9

Zamanlama: Orantılı Paylaşım

Bu bölümde, orantısal-paylaşım(**proportional-share**) zamanlayıcısı olarak da bilinen farklı bir zamanlayıcı türünü inceleyeceğiz, ayrıca bazen adil-paylaşım(**fair-share**) zamanlayıcısı olarak da adlandırılır. Orantısal-paylaşım basit bir konsepte dayanır: geri dönüş veya yanıt süresi için optimize etmek yerine , zamanlayıcı bunun yerine her işin belirli bir işlemci süresinin yüzdesini almasını garanti etmeye çalışacaktır.

Orantısal-paylaşım zamanlamasının mükemmel bir erken örneği Waldspurger ve Weihl[WW94] tarafından yapılan araştırmada bulundu, ve piyango-zamanlama(**lottery scheduling**) olarak bilinir; ancak, fikir kesinlikle daha eskidir[KL88]. Temel fikir oldukça basittir: sık sık, sıradaki hangi işlemin çalışması gerektiğini piyangoyla düzenleyerek belirler; daha sık çalıştırılması gereken süreçlere piyangoyu kazanmaları için daha fazla şans verilmelidir. Kolay, değil mi? Şimdi, detaylara geçelim! Ama kritik anımızdan önce değil:

Kritik an: İŞLEMCİ ORANTILI OLARAK NASIL PAYLAŞTIRILIR

İşlemciyi orantılı bir şekilde paylaşmak için nasıl bir zamanlayıcı tasarlayabiliriz? Bunu yapmak için temel mekanizmalar nelerdir? Ne kadar etkililer?

9.1 Temel Konsept: Biletler Payınızı Temsil Eder

Piyango planlamasının altında yatan çok temel bir kavram vardır: biletler(**tickets**), bir sürecin (veya kullanıcının veya her neyse) alması gereken bir kaynağın payını temsil etmek için kullanılır. Bir sürecin sahip olduğu bilet yüzdesi, söz konusu sistem kaynağından aldığı payı temsil eder.

Bir örneğe göz atalım. İki süreç hayal edin, A ve B, ve dahası A'nın 75 bileti varken B'nin sadece 25 bileti var. Dolayısıyla, istediğimiz şey A'nın İşlemcinin %75'ini, B'nin ise kalan %25'ini alması.

Piyango zamanlaması bunu olasılıksal olarak (ancak deterministik olarak değil) sık sık (örneğin her zaman diliminde) bir piyango düzenleyerek başarır. Bir piyango düzenlemek basittir: zamanlayıcı toplam kaç bilet olduğunu bilmelidir (örneğimizde 100 bilet vardır). Planlayıcı daha sonra şunları seçer

İPUCU: RASTGELELİK KULLANIN

Piyango planlamasının en güzel yönlerinden biri rastgeleliği(**ran domness**) Bir karar vermeniz gerektiğinde, bu tür rastgele bir yaklaşım kullanmak genellikle bunu yapmanın sağlam ve basit bir yoludur.

Rastgele yaklaşımların daha geleneksel kararlara göre en az üç avantajı vardır. İlk olarak, rastgele genellikle daha geleneksel bir algoritmanın işlemede sorun yaşayabileceği garip köşe durumu davranışlarından kaçınır. Örneğin, LRU(Least Recently Used(en az kullanılmayan öğe)) değiştirme politikasını ele alalım (ileride sanal bellekle ilgili 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, rastgele de hafif bir yaıya sahiptir, alternatifleri izlemek için çok az durum gerektirir. Geleneksel adil paylaşım zamanlama algoritmasında, her bir sürecin ne kadar CPU aldığını takip etmek, her bir süreci çalıştırdıktan sonra güncellenmesi gereken süreç başına hesaplama gerektirir. Bunu rastgele yapmak, yalnızca işlem başına en az durumu 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. Tabii ki, ihtiyaç ne kadar hızlı olursa, sözde rastgele eğilimi de o kadar artar.

bilet kazanmak, 0 dan 99[1] e kadar olan bir sayıdır. A'nın 0'dan 74'e ve B 75'ten 99'a kadar olan biletleri tuttuğunu varsayarsak, kazanan bilet sadece A veya B'nin çalışıp çalışmadığını belirler. Zamanlayıcı daha sonra kazanan sürecin durumunu yükler ve onu çalıştırır.

İşte bir piyango planlayıcısının 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 zamanlama:

```
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 planlamasında rastgeleliğin kullanılması, istenen oranın 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. Ancak, bu iki iş ne kadar uzun süre rekabet ederse, istenen yüzdelere ulaşma olasılıkları da 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 TEMSİL ETMEK İÇİN BİLETLERİ KULLANIN

Piyango (ve adım) planlamasının tasarımındaki en güçlü (ve temel) mekanizmalardan biri bilettir(**Ticket**). Bilet, bu örneklerde bir sürecin CPU'daki payını temsil etmek için kullanılmıştır, 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 mülkiyet oranını temsil edecek bir mekanizmaya ihtiyaç duyarsanız, bu kavram ... (bekleyin) ... tam size göre olabilir.

9.2 Bilet Mekanizmaları

Piyango planlaması ayrıca biletleri farklı ve bazen faydalı şekillerde manipüle etmek için bir dizi mekanizma sağlar.Bunun bir yolu 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.

Örneğin, A ve B kullanıcılarının her birine 100 bilet verildiğini varsayalım. A Kullanıcısı, A1 ve A2 olmak üzere iki iş yürütüyor ve her birine A'nın para biriminde 500 bilet (toplam 1000 üzerinden) veriyor. B kullanıcısı sadece 1 iş çalıştırıyor ve ona 10 bilet veriyor (toplam 10 üzerinden). Sistem, A1'in ve A2'nin tahsisatını A'nın para biriminde 500'den küresel para biriminde 50'ye dönüştürür; Benzer şekilde, B1'in 10 bileti 100 bilete dönüştürülür. Piyango daha sonra hangi işin çalıştığını belirlemek için küresel bilet para birimi (toplam 200) üzerinden yapılır.

```
  Kullanıcı A -> 500 (A'nın parası) A1 e ->  50 (küresel
                        para)
           -> 500 (A'nın parası) A2 e ->  50 (küresel
para)
    Kullanıcı B ->  10 (B'nin parası) B1 e -> 100 (küresel
para)
```

Bir diğer faydalı mekanizma ise bilet transferidir(**ticket transfer**). Transferlerde, bir işlem biletlerini geçici olarak başka bir işleme devredebilir. Bu yetenek özellikle, bir istemci işleminin bir sunucuya mesaj göndererek istemci adına bazı işler yapmasını istediği bir istemci/sunucu ortamında kullanışlıdır. İşi hızlandırmak için, istemci biletleri sunucuya iletebilir ve böylece sunucu istemcinin isteğini yerine getirirken sunucunun performansını en üst düzeye çıkarmaya çalışabilir. Bittiğinde, sunucu biletleri istemciye geri aktarır ve her şey eskisi gibi olur.

Son olarak, bilet şişirme(**ticket inflation**) bazen faydalı bir teknik olabilir. Şişirmeyle birlikte, bir süreç sahip olduğu bilet sayısını geçici olarak artırabilir veya azaltabilir. Elbette, birbirine güvenmeyen süreçlerin olduğu rekabetçi bir senaryoda bu pek mantıklı değildir; açgözlü bir süreç kendisine çok sayıda bilet verebilir ve makineyi ele geçirebilir. Şişirme daha ziyade, 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  // sayaç: kazananı henüz bulup bulmadığımızı takip etmek
2  //   için kullanılır
3  int counter = 0;
4
5  //kazanan: 0 ile toplam bilet sayısı arasında bir değer elde
6  //   etmek için rastgele bir sayı üreteci çağrısı kullanın
7  int winner = getrandom(0, totaltickets);
8
9  // güncel: iş listesinde gezinmek için bunu kullanın
10 node_t *current = head;
11 while (current) {
12 counter = counter + current->tickets;
13 if (counter > winner)
14 break; // found the winner
15 current = current->next;
16 }
17 // 'current' is the winner: schedule it...
```

Şekil 9.1: **Lottery Scheduling Decision Code**

9.3 Uygulama

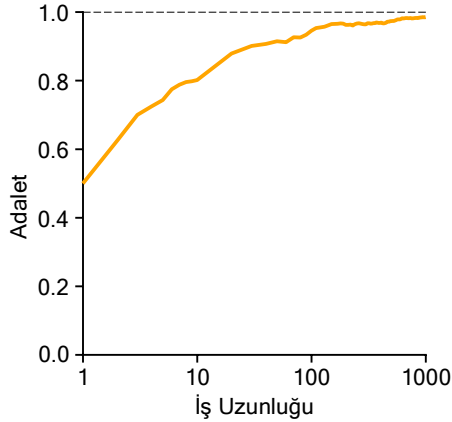
Muhtemelen piyango planlamasıyla ilgili en şaşırtıcı şey, uygulamasının basitliğidir. İhtiyacınız olan tek şey kazanan bileti seçmek için iyi bir rastgele sayı üreteci, sistemin süreçlerini izlemek için bir veri yapısı (örneğin, bir liste) ve toplam bilet sayısıdır.

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

Bir planlama kararı vermek için öncelikle toplam bilet sayısından (400)[2] rastgele bir sayı (kazanan) seçmemiz gerekir. 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 bilet değerini sayaca ekleyerek işlem listesinde yürür. Durum böyle olduğunda, geçerli liste öğesi kazanan olur. Kazanan biletin 300 olduğu örneğimizde, aşağıdakiler gerçekleşir. İlk olarak, A'nın biletlerini hesaba katmak 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 tekrar devam ederiz. Son olarak, sayaç 400'e güncellenir (300'den açıkça daha büyük) ve böylece C'ye (kazanan) geçerli noktalama ile döngüden çıkarız.

[2]Surprisingly, as pointed out by Björn Lindberg, this can be challenging to do correctly; for more details, see `http://stackoverflow.com/questions/2509679/how-to-generate-a-random-number-from-within-a-range`.



Şekil 9.2: **Lottery Fairness Study**

Bu süreci en verimli hale getirmek için, genellikle listeyi en yüksek bilet sayısından en düşüğe doğru sıralı bir şekilde düzenlemek en iyisi olabilir. Sıralama algoritmanın doğruluğunu etkilemez; ancak, özellikle biletlerin çoğuna sahip olan birkaç süreç varsa, genel olarak en az sayıda liste yinelemesinin yapılmasını sağlar.

Bu farkı ölçmek için, basit bir adalet metriği tanımlarız, F basitçe ilk işin tamamlandığı zamanın, ikinci işin tamamlandığı zamana bölünmesidir.

9.4 Bir Örnek

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

Bu senaryoda, her işin aşağı yukarı aynı zamanda bitmesini istiyoruz, ancak piyango çizelgelemesinin rastlantısallığı nedeniyle bazen bir iş diğerinden önce bitiyor. Bu farkı ölçmek için, basit bir adalet metriği(**fairness metric**) tanımlarız, F basitçe ilk işin tamamlandığı zamanın, ikinci işin tamamlandığı zamana bölünmesidir. 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.

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. Yalnızca işler önemli sayıda zaman dilimi için çalıştığında piyango zamanlayıcısı istenen adil sonuca yaklaşır.

9.5 Bilet Ataması Nasıl Yapılır?

Piyango planlamasıyla ilgili 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. Bir yaklaşım, kullanıcıların en iyisini bildiğini varsaymaktır; böyle bir durumda, her kullanıcıya belirli sayıda bilet verilir ve bir kullanıcı çalıştırdığı herhangi bir işe istediği gibi bilet tahsis edebilir. Ancak, bu çözüm bir çözüm değildir: size gerçekten ne yapmanız gerektiğini söylemez. Böylece, bir dizi iş verildiğinde, "bilet atama problemi" açık kalır.

9.6 Adım Planlaması

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

Adım planlaması da oldukça basittir. Sistemdeki her işin, sahip olduğu bilet sayısıyla ters orantılı bir adımı vardır. Yukarıdaki örneğimizde, sırasıyla 100, 50 ve 250 bilete sahip A, B ve C işleri ile, 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 sürecin adım(**stride**) değeri diyoruz; bir süreç her çalıştığında, küresel ilerlemesini izlemek için onun için bir sayacı (geçiş(**pass**) değeri olarak adlandırılır) adım değeri kadar artıracağız.

Zamanlayıcı daha sonra hangi sürecin daha sonra çalışması gerektiğini belirlemek için stride ve pass değerlerini kullanır. Temel fikir basittir: herhangi bir zamanda, şimdiye kadarki en düşük geçiş değerine sahip olan çalıştırma işlemini seçin; Bir işlemi çalıştırdığınızda, geçiş sayacını adımlarıyla artırın. Waldspurger [W95] tarafından bir sözde kod uygulaması sağlanmıştır:

```
curr = remove_min(queue);    //en az geçişli müşteri seçin
schedule(curr);              //miktar için çalıştır
curr->pass += curr->stride; //adım kullanarak geçişi güncelle
insert(queue, curr);         //curr'u kuyruğa döndür
```

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

geçişini 80 olarak güncelleyerek çalıştıracaktır (hatırladığınız gibi C'nin adımı 40'tır). Ardından C tekrar çalışacak (hala en düşük geçiş değeri) ve geçiş değerini 120'ye yükseltecektir. A şimdi çalışacak ve geçişini 200'e (şimdi B'ninkine eşit) güncelleyecektir. Ardından C iki kez daha çalışarak geçişini 160'a ve ardından 200'e güncelleyecektir. Bu noktada, tüm geçiş değerleri tekrar eşit olur ve süreç sonsuza kadar tekrar eder. Şekil 9.3 zamanlayıcı davranışını zaman içinde izler.

Şekilden de görebileceğimiz gibi, C beş kez, A iki kez ve B sadece bir kez, tam olarak 250, 100 ve 50 bilet değerleriyle orantılı olarak çalışmıştır. Piyango çizelgeleme, zaman içinde olasılıksal olarak oranlara ulaşır; adım çizelgeleme, her çizelgeleme döngüsünün sonunda bunları tam olarak doğru hale getirir.

Merak ediyor olabilirsiniz: Adım çizelgelemenin hassasiyeti göz önüne alındığında, neden kura çizelgeleme kullanılsın ki? Piyango çizelgelemenin, adım çizelgelemenin sahip olmadığı güzel bir özelliği vardır: küresel durum yoktur. Yukarıdaki adım çizelgeleme örneğimizin ortasında yeni bir işin girdiğini düşünün; geçiş değeri ne olmalıdır? 0'a ayarlanmalı mı? Eğer öyleyse, CPU'yu tekeline alacaktır. Piyango çizelgelemede, süreç başına global bir durum yoktur; sadece sahip olduğu biletlerle yeni bir süreç ekleriz, toplam kaç biletimiz olduğunu izlemek için tek global değişkeni güncelleriz ve oradan devam ederiz. Bu şekilde piyango, yeni süreçlerin mantıklı bir şekilde dahil edilmesini çok daha kolay hale getirir.

Completely Fair Scheduler (veya CFS) [J09] olarak adlandırılan zamanlayıcı, adil paylaşım zamanlamasını uygular, ancak bunu oldukça verimli ve ölçeklenebilir bir şekilde yapar.

9.7 Linux Tamamen Adil Zamanlayıcı (CFS)

Adil paylaşımlı çizelgelemedeki 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 paylaşım zamanlamasını uygular, ancak bunu oldukça verimli ve ölçeklenebilir bir şekilde yapar.

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

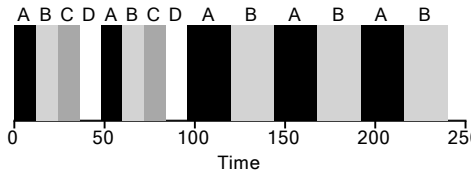


Figure 9.4: **CFS Simple Example**

Temel Operasyon

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 sayma tabanlı teknikle yapar.

Her işlem 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 çalıştırmak için en düşük `vruntime`'a sahip işlemi seçecektir.

Bu da bir soruyu gündeme getiriyor: Zamanlayıcı o anda çalışmakta olan süreci ne zaman durduracağını ve bir sonraki süreci ne zaman çalıştıracağını nasıl biliyor? Buradaki gerilim açıktır: CFS çok sık geçiş yaparsa, CFS her sürecin çok küçük zaman pencerelerinde bile CPU'dan pay almasını sağlayacağından adalet artar, ancak performans pahasına (çok fazla bağlam değiştirme); CFS daha az sıklıkta geçiş yaparsa, performans artar (daha az bağlam değiştirme), ancak yakın vadeli adalet pahasına.

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 (etkin bir şekilde zaman dilimini belirler, ancak dinamik bir şekilde). Tipik bir zamanlama gecikme(`sched latency`) değeri 48'dir (milisaniye); CFS bu değeri CPU'da 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 süreç varsa, CFS 12 ms'lik bir süreç başına zaman dilimine ulaşmak için zamanlama gecikmesi(`sched latency`) 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, var ve CFS diğer üç işten birine geçecek ve bu böyle devam edecek. 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 kalmaktadır ve bu işlerin her biri round-robin tarzında 24 ms boyunca çalışmaktadır.

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 min granularity adında başka bir parametre ekler. CFS, bir işlemin zaman dilimini asla bu değerden daha düşük bir değere ayarlamaz ve

böylece zamanlama ek yükü 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 ayarlayacaktır. CFS, 48 ms'lik hedef zamanlama gecikmesi (sched latency) üzerinde (tam olarak) mükemmel bir şekilde adil olmayacak olsa da, yine de yüksek CPU verimliliği elde ederken yakın olacaktır.

CFS'nin periyodik bir zamanlayıcı kesmesi kullandığını unutmayın, bu da yalnızca sabit zaman aralıklarında karar verebileceği anlamına gelir. Bu kesme sık sık (örneğin her 1 ms'de bir) gerçekleşerek 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 (Nicelik)

CFS ayrıca işlem önceliği üzerinde kontroller sağlayarak kullanıcıların veya yöneticilerin bazı işlemlere CPU'dan daha yüksek pay vermesine olanak tanır. Bunu biletlerle değil, bir sürecin iyi(**nice**) seviyesi olarak bilinen klasik bir UNIX mekanizması aracılığıyla yapar. Nice parametresi bir süreç için -20 ila +19 arasında herhangi bir yere ayarlanabilir, varsayılan değer 0'dır. Pozitif iyi değerler daha düşük önceliğe, negatif değerler ise daha yüksek önceliğe işaret eder; çok iyi olduğunuzda, ne yazık ki o kadar fazla (zamanlama) ilgi görmezsiniz.

CFS, burada gösterildiği gibi her bir işlemin güzel 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, ona -5 gibi güzel

bir değer atayarak daha yüksek bir öncelik verilir; B, ondan nefret ettiğimiz için3, sadece varsayılan önceliğe sahiptir (iyi değer 0'a eşittir). Bu, ağırlıkA'nın (ta- ble'den) 3121, ağırlıkB'nin ise 1024 olduğu anlamına gelir. Daha sonra her bir işin zaman dilimini hesaplarsanız, A'nın zaman diliminin yaklaşık 3/4 olduğunu görürsünüz zamanlama gecikmesi (dolayısıyla 36 ms) ve B'nin yaklaşık 1/4'ü (dolayısıyla, 12 ms).

zaman dilimi hesaplamasının genelleştirilmesine ek olarak, CFS'nin vruntime hesaplama şekli de uyarlanmalıdır. İşte i işleminin tahakkuk ettiği gerçek çalışma süresini (runtimei) alan ve 1024'ün varsayılan ağırlığını (weight0) kendi ağırlığı olan weighti'ye bölerek işlemin ağırlığıyla ters orantılı olarak ölçeklendiren yeni formül. 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 oluşturulmasının akıllıca bir yönü, güzel değerlerdeki fark sabit olduğunda tablonun CPU orantılılık oranlarını korumasıdır. Örneğin, A süreci bunun yerine 5 (-5 değil) ve B süreci de 10 (0 değil) gibi güzel bir değere sahip olsaydı, CFS bunları daha önce olduğu gibi tam olarak aynı şekilde programlardı. Nedenini görmek için matematiği kendiniz çalıştırın.

Kırmızı-Siyah Ağaçları Kullanma

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 işlemden oluşur ve bu nedenle her milisaniyede bir uzun bir listede arama yapmak israftır.

CFS bunu süreçleri kırmızı-siyah(**red-black tree**) ağacında tutarak ele alır [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 daha fazla 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 işleminin tamamlanmasını veya 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.

[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.

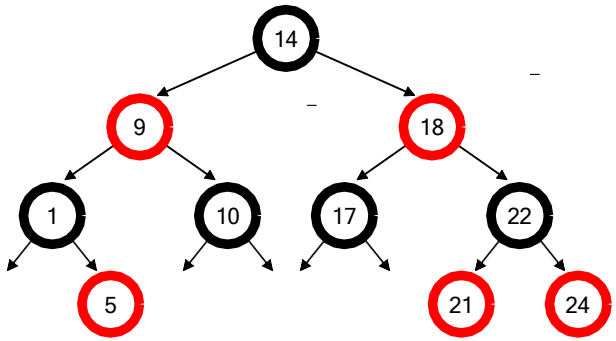


Figure 9.5: **CFS Red-Black Tree**

Ancak bu işi listeye geri yerleştirirken (sırayla), listeyi taramamız ve yerleştirmek için doğru noktayı aramamız gerekir, bu da bir O(n) işlemidir.

Şekil 9.5'te gösterildiği gibi, kırmızı-siyah ağaçta aynı değerleri tutmak çoğu işlemi daha verimli hale getirir. İşlemler ağaçta vruntime ile sıralanır ve çoğu işlem (ekleme ve silme gibi) zamanda logaritmiktir, yani O(log n). N binlerce olduğunda, logaritmik doğrusaldan belirgin şekilde daha verimlidir.

G/Ç ve Uyku Süreçleri ile Başa Çıkma

Bir sonraki çalıştırma için 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) uykuya geçen A ve B olmak üzere iki süreç düşünün. B uyandığında, vruntime'ı A'nınkinden 10 saniye geride olacak ve bu nedenle (dikkatli olmazsak) B, A'yı etkili bir şekilde aç bırakarak yetişirken önümüzdeki 10 saniye boyunca CPU'yu tekeline alacak.

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'nin, kitabın bu noktasında tartışılamayacak kadar çok sayıda başka özelliği vardır. Önbellek performansını iyileştirmek için çok sayıda sezgisel yöntem içerir, birden fazla CPU'yu etkili bir şekilde ele almak için stratejilere sahiptir (kitabın ilerleyen bölümlerinde tartışıldığı gibi), büyük süreç grupları arasında programlama yapabilir

IPUCU: 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 çalışmamaktadır. Bu tür sistemler binlerce aktif işlem içerir; her birkaç milisaniyede bir her çekirdekte çalışacak 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 doğru yapıyı uygulayabileceksiniz.

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

9.8 Özet

Orantılı paylaşım çizelgeleme kavramını tanıttık ve üç yaklaşımı kısaca tartıştık: piyango çizelgeleme, adım çizelgeleme ve Linux'un Tamamen Adil Çizelgeleyicisi (CFS). Piyango, orantılı paylaşıma ulaşmak için rastgeleliği akıllıca bir şekilde kullanır; stride ise bunu caydırıcı bir şekilde yapar. Bu bölümde tartışılan tek "gerçek" zamanlayıcı olan CFS, dinamik zaman dilimleri ile ağırlıklı round-robin'e benzer, ancak yük altında ölçeklendirmek ve iyi performans göstermek için inşa edilmiştir; bildiğimiz kadarıyla, bugün var olan en yaygın kullanılan adil paylaşım zamanlayıcısıdır.

Hiçbir çizelgeleyici her derde deva değildir ve adil paylaşımlı çizelgeleyicilerin 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 problemi açık bırakmalarıdır, yani tarayıcınıza kaç bilet ayrılması gerektiğini ya da metin editörünüzü hangi güzel değere ayarlamanız gerektiğini nasıl bilebilirsiniz? 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 dağıtılabilir.

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

Referans

[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 (Simulasyon)

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

Sorular

1. 3 iş ve 1, 2 ve 3 rastgele tohum içeren simülasyonlar için çözümleri hesaplayın.

```
** Solutions **

Random 13168 -> Winning ticket 88 (of 120) -> Run 1
  Jobs:
 (  job:0 timeleft:2 tix:54 )  (* job:1 timeleft:3 tix:60 )  (  job:2 timeleft:6 tix:6 )
Random 837469 -> Winning ticket 109 (of 120) -> Run 1
  Jobs:
 (  job:0 timeleft:2 tix:54 )  (* job:1 timeleft:2 tix:60 )  (  job:2 timeleft:6 tix:6 )
Random 259354 -> Winning ticket 34 (of 120) -> Run 0
  Jobs:
 (* job:0 timeleft:2 tix:54 )  (  job:1 timeleft:1 tix:60 )  (  job:2 timeleft:6 tix:6 )
Random 234331 -> Winning ticket 91 (of 120) -> Run 1
  Jobs:
 (  job:0 timeleft:1 tix:54 )  (* job:1 timeleft:1 tix:60 )  (  job:2 timeleft:6 tix:6 )
--> JOB 1 DONE at time 4
Random 995645 -> Winning ticket 5 (of 60) -> Run 0
  Jobs:
 (* job:0 timeleft:1 tix:54 )  (  job:1 timeleft:0 tix:--- )  (  job:2 timeleft:6 tix:6 )
--> JOB 0 DONE at time 5
Random 470263 -> Winning ticket 1 (of 6) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:0 tix:--- )  (* job:2 timeleft:6 tix:6 )
Random 836462 -> Winning ticket 2 (of 6) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:0 tix:--- )  (* job:2 timeleft:5 tix:6 )
Random 476353 -> Winning ticket 1 (of 6) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:0 tix:--- )  (* job:2 timeleft:4 tix:6 )
Random 639068 -> Winning ticket 2 (of 6) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:0 tix:--- )  (* job:2 timeleft:3 tix:6 )
Random 150616 -> Winning ticket 4 (of 6) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:0 tix:--- )  (* job:2 timeleft:2 tix:6 )
Random 634861 -> Winning ticket 1 (of 6) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:0 tix:--- )  (* job:2 timeleft:1 tix:6 )
--> JOB 2 DONE at time 11
```

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? Genel olarak, böyle bir bilet dengesizliği piyango çizelgeleme davranışına ne yapar?

   Daha fazla bileti olan bir işin, daha az bileti olan bir işe göre çalıştırılmak üzere seçilme olasılığı çok daha yüksek olacaktır. Bu durum, daha az bileti olan işin daha az CPU zamanı almasına ve potansiyel olarak daha uzun sürede tamamlanmasına neden olabilir. İş 1'in daha fazla çalışma fırsatı olacaktır.

   0 numaralı işin 1 numaralı iş tamamlanmadan önce çalışması pek olası değildir.

1/100

Düşük biletli işler için adil değildir.

```
  Jobs:
 (  job:0 timeleft:10 tix:1 )  (* job:1 timeleft:3 tix:100 )
Random 476597 -> Winning ticket 79 (of 101) -> Run 1
  Jobs:
 (  job:0 timeleft:10 tix:1 )  (* job:1 timeleft:2 tix:100 )
Random 583382 -> Winning ticket 6 (of 101) -> Run 1
  Jobs:
 (  job:0 timeleft:10 tix:1 )  (* job:1 timeleft:1 tix:100 )
--> JOB 1 DONE at time 10
Random 908113 -> Winning ticket 0 (of 1) -> Run 0
  Jobs:
 (* job:0 timeleft:10 tix:1 )  (  job:1 timeleft:0 tix:--- )
Random 504687 -> Winning ticket 0 (of 1) -> Run 0
  Jobs:
 (* job:0 timeleft:9 tix:1 )  (  job:1 timeleft:0 tix:--- )
Random 281838 -> Winning ticket 0 (of 1) -> Run 0
  Jobs:
 (* job:0 timeleft:8 tix:1 )  (  job:1 timeleft:0 tix:--- )
Random 755804 -> Winning ticket 0 (of 1) -> Run 0
  Jobs:
 (* job:0 timeleft:7 tix:1 )  (  job:1 timeleft:0 tix:--- )
Random 618369 -> Winning ticket 0 (of 1) -> Run 0
  Jobs:
 (* job:0 timeleft:6 tix:1 )  (  job:1 timeleft:0 tix:--- )
Random 250506 -> Winning ticket 0 (of 1) -> Run 0
  Jobs:
 (* job:0 timeleft:5 tix:1 )  (  job:1 timeleft:0 tix:--- )
Random 909747 -> Winning ticket 0 (of 1) -> Run 0
  Jobs:
 (* job:0 timeleft:4 tix:1 )  (  job:1 timeleft:0 tix:--- )
Random 982786 -> Winning ticket 0 (of 1) -> Run 0
  Jobs:
 (* job:0 timeleft:3 tix:1 )  (  job:1 timeleft:0 tix:--- )
Random 810218 -> Winning ticket 0 (of 1) -> Run 0
  Jobs:
 (* job:0 timeleft:2 tix:1 )  (  job:1 timeleft:0 tix:--- )
Random 902166 -> Winning ticket 0 (of 1) -> Run 0
  Jobs:
 (* job:0 timeleft:1 tix:1 )  (  job:1 timeleft:0 tix:--- )
--> JOB 0 DONE at time 20
```

3. Uzunluğu 100 olan ve 100'lük eşit bilet tahsisleri olan iki işle 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.

Bitiş sürelerindeki fark küçük oldu[undan, zamanlayıcı adil davranıyor demektir.

```
  Jobs:
 (  job:0 timeleft:3 tix:100 )  (* job:1 timeleft:8 tix:100 )
Random 927378 -> Winning ticket 178 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:3 tix:100 )  (* job:1 timeleft:7 tix:100 )
Random 222090 -> Winning ticket 90 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:3 tix:100 )  (  job:1 timeleft:6 tix:100 )
Random 745523 -> Winning ticket 123 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:2 tix:100 )  (* job:1 timeleft:6 tix:100 )
Random 836699 -> Winning ticket 99 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:2 tix:100 )  (  job:1 timeleft:5 tix:100 )
Random 662987 -> Winning ticket 187 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:1 tix:100 )  (* job:1 timeleft:5 tix:100 )
Random 519015 -> Winning ticket 15 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:1 tix:100 )  (  job:1 timeleft:4 tix:100 )
--> JOB 0 DONE at time 196
Random 289042 -> Winning ticket 42 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:4 tix:100 )
Random 341069 -> Winning ticket 69 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:3 tix:100 )
Random 227466 -> Winning ticket 66 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:2 tix:100 )
Random 68067 -> Winning ticket 67 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:1 tix:100 )
--> JOB 1 DONE at time 200



** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

4. Bir önceki soruya verdiğiniz yanıt, kuantum boyutu (-q) büyüdükçe nasıl değişiyor?

Daha küçük kuantum boyutu daha adildir.

5. Bölümde bulunan grafiğin bir versiyonunu yapabilir misiniz? Keşfetmeye değer başka ne olabilir? Grafik bir adım zamanlayıcı ile nasıl görünürdü?