9

Planlama : Orantılı Paylaşım

Bu bölümde, **orantılı paylaşım planlayıcı (proportional-share scheduler)** olarak bilinen ve bazen **adil paylaşım planlayıcı (fair-share scheduler)** olarak da adlandırılan farklı türde bir planlayıcıyı inceleyeceğiz. Orantılı paylaşım basit bir konsepte dayanır: geri dönüş veya yanıt süresi için optimizasyon yapmak yerine, bir planlayıcı bunun yerine her işin belirli bir CPU süresi yüzdesi almasını garanti etmeye çalışabilir.

Orantılı paylaşım planlamasının mükemmel bir modern örneği, Waldspurger ve Weihl [WW94] tarafından yapılan araştırmada bulunur ve **piyango planlayıcı (lottery scheduling)** olarak bilinir; ancak, fikir kesinlikle çok daha eskidir [KL88]. Temel fikir oldukça basit: Arada sırada hangi sürecin çalıştırılacağını belirlemek için bir çekiliş düzenir; daha sık çalışması gereken süreçlere piyangoyu kazanmaları için daha fazla şans verilmelidir. Kolay, değil mi? Şimdi detaylara geçelim! Ama püf noktamızdan önce değil:

PÜF NOKTASI: CPU ORANTILI OLARAK NASIL PAYLAŞILIR?
CPU'yu orantılı bir şekilde paylaşmak için bir planlayıcıyı nasıl tasarlayabiliriz? Bunu yapmak için kilit mekanizmalar nelerdir? Ne kadar etkilidirler?

9.1 Temel Konsept: Biletler Payınızı Temsil Eder

Biletler (ticket), piyango planlamasının altında yatan çok temel bir kavramdır. Bir sürecin (veya kullanıcının, her neyse) alması gereken kaynak payını temsil etmek için kullanılır. Bir sürecin sahip olduğu biletlerin yüzdesi, söz konusu sistem kaynağındaki payını temsil eder.

Bir örneğe bakalım. A ve B olmak üzere iki süreç düşünün ve ayrıca A'nın 75 bileti varken B'nin yalnızca 25 bileti olduğunu düşünün. Dolayısıyla, A'nın CPU'nun %75'ini ve B'nin kalan %25'ini almasını istiyoruz. Piyango planlaması, bunu olasılıksal olarak (ancak deterministik olarak değil) arada bir (diyelim ki her zaman diliminde) bir piyango düzenleyerek başarır. Bir piyango tutmak basittir: planlayıcının toplam bilet sayısını bilmesi gerekir (bizim örneğimizde 100 tane vardır). Sonrasında zamanlayıcı,

İPUCU: RASTGELELİK KULLANIMI

Piyango planlamasının en güzel yönlerinden biri, **rastgelelik (randomness)** kullanmasıdır. 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 ele almada sorun yaşayabileceği garip köşe durum davranışlarından kaçınır. Örneğin, LRU sayfa değiştirmeyi düşünün (sanal bellekle ilgili gelecekteki bir bölümde daha ayrıntılı olarak çalışılacaktır); genellikle iyi bir değiştirme algoritması olsa da, LRU bazı döngüsel-sıralı iş yükleri için kötü performans gösterir. Öte yandan rastgelenin böyle bir kötü durumu yoktur.

İkincisi, rastgele aynı zamanda hafiftir ve alternatifleri izlemek için çok az durum gerektirir. Geleneksel bir adil paylaşım planlama algoritmasında, her bir işlemin ne kadar CPU aldığını izlemek, her işlem çalıştırıldıktan sonra güncellenmesi gereken işlem başına hesaplama gerektirir. Bunu rastgele yapmak, işlem başına yalnızca 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 dolayısıyla hızın gerekli olduğu birçok yerde rastgele kullanılabilir. Tabii ki, ihtiyaç ne kadar hızlıysa, o kadar rastgele sözde rastgele olma eğilimindedir.

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

İşte bir piyango planlayıcısının kazanan biletlerinin bir örnek çıktısı:

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

Ve sonuç programı:

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

Örnekten de görebileceğiniz gibi, piyango çizelgelemede rastgelelik kullanımı, istenen oranın karşılanmasında olasılıksal bir doğruluğa yol açar, ancak garantisi yoktur. Yukarıdaki örneğimizde B, istenen %25'lik ayırma yerine yalnızca 20 zaman diliminden 4'ünü (%20) çalıştırır. Bununla birlikte, bu iki iş ne kadar uzun süre rekabet ederse, istenen yüzdelere ulaşma olasılıkları o kadar artar.

[1]Bilgisayar Bilimcileri her zaman 0'dan saymaya başlarlar. Bilgisayar ile pek ilgisi olmayan insanlar için o kadar gariptir ki bu durum, bilgisayar dünyasının ünlüleri neden böyle yaptığımızı yazmak zorunda hissettiler. [D82].

İPUCU: PAYLARI TEMSİL ETMEK İÇİN BİLET KULLANIMI

Piyango (ve adım) planlama tasarımındaki en güçlü (ve temel) mekanizmalardan biri, **biletinkidir (ticket)**. Bilet, bu örneklerde bir işlemin CPU payını temsil etmek için kullanılır, ancak çok daha geniş bir şekilde uygulanabilir. Örneğin, hipervizörler için sanal bellek yönetimi üzerine daha yakın tarihli bir çalışmada Waldspurger, biletlerin konuk işletim sisteminin bellek payını temsil etmek için nasıl kullanılabileceğini gösteriyor [W02]. Bu nedenle, sahiplik oranını temsil edecek bir mekanizmaya ihtiyacınız olursa, bu kavram tam olarak bilet olabilir.

9.2 Bilet Mekanizmaları

Piyango planlaması aynı zamanda biletleri farklı ve bazen yararlı şekillerde manipüle etmek için bir dizi mekanizma sağlar. Bunun bir yolu, bilet para birimi kavramıdır. Para birimi, bir dizi bilete sahip bir kullanıcının biletleri kendi işleri arasında istedikleri para biriminde tahsis etmesine olanak tanır; sistem daha sonra söz konusu para birimini otomatik olarak doğru küresel değere dönüştürür.

Örneğin, A ve B kullanıcılarına 100 bilet verildiğini varsayalım. A Kullanıcısı, A1 ve A2 olmak üzere iki iş çalıştırıyor ve her birine A Kullanıcısının kendi para biriminde 500 bilet (toplam 1000 üzerinden) veriyor. B Kullanıcısı yalnızca 1 iş çalıştırıyor ve ona 10 bilet veriyor (toplam 10 üzerinden). Sistem, A1'in ve A2'nin tahsisini A'nın para birimi cinsinden her biri 500'den küresel para birimi cinsinden 50'ye çevirecek; aynı şekilde B1'in 10 bileti 100 bilete dönüştürülecektir. Daha sonra hangi işin çalışacağını belirlemek için küresel bilet para birimi (toplam 200) üzerinden çekiliş yapılacaktır.

```
User A -> 500 (A'nın para birimi) to A1 ->  50 (küresel p.b.)
       -> 500 (A'nın p.b.) to A2 ->  50 (küresel p.b.)
User B ->  10 (B'nin p.b.) to B1 -> 100 (küresel p.b.)
```

Bir diğer yararlı mekanizma da **bilet transferidir (ticket transfer)**. Transferlerle, bir süreç biletlerini geçici olarak başka bir sürece devredebilir. Bu yetenek, bir istemci işleminin bir sunucuya istemci adına bazı işler yapmasını isteyen bir mesaj gönderdiği bir istemci/sunucu ayarında özellikle kullanışlıdır. İşi hızlandırmak için istemci, biletleri sunucuya iletebilir ve böylece sunucu, istemcinin isteğini işlerken sunucunun performansını en üst düzeye çıkarmaya çalışabilir. Bittiğinde, sunucu daha sonra biletleri istemciye geri aktarır ve her şey eskisi gibidir.

Son olarak, bilet şişirme bazen yararlı bir teknik olabilir. Şişirme ile bir süreç, sahip olduğu bilet sayısını geçici olarak artırabilir veya azaltabilir. Tabii ki, birbirine güvenmeyen süreçlerin olduğu rekabetçi bir senaryoda bu pek mantıklı değil; açgözlü bir süreç kendisine çok sayıda bilet verebilir ve makineyi ele geçirebilir. Aksine şişirme, bir grup sürecin birbirine güvendiği bir ortamda uygulanabilir; Böyle bir durumda, herhangi bir işlem daha fazla CPU süresine ihtiyaç duyduğunu bilirse, diğer işlemlerle iletişim kurmadan bu ihtiyacı sisteme yansıtmanın bir yolu olarak bilet değerini artırabilir.

```
1  // counter:winner'ı henüz bulup bulmadığımızı takip etmek için kullanılır
2  int counter     = 0;
3
4  // winner: 0 ile toplam bilet sayısı arasında bir değer elde etmek için
5  //  rastgele bir sayı oluşturucu kullanılır
6  int winner      = getrandom(0, totaltickets);
7
8  // current: iş listesinde gezinmek için bu kullanılır
9  node_t *current = head;
10
11 // bilet değerlerinin toplamı > winner olana kadar döngüye sokulur
12 while (current) {
13     counter = counter + current->tickets;
14     if (counter > winner)
15         break; // found the winner
16     current = current->next;
17 }
18 // kazanan 'current': planlanır...
```

Şekil 9.1: **Piyango Planlama Karar Kodu**

9.3 Uygulama

Muhtemelen piyango planlamasıyla ilgili en şaşırtıcı şey, uygulamanın basitliğidir. Tek ihtiyacınız olan kazanan bileti seçmek için iyi bir rasgele sayı üreteci, sistem süreçlerini izlemek için bir veri yapısı (örneğin bir liste) ve toplam bilet sayısı.

Süreçleri bir listede tuttuğumuzu varsayalım. İşte her biri belirli sayıda bileti olan A, B ve C olmak üzere üç süreçten oluşan bir örnek.

Bir planlama kararı vermek için, önce toplam bilet sayısından (400)[2] rastgele bir sayı (kazanan) seçmeliyiz. Diyelim ki 300 sayısını seçtik. Ardından, kazananı bulmamıza yardımcı olacak basit bir sayaç kullanarak listeyi dolaşıyoruz (Şekil 9.1).

Kod, değer `winner`'ı geçene kadar `counter`'a her bilet değerini ekleyerek süreçler listesinde yürür. Durum bu olduğunda, geçerli liste öğesi kazanır. Kazanan biletin 300 olduğu örneğimizde aşağıdakiler gerçekleşir. İlk olarak, A'nın biletlerini hesaba katmak için `counter`100'e yükseltilir; 100, 300'den küçük olduğu için döngü devam eder. Daha sonra `counter` 150'ye (B'nin biletleri) güncellenir, yine 300'ün altındadır ve böylece tekrar devam ederiz. Son olarak, `counter` 400'e (açıkça 300'den büyük) güncellenir ve böylece `current` C'yi (kazanan) gösteren döngüden çıkarız.

Bu süreci en verimli hale getirmek için, listeyi en yüksek bilet sayısından en düşüğe doğru sıralamak genellikle

[2] Şaşırtıcı bir şekilde, Björn Lindberg'in işaret ettiği gibi, bunu doğru şekilde yapmak zor olabilir; daha fazla ayrıntı için bkz: `http://stackoverflow.com/questions/2509679/how-to-generate-a-random-number-from-within-a-range`.

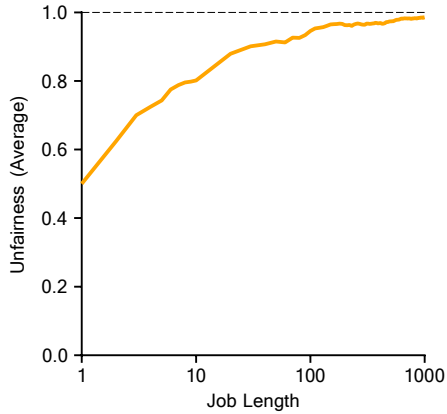


Figure 9.2: **Piyango Adalet Çalışması (Lottery Fairness Study)**

en iyisi olabilir. Sıralama, algoritmanın doğruluğunu etkilemez; ancak, özellikle biletlerin çoğuna sahip birkaç işlem varsa, genel olarak en az sayıda liste yinelemesinin yapılmasını sağlar.

9.4 Örnek

Piyango planlama dinamiklerini daha anlaşılır kılmak için, şimdi her biri aynı sayıda bilete (100) ve aynı çalışma süresine (R, değişecek olan) sahip birbiriyle rekabet eden iki işin tamamlanma süresi hakkında kısa bir çalışma yapıyoruz. .

Bu senaryoda, her işin kabaca aynı zamanda bitmesini istiyoruz, ancak piyango planlamasının rastgele olması nedeniyle bazen bir iş diğerinden önce bitiyor. Bu farkı ölçmek için, basit bir **adaletsizlik metriği (unfairness metric)**, U tanımlıyoruz; bu, basitçe ilk işin tamamlandığı zamanın ikinci işin tamamlandığı zamana bölünmesiyle elde edilir. Örneğin, R = 10 ise ve ilk iş 10 zamanında (ve ikinci iş 20'de) bitiyorsa, U = $\frac{10}{20}$ = 0,5. Her iki iş de hemen hemen aynı anda bittiğinde, U 1'e oldukça yakın olacaktır. Bu senaryoda hedefimiz budur: tamamen adil bir planlayıcı U = 1'i elde edecektir.

Şekil 9.2, iki işin uzunluğu (R) otuz denemede 1 ile 1000 arasında değiştiği için ortalama adaletsizliği gösterir (sonuçlar, bölümün sonunda sağlanan simülatör aracılığıyla üretilir). Grafikten de görebileceğiniz gibi, iş süresi çok uzun olmadığında, ortalama adaletsizlik oldukça şiddetli olabiliyor. Piyango planlayıcı, yalnızca işler önemli sayıda zaman dilimleri boyunca yürütüldüğünde istenen sonuca yaklaşır.

9.5 Biletler Nasıl Atanır?

Piyango planlamasında ele almadığımız bir sorun şudur: işlere nasıl bilet atanır? Bu sorun zordur, çünkü sistemin nasıl davranacağı, biletlerin nasıl dağıtıldığına büyük ölçüde bağlıdır. Bir yaklaşım, kullanıcıların en iyisini bildiğini varsaymaktır; böyle bir durumda, her kullanıcıya bir miktar bilet verilir ve kullanıcı, çalıştırdığı herhangi bir işe istediği şekilde bilet tahsis edebilir. Ancak bu çözüm, bir çözüm değildir: size gerçekten ne yapmanız gerektiğini söylemez. Böylece, bir dizi iş verildiğinde, "bilet atama sorunu" açık kalır.

9.6 Neden Belirleyici Değil?

Şunu da merak ediyor olabilirsiniz: neden rastgelelik kullanıyorsunuz? Yukarıda gördüğümüz gibi, rastgelelik bize basit (ve yaklaşık olarak doğru) bir planlayıcı sağlarken, bazen, özellikle kısa zaman ölçeklerinde tam olarak doğru oranları sağlamaz. Bu nedenle Waldspurger, deterministik bir adil paylaşım planlayıcı olan **adım planlamayı (stride scheduling)** icat etti [W95].

Adım planlaması da basittir. Sistemdeki her işin, sahip olduğu bilet sayısına göre ters orantılı bir adımı vardır. Yukarıdaki örneğimizde, sırasıyla 100, 50 ve 250 biletli A, B ve C işleriyle, büyük bir sayıyı her bir işleme atanan destek talebi sayısına bölerek her birinin adımını hesaplayabiliriz. Örneğin, 10.000'i bu bilet değerlerinin her birine bölersek, A, B ve C için bu adım değerlerini elde ederiz: 100, 200 ve 40. Bu değere her sürecin **adımı (stride)** diyoruz; Bir süreç her çalıştığında, küresel ilerlemesini izlemek için adım adım bir sayacı (**geçiş (pass)** değeri olarak adlandırılır) artıracağız.

Planlayıcı daha sonra hangi işlemin çalıştırılacağını belirlemek için adımı ve geçişi kullanır. Temel fikir basittir: herhangi bir zamanda, o ana kadarki en düşük geçiş değerine sahip olan çalıştırılacak süreç seçilir; bir süreç çalıştırıldığında, geçiş sayacı adım adım arttırılır. Waldspurger tarafından bir sözde kod uygulaması sağlanmıştır [W95]:

```
current = remove_min(queue);        // minimum geçişle istemci seç
schedule(current);                  // kaynağı kuantum için kullan
current->pass += current->stride;   // adımı kullanarak bir sonraki
                                    // geçişi hesapla
insert(queue, current);             // sıraya geri koyma
```

Örneğimizde, adım değerleri 100, 200 ve 40 olan ve tümü geçiş değerleri başlangıçta 0 olan üç işlemle (A, B ve C) başlıyoruz. Geçiş değerleri eşit derecede düşüktür. A'yı seçtiğimizi varsayalım (keyfi olarak; eşit düşük geçiş değerlerine sahip işlemlerden herhangi biri seçilebilir). A çalışır; zaman dilimini bitirdiğimizde geçiş değerini 100 olarak güncelleriz. Ardından geçiş değeri 200 olarak ayarlanan B'yi çalıştırırız. Son olarak geçiş değeri 40'a çıkarılan C'yi çalıştırırız. Bu noktada algoritma C'nin en düşük geçiş değerini seçin ve geçişini 80'e güncelleyerek çalıştırın (hatırladığınız gibi C'nin adımı 40'tır). Ardından C tekrar

Geçiş (A) (adım=100)	Geçiş (B) (adım=200)	Geçiş (C) (adım=400)	Hangisi Çalışır?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Table 9.1: **Adım Planlama: Bir İz (Stride Scheduling: A Trace)**

çalışacak (hala en düşük geçiş değeri), geçişini 120'ye yükseltecek. A şimdi çalışacak ve geçişini 200'e güncelleyecek (şimdi B'ninkine eşit). Ardından C iki kez daha çalışacak ve geçişini 160'a, ardından 200'e güncelleyecektir. Bu noktada, tüm geçiş değerleri tekrar eşittir ve süreç sonsuza kadar tekrarlanacaktır. Tablo 9.1 , planlayıcının zaman içindeki davranışını izler.

Tablodan da görebileceğimiz gibi, bilet değerleri olan 250, 100 ve 50 ile tam orantılı olarak C beş kez, A iki kez ve B yalnızca bir kez koştu. Piyango planlaması oranlara zaman içinde olasılıksal olarak ulaşır; adım planlama, onları her planlama döngüsünün sonunda tam olarak doğru hale getirir.

Öyleyse şunu merak ediyor olabilirsiniz: adım planlamanın kesinliği göz önüne alındığında, neden piyango planlamayı kullanasınız ki? Pekala, piyango planlamasının adım planlamasının sahip olmadığı güzel bir özelliği var: küresel durum yok. Yukarıdaki adım planlama örneğimizin ortasına yeni bir işin girdiğini düşünün; geçiş değeri ne olmalıdır? 0 olarak ayarlanmalı mı? Eğer öyleyse, CPU'yu tekelleştirecektir. Piyango planlamasında, süreç başına küresel bir durum yoktur; sadece sahip olduğu biletlerle yeni bir süreç ekleriz, tek global değişkeni toplam kaç biletimiz olduğunu takip edecek şekilde güncelleriz ve oradan devam ederiz. Bu şekilde, piyango, yeni süreçleri mantıklı bir şekilde dahil etmeyi çok daha kolaylaştırır.

9.1 Linux Tamamen Adil Planlayıcısı (CFS)

Adil paylaşım planlamasındaki bu önceki çalışmalara rağmen, mevcut Linux yaklaşımı benzer hedeflere alternatif bir şekilde ulaşıyor. **Tamamen Adil Planlayıcısı (Completely Fair Scheduler) veya (CFS)** [J09] adlı programlayıcı, adil paylaşım programlaması uygular, ancak bunu oldukça verimli ve ölçeklenebilir bir şekilde yapar.

Verimlilik hedeflerine ulaşmak için CFS, hem kendi tasarımı hem de göreve çok uygun veri yapılarının akıllı kullanımı yoluyla planlama kararları vermek için çok az zaman harcamayı amaçlar. Son çalışmalar, programlayıcı verimliliğinin şaşırtıcı derecede önemli olduğunu göstermiştir; özellikle, Google veri merkezleriyle ilgili bir çalışmada, Kanev ve diğerleri. Agresif optimizasyondan sonra bile programlamanın genel veri merkezi CPU zamanının yaklaşık %5'ini kullandığını gösterir. Bu ek yükü mümkün olduğu kadar azaltmak, modern programlayıcı mimarisinde önemli bir hedeftir.

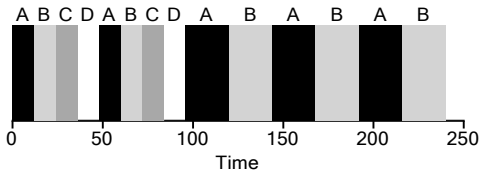


Figure 9.4: **CFS Basit Örnek**

Temel İşlem

Çoğu programlayıcı sabit bir zaman dilimi kavramına dayalı olsa da, CFS biraz farklı çalışır. Amacı basit: bir CPU'yu tüm rakip süreçler arasında adil bir şekilde bölmek. Bunu**, sanal çalışma zamanı** (**virtual runtime)** (**`vruntime`**) olarak bilinen basit bir sayım tabanlı teknik aracılığıyla yapar.

Her işlem çalışırken vruntime biriktirir. En temel durumda, her işlemin çalışma süresi, fiziksel (gerçek) zamanla orantılı olarak aynı oranda artar. Bir planlama kararı alındığında, CFS bir sonraki çalıştırma için en düşük vruntime'a sahip süreci seçecektir. Bu, bir soruyu gündeme getiriyor: Planlayıcı, halihazırda çalışan işlemi ne zaman durduracağını ve bir sonrakini çalıştıracağını nasıl biliyor? Buradaki gerilim açıktır: CFS çok sık değişirse adalet artar, çünkü CFS her işlemin CPU payını çok küçük zaman pencerelerinde bile almasını sağlar, ancak performans pahasına (çok fazla bağlam değiştirme); CFS daha az sıklıkta değişirse, performans artar (bağlam değiştirmede azalma), ancak bunun bedeli kısa vadede adalettir. CFS, bu gerilimi çeşitli kontrol parametreleri aracılığıyla yönetir. Birincisi **planlanmış gecikmedir (sched latency)**. CFS, bir işlemin bir anahtarı dikkate almadan önce ne kadar süre çalışması gerektiğini belirlemek için bu değeri kullanır (zaman dilimini etkili bir şekilde ancak dinamik bir şekilde belirler). Tipik bir planlanmış gecikme değeri 48'dir (milisaniye); CFS, bir işlemin zaman dilimini belirlemek için bu değeri CPU üzerinde çalışan işlem sayısına (n) böler ve böylece bu süre boyunca CFS'nin tamamen adil olmasını sağlar.

Örneğin, çalışan n = 4 işlem varsa, CFS, işlem başına 12 ms'lik bir zaman dilimine ulaşmak için programlanmış gecikme değerini n'ye böler. CFS daha sonra ilk işi planlar ve 12 ms (sanal) çalışma zamanı kullanana kadar onu çalıştırır ve bunun yerine çalıştırılacak daha düşük vruntime'a sahip bir iş olup olmadığını kontrol eder. Bu durumda, vardır ve CFS diğer üç işten birine geçer ve bu böyle devam eder. Şekil 9.4, dört işin (A, B, C, D) her birinin bu şekilde iki zaman diliminde çalıştığı bir örneği göstermektedir; bunlardan ikisi (C, D) daha sonra tamamlanır, geriye sadece iki tanesi kalır ve bunların her biri sırayla 24 ms çalışır.

Peki ya çalışan "çok fazla" süreç varsa? Bu, çok küçük bir zaman dilimine ve dolayısıyla çok fazla bağlam değişikliğine yol açmaz mı? İyi soru! Ve cevap evet.

Bu sorunu çözmek için CFS, genellikle 6 ms gibi bir değere ayarlanan başka bir parametre olan minimum **`ayrıntı düzeyi (min granularity)`** ekler. CFS, bir işlemin zaman dilimini hiçbir zaman

bu değerden daha düşük bir değere ayarlamaz, bu da planlama ek yükünde çok fazla zaman harcanmamasını sağlar.

Örneğin, çalışan on işlem varsa, orijinal hesaplamamız, zaman dilimini belirlemek için planlanan gecikmeyi ona böler (sonuç: 4,8 ms). Ancak, minimum ayrıntı düzeyi nedeniyle, CFS bunun yerine her işlemin zaman dilimini 6 ms olarak ayarlayacaktır. CFS, 48 ms'lik hedef programlama gecikmesi (programlanmış gecikme) konusunda (tam olarak) tam olarak adil olmasa da, yine de yüksek CPU verimliliği elde ederken buna yakın olacaktır..

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

Ağırlıklandırma (Güzellik) (Weighting (Niceness))

CFS aynı zamanda süreç önceliği üzerinde kontroller sağlayarak kullanıcıların veya yöneticilerin bazı süreçlere CPU'dan daha yüksek bir pay vermesini sağlar. Bunu biletlerle değil, sürecin güzel düzeyi olarak bilinen klasik bir UNIX mekanizması aracılığıyla yapar. nice parametresi, bir işlem için -20 ila +19 arasında herhangi bir yere ayarlanabilir ve varsayılan değeri 0'dır. Pozitif nice değerleri daha düşük önceliği, negatif değerler daha yüksek önceliği ifade eder; çok iyi olduğunuzda, ne yazık ki (planlama) kadar dikkat çekmiyorsunuz.

CFS, burada gösterildiği gibi her işlemin güzel değerini bir ağırlığa 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, (daha önce yaptığımız gibi) her sürecin etkili zaman dilimini hesaplamamıza izin verir, ancak şimdi öncelik farklılıklarını da hesaba katar. Bunu yapmak için kullanılan formül, n süreç 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, bizim en değerli işimiz olduğu için ona

-5 gibi güzel bir değer atanarak daha yüksek bir öncelik verilir; B, ondan[3] nefret ettiğimiz için, yalnızca varsayılan önceliğe sahiptir (güzel değer 0'a eşittir). Bu $\texttt{weight}_A$ (tablodan) 3121, $\texttt{weight}_B$'nin ise 1024 olduğu anlamına gelir Daha sonra her işin zaman dilimini hesaplarsanız, A'nın zaman diliminin planlanan gecikmenin yaklaşık 3/4'ü (dolayısıyla 36 ms) ve B'nin yaklaşık 1/4'ü (dolayısıyla 12 ms) olduğunu göreceksiniz. Zaman dilimi hesaplamasını genelleştirmenin yanı sıra, CFS'nin `vruntime` 'ı hesaplama yöntemi de uyarlanmalıdır. İşlem i'nin elde ettiği gerçek çalışma süresini ($\texttt{runtime}_i$) alan ve varsayılan ağırlığı 1024'ü ($\texttt{weight}_o$) kendi ağırlığı olan $\texttt{weight}_i$ 'ye bölerek işlemin ağırlığıyla ters orantıda ölçeklendiren yeni formül aşağıdadır. Çalışan örneğimizde, A'nın vruntime'ı, B'ninkinin üçte biri oranında birikecektir.

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

Yukarıdaki ağırlıklar tablosunun oluşturulmasının akıllıca bir yönü, Nice değerleri arasındaki fark sabit olduğunda tablonun CPU orantılılık oranlarını korumasıdır. Örneğin, A işleminin güzel bir değeri 5 (-5 değil) ve B işleminin güzel bir değeri 10 (0 değil) olsaydı, CFS bunları daha önce olduğu gibi tam olarak aynı şekilde planlardı. Nedenini görmek için matematiği kendiniz çalıştırın.

Kırmızı-Siyah Ağaçları Kullanmak (Using Red-Black Trees)

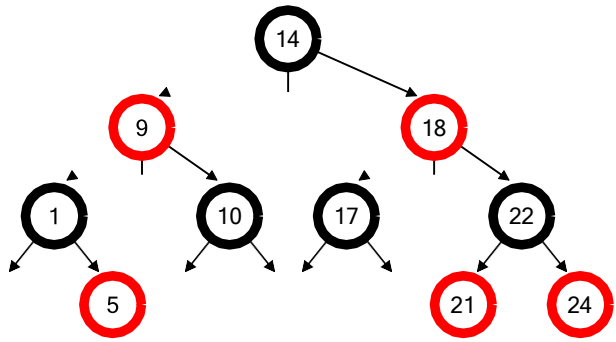
CFS'nin ana odak noktalarından biri, yukarıda belirtildiği gibi verimliliktir. Bir programlayıcı için verimliliğin pek çok yönü vardır, ancak bunlardan biri şu kadar basittir: Programlayıcı çalıştırılacak bir sonraki işi bulması gerektiğinde, bunu olabildiğince çabuk yapmalıdır. Listeler gibi basit veri yapıları ölçeklenemez: Modern sistemler bazen 1000'lerce süreçten oluşur ve bu nedenle her milisaniyede bir uzun bir listede arama yapmak boşa gider.

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

CFS tüm işlemleri bu yapıda tutmaz; daha doğrusu sadece koşma (veya çalıştırılabilir) süreçler burada tutulur. Bir işlem uyku moduna geçerse (örneğin, bir G/Ç'nin tamamlanmasını veya bir ağ paketinin gelmesini beklemek), ağaçtan kaldırılır ve başka bir yerde izlenir.

Bunu daha açık hale getirmek için bir örneğe bakalım. On iş olduğunu ve bunların aşağıdaki `vruntime` değerlerine sahip olduğunu varsayalım: 1, 5, 9, 10, 14, 18, 17, 21, 22 ve 24. Bu işleri sıralı bir listede tutarsak, bir sonrakini buluruz. çalıştırılacak iş basit olurdu: sadece ilk elemanı kaldırın. Ancak,

[3] Evet, evet, burada kasıtlı olarak kötü dilbilgisi kullanıyoruz, lütfen bir hata düzeltmesi göndermeyin. Neden? Yüzüklerin Efendisi'ne ve en sevdiğimiz anti-kahraman Gollum'a çok hafif bir gönderme, heyecanlanacak bir şey yok.



Şekil 9.5: **CFS Kırmızı-Siyah Ağaç**

bu işi (sırayla) listeye geri yerleştirirken, onu eklemek için doğru noktayı arayarak listeyi taramamız gerekirdi, bir $O(n)$ işlemi. 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) zaman açısından logaritmiktir, yani $O(\log n)$. n binlerde olduğunda, logaritmik doğrusaldan belirgin şekilde daha verimlidir.

G/Ç ve Uyku Süreçleriyle Başa Çıkma (Dealing With I/O And Sleeping Processes)

Daha sonra çalıştırılacak en düşük `vruntime`'ı seçmeyle ilgili bir sorun, uzun süre uyku moduna geçen işlerde ortaya çıkar. Biri (A) sürekli çalışan, diğeri (B) uzun bir süre (diyelim ki 10 saniye) uyku moduna geçen iki süreç, A ve B düşünün. B uyandığında, `vruntime` A'nın 10 saniye gerisinde olacak ve böylece (dikkatli olmazsak) B, onu yakalarken sonraki 10 saniye boyunca CPU'yu tekeline alacak ve A'yı fiilen aç bırakacaktır.

CFS, uyandığında bir işin `vruntime`değiştirerek bu durumu ele alır. Spesifik olarak, CFS o işin `vruntime`'ını ağaçta bulunan minimum değere ayarlar (unutmayın, ağaç yalnızca çalışan işleri içerir) [B+18]. Bu şekilde, CFS açlıktan ölmeyi önler, ancak bunun bir bedeli vardır: Kısa süreler boyunca uyuyan işler, genellikle CPU'dan adil paylarını almazlar [AC97].

Diğer CFS Eğlencesi (Other CFS Fun)

CFS'nin, kitabın bu noktasında tartışılamayacak kadar çok başka özelliği vardır. Önbellek performansını iyileştirmek için çok sayıda buluşsal yöntem içerir, birden çok CPU'yu etkili bir şekilde işlemek için stratejilere sahiptir (kitapta daha sonra tartışılacağı gibi), büyük işlem grupları arasında

İPUCU: UYGUN OLDUĞUNDA ETKİLİ VERİ YAPILARI KULLANIN
Çoğu durumda, bir liste yapacaktır. Çoğu durumda, olmayacak. Hangi veri yapısının ne zaman kullanılacağını bilmek, iyi mühendisliğin ayırt edici özelliğidir. Burada tartışılan durumda, daha önceki planlayıcılarda bulunan basit listeler, modern sistemlerde, özellikle veri merkezlerinde bulunan aşırı yüklü sunuculara iyi çalışmaz. Bu tür sistemler binlerce aktif süreç içerir; her bir çekirdekte birkaç milisaniyede bir çalışacak bir sonraki işi bulmak için uzun bir listede arama yapmak, değerli CPU döngülerini boşa harcar. Daha iyi bir yapıya ihtiyaç vardı ve CFS, kırmızı-siyah ağacın mükemmel bir uygulamasını ekleyerek bunu sağladı. Daha genel olarak, inşa etmekte olduğunuz bir sistem için bir veri yapısı seçerken, erişim modellerini ve kullanım sıklığını dikkatle değerlendirin; bunları anlayarak, eldeki görev için doğru yapıyı uygulayabileceksiniz.

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

9.2 Summary

Orantılı paylaşım planlaması kavramını tanıttık ve üç yaklaşımı kısaca tartıştık: piyango planlaması, adım planlaması ve Linux'un Tamamen Adil Planlayıcısı (CFS). Piyango, orantılı paylaşım elde etmek için rastgeleliği akıllı bir şekilde kullanır; adım bunu kesin olarak yapar. Bu bölümde tartışılan tek "gerçek" programlayıcı olan CFS, dinamik zaman dilimleri ile biraz ağırlıklı döngüye benzer, ancak ölçeklendirmek ve yük altında iyi performans göstermek için oluşturulmuştur; bildiğimiz kadarıyla, günümüzde var olan en yaygın olarak kullanılan adil paylaşım planlayıcısıdır.

Hiçbir programlayıcı her derde deva değildir ve adil paylaşım planlayıcılarının sorunlardan adil bir payı vardır. Bir sorun, bu tür yaklaşımların özellikle G/Ç [AC97] ile pek iyi örtüşmemesidir; yukarıda bahsedildiği gibi, ara sıra G/Ç gerçekleştiren işler CPU'dan adil paylarını alamayabilir. Başka bir sorun da, bilet veya öncelik ataması gibi zor bir sorunu açık bırakmalarıdır, yani, tarayıcınızın kaç bilet ayırması gerektiğini veya metin düzenleyicinizi hangi güzel değere ayarlamanız gerektiğini nasıl bilebilirsiniz? Diğer genel amaçlı programlayıcılar (daha önce tartıştığımız MLFQ ve diğer benzer Linux programlayıcıları gibi) bu sorunları otomatik olarak ele alır ve bu nedenle daha kolay dağıtılabilir.

İyi haber şu ki, bu sorunların baskın endişe olmadığı birçok alan var ve orantılı paylaşım planlayıcıları büyük etki için kullanılıyor. Örneğin, CPU döngülerinizin dörtte birini Windows VM'ye ve geri kalanını temel Linux kurulumunuza atamak isteyebileceğiniz **sanallaştırılmış (virtualized)** bir veri merkezinde (veya **bulutta (cloud)**), orantılı paylaşım basit ve etkili olabilir. Fikir ayrıca diğer kaynaklara da genişletilebilir; VMWare'in ESX Sunucusunda belleğin orantılı olarak nasıl paylaşılacağı hakkında daha fazla ayrıntı için bkz. Waldspurger [W02].

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 scheduleralone!*

[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 goodone.*

[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.*

Ödev

`lottery.py` programı, bir piyango planlayıcısının nasıl çalıştığını görmenizi sağlar.Detaylar için bkz. README.

Sorular

1. 3 iş (job) ve 1, 2 ve 3 rastgele tohumlu (seed) simülasyonlar için çözümleri hesaplayın.

-j flagı ile job sayısını (3), -s flagı ile seed (1, 2 ve 3) giriyoruz. -c flagı ile hesaplıyoruz.

-s1 için;

```
ubuntu@ubuntu:~/Desktop/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j3 -s 1 -c
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 1

Here is the job list, with the run time of each job:
  Job 0 ( length = 1, tickets = 84 )
  Job 1 ( length = 7, tickets = 25 )
  Job 2 ( length = 4, tickets = 44 )


** Solutions **

Random 651593 -> Winning ticket 119 (of 153) -> Run 2
  Jobs:
 (  job:0 timeleft:1 tix:84 )  (  job:1 timeleft:7 tix:25 )  (* job:2 timeleft:4 tix:44 )
Random 788724 -> Winning ticket 9 (of 153) -> Run 0
  Jobs:
 (* job:0 timeleft:1 tix:84 )  (  job:1 timeleft:7 tix:25 )  (  job:2 timeleft:3 tix:44 )
--> JOB 0 DONE at time 2
Random 93859 -> Winning ticket 19 (of 69) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:7 tix:25 )  (  job:2 timeleft:3 tix:44 )
Random 28347 -> Winning ticket 57 (of 69) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:6 tix:25 )  (* job:2 timeleft:3 tix:44 )
Random 835765 -> Winning ticket 37 (of 69) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:6 tix:25 )  (* job:2 timeleft:2 tix:44 )
Random 432767 -> Winning ticket 68 (of 69) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:6 tix:25 )  (* job:2 timeleft:1 tix:44 )
--> JOB 2 DONE at time 6
Random 762280 -> Winning ticket 5 (of 25) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:6 tix:25 )  (  job:2 timeleft:0 tix:--- )
Random 2106 -> Winning ticket 6 (of 25) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:5 tix:25 )  (  job:2 timeleft:0 tix:--- )
Random 445387 -> Winning ticket 12 (of 25) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:4 tix:25 )  (  job:2 timeleft:0 tix:--- )
Random 721540 -> Winning ticket 15 (of 25) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:3 tix:25 )  (  job:2 timeleft:0 tix:--- )
Random 228762 -> Winning ticket 12 (of 25) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:2 tix:25 )  (  job:2 timeleft:0 tix:--- )
Random 945271 -> Winning ticket 21 (of 25) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:1 tix:25 )  (  job:2 timeleft:0 tix:--- )
--> JOB 1 DONE at time 12
```

-s2 için;

```
ubuntu@ubuntu:~/Desktop/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j3 -s 2 -c
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 2

Here is the job list, with the run time of each job:
  Job 0 ( length = 9, tickets = 94 )
  Job 1 ( length = 8, tickets = 73 )
  Job 2 ( length = 6, tickets = 30 )


** Solutions **

Random 605944 -> Winning ticket 169 (of 197) -> Run 2
  Jobs:
 (  job:0 timeleft:9 tix:94 )  (  job:1 timeleft:8 tix:73 )  (* job:2 timeleft:6 tix:30 )
Random 606802 -> Winning ticket 42 (of 197) -> Run 0
  Jobs:
 (* job:0 timeleft:9 tix:94 )  (  job:1 timeleft:8 tix:73 )  (  job:2 timeleft:5 tix:30 )
Random 581204 -> Winning ticket 54 (of 197) -> Run 0
  Jobs:
 (* job:0 timeleft:8 tix:94 )  (  job:1 timeleft:8 tix:73 )  (  job:2 timeleft:5 tix:30 )
Random 158383 -> Winning ticket 192 (of 197) -> Run 2
  Jobs:
 (  job:0 timeleft:7 tix:94 )  (  job:1 timeleft:8 tix:73 )  (* job:2 timeleft:5 tix:30 )
Random 430670 -> Winning ticket 28 (of 197) -> Run 0
  Jobs:
 (* job:0 timeleft:7 tix:94 )  (  job:1 timeleft:8 tix:73 )  (  job:2 timeleft:4 tix:30 )
Random 393532 -> Winning ticket 123 (of 197) -> Run 1
  Jobs:
 (  job:0 timeleft:6 tix:94 )  (* job:1 timeleft:8 tix:73 )  (  job:2 timeleft:4 tix:30 )
Random 723012 -> Winning ticket 22 (of 197) -> Run 0
  Jobs:
 (* job:0 timeleft:6 tix:94 )  (  job:1 timeleft:7 tix:73 )  (  job:2 timeleft:4 tix:30 )
Random 994820 -> Winning ticket 167 (of 197) -> Run 2
  Jobs:
 (  job:0 timeleft:5 tix:94 )  (  job:1 timeleft:7 tix:73 )  (* job:2 timeleft:4 tix:30 )
Random 949396 -> Winning ticket 53 (of 197) -> Run 0
  Jobs:
 (* job:0 timeleft:5 tix:94 )  (  job:1 timeleft:7 tix:73 )  (  job:2 timeleft:3 tix:30 )
Random 544177 -> Winning ticket 63 (of 197) -> Run 0
  Jobs:
 (* job:0 timeleft:5 tix:94 )  (  job:1 timeleft:7 tix:73 )  (  job:2 timeleft:3 tix:30 )
Random 544177 -> Winning ticket 63 (of 197) -> Run 0
  Jobs:
 (* job:0 timeleft:4 tix:94 )  (  job:1 timeleft:7 tix:73 )  (  job:2 timeleft:3 tix:30 )
Random 444854 -> Winning ticket 28 (of 197) -> Run 0
  Jobs:
 (* job:0 timeleft:3 tix:94 )  (  job:1 timeleft:7 tix:73 )  (  job:2 timeleft:3 tix:30 )
Random 268241 -> Winning ticket 124 (of 197) -> Run 1
  Jobs:
 (  job:0 timeleft:2 tix:94 )  (* job:1 timeleft:7 tix:73 )  (  job:2 timeleft:3 tix:30 )
Random 35924 -> Winning ticket 70 (of 197) -> Run 0
  Jobs:
 (* job:0 timeleft:2 tix:94 )  (  job:1 timeleft:6 tix:73 )  (  job:2 timeleft:3 tix:30 )
Random 27444 -> Winning ticket 61 (of 197) -> Run 0
  Jobs:
 (* job:0 timeleft:1 tix:94 )  (  job:1 timeleft:6 tix:73 )  (  job:2 timeleft:3 tix:30 )
-> JOB 0 DONE at time 14
Random 464894 -> Winning ticket 55 (of 103) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:6 tix:73 )  (  job:2 timeleft:3 tix:30 )
Random 318465 -> Winning ticket 92 (of 103) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:5 tix:73 )  (* job:2 timeleft:3 tix:30 )
Random 380815 -> Winning ticket 48 (of 103) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:5 tix:73 )  (  job:2 timeleft:2 tix:30 )
Random 891790 -> Winning ticket 16 (of 103) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:4 tix:73 )  (  job:2 timeleft:2 tix:30 )
Random 525753 -> Winning ticket 41 (of 103) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:3 tix:73 )  (  job:2 timeleft:2 tix:30 )
Random 560510 -> Winning ticket 87 (of 103) -> Run 2
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:2 tix:73 )  (* job:2 timeleft:2 tix:30 )
Random 236123 -> Winning ticket 47 (of 103) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:2 tix:73 )  (  job:2 timeleft:1 tix:30 )
Random 23858 -> Winning ticket 65 (of 103) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (  job:1 timeleft:0 tix:--- )  (* job:2 timeleft:1 tix:30 )
-> JOB 2 DONE at time 23
```

-s3 için;

```
ubuntu@ubuntu:~/Desktop/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j3 -s 3 -c
ARG jlist
ARG jobs 3
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 3

Here is the job list, with the run time of each job:
  Job 0 ( length = 2, tickets = 54 )
  Job 1 ( length = 3, tickets = 60 )
  Job 2 ( length = 6, tickets = 6 )


** 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) yalnızca 1 biletle ve diğeri (iş 1) 100 (ör. -l 10:1,10:100).
   Bilet sayısı bu kadar dengesiz olunca ne oluyor? İş 0, iş 1 tamamlanmadan çalışacak mı? Ne sıklıkla? Genel olarak, böyle bir bilet dengesizliği piyango planlamasının davranışına ne yapar?

-j flagı ile job sayısını (2), -l flagı ile uzunluk ve bilet sayısını (ikisi de 10 uzunluğunda ama iş 0, 1 bilet; iş 1, 100 bilet) giriyoruz. -c flagı ile hesaplıyoruz.

```
ubuntu@ubuntu:~/Desktop/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j2 -l10:1,10:100 -c
ARG jlist 10:1,10:100
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 = 10, tickets = 1 )
  Job 1 ( length = 10, tickets = 100 )


** Solutions **

Random 844422 -> Winning ticket 62 (of 101) -> Run 1
  Jobs:
 (  job:0 timeleft:10 tix:1 )  (* job:1 timeleft:10 tix:100 )
Random 757955 -> Winning ticket 51 (of 101) -> Run 1
  Jobs:
 (  job:0 timeleft:10 tix:1 )  (* job:1 timeleft:9 tix:100 )
Random 420572 -> Winning ticket 8 (of 101) -> Run 1
  Jobs:
 (  job:0 timeleft:10 tix:1 )  (* job:1 timeleft:8 tix:100 )
Random 258917 -> Winning ticket 54 (of 101) -> Run 1
  Jobs:
 (  job:0 timeleft:10 tix:1 )  (* job:1 timeleft:7 tix:100 )
Random 511275 -> Winning ticket 13 (of 101) -> Run 1
  Jobs:
 (  job:0 timeleft:10 tix:1 )  (* job:1 timeleft:6 tix:100 )
Random 404934 -> Winning ticket 25 (of 101) -> Run 1
  Jobs:
 (  job:0 timeleft:10 tix:1 )  (* job:1 timeleft:5 tix:100 )
Random 783799 -> Winning ticket 39 (of 101) -> Run 1
  Jobs:
 (  job:0 timeleft:10 tix:1 )  (* job:1 timeleft:4 tix:100 )
Random 303313 -> Winning ticket 10 (of 101) -> Run 1
  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
```

Görüleceği üzere iş 1 tamamlanmadan (bilet sayısı 100 olan) iş 0 çalışmıyor.

3. 100 uzunluğunda iki işle ve 100'lük eşit bilet tahsisiyle çalışırken (-l 100:100,100:100), planlayıcı ne kadar adaletsiz? (Olasılığa dayalı) cevabı belirlemek için bazı farklı rastgele tohumlarla çalıştırın; adaletsizliğin bir işin diğerinden ne kadar önce bittiğine göre belirlenmesine izin verin.

-j flagı ile job sayısını (2), -s flagı ile seed (1, 2 ve 3), -l flagı ile uzunluk ve bilet sayısını (ikisi de eşit uzunluk ve bilete sahip: 100:100) giriyoruz. -c flagı ile hesaplıyoruz. Program sonucu çok uzun olduğu için işlerin bitiş zamanlarının ekran görüntülerini ekliyorum.

-s0 için;

```
ubuntu@ubuntu:~/Desktop/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j2 -l100:100,100:100 -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 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
```

Görüleceği üzere iş 0, iş 1'den 8 zaman önce bitmiş.

-s1 için;

```
ubuntu@ubuntu:~/Desktop/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j2 -s1 -l100:100,100:100 -c
ARG jlist 100:100,100:100
ARG jobs 2
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 1

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 194118 -> Winning ticket 118 (of 200) -> Run 1
  Jobs:
 (   job:0 timeleft:4 tix:100 )  (* job:1 timeleft:1 tix:100 )
-> JOB 1 DONE at time 196
Random 104424 -> Winning ticket 24 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:4 tix:100 )  (   job:1 timeleft:0 tix:--- )
Random 665958 -> Winning ticket 58 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:3 tix:100 )  (   job:1 timeleft:0 tix:--- )
Random 296072 -> Winning ticket 72 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:2 tix:100 )  (   job:1 timeleft:0 tix:--- )
Random 499800 -> Winning ticket 0 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:1 tix:100 )  (   job:1 timeleft:0 tix:--- )
-> JOB 0 DONE at time 200
```

Bu sefer iş 1, iş 0'dan 4 zaman önce bitmiş.

-s2 için;

```
ubuntu@ubuntu:~/Desktop/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j2 -s2 -l100:100,100:100 -c
ARG jlist 100:100,100:100
ARG jobs 2
ARG maxlen 10
ARG maxticket 100
ARG quantum 1
ARG seed 2

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 511501 -> Winning ticket 101 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:10 tix:100 )  (* job:1 timeleft:1 tix:100 )
 ++> JOB 1 DONE at time 190
Random 581076 -> Winning ticket 76 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:10 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 51234 -> Winning ticket 34 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:9 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 418016 -> Winning ticket 16 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:8 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 525065 -> Winning ticket 65 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:7 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 181225 -> Winning ticket 25 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:6 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 93786 -> Winning ticket 86 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:5 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 802656 -> Winning ticket 56 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:4 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 366184 -> Winning ticket 84 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:3 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 519210 -> Winning ticket 10 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:2 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 921451 -> Winning ticket 51 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:1 tix:100 )  (  job:1 timeleft:0 tix:--- )
 ++> JOB 0 DONE at time 200
```

Yine iş 1, iş 0'dan önce bitmiş. 10 zaman önce.

4. Kuantum boyutu (-q) büyüdükçe önceki soruya verdiğiniz yanıt nasıl değişir?

   -j flagı ile job sayısını (2), -q flagı ile kuantumu(10 ve 100 girdim), -l flagı ile uzunluk ve bilet sayısını (ikisi de eşit uzunluk ve bilete sahip: 100:100) giriyoruz. -c flagı ile hesaplıyoruz.

```
ubuntu@ubuntu:~/Desktop/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j2 -q10 -l100:100,100:100 -c
ARG jlist 100:100,100:100
ARG jobs 2
ARG maxlen 10
ARG maxticket 100
ARG quantum 10
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:90 tix:100 )  (* job:1 timeleft:100 tix:100 )
Random 420572 -> Winning ticket 172 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:90 tix:100 )  (* job:1 timeleft:90 tix:100 )
Random 258917 -> Winning ticket 117 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:90 tix:100 )  (* job:1 timeleft:80 tix:100 )
Random 511275 -> Winning ticket 75 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:90 tix:100 )  (  job:1 timeleft:70 tix:100 )
Random 404934 -> Winning ticket 134 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:80 tix:100 )  (* job:1 timeleft:70 tix:100 )
Random 783799 -> Winning ticket 199 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:80 tix:100 )  (* job:1 timeleft:60 tix:100 )
Random 303313 -> Winning ticket 113 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:80 tix:100 )  (* job:1 timeleft:50 tix:100 )
Random 476597 -> Winning ticket 197 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:80 tix:100 )  (* job:1 timeleft:40 tix:100 )
Random 583382 -> Winning ticket 182 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:80 tix:100 )  (* job:1 timeleft:30 tix:100 )
Random 908113 -> Winning ticket 113 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:80 tix:100 )  (* job:1 timeleft:20 tix:100 )
Random 504687 -> Winning ticket 87 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:80 tix:100 )  (  job:1 timeleft:10 tix:100 )
Random 281838 -> Winning ticket 38 (of 200) -> Run 0
  Jobs:
 (* job:0 timeleft:60 tix:100 )  (  job:1 timeleft:10 tix:100 )
Random 618369 -> Winning ticket 169 (of 200) -> Run 1
  Jobs:
 (  job:0 timeleft:50 tix:100 )  (* job:1 timeleft:10 tix:100 )
--> JOB 1 DONE at time 150
Random 250506 -> Winning ticket 6 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:50 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 909747 -> Winning ticket 47 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:40 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 982786 -> Winning ticket 86 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:30 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 810218 -> Winning ticket 18 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:20 tix:100 )  (  job:1 timeleft:0 tix:--- )
Random 902166 -> Winning ticket 66 (of 100) -> Run 0
  Jobs:
 (* job:0 timeleft:10 tix:100 )  (  job:1 timeleft:0 tix:--- )
--> JOB 0 DONE at time 200
```

-q10 için;

-q100 için;

```
ubuntu@ubuntu:~/Desktop/ostep/ostep-homework/cpu-sched-lottery$ python3 lottery.py -j2 -q100 -l100:100,100:100 -c
ARG jlist 100:100,100:100
ARG jobs 2
ARG maxlen 10
ARG maxticket 100
ARG quantum 100
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 )
--> JOB 0 DONE at time 100
Random 757955 -> Winning ticket 55 (of 100) -> Run 1
  Jobs:
 (  job:0 timeleft:0 tix:--- )  (* job:1 timeleft:100 tix:100 )
--> JOB 1 DONE at time 200
```

Görüldüğü üzere kuantum sayısı arttıkça iş'lerin çözümü gittikçe kısalıyor.

5. Can you make a version of the graph that is found in the chapter? What else would be worth exploring? How would the graph look with a stride scheduler?