9

Zamanlama: Orantılı Paylaşım

Bu bölümde, **orantılı paylaşım zamanlayıcısı olarak(proportional-share)** bilinen ve bazen **adil paylaşım(fair-share)** zamanlayıcısı olarak da adlandırılan farklı bir zamanlayıcı türünü inceleyeceğiz. Orantılı paylaşım basit bir kavrama dayanır: Bir zamanlayıcı, geri dönüş veya yanıt süresi için optimizasyon yapmak yerine, her işin CPU zamanının belirli bir yüzdesini almasını garanti etmeye çalışır.

Orantılı pay zamanlamsının mükemmel bir erken örneği Waldspurger ve Weihl [WW94] tarafından yapılan araştırmada bulunmuştur ve **piyango zamanlama(lottery scheduling)** olarak bilinir; ancak fikir kesinlikle daha eskidir [KL88]. Temel fikir oldukça basittir: hangi sürecin daha sonra çalışacağını belirlemek için sık sık bir piyango düzenleyin; daha sık çalışması gereken süreçlere piyangoyu kazanması için daha fazla şans verilmelidir. Kolay, değil mi? Şimdi, ayrıntılara gelelim! Ama bizim en önemli noktamızdan önce değil:

Dönüm Noktası: CPU Orantılı Olarak Nasıl Paylaşılır?

CPU'yu orantılı bir şekilde paylaşmak için nasıl bir zamanlayıcı tasarlayabiliriz? Bunu yapmak için temel mekanizmalar nelerdir? Bu mekanizmalar ne kadar etkililer ?

9.1 Temel Kavram: Biletler Payınızı Temsil Eder

Piyango Zamanlamanın altında yatan çok temel bir kavram vardır:, bir sürecin (ya da kullanıcının ya da her neyse) alması gereken kaynak payını temsil etmek için kullanılan biletler**(tickets)**. Bir sürecin sahip olduğu bilet yüzdesi, söz konusu sistem kaynağındaki payını temsil eder

Bunu bir örnekte inceleyelim. A ve B isminde iki süreç olduğunu düşünün ve ayrıca A'nın 75 bileti varken B'nin sadece 25 bileti düşünün.Dolayısıyla A'nın CPU'nun %75'ini, B'nin ise kalan %25'ini almasını istiyoruz.

Piyango zamanlama bunu olasılıksal olarak (ancak kesin olarak değil)sık sık piyango düzenleyerek (örneğin, her zaman dilimi) bir piyango düzenleyerek başarır. Bir piyango düzenlemek basittir: zamanlayıcı toplam kaç bilet olduğunu bilmelidir (örneğimizde 100 bilet var). Zamanlayıcı daha sonra şunları seçer:

İpucu: Rastgeleliği Kulanın

Piyango zamanlamasının en güzel yönlerinden biri, **rastgelelik(randomness)** kullanımıdır. Bir karar vermeniz gerektiğinde, böyle rastgele bir yaklaşım kullanmak genellikle bunu yapmanın sağlam ve basit bir yoludur.

Rastgele yaklaşımların daha geleneksel kararlara göre en az üç avantajı vardır. Birincisi, rastgele genellikle daha geleneksel bir algoritmanın başa çıkmakta zorlanabileceği garip köşe durum davranışlarından kaçınır. Örneğin, LRU değiştirme politikasını düşünün (sanal bellekle ilgili gelecek bir bölümde daha ayrıntılı olarak incelenecektir); genellikle iyi bir değiştirme algoritması olsa da, LRU bazı döngüsel-sıralı iş yükleri için en kötü durum performansına ulaşır. Öte yandan rastgele, böyle bir en kötü duruma sahip değildir.

İkinci olarak, ayrıca rastgele hafiftir, değişiklikleri izlemek için çok az durum gerektirir. Geleneksel bir adil paylaşım zamanlama algoritmasında, her bir sürecin ne kadar CPU aldığını takip etmek, her bir süreci çalıştırdıktan sonra güncellenmesi gereken süreç başına muhasebe gerektirir. Bunu rastgele yapmak, yalnızca en az işlem başına durum gerektirir (örneğin, her birinin sahip olduğu bilet sayısı).

Son olarak, rastgele oldukça hızlı olabilir. Rastgele bir sayı üretmek hızlı olduğu sürece, karar vermek de hızlıdır ve bu nedenle rastgele, hızın gerekli olduğu birçok yerde kullanılabilir. Elbette, ihtiyaç ne kadar hızlı olursa, rastgele sözde rastgeleye doğru o kadar fazla eğilim gösterir.

.

0'dan 99'a kadar bir sayı olan kazanan bilet[1] . A'nın 0'dan 74'e ve B'nin 75'ten 99'a kadar olan biletleri tuttuğunu varsayarsak, kazanan bilet basitçe A'nın mı yoksa B'nin mi çalışacağını belirler. Zamanlayıcı daha sonra bu kazanan sürecin durumunu yükler ve onu çalıştırır.

İşte bir piyango zamanlayıcısının kazanan biletlerinin örnek bir çıktısı:

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49 12
```

İşte ortaya çıkan program:

```
A     A  A     A  A  A  A  A  A     A     A  A  A  A  A  A
   B        B                    B     B
```

Örnekten de görebileceğiniz gibi, piyango zamanlamasında rastgeleliğin kullanılması, istenen payın karşılanmasında olasılıksal bir doğruluğa yol açar, ancak garanti vermez. Yukarıdaki örneğimizde B, istenen %25'lik tahsisat yerine 20 zaman diliminden yalnızca 4'ünü (%20) çalıştırabiliyor. Bununla birlikte, bu iki iş ne kadar uzun süre rekabet ederse, istenen yüzdelere ulaşma olasılıkları o kadar artar.

[1]Computer Scientists always start counting at 0. It is so odd to non-computer-types that famous people have felt obliged to write about why we do it this way [D82].

İPUCU: PAYLAŞIMLARI TEMSIL ETMEK IÇIN BILETLERI KULLANMA

Piyango (ve adım) zamanlama tasarımındaki en güçlü (ve temel) mekanizmalardan biri **bilettir(ticket)**. Bilet, bu örneklerde bir sürecin CPU'daki payını temsil etmek için kullanılmaktadır, ancak çok daha geniş bir şekilde uygulanabilir. Örneğin, hipervizörler için sanal bellek yönetimi üzerine daha yeni bir çalışmada Waldspurger, biletlerin bir konuk işletim sisteminin bellek payını temsil etmek için nasıl kullanılabileceğini göstermektedir [W02]. Dolayısıyla, eğer sahiplik oranını temsil edecek bir mekanizmaya ihtiyaç duyarsanız, bu kavram ... (bekleyin) ... bilet olabilir

9.2 Bilet Mekanizmaları

10 Piyango zamanlaması ayrıca biletleri farklı ve bazen faydalı şekillerde manipüle etmek için bir dizi mekanizma sağlar. Bu yollardan biri **bilet para birimi(ticket currency)** kavramıdır. Para birimi, bir dizi bilete sahip bir kullanıcının biletleri kendi işleri arasında istediği para biriminde tahsis etmesine olanak tanır; sistem daha sonra söz konusu para birimini otomatik olarak doğru global değere dönüştürür.

11 Örneğin, A ve B kullanıcılarının her birine 100 bilet verildiğini varsayalım. Kullanıcı A, A1 ve A2 olmak üzere iki iş yürütüyor ve her birine A'nın para birimi cinsinden 500 bilet (toplam 1000 biletten) veriyor. B kullanıcısı sadece 1 iş yürütüyor ve ona 10 bilet veriyor (toplam 10 biletten). Sistem A1 ve A2'nin A'nın para birimindeki 500'er biletini küresel para biriminde 50'şer bilete dönüştürür; benzer şekilde B1'in 10 bileti de 100 bilete dönüştürülür. Daha sonra hangi işin çalışacağını belirlemek için küresel bilet para birimi (toplam 200) üzerinden çekiliş yapılır.

```
User A -> 500 (A's currency) to A1 ->  50 (global currency)
       -> 500 (A's currency) to A2 ->  50 (global currency)
User B ->  10 (B's currency) to B1 -> 100 (global currency)
```

Bir başka kullanışlı mekanizma da **bilet transferidir**.(**ticket transfer)** Transferler sayesinde, bir süreç biletlerini geçici olarak başka bir sürece devredebilir. Bu özellik özellikle bir istemci sürecinin sunucuya mesaj göndererek istemci adına bazı işler yapmasını istediği istemci/sunucu ortamlarında kullanışlıdır. İşi hızlandırmak için, istemci biletleri sunucuya aktarabilir ve böylece sunucu istemcinin isteğini yerine getirirken sunucunun performansını en üst düzeye çıkarmaya çalışabilir. İş bittiğinde, sunucu biletleri istemciye geri aktarır ve her şey eskisi gibi devam eder.

Son olarak, **bilet enflasyonu(ticket inflation)** bazen yararlı bir teknik olabilir. Enflasyon ile bir süreç sahip olduğu bilet sayısını geçici olarak artırabilir veya azaltabilir. Elbette, birbirlerine güvenmeyen süreçlerin bulunduğu rekabetçi bir senaryoda bunun pek bir anlamı yoktur; açgözlübir süreç kendisine çok sayıda bilet verebilir ve makineyi ele geçirebilir. Bunun yerine, enflasyon bir grup sürecin birbirine güvendiği bir ortamda uygulanabilir; böyle bir durumda, herhangi bir süreç daha fazla CPU süresine ihtiyacı olduğunu bilirse, bu ihtiyacı sisteme yansıtmanın bir yolu olarak, diğer süreçlerle iletişim kurmadan bilet değerini artırabilir.

```
1  // counter: used to track if we've found the winner yet
2  // sayaç: kazananı henüz bulup bulmadığımızı takip etmek
   için kullanılır
3
2  int counter = 0;
3
4  // winner: use some call to a random number generator to
5  // kazanan: rastgele sayı üreteci için bir çağrı kullanın
5  //         get a value, between 0 and the total # of tickets
   //0 ile toplam bilet sayısı arasında bir değer al
6  int winner = getrandom(0, totaltickets);
7
8  // current: use this to walk through the list of jobs
   // geçerli(şimdiki): iş listesinde gezinmek için bunu kullanın
9  node_t *current = head;
10 while (current) {
11     counter = counter + current->tickets;
12     if (counter > winner)
13         break; // found the winner
14     current = current->next;
15 }
16 // 'current' is the winner: schedule it...
   // 'geçerli(şimdiki)' kazanır: zamanlayın...
```

Figure 9.1: **Lottery Scheduling Decision Code**

11.1 Uygulama

Muhtemelen piyango zamanlamasıyla ilgili en şaşırtıcı şey, uygulamasının basitliğidir. İhtiyacınız olan tek şey kazanan bileti seçmek için iyi bir rastgele sayı üreteci, sistemin işlemlerini takip etmekiçin bir veri yapısı (örneğin bir liste) ve toplam bilet sayısıdır.

Süreçleri bir liste halinde tuttuğumuzu varsayalım. Aşağıda her biri belli sayıda bilete sahip A, B ve C olmak üzere üç süreçten oluşan bir örnek verilmiştir.

Bir zamanlama kararı vermek için öncelikle toplam bilet sayısından (400) rastgele bir sayı (kazanan) seçmemiz gerekir[2] Diyelim ki 300 sayısını seçtik. Ardından, kazananı bulmamıza yardımcı olması için basit bir sayaç kullanarak listeyi basitçe dolaşırız (Şekil 9.1).

Kod, değer `kazananı` aşana kadar her bir bilet değerini `sayaca`ekleyerek işlem listesinde dolaşır. Bu durumda, mevcut liste elemanı kazanan olur. Kazanan biletin 300 olduğu örneğimizde aşağıdakiler gerçekleşir. İlk olarak, A'nın biletlerini saymak için sayaç 100'e yükseltilir;100, 300'den küçük olduğu için döngü devam eder. Daha sonra `sayaç` 150'ye (B'nin biletleri) güncellenir, hala 300'den azdır ve böylece yine devam ederiz. Son olarak, `sayaç` 400'e güncellenir (açıkça 300'den büyüktür) ve böylece C'yi (kazanan) işaret eden `akımla` döngüden çıkarız.

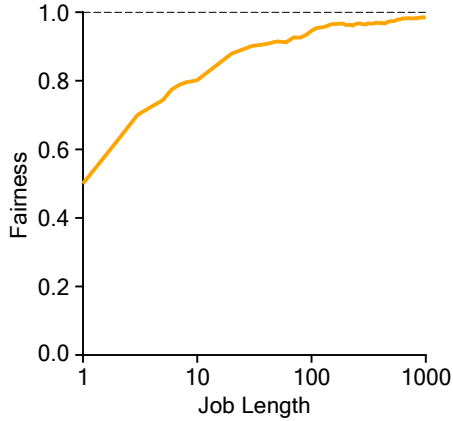


Figure 9.2: **Lottery Fairness Study**

Bu süreci en verimli hale getirmek için, listeyi en yüksek bilet sayısından en düşük bilet sayısına doğru sıralamak genellikle en iyisi olabilir. Sıralama algoritmanın doğruluğunu etkilemez; ancak, özellikle biletlerin çoğuna sahip olan birkaç süreç varsa, genel olarak en azsayıda liste yinelemesinin yapılmasını sağlar.

11.2 Bir Örnek

Piyango zamanlamasının dinamiklerini daha anlaşılır kılmak için, şimdi her biri aynı sayıda bilete (100) ve aynı çalışma süresine (R, ki bunu değiştireceğiz) sahip, birbiriyle rekabet eden iki işin tamamlanma süresine ilişkin kısa bir çalışma gerçekleştireceğiz.

Bu senaryoda, her işin aşağı yukarı aynı zamanda bitmesini isteriz, ancak piyango zamanlamasının rastlantısallığı nedeniyle bazen bir iş diğerinden önce biter. Bu farkı ölçmek için basit bir **adalet ölçütü(fairness metric)** tanımlarız: F, basitçe ilk işin tamamlanma süresinin ikinci işin tamamlanma süresine bölünmesiyle elde edilir. Örneğin, R = 10 ise ve ilk iş 10. zamanda (ve ikinci iş 20. zamanda) bitiyorsa, F =10 = 0,5 olur. Her iki iş de neredeyse aynı zamanda bittiğinde, F 1'e oldukça yakın olacaktır. Bu senaryoda hedefimiz budur: mükemmel adil bir zamanlayıcı F = 1'e ulaşacaktır F = 1.

Şekil 9.2, iki işin uzunluğu (R) otuz deneme boyunca 1 ila 1000 arasında değiştikçe ortalama adaleti göstermektedir (sonuçlar bölümün sonunda verilen simülatör aracılığıyla oluşturulmuştur). Grafikten de görebileceğiniz gibi, iş uzunluğu çok uzun olmadığında, ortalama adalet oldukça düşük olabilir. Ancak işler önemli sayıda zaman dilimi boyunca çalıştığında piyango zamanlayıcısı istenen adil sonuca yaklaşır.

11.3 Biletler Nasıl Atanır ?

12 Piyango çizelgelemesinde ele almadığımız bir sorun şudur: Biletler işlere nasıl atanır? Bu sorun zor bir sorundur, çünkü elbette sistemin nasıl davranacağı biletlerin nasıl tahsis edildiğine büyük ölçüde bağlıdır.Yaklaşımlardan biri, kullanıcıların en iyisini bildiğini varsaymaktır; böyle bir durumda, her kullanıcıya belirli sayıda bilet verilir ve bir kullanıcı çalıştırdığı herhangi bir işe biletleri istediği gibi tahsis edebilir. Ancak, bu çözüm bir çözüm değildir: size gerçekten ne yapmanız gerektiğini söylemez. Dolayısıyla, bir dizi iş verildiğinde, "bilet atama sorunu" açık kalır.

12.1 Adım Zamanlama

Şunu da merak ediyor olabilirsiniz: Neden rastgelelik kullanılsın ki? Yukarıda gördüğümüz gibi, rastgelelik bize basit (ve yaklaşık olarak doğru) bir zamanlayıcı sağlarken, özellikle kısa zaman ölçeklerinde bazen tam olarak doğru oranları vermeyecektir. Bu nedenle Waldspurger, deterministik bir adil paylaşım zamanlayıcısı olan **adım zamanlamayı(stride scheduling)** icat etmiştir [W95].

Adım zamanlama da anlaşılırdır. Sistemdeki her işin, sahip olduğu bilet sayısıyla ters orantılı bir adım aralığı vardır. Yukarıdaki örneğimizde, sırasıyla 100, 50 ve 250 bilete sahip A, B ve C işleriyle,her bir işlemin atandığı bilet sayısına büyük bir sayı bölerek her birinin adımını hesaplayabiliriz. Örneğin, 10.000'i bu bilet değerlerinin her birine bölersek, A, B ve C için şu adım değerlerini elde ederiz: 100, 200 ve 40. Bu değere her bir sürecin **adım(stride)** sayısı diyoruz; bir süreç her çalıştığında, küresel ilerlemesini izlemek için onun için bir sayacı (**geçiş(pass)** değeri olarak adlandırılır) adım sayısı kadar artıracağız

Zamanlayıcı daha sonra hangi işlemin daha sonra çalışması gerektiğini belirlemek için adım ve geçişi kullanır. Temel fikir basittir:herhangi bir zamanda, o ana kadar en düşük geçiş değerine sahip olan süreci çalıştırmak için seçin; süreci çalıştırdığınızda, geçiş sayacını adımıyla artırın. Waldspurger [W95] tarafından bir sözde kod uygulaması sağlanmıştır:

```
curr = remove_min(queue);    // pick client with min pass
schedule(curr);              // run for quantum
curr->pass += curr->stride; // update pass using stride
insert(queue, curr);         // return curr to queue
```

Örneğimizde, adım değerleri 100, 200 ve 40 olan ve geçiş değerleri başlangıçta 0 olan üç süreçle (A, B ve C) başlıyoruz. Dolayısıyla, geçiş değerleri eşit derecede düşük olduğu için ilk başta süreçlerden herhangi biri çalışabilir. A'yı seçtiğimizi varsayalım (keyfi olarak; eşit düşük geçiş değerlerine sahip süreçlerden herhangi biri seçilebilir). A çalışır; zaman dilimiyle işi bittiğinde geçiş değerini 100 olarak güncelleriz. Daha sonra B'yi çalıştırırız ve onun geçiş değeri 200 olarak ayarlanır. Son olarak, geçiş değeri 40'a yükseltilen C'yi çalıştırırız. Bu noktada, algoritma en düşük geçiş değerini, yani C'nin geçiş değerini seçecek ve

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
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	...

Figure 9.3: **Stride Scheduling: A Trace**

çalıştırarak geçiş değerini 80'e yükseltir (hatırladığınız gibi C'nin adım sayısı 40'tır). Sonra C tekrar çalışacak (hala en düşük geçiş değeri) ve geçişini 120'ye yükseltecek. A şimdi çalışacak ve geçişini 200'e güncelleyecek (şimdi B'ninkine eşit). Ardından C iki kez daha çalışacakve geçiş değerini önce 160'a sonra 200'e yükseltecektir. Bu noktada, tüm geçiş değerleri tekrar eşit olur ve süreç sonsuza kadar tekrar eder. Şekil9.3 zamanlayıcı davranışını zaman içinde izler.

Şekilden de görebileceğimiz gibi, C beş kez, A iki kez ve B sadece birkez, tam olarak 250, 100 ve 50 bilet değerleriyle orantılı olarak çalışmıştır. piyango zamanlaması zaman içinde olasılıksal olarak oranlara ulaşır; adım zamanlaması her zamanlama döngüsünün sonunda bunları tam olarak doğru yapar.

.

13 Merak ediyor olabilirsiniz: adım zamanlama hassasiyeti göz önüne alındığında, neden piyango zamanlama kullanılsın ki? Piyango zamanlamanın adım zamanlamada olmayan güzel bir özelliği var: global durum yok. Yukarıdaki adım zamanlama örneğimizin ortasına yeni bir işin girdiğini düşünün; geçiş değeri ne olmalı? 0'a mı ayarlanmalı? Eğer öyleyse, CPU'yu tekeline alacaktır. Piyango zamanlamasında, her işlem için global bir durum yoktur; sadece sahip olduğu biletlerle yeni bir işlem ekleriz, toplam kaç biletimiz olduğunu izlemek için tek global değişkeni güncelleriz ve oradan devam ederiz. Bu şekilde piyango, yeni süreçleri mantıklı bir şekilde dahil etmeyi çok daha kolay hale getirir.

13.1 Linux Tamamen Adil Zamanlayıcı (CFS)

Adil paylaşım zamanlamasındaki bu önceki çalışmalara rağmen, mevcut Linux yaklaşımı benzer hedeflere alternatif bir şekilde ulaşmaktadır.

Tamamen Adil Zamanlayıcı (Completely Fair Scheduler) (veya **CFS**) [J09] olarak adlandırılan zamanlayıcı, adil paylaşımlı zamanlamayı uygular, ancak bunu oldukça verimli ve ölçeklenebilir bir şekilde yapar

CFS, verimlilik hedeflerine ulaşmak için hem kendi tasarımı hem de göreve uygun veri yapılarını akıllıca kullanarak zamanlama kararları almak için çok az zaman harcamayı amaçlamaktadır. Yakın zamanda yapılan çalışmalar, zamanlayıcı verimliliğinin şaşırtıcı derecede önemli olduğunu göstermiştir; özellikle, Google veri merkezlerinde yapılan bir çalışmada, Kanev ve diğerleri, agresif optimizasyondan sonra bile, zamanlamanın toplam veri merkezi CPU zamanının yaklaşık %5'ini kullandığını göstermiştir. Dolayısıyla bu ek yükü mümkün olduğunca azaltmak, modern zamanlayıcı mimarisinde önemli bir hedeftir.

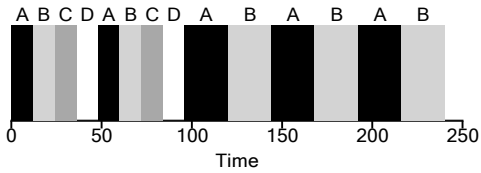


Figure 9.4: **CFS Simple Example**

Basit Operasyonlar

Çoğu zamanlayıcı sabit bir zaman dilimi kavramına dayanırken, CFS biraz farklı çalışır. Amacı basittir: bir CPU'yu tüm rakip süreçler arasında eşit olarak bölmek. Bunu **sanal çalışma zamanı(virtual runtime)**(`vruntime`) olarak bilinen basit bir sayım tabanlı teknikle yapar.

Her süreç çalıştıkça `vruntime` biriktirir. En temel durumda, her sürecin `vruntime`'ı fiziksel (gerçek) zamanla orantılı olarak aynı oranda artar. Bir zamanlama kararı oluştuğunda, CFS bir sonraki çalıştırılacak en düşük çalışma zamanına sahip işlemi seçer.

Bu bir soruyu gündeme getiriyor: Zamanlayıcı şu anda çalışan işlemi ne zaman durduracağını ve bir sonrakini ne zaman çalıştıracağını nasıl biliyor? Buradaki gerilim açıktır: CFS çok sık geçiş yaparsa, CFS her sürecin çok küçük zaman aralıklarında bile CPU'dan pay almasını sağlayacağı için adalet artar, ancak performans azalır(çok fazla bağlam değiştirme yapılır.); CFS daha az sıklıkta geçiş yaparsa, performans artar (daha az bağlam değiştirme yapılır), ancak yakın vadeli adalet azalır.

CFS bu gerilimi çeşitli kontrol parametreleri aracılığıyla yönetir. Bunlardan ilki **zamanlama gecikmesidir(sched latency)**. CFS bu değeri, bir sürecin bir geçişi değerlendirmeden önce ne kadar süre çalışması gerektiğini belirlemek için kullanır (zaman dilimini dinamik ve etkin bir şekilde belirler). Tipik bir zamanlama gecikmesi değeri 48'dir (milisaniye); CFS bu değeri CPU üzerinde çalışan işlem sayısına (n) bölerek bir işlem için zaman dilimini belirler ve böylece bu süre boyunca CFS'nin tamamen adil olmasını sağlar.

Örneğin, çalışan n = 4 işlem varsa, CFS 12 ms'lik işlem başına bir zaman dilimine ulaşmak için zamanlama gecikmesi değerini n'ye böler. CFS daha sonra ilk işi planlar ve 12 ms'lik (sanal) çalışma süresini kullanana kadar çalıştırır ve ardından bunun yerine çalıştırılacak daha düşük `vruntime`'a sahip bir iş olup olmadığını kontrol eder. Bu durumda, CFS diğer üç işten birine geçer ve bu böyle devam eder. Şekil 9.4'te dört işin (A, B, C, D) her birinin bu şekilde iki zaman dilimi boyunca çalıştığı bir örnek gösterilmektedir; daha sonra bunlardan ikisi (C, D) tamamlanarak geriye sadece iki iş kalır ve bu işlerin her biri yuvarlak robin şeklinde 24 ms boyunca çalışır.

Peki ya çalışan "çok fazla" süreç varsa? Bu çok küçük bir zaman dilimine ve dolayısıyla çok fazla bağlam değişimine yol açmaz mı? Güzel soru! Ve cevabı evet.

Bu sorunu çözmek için CFS, genellikle 6 ms gibi bir değere ayarlanan **minimum ayrıntı düzeyi (min granularity)** adında başka bir parametre ekler. CFS zaman dilimini asla ayarlamaz

bu değerden daha düşük bir değere ayarlamaz ve genel giderleri zamanlamak için çok fazla zaman harcanmamasını sağlar.

Örneğin, çalışan on süreç varsa, orijinal hesaplamamız zaman dilimini belirlemek için zamanlama gecikmesini ona bölecektir (sonuç: 4,8 ms). Ancak, minimum ayrıntı düzeyi nedeniyle CFS her bir işlemin zaman dilimini 6 ms olarak belirleyecektir. CFS, 48 ms'lik hedef zamanlama gecikmesi **(sched latency)** üzerinde (tam olarak) mükemmel bir şekilde adil olmayacak olsa da, yüksek CPU verimliliği elde ederken yakın olacaktır.

CFS'nin periyodik bir zamanlayıcı kesmesi kullandığını, yani yalnızca sabit zaman aralıklarında karar verebileceğini unutmayın. Bu kesme sık sık (örneğin her 1 ms'de bir) kesilerek CFS'ye uyanma ve mevcut işin çalışma süresinin sonuna ulaşıp ulaşmadığını belirleme şansı verir. Bir işin zamanlayıcı kesme aralığının tam katı olmayan bir zaman dilimi varsa, bu sorun değildir; CFS vruntime'ı tam olarak izler, bu da uzun vadede CPU'nun ideal paylaşımına yaklaşacağı anlamına gelir.

Ağırlıklandırma (İyilik)

CFS ayrıca işlem önceliği üzerinde kontroller sağlayarak kullanıcıların ya da yöneticilerin bazı işlemlere CPU'dan daha fazla pay vermesini sağlar. Bunu biletlerle değil, bir sürecin **iyi(nice)** seviyesi olarak bilinen klasik bir UNIX mekanizması aracılığıyla yapar.İyilik(nice) parametresi bir süreç için -20 ila +19 arasında herhangi bir yere ayarlanabilir, varsayılan değer 0'dır. Pozitif iyi(nice) değerleri daha *düşük* öncelik anlamına gelir ve negatif değerler daha *yüksek öncelik* anlamına gelir; çok iyi(nice) olduğunuzda, ne yazık ki çok fazla (zamanlama) dikkat çekmiyorsunuz.

CFS, burada gösterildiği gibi her bir sürecin iyi(nice) değerini bir ağırlıkla eşler:

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

Bu ağırlıklar, her bir sürecin etkin zaman dilimini hesaplamamızı sağlar (daha önce yaptığımız gibi), ancak şimdi öncelik farklılıklarını hesaba katar. Bunu yapmak için kullanılan formül, n süreç olduğu varsayılarak aşağıdaki gibidir:

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency} \qquad (9.1)$$

Bunun nasıl çalıştığını görmek için bir örnek yapalım. A ve B olmak üzere iki iş olduğunu varsayalım. A, en değerli işimiz olduğu için.

-5 gibi iyi(nice) bir değer atanarak daha yüksek bir öncelik verilir; B, ondan nefret ettiğimiz için, yalnızca varsayılan önceliğe sahiptir (0'a eşit iyi(nice) değer). Bu, A'nın ağırlığı (tablodan) 3121, B ağırlığı ise 1024 olduğu anlamına gelir Daha sonra her işin zaman dilimini hesaplarsanız, A'nın zaman diliminin zamanlama gecikmesinin yaklaşık 3'ü olduğunu göreceksiniz. (bu nedenle, 36 ms) ve B'nin yaklaşık 1'i (dolayısıyla 12 ms).

Zaman dilimi hesaplamasının genelleştirilmesine ek olarak, CFS'nin `vruntime hesaplama` şekli de uyarlanmalıdır. İşte yeni formül, i işleminin tahakkuk ettiği gerçek çalışma süresini (`runtimei`) alır ve 1024 varsayılan ağırlığını ($\texttt{weight}_0$) kendi ağırlığı olan `weighti`'ye bölerek işlemin ağırlığıyla ters orantılı olarak ölçeklendirir. Çalıştırma örneğimizde, A'nın `vruntime'ı` B'ninkinin üçte biri oranında birikecektir

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i \qquad (9.2)$$

Yukarıdaki ağırlık tablosunun yapısının akıllıca bir yönü, güzel değerlerdeki fark sabit olduğunda tablonun CPU orantılılık oranlarını korumasıdır. Örneğin, eğer A süreci bunun yerine 5 (-5 değil) ve B süreci 10 (0 değil) değerine sahipse, CFS bunları daha önce olduğu gibi tam olarak aynı şekilde programlayacaktır. Nedenini görmek için işlemi kendiniz gözden geçirin.

Kırmızı Siyah Ağaçların Kullanılması

CFS'nin ana odak noktalarından biri yukarıda da belirtildiği gibi verimliliktir. Bir zamanlayıcı için verimliliğin birçok yönü vardır, ancak bunlardan biri şu kadar basittir: zamanlayıcı çalıştırılacak bir sonraki işi bulmak zorunda olduğunda, bunu mümkün olduğunca çabuk yapmalıdır. Listeler gibi basit veri yapıları ölçeklenemez: modern sistemler bazen 1000'lerce süreçten oluşur ve bu nedenle her milisaniyede bir uzun bir listede arama yapmak israftır.

CFS, süreçleri **kırmızı-siyah ağaçta(red-black tree)** tutarak bu sorunu çözer [B72]. Kırmızı-siyah ağaç, birçok dengeli ağaç türünden biridir; basit bir ikili ağacın aksine (en kötü durum ekleme modellerinde liste benzeri performansa dönüşebilir), dengeli ağaçlar düşük derinlikleri korumak için biraz ekstra iş yapar ve böylece işlemlerin zaman içinde logaritmik (doğrusal değil) olmasını sağlar.

CFS *tüm* süreçleri bu yapıda tutmaz; bunun yerine, yalnızca çalışan (veya çalıştırılabilir) süreçler burada tutulur. Bir süreç uykuya geçerse (örneğin, bir G/Ç'nin tamamlanmasını ya da bir ağ paketinin gelmesini beklerse), ağaçtan çıkarılır ve başka bir yerde takip edilir.

Bunu daha açık hale getirmek için bir örneğe bakalım. On iş olduğunu ve bunların şu `vruntime` değerlerine sahip olduğunu varsayalım: 1, 5, 9,10, 14, 18, 17, 21, 22 ve 24. Bu işleri sıralı bir listede tutarsak, çalıştırılacak bir sonraki işi bulmak basit olacaktır: sadece ilk öğeyi kaldırın. Ancak,

[3]Yes, yes, we are using bad grammar here on purpose, please don't send in a bug fix. Why? Well, just a most mild of references to the Lord of the Rings, and our favorite anti-hero Gollum, nothing to get too excited about.

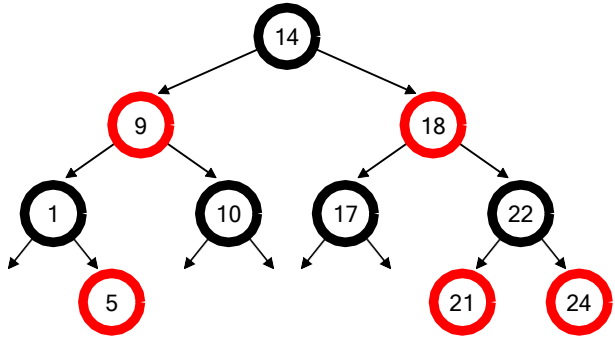


Figure 9.5: **CFS Red-Black Tree**

bu işi listeye geri yerleştirirken (sırayla), listeyi taramamız ve yerleştirmek için doğru noktayı aramamız gerekir, bu da bir $O(n)$ işlemidir. Herhangi bir arama da oldukça verimsizdir ve ortalama olarak doğrusal zaman alır.

Kırmızı-siyah ağaçta aynı değerlerin tutulması, Şekil 9.5'te gösterildiği gibi çoğu işlemi daha verimli hale getirir. İşlemler ağaçta `vruntime'a` göre sıralanır ve çoğu işlem (ekleme ve silme gibi) logaritmik `zamanlıdır`, yani $O(\log n)$. N binlerce olduğunda, logaritmik doğrusaldan belirgin şekilde daha verimlidir..

G/Ç ve Uyku Süreçleri ile Başa Çıkma

Bir sonraki çalıştırılacak en düşük `vruntime'ın` seçilmesiyle ilgili bir sorun, uzun bir süre uykuya geçen işlerde ortaya çıkar. Biri (A) sürekli çalışan, diğeri (B) ise uzun bir süre (diyelim ki 10 saniye) uykuda kalan A ve B olmak üzere iki iş düşünün. B uyandığında, `vruntime'ı` A'nınkinden 10 saniye geride olacaktır ve bu nedenle (eğer dikkatli olmazsak), B, A'yı etkili bir şekilde aç bırakarak yetişirken önümüzdeki 10 saniye boyunca CPU'yu tekeline alacaktır.

CFS bu durumu, bir iş uyandığında `vruntime'ını` değiştirerek ele alır. Özellikle, CFS o işin `vruntime'ını` ağaçta bulunan minimum değere ayarlar (unutmayın, ağaç sadece çalışan işleri içerir) [B+18]. Bu şekilde CFS açlıktan ölmeyi önler, ancak bunun bir bedeli vardır: kısa süreler boyunca uyuyan işler CPU'dan adil bir pay alamazlar [AC97].

Diğer CFS Eğlenceleri

CFS, kitabın bu noktasında tartışılamayacak kadar çok sayıda başka özelliğe sahiptir. Önbellek performansını artırmak için çok sayıda sezgisel yöntem içerir, birden fazla CPU'yu etkili bir şekildeele almak için stratejilere sahiptir (kitabın ilerleyen bölümlerinde tartışıldığı gibi), büyük işlem grupları arasında zamanlama yapabilir (her süreci bağımsız

İPUCU: UYGUN OLDUĞUNDA VERIMLI VERI YAPILARI KULLANIN

Birçok durumda bir liste işinizi görecektir. Çoğu durumda da işe yaramaz. Hangi veri yapısının ne zaman kullanılacağını bilmek iyi bir mühendisliğin ayırt edici özelliğidir. Burada tartışılan durumda, daha önceki zamanlayıcılarda bulunan basit listeler modern sistemlerde, özellikle de veri merkezlerinde bulunan ağır yüklü sunucularda iyi çalışmaz. Bu tür sistemler binlerce aktif işlem içerir; her birkaç milisaniyede bir her çekirdekte çalıştırılacak bir sonraki işi bulmak için uzun bir listede arama yapmak değerli CPU döngülerini boşa harcayacaktır. Daha iyi bir yapıya ihtiyaç vardı ve CFS, kırmızı-siyah ağacın mükemmel bir uygulamasını ekleyerek bir yapı sağladı. Daha genel olarak, kurmakta olduğunuz bir sistem için bir veri yapısı seçerken, erişim yollarını ve kullanım sıklığını dikkatlice düşünün;bunları anlayarak elinizdeki görev için doğu yapıyı uygulayabileceksiniz.

bir varlık olarak ele almak yerine) ve diğer birçok ilginç özellik. Daha fazla bilgi edinmek için Bouron [B+18] ile başlayan son araştırmaları okuyun.

13.2 Özet

Orantılı paylaşım zamanlama kavramını tanıttık ve üç yaklaşımı kısaca tartıştık: piyango zamanlama, adım zamanlama ve Linux'un Tamamen Adil Zamanlayıcısı (CFS). Piyango, orantılı paylaşıma ulaşmak için rastgeleliği akıllıca bir şekilde kullanır; adım(stride) ise bunu caydırıcı bir şekilde yapar. Bu bölümde tartışılan tek "gerçek" zamanlayıcı olan CFS, dinamik zaman dilimlerine sahip ağırlıklı **round- robin'e(Bir zamanlama (scheduling) algoritmasıdır. Özellikle işletim sistemi tasarımında işlemcinin (CPU) zamanlamasında kullanılan meşhur algoritmalardan birisidir)** benzer, ancak yük altında ölçeklendirmek ve iyi performans göstermek için üretilmiştir; bildiğimiz kadarıyla, bugün var olan en yaygın kullanılan adil paylaşım zamanlayıcısıdır.

Hiçbir zamanlayıcı her derde deva değildir ve adil paylaşımlızamanlayıcıların de kendi paylarına düşen sorunları vardır. Sorunlardan biri, bu tür yaklaşımların özellikle I/O [AC97] ile iyi uyum sağlamamasıdır; yukarıda belirtildiği gibi, ara sıra I/O gerçekleştiren işler CPU'dan adil pay alamayabilir. Diğer bir sorun ise bilet ya da öncelik atama gibi zor bir sorunu açıkta bırakmalarıdır, yani tarayıcınıza kaç bilet ayrılması gerektiğini ya da metin editörünüzü hangi iyi(nice) değere ayarlamanız gerektiğini nasıl bileceksiniz? Diğer genel amaçlı zamanlayıcılar (daha önce tartıştığımız MLFQ ve diğer benzer Linux zamanlayıcıları gibi) bu sorunları otomatik olarak ele alır ve bu nedenle daha kolay dağıtılabilirler.

İyi haber şu ki, bu sorunların baskın olmadığı birçok alan var ve orantılı paylaşım zamanlayıcıları büyük etki için kullanılıyor. Örneğin, CPU döngülerinin dörtte birini Windows **sanal** makinesine ve geri kalanını temel Linux kurulumunuza atamak isteyebileceğiniz **sanallaştırılmış(virtualized)** bir veri merkezinde (veya **bulutta(cloud)**), oransal paylaşım basit ve etkili olabilir. Bu fikir diğer kaynaklara da genişletilebilir; VMWare'in ESX Sunucusunda belleğin orantılı olarak nasıl paylaştırılacağı hakkında daha fazla ayrıntı için Waldspurger'e [W02] bakın.

References

[AC97] "Extending Proportional-Share Scheduling to a Network of Workstations" by Andrea C. Arpaci-Dusseau and David E. Culler. PDPTA'97, June 1997. *A paper by one of the authors on how to extend proportional-share scheduling to work better in a clustered environment.*

[B+18] "The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS" by J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, J. Sopena. USENIX ATC '18, July 2018, Boston, Massachusetts. *A recent, detailed work comparing Linux CFS and the FreeBSD schedulers. An excellent overview of each scheduler is also provided. The result of the comparison: inconclusive (in some cases CFS was better, and in others, ULE (the BSD scheduler), was. Sometimes in life there are no easy answers.*

[B72] "Symmetric binary B-Trees: Data Structure And Maintenance Algorithms" by Rudolf Bayer. Acta Informatica, Volume 1, Number 4, December 1972. *A cool balanced tree introduced before you were born (most likely). One of many balanced trees out there; study your algorithms book for more alternatives!*

[D82] "Why Numbering Should Start At Zero" by Edsger Dijkstra, August 1982. Available: `http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF`. *A short note from E. Dijkstra, one of the pioneers of computer science. We'll be hearing much more on this guy in the section on Concurrency. In the meanwhile, enjoy this note, which includes this motivating quote: "One of my colleagues — not a computing scientist — accused a number of younger computing scientists of 'pedantry' because they started numbering at zero." The note explains why doing so is logical.*

[K+15] "Profiling A Warehouse-scale Computer" by S. Kanev, P. Ranganathan, J. P. Darago, K. Hazelwood, T. Moseley, G. Wei, D. Brooks. ISCA '15, June, 2015, Portland, Oregon. *A fascinating study of where the cycles go in modern data centers, which are increasingly where most of computing happens. Almost 20% of CPU time is spent in the operating system, 5% in the scheduler alone!*

[J09] "Inside The Linux 2.6 Completely Fair Scheduler" by M. Tim Jones. December 15, 2009. `http://ostep.org/Citations/inside-cfs.pdf`. *A simple overview of CFS from its earlier days. CFS was created by Ingo Molnar in a short burst of creativity which led to a 100K kernel patch developed in 62 hours.*

[KL88] "A Fair Share Scheduler" by J. Kay and P. Lauder. CACM, Volume 31 Issue 1, January 1988. *An early reference to a fair-share scheduler.*

[WW94] "Lottery Scheduling: Flexible Proportional-Share Resource Management" by Carl A. Waldspurger and William E. Weihl. OSDI '94, November 1994. *The landmark paper on lottery scheduling that got the systems community re-energized about scheduling, fair sharing, and the power of simple randomized algorithms.*

[W95] "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management" by Carl A. Waldspurger. Ph.D. Thesis, MIT, 1995. *The award-winning thesis of Waldspurger's that outlines lottery and stride scheduling. If you're thinking of writing a Ph.D. dissertation at some point, you should always have a good example around, to give you something to strive for: this is such a good one.*

[W02] "Memory Resource Management in VMware ESX Server" by Carl A. Waldspurger. OSDI '02, Boston, Massachusetts. *The paper to read about memory management in VMMs (a.k.a., hypervisors). In addition to being relatively easy to read, the paper contains numerous cool ideas about this new type of VMM-level memory management.*

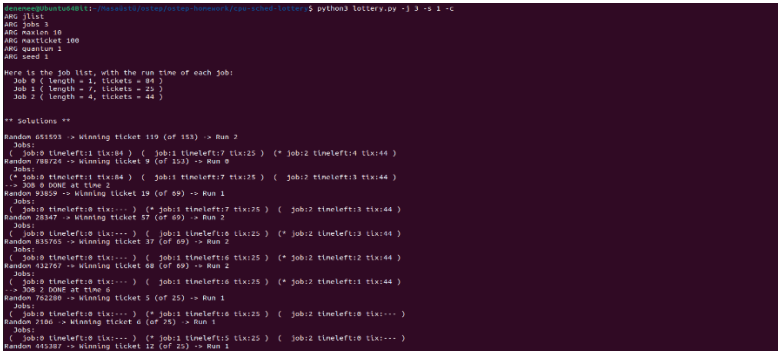
Ev Ödevi (Simülasyon)

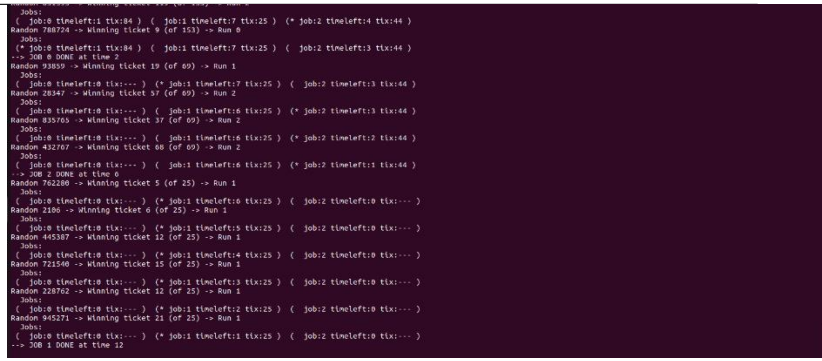
Bu program, `lottery.py`, bir piyango zamanlayıcısının nasıl çalıştığını görmenizi sağlar. Ayrıntılar için README'ye bakın.

Questions

1. 3 iş ve 1,2,3 tohum değerleri ile simülasyon için çözümleri hesaplayın.
2. Şimdi iki özel işle çalıştırın: her biri 10 uzunluğunda, ancak biri (iş 0) sadece 1 biletle ve diğeri (iş 1) 100 biletle (örneğin, `-l 10:1,10:100`).
   Bilet sayısı bu kadar dengesiz olduğunda ne olur? İş 0, iş 1 tamamlanmadan önce hiç çalışacak mı? Ne sıklıkla çalışır? Genel olarak, böyle bir bilet dengesizliği piyango zamanlama davranışına ne yapar?
3. Uzunluğu 100 olan iki işle ve 100'`lük` eşit bilet tahsisleriyle çalışırken (`-l 100:100,100:100`), zamanlayıcı ne kadar adaletsizdir? (Olasılıklı) cevabı belirlemek için bazı farklı rastgele tohumlarla çalıştırın; adaletsizlik, bir işin diğerinden ne kadar önce bittiğine göre belirlensin.
4. Kuantum boyutu (`-q`) büyüdükçe bir önceki soruya verdiğiniz yanıt nasıl değişiyor?
5. Bölümde bulunan grafiğin bir versiyonunu yapabilir misiniz? Başka neler keşfetmeye değer olabilir? Bir adım zamanlayıcı ile grafik nasıl görünürdü?

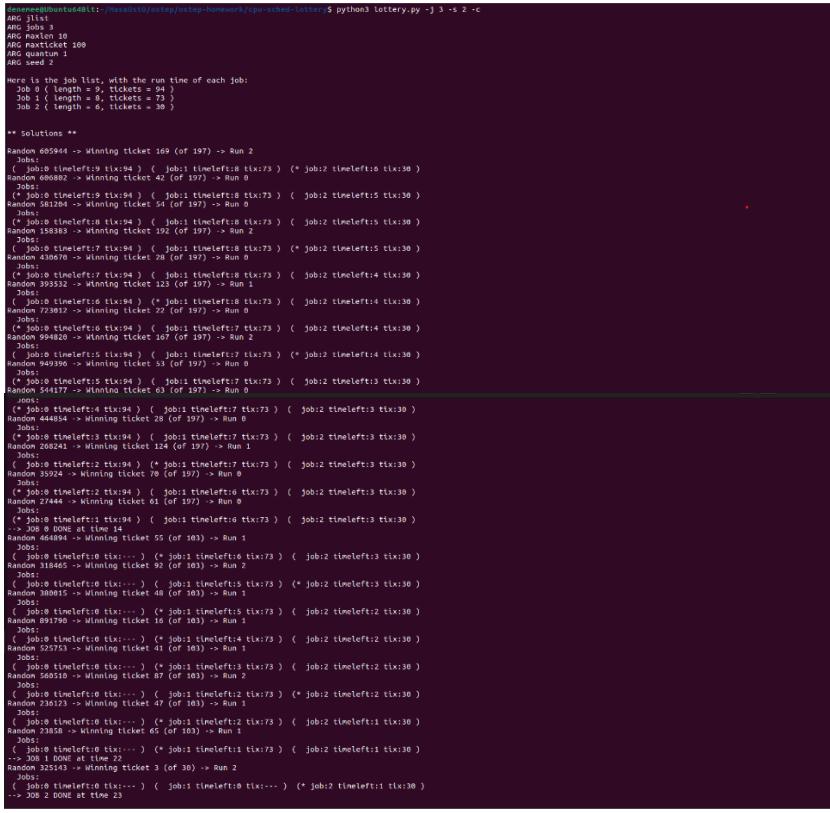
1.1 Öncelikle seed(tohum) değerimizi 1 olarak ayarlayıp elde ettiğimiz sonuç :



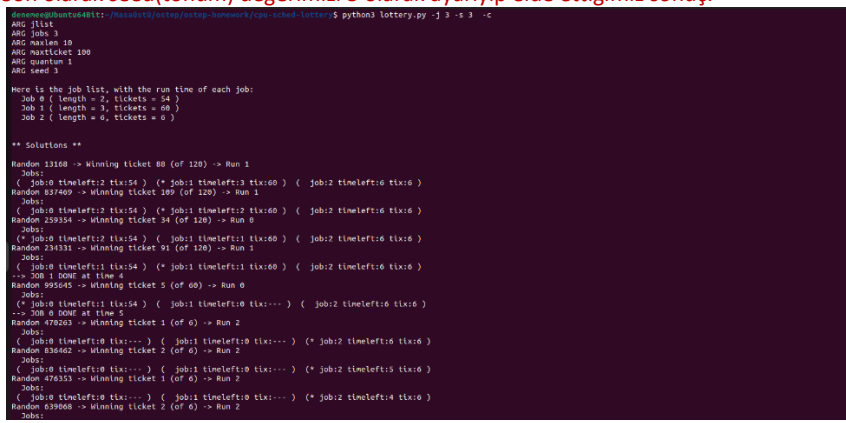


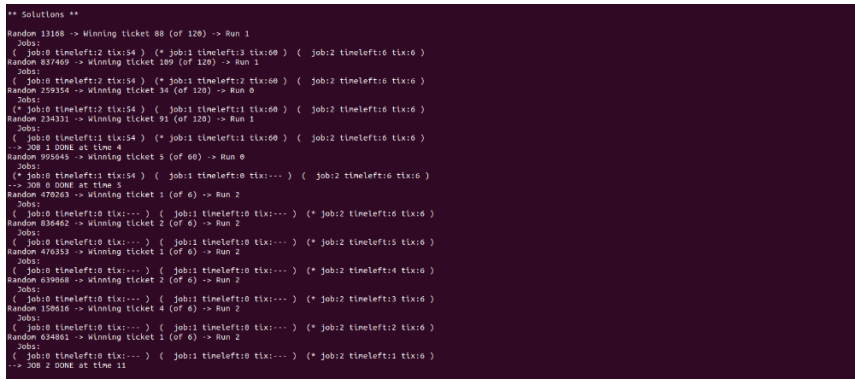
(Not:1.1 için iki görseli bir şekilde aynı yere orantılıyamadım bundan dolayı aşağı kaydı)

1.2 Seed(tohum) değerimizi 2 olarak girdiğimizde elde ettiğimiz sonuç :



1.3 Son olarak seed(tohum) değerimizi 3 olarak ayarlyıp elde ettiğimiz sonuç:



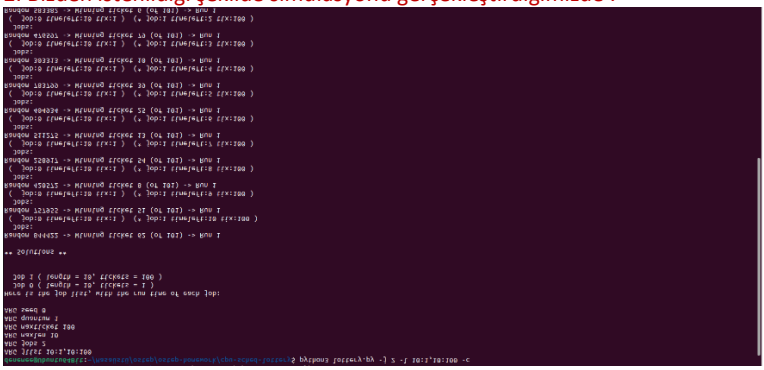


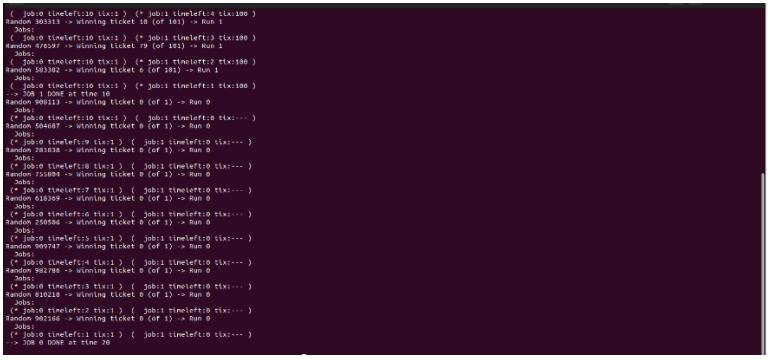
-Öncelikle bu kodumuzdaki seed parametremiz rastegele sayı akışını control etmektedir. Eğer seed değeri girmezsek aynı kodu aynı şekilde çalıştırsak dahi farklı sonuçlar elde ederiz.

-Seed değerimizi belirlediğimizde örneğin 1, 2 veya 3 ya da farklı bir sayı bu değer ile ne kadar çalıştırırsak çalıştıralım her zaman aynı değerleri alırız.

-Yukarıda görüleceği üzere seed değeri değiştiğinde işlerin tamamlanma süreleri de değişmiştir.

2. Bizden istenildiği şekilde simülasyonu gerçekleştirdiğimizde :





- Öncelikle lottery scheduling yapısından basitçe bahsedersek bütün processlere sistem kaynakları için piyango bileti dağıtılıyor sonrasında çekiliş yapılıyor ve kazanan processler ödül olarak kaynağı kullanıyor.

-Burada da yine öncelikli process'lerin şansı var. Örneğin her process 10 bilet alabiliyorken, öncelikli process'lere 20 bilet veriliyor. Kazanma şansları kasten arttırılıyor. Dolayısıyla, biletlerin %n oranına sahip process, uzun vadede, kaynakların da %n'ine sahip oluyor.

-Bundan dolayı da bilet sayısı bu kadar dengesiz olduğu zamanda bilet sayısı daha fazla olan iş kaynağı daha fazla kullanacaktır. Ayrıca bilet sayısı fazla olan iş bitene kadar diğer işlem çalışmayacaktır.

-Bu şekilde olan bir bilet dengesizliğinde bilet sayısı az olan işlem daha fazla olan işlem bitene kadar çalışmayacak bu durumdan dolayı da bekleme süresi artıp daha yavaş çalışacaktır.Bu tarz bir durum piyango zamanlamayı olumsuz şekilde etkiler.

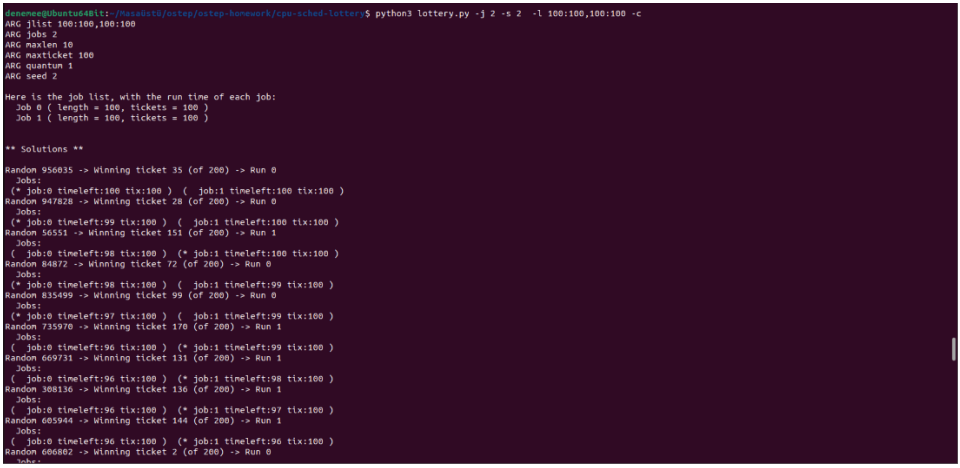
3- Öncelikle bizden istenildiği gibi farklı seed değerleri ile simüle edelim.

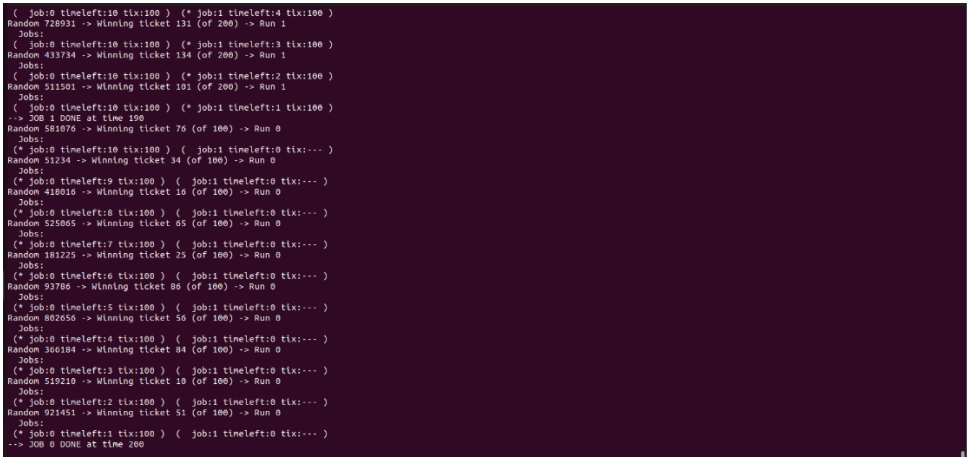
Seed değerini 0 olarak ayarladığımızda aldığımız sonuç :

```
denemee@Ubuntu64Bit:~/Masaüstü/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j 2 -  -l 100:100,100:100 -c
ARG jlist 100:100,100:100
ARG jobs 2
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 0

Here is the job list, with the run time of each job:
  Job 0 ( length = 100, tickets = 100 )
  Job 1 ( length = 100, tickets = 100 )


** Solutions **

Random 844422 -> Winning ticket 22 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:100 tix:100 )  (  job:1 timeleft:100 tix:100 )
Random 757955 -> Winning ticket 155 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:99 tix:100 )  (* job:1 timeleft:100 tix:100 )
Random 420572 -> Winning ticket 172 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:99 tix:100 )  (* job:1 timeleft:99 tix:100 )
Random 258917 -> Winning ticket 117 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:99 tix:100 )  (* job:1 timeleft:98 tix:100 )
Random 511275 -> Winning ticket 75 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:99 tix:100 )  (  job:1 timeleft:97 tix:100 )
Random 404934 -> Winning ticket 134 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:98 tix:100 )  (* job:1 timeleft:97 tix:100 )
Random 783799 -> Winning ticket 199 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:98 tix:100 )  (* job:1 timeleft:96 tix:100 )
Random 303313 -> Winning ticket 113 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:98 tix:100 )  (* job:1 timeleft:95 tix:100 )
Random 476597 -> Winning ticket 197 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:98 tix:100 )  (* job:1 timeleft:94 tix:100 )
Random 583382 -> Winning ticket 182 (of 200) -> Run 1
  Jobs:
```

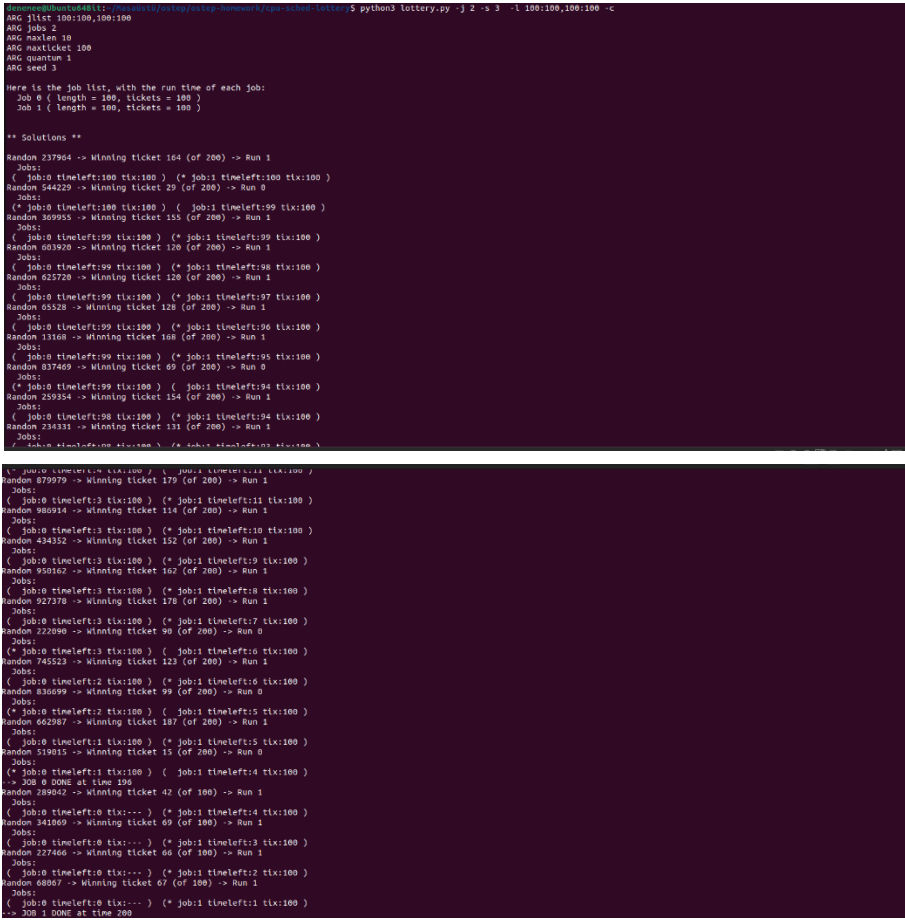
```
 (  job:0 timeleft:4 tix:100 )  (* job:1 timeleft:10 tix:100 )
Random 192309 -> Winning ticket 109 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:4 tix:100 )  (* job:1 timeleft:9 tix:100 )
Random 334402 -> Winning ticket 2 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:4 tix:100 )  (  job:1 timeleft:8 tix:100 )
Random 239416 -> Winning ticket 16 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:3 tix:100 )  (  job:1 timeleft:8 tix:100 )
Random 637400 -> Winning ticket 0 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:2 tix:100 )  (  job:1 timeleft:8 tix:100 )
Random 378648 -> Winning ticket 48 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:1 tix:100 )  (  job:1 timeleft:8 tix:100 )
--> JOB 0 DONE at time 192
Random 875424 -> Winning ticket 24 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:8 tix:100 )
Random 568151 -> Winning ticket 51 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:7 tix:100 )
Random 414406 -> Winning ticket 6 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:6 tix:100 )
Random 402267 -> Winning ticket 67 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:5 tix:100 )
Random 701830 -> Winning ticket 30 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:4 tix:100 )
Random 418226 -> Winning ticket 26 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:3 tix:100 )
Random 662196 -> Winning ticket 96 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:2 tix:100 )
Random 46779 -> Winning ticket 79 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:1 tix:100 )
--> JOB 1 DONE at time 200
```

Seed değerimizi 2 olarak ayarladığımızda aldığımız sonuç :





Seed değerimizi 3 olarak ayarladığımızda elde ettiğimiz sonuç :



Çıkan sonuçlara baktığımız zaman seed değerlerimiz değiştiğinde bitme süreleri değişse bile her seferinde birbirine yakın olduğunu görüyoruz zaten lottery scheduling de bir işin sahip olduğu bilet sayısı o işin kaynakların ne kadarını kullanacağını da belirlemeye yardımcı oluyor.

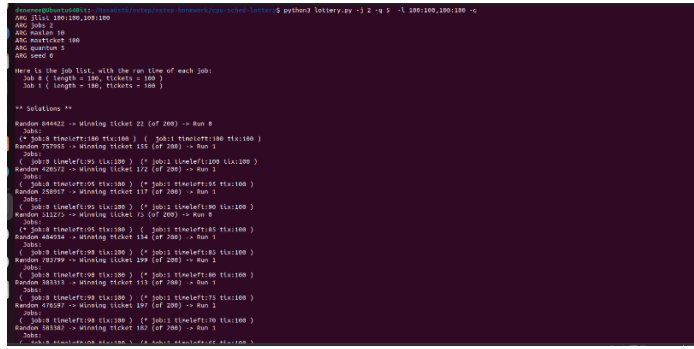
Bu nedenle aynı sayıda bilete sahip 2 işimiz olduğu için birbirine yakın zamanda tamamlanmaları mantıklı oluyor.Yani bu simülasyonun adaletinin yüksek olduğunu söyleyebiliriz.

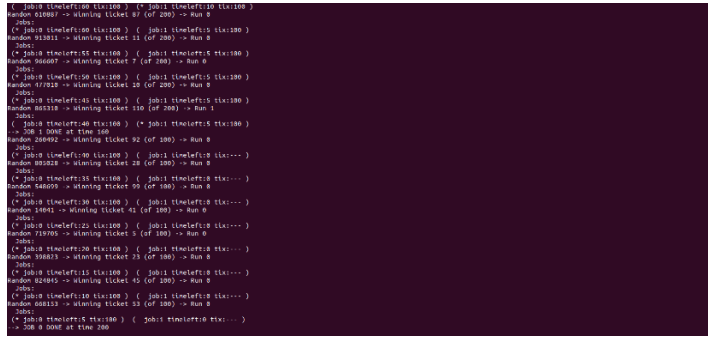
4-Öncelikle bizden istenildiği gibi q değerini değiştirmeden ve değiştirince elde ettiğimiz sonuçlara bakalım.Öncelikle q değerimizi değiştirmeden çalıştırdığımızda elde ettiğimiz sonuç:

```
denemee@Ubuntu64Bit:~/Masaüstü/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j 2 -  -l 100:100,100:100 -c
ARG jlist 100:100,100:100
ARG jobs 2
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 0

Here is the job list, with the run time of each job:
  Job 0 ( length = 100, tickets = 100 )
  Job 1 ( length = 100, tickets = 100 )


** Solutions **

Random 844422 -> Winning ticket 22 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:100 tix:100 )  (  job:1 timeleft:100 tix:100 )
Random 757955 -> Winning ticket 155 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:99 tix:100 )  (* job:1 timeleft:100 tix:100 )
Random 420572 -> Winning ticket 172 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:99 tix:100 )  (* job:1 timeleft:99 tix:100 )
Random 258917 -> Winning ticket 117 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:99 tix:100 )  (* job:1 timeleft:98 tix:100 )
Random 511275 -> Winning ticket 75 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:99 tix:100 )  (  job:1 timeleft:97 tix:100 )
Random 484934 -> Winning ticket 134 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:98 tix:100 )  (* job:1 timeleft:97 tix:100 )
Random 783799 -> Winning ticket 199 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:98 tix:100 )  (* job:1 timeleft:96 tix:100 )
Random 383313 -> Winning ticket 113 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:98 tix:100 )  (* job:1 timeleft:95 tix:100 )
Random 476597 -> Winning ticket 197 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:98 tix:100 )  (* job:1 timeleft:94 tix:100 )
Random 583302 -> Winning ticket 102 (of 200) -> Run 1
  Jobs:
```

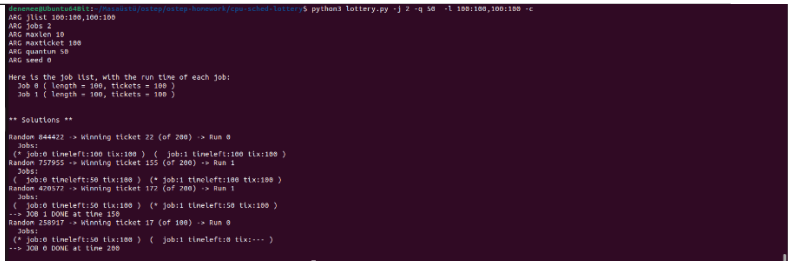
```
 (  job:0 timeleft:4 tix:100 )  (* job:1 timeleft:10 tix:100 )
Random 192309 -> Winning ticket 109 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:4 tix:100 )  (* job:1 timeleft:9 tix:100 )
Random 334402 -> Winning ticket 2 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:4 tix:100 )  (  job:1 timeleft:8 tix:100 )
Random 239416 -> Winning ticket 16 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:3 tix:100 )  (  job:1 timeleft:8 tix:100 )
Random 637400 -> Winning ticket 0 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:2 tix:100 )  (  job:1 timeleft:8 tix:100 )
Random 378648 -> Winning ticket 48 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:1 tix:100 )  (  job:1 timeleft:8 tix:100 )
 --> JOB 0 DONE at time 192
Random 875424 -> Winning ticket 24 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:8 tix:100 )
Random 568151 -> Winning ticket 51 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:7 tix:100 )
Random 414486 -> Winning ticket 6 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:6 tix:100 )
Random 402267 -> Winning ticket 67 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:5 tix:100 )
Random 701030 -> Winning ticket 30 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:4 tix:100 )
Random 418226 -> Winning ticket 26 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:3 tix:100 )
Random 662196 -> Winning ticket 96 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:2 tix:100 )
Random 46779 -> Winning ticket 79 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:1 tix:100 )
--> JOB 1 DONE at time 200
```

-Ardından q değerimizi değiştirerek programımızı çalıştıralım :





-İşlemlerin belirlenen bir süreye göre sıra sıra işleme sokulmasıdır. İşlemlerin CPU'da kalabileceği maksimum süreye **time quantum** denir. İşlemler time quantum'a göre sıra sıra işleme girer ve çıkarlar.
-Time quantumun artması ve azalmasıyla ilgili bazı problemler var ve bunlar :
-Çok yüksek bir quantum belirlersek örneğin time quantum'a 1 yıl dersek; bu bir yılda sadece bir işlem çalışacak demek oluyor yani başka bir process çalışmayacak. Tabi bu durumda işlem bir yıldan önce biteceği için. İşlem bittikten sonra diğer işlem gelecek.Bir noktadan sonra aslında iş FCFS'a dönmüş oluyor.
-Çok düşük bir quantum değeri belirlersek CPU'daki context switchlerin maliyeti artamaya başlıyor. Çünkü time quantum'u küçük belirlememiz çok sık process değişikliği yapmamıza sebep oluyor.
-Örneğimizde de quantum değerimizi artırdığımız zaman 1. Işlemin daha hızlı bitirildiğini görüyoruz.Ancak programa farklı quantum değerleri girsek bile 2. Işin bitiş süresi değişmiyor.Kendim çalıştırıken quantum değerimizi 50 yapıp tekrar denemedim ancak 2. Iş yine değişmedi(aşağıdaki görselde gösterdim.)



(quantum değerini 50 yapınca aldığım sonuç)

5-**Stride scheduling nedir öncelikle kısaca bunu açıklarsak** Sistemdeki her işin, sahip olduğu bilet sayısıyla ters orantılı bir adım aralığı vardır. Örneğin elimizde , sırasıyla 100, 50 ve 250 bilete sahip A, B ve C işleri olsun, her bir işlemin atandığı bilet sayısına büyük bir sayı bölerek her birinin adımını hesaplayabiliriz. Örneğin, 10.000'i bu bilet değerlerinin her birine bölersek, A, B ve C için şu adım değerlerini elde ederiz: 100, 200 ve 40. Bu değere her bir sürecin **adım** sayısı diyoruz; bir süreç her çalıştığında, küresel ilerlemesini izlemek için onun için bir sayacı (**geçiş** değeri olarak adlandırılır) adım sayısı kadar artıracağız.
-Bundan dolayı da tablodaki adalet oranının artmasını bekleriz.