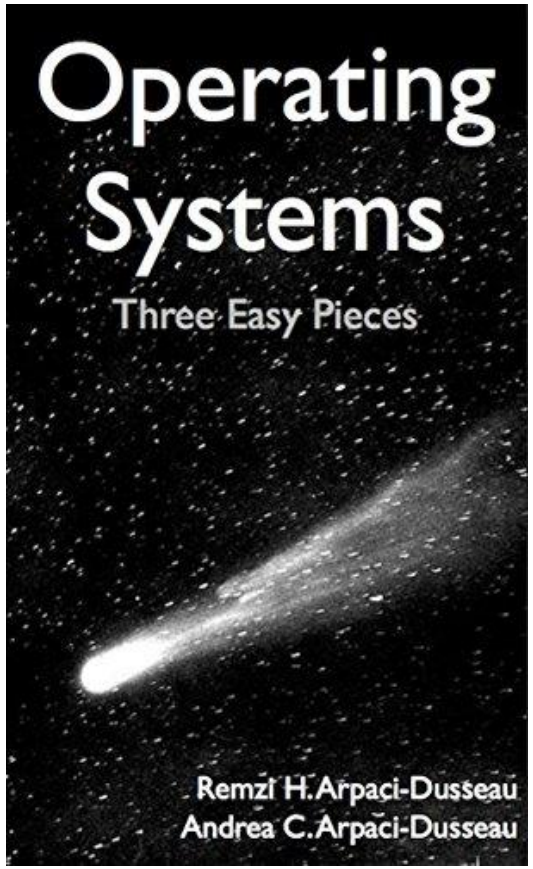
İşletim Sistemleri



3. Ders

Prof. Dr. Kemal Bıçakcı

Alıştırma Sorusu

- Aşağıdaki işlem durum geçişlerinden hangisi mümkün değildir?

a) Hazır → Çalışan
b) Bloklanmış → Çalışan
c) Çalışan → Hazır
d) Çalışan → Bloklanmış

7. Zamanlama (Scheduling): Giriş

İşletim Sistemleri: Üç Basit Parça

Zamanlama: Genel ve Önemli bir Problem

Başka hangi alanlarda benzer bir zamanlama problemi üzerinde çalışılmaktadır?

Zamanlama: Giriş

- (Başlangıçtaki) iş yükü varsayımlarımız:
  1. Her iş aynı süre boyunca çalışır.
  2. Tüm işler aynı anda gelir.
  3. Tüm işler yalnızca CPU'yu kullanır (I/O gerçekleştirmezler).
  4. Her işin çalışma süresi önceden bilinmektedir.

Zamanlama Ölçütleri (Scheduling Metrics)

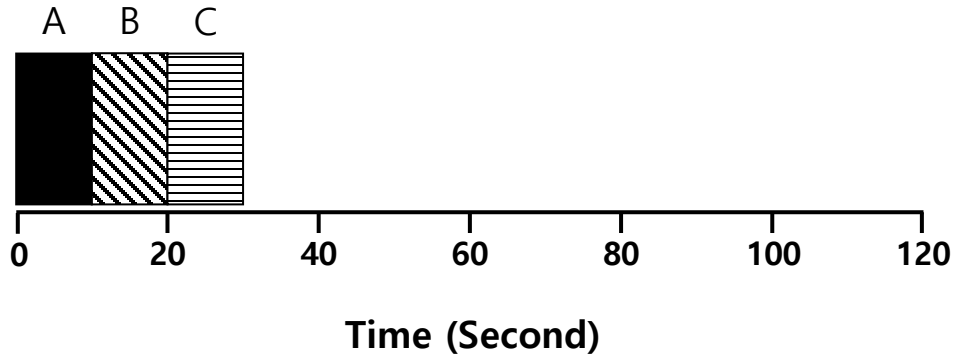
- Performans ölçütü: Geri dönüş süresi (Turnaround time)
  - İşin tamamlandığı saat - İşin sisteme ulaştığı saat.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

- Diğer bir ölçüt: adalet (fairness).
  - Performans ve adalet ölçütleri, zamanlama bağlamında, genellikle birbiriyle çelişir.

İlk Giren İlk Çıkar
First In, First Out (FIFO)

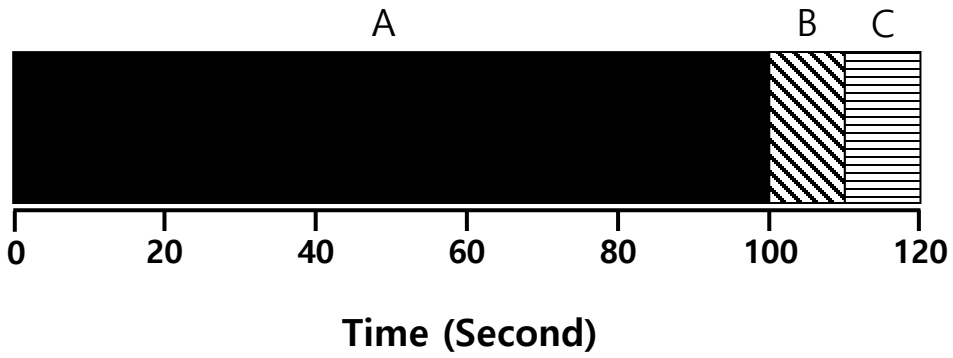
- İlk Gelen, İlk Hizmet Alır - First Come, First Served (FCFS)
  - Basit ve kolay gerçekleştirilebilir
- Örnek:
  - A, B ve C neredeyse aynı anda gelir.
  - Her iş 10 saniye çalışır.



$$Average\ turnaround\ time = \frac{10 + 20 + 30}{3} = 20\ sec$$

FIFO'nun problemi nedir?
Konvoy etkisi

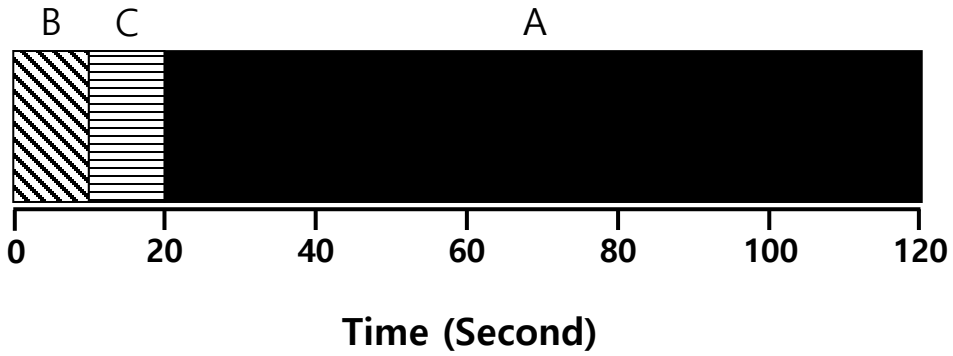
- Varsayım-1'i gevşetelim: Artık her iş aynı süre boyunca çalışmıyor olsun.
- Örnek:
  - A, B ve C yine neredeyse aynı anda gelmiş ve A burun farkıyla en önce olsun.
  - A, 100 saniye; B ve C'nin her birisi 10 saniye çalışıyor olsun.



$$Average\ turnaround\ time = \frac{100 + 110 + 120}{3} = 110\ sec$$

Önce En Kısa İş
Shortest Job First (SJF)

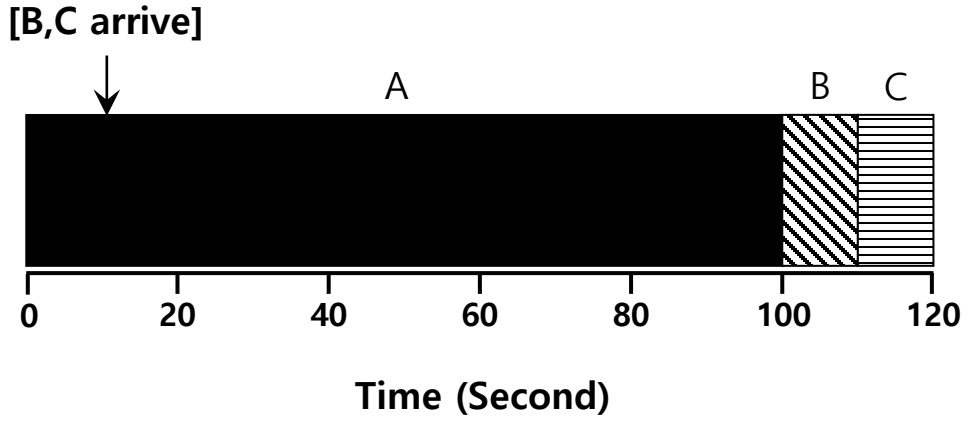
- Önce en kısa işi çalıştıralım, sonra bir sonraki en kısa işi, vb.
  - Boşaltmasız Zamanlayıcı (Non-preemptive scheduler)
- Örnek:
  - A, B ve C aynı anda gelsin.
  - A, 100 saniye; B ve C'nin her birisi 10 saniye çalışıyor olsun.



$$Average\ turnaround\ time = \frac{10 + 20 + 120}{3} = 50\ sec$$

B ve C daha geç gelirse?

- Varsayım-2'yi de gevşetelim. İşler herhangi bir zaman gelebiliyor olsun.
- Örnek:
  - A, t=0'da gesin ve 100 saniye çalışsın.
  - B ve C, t=10'da gelsinler ve 10 saniye çalışsınlar.



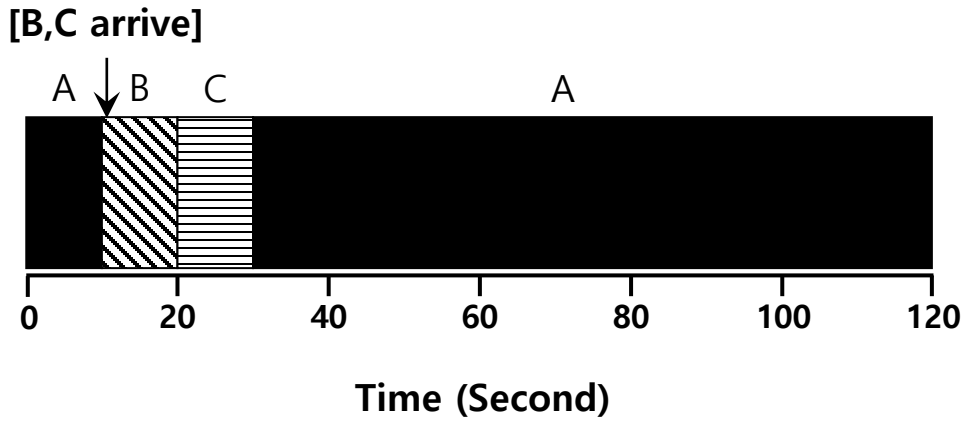
$$Average\ turnaround\ time = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33\ sec$$

Önce En Kısa Tamamlanma Süresi
Shortest Time-to-Completion First (STCF)

- Önce En Kısa İş yöntemine boşaltma (preemption) ekleyelim.
  - Aynı zamanda Boşaltmalı En Kısa İş olarak bilinir
- Yeni bir iş geldiğinde:
  - Yeni gelen işin çalışma süresini ve diğer işlerin tamamlanma sürelerini karşılaştır
  - Hangi süre en az ise o işi zamanla

Önce En Kısa Tamamlanma Süresi

- Örnek:
  - A, t=0'da gesin ve 100 saniye çalışsın.
  - B ve C, t=10'da gelsinler ve 10 saniye çalışsınlar.



$$\textbf{\textit{Average turnaround time}} = \frac{(120-0)+(20-10)+(30-10)}{3} = 50\ sec$$

Yeni bir zamanlama ölçütü: Tepki süresi (Response time)

- İşin gelmesi ile ilk kez zamanlanması arasında geçen süre.

$$T_{response} = T_{firstrun} - T_{arrival}$$

- Önce En Kısa Tamamlanma Süresi ve benzer zamanlama yöntemleri Tepki Süresi ölçütü ile değerlendirildiklerinde iyi sonuç vermezler

Tepki süresi ölçütünde iyi olan zamanlama yöntemleri nelerdir?

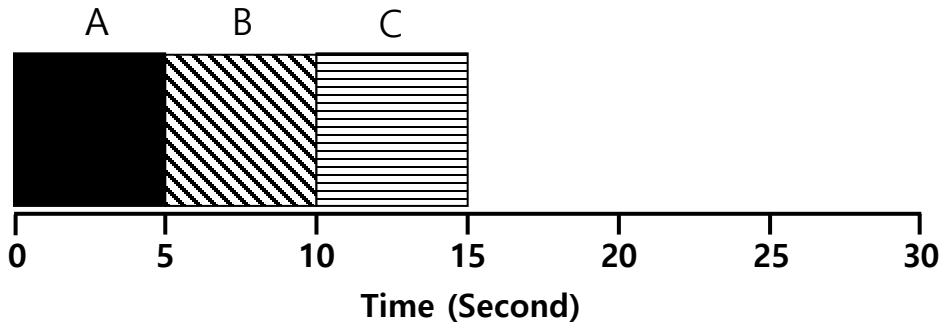
Round Robin (RR) Zamanlama

- Dilimlemeli Zamanlama
  - Bir işi bir zaman dilimi için çalıştırın ve ardından tüm işler bitene kadar çalıştırma kuyruğundaki bir sonraki işe geçin.
    - Zaman dilimine bazen çizelgeleme kuantumu (scheduling quantum) da denir.
  - Bir zaman diliminin uzunluğu, saat kesmesi süresinin tam katı olmalıdır.

RR adildir,
fakat geri dönüş süresi gibi ölçütlere göre performansı düşüktür.

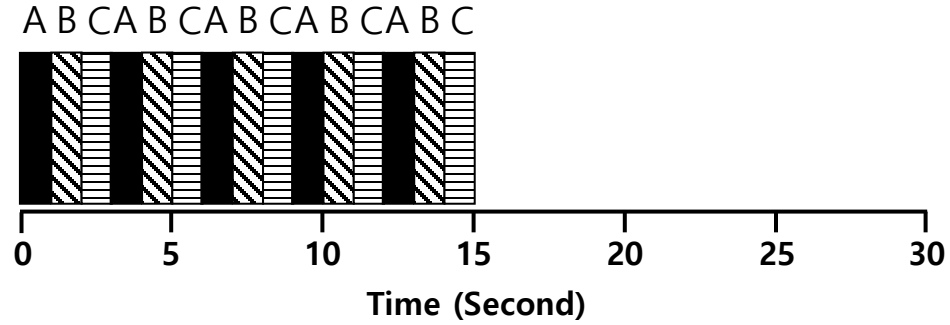
RR Zamanlama Örneği

- A, B and C aynı anda gelir.
- Her birisi 5 saniye çalışmak isterler.



SJF (Bad for Response Time)

$$T_{average\ response} = \frac{0 + 5 + 10}{3} = 5sec$$



RR with a time-slice of 1sec (Good for Response Time)

$$T_{average\ response} = \frac{0 + 1 + 2}{3} = 1sec$$

Zaman diliminin süresi kritik önemdedir

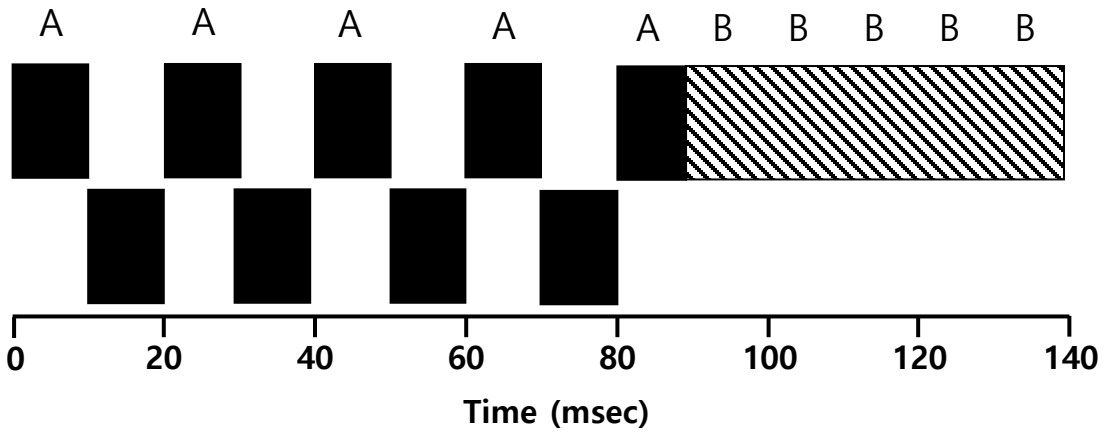
- Zaman dilimi kısa ise
  - Tepki süresi daha iyi olur
  - Context switch maliyeti artar
- Zaman dilimi daha uzun ise
  - Context switch maliyeti düşer (amortize olur)
  - Tepki süresi kötüleşir

Zaman diliminin uzunluğuna karar vermek sistem tasarımcıları için bir ödünleşme (trade-off) sorununu beraberinde getirir

I/O'yu da dikkate alırsak?

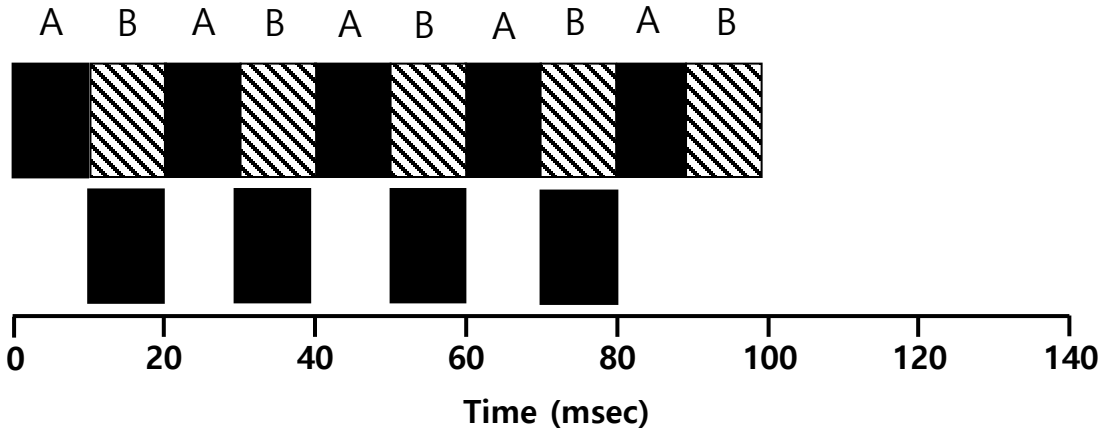
- Varsayım-3'ü de gevşetelim: İşlemler I/O da yapabiliyor olsun.
- Örnek:
  - A ve B, her ikisi de, 50 ms CPU süresi istiyor
  - A, 10 ms çalışıyor sonra I/O isteğinde bulunuyor
    - I/O işlemleri 10 ms sürüyor
  - B ise 50 ms süresi boyunca işlemciyi kullansın, I/O yapmasın
  - Zamanlayıcı önce A'yı, sonra B'yi çalıştırsın

I/O'yu dikkate almak (Devam)



Kaynakların Verimsiz Kullanılması

İşlemci Kullanımını Maksimize Et



Örtüşme sağlayarak kaynakların Verimli Kullanılması

I/O’yu dikkate almak (Devam)

- Bir iş I/O isteğinde bulunduğu zaman
  - O iş, I/O bitene kadar bloklanır
  - Zamanlayıcı başka bir işi planlar
- I/O bittiğinde ise
  - Bir kesme yapılır
  - İşletim sistemi bloklanan işlemi hazır durumuna geçirir

8. Zamanlama:
Çok Seviyeli Geribeslemeli Kuyruk (Multi-Level Feedback Queue)

İşletim Sistemleri: Üç Basit Parça

Çok Seviyeli Geribeslemeli Kuyruk (MLFQ)

- Bir işin çalışma süresi bilinmiyor.
- Geçmişe bakarak geleceği tahmin eden bir zamanlayıcı mümkün mü?
- Amaç:
  - Geri dönüş zamanını optimize edelim → Önce daha kısa işleri çalıştıralım.
  - Tepki zamanını optimize edelim → Bunu iş çalışma süresini bilmeden yapalım

MLFQ: Temel Kurallar

- MLFQ birden fazla farklı kuyruğa sahiptir.
  - Her kuyruğa farklı bir öncelik seviyesi atanır.

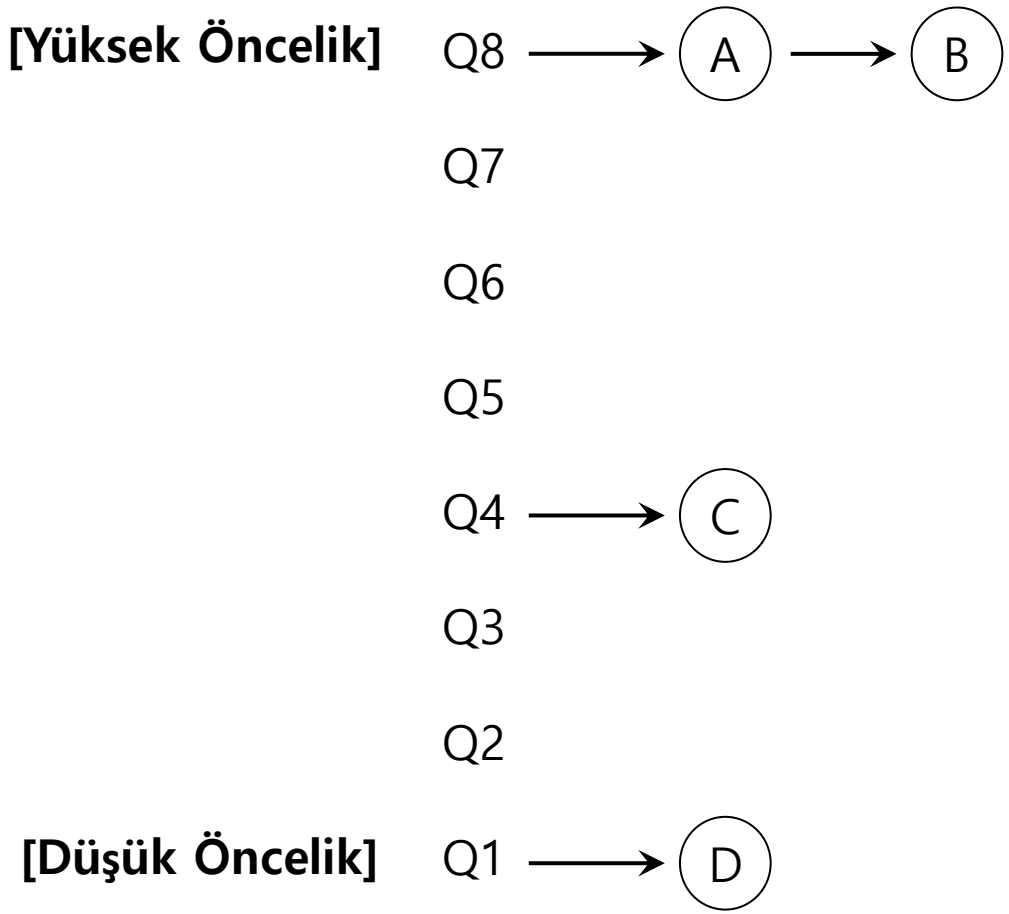
- İşler herhangi bir kuyrukta çalışmaya hazır ise:
  - Daha yüksek kuyruktaki işi seçeriz.
  - Aynı kuyrukta olan işler arasında RR (Round Robin) yöntemini uygularız.

Kural 1: If Öncelik(A) > Öncelik(B), A çalışır (B çalışmaz).
Kural 2: If Öncelik(A) = Öncelik(B), A & B RR ile çalışır.

MLFQ: Temel Kurallar (Devam)

- MLFQ bir işin önceliğini gözlemlenen davranışına bakarak değiştirebilir.
- Örnek:
  - Bir iş I/O için beklediği için CPU'yu bırakıyor ise → Önceliğini yüksek olarak koru
  - Bir iş CPU'yu uzun süreler zarfınca kullanıyor ise → Önceliğini düşür

MLFQ Örnek



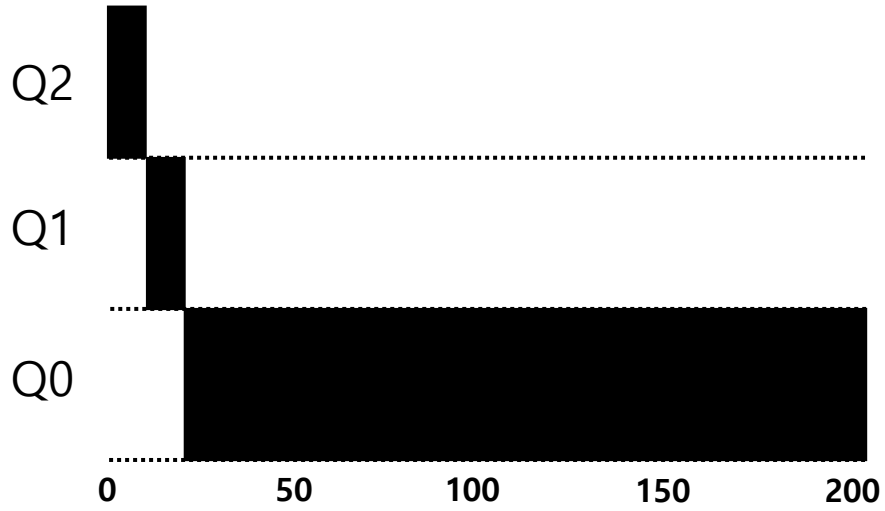
MLFQ: Öncelik Nasıl Değiştirilir?

- MLFQ öncelik belirleme algoritması:
  - **Kural 3**: Bir iş sisteme girdiği zaman en yüksek önceliğe koy
  - **Kural 4a**: Eğer bir iş çalışırken tüm zaman dilimini kullanmış ise, önceliğini düşür (bir sonraki kuyruğa taşı).
  - **Kural 4b**: Eğer bir iş çalışırken zaman dilimini tamamen harcamadan CPU'yu boşaltmış ise, önceliğini koru.

Bu sayede, MLFQ, Önce En Kısa iş yöntemi gibi davranır.

Örnek 1: Tek ve Uzun Bir İş

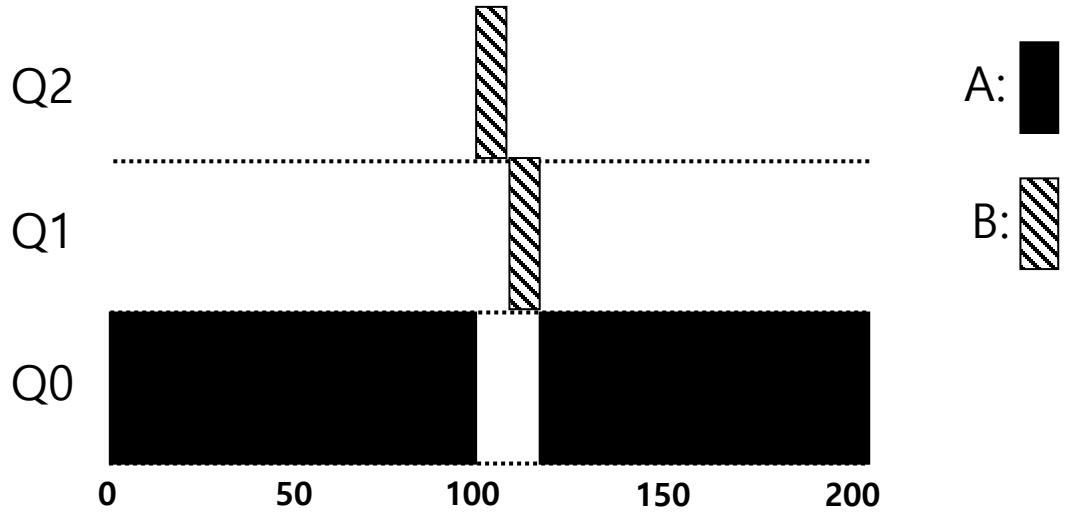
- Üç kuyruklu ve 10 ms zaman dilimine sahip bir zamanlayıcı



Uzun süre çalışan tek bir işin zamanlaması (msec)

Örnek 2: Kısa bir İş de Gelse?

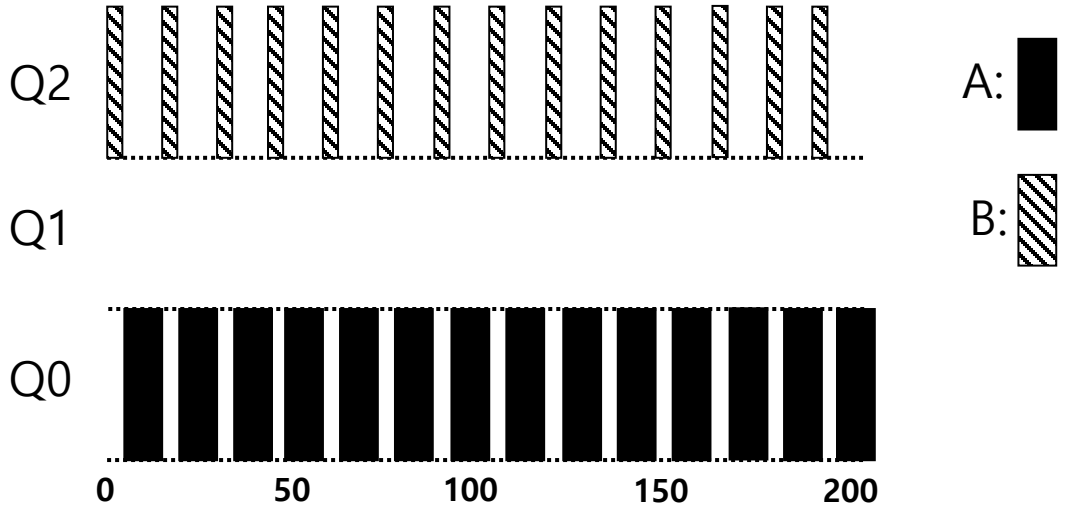
- Varsayımlar:
  - **İş A**: Uzun ve CPU'yu yoğun kullanan bir iş
  - **İş B**: İnteraktif ve kısa süreli bir iş (çalışma süresi: 20ms)
  - A belli bir süredir çalışmaktadır, B, T=100 msn zamanında gelir.



İnteraktif bir iş gelmesi durumu (msec)

Örnek 3: I/O da var ise?

- Varsayımlar:
  - **İş A**: Uzun ve CPU'yu yoğun kullanan bir iş
  - **İş B**: İnteraktif ve I/O yapmadan önce CPU'yu sadece 1 msn. kullanan bir iş



I/O yoğun ve CPU-yoğun iki iş olduğunda zamanlama (msn)

MLFQ yaklaşımı interaktif işi en yüksek öncelikte tutmaya devam eder

Temel MLFQ’nün problemleri

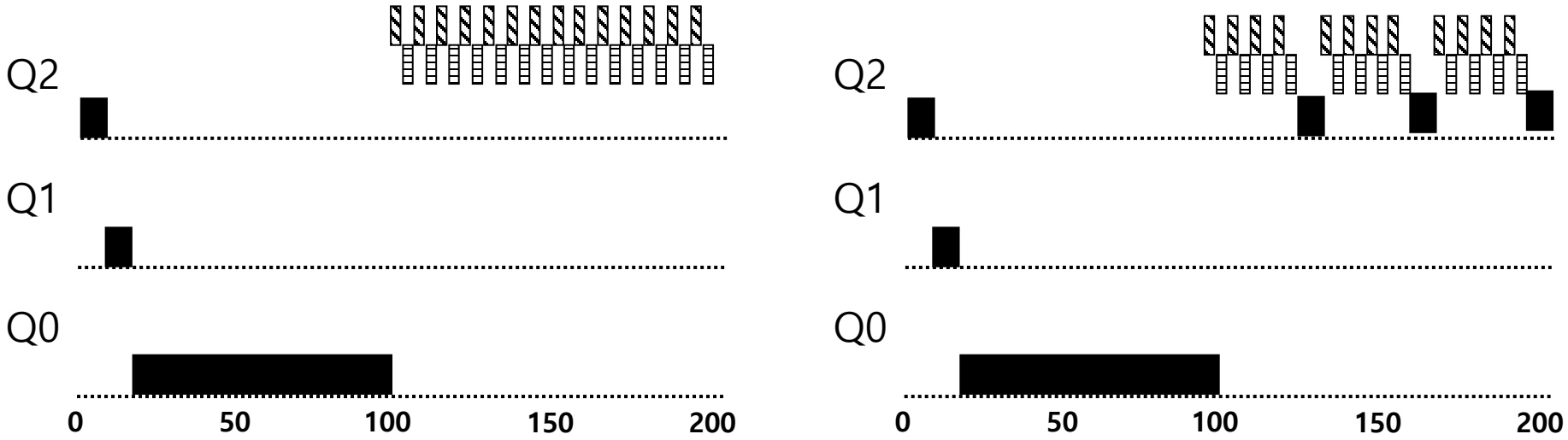
- Açlık
  - Eğer sistemde çok fazla sayıda interaktif iş var ise.
  - Uzun süreli çalışan işlemler CPU’yu kullanamazlar.

- Zamanlayıcıyı Aldatma
  - Zaman diliminin %99’unu kullandıktan sonra I/O yap.
  - Böylece CPU’yu haksız olarak daha fazla kullanmaya başla.

- Bir işin davranışı zaman içerisinde değişebilir.
  - CPU yoğun işlem → I/O yoğun işlem

Öncelik Artırma

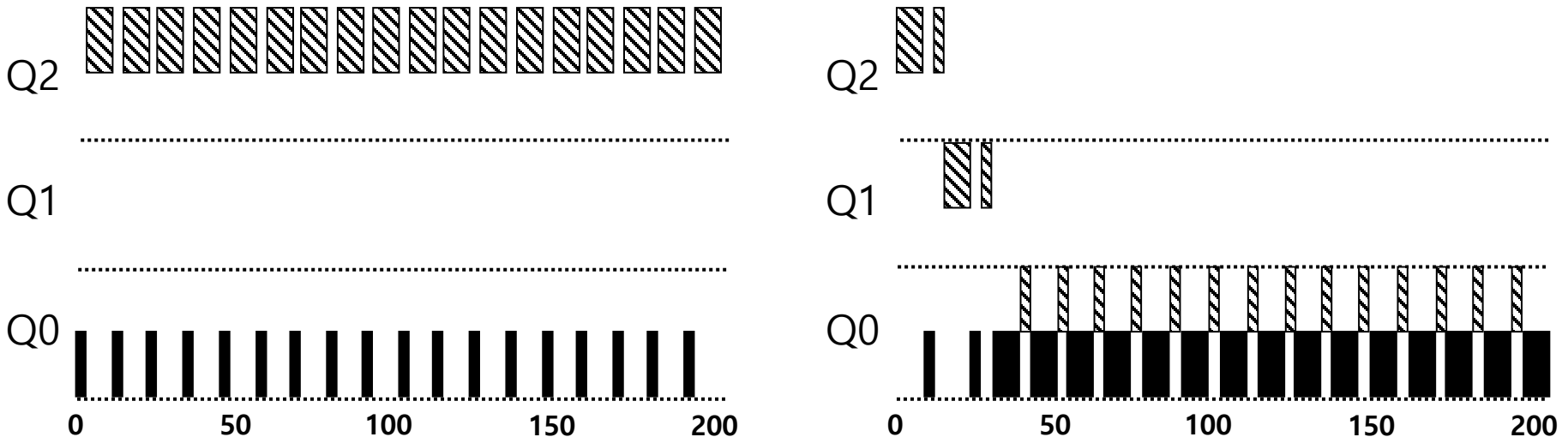
- **Kural 5:** Belli bir S süresi sonunda, sistemdeki tüm işleri en üst kuyruğa taşı.
  - Örnek: Uzun süreli (A) ve iki tane kısa süreli interaktif (B, C) işleri



(Sol) Öncelik Artırma Yok ve (Sağ) Öncelik Artırma Var

Daha İyi Hesap Tutma

- Zamanlayıcının aldatılmasını nasıl engelleriz?
- Çözüm:
  - **Kural 4** (4a ve 4b kurallarını yeniden yazalım): Bir iş herhangi bir seviyede tüm zaman dilimini kullanmış ise (CPU'yu kaç kere bıraktığından bağımsız olarak), önceliğini düşür (aşağı kuyruğa taşı).

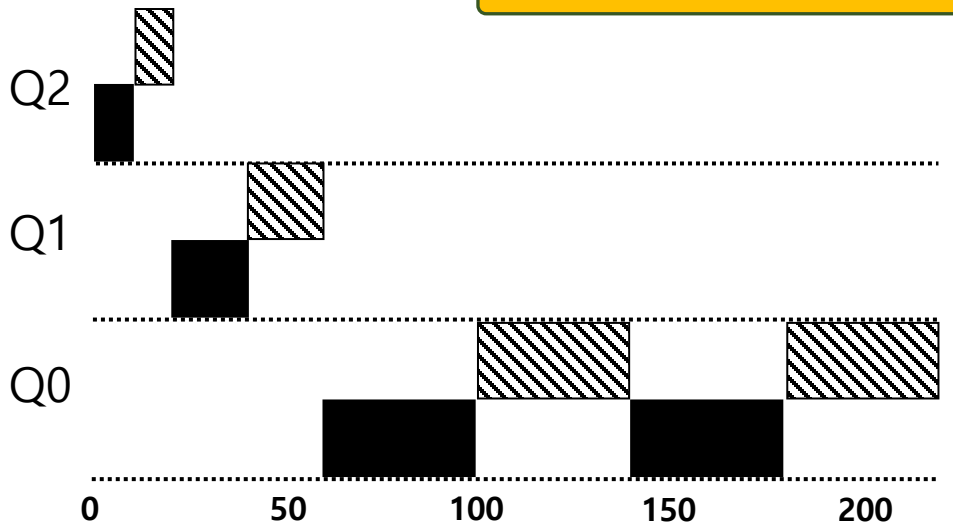


(Sol) Aldatma Var ve (Sağ) Aldatma Yok

MLFQ ile ilgili diğer konular

- Yüksek öncelikli kuyruklar → Kısa zaman dilimleri
  - Örneğin, 10 veya daha az milisaniye
- Düşük öncelikli kuyruklar → Daha uzun zaman dilimleri
  - Örneğin, 100 milisaniye

Düşük Öncelik → Düşük Kuantum değeri



Örnek: en yüksek öncelikli kuyruk için 10ms, ortadaki için 20 ms, en düşük için 40 ms.

Solaris İşletim Sisteminde MLFQ

- Zamanlayıcı:
  - 60 Kuyruk
  - Yavaşça artan zaman dilim süreleri
    - En Yüksek Öncelik: 20ms
    - En düşük öncelik: Birkaç yüz milisaniye
  - Her 1 saniyede bir öncelik artırma.

MLFQ: Özet

- İyileştirilmiş MLFQ kuralları:
  - **Kural 1:** Eğer Öncelik(A) > Öncelik(B), A çalışır (B çalışmaz).
  - **Kural 2:** Eğer Öncelik(A) = Öncelik(B), A & B, RR yöntemi ile beraber çalışır.
  - **Kural 3:** Sisteme yeni bir iş geldiği zaman, en öncelikli kuyruğa eklenir.
  - **Kural 4:** Bir iş herhangi bir seviyede kendisine verilen tüm zaman dilimini kullanırsa (CPU'yu kaç kere bıraktığından bağımsız olarak), önceliği bir düşürülür (bir aşağı kuyruğa kaydırılır).
  - **Kural 5:** Bir S süresi sonunda, sistemdeki tüm işler tekrar en üst kuyruğa taşınır.

9. Zamanlama:
Orantılı Pay (Proportional Share)

İşletim Sistemleri: Üç Basit Parça

Orantılı Pay Zamanlama

- Adil-pay (Fair-share) zamanlama
  - Her işin CPU zamanını belli bir oranda kullanması garanti edilir.
  - Geri dönüş veya tepki süresi açısından en optimum çözüm değildir.

Temel Kavram

- Biletler
  - Bir işlemin bir kaynaktan alacağı payı temsil eder.
  - Sahip olunan biletlerin oranı sistem kaynağından alınan payını gösterir

- Örnek
  - İki işlem olsun, A ve B.
    - İşlem A 75 bilete sahip → %75 oranında CPU'yu kullanır
    - İşlem B 25 bilete sahip → %25 oranında CPU'yu kullanır

Piyango Zamanlama

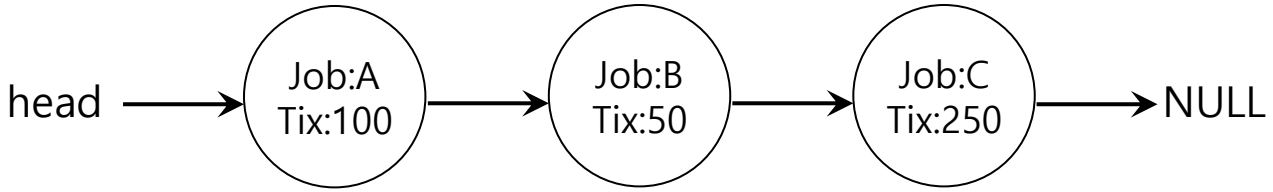
- Zamanlayıcı, piyangoyu kazanan bileti seçer.
- Piyangoyu hangi işlem kazandıysa CPU'yu o kullanır.
- Örnek
  - 100 bilet olsun
    - İşlem A 75 bilete sahip : 0 ~ 74
    - İşlem A 25 bilete sahip : 75 ~ 99

Kazanan bilet:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Zamanlama sonucu:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

**Bu iki iş birbiriyle ne kadar daha uzun süre yarışırsa,
o ölçüde belirlenen oranlara yaklaşılır.**

Implementasyon

- Örnek: Üç işlem olsun: A, B ve C.
- Bu işlemlere bir listeye koyalım:




```
1         // counter: used to track if we've found the winner yet
2         int counter = 0;
3
4         // winner: use some call to a random number generator to
5         // get a value, between 0 and the total # of tickets
6         int winner = getrandom(0, totaltickets);
7
8         // current: use this to walk through the list of jobs
9         node_t *current = head;
10
11        // loop until the sum of ticket values is > the winner
12        while (current) {
13                  counter = counter + current->tickets;
14                  if (counter > winner)
15                            break; // found the winner
16                  current = current->next;
17        }
18        // 'current' is the winner: schedule it...
```

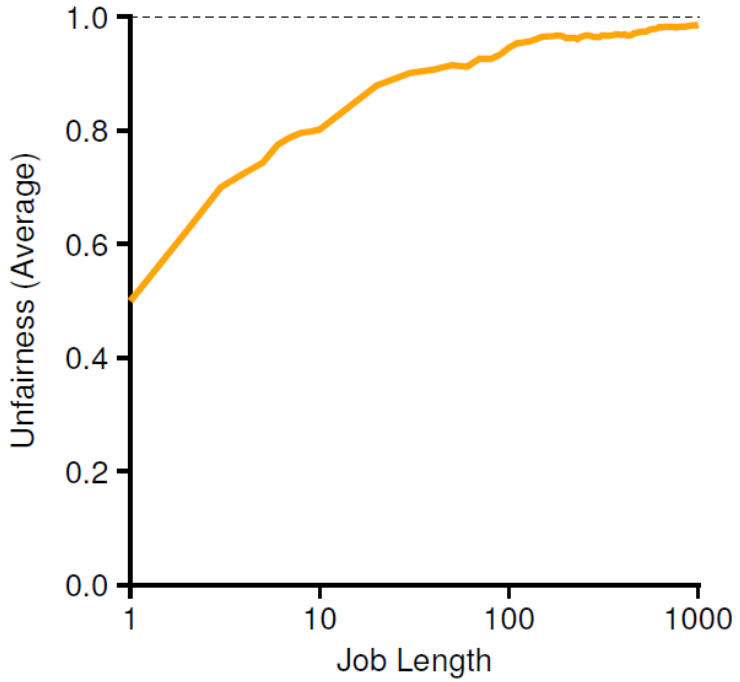
Kodun tamamı için: https://github.com/remzi-arpacidusseau/ostep-code/blob/master/cpu-sched-lottery/lottery.c

Implementasyon (Devam)

- U: adaletsizlik ölçütü
  - İlk işin tamamlandığı zaman / İkinci işin tamamlandığı zaman.
- Örnek:
  - İki iş var ise ve her işi 10 saniye çalışıyor ise
    - İlk iş önce çalışıp 10 saniye sonra biterse
    - Sonra ikinci iş çalışıp 20. saniyede biterse
  - $\mathrm{U} = \frac{10}{20} = 0.5$
  - U her iki iş de yaklaşık aynı anda biterse 1 değerini alır.

Piyango Adillik Analizi

- İki iş var ve her biri 100 bilet sahibi durumda.



Eğer işler yeteri kadar uzun değilse, ortalama adaletsizlik ciddi boyutta olabilir

Adımlı Zamanlama (Stride Scheduling)

- Deterministik adil-pay zamanlama mümkün müdür?
- Her işlemin farklı büyüklükte adımları olsun

  Adım: (Büyük bir sayı) / (her işlemin bilet sayısı)
  - Örnek: Büyük sayı = 10,000
    - İşlem A 100 bilete sahip → A'nın adımı: 100
    - İşlem B 50 bilete sahip → B'nın adımı: 200
- Bir işlem çalışır, o işlemin sayacı (pass) işlemin adımı kadar artırılır.
  - Daha sonra çalıştırmak için o anki en düşük sayaç değerine sahip işlem seçilir

```
current = remove_min(queue);            // pick client with minimum pass
schedule(current);                      // use resource for quantum
current->pass += current->stride;       // compute next pass using stride
insert(queue, current);                 // put back into the queue
```

Adımlı Zamanlama: Örnek

Pass(A) (stride=100)	**Pass(B)** (stride=200)	**Pass(C)** (stride=40)	**Hangisi Çalışır?**
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

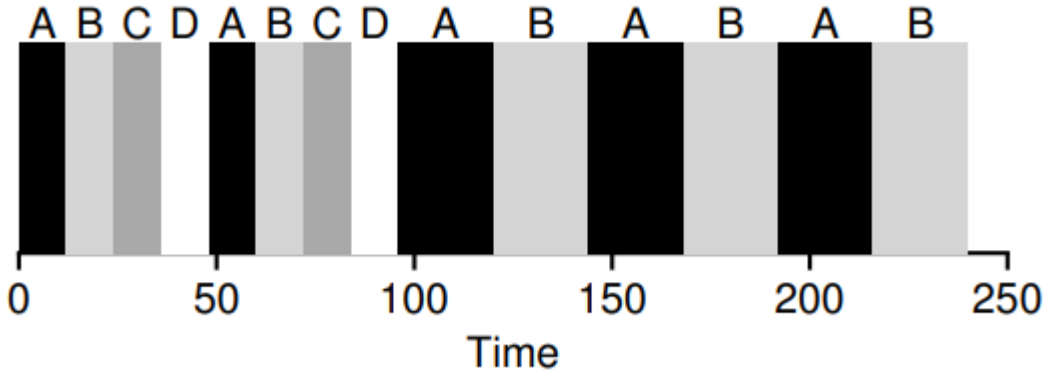
Problem: Yeni bir işlem gelirse ne olacak?

Linux Tamamen Adil Zamanlayıcısı
Linux Completely Fair Scheduler (CFS)

- CFS, farklı bir şekilde adil-pay sağlar.
- CFS, zamanlama kararları için en az zaman harcar.
- CFS, zamanlama kararları vermek için çok az zaman harcamayı amaçlar.
- Son çalışmalar, zamanlayıcı verimliliğinin şaşırtıcı derecede önemli olduğunu göstermiştir. Zamanlayıcı, veri merkezlerinde CPU süresinin yaklaşık %5'ini kullanmaktadır.

Temel Çalışma Mantığı

- Her işlem çalışırken, o işlemin vruntime değeri artar.
- En temel durumda, her işlemin vruntime değeri, fiziksel (gerçek) zamanla orantılı olarak aynı oranda artar.
- Bir zamanlama kararı alınacağı zaman, CFS bir sonraki çalıştırma için en düşük vruntime'a sahip işlemi seçer.
- CFS, işlemler arası bir geçiş öncesinde, bir işlemin ne kadar süre çalışması gerektiğini belirlemek için sched latency değerini kullanır (zaman dilimini dinamik bir şekilde belirler).



Ağırlıklandırma (*nice* mekanizması)

- CFS ayrıca işlem önceliği üzerinde kontrol sağlamak için kullanıcıların veya yöneticilerin bazı işlemlere daha fazla CPU payı vermesini sağlar.
- Bunu biletlerle değil, klasik bir UNIX mekanizması olan işlemin *nice* değerini değiştirerek yapar.
- *nice* parametresi, bir işlem için -20 ile +19 arasında herhangi bir değere ayarlanabilir (varsayılan değeri 0'dır).
- CFS, her işlemin *nice* değerini bir ağırlık (weight) ile eşler. Böylelikle CFS, vruntime'ı ağırlıklandırılmış bir şekilde hesaplar:

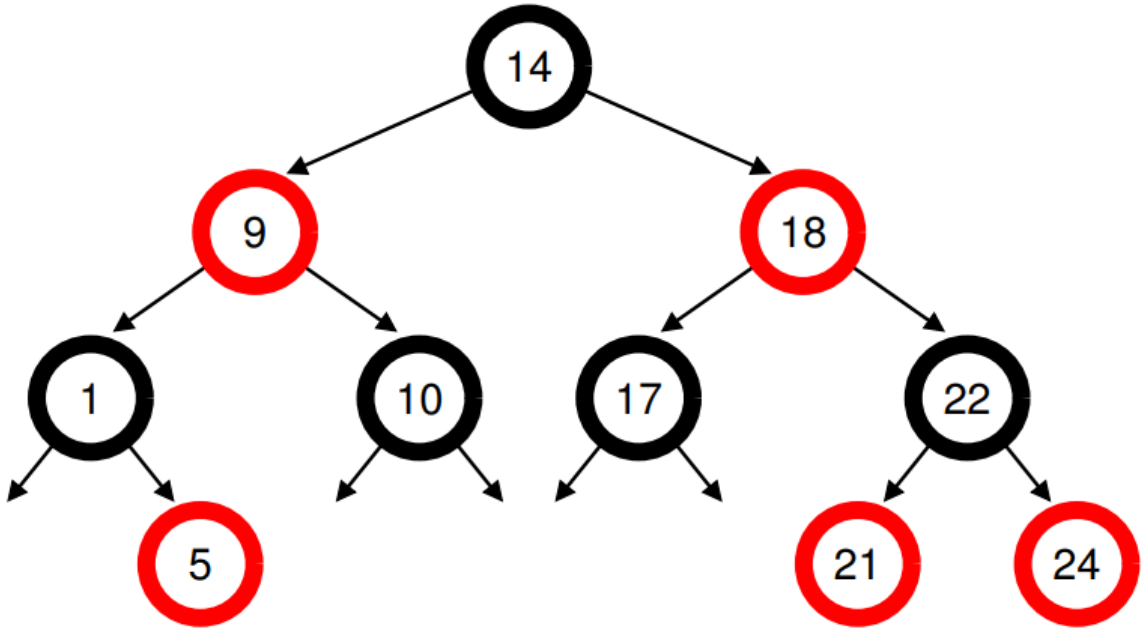
$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i$$

Kırmızı-Siyah Ağaçlar (Red-Black Trees)

- Hazır (çalıştırılabilir) işleri vruntime değerine bağlı olarak sıralı bir listede tutarsak, çalıştırılacak bir sonraki işi bulmak basit olur: listedeki ilk öğeyi al.
- Ancak, bu işi sıralı listeye geri yerleştirirken, ekleme için doğru yeri arayarak listeyi taramamız gerekir: *O(n) (doğrusal) karmaşıklık*.
- Daha iyisi yapılabilir mi?

Kırmızı-Siyah Ağaçlar (Devam)

- İşlemler kırmızı-siyah ağaçta vruntime'a göre sıralanırsa, çoğu opreasyon (ekleme ve silme gibi) zaman açısından logaritmiktir: *O(log n) karmaşıklık*
- *n* sayısı binler ile ifade edildiğinde, logaritmik, doğrusaldan belirgin şekilde daha verimlidir.



CFS Red-Black Tree

I/O ve Uyuyan İşlemlerle ile Başa Çıkma

- En düşük vruntime değerine sahip olan işlemi daha sonra çalıştırılacak işlem olarak seçmeyle ilgili bir sorun, uzun süre uyku moduna geçen işlerde ortaya çıkar.
- İşlem A sürekli çalışan ve İşlem B uzun bir süre (örneğin 10 saniye) uyku moduna geçen iki işlem olsun.
- B uyandığında, bu işlemin vruntime değeri A'nın 10 saniye gerisinde olacak ve bu nedenle (eğer dikkatli olmazsak), B sonraki 10 saniye boyunca CPU'yu tekeline alacak ve A'yı fiilen aç bırakacaktır.
- CFS, bu sorunu bir iş uyandığında onun çalışma zamanını değiştirerek ele alır. CFS, o işin vruntime değerini ağaçta bulunan minimum değere eşitler (Önemli Not: ağaçta yalnızca çalışan işler vardır)