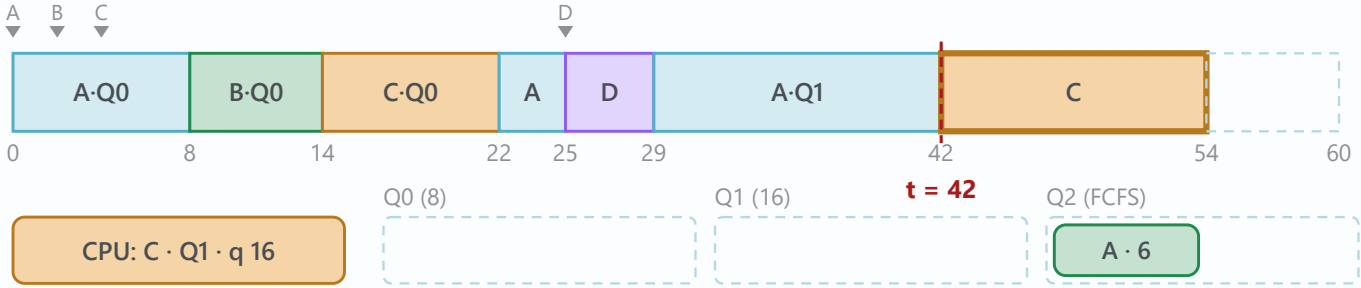
Üç kuyruk: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). İşler: A (0, 30), B (2, 6), C (4, 20), D (25, 4). Kesilen iş kuyruğunun başına döner ve dilimini tamamlar.



t = 29: D 4 ms'de bitti (tepki 0, bekleme 0). A, Q1'de kalan 13 ms'lik dilimini kullanır.

MLFQ Adım Adım

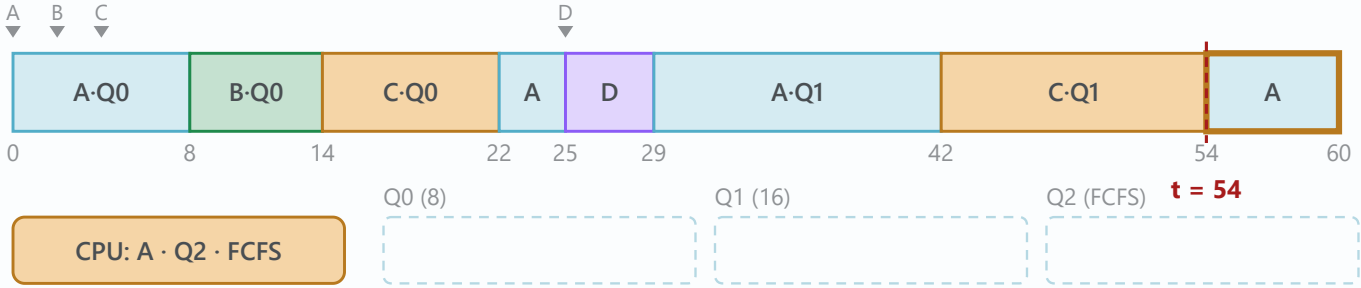
Üç kuyruk: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). İşler: A (0, 30), B (2, 6), C (4, 20), D (25, 4). Kesilen iş kuyruğunun başına döner ve dilimini tamamlar.



t = 42: A Q1 dilimini de doldurdu → **Q2**. C, Q1'de çalışır; 12 ms yeterli.

MLFQ Adım Adım

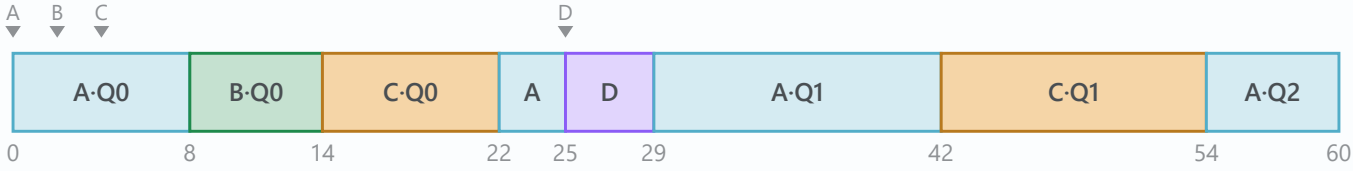
Üç kuyruk: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). İşler: A (0, 30), B (2, 6), C (4, 20), D (25, 4). Kesilen iş kuyruğunun başına döner ve dilimini tamamlar.



t = 54: Q0 ve Q1 boş; A Q2'de FCFS ile kalan 6 ms'yi çalışır ve t = 60'ta biter.

MLFQ Adım Adım

Üç kuyruk: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). İşler: A (0, 30), B (2, 6), C (4, 20), D (25, 4). Kesilen iş kuyruğunun başına döner ve dilimini tamamlar.



İş	Geliş	CPU	Bitiş	Dönüş	Bekleme	Tepki
A	0	30	60	60	30	0
B	2	6	14	12	6	6
C	4	20	54	50	30	10
D	25	4	29	4	0	0
Ort.				**31,5**	**16,5**	**4**

Kısa B ve D üst kuyrukta hızla biter (tepki 6 ve 0); uzun A aşağı süzülür. Ortalama tepki **4 ms**.

MLFQ Uygulamada

Oyun oynama (gaming)

Dilim bitmeden hemen önce G/Ç yapan bir iş, “bloke olan üstte kalır” kuralıyla CPU’yu tekeline alabilir. Çözüm: düzeydeki toplam kullanımı saymak (kural 3).

Açlık ve davranış değişimi

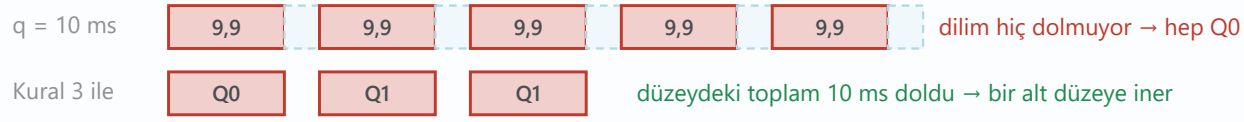
Çok sayıda etkileşimli iş uzun işleri aç bırakır; uzun işin sonradan etkileşimli olması da fark edilmez. Çözüm: periyodik öncelik yükseltme (kural 4).

Tarihçe: CTSS (1962)

Öncelik sınıfları 1, 2, 4, 8 dilim çalışır; dilimini bitiren iş bir alt sınıfa iner. Terminalde Enter’a basılınca süreç en üst sınıfa taşınıyordu; kullanıcılar bunu fark edip Enter’a basarak işlerini hızlandırdı.

Bugün

Solaris TS sınıfı (tabloyla ayarlanan dilim ve öncelik), FreeBSD ULE, Windows’un dinamik öncelikleri ve macOS MLFQ fikrini kullanır. Linux 2.6.23’ten beri öncelik sezgileri yerine adil paylaşım (CFS, sonra EEVDF) kullanır.



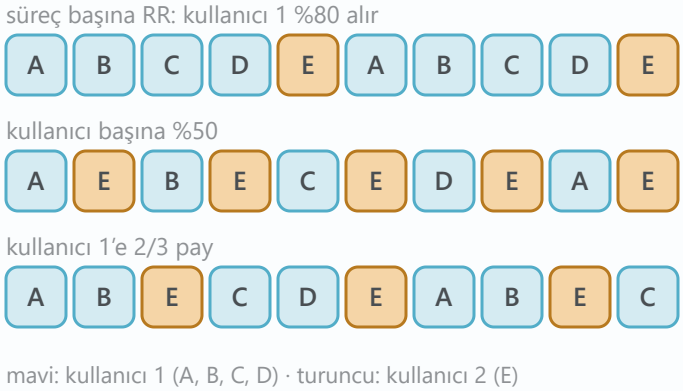
ORANTILI PAY

Garantili ve Adil Paylaşım

Garantili çizelgeleme: n kullanıcı (ya da süreç) varsa her birine CPU'nun 1/n'i söz verilir. Her biri için **oran = aldığı CPU / hak ettiği CPU** hesaplanır; en düşük oranlı çalışır. Oranı 0,5 olan (payının yarısını almış) öne geçer, 2,0 olan bekler.

Adil paylaşım (fair-share): pay süreçlere değil **kullanıcılara ya da gruplara** verilir; çok süreç açan kullanıcı daha fazla CPU alamaz.

Bugün: Linux **cgroup v2** `cpu.weight` ile grup çizelgeleme; systemd her kullanıcıyı bir `user-UID.slice` içine koyar, Kubernetes CPU istekleri de ağırlığa çevrilir.

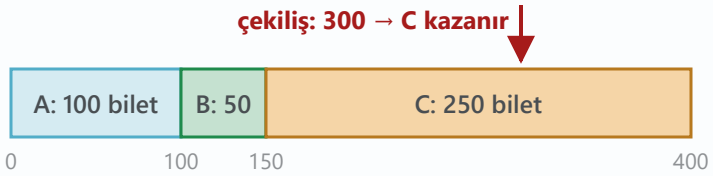


mavi: kullanıcı 1 (A, B, C, D) · turuncu: kullanıcı 2 (E)

Tanenbaum'daki anlamıyla garantili çizelgeleme yukarıdaki pay garantisidir; önceden belirlenmiş bir programa göre çalışma değildir.

ORANTILI PAY

Piyango Çizelgeleme



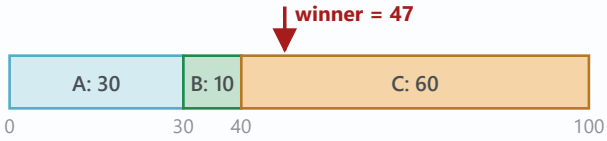
Olasılıklar: A 100/400 = %25 · B %12,5 · C %62,5

- Her sürece **bilet** verilir; her dilimde rastgele bir bilet çekilir, sahibi çalışır. Uzun vadede pay = bilet / toplam bilet.
- **Gerçeklemesi basittir:** rastgele sayı + listede birikimli toplam (Waldspurger ve Weihl, 1994).
- Yeni süreç bilet alır almaz şansa sahiptir: **duyarlıdır**; öncelik "daha çok bilet" demektir.
- **Bilet devri:** sunucuyu bekleyen istemci biletlerini sunucuya verir. **Bilet para birimi** ve güvenilen ortamda **enflasyon**.
- Olasılıksaldır: kısa vadede adaletsiz olabilir (eşit biletli iki iş 10 dilimde 7–3 bölüşebilir). Deterministik eşdeğeri **stride** çizelgelemedir.

Piyango çizelgeleme en basit algoritmalardandır; öncelik bilet sayısıyla ifade edilir.

Piyango Çizelgeleme: Kod İzleme

```
1  /* her dilimde bir çekiliş */
▶  int counter = 0;
▶  int winner = rastgele_al(0, totaltickets);
▶  node_t *current = head;
5  while (current) {
6    counter = counter + current->tickets;
7    if (counter > winner)
8      break;          /* kazanan bulundu */
9    current = current->next;
10 }
11 /* current: CPU'yu alacak süreç */
12 calistir(current);
```



Bilet listesi A 30, B 10, C 60 (toplam 100). Çekilen sayı **47**; `counter` birikimli toplamı tutar.

Adım 1 / 10

DEĞİŞKENLER

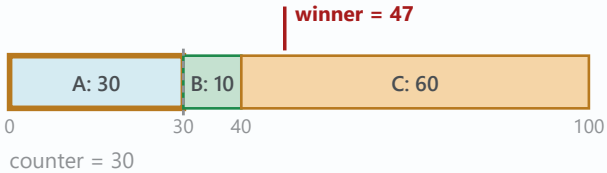
`winner = 47` `counter = 0`

`current = A` `çalışanlar = –`

Kod OSTEP Bölüm 9'daki karar döngüsüdür; çekilen sayılar örnek için sabitlenmiştir.

Piyango Çizelgeleme: Kod İzleme

```
1  /* her dilimde bir çekiliş */
2  int counter = 0;
3  int winner = rastgele_al(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
8      break;           /* kazanan bulundu */
▶    current = current->next;
10 }
11 /* current: CPU'yu alacak süreç */
12 calistir(current);
```



A: counter = 30, 47'den büyük değil → çekilen bilet A'nın aralığında değil, sonraki düğüm.

Adım 2 / 10

DEĞİŞKENLER

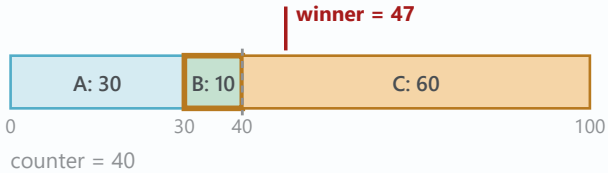
`winner = 47` `counter = 30`

`current = B` `çalışanlar = –`

Kod OSTEP Bölüm 9'daki karar döngüsüdür; çekilen sayılar örnek için sabitlenmiştir.

Piyango Çizelgeleme: Kod İzleme

```
1  /* her dilimde bir çekiliş */
2  int counter = 0;
3  int winner = rastgele_al(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
8      break;           /* kazanan bulundu */
▶    current = current->next;
10 }
11 /* current: CPU'yu alacak süreç */
12 calistir(current);
```



counter = 40

B: counter = 40, 47'den büyük değil → çekilen bilet B'nin aralığında değil, sonraki düğüm.

Adım 3 / 10

DEĞİŞKENLER

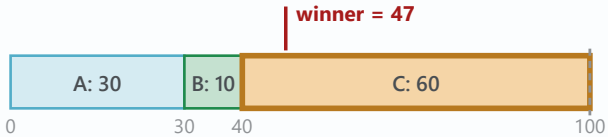
`winner = 47` `counter = 40`

`current = C` `çalışanlar = –`

Kod OSTEP Bölüm 9'daki karar döngüsüdür; çekilen sayılar örnek için sabitlenmiştir.

Piyango Çizelgeleme: Kod İzleme

```
1  /* her dilimde bir çekiliş */
2  int counter = 0;
3  int winner = rastgele_al(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
▶      break;          /* kazanan bulundu */
9    current = current->next;
10 }
11 /* current: CPU'yu alacak süreç */
12 calistir(current);
```



counter = 100 · kazanan: C

C: counter = 100 > 47 → **C kazanır**; döngü kırılır.

Adım 4 / 10

DEĞİŞKENLER

`winner = 47` `counter = 100` `current = C` `çalışanlar = –`

Kod OSTEP Bölüm 9'daki karar döngüsüdür; çekilen sayılar örnek için sabitlenmiştir.

Piyango Çizelgeleme: Kod İzleme

```
1  /* her dilimde bir çekiliş */
▶  int counter = 0;
▶  int winner = rastgele_al(0, totaltickets);
▶  node_t *current = head;
5  while (current) {
6    counter = counter + current->tickets;
7    if (counter > winner)
8      break;          /* kazanan bulundu */
9    current = current->next;
10 }
11 /* current: CPU'yu alacak süreç */
12 calistir(current);
```



Sonraki dilim: yeni çekiliş **35**; sayaç sıfırdan başlar, liste yine baştan taranır.

Adım 5 / 10

DEĞİŞKENLER

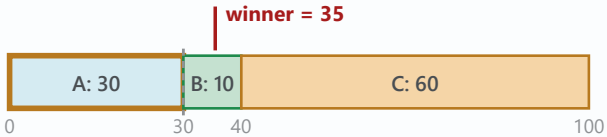
`winner = 35` `counter = 0`

`current = A` `çalışanlar = C`

Kod OSTEP Bölüm 9'daki karar döngüsüdür; çekilen sayılar örnek için sabitlenmiştir.

Piyango Çizelgeleme: Kod İzleme

```
1  /* her dilimde bir çekiliş */
2  int counter = 0;
3  int winner = rastgele_al(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
8      break;          /* kazanan bulundu */
▶    current = current->next;
10 }
11 /* current: CPU'yu alacak süreç */
12 calistir(current);
```



counter = 30

A: counter = 30, 35'ten büyük değil → çekilen bilet A'nın aralığında değil, sonraki düğüm.

Adım 6 / 10

DEĞİŞKENLER

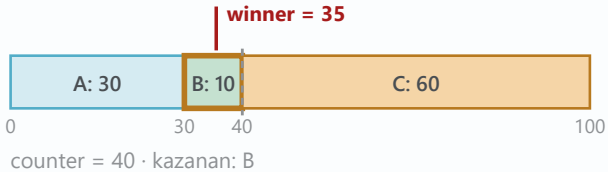
`winner = 35` `counter = 30`

`current = B` `çalışanlar = C`

Kod OSTEP Bölüm 9'daki karar döngüsüdür; çekilen sayılar örnek için sabitlenmiştir.

Piyango Çizelgeleme: Kod İzleme

```
1  /* her dilimde bir çekiliş */
2  int counter = 0;
3  int winner = rastgele_al(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
▶      break;          /* kazanan bulundu */
9    current = current->next;
10 }
11 /* current: CPU'yu alacak süreç */
12 calistir(current);
```



counter = 40 · kazanan: B

B: counter = 40 > 35 → **B kazanır**; döngü kırılır.

Adım 7 / 10

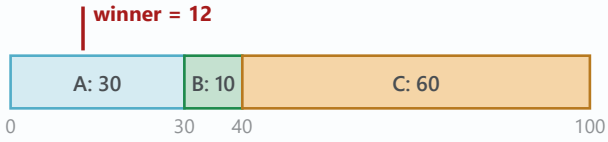
DEĞİŞKENLER

`winner = 35` `counter = 40` `current = B` `çalışanlar = C`

Kod OSTEP Bölüm 9'daki karar döngüsüdür; çekilen sayılar örnek için sabitlenmiştir.

Piyango Çizelgeleme: Kod İzleme

```
1  /* her dilimde bir çekiliş */
▶  int counter = 0;
▶  int winner = rastgele_al(0, totaltickets);
▶  node_t *current = head;
5  while (current) {
6    counter = counter + current->tickets;
7    if (counter > winner)
8      break;          /* kazanan bulundu */
9    current = current->next;
10 }
11 /* current: CPU'yu alacak süreç */
12 calistir(current);
```



Sonraki dilim: yeni çekiliş **12**; sayaç sıfırdan başlar, liste yine baştan taranır.

Adım 8 / 10

DEĞİŞKENLER

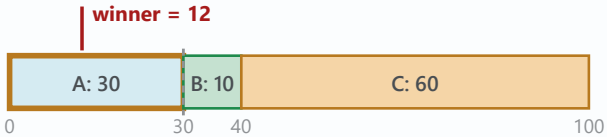
`winner = 12` `counter = 0`

`current = A` `çalışanlar = C B`

Kod OSTEP Bölüm 9'daki karar döngüsüdür; çekilen sayılar örnek için sabitlenmiştir.

Piyango Çizelgeleme: Kod İzleme

```
1  /* her dilimde bir çekiliş */
2  int counter = 0;
3  int winner = rastgele_al(0, totaltickets);
4  node_t *current = head;
5  while (current) {
▶    counter = counter + current->tickets;
▶    if (counter > winner)
▶      break;          /* kazanan bulundu */
9    current = current->next;
10 }
11 /* current: CPU'yu alacak süreç */
12 calistir(current);
```



counter = 30 · kazanan: A

A: counter = 30 > 12 → **A kazanır**; döngü kırılır.

Adım 9 / 10

DEĞİŞKENLER

`winner = 12` `counter = 30`

`current = A` `çalışanlar = C B`

Kod OSTEP Bölüm 9'daki karar döngüsüdür; çekilen sayılar örnek için sabitlenmiştir.

Piyango Çizelgeleme: Kod İzleme

```
1  /* her dilimde bir çekiliş */
2  int counter = 0;
3  int winner = rastgele_al(0, totaltickets);
4  node_t *current = head;
5  while (current) {
6    counter = counter + current->tickets;
7    if (counter > winner)
8      break;          /* kazanan bulundu */
9    current = current->next;
10 }
11 /* current: CPU'yu alacak süreç */
▶  calistir(current);
```

Çekilişler: 47 → C · 35 → B · 12 → A

Beklenen pay: A %30 · B %10 · C %60

Üç dilimde her biri bir kez çalıştı:

kısa vadede pay biletlerle orantılı değil

Üç dilimde A, B ve C birer kez çalıştı; pay ancak **uzun vadede** 30 : 10 : 60'a yaklaşır. Liste bilete göre azalan sıralanırsa tarama kısalır.

Adım 10 / 10

DEĞİŞKENLER

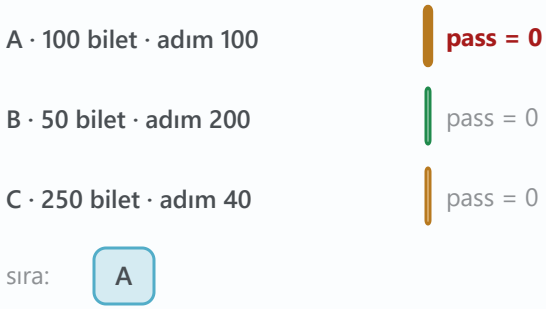
`çalışanlar = C B A`

Kod OSTEP Bölüm 9'daki karar döngüsüdür; çekilen sayılar örnek için sabitlenmiştir.

ORANTILI PAY

Stride Çizelgeleme Adım Adım

Adım = büyük sayı / bilet (10.000 / bilet): A 100, B 200, C 40. Her dilimde en küçük **pass** değerli süreç çalışır, pass değerine adımını ekler; eşitlikte A, B, C sırası.



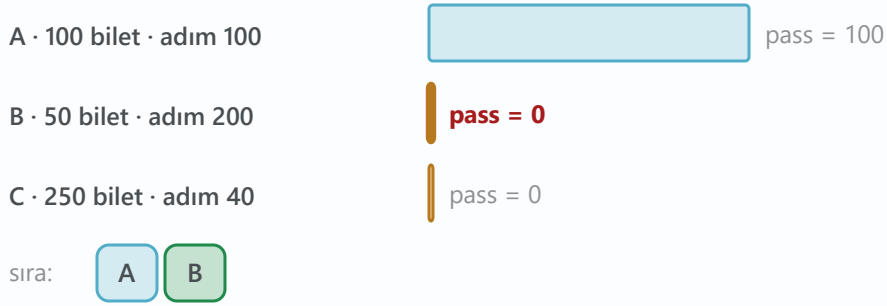
Adım 1: pass değerleri A 0, B 0, C 0 → en küçük **A** çalışır; pass(A) += 100.

Adım 1 / 9

Stride, piyangonun deterministik hali (Waldspurger, 1995). Dezavantajı küresel durum: yeni gelen sürecin pass değeri 0 olursa CPU'yu tekeline alır; bu yüzden mevcut en küçük pass ile başlatılır. Linux CFS'in vruntime fikri buna çok yakındır.

Stride Çizelgeleme Adım Adım

Adım = büyük sayı / bilet (10.000 / bilet): A 100, B 200, C 40. Her dilimde en küçük **pass** değerli süreç çalışır, pass değerine adımını ekler; eşitlikte A, B, C sırası.



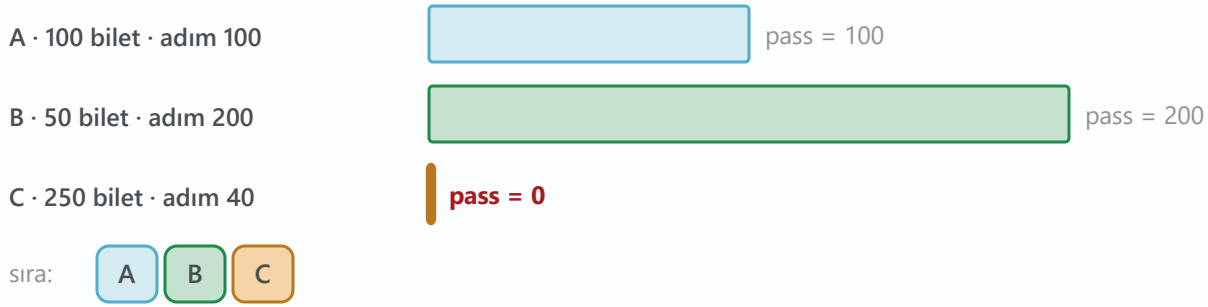
Adım 2: pass değerleri A 100, B 0, C 0 → en küçük **B** çalışır; pass(B) += 200.

Stride, piyangonun deterministik hali (Waldspurger, 1995). Dezavantajı küresel durum: yeni gelen sürecin pass değeri 0 olursa CPU'yu tekeline alır; bu yüzden mevcut en küçük pass ile başlatılır. Linux CFS'in vruntime fikri buna çok yakındır.

ORANTILI PAY

Stride Çizelgeleme Adım Adım

Adım = büyük sayı / bilet (10.000 / bilet): A 100, B 200, C 40. Her dilimde en küçük **pass** değerli süreç çalışır, pass değerine adımını ekler; eşitlikte A, B, C sırası.

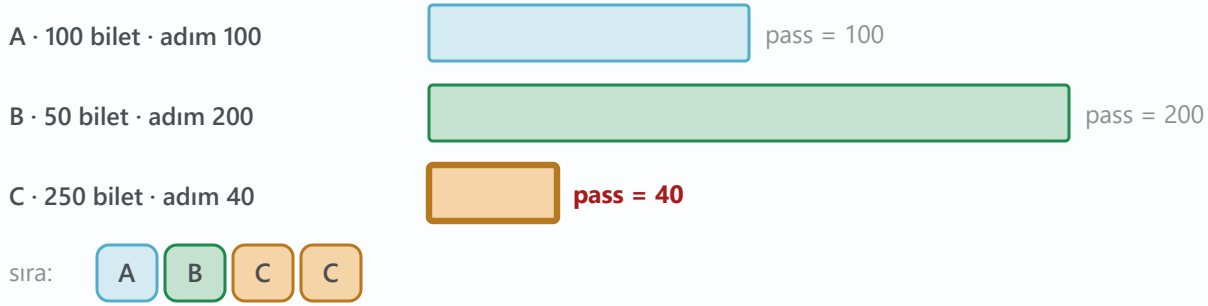


Adım 3: pass değerleri A 100, B 200, C 0 → en küçük **C** çalışır; pass(C) += 40.

Stride, piyangonun deterministik hali (Waldspurger, 1995). Dezavantajı küresel durum: yeni gelen sürecin pass değeri 0 olursa CPU'yu tekeline alır; bu yüzden mevcut en küçük pass ile başlatılır. Linux CFS'in vruntime fikri buna çok yakındır.

Stride Çizelgeleme Adım Adım

Adım = büyük sayı / bilet (10.000 / bilet): A 100, B 200, C 40. Her dilimde en küçük **pass** değerli süreç çalışır, pass değerine adımını ekler; eşitlikte A, B, C sırası.

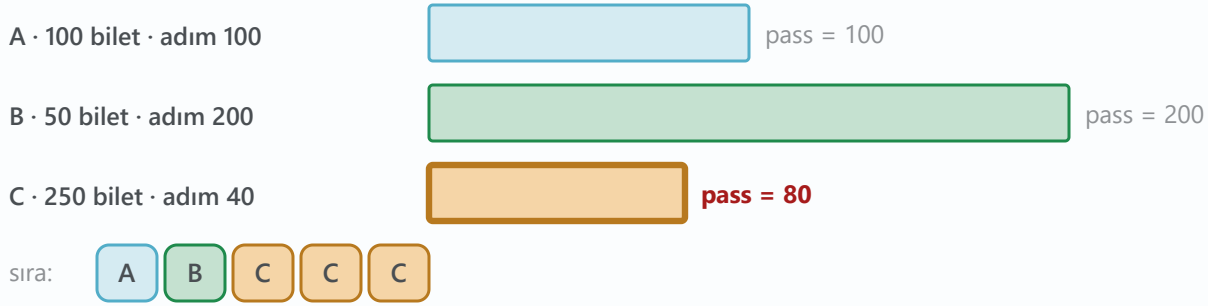


Adım 4: pass değerleri A 100, B 200, C 40 → en küçük **C** çalışır; pass(C) += 40.

Stride, piyangonun deterministik hali (Waldspurger, 1995). Dezavantajı küresel durum: yeni gelen sürecin pass değeri 0 olursa CPU'yu tekeline alır; bu yüzden mevcut en küçük pass ile başlatılır. Linux CFS'in vruntime fikri buna çok yakındır.

Stride Çizelgeleme Adım Adım

Adım = büyük sayı / bilet (10.000 / bilet): A 100, B 200, C 40. Her dilimde en küçük **pass** değerli süreç çalışır, pass değerine adımını ekler; eşitlikte A, B, C sırası.

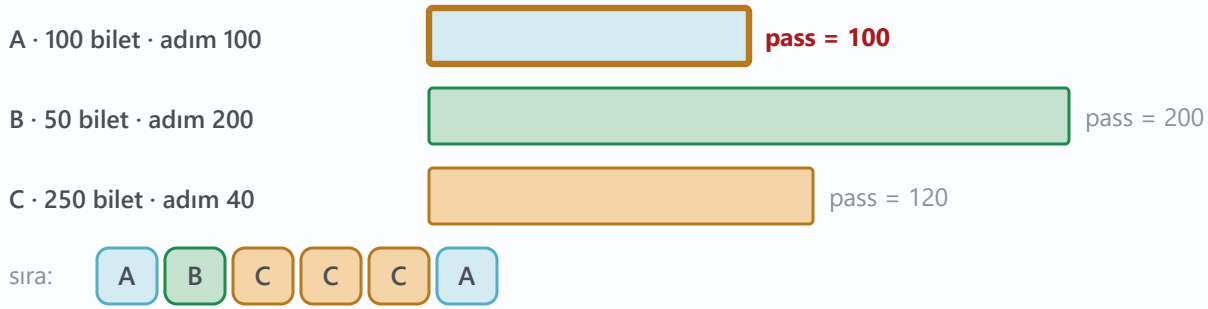


Adım 5: pass değerleri A 100, B 200, C 80 → en küçük **C** çalışır; pass(C) += 40.

Stride, piyangonun deterministik hali (Waldspurger, 1995). Dezavantajı küresel durum: yeni gelen sürecin pass değeri 0 olursa CPU'yu tekeline alır; bu yüzden mevcut en küçük pass ile başlatılır. Linux CFS'in vruntime fikri buna çok yakındır.

Stride Çizelgeleme Adım Adım

Adım = büyük sayı / bilet (10.000 / bilet): A 100, B 200, C 40. Her dilimde en küçük **pass** değerli süreç çalışır, pass değerine adımını ekler; eşitlikte A, B, C sırası.

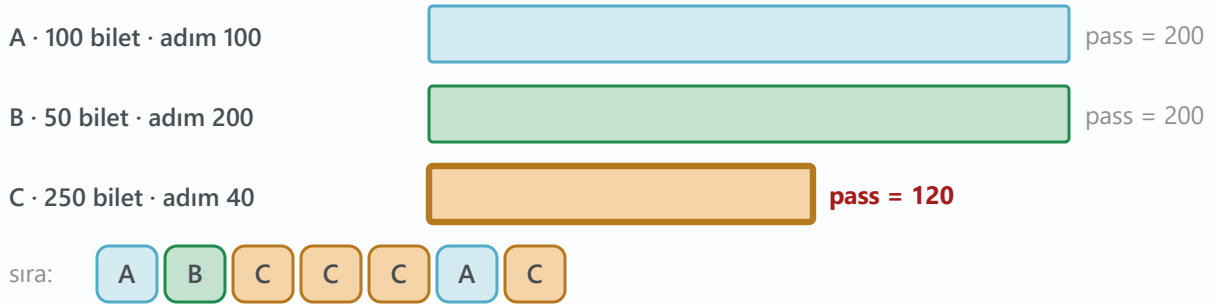


Adım 6: pass değerleri A 100, B 200, C 120 → en küçük **A** çalışır; pass(A) += 100.

Stride, piyangonun deterministik hali (Waldspurger, 1995). Dezavantajı küresel durum: yeni gelen sürecin pass değeri 0 olursa CPU'yu tekeline alır; bu yüzden mevcut en küçük pass ile başlatılır. Linux CFS'in vruntime fikri buna çok yakındır.

Stride Çizelgeleme Adım Adım

Adım = büyük sayı / bilet (10.000 / bilet): A 100, B 200, C 40. Her dilimde en küçük **pass** değerli süreç çalışır, pass değerine adımını ekler; eşitlikte A, B, C sırası.



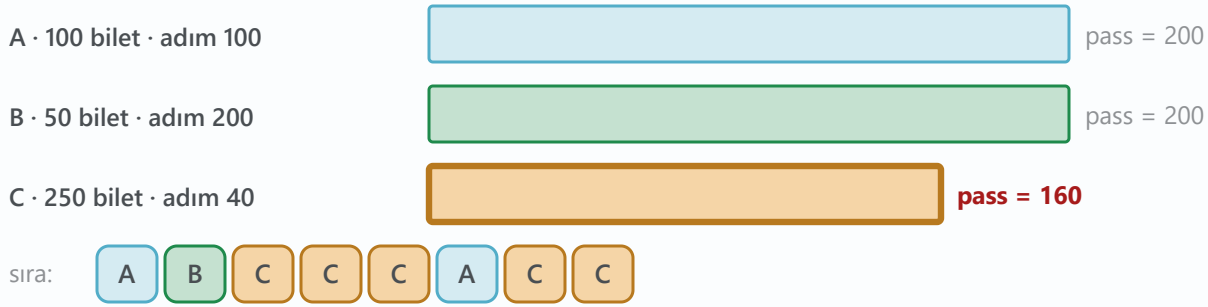
Adım 7: pass değerleri A 200, B 200, C 120 → en küçük **C** çalışır; pass(C) += 40.

Stride, piyangonun deterministik hali (Waldspurger, 1995). Dezavantajı küresel durum: yeni gelen sürecin pass değeri 0 olursa CPU'yu tekeline alır; bu yüzden mevcut en küçük pass ile başlatılır. Linux CFS'in vruntime fikri buna çok yakındır.

ORANTILI PAY

Stride Çizelgeleme Adım Adım

Adım = büyük sayı / bilet (10.000 / bilet): A 100, B 200, C 40. Her dilimde en küçük **pass** değerli süreç çalışır, pass değerine adımını ekler; eşitlikte A, B, C sırası.



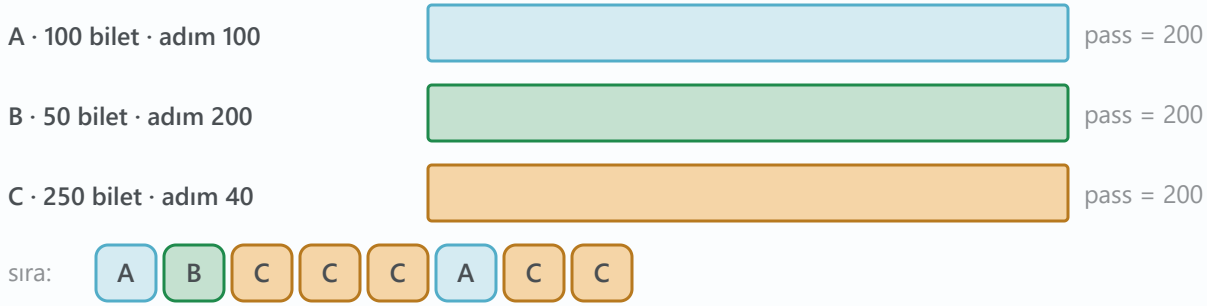
Adım 8: pass değerleri A 200, B 200, C 160 → en küçük **C** çalışır; pass(C) += 40.

Stride, piyangonun deterministik hali (Waldspurger, 1995). Dezavantajı küresel durum: yeni gelen sürecin pass değeri 0 olursa CPU'yu tekeline alır; bu yüzden mevcut en küçük pass ile başlatılır. Linux CFS'in vruntime fikri buna çok yakındır.

ORANTILI PAY

Stride Çizelgeleme Adım Adım

Adım = büyük sayı / bilet (10.000 / bilet): A 100, B 200, C 40. Her dilimde en küçük **pass** değerli süreç çalışır, pass değerine adımını ekler; eşitlikte A, B, C sırası.

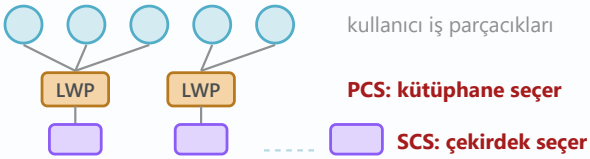


8 dilimde A 2, B 1, C 5 kez çalıştı: tam olarak 100 : 50 : 250 = 2 : 1 : 5 oranı. Piyangoda bu oran ancak uzun vadede tutar.

Stride, piyangonun deterministik hali (Waldspurger, 1995). Dezavantajı küresel durum: yeni gelen sürecin pass değeri 0 olursa CPU'yu tekeline alır; bu yüzden mevcut en küçük pass ile başlatılır. Linux CFS'in vruntime fikri buna çok yakındır.

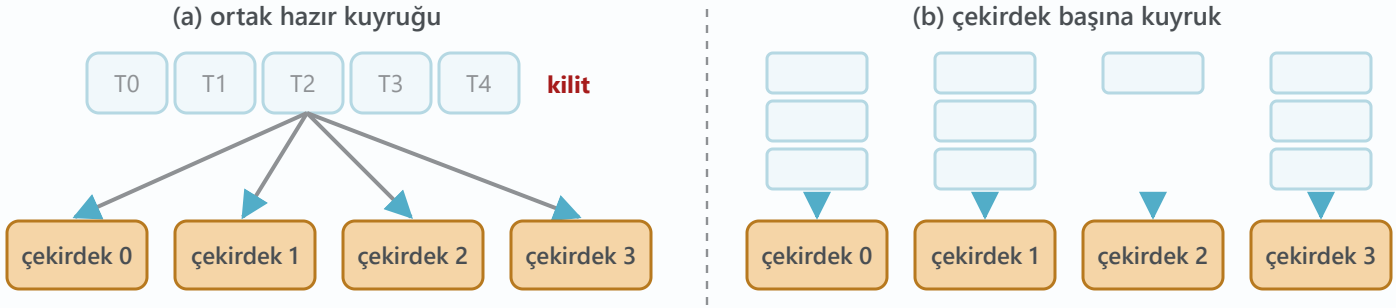
İş Parçacığı Çizelgeleme: PCS ve SCS

- Çekirdek, süreçleri değil **çekirdek iş parçacıklarını** çizelgeler. Kullanıcı iş parçacıklarını bir kütüphane yönetir.
- **PCS** (process-contention scope): N:1 ve M:N modellerinde kütüphane, kullanıcı iş parçacıklarını kendi süreci içindeki LWP'ler üzerinde yarıştırır.
- **SCS** (system-contention scope): çekirdek, sistemdeki tüm iş parçacıkları arasında seçim yapar. 1:1 modelli Linux, Windows ve macOS yalnızca SCS kullanır.
- Güncel PCS örneği: Java 21 **sanal iş parçacıkları** ve Go goroutine'leri, çalışma zamanının kendi çizelgeleyicisiyle az sayıda taşıyıcı iş parçacığına eşlenir.



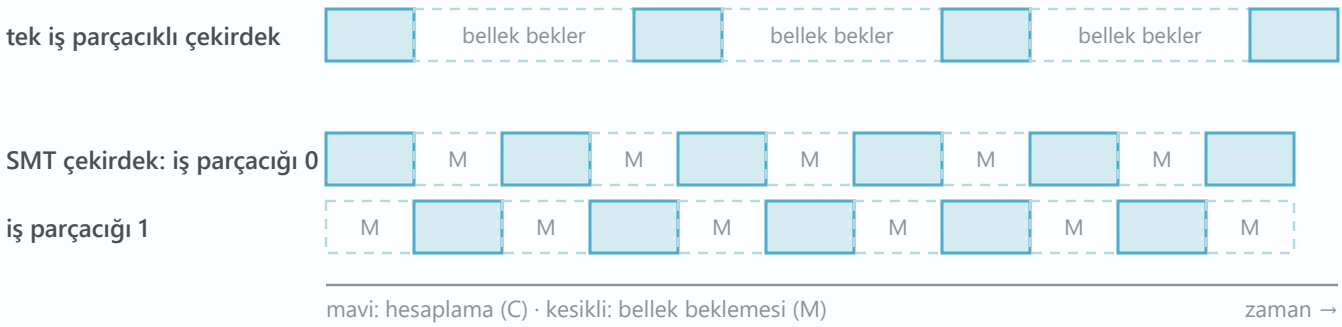

```c
pthread_attr_t attr;
int scope;
pthread_attr_init(&attr);
pthread_attr_getscope(&attr, &scope);
if (scope == PTHREAD_SCOPE_PROCESS)
  printf("PCS\n");
/* Linux ve macOS yalnızca SYSTEM */
pthread_attr_setscope(&attr,
                      PTHREAD_SCOPE_SYSTEM);
pthread_create(&tid, &attr, runner, NULL);
```

Çok İşlemcili Çizelgeleme Yaklaşımları



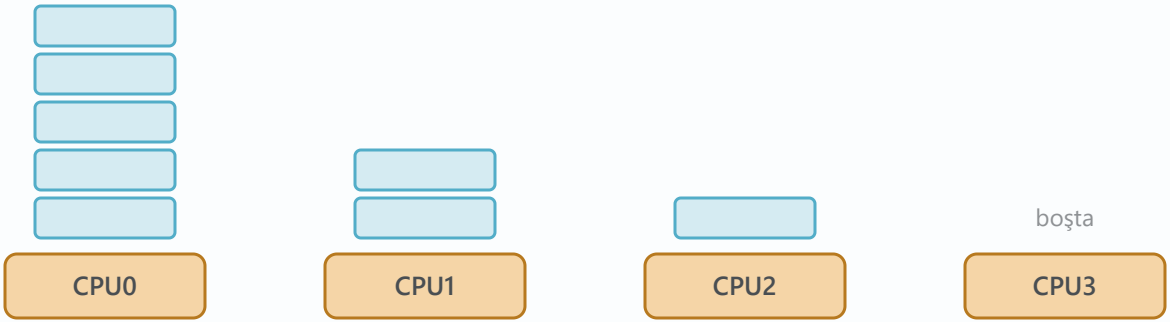
- **Asimetrik** (AMP): tek bir ana işlemci tüm çizelgeleme ve G/Ç'yi yapar; basit ama darboğaz.
- **Simetrik** (SMP): her işlemci kendi kendini çizelgeler; tüm modern sistemler.
- (a) Ortak kuyruk: yük kendiliğinden dengelenir ama kilit çekişmesi ve zayıf önbellek yakınlığı.
- (b) Çekirdek başına kuyruk (**iki düzeyli çizelgeleme**): iş önce bir çekirdeğe atanır, sonra onun kuyruğundan çalışır. Kilit yok denecek kadar azdır, önbellek sıcak kalır. Linux, Windows ve FreeBSD bunu kullanır.
- Bedeli: kuyruklar dengesizleşebilir → **yük dengeleme** gerekir.

Çok Çekirdek ve Donanım İş Parçacıkları



- **Bellek beklemesi** (memory stall): önbellek ıskalarında çekirdek zamanın %50'ye varan kısmını veri beklerken geçirebilir.
- **SMT** (Intel Hyper-Threading, AMD SMT): bir çekirdekte 2 donanım iş parçacığı; biri beklerken diğeri yürür. İşletim sistemi her donanım iş parçacığını ayrı bir **mantıksal CPU** olarak görür.
- İki düzeyli çizelgeleme: işletim sistemi hangi yazılım iş parçacığının hangi mantıksal CPU'da çalışacağını, çekirdek donanımı hangi donanım iş parçacığının o an yürüyeceğini seçer.
- Kardeş mantıksal CPU'lar yürütme birimlerini paylaşır: boş fiziksel çekirdek varken iki yoğun işi aynı çekirdeğe koymak kötü bir karardır.

Yük Dengeleme

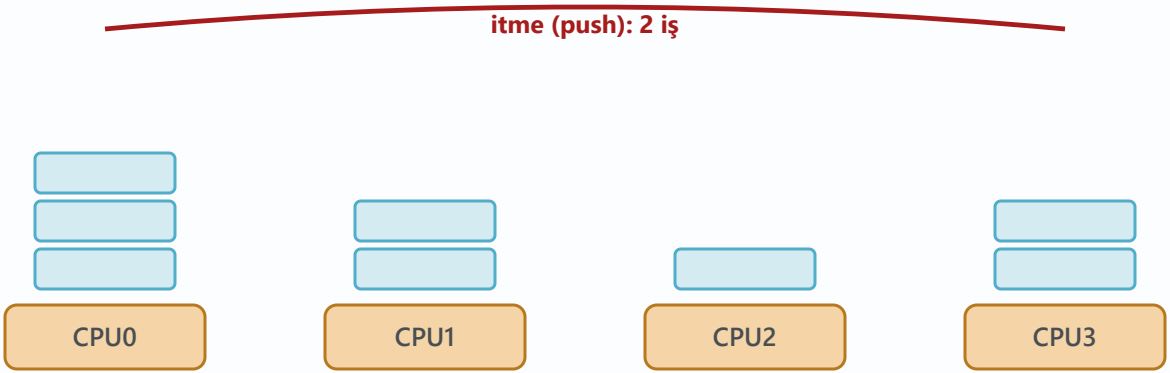


Başlangıç: CPU0'ın kuyruğunda 5 iş var, CPU3 boşta. Çekirdek başına kuyrukta dengesizlik kendiliğinden düzelmez.

Adım 1 / 5

Etkileşimli simülatör: Çok Çekirdekli Çizelgeleme ve Yük Dengeleme Simülatörü: görevlerin çekirdeklere dağıtımında afinite ve yük dengelemenin etkisi.

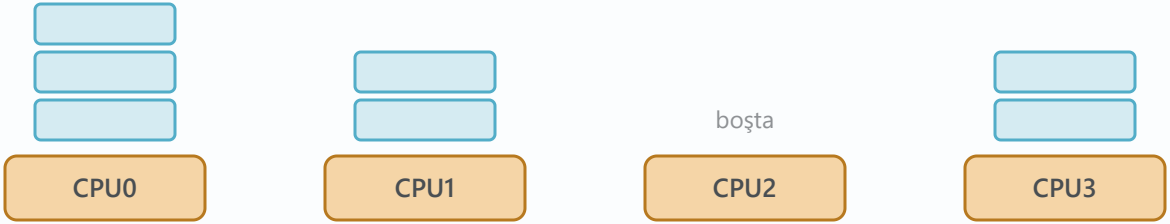
Yük Dengeleme



İtme göçü (push): periyodik dengeleyici yükleri karşılaştırır, aşırı yüklü CPU0'dan CPU3'e 2 iş iter.

Etkileşimli simülatör: Çok Çekirdekli Çizelgeleme ve Yük Dengeleme Simülatörü: görevlerin çekirdeklere dağıtımında afinite ve yük dengelemenin etkisi.

Yük Dengeleme

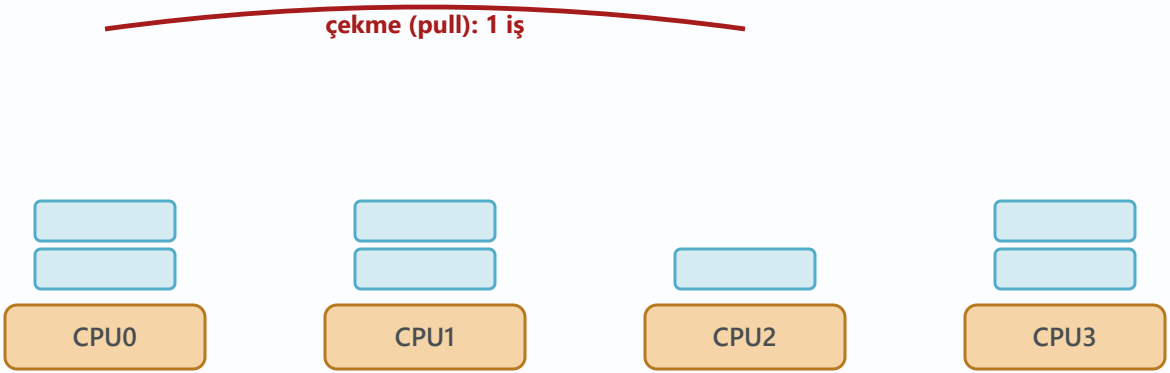


CPU2'deki tek iş bitti; CPU2 boşta kaldı.

Adım 3 / 5

Etkileşimli simülatör: Çok Çekirdekli Çizelgeleme ve Yük Dengeleme Simülatörü: görevlerin çekirdeklere dağıtımında afinite ve yük dengelemenin etkisi.

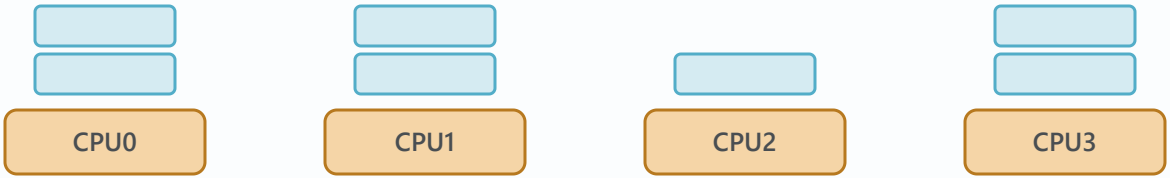
Yük Dengeleme



Çekme göçü (pull): boşta kalan CPU2 en yüklü kuyruktan bir iş çalar. Linux her ikisini de kullanır.

Etkileşimli simülatör: Çok Çekirdekli Çizelgeleme ve Yük Dengeleme Simülatörü: görevlerin çekirdeklere dağıtımında afinite ve yük dengelemenin etkisi.

Yük Dengeleme

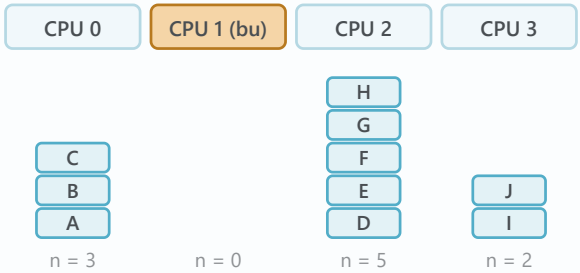


Denge sağlandı; ama taşınan işin önbelleği yeni CPU'da **soğuktur**. Linux önce SMT kardeşleri ve ortak önbellek içinde, NUMA düğümleri arasında daha seyrek dengeler (scheduling domains).

Etkileşimli simülatör: Çok Çekirdekli Çizelgeleme ve Yük Dengeleme Simülatörü: görevlerin çekirdeklere dağıtımında afinite ve yük dengelemenin etkisi.

Yük Dengeleme: Kod İzleme

```
▶ void dengele(int bu) {   /* boştaki CPU */
▶   int en = bu;
3   for (int c = 0; c < NCPU; c++)
4     if (q[c].n > q[en].n) /* daha yüklü */
5       en = c;
6   int t = (q[en].n - q[bu].n) / 2;
7   for (int k = 0; k < t; k++)
8     gocur(&q[en], &q[bu]); /* sondan */
9 }
```



CPU 1'in kuyruğu boşaldı. Çekme (pull) dengelemesi: `dengele(1)`. Başlangıçta en = bu = 1.

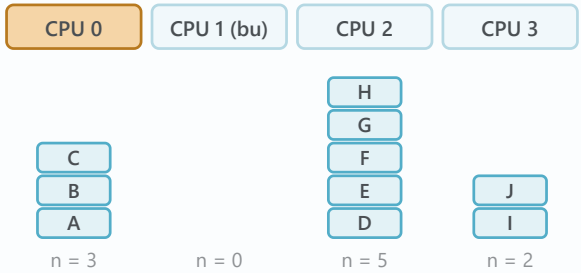
Adım 1 / 8

DEĞİŞKENLER

`en = 1` `c = –` `t = –` `k = –`

Yük Dengeleme: Kod İzleme

```
1 void dengele(int bu) {   /* boştaki CPU */
2   int en = bu;
▶   for (int c = 0; c < NCPU; c++)
▶     if (q[c].n > q[en].n) /* daha yüklü */
▶       en = c;
6   int t = (q[en].n - q[bu].n) / 2;
7   for (int k = 0; k < t; k++)
8     gocur(&q[en], &q[bu]); /* sondan */
9 }
```



c = 0: n = 3 > önceki en yüklü → en = 0.

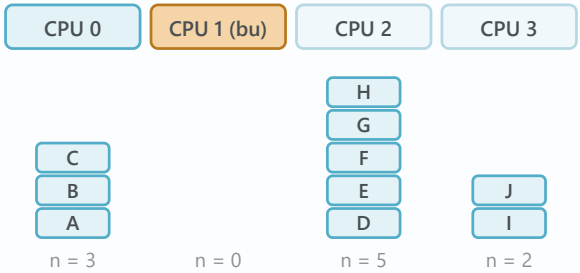
Adım 2 / 8

DEĞİŞKENLER

`en = 0` `c = 0` `t = –` `k = –`

Yük Dengeleme: Kod İzleme

```
1 void dengele(int bu) {   /* boştaki CPU */
2   int en = bu;
▶   for (int c = 0; c < NCPU; c++)
▶     if (q[c].n > q[en].n) /* daha yüklü */
5       en = c;
6   int t = (q[en].n - q[bu].n) / 2;
7   for (int k = 0; k < t; k++)
8     gocur(&q[en], &q[bu]); /* sondan */
9 }
```



c = 1: n = 0, daha yüklü değil.

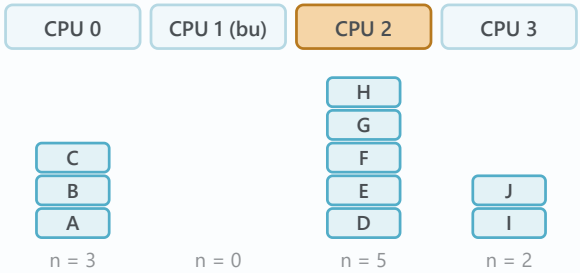
Adım 3 / 8

DEĞİŞKENLER

en = 0 | c = 1 | t = – | k = –

Yük Dengeleme: Kod İzleme

```
1 void dengele(int bu) {   /* boştaki CPU */
2   int en = bu;
▶   for (int c = 0; c < NCPU; c++)
▶     if (q[c].n > q[en].n) /* daha yüklü */
▶       en = c;
6   int t = (q[en].n - q[bu].n) / 2;
7   for (int k = 0; k < t; k++)
8     gocur(&q[en], &q[bu]); /* sondan */
9 }
```



c = 2: n = 5 > önceki en yüklü → en = 2.

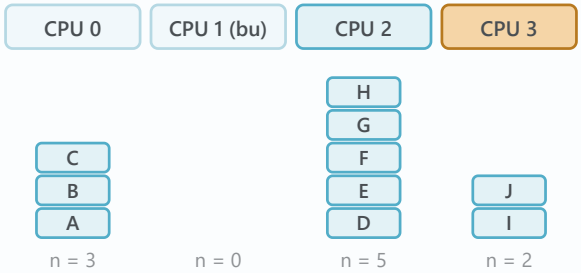
Adım 4 / 8

DEĞİŞKENLER

`en = 2` `c = 2` `t = –` `k = –`

Yük Dengeleme: Kod İzleme

```
1 void dengele(int bu) {   /* boştaki CPU */
2   int en = bu;
▶   for (int c = 0; c < NCPU; c++)
▶     if (q[c].n > q[en].n) /* daha yüklü */
5       en = c;
6   int t = (q[en].n - q[bu].n) / 2;
7   for (int k = 0; k < t; k++)
8     gocur(&q[en], &q[bu]); /* sondan */
9 }
```



c = 3: n = 2, daha yüklü değil.

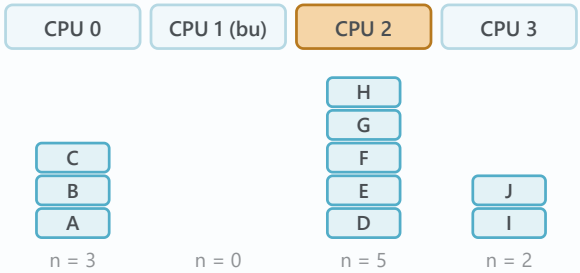
Adım 5 / 8

DEĞİŞKENLER

en = **2** | c = **3** | t = **–** | k = **–**

Yük Dengeleme: Kod İzleme

```
1 void dengele(int bu) {   /* boştaki CPU */
2   int en = bu;
3   for (int c = 0; c < NCPU; c++)
4     if (q[c].n > q[en].n) /* daha yüklü */
5       en = c;
▶ int t = (q[en].n - q[bu].n) / 2;
7   for (int k = 0; k < t; k++)
8     gocur(&q[en], &q[bu]); /* sondan */
9 }
```



t = (5 − 0) / 2 = 2: farkın yarısı taşınırsa iki kuyruk eşitlenmeye yaklaşır (CPU 2: 3, CPU 1: 2).

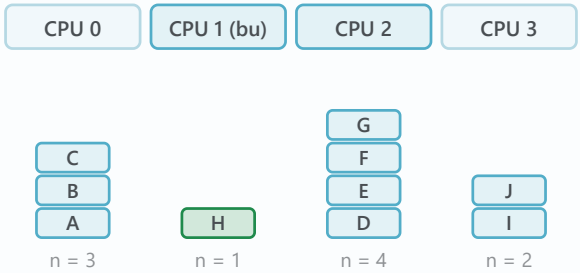
Adım 6 / 8

DEĞİŞKENLER

`en = 2` `c = 3` `t = 2` `k = –`

Yük Dengeleme: Kod İzleme

```
1 void dengele(int bu) {   /* boştaki CPU */
2   int en = bu;
3   for (int c = 0; c < NCPU; c++)
4     if (q[c].n > q[en].n) /* daha yüklü */
5       en = c;
6   int t = (q[en].n - q[bu].n) / 2;
▶   for (int k = 0; k < t; k++)
▶     gocur(&q[en], &q[bu]); /* sondan */
9 }
```



k = 0: CPU 2'nin kuyruğunun sonundaki **H** CPU 1'e göçer. Kuyruk sonundakiler en uzun süre beklemiş olanlar değil, en son gelenlerdir: önbellekleri en soğuk olanlar.

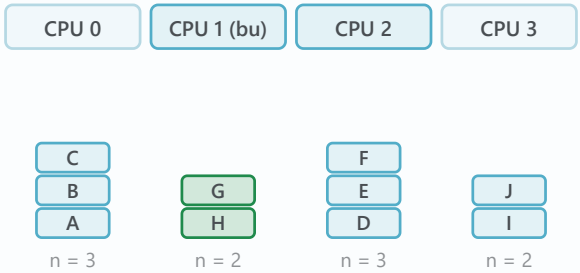
Adım 7 / 8

DEĞİŞKENLER

`en = 2` `c = 3` `t = 2` `k = 0`

Yük Dengeleme: Kod İzleme

```
1 void dengele(int bu) {   /* boştaki CPU */
2   int en = bu;
3   for (int c = 0; c < NCPU; c++)
4     if (q[c].n > q[en].n) /* daha yüklü */
5       en = c;
6   int t = (q[en].n - q[bu].n) / 2;
▶   for (int k = 0; k < t; k++)
▶     gocur(&q[en], &q[bu]); /* sondan */
9 }
```



k = 1: CPU 2'nin kuyruğunun sonundaki **G** CPU 1'e göçer. Kuyruk uzunlukları 3, 2, 3, 2. Göç eden işler yeni CPU'da önbellek ve TLB'yi soğuk bulur: afinitenin bedeli.

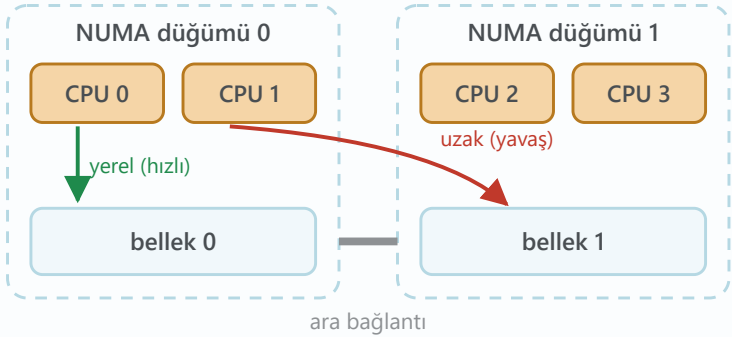
Adım 8 / 8

DEĞİŞKENLER

`en = 2` `c = 3` `t = 2` `k = 1`

İşlemci Afinitesi, NUMA ve Heterojen Çekirdekler

- **İşlemci afinitesi** (affinity): iş parçacığını önbelleği ve TLB'si sıcak olan CPU'da tutmak. **Yumuşak**: çizelgeleyici dener ama garanti etmez. **Katı**: izin verilen CPU kümesi (`sched_setaffinity()` , `taskset -c 0-3`).
- **NUMA**: iş parçacığını belleğinin bulunduğu düğümde çalıştırmak; Linux otomatik NUMA dengelemesi sayfaları ve görevleri yaklaştırır.
- Yük dengeleme ile yakınlık **çelişir**: göç dengeyi sağlar ama önbelleği ve yerelliği bozar.



Heterojen çok işlemci: ARM big.LITTLE / DynamIQ, Intel 12. nesilden beri P/E çekirdekleri (Thread Director donanım ipuçları verir), Apple M serisi. Linux **EAS** (Energy Aware Scheduling) işi, performans hedefini karşılayan en az enerji harcayan çekirdeğe koyar; arka plan işleri verimli çekirdeklere gider.

Gerçek Zamanlı Sistemler ve Gecikme

Katı (hard) gerçek zaman

Son tarih kaçırılırsa sistem başarısızdır: ABS fren, hava yastığı, endüstriyel robot kontrolü, uçuş denetimi.

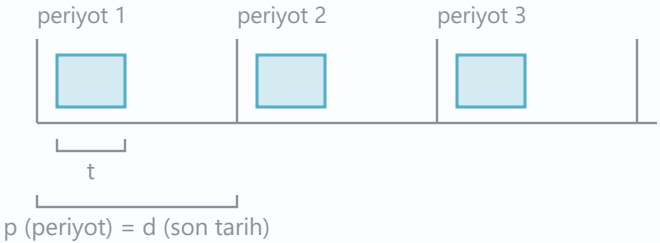
Yumuşak (soft) gerçek zaman

Kritik görev öncelikli dir ama garanti yoktur; kaçırma kaliteyi düşürür: video akışı, VoIP, oyun, ses çalma.

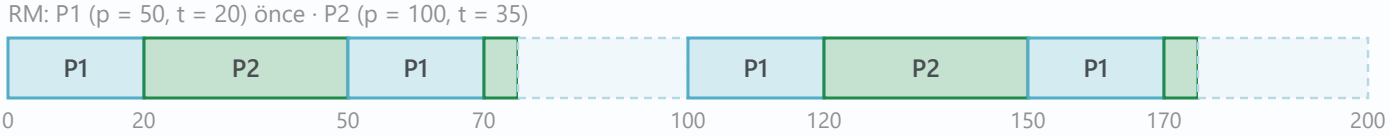


- **Çakışmalar:** çekirdekte çalışan herhangi bir işin kesilmesi ve düşük öncelikli sürecin tuttuğu kaynağın bırakılması. Bu yüzden kesmeli, öncelik tabanlı çekirdek gerekir; katı garanti için ayrıca **kabul denetimi** (admission control).
- Algoritmalar **statik** (öncelikler önceden hesaplanır, ör. RM) ya da **dinamik** (kararlar çalışma anında, ör. EDF) olabilir.

Periyodik Görevler ve Hız Monoton (RM)



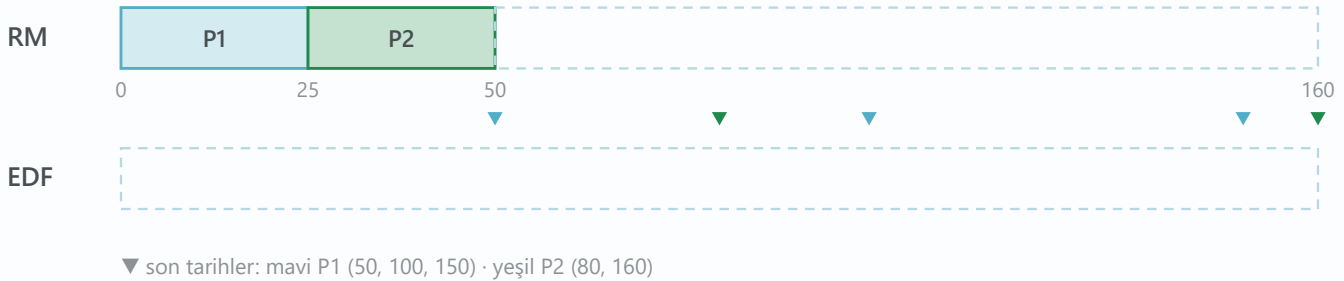
- Periyodik görev: $0 \le t \le d \le p$; kullanım $U = t / p$.
- **Hız monoton** (rate-monotonic, RM): statik öncelik = 1 / p; kısa periyot yüksek öncelik. Statik öncelikli algoritmalar içinde **optimaldir**.
- Liu–Layland sınırı: $\Sigma U \le n(2^{1/n} - 1)$ ise RM tüm son tarihleri karşılar. n = 1: 1 · n = 2: ≈ 0,828 · n = 3: ≈ 0,780 · n → ∞: ln 2 ≈ 0,693. Yeterli koşuldur, gerekli değil.



Örnek: U = 20/50 + 35/100 = 0,40 + 0,35 = 0,75 ≤ 0,828 → RM ile çizelgelenebilir. Silberschatz Şekil 5.21'den yeniden hesaplanmıştır.

GERÇEK ZAMANLI ÇİZELGELEME

RM ve EDF Adım Adım



▼ son tarihler: mavi P1 (50, 100, 150) · yeşil P2 (80, 160)

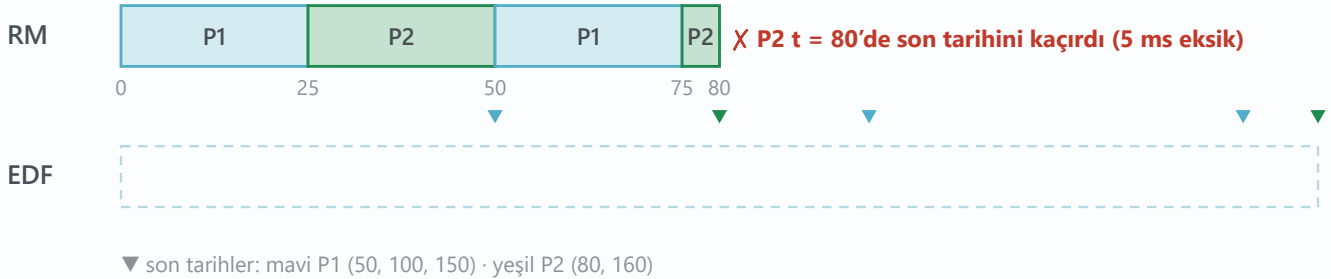
U = 25/50 + 35/80 = 0,5 + 0,4375 = **0,94** > 0,828: RM için garanti yok. RM: P1 (kısa periyot) önce, 0–25; P2 25–50.

Adım 1 / 5

Etkileşimli simülatör: Gerçek Zamanlı Çizelgeleme Simülatörü: periyodik görev kümeleri için RM ve EDF çizelgeleri, kullanım oranı ve kaçırılan son tarihler.

Orantılı pay (proportional share) çizelgeleme: toplam T pay dağıtılır; N pay alan uygulama CPU'nun N/T'sini alır, kabul denetimi toplamı aşan isteği reddeder. Linux `SCHED_DEADLINE` (3.14+) EDF ile CBS bant genişliği korumasını birleştirir.

RM ve EDF Adım Adım



▼ son tarihler: mavi P1 (50, 100, 150) · yeşil P2 (80, 160)

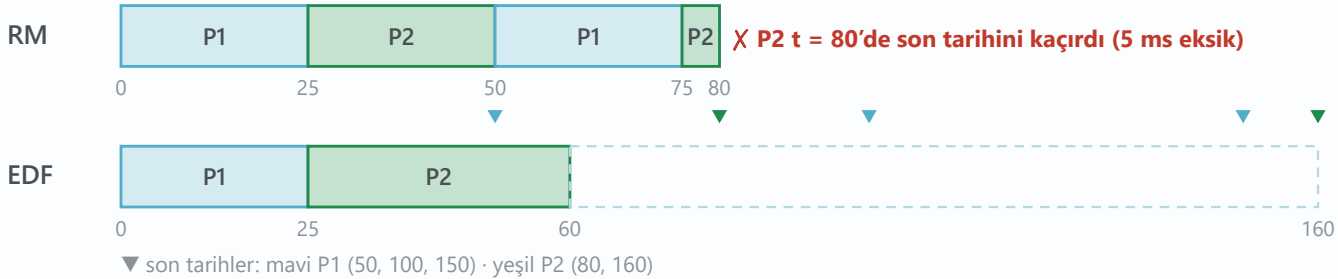
t = 50: P1'in yeni işi gelir ve statik önceliği yüksek olduğu için P2'yi keser. P2 75–80'de 5 ms daha çalışır ama süre biter: **son tarih kaçırıldı**.

Etkileşimli simülatör: Gerçek Zamanlı Çizelgeleme Simülatörü: periyodik görev kümeleri için RM ve EDF çizelgeleri, kullanım oranı ve kaçırılan son tarihler.

Orantılı pay (proportional share) çizelgeleme: toplam T pay dağıtılır; N pay alan uygulama CPU'nun N/T'sini alır, kabul denetimi toplamı aşan isteği reddeder. Linux `SCHED_DEADLINE` (3.14+) EDF ile CBS bant genişliği korumasını birleştirir.

GERÇEK ZAMANLI ÇİZELGELEME

RM ve EDF Adım Adım



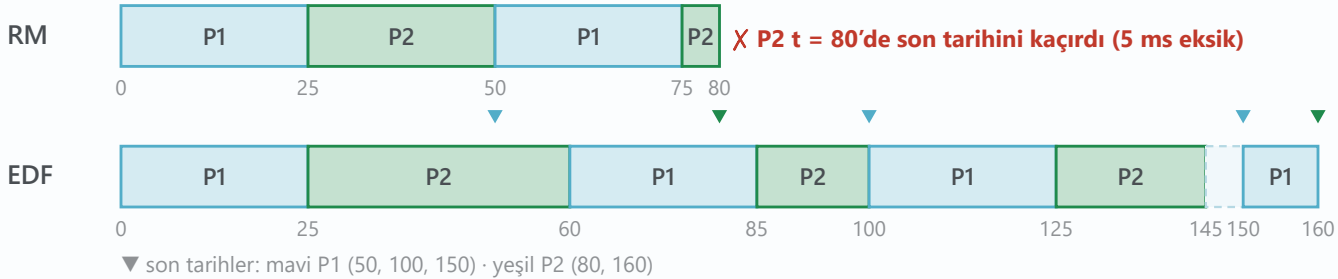
EDF: önceliği son tarih belirler. t = 50'de P1'in son tarihi 100, P2'ninki 80 → **P2 devam eder** ve 60'ta biter.

Adım 3 / 5

Etkileşimli simülatör: Gerçek Zamanlı Çizelgeleme Simülatörü: periyodik görev kümeleri için RM ve EDF çizelgeleri, kullanım oranı ve kaçırılan son tarihler.

Orantılı pay (proportional share) çizelgeleme: toplam T pay dağıtılır; N pay alan uygulama CPU'nun N/T'sini alır, kabul denetimi toplamı aşan isteği reddeder. Linux `SCHED_DEADLINE` (3.14+) EDF ile CBS bant genişliği korumasını birleştirir.

RM ve EDF Adım Adım

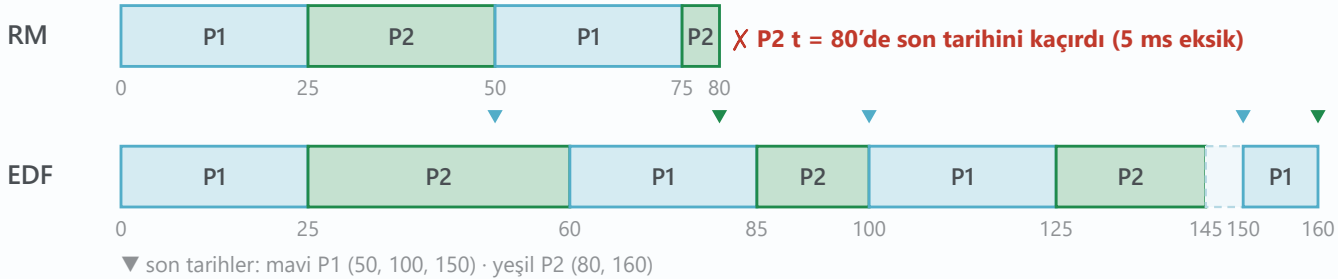


t = 80'de P2'nin yeni işinin son tarihi 160, P1'inki 100 → P1 85'e kadar sürer. 100'de P1 (150) yine önce. Tüm son tarihler karşılanır.

Etkileşimli simülatör: Gerçek Zamanlı Çizelgeleme Simülatörü: periyodik görev kümeleri için RM ve EDF çizelgeleri, kullanım oranı ve kaçırılan son tarihler.

Orantılı pay (proportional share) çizelgeleme: toplam T pay dağıtılır; N pay alan uygulama CPU'nun N/T'sini alır, kabul denetimi toplamı aşan isteği reddeder. Linux `SCHED_DEADLINE` (3.14+) EDF ile CBS bant genişliği korumasını birleştirir.

RM ve EDF Adım Adım



EDF'de koşul ΣU ≤ 1 (kesmeli, bağımsız görevler, d = p): 0,94 ≤ 1 ve 400 ms'lik hiperperiyot boyunca hiç kaçırma yok. Bedeli: dinamik öncelik ve aşırı yükte öngörülemez kayıplar.

Etkileşimli simülatör: Gerçek Zamanlı Çizelgeleme Simülatörü: periyodik görev kümeleri için RM ve EDF çizelgeleri, kullanım oranı ve kaçırılan son tarihler.

Orantılı pay (proportional share) çizelgeleme: toplam T pay dağıtılır; N pay alan uygulama CPU'nun N/T'sini alır, kabul denetimi toplamı aşan isteği reddeder. Linux `SCHED_DEADLINE` (3.14+) EDF ile CBS bant genişliği korumasını birleştirir.

Liu–Layland Testi: Kod İzleme

```
▶ int rm_test(Gorev g[], int n) {
▶   double U = 0;
3   for (int i = 0; i < n; i++)
4     U += (double) g[i].t / g[i].p;
5   double sinir = n * (pow(2, 1.0/n) - 1);
6   if (U <= sinir) return 1; /* yeter */
7   if (U <= 1.0) return 0;   /* belirsiz */
8   return -1;            /* imkânsız */
9 }
```



Görevler: T1(p=20, t=5), T2(p=50, t=10), T3(p=100, t=30). Toplam kullanım U sıfırdan başlar.

Adım 1 / 9

DEĞİŞKENLER

`n = 3` `U = 0`

ÇIKTI

Test yeterli koşuldur: 1 dönerse RM kesin yeter; 0 dönerse karar için çizelge (ör. 0'dan en büyük periyoda kadar) çizilmelidir.

Liu–Layland Testi: Kod İzleme

```
1 int rm_test(Gorev g[], int n) {
2   double U = 0;
▶   for (int i = 0; i < n; i++)
▶     U += (double) g[i].t / g[i].p;
5   double sinir = n * (pow(2, 1.0/n) - 1);
6   if (U <= sinir) return 1; /* yeter */
7   if (U <= 1.0) return 0;   /* belirsiz */
8   return -1;            /* imkânsız */
9 }
```



U = 0,25

T1: 5 / 20 = 0,25 eklenir → U = **0,25**.

Adım 2 / 9

DEĞİŞKENLER

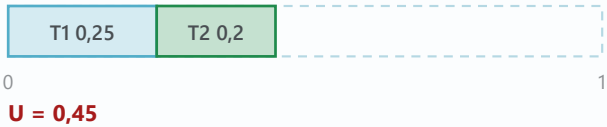
`n = 3` `i = 0` `U = 0.25`

ÇIKTI

Test yeterli koşuldur: 1 dönerse RM kesin yeter; 0 dönerse karar için çizelge (ör. 0'dan en büyük periyoda kadar) çizilmelidir.

Liu–Layland Testi: Kod İzleme

```
1 int rm_test(Gorev g[], int n) {
2   double U = 0;
▶   for (int i = 0; i < n; i++)
▶     U += (double) g[i].t / g[i].p;
5   double sinir = n * (pow(2, 1.0/n) - 1);
6   if (U <= sinir) return 1; /* yeter */
7   if (U <= 1.0) return 0;   /* belirsiz */
8   return -1;            /* imkânsız */
9 }
```



U = 0,45

T2: 10 / 50 = 0,2 eklenir → U = **0,45**.

Adım 3 / 9

DEĞİŞKENLER

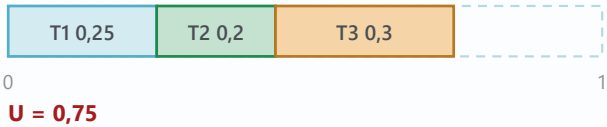
`n = 3` `i = 1` `U = 0.45`

ÇIKTI

Test yeterli koşuldur: 1 dönerse RM kesin yeter; 0 dönerse karar için çizelge (ör. 0'dan en büyük periyoda kadar) çizilmelidir.

Liu–Layland Testi: Kod İzleme

```
1 int rm_test(Gorev g[], int n) {
2   double U = 0;
▶   for (int i = 0; i < n; i++)
▶     U += (double) g[i].t / g[i].p;
5   double sinir = n * (pow(2, 1.0/n) - 1);
6   if (U <= sinir) return 1; /* yeter */
7   if (U <= 1.0) return 0;   /* belirsiz */
8   return -1;            /* imkânsız */
9 }
```



U = 0,75

T3: 30 / 100 = 0,3 eklenir → U = **0,75**.

Adım 4 / 9

DEĞİŞKENLER

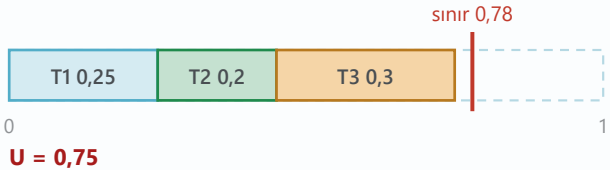
`n = 3` `i = 2` `U = 0.75`

ÇIKTI

Test yeterli koşuldur: 1 dönerse RM kesin yeter; 0 dönerse karar için çizelge (ör. 0'dan en büyük periyoda kadar) çizilmelidir.

Liu–Layland Testi: Kod İzleme

```
1 int rm_test(Gorev g[], int n) {
2   double U = 0;
3   for (int i = 0; i < n; i++)
4     U += (double) g[i].t / g[i].p;
▶   double sinir = n * (pow(2, 1.0/n) - 1);
▶   if (U <= sinir) return 1; /* yeter */
7   if (U <= 1.0) return 0;    /* belirsiz */
8   return -1;                 /* imkânsız */
9 }
```



sınır = $3(2^{1/3} - 1) = 0{,}7798$; U = 0,75 ≤ sınır → **1**: RM tüm son tarihleri karşılar.

Adım 5 / 9

DEĞİŞKENLER

`n = 3` `U = 0.75` `sinir = 0.7798`

ÇIKTI

```
rm_test = 1
```

Test yeterli koşuldur: 1 dönerse RM kesin yeter; 0 dönerse karar için çizelge (ör. 0'dan en büyük periyoda kadar) çizilmelidir.

Liu–Layland Testi: Kod İzleme

```
▶ int rm_test(Gorev g[], int n) {
▶   double U = 0;
3   for (int i = 0; i < n; i++)
4     U += (double) g[i].t / g[i].p;
5   double sinir = n * (pow(2, 1.0/n) - 1);
6   if (U <= sinir) return 1; /* yeter */
7   if (U <= 1.0) return 0;   /* belirsiz */
8   return -1;              /* imkânsız */
9 }
```



İkinci çağrı, RM ve EDF örneğindeki küme: P1(p=50, t=25), P2(p=80, t=35).

Adım 6 / 9

DEĞİŞKENLER

`n = 2` `U = 0`

ÇIKTI

```
rm_test = 1
```

Test yeterli koşuldur: 1 dönerse RM kesin yeter; 0 dönerse karar için çizelge (ör. 0'dan en büyük periyoda kadar) çizilmelidir.

Liu–Layland Testi: Kod İzleme

```
1 int rm_test(Gorev g[], int n) {
2   double U = 0;
▶   for (int i = 0; i < n; i++)
▶     U += (double) g[i].t / g[i].p;
5   double sinir = n * (pow(2, 1.0/n) - 1);
6   if (U <= sinir) return 1; /* yeter */
7   if (U <= 1.0) return 0;   /* belirsiz */
8   return -1;            /* imkânsız */
9 }
```



U = 0,5

P1: 25 / 50 = 0,5 eklenir → U = **0,5**.

Adım 7 / 9

DEĞİŞKENLER

`n = 2` `i = 0` `U = 0.5`

ÇIKTI

```
rm_test = 1
```

Test yeterli koşuldur: 1 dönerse RM kesin yeter; 0 dönerse karar için çizelge (ör. 0'dan en büyük periyoda kadar) çizilmelidir.

Liu–Layland Testi: Kod İzleme

```
1 int rm_test(Gorev g[], int n) {
2   double U = 0;
▶   for (int i = 0; i < n; i++)
▶     U += (double) g[i].t / g[i].p;
5   double sinir = n * (pow(2, 1.0/n) - 1);
6   if (U <= sinir) return 1; /* yeter */
7   if (U <= 1.0) return 0;   /* belirsiz */
8   return -1;            /* imkânsız */
9 }
```



U = 0,9375

P2: 35 / 80 = 0,4375 eklenir → U = **0,9375**.

Adım 8 / 9

DEĞİŞKENLER

`n = 2` `i = 1` `U = 0.9375`

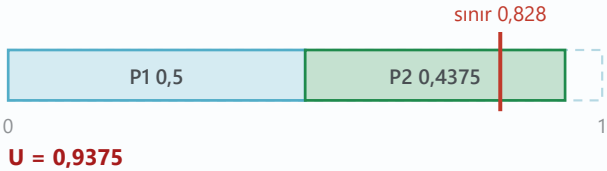
ÇIKTI

```
rm_test = 1
```

Test yeterli koşuldur: 1 dönerse RM kesin yeter; 0 dönerse karar için çizelge (ör. 0'dan en büyük periyoda kadar) çizilmelidir.

Liu–Layland Testi: Kod İzleme

```
1 int rm_test(Gorev g[], int n) {
2   double U = 0;
3   for (int i = 0; i < n; i++)
4     U += (double) g[i].t / g[i].p;
▶   double sinir = n * (pow(2, 1.0/n) - 1);
▶   if (U <= sinir) return 1; /* yeter */
▶   if (U <= 1.0) return 0;   /* belirsiz */
8   return -1;              /* imkânsız */
9 }
```



sınır = 0,8284; U = 0,9375 sınırı aşıyor ama ≤ 1 → **0**: test karar veremez. Bu kümede RM gerçekten son tarih kaçırır, EDF karşılar.

Adım 9 / 9

DEĞİŞKENLER

`n = 2` `U = 0.9375` `sinir = 0.8284`

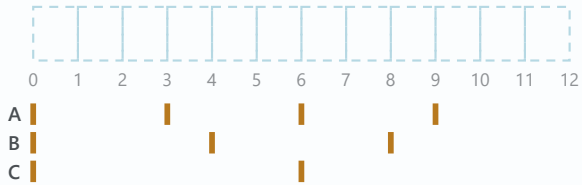
ÇIKTI

```
rm_test = 1
rm_test = 0
```

Test yeterli koşuldur: 1 dönerse RM kesin yeter; 0 dönerse karar için çizelge (ör. 0'dan en büyük periyoda kadar) çizilmelidir.

EDF Çizelgeleyici: Kod İzleme

```
▶ /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
2 for (int t = 0; t < 12; t++) {
3     for (int i = 0; i < n; i++)
4         if (t % P[i] == 0) {      /* yeni iş */
5             kalan[i] = C[i];
6             teslim[i] = t + P[i];
7         }
8     int s = -1;
9     for (int i = 0; i < n; i++)
10        if (kalan[i] > 0 &&
11            (s < 0 || teslim[i] < teslim[s]))
12            s = i;
13    if (s >= 0) kalan[s]--;     /* 1 ms */
14 }
```



Üç periyodik iş, U = 1/3 + 1/4 + 2/6 = **0,917** ≤ 1. Ortak periyot 12 ms; her milisaniye bir karar verilir. Oklar iş gelişlerini, `teslim` dizisi her işin son tarihini gösterir.

Adım 1 / 10

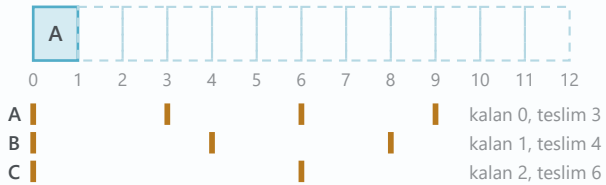
DEĞİŞKENLER

U = **0,917**

ÇIKTI

EDF Çizelgeleyici: Kod İzleme

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
   for (int t = 0; t < 12; t++) {
       for (int i = 0; i < n; i++)
           if (t % P[i] == 0) {      /* yeni iş */
               kalan[i] = C[i];
               teslim[i] = t + P[i];
7          }
       int s = -1;
       for (int i = 0; i < n; i++)
           if (kalan[i] > 0 &&
               (s < 0 || teslim[i] < teslim[s]))
               s = i;
       if (s >= 0) kalan[s]--;     /* 1 ms */
14 }
```



t = 0: A, B, C yeni periyoda başladı. Hazır: A, B, C; son tarihler A 3, B 4, C 6 → en yakını **A** çalışır.

Adım 2 / 10

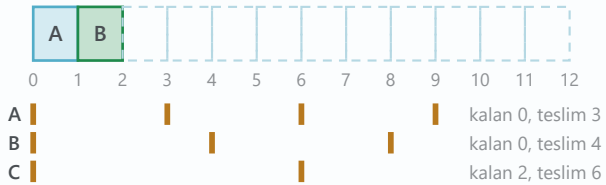
DEĞİŞKENLER

`t = 0` `kalan = [0,1,2]` `teslim = [3,4,6]` `s = A`

ÇIKTI

EDF Çizelgeleyici: Kod İzleme

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
▶  for (int t = 0; t < 12; t++) {
3      for (int i = 0; i < n; i++)
4          if (t % P[i] == 0) {      /* yeni iş */
5              kalan[i] = C[i];
6              teslim[i] = t + P[i];
7          }
▶      int s = -1;
▶      for (int i = 0; i < n; i++)
▶          if (kalan[i] > 0 &&
▶              (s < 0 || teslim[i] < teslim[s]))
▶              s = i;
▶      if (s >= 0) kalan[s]--;     /* 1 ms */
14 }
```



Hazır: B, C; son tarihler B 4, C 6 → en yakını **B** çalışır.

Adım 3 / 10

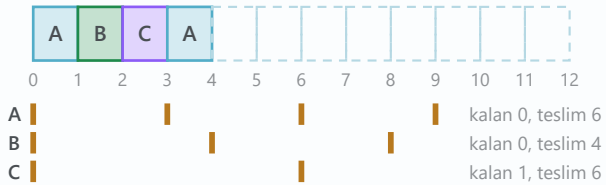
DEĞİŞKENLER

`t = 1` `kalan = [0,0,2]` `teslim = [3,4,6]` `s = B`

ÇIKTI

EDF Çizelgeleyici: Kod İzleme

```
/* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
for (int t = 0; t < 12; t++) {
    for (int i = 0; i < n; i++)
        if (t % P[i] == 0) {      /* yeni iş */
            kalan[i] = C[i];
            teslim[i] = t + P[i];
        }
    int s = -1;
    for (int i = 0; i < n; i++)
        if (kalan[i] > 0 &&
            (s < 0 || teslim[i] < teslim[s]))
            s = i;
    if (s >= 0) kalan[s]--;     /* 1 ms */
}
```



t = 2–3: C → A. Gelen: t = 3'te A. Her milisaniyede son tarihi en yakın hazır iş seçilir.

Adım 4 / 10

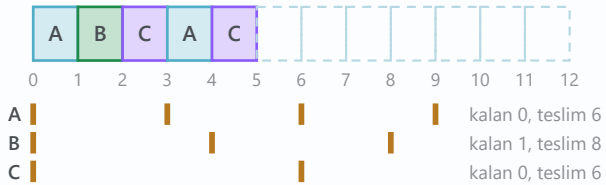
DEĞİŞKENLER

`t = 3` `kalan = [0,0,1]` `teslim = [6,4,6]` `s = A`

ÇIKTI

EDF Çizelgeleyici: Kod İzleme

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
   for (int t = 0; t < 12; t++) {
       for (int i = 0; i < n; i++)
           if (t % P[i] == 0) {      /* yeni iş */
               kalan[i] = C[i];
               teslim[i] = t + P[i];
7          }
       int s = -1;
       for (int i = 0; i < n; i++)
           if (kalan[i] > 0 &&
               (s < 0 || teslim[i] < teslim[s]))
               s = i;
       if (s >= 0) kalan[s]--;     /* 1 ms */
14 }
```



t = 4: B yeni periyoda başladı. Hazır: B, C; son tarihler B 8, C 6 → en yakını **C** çalışır.

Adım 5 / 10

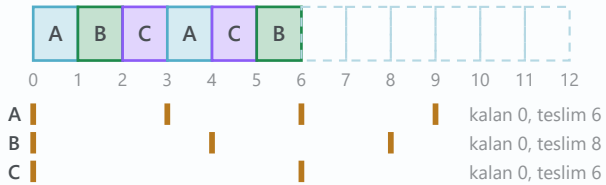
DEĞİŞKENLER

`t = 4` `kalan = [0,1,0]` `teslim = [6,8,6]` `s = C`

ÇIKTI

EDF Çizelgeleyici: Kod İzleme

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
▶  for (int t = 0; t < 12; t++) {
3      for (int i = 0; i < n; i++)
4          if (t % P[i] == 0) {      /* yeni iş */
5              kalan[i] = C[i];
6              teslim[i] = t + P[i];
7          }
▶      int s = -1;
▶      for (int i = 0; i < n; i++)
▶          if (kalan[i] > 0 &&
▶              (s < 0 || teslim[i] < teslim[s]))
▶              s = i;
▶      if (s >= 0) kalan[s]--;     /* 1 ms */
14 }
```



Hazır: B; son tarihi 8 → en yakını **B** çalışır.

Adım 6 / 10

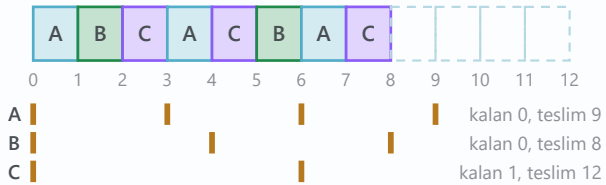
DEĞİŞKENLER

`t = 5` `kalan = [0,0,0]` `teslim = [6,8,6]` `s = B`

ÇIKTI

EDF Çizelgeleyici: Kod İzleme

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
▶  for (int t = 0; t < 12; t++) {
3      for (int i = 0; i < n; i++)
4          if (t % P[i] == 0) {      /* yeni iş */
5              kalan[i] = C[i];
6              teslim[i] = t + P[i];
7          }
▶      int s = -1;
▶      for (int i = 0; i < n; i++)
▶          if (kalan[i] > 0 &&
▶              (s < 0 || teslim[i] < teslim[s]))
▶              s = i;
▶      if (s >= 0) kalan[s]--;     /* 1 ms */
14 }
```



t = 6–7: A → C. Gelen: t = 6'da A, C. Her milisaniyede son tarihi en yakın hazır iş seçilir.

Adım 7 / 10

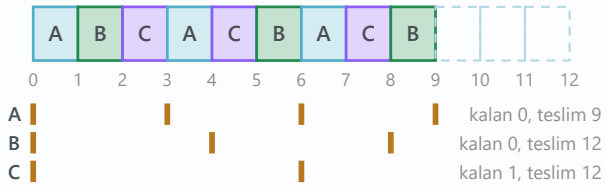
DEĞİŞKENLER

`t = 7` `kalan = [0,0,1]` `teslim = [9,8,12]` `s = C`

ÇIKTI

EDF Çizelgeleyici: Kod İzleme

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
   for (int t = 0; t < 12; t++) {
       for (int i = 0; i < n; i++)
           if (t % P[i] == 0) {      /* yeni iş */
               kalan[i] = C[i];
               teslim[i] = t + P[i];
7          }
       int s = -1;
       for (int i = 0; i < n; i++)
           if (kalan[i] > 0 &&
               (s < 0 || teslim[i] < teslim[s]))
               s = i;
       if (s >= 0) kalan[s]--;     /* 1 ms */
14 }
```



t = 8: B yeni periyoda başladı. Hazır: B, C; son tarihler B 12, C 12 → en yakını **B** çalışır.

Adım 8 / 10

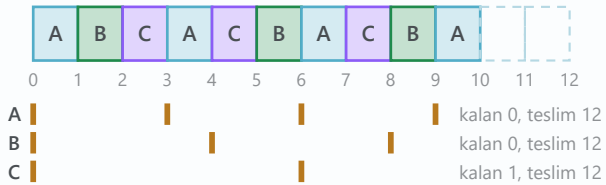
DEĞİŞKENLER

`t = 8` `kalan = [0,0,1]` `teslim = [9,12,12]` `s = B`

ÇIKTI

EDF Çizelgeleyici: Kod İzleme

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
   for (int t = 0; t < 12; t++) {
       for (int i = 0; i < n; i++)
           if (t % P[i] == 0) {      /* yeni iş */
               kalan[i] = C[i];
               teslim[i] = t + P[i];
7          }
       int s = -1;
       for (int i = 0; i < n; i++)
           if (kalan[i] > 0 &&
               (s < 0 || teslim[i] < teslim[s]))
               s = i;
       if (s >= 0) kalan[s]--;     /* 1 ms */
14 }
```



t = 9: A yeni periyoda başladı. Hazır: A, C; son tarihler A 12, C 12 → en yakını **A** çalışır.

Adım 9 / 10

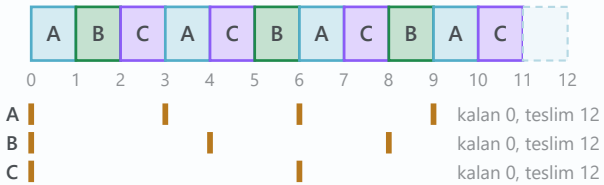
DEĞİŞKENLER

`t = 9` `kalan = [0,0,1]` `teslim = [12,12,12]` `s = A`

ÇIKTI

EDF Çizelgeleyici: Kod İzleme

```
1  /* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
▶  for (int t = 0; t < 12; t++) {
3      for (int i = 0; i < n; i++)
4          if (t % P[i] == 0) {      /* yeni iş */
5              kalan[i] = C[i];
6              teslim[i] = t + P[i];
7          }
▶      int s = -1;
▶      for (int i = 0; i < n; i++)
▶          if (kalan[i] > 0 &&
▶              (s < 0 || teslim[i] < teslim[s]))
▶              s = i;
▶      if (s >= 0) kalan[s]--;     /* 1 ms */
14 }
```



t = 10–11: C → boş. 12 ms'de 11 ms iş: 1 ms boşluk (1 − 0,917). Hiçbir iş son tarihini kaçırmadı.

Adım 10 / 10

DEĞİŞKENLER

`t = 11` `kalan = [0,0,0]` `teslim = [12,12,12]` `s = -1`

ÇIKTI

POSIX Gerçek Zamanlı Çizelgeleme

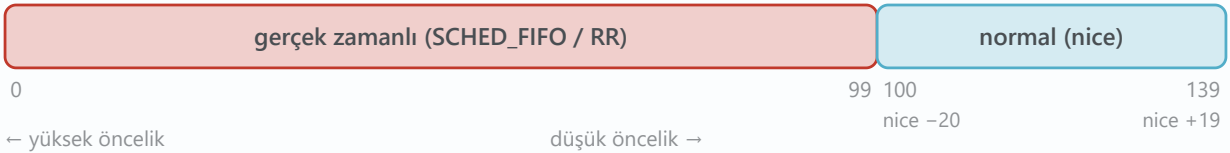
```
pthread_attr_t attr;
struct sched_param sp;
int policy;
pthread_attr_init(&attr);
pthread_attr_getschedpolicy(&attr, &policy);
/* kalıtım kapatılmazsa politika yok sayılır */
pthread_attr_setinheritsched(&attr,
                     PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
sp.sched_priority = 50;          /* 1..99 */
pthread_attr_setschedparam(&attr, &sp);
pthread_create(&tid, &attr, runner, NULL);
```

Politika	Davranış
`SCHED_FIFO`	Sabit öncelik (1–99), dilim yok; bloke olana ya da bırakana kadar.
`SCHED_RR`	FIFO + eşit öncelikliler arasında dilim.
`SCHED_OTHER`	Varsayılan (Linux'ta EEVDF); ayrıca `BATCH`, `IDLE`.
`SCHED_DEADLINE`	Linux: çalışma / son tarih / periyot ile EDF.

- POSIX.1b uzantısı; ayrıcalık ister (`CAP_SYS_NICE`), kabukta `chrt -f 50 ./prog` . Linux RT işlere saniyenin en fazla 0,95 s'sini verir (`sched_rt_runtime_us`).

İşlev adları `pthread_attr_getschedpolicy` ve `pthread_attr_setschedpolicy` 'dir: "sched" ile "policy" arasında alt çizgi yoktur.

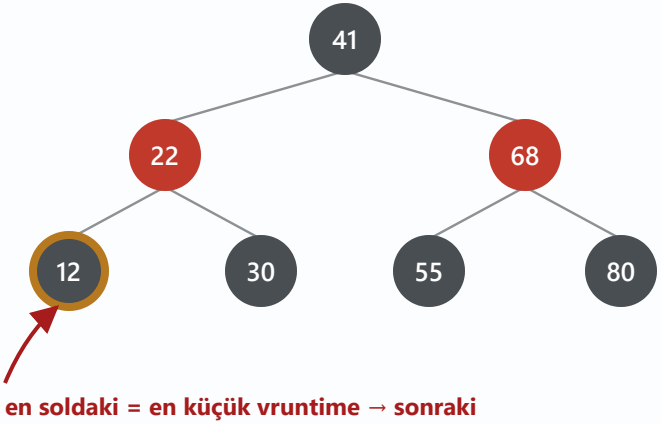
Linux: Çizelgeleme Sınıfları



Sınıf (öncelik sırasıyla)	Politikalar	Seçim
`stop`	— (çekirdek içi)	CPU durdurma, göç; her şeyi keser
`deadline`	`SCHED_DEADLINE`	EDF + CBS
`rt`	`SCHED_FIFO` , `SCHED_RR`	100 öncelik düzeyli dizi + biteşlem, O(1)
`fair`	`SCHED_NORMAL` , `SCHED_BATCH` , `SCHED_IDLE`	CFS (2.6.23–6.5), **EEVDF** (6.6+)
`ext`	`SCHED_EXT`	BPF ile yazılan çizelgeleyici (sched_ext, 6.12+)
`idle`	—	hiç iş yoksa boşta döngüsü

Çekirdek öncelik ölçeği (küçük sayı yüksek): 0–99 gerçek zamanlı (99 − rt_priority), 100–139 adil (nice −20 … 19 → 100 … 139). `SCHED_RR` dilimi 100 ms.

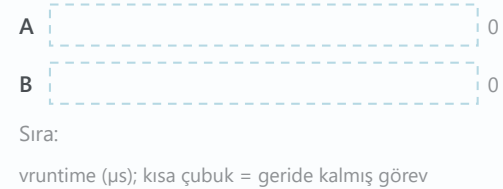
Linux CFS: Sanal Çalışma Süresi



- **Completely Fair Scheduler** (2.6.23–6.5): sabit dilim yerine her görevin **sanal çalışma süresi** (vruntime) tutulur; en küçüğü çalışır.
- vruntime += Δ × 1024 / ağırlık. Ağırlık nice'tan gelir: nice 0 → 1024, nice 5 → 335, nice −5 → 3121; her nice adımı ≈ ×1,25 (≈ %10 CPU farkı).
- Örnek: CPU'ya bağlı A (nice 0) ve B (nice 5): A 1024 / 1359 ≈ **%75**, B ≈ **%25** alır. B 10 ms çalışınca vruntime'ı 10 × 1024 / 335 ≈ 30,6 ms artar.
- Uyuyan G/Ç'ye bağlı görevin vruntime'ı az artar; uyanınca hemen öne geçer (yine de geride kalmışlara sınırlı avantaj verilir).
- Ağaç: ekleme O(log n), en soldaki önbellekte → seçim O(1).

CFS: vruntime Kod İzleme

```
#define DELTA 3000  /* 3 ms, µs olarak */
/* nice 0 → 1024, nice 5 → 335 */
void tik(void) {
  Gorev *g = en_soldaki(); /* en solda */
  calistir(g, DELTA);
  g->vruntime += DELTA * 1024 / g->agirlik;
  agaca_ekle(g);      /* yeniden sırala */
}
```



A (nice 0, ağırlık 1024) ve B (nice 5, ağırlık 335) CPU'ya bağlı; ikisinin vruntime'ı 0. Eşitlikte ağaçta önce eklenen A soldadır.

Adım 1 / 11

DEĞİŞKENLER

`A.vruntime = 0` `B.vruntime = 0`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine "zamanlayıcı gecikmesi" kullanır; seçim kuralı aynıdır.

CFS: vruntime Kod İzleme

```
1 #define DELTA 3000  /* 3 ms, µs olarak */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tik(void) {
▶   Gorev *g = en_soldaki(); /* en solda */
▶   calistir(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->agirlik;
7   agaca_ekle(g);       /* yeniden sırala */
8 }
```



A'nın vruntime'ı (0) B'ninkinden (0) küçük ya da eşit → A çalışır; 3000 × 1024 / 1024 = **3000** eklenir.

Adım 2 / 11

DEĞİŞKENLER

`g = A` `A.vruntime = 3000` `B.vruntime = 0`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine "zamanlayıcı gecikmesi" kullanır; seçim kuralı aynıdır.

CFS: vruntime Kod İzleme

```
1 #define DELTA 3000  /* 3 ms, µs olarak */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tik(void) {
▶   Gorev *g = en_soldaki(); /* en solda */
▶   calistir(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->agirlik;
7   agaca_ekle(g);      /* yeniden sırala */
8 }
```



Artık en solda B (0 < 3.000) → B çalışır; 3000 × 1024 / 335 = **9170** eklenir: düşük ağırlık, hızlı "yaşlanma".

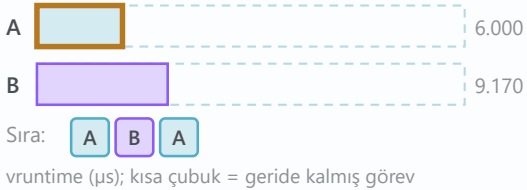
DEĞİŞKENLER

`g = B` `A.vruntime = 3000` `B.vruntime = 9170`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine "zamanlayıcı gecikmesi" kullanır; seçim kuralı aynıdır.

CFS: vruntime Kod İzleme

```
1 #define DELTA 3000  /* 3 ms, µs olarak */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tik(void) {
▶   Gorev *g = en_soldaki(); /* en solda */
▶   calistir(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->agirlik;
7   agaca_ekle(g);       /* yeniden sırala */
8 }
```



A'nın vruntime'ı (3.000) B'ninkinden (9.170) küçük ya da eşit → A çalışır; 3000 × 1024 / 1024 = **3000** eklenir.

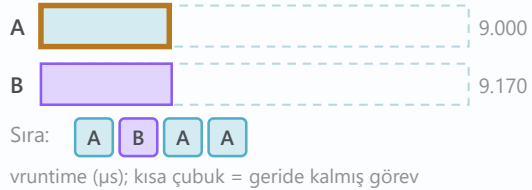
DEĞİŞKENLER

`g = A` `A.vruntime = 6000` `B.vruntime = 9170`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine "zamanlayıcı gecikmesi" kullanır; seçim kuralı aynıdır.

CFS: vruntime Kod İzleme

```
1 #define DELTA 3000  /* 3 ms, µs olarak */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tik(void) {
▶   Gorev *g = en_soldaki(); /* en solda */
▶   calistir(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->agirlik;
7   agaca_ekle(g);      /* yeniden sırala */
8 }
```



A yine en solda (6.000 ≤ 9.170) → A çalışır, +3000.

Adım 5 / 11

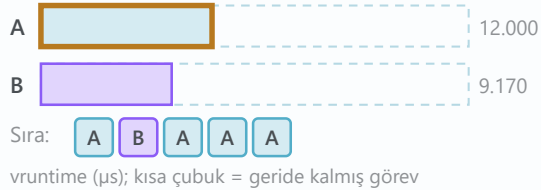
DEĞİŞKENLER

`g = A` `A.vruntime = 9000` `B.vruntime = 9170`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine “zamanlayıcı gecikmesi” kullanır; seçim kuralı aynıdır.

CFS: vruntime Kod İzleme

```
1 #define DELTA 3000  /* 3 ms, µs olarak */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tik(void) {
▶   Gorev *g = en_soldaki(); /* en solda */
▶   calistir(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->agirlik;
7   agaca_ekle(g);      /* yeniden sırala */
8 }
```



A yine en solda (9.000 ≤ 9.170) → A çalışır, +3000.

Adım 6 / 11

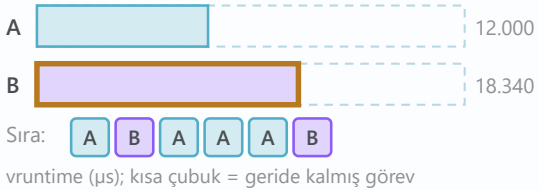
DEĞİŞKENLER

`g = A` `A.vruntime = 12000` `B.vruntime = 9170`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine “zamanlayıcı gecikmesi” kullanır; seçim kuralı aynıdır.

CFS: vruntime Kod İzleme

```
1 #define DELTA 3000  /* 3 ms, µs olarak */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tik(void) {
▶   Gorev *g = en_soldaki(); /* en solda */
▶   calistir(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->agirlik;
7   agaca_ekle(g);      /* yeniden sırala */
8 }
```



Artık en solda B (9.170 < 12.000) → B çalışır; 3000 × 1024 / 335 = **9170** eklenir: düşük ağırlık, hızlı "yaşlanma".

Adım 7 / 11

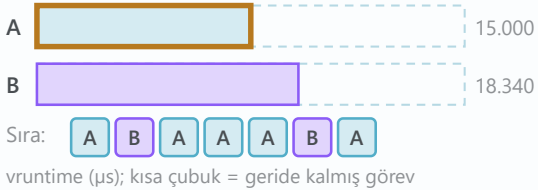
DEĞİŞKENLER

`g = B` `A.vruntime = 12000` `B.vruntime = 18340`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine "zamanlayıcı gecikmesi" kullanır; seçim kuralı aynıdır.

CFS: vruntime Kod İzleme

```
1 #define DELTA 3000  /* 3 ms, µs olarak */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tik(void) {
▶   Gorev *g = en_soldaki(); /* en solda */
▶   calistir(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->agirlik;
7   agaca_ekle(g);      /* yeniden sırala */
8 }
```



A'nın vruntime'ı (12.000) B'ninkinden (18.340) küçük ya da eşit → A çalışır; 3000 × 1024 / 1024 = **3000** eklenir.

Adım 8 / 11

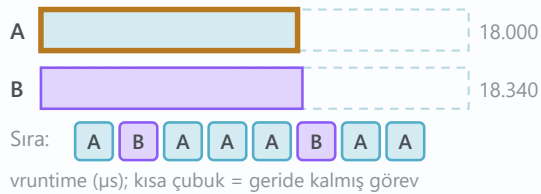
DEĞİŞKENLER

`g = A` `A.vruntime = 15000` `B.vruntime = 18340`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine "zamanlayıcı gecikmesi" kullanır; seçim kuralı aynıdır.

CFS: vruntime Kod İzleme

```
1 #define DELTA 3000  /* 3 ms, µs olarak */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tik(void) {
▶   Gorev *g = en_soldaki(); /* en solda */
▶   calistir(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->agirlik;
7   agaca_ekle(g);      /* yeniden sırala */
8 }
```



A yine en solda (15.000 ≤ 18.340) → A çalışır, +3000.

Adım 9 / 11

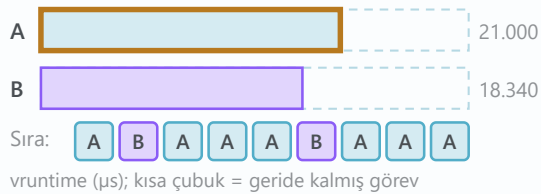
DEĞİŞKENLER

`g = A` `A.vruntime = 18000` `B.vruntime = 18340`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine “zamanlayıcı gecikmesi” kullanır; seçim kuralı aynıdır.

CFS: vruntime Kod İzleme

```
1 #define DELTA 3000  /* 3 ms, µs olarak */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tik(void) {
▶   Gorev *g = en_soldaki(); /* en solda */
▶   calistir(g, DELTA);
▶   g->vruntime += DELTA * 1024 / g->agirlik;
7   agaca_ekle(g);      /* yeniden sırala */
8 }
```



A yine en solda (18.000 ≤ 18.340) → A çalışır, +3000.

Adım 10 / 11

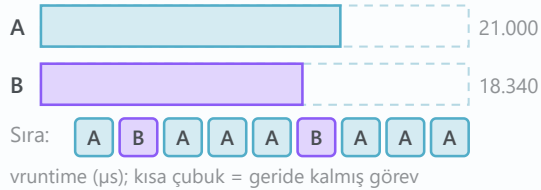
DEĞİŞKENLER

`g = A` `A.vruntime = 21000` `B.vruntime = 18340`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine “zamanlayıcı gecikmesi” kullanır; seçim kuralı aynıdır.

CFS: vruntime Kod İzleme

```
1 #define DELTA 3000  /* 3 ms, µs olarak */
2 /* nice 0 → 1024, nice 5 → 335 */
3 void tik(void) {
4   Gorev *g = en_soldaki(); /* en solda */
5   calistir(g, DELTA);
6   g->vruntime += DELTA * 1024 / g->agirlik;
▶   agaca_ekle(g);       /* yeniden sırala */
8 }
```



9 tıkta A 7, B 2 kez çalıştı; uzun vadede pay 1024 : 335 ağırlık oranına (≈ %75 – %25) yaklaşır. Sabit dilim yok; pay ağırlıktan çıkar.

Adım 11 / 11

DEĞİŞKENLER

`A.vruntime = 21000`

`B.vruntime = 18340`

`A = 7 tık`

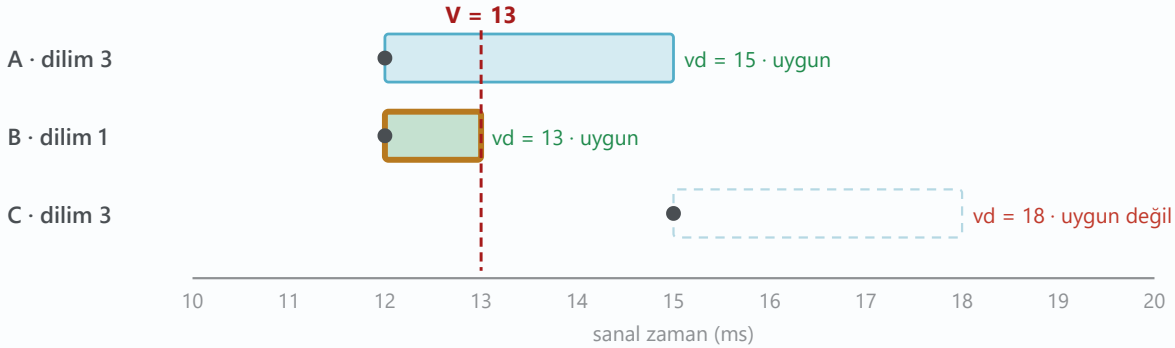
`B = 2 tık`

Basitleştirilmiş tık işleyicisi: gerçek CFS süreyi ns ile ölçer ve dilim yerine “zamanlayıcı gecikmesi” kullanır; seçim kuralı aynıdır.

İŞLETİM SİSTEMİ ÖRNEKLERİ

EEVDF: CFS'in Halefi

EEVDF (Earliest Eligible Virtual Deadline First, Stoica 1995; Linux 6.6+): görev, payından geride ya da tam payındaysa (gecikme lag ≥ 0, yani v ≤ V) **uygundur**; sanal son tarih vd = v + dilim / ağırlık. Uygunlar içinde vd en erken olan çalışır. Üç eşit ağırlıklı görev:



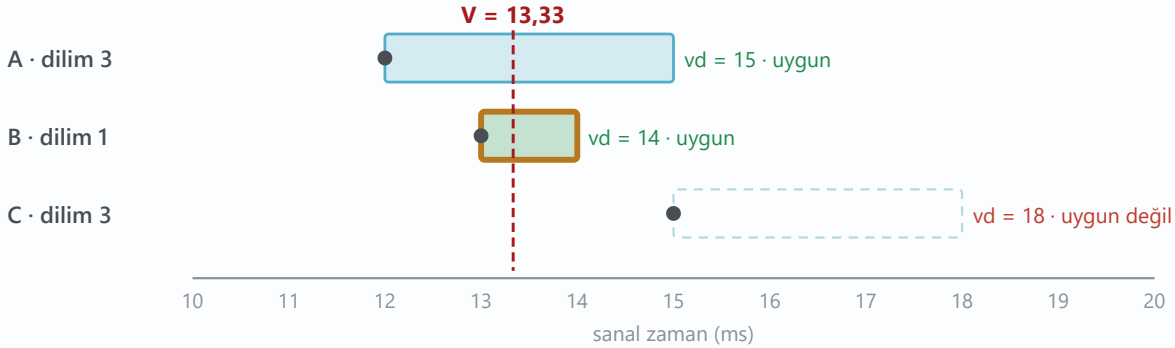
v: A 12, B 12, C 15 → V = ortalama = **13**. C önde (v > V, negatif gecikme): uygun değil. A ve B uygun; vd: A 15, B 13 → **B** (kısa dilim, erken son tarih).

Adım 1 / 5

CFS'teki gecikme sezgileri yerine iki net ölçüt (uygunluk ve son tarih) gelir; gecikmeye duyarlı görev `sched_setattr()` ile kısa dilim isteyebilir (6.12+). Ağırlıklar nice ile CFS'tekiyle aynıdır.

EEVDF: CFS'in Halefi

EEVDF (Earliest Eligible Virtual Deadline First, Stoica 1995; Linux 6.6+): görev, payından geride ya da tam payındaysa (gecikme lag ≥ 0, yani v ≤ V) **uygundur**; sanal son tarih vd = v + dilim / ağırlık. Uygunlar içinde vd en erken olan çalışır. Üç eşit ağırlıklı görev:

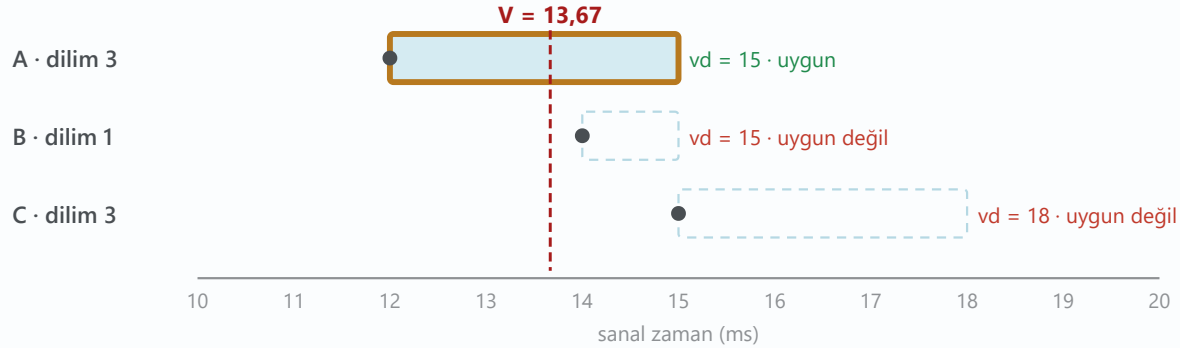


B 1 ms çalıştı → v = 13, yeni vd = 14. V = 13,33; uygun: A ve B → en erken vd **B**.

CFS'teki gecikme sezgileri yerine iki net ölçüt (uygunluk ve son tarih) gelir; gecikmeye duyarlı görev `sched_setattr()` ile kısa dilim isteyebilir (6.12+). Ağırlıklar nice ile CFS'tekiyle aynıdır.

EEVDF: CFS'in Halefi

EEVDF (Earliest Eligible Virtual Deadline First, Stoica 1995; Linux 6.6+): görev, payından geride ya da tam payındaysa (gecikme lag ≥ 0, yani v ≤ V) **uygundur**; sanal son tarih vd = v + dilim / ağırlık. Uygunlar içinde vd en erken olan çalışır. Üç eşit ağırlıklı görev:

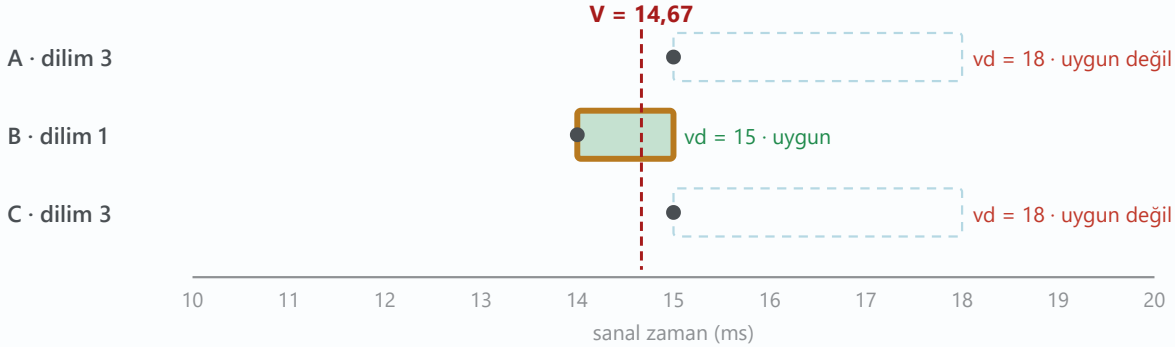


B 14'e ilerledi; V = 13,67. Artık yalnızca **A** uygun (B ve C ortalamanın önünde) → A 3 ms'lik dilimini çalışır.

CFS'teki gecikme sezgileri yerine iki net ölçüt (uygunluk ve son tarih) gelir; gecikmeye duyarlı görev `sched_setattr()` ile kısa dilim isteyebilir (6.12+). Ağırlıklar nice ile CFS'tekiyle aynıdır.

EEVDF: CFS'in Halefi

EEVDF (Earliest Eligible Virtual Deadline First, Stoica 1995; Linux 6.6+): görev, payından geride ya da tam payındaysa (gecikme lag ≥ 0, yani v ≤ V) **uygundur**; sanal son tarih vd = v + dilim / ağırlık. Uygunlar içinde vd en erken olan çalışır. Üç eşit ağırlıklı görev:

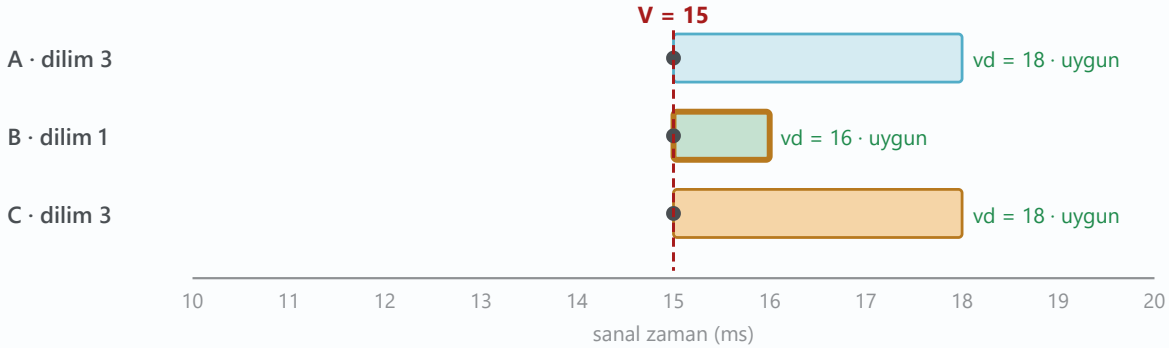


A 3 ms çalıştı → v = 15, yeni vd = 18. V = 14,67; uygun yalnızca B (v = 14) → **B**.

CFS'teki gecikme sezgileri yerine iki net ölçüt (uygunluk ve son tarih) gelir; gecikmeye duyarlı görev `sched_setattr()` ile kısa dilim isteyebilir (6.12+). Ağırlıklar nice ile CFS'tekiyle aynıdır.

EEVDF: CFS'in Halefi

EEVDF (Earliest Eligible Virtual Deadline First, Stoica 1995; Linux 6.6+): görev, payından geride ya da tam payındaysa (gecikme lag ≥ 0, yani v ≤ V) **uygundur**; sanal son tarih vd = v + dilim / ağırlık. Uygunlar içinde vd en erken olan çalışır. Üç eşit ağırlıklı görev:



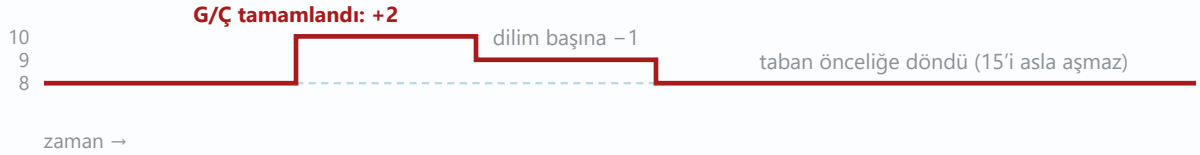
V = 15: üçü de uygun. vd: A 18, B 16, C 18 → **B**. Kısa dilim isteyen B daha sık ama kısa çalışır; uygunluk koşulu sayesinde uzun vadede payı yine eşittir.

CFS'teki gecikme sezgileri yerine iki net ölçüt (uygunluk ve son tarih) gelir; gecikmeye duyarlı görev `sched_setattr()` ile kısa dilim isteyebilir (6.12+). Ağırlıklar nice ile CFS'tekiyle aynıdır.

Windows Çizelgeleme ve Öncelikleri

Göreli öncelik	REALTIME	HIGH	ABOVE_NORMAL	NORMAL	BELOW_NORMAL	IDLE
zaman-kritik	31	15	15	15	15	15
en yüksek	26	15	12	10	8	6
normalin üstü	25	14	11	9	7	5
normal	**24**	**13**	**10**	**8**	**6**	**4**
normalin altı	23	12	9	7	5	3
en düşük	22	11	8	6	4	2
boşta	16	1	1	1	1	1

- Kesmeli, öncelik tabanlı: en yüksek öncelikli hazır iş parçacığı çalışır; bloke olana, dilimi bitene ya da daha yüksek öncelikli biri gelene kadar. 32 düzey: **1–15 değişken**, **16–31 gerçek zamanlı**, 0 sayfa sıfırlama (zero-page) iş parçacığı. Her düzey için bir kuyruk; iş yoksa boşta (idle) iş parçacığı.
- Değişken sınıfta **dinamik artırma**: G/Ç tamamlanınca (klavye/fare için daha çok), ön plan penceresi ve GUI uyanmalarında öncelik geçici yükselir, her dilimde 1 azalır, 15'i aşmaz; ≈ 4 saniye aç kalan iş parçacığı bir dilimlik 15'e çıkarılır. Dilim istemcide ≈ 2, sunucuda 12 saat tıkı.



Algoritmaları Değerlendirme

Deterministik modelleme

Önceden verilen bir iş yükü için her algoritmanın sonucu hesaplanır (bu bölümdeki Gantt şemaları). Kesin ve basittir ama yalnızca o girdi için geçerlidir.

Kuyruk modelleri

Geliş ve hizmet dağılımlarından ortalamalar. **Little yasası**: $n = \lambda \times W$. Saniyede $\lambda = 7$ süreç gelir, her biri sistemde ortalama $W = 2$ s kalırsa sistemde ortalama $n = 14$ süreç vardır.

Simülasyon

Sistemin programlanmış modeli, rastgele üretilmiş ya da gerçek sistemden kaydedilmiş **iz** (trace) verisiyle çalıştırılır. Doğrudur ama saatler sürebilir.

Gerçekleme

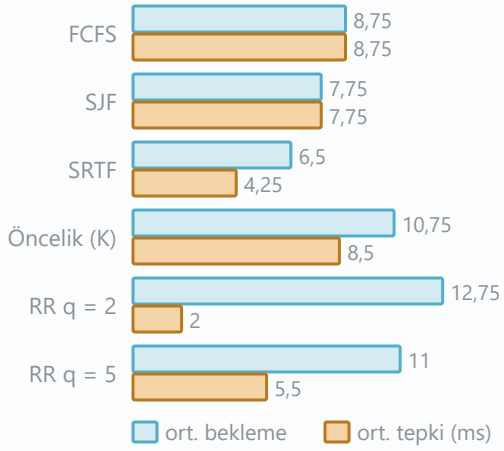
Algoritma gerçek çekirdekte denenir. Bugün **sched_ext** ile BPF çizelgeleyicileri çekirdeği yeniden derlemeden yüklenip üretimde A/B test edilebiliyor. Bedeli: kullanıcı davranışı da algoritmaya uyum sağlar.



Ortak Kümede Karşılaştırma

Algoritma	Bekleme	Dönüş	Tepki	Geçiş
FCFS	8,75	15,25	8,75	3
SJF	7,75	14,25	7,75	3
SRTF	6,5	13	4,25	4
Öncelik (K)	10,75	17,25	8,5	4
RR q = 2	12,75	19,25	2	12
RR q = 5	11	17,5	5,5	5

Ortalamalar ms cinsinden; K = kesmeli; geçiş = bağlam değiştirme sayısı.



- **SRTF** ortalama beklemede en iyisidir (6,5) ama sürelerin bilinmesini ister; **RR q = 2** tepkide en iyisidir (2) ama en çok geçişi yapar.
- Öncelik bekleme süresini değil **önemi** optimize eder. Tek bir “en iyi” algoritma yoktur: seçim, hedefe (tepki, verim, adalet, son tarih) bağlıdır.

Alıştırmalar: Dönüş, Yanıt ve Adalet

1. A, B, C üçü de t = 0'da gelir ve her biri 5 ms CPU ister. SJF ile RR (q = 1 ms) için ortalama dönüş ve ortalama tepki süresi nedir?
2. A (geliş 0, süre 100), B (10, 10), C (10, 10). Kesmesiz SJF ve STCF (SRTF) için ortalama dönüş ve tepki süresi nedir?
3. Adaletsizlik ölçütü U = ilk bitenin bitiş zamanı / ikinci bitenin bitiş zamanı. Eşit biletli iki iş, her biri L dilim istiyor. RR ile U kaçtır? Piyangoda L büyüdükçe U nasıl değişir?

▶ **Cevaplar**

Kontrol Soruları (1/2)

1. Kesmesiz bir çizelgeleyici hangi durum geçişlerinde karar verir?
2. P1 = 5, P2 = 2, P3 = 1 ms, hepsi t = 0'da ve bu sırayla geliyor. FCFS ve SJF için ortalama bekleme nedir?
3. $\alpha = 0{,}5$ ve $\tau_0 = 8$. Ölçülen patlamalar 4 ve 12 ms ise sonraki tahmin kaçtır?
4. RR'de zaman dilimi çok büyük ya da çok küçük seçilirse ne olur? q = 4 ms, geçiş 1 ms ise kayıp oranı nedir?

▶ **Cevaplar**

Kontrol Soruları (2/2)

5. Ortak kümede SRTF, P1'i neden t = 1'de keser?
6. Çekirdek başına kuyruklarda CPU 1 boşaldı; CPU 0, 2 ve 3'te 3, 5 ve 2 iş var. Çekme dengelemesi nereden kaç iş taşır, bedeli nedir?
7. P1 (p = 40, t = 10) ve P2 (p = 60, t = 30) RM ile çizelgelenebilir mi?
8. CFS'te nice 0 ve nice 5 olan iki CPU'ya bağlı görev CPU'yu nasıl paylaşır? EEVDF'de "uygun" görev ne demektir?

▶ **Cevaplar**

Kaynaklar

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 5: CPU Scheduling.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 2.4: Scheduling.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — Bölüm 7–10: Scheduling, MLFQ, Lottery, Multiprocessor.
- P. Yosifovich, A. Ionescu, M. Russinovich, D. Solomon, *Windows Internals*, 7. baskı, Kısım 1, 2017 — Bölüm 4: Thread scheduling.
- Linux çekirdek belgeleri: `sched-eevdf.rst`, `sched-design-CFS.rst`, `sched-deadline.rst`, `sched-ext.rst`; `sched(7)` kılavuz sayfası.
- I. Stoica, H. Abdel-Wahab, "Earliest Eligible Virtual Deadline First", 1995; C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *JACM*, 1973; C. A. Waldspurger, W. E. Weihl, "Lottery Scheduling", OSDI 1994.

Son

Bölüm 5: CPU Çizelgeleme

Sonraki bölüm: Senkronizasyon · Terim sözlüğü