9

Zamanlama: Orantılı Paylaşım

Bu bölümde, **orantılı paylaşım(proportional-share)** zamanlayıcısı olarak bilinen ve bazen **adil paylaşım(fair-share)** zamanlayıcısı olarak da adlandırılan farklı bir zamanlayıcı türünü inceleyeceğiz. Orantılı paylaşım, basit bir kavrama dayanır: geri dönüş veya yanıt süresi için optimizasyon yapmak yerine, bir zamanlayıcı bunun yerine her işin CPU süresinin belirli bir yüzdesini elde etmesini garanti etmeye çalışabilir. Orantılı hisse planlamasının mükemmel bir erken örneği, Waldspurger ve Weihl [WW94] tarafından yapılan araştırmalarda bulunur ve **piyango planlaması(lottery scheduling)** olarak bilinir; ancak, fikir kesinlikle daha eskidir [KL88]. Temel fikir oldukça basittir: sık sık, hangi sürecin bir sonraki adımda çalıştırılması gerektiğini belirlemek için bir piyango tutun; Daha sık işlemesi gereken süreçlere piyangoyu kazanmak için daha fazla şans verilmelidir. Kolay, değil mi? Şimdi, ayrıntılara gelelim! Ama püf noktamızdan önce değil;

Püf Nokta: CPU ORANSAL OLARAK NASIL PAYLAŞILIR?
CPU'yu orantılı bir şekilde paylaşmak için nasıl bir zamanlayıcı tasarlayabiliriz? Bunu yapmak için temel mekanizmalar nelerdir? Ne kadar etkililer?

9.1 Temel Kavram: Biletler Payınızı Temsil Eder

Piyango planlamasının altında yatan çok temel bir kavramdır: **biletler(tickets)**, bir sürecin (veya kullanıcının veya her neyse) alması gereken bir kaynağın payını temsil etmek için kullanılır. Bir işlemin sahip olduğu destek taleplerinin 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 sadece 25 bileti olduğunu hayal edin. Bu nedenle, istediğimiz şey, A'nın CPU'nun %75'ini ve B'nin kalan %25'ini almasıdır. Piyango planlaması bunu olasılıksal olarak (ancak belirleyici olarak değil) sık sık (örneğin, her zaman dilimi) bir piyango düzenleyerek gerçekleştirir. Bir piyango düzenlemek basittir: zamanlayıcı toplam kaç bilet olduğunu bilmelidir (örneğimizde 100 tane vardır). Zamanlayıcı daha sonra seçer.

İPUCU: RASTGELELİĞİ KULLANIN

Piyango planlamasının en güzel yönlerinden biri, **ran-domluk(ran-domness)** kullanımıdır. Bir karar vermeniz gerektiğinde, böyle rastgele bir yaklaşım kullanmak genellikle bunu yapmanın sağlam ve basit bir yoludur. Rastgele yaklaşımların daha geleneksel kararlara göre en az üç avantajı vardır. İlk olarak, rastgele genellikle daha geleneksel bir algoritmanın işlemede sorun yaşayabileceği garip köşe durumu davranışlarından kaçınır. Örneğin, LRU değiştirme politikasını düşünün (sanal bellekle ilgili gelecekteki bir bölümde daha ayrıntılı olarak incelenmiştir); Genellikle iyi bir değiştirme algoritması olsa da, LRU bazı döngüsel sıralı iş yükleri için en kötü durum performansını elde eder. Öte yandan, rastgelede böyle bir kötü durum yoktur.

İkincisi, rastgele de hafiftir ve alternatif yerlileri izlemek için çok az durum gerektirir. Geleneksel bir adil paylaşım zamanlama algoritmasında, her işlemin ne kadar CPU aldığını izlemek, her işlem çalıştırıldıktan sonra güncellenmesi gereken işlem başına muhasebe gerektirir. Bunu rastgele yapmak, yalnızca işlem başına en az sayıda durumu gerektirir (örneğin, her birinin sahip olduğu bilet sayısı).

Son olarak, rastgele oldukça hızlı olabilir. Rastgele bir sayı oluşturmak hızlı olduğu sürece, karar vermek de öyledir ve bu nedenle hızın gerekli olduğu birçok yerde rastgele kullanılabilir. Tabii ki, ihtiyaç ne kadar hızlı olursa, o kadar rastgele sözde rastgele olma eğilimindedir.

Kazanan bilet, 0'dan 99[1]'e kadar bir sayıdır. A'nın 0'dan 74'e ve B 75'ten 99'a kadar olan biletleri tuttuğunu varsayarsak, kazanan bilet basitçe A veya B'nin çalışıp çalışmadığını sona erdirir. Zamanlayıcı daha sonra bu kazanan işlemin durumunu yükler ve 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 program:

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

Örnekten de görebileceğiniz gibi, piyango planlamasında rastgeleliğin kullanılması, istenen pro-kısmı karşılamada olasılıksal bir doğruluğa yol açar, ancak garanti vermez. Yukarıdaki örneğimizde, B, istenen %25'lik tahsisat yerine 20 zaman diliminden yalnızca 4'ünü (%20) çalıştırabilir. Bununla birlikte, bu iki iş ne kadar uzun süre rekabet ederse, istenen yüzdeleri elde etme olasılıkları o kadar artar.

[1]Bilgisayar Bilimcileri her zaman 0'da saymaya başlar. Bilgisayar dışı tipler için o kadar gariptir ki, ünlü insanlar bunu neden bu şekilde yaptığımız hakkında yazmak zorunda hissetmişlerdir [D82].

İPUCU: PAYLAŞIMLARI TEMSIL ETMEK IÇIN BILETLERI KULLANMAK

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 yeni bir çalışmada, Waldspurger, bir konuk işletim sisteminin bellek payını temsil etmek için biletlerin nasıl kullanılabileceğini göstermektedir [W02]. Bu nedenle, mülkiyetin bir oranını temsil edecek bir mekanizmaya ihtiyacınız varsa, bu kavram sadece ... (bekleyin) ... bilet.

9.1 Bilet Mekanizmaları

Piyango planlaması ayrıca biletleri farklı ve bazen yararlı şekillerde manipule etmek için bir dizi mekanizma sağlar. Bir yol, **bilet para birimi(ticket currency)** kavramı iledir. Para birimi, bir dizi onay işaretine 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 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 para birimi) to A1 ->  50 (küresel para
birimi)
            -> 500 (A's currency) to A2 ->  50 (global currency)
Kullanıcı B ->  10 (B'nin para birimi) to B1 -> 100
(küresel para birimi)
```

Bir diğer kullanışlı mekanizma ise **bilet transferidir(ticket transfer).** Transferlerde, bir işlem biletlerini geçici olarak başka bir işleme devredebilir. Bu özellik, bir istemci işleminin bir sunucuya istemci adına bazı işler yapmasını isteyen bir ileti gönderdiği bir istemci/sunucu ayarında özellikle yararlıdır. İşi hızlandırmak için, istemci biletleri sunucuya geçirebilir ve böylece sunucu istemcinin isteğini işlerken sunucunun performansını en üst düzeye çıkarmaya çalışabilir. Tamamlandığında, sunucu daha sonra biletleri istemciye geri aktarır ve her şey daha önce olduğu gibidir.

Son olarak, bilet enflasyonu bazen yararlı bir teknik olabilir. Enflasyonla birlikte, bir süreç sahip olduğu bilet sayısını geçici olarak artırabilir veya azaltabilir. Tabii ki, birbirine güvenmeyen süreçlerle rekabetçi bir senaryoda, bu çok az anlam ifade ediyor; açgözlü bir süreç kendisine çok sayıda bilet verebilir ve makineyi ele geçirebilir. Aksine, enflasyon bir grup sürecin birbirine güvendiği bir ortamda uygulanabilir; Böyle bir durumda, herhangi bir işlem daha fazla CPU süresine ihtiyaç duyduğunu biliyorsa, diğer işlemlerle iletişim kurmadan, bu ihtiyacı sisteme yansıtmanın bir yolu olarak bilet değerini artırabilir.

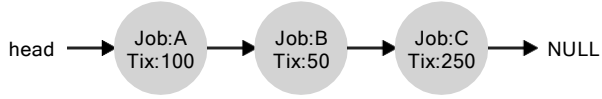
```
// counter: kazananı henüz bulup bulmadığımızı takip etmek
// için kullanılır
int counter = 0;

// kazanan: 0 ile toplam bilet sayısı arasında bir değer elde
//etmek için rastgele bir sayı üreticisine yapılan bazı
//çağrıları kullanın
int winner = getrandom(0, totaltickets);

// current: iş listesinde gezinmek için bunu kullanın
node_t *current = head;
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

Figure 9.1: **Piyango Çizelgeleme Karar Kodu**

9.2 Uygulama

Muhtemelen piyango planlaması ile ilgili en şaşırtıcı şey, uygulanmasının 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 listede tuttuğumuzu varsayalım. İşte her biri bir miktar bilete sahip üç işlemden oluşan bir örnek, A, B ve C.



Bir planlama kararı vermek için, önce toplam bilet sayısından (400)[2] rastgele bir sayı (kazanan) seçmemiz gerekir. Diyelim ki 300 sayısını seçtik. Ardından, kazananı bulmamıza yardımcı olmak için kullanılan basit bir sayaçla listeyi basitçe geçeriz. (Şekil 9.1).

Kod, işlem listesini yürür ve değer kazananı aşana kadar her bilet değerini sayaca ekler. Durum böyle olduğunda, mevcut liste kazanan olur. Kazanan biletin 300 olması örneğimizde, aşağıdakiler gerçekleşir. İlk olarak, A'nın biletlerini hesaba katmak için sayaç 100'e yükseltilir; 100, 300'den az olduğu için, döngü devam eder. Daha sonra sayaç 150'ye (B'nin biletleri), hala 300'den az olacak ve böylece tekrar devam edeceğiz. Son olarak, sayaç 400'e (açıkça 300'den büyük) güncellenir ve böylece C'de (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.

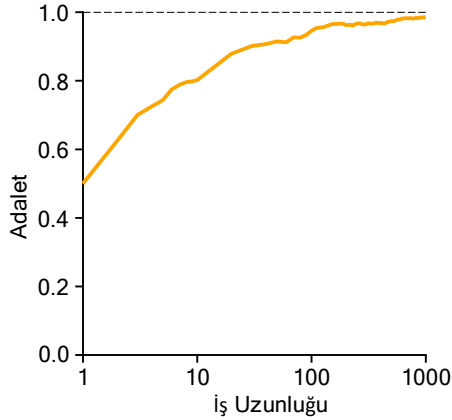


Figure 9.2: **Piyango Adalet Çalışması**

Bu süreci en verimli hale getirmek için, listeyi en yüksek bilet sayısından en düşüğe doğru sıralanmış sırayla veya ganize etmek genellikle en iyisi olabilir. Sıralama, algoritmanın doğruluğunu etkilemez; Bununla birlikte, genel olarak, özellikle biletlerin çoğuna sahip olan birkaç işlem varsa, en az sayıda liste işleminin alınmasını sağlar.

9.3 Bir Örnek

Piyango planlamasının dinamiklerini daha anlaşılır hale getirmek için, şimdi her biri aynı sayıda bilete (100) ve aynı çalışma süresine (R , değiştireceğiz) sahip olan birbirine karşı rekabet eden iki işin tamamlanma süresi hakkında kısa bir çalışma yapıyoruz .

Bu senaryoda, her işin kabaca aynı anda bitmesini isteriz, ancak piyango zamanlamasının rastgeleliği nedeniyle, bazen bir iş diğerinden önce biter.. 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 zamanında bitiyorsa (ve ikinci iş 20'deyse), $F = \frac{10}{20} = 0.5$. Her iki iş de neredeyse aynı anda bittiğinde, F 1'e oldukça yakın olacaktır. Bu senaryoda, hedefimiz budur: mükemmel adil bir zamanlayıcı F = 1'e ulaşacaktır.

Şekil 9.2, iki işin (R) uzunluğu otuz denemeden 1 ila 1000 arasında değiştiği için ortalama adaleti çizer (sonuçlar, bölümün sonunda sağlanan simülatör aracılığıyla üretilir). 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.

9.4 Biletler nasıl tahsis edilir?

Piyango planlaması ile ele almadığımız bir sorun şudur: biletler nasıl tahsis edilir? Bu sorun zor bir sorundur, çünkü elbette sistemin nasıl davrandığı, biletlerin nasıl tahsis edildiğine bağlıdır. Bir yaklaşım, kullanıcıların en iyisini bildiğini varsaymaktır; böyle bir durumda, her kullanıcıya bir miktar bilet verilir ve bir kullanıcı istediği gibi çalıştırdığı herhangi bir işe bilet tahsis edebilir. Ancak, bu çözüm bir çözüm değildir: size gerçekten ne yapacağınızı söylemez. Böylece, bir dizi iş göz önüne alındığında, "bilet-atama sorunu" açık kalır.

9.5 Adım Çizelgeleme

Ayrıca merak ediyor olabilirsiniz: neden rastgeleliği kullanıyorsunuz? Yukarıda gördüğümüz gibi, rastgelelik bize basit (ve yaklaşık olarak doğru) bir zamanlayıcı kazandırırken, bazen özellikle kısa zaman ölçeklerinde tam olarak doğru oranları sağlamayacaktır. Bu nedenle Waldspurger, deterministik bir adil paylaşım zamanlayıcısı olan **adım** zamanlamasını**(stride scheduling)** icat etti [W95].

Adım zamanlaması da basittir. Sistemdeki her işin, sahip olduğu bilet sayısıyla orantılı olarak ters yönde bir adımı vardır. Yukarıdaki örneğimizde, sırasıyla 100, 50 ve 250 biletle A, B ve C işlerinde, her bir işlemin atandığı bilet sayısına çok sayıda bölerek her birinin adımını hesaplayabiliriz. Örneğin, 10.000'i bu bilet değerlerinin her birine bölersek, A, B ve C için aşağıdaki adım değerlerini elde ederiz: 100, 200 ve 40. Bu değere her sürecin **adımı(stride)** diyoruz; Bir süreç her çalıştığında, küresel ilerlemesini izlemek için attığı adımla onun için bir sayacı (**geçiş(pass)** değeri olarak adlandırılır) artıracağız. Zamanlayıcı daha sonra hangi işlemin daha sonra çalışması gerektiğini belirlemek için adım ve geçişi 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 sağlanmaktadır:

```
curr = remove_min(queue);   // minimum geçişli müşteri seçin
schedule(curr);             // kuantum için çalıştırın
curr->pass += curr->stride; // adım kullanarak güncelleme
geçişi
insert(queue, curr);        // curr'ı kuyruğa döndür
```

Örneğimizde, adım değerleri 100, 200 ve 40 olan ve tümü başlangıçta 0 olan geçiş değerlerine sahip üç işlemle (A, B ve C) başlıyoruz. Bu nedenle, ilk başta, geçiş değerleri eşit derecede düşük olduğu için işlemlerden herhangi biri çalışabilir. A'yı seçtiğimizi varsayalım (keyfi olarak; eşit düşük geçiş değerlerine sahip işlemlerden herhangi biri seçilebilir). A koşar; zaman dilimi bittiğinde, geçiş değerini 100 olarak güncelleriz. Ardından, geçiş değeri daha sonra 200 olarak ayarlanan B'yi çalıştırırız. Son olarak, geçiş değeri 40'a yükseltilen C'yi çalıştırırız. Bu noktada, algoritma C'ler olan en düşük geçiş değerini seçecek ve

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Figure 9.3: **Adım Zamanlaması: Bir İz**

çalıştırın, geçişini 80'e güncelleyin (hatırladığınız gibi C'nin adımı 40'tır). Sonra C tekrar koşacak (hala en düşük pas değeri) ve geçişini 120'ye yükseltecek. A şimdi çalışacak ve geçişini 200'e güncelleyecek (şimdi B'ye eşit). Daha sonra C iki kez daha çalışacak ve geçişini 160'a ve ardından 200'e güncelleyecek. Bu noktada, tüm geçiş değerleri tekrar eşittir ve işlem sonsuza dek tekrarlanır. Şekil 9.3, zamanlayıcının zaman içindeki davranışını 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 koştu. Piyango planlaması, zaman içinde olasılıksal olarak oranlara ulaşır; adım zamanlaması, her zamanlama döngüsünün sonunda onları tam olarak doğru şekilde sağlar.

Bu yüzden merak ediyor olabilirsiniz: adım planlamasının hassasiyeti göz önüne alındığında, neden piyango planlamasını kullanıyorsunuz? Piyango planlamasının, adım zamanlamasının yapmadığı güzel bir özelliği vardır: küresel devlet yok. Yukarıdaki adım planlama örneğimizin ortasına yeni bir işin girdiğini hayal edin; geçiş değeri ne olmalıdır? 0 olarak ayarlanmalı mı? Eğer öyleyse, CPU'yu tekelleştirecektir. Piyango planlaması ile, süreç başına küresel bir devlet yoktur; sadece sahip olduğu biletlerle yeni bir süreç ekliyoruz, toplam kaç biletimiz olduğunu izlemek için tek bir global değişkeni güncelliyoruz ve oradan gidiyoruz. Bu şekilde, piyango, yeni süreçleri mantıklı bir şekilde dahil etmeyi çok daha kolay hale getirir.

9.6 Linux Tamamen Adil Tamamlayıcısı(CFS)

Adil paylaşım zamanlamasındaki bu önceki çalışmalara rağmen, mevcut Linux yaklaşımı alternatif bir şekilde benzer hedeflere ulaşmaktadır. Tamamen **Adil Zamanlayıcı(Completely Fair Scheduler)** (veya **CFS**) [J09] başlıklı zamanlayıcı, adil paylaşım zamanlamasını uygular, ancak bunu oldukça verimli ve ölçeklenebilir bir şekilde yapar.

CFS, verimlilik hedeflerine ulaşmak için, hem doğal tasarımı hem de göreve çok uygun veri yapılarını akıllıca kullanması yoluyla zamanlama kararları oluşturmak için çok az zaman harcamayı amaçlamaktadır. Son zamanlarda yapılan çalışmalar, zamanlayıcı verimliliğinin şaşırtıcı derecede önemli olduğunu göstermiştir; Özellikle, Google veri merkezleri üzerinde yapılan bir çalışmada, Kanev ve ark. agresif optimizasyondan sonra bile, zamanlamanın genel veri merkezi CPU süresinin yaklaşık% 5'ini kullandığını göstermektedir. Bu ek yükü mümkün olduğunca yeniden azaltmak, modern zamanlayıcı mimarisinde önemli bir hedeftir.

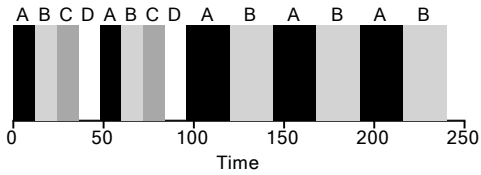


Figure 9.4: **CFS Basit Örnek**

Temel Çalışma

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(vruntime)(virtual runtime)** olarak bilinen basit bir sayma tabanlı teknikle yapar.

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

Bu bir soruyu gündeme getiriyor: Zamanlayıcı şu anda çalışan işlemi ne zaman durduracağını ve bir sonrakini ne zaman çalıştıracağını nasıl biliyor? Buradaki gerilim açıktır: CFS çok sık geçiş yaparsa, CFS her işlemin CPU'dan payını almasını sağlayacağından, adalet artar, ancak performans pahasına (çok fazla bağlam değiştirme); CFS daha az sıklıkta geçiş yaparsa, performans artar (bağlam değiştirmede azalma), ancak kısa vadeli adalet pahasına.

CFS bu gerilimi çeşitli kontrol parametreleri ile yönetir. Birincisi, **sched gecikme süresidir(sched_latency)**. CFS, bir anahtarı düşünmeden önce bir işlemin ne kadar süre çalışması gerektiğini belirlemek için bu değeri kullanır (zaman dilimini etkili bir şekilde ancak dinamik bir şekilde belirler). Tipik bir sched gecikme süresi değeri 48'dir (milisaniye); CFS, bir işlemin zaman dilimini belirlemek için bu değeri CPU üzerinde çalışan işlemlerin sayısına (n) böler ve böylece bu süre zarfında CFS'nin tamamen adil olmasını sağlar.

Örneğin, çalışan n = 4 işlem varsa, CFS, 12 ms'lik bir işlem başına zaman dilimine ulaşmak için sched gecikme süresinin değerini n'ye böler. CFS, ardından ilk işi zamanlar ve 12 ms (sanal) çalışma zamanı kullanana kadar çalıştırır ve ardından bunun yerine çalıştırılacak daha düşük vruntime değerine sahip bir iş olup olmadığını denetler. Bu durumda, vardır ve CFS diğer üç işten birine geçer ve bu şekilde devam eder. Şekil 9.4, dört işin (A, B, C, D) her birinin bu şekilde iki zaman dilimi için çalıştığı bir örneği göstermektedir; Bunlardan ikisi (C, D) daha sonra tamamlanır, sadece iki tanesi kalır, daha sonra her biri yuvarlak robin tarzında 24 ms boyunca koşar.

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

Bu sorunu gidermek için CFS, genellikle 6 ms gibi bir değere ayarlanan **minimum ayrıntı düzeyi(min granularity)** adlı başka bir parametre ekler.

Bu değerden daha düşük bir işlemin, genel giderlerin programlanmasında çok fazla zaman harcanmamasını sağlamak.

Örneğin, çalışan on işlem varsa, özgün hesaplamamız zaman dilimini belirlemek için sched gecikme süresini ona böler (sonuç: 4,8 ms). Bununla birlikte, minimum ayrıntı düzeyi nedeniyle, CFS her işlemin zaman dilimini bunun yerine 6 ms'ye ayarlayacaktır. CFS, 48 ms'lik hedef zamanlama gecikmesi (sched gecikmesi) üzerinde (tam olarak) mükemmel bir şekilde adil olmayacak olsa da, yine de yüksek CPU verimliliği elde ederken yakın olacaktır.

CFS'nin periyodik bir zamanlayıcı kesintisi kullandığını unutmayın, bu da yalnızca sabit zaman aralıklarında karar verebileceği anlamına gelir. Bu kesinti sık sık (örneğin, her 1 ms'de bir) söner ve CFS'ye uyanma ve mevcut işin çalışmasının sonuna ulaşıp ulaşmadığını belirleme şansı verir. Bir iş, zamanlayıcı kesme aralığının mükemmel bir katı olmayan bir zaman dilimine sahipse, sorun değil; CFS, vruntime'u tam olarak izler, bu da uzun mesafe boyunca 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 bir pay vermelerini sağlar. Bunu biletlerle değil, bir sürecin **güzel(nice)** seviyesi olarak bilinen klasik bir UNIX mekanizması aracılığıyla yapar. nice parametresi, varsayılan değer 0 olan bir işlem için -20 ila +19 arasında herhangi bir yere ayarlanabilir. Pozitif güzel(nice) değerler daha düşük öncelikli, negatif değerler ise daha yüksek önceliğe işaret eder; Çok iyi(nice) olduğunuzda, ne yazık ki, çok fazla (zamanlama) 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, her bir işlemin etkili zaman dilimini hesaplamamıza izin verir (daha önce yaptığımız gibi), ancak şimdi öncelik farklılıklarını hesaba katar. Bunu yapmak için kullanılan formül, n işlemi varsayarak 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. İki iş olduğunu varsayalım, A ve B. A, en değerli işimiz olduğu için, -5 gibi güzel bir değer atanarak daha yüksek bir öncelik verilir;

B, ondan nefret ettiğimiz için[3], yalnızca varsayılan önceliğe sahiptir (0'a eşit güzel değer).
Bu, A ağırlığının (tablodan) 3121 olduğu, B ağırlığının ise 1024 olduğu anlamına gelir. Daha sonra her işin zaman dilimini hesaplarsanız, A'nın zaman diliminin sched gecikmesinin yaklaşık 3/4'ü olduğunu göreceksiniz.(bu nedenle, 36 ms) ve B'nin yaklaşık 1/4'ü (dolayısıyla 12 ms).

Zaman dilimi hesaplamasını genelleştirmenin yanı sıra, CFS'nin vruntime'u hesaplama şekli de uyarlanmalıdır. İşte i'nin tahakkuk ettirdiği gerçek çalışma süresini (runtimei) alan ve varsayılan 1024 (ağırlık 0) ağırlığını ağırlığına, weighti'ye bölerek işlemin ağırlığına göre ters ölçeklendiren yeni formül. Çalışan örneğimizde, A'nın vruntime'u B'nin üçte biri oranında birikecektir.

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

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

Kırmızı-Siyah Ağaçları Kullanma

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

CFS, süreçleri **kırmızı-siyah bir ağaçta(red-blacktree)** tutarak bunu ele alır [B72]. Kırmızı-siyah bir ağaç, birçok dengeli ağaç türünden biridir; basit bir ikili ağacın aksine (en kötü ekleme desenleri altında liste benzeri performansa dejenere olabilir), dengeli ağaçlar düşük derinlikleri korumak için biraz ekstra iş yapar ve böylece işlemlerin zaman içinde logaritmik (doğrusal değil) olmasını sağlar.
CFS tüm süreci bu yapıda tutmaz; daha ziyade, sadece koşmak (veya çalıştırılabilir) süreçler burada tutulur. Bir işlem uyku moduna geçerse (örneğin, bir G/Ç'nin tamamlanmasını veya bir ağ paketinin gelmesini beklerse), ağaçtan kaldırılır ve başka bir yerin izi tutulur.

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, çalıştırılacak bir sonraki işi bulmak basit olurdu: sadece ilk öğeyi kaldırın. Ancak,

[3] Evet, evet, burada bilerek kötü dilbilgisi kullanıyoruz, lütfen bir hata düzeltmesi göndermeyin. Neden? Yüzüklerin Efendisi'ne ve en sevdiğimiz anti-kahraman Gollum'a yapılan göndermelerin sadece en hafifi, çok heyecanlanacak bir şey yok.

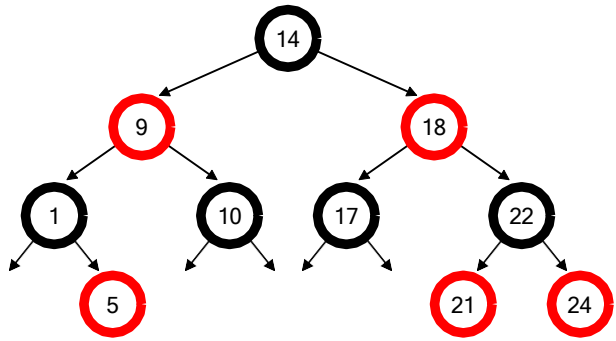


Figure 9.5: **CFS Kırmızı-Siyah Ağaç**

Bu işi listeye geri yerleştirirken (sırayla), listeyi taramamız, eklemek için doğru noktayı, bir O(n) işlemini aramamız gerekir. Herhangi bir arama da oldukça verimsizdir, ayrıca ortalama olarak doğrusal zaman alır.

Aynı değerleri kırmızı-siyah bir ağaçta tutmak, Şekil 9.5'te gösterildiği gibi çoğu işlemi daha verimli hale getirir. İşlemler ağaçta vruntime tarafından sıralanır ve çoğu işlem (ekleme ve silme gibi) zaman içinde logaritmiktir, yani O(log n). n binlerde olduğunda, logaritmik doğrusal olandan belirgin şekilde daha verimlidir.

G/Ç ve Uyku Süreçleriyle Başa Çıkma

Bir sonraki çalıştırmak için en düşük çalışma zamanını seçmeyle ilgili bir sorun, uzun süre uykuya dalmış işlerde ortaya çıkar. İki süreç hayal edin, biri (A) sürekli çalışan ve diğeri (B) uzun bir süre (örneğin, 10 saniye) uykuya dalmış olan A ve B. B uyandığında, çalışma süresi A'nın 10 saniye gerisinde olacak ve böylece (dikkatli olmazsak), B artık CPU'yu yakalarken sonraki 10 saniye boyunca tekelleştirecek ve A'yı etkili bir şekilde aç bırakacaktır.

CFS, uyandığında bir işin çalışma zamanını değiştirerek bu durumu ele alır. Özellikle, CFS bu işin çalışma zamanını ağaçta bulunan en düşük değere ayarlar (unutmayın, ağaç yalnızca çalışan işleri içerir) [B+18]. Bu şekilde, CFS açlıktan kaçınır, ancak bir maliyet olmadan değil: kısa süre uyuyan işler genellikle CPU'dan adil paylarını alamazlar [AC97].

Diğer CFS Eğlencesi

CFS'nin kitabın bu noktasında tartışılamayacak kadar çok başka özelliği var. Önbellek performansını artırmak için çok sayıda sezgisel yöntem içerir, birden fazla CPU'yu etkili bir şekilde işlemek için stratejilere sahiptir (kitapta daha sonra tartışıldığı gibi), büyük işlem grupları arasında zamanlama yapabilir (her işlemi bağımsız bir varlık olarak ele almak yerine)

İPUCU: UYGUN OLDUĞUNDA VERIMLI VERI YAPILARI KULLANIN

Çoğu durumda, bir liste yapar. Çoğu durumda, olmaz. Hangi veri yapısının ne zaman kullanılacağını bilmek, iyi mühendisliğin ayırt edici bir özelliğidir. Burada tartışılan durumda, önceki zamanlayıcılarda bulunan basit listeler, özellikle veri merkezlerinde bulunan ağır yüklü sunucularda, modern sistemlerde iyi çalışmaz. Bu tür sistemler binlerce aktif süreç içerir; Her birkaç milisaniyede bir her çekirdekte 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 bir ağacın mükemmel bir şekilde tamamlanmasını sağlayarak bir tane sağladı. Daha genel olarak, inşa ettiğiniz bir sistem için bir veri yapısı seçerken, erişim adımlarını ve kullanım sıklığını dikkatlice göz önünde bulundurun; Bunları anlayarak, eldeki görev için doğru yapıyı uygulayabileceksiniz..

ve diğer birçok ilginç özellik. Daha fazla bilgi edinmek için Bouron [B+18] ile başlayan son araştırmaları okuyun.

9.7 Özet

Orantılı hisse zamanlaması kavramını tanıttık ve üç yaklaşımı kısaca tartıştık: piyango zamanlaması, adım zamanlaması ve Linux'un Tamamen Adil Zamanlayıcısı (CFS). Piyango, orantılı pay elde etmek için rastgeleliği akıllıca kullanır; stride bunu deterministik olarak yapar. Bu bölümde tartışılan tek "gerçek" zamanlayıcı olan CFS, dinamik zaman dilimlerine sahip ağırlıklı hepsini bir kez deneme gibidir, ancak yük altında iyi ölçeklendirmek ve iyi performans göstermek için üretilmiştir; Bildiğimiz kadarıyla, bugün var olan en yaygın kullanılan adil paylaşım zamanlayıcısıdır.

Hiçbir zamanlayıcı her derde deva değildir ve adil paylaşımlı zamanlayıcılar sorunlardan adil paylarına sahiptir. Bir sorun, bu tür yaklaşımların özellikle G/Ç [AC97] ile iyi örtüşmemesidir; Yukarıda belirtildiği gibi, zaman zaman G/Ç gerçekleştiren işler CPU'dan adil paylarını alamayabilir. Diğer bir sorun ise, bilet veya öncelik atama gibi zor bir sorunu açık bırakmalarıdır, yani tarayıcınızın kaç bilet tahsis edilmesi gerektiğini veya metin editörünüzü ayarlamak için hangi güzel değere sahip olduğunu nasıl bilebilirsiniz? Diğer genel amaçlı zamanlayıcılar (daha önce tartıştığımız MLFQ ve diğer benzer Linux zamanlayıcıları gibi) bu sorunları otomatik olarak ele alır ve böylece daha kolay dağıtılabilir.

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

References

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

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

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

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

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

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

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

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

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

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

Ödev (Simülasyon)

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

Sorular

1) 3 iş ve rastgele 1,2 ve 3 tohumları ile simülasyon çözümlerini hesaplayınız.

1.işler şu sırayla çalışır: 2, 0, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1 **(Piyango Zamanlaması(Lottery Scheduling)**)

2.işler şu sırayla çalışır: 2, 0, 0, 2, 0, 1, 0, 2, 0, 0, 0, 1, 0, 0, 1, 2, 1, 1, 1, 2, 1, 1, 2 **(Adım Zamanlaması(Stride Scheduling))**

3.işler şu sırayla çalışır: 1, 1, 0, 1, 0, 2, 2, 2, 2, 2, 2, 2 **(Linux'un Tamamen Adil Zamanlayıcısı(CFS)(Completely Fair Scheduler))**

2) Şimdi iki özel işle çalışın: her biri 10 uzunluğunda, ancak biri (iş 0) sadece 1 biletle ve diğeri (iş 1) 100 ile (örneğin, -l 10: 1,10:100). Bilet sayısı bu kadar dengesiz olduğunda ne olur? İş 0, iş 1 tamamlanmadan önce çalışacak mı? Ne sıklıkla? Genel olarak, böyle bir bilet dengesizliği piyango planlama davranışına ne yapar?
iş 0'ın 1/101 şansı vardır (< %1), bu nedenle ikincisinin(iş 1) birincisinden(iş 0) önce tamamlanması garanti edilir. Yani iş 1, çalıştırmak için daha fazla fırsata sahip olacak. Düşük bilet işleri için adil değil.

3) 100 uzunluğundaki iki işle ve 100 eşit bilet tahsisatıyla (-l 100:100,100:100) çalışırken, zamanlayıcı ne kadar haksızdır? (Olasılıksal) cevabı belirlemek için bazı farklı rastgele tohumlarla çalıştırın; adaletsizlik, bir işin diğerinden ne kadar erken bittiğiyle belirlensin.

Oldukça adildi. İş ne kadar uzun olursa, zaman içinde adalet olasılığı o kadar yüksek olur.

$ ./lottery.py -s 0 -l 100:100,100:100 -c

U = 192/200 = 0.96

$ ./lottery.py -s 1 -l 100:100,100:100 -c

U = 192/200 = 0.96

$ ./lottery.py -s 2 -l 100:100,100:100 -c

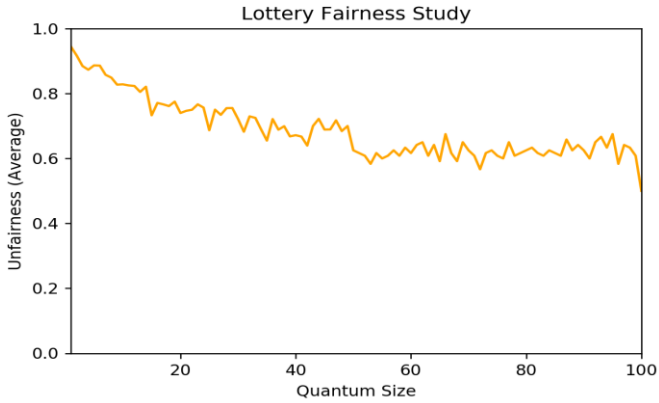
U = 190/200 = 0.95

$ ./lottery.py -s 3 -l 100:100,100:100 -c

U = 196/200 = 0.98

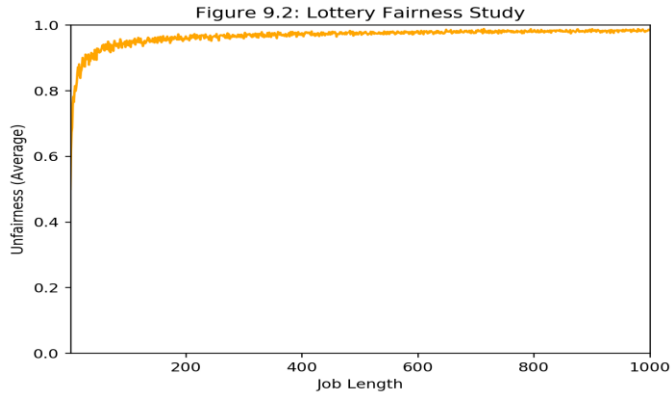
4) Önceki soruya verdiğiniz cevap, kuantum boyutu (-q) büyüdükçe nasıl değişiyor?

Bu, iş uzunluğunu azaltmakla aynı etkiye sahiptir, yani daha az güvenilir bir şekilde adildir (quantum size-adaletsizlik grafiği).



5) Bölümde bulunan grafiğin bir versiyonunu yapabilir misiniz? Başka ne keşfetmeye değer olurdu? Bir adım planlayıcıyla(stride scheduling) grafik nasıl görünür?

Farklı bir versiyonu (bölümde iş uzunluğu-adalet grafiği vardı. Burada iş uzunluğu-adaletsizlik grafiği);



Adım planlayıcı(stride schudeling) grafiği (iş uzunluğu-adaletsizlik);