10

Çok İşlemcili Zamanlama (Gelişmiş) Multiprocessor Scheduling (Advanced)

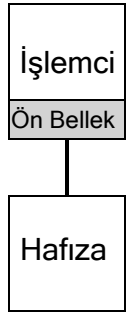
Bu bölüm, **çok işlemcili programlamanın**(**multiprocessor scheduling)** temellerini tanıtacaktır.. Bu konu nispeten ileri düzeyde olduğundan, eşzamanlılık konusunu biraz ayrıntılı olarak inceledikten *sonra* ele almak en iyisi olabilir. (yani, kitabın ikinci büyük "kolay parçası").

Bilgi işlem yelpazesinin yalnızca en üst noktasında yıllarca var olduktan sonra**, çok işlemcili (multiprocessor)** sistemler giderek daha yaygın hale geldi ve masaüstü makinelere, dizüstü bilgisayarlara ve hatta mobil cihazlara girdiler. Bu çipler, bilgisayar mimarlarının çok fazla güç (çok) kullanmadan tek bir CPU'yu çok daha hızlı yapmakta zorlandıkları için popüler hale geldi; birden çok CPU çekirdeğinin tek bir çipte toplandığı **çok çekirdekli(multicore)** işlemcinin yükselişi, bu yaygınlaşmanın kaynağıdır. Ve böylece hepimizin kullanabileceği birkaç CPU var, bu iyi bir şey, değil mi?

Tabii ki, birden fazla CPU'nun gelmesiyle ortaya çıkan birçok zorluk var. Birincisi, tipik bir uygulamanın (yani, yazdığınız bazı C programlarının) yalnızca tek bir CPU kullanmasıdır; daha fazla CPU eklemek, tek bir uygulamanın daha hızlı çalışmasını sağlamaz. Bu sorunu çözmek için, uygulamanızı **parallel** (**paralel**) çalışacak şekilde, belki de **iş parçacıklarını (threads)** kullanarak yeniden yazmanız gerekecek (bu kitabın ikinci bölümünde ayrıntılı olarak ele alındığı gibi.). Çok iş parçacıklı uygulamalar, işi birden çok CPU'ya yayabilir ve böylece daha fazla CPU kaynağı verildiğinde daha hızlı çalışabilir.

Öte Yandan: Gelişmiş Bölümler

Gelişmiş bölümler, kitabın geniş bir alanından materyalin gerçekten anlaşılmasını gerektirirken, mantıksal olarak söz konusu önkoşul materyallerinden daha önceki bir bölüme sığar. Örneğin, çok işlemcili zamanlama hakkındaki bu bölüm, ilk olarak eşzamanlılık hakkındaki ortadaki parçayı okuduysanız çok daha anlamlı olur; bununla birlikte, mantıksal olarak kitabın sanallaştırma (genel olarak) ve CPU zamanlaması (özel olarak) ile ilgili bölümüne uyar. Bu nedenle, bu tür bölümlerin sıra dışı ele alınması tavsiye edilir; bu durumda, kitabın ikinci bölümünden sonra.



Şekil 10.1: **Önbellekli Tek İşlemci (Single CPU With Cache)**

Uygulamaların ötesinde, işletim sistemi için ortaya çıkan yeni bir sorun da (şaşırttı mı, tabii ki hayır!) **çok işlemcili planlamadır (multiprocessor scheduling)**. Şimdiye kadar tek işlemcili planlamanın ardındaki bir dizi ilkeyi tartıştık; bu fikirleri birden fazla CPU üzerinde çalışacak şekilde nasıl genişletebiliriz? Hangi yeni sorunların üstesinden gelmeliyiz? İşte bizim sorunumuz:

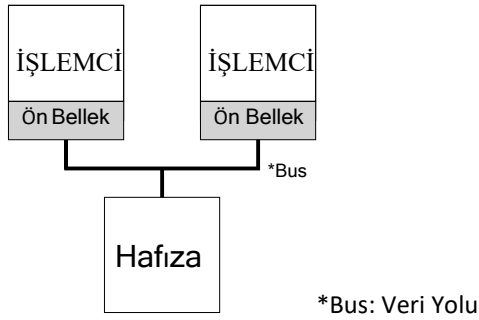
PÜF NOKTA: ÇOKLU İŞLEMCİLERDE İŞLER NASIL PLANLANIR

İşletim sistemi, birden fazla CPU'daki işleri nasıl planlamalıdır? Hangi yeni sorunlar ortaya çıkıyor? Aynı eski teknikler işe yarıyor mu, yoksa yeni fikirler mi gerekli?

10.1 Arka Plan: Çok İşlemcili Mimari (Multiprocessor Architecture)

Çok işlemcili planlamayı çevreleyen yeni sorunları anlamak için, tek CPU donanımı ile çoklu CPU donanımı arasındaki yeni ve temel farkı anlamamız gerekir. Bu fark, donanım önbelleklerinin kullanımı (ör. Şekil 10.1) ve verilerin birden fazla işlemci arasında tam olarak nasıl paylaşıldığı etrafında toplanır. Şimdi bu konuyu daha üst düzeyde tartışacağız. Ayrıntılar, başka bir yerde [CSG99], özellikle üst düzey veya belki de yüksek lisans bilgisayar mimarisi kursunda mevcuttur.

Tek CPU'lu bir sistemde, genellikle işlemcinin programları daha hızlı çalıştırmasına yardımcı olan bir **donanım önbellek(hardware caches)** hiyerarşisi vardır. Önbellekler, (genel olarak) sistemin ana belleğinde bulunan popüler verilerin kopyalarını tutan küçük, hızlı belleklerdir. Buna karşın ana bellek, tüm verileri tutar, ancak bu daha büyük belleğe erişim daha yavaştır. Sistem, sık erişilen verileri bir önbellekte tutarak, büyük, yavaş belleğin hızlıymış gibi görünmesini sağlayabilir.



Şekil 10.2: **Belleği Paylaşan Önbelleğe Sahip İki CPU**
(Two CPUs With Caches Sharing Memory)

Örnek olarak, bellekten bir değer getirmek için açık bir yükleme talimatı veren bir programı ve yalnızca tek bir CPU'ya sahip basit bir sistemi düşünün; CPU'nun küçük bir önbelleği (örneğin 64 KB) ve büyük bir ana belleği vardır.Bir program bu yükü ilk kez verdiğinde, veriler ana bellekte bulunur ve bu nedenle getirilmesi uzun zaman alır (belki onlarca hatta yüzlerce nanosaniye). İşlemci, verilerin yeniden kullanılabileceğini tahmin ederek, yüklenen verilerin bir kopyasını işlemci önbelleğine koyar. Program daha sonra bu aynı veri öğesini tekrar getirirse, CPU ilk olarak önbellekte bunu kontrol eder.; eğer onu orada bulursa, veriler çok daha hızlı getirilir (örneğin sadece birkaç nanosaniye) ve böylece program daha hızlı çalışır.

Bu nedenle önbellekler, iki türü olan **yerellik(locality)** kavramına dayanır.: **zamansal yerellik(temporal locality)** ve **uzamsal yerellik(spatial locality**). Zamansal yerelliğin ardındaki fikir, bir veri parçasına erişildiğinde, yakın gelecekte ona tekrar erişilmesinin muhtemel olmasıdır; değişkenlere ve hatta komutlara bir döngüde tekrar tekrar erişildiğini hayal edin. Uzamsal yerelliğin ardındaki fikir, eğer bir program x adresindeki bir veri öğesine erişirse, x yakınındaki veri öğelerine de erişmesi muhtemeldir; burada, bir dizi boyunca akan bir programı veya birbiri ardına yürütülen talimatları düşünün. Birçok programda bu türlerin yerelliği bulunduğundan, donanım sistemleri önbelleğe hangi verilerin konulacağı konusunda iyi tahminler yapabilir ve bu nedenle iyi çalışır.

Şimdi zor kısma geçelim: Şekil 10.2'de gördüğümüz gibi, paylaşılan tek bir ana belleğe sahip sistemde birden fazla işlemciniz olduğunda ne olur? Görünen o ki, birden fazla CPU ile önbelleğe alma işlemi çok daha karmaşık. Örneğin, CPU 1'de çalışan bir programın A adresindeki bir veri öğesini (D değerine sahip) okuduğunu düşünün; veriler CPU 1'deki önbellekte olmadığı için sistem onu ana bellekten alır ve D değerini alır.

Program daha sonra A adresindeki değeri değiştirir, sadece önbelleğini yeni D değeriyle günceller; verileri baştan sona ana belleğe yazmak yavaştır, bu nedenle sistem (genellikle) bunu daha sonra yapar. Ardından işletim sisteminin programı çalıştırmayı durdurmaya ve onu CPU 2'ye taşımaya karar verdiğini varsayalım. Program A adresindeki değeri tekrar okur; CPU 2'nin önbelleğinde böyle bir veri yoktur ve bu nedenle sistem değeri ana bellekten alır ve doğru değer olan D yerine eski değer olan D'yi alır. Tüh! Bu genel soruna **önbellek tutarlılığı(cache coherence)** sorunu denir ve sorunun çözümüyle ilgili birçok farklı inceliği açıklayan geniş bir araştırma literatürü vardır. [SHW11]. Burada tüm nüansları atlayıp bazı önemli noktalara değineceğiz; daha fazlasını öğrenmek için bir bilgisayar mimarisi dersi al (veya üç).

Temel çözüm, donanım tarafından sağlanır: donanım, bellek erişimlerini izleyerek, temelde "doğru olanın" olmasını ve tek bir paylaşılan bellek görünümünün korunmasını sağlayabilir. Bunu veri yolu tabanlı bir sistemde (yukarıda açıklandığı gibi) yapmanın bir yolu, **veri yolu gözetleme(bus snooping)** [G83] olarak bilinen eski bir tekniği kullanmaktır; her önbellek, kendilerini ana belleğe bağlayan veri yolunu gözlemleyerek bellek güncellemelerine dikkat eder. Bir CPU daha sonra önbelleğinde tuttuğu bir veri öğesi için bir güncelleme gördüğünde, değişikliği fark edecek ve kopyasını **geçersiz kılacak(invalidate)** (yani kendi önbelleğinden kaldıracak) veya **güncelleyecektir(update)** (yani, yeni değeri de önbelleğe koyacaktır). Geri yazma önbellekleri, yukarıda ima edildiği gibi, bunu daha karmaşık hale getirir (çünkü ana belleğe yazma daha sonra görünür değildir), ancak temel şemanın nasıl çalıştığını hayal edebilirsiniz.

10.2 Senkronizasyonu Unutmayın (Don’t Forget Synchronization)

Tutarlılığı sağlamak için önbelleklerin tüm bu işi yaptığı düşünülürse, paylaşılan verilere eriştiklerinde programların (veya işletim sisteminin kendisinin) herhangi bir şey için endişelenmesine gerek var mı? Cevap ne yazık ki evet ve bu kitabın eşzamanlılık konusundaki ikinci bölümünde ayrıntılı olarak belgelenmiştir. Burada ayrıntılara girmeyecek olsak da, burada bazı temel fikirlerin taslağını/incelemesini yapacağız (eşzamanlılığa aşina olduğunuzu varsayarak).

CPU'lar genelinde paylaşılan veri öğelerine veya yapılara erişirken (ve özellikle güncellerken), doğruluğu garanti etmek için büyük olasılıkla karşılıklı dışlama ilkelleri (kilitler gibi) kullanılmalıdır (**kilitsiz(lock-free)** veri yapıları oluşturmak gibi diğer yaklaşımlar karmaşıktır ve yalnızca ara sıra kullanılır; ayrıntılar için eşzamanlılık ile ilgili parçadaki çıkmaza ilişkin bölüme bakın). Örneğin, aynı anda birden çok CPU'da erişilen paylaşılan bir kuyruğa sahip olduğumuzu varsayalım. Kilitler olmadan, kuyruğa aynı anda eleman eklemek veya çıkarmak, altta yatan tutarlılıkprotokolleriyle bile beklendiği gibi çalışmaz; veri yapısını yeni durumuna atomik olarak güncellemek için kilitlere ihtiyaç duyulur.

Bunu daha somut hale getirmek için, Şekil 10.3'te gördüğümüz gibi, paylaşılan bir bağlantılı listeden bir öğeyi kaldırmak için kullanılan bu kod dizisini hayal edin. İki CPU'daki iş parçacıklarının bu rutine aynı anda girdiğini hayal edin.

```
1  typedef struct __Node_t {
2      int              value;
3      struct __Node_t *next;
4  } Node_t;
5
6  int List_Pop() {
7      Node_t *tmp = head;          // remember old head ...
8      int value   = head->value;   // ... and its value
9      head        = head->next;    // advance head to next pointer
10     free(tmp);                   // free old head
11     return value;                // return value at head
12 }
```

Şekil 10.3: **Basit Liste Silme Kodu (Simple List Delete Code)**

İş parçacığı 1, ilk satırı yürütür, tmp değişkeninde depolanan geçerli kafa değerine sahip olacaktır; Eğer İş Parçacığı 2 ilk satırı da yürütürse, kendi özel tmp değişkeninde depolanan aynı kafa değerine sahip olacaktır (yığın üzerinde tmp tahsis edilmiştir ve bu nedenle her iş parçacığının bunun için kendi özel deposu olacaktır). Böylece, her iş parçacığı listenin başından bir öğeyi kaldırmak yerine, her iş parçacığı aynı baş öğeyi çıkarmaya çalışacak ve bu da her türlü soruna yol açacaktır (Örneğin, 10. Satırda ana öğeden iki kez kurtulma girişimi ve potansiyel olarak aynı veri değerini iki kez döndürme gibi).

Elbette çözüm, bu tür rutinleri **kilitleme(locking)** yoluyla düzeltmektir. Bu durumda, basit bir muteks (ör. *pthread_mutex_t m*;) tahsis etmek ve ardından rutinin başına bir *kilit(&m)* ve sonuna bir *kilit açma(&m)* eklemek sorunu çözerek istediğiniz gibi kodun çalışmasını sağlar. Maalesef, göreceğimiz gibi, böyle bir yaklaşım, özellikle performans açısından sorunsuz değildir. Özellikle, CPU sayısı arttıkça, senkronize paylaşımlı bir veri yapısına erişim oldukça yavaş hale gelir.

10.3 Son Bir Sorun: Önbellek Benzeşimi(One Final Issue: Cache Affinity)

Önbellek yakınlığı olarak bilinen çok işlemcili bir önbellek planlayıcının oluşturulmasında son bir sorun ortaya çıkar [TTG95]. Bu kavram basittir: bir işlem, belirli bir CPU üzerinde çalıştırıldığında, CPU'nun önbelleklerinde (ve TLB'lerde) adil bir bit durum oluşturur. İşlemin bir sonraki çalıştırılışında, aynı CPU'da çalıştırmak genellikle avantajlıdır, çünkü durumunun bir kısmı zaten o CPU'daki önbelleklerde mevcutsa daha hızlı çalışacaktır. Bunun yerine, her seferinde farklı bir CPU üzerinde bir işlem çalıştırılırsa, işlemin performansı daha kötü olacaktır, çünkü her çalıştırıldığında durumu yeniden yüklemek zorunda kalacaktır (önbellek sayesinde donanım tutarlılık protokolleri farklı bir CPU üzerinde doğru şekilde çalışacağını unutmayın). Bu nedenle, çok işlemcili bir programlayıcı, zamanlama kararlarını verirken önbellek yakınlığını göz önünde bulundurmalı, belki de mümkünse bir işlemi aynı CPU'da tutmayı tercih etmelidir.

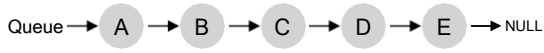
10.4 Tek Kuyruklu Zamanlama(Single-Queue Scheduling)

Bu arka planı hazırladıktan sonra, şimdi çok işlemcili bir sistem için bir programlayıcının nasıl oluşturulacağını tartışacağız. En temel yaklaşım, programlanması gereken tüm işleri tek bir kuyruğa koyarak tek işlemcili zamanlama için temel çerçeveyi basitçe yeniden kullanmaktır; biz buna tek kuyruklu çok işlemcili çizelgeleme veya kısaca SQMS diyoruz. Bu yaklaşımın avantajı basitliktir; bir sonraki çalıştırılacak en iyi işi seçen mevcut bir ilkeyi alıp birden fazla CPU üzerinde çalışacak şekilde uyarlamak için fazla çalışma gerektirmez (burada, örneğin iki CPU varsa çalıştırılacak en iyi iki işi seçebilir) .

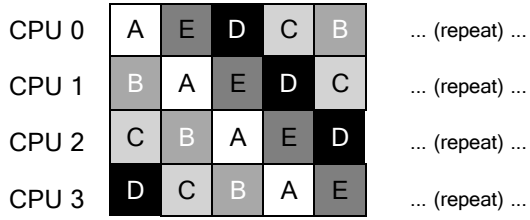
Bununla birlikte, SQMS'nin bariz eksiklikleri vardır. İlk sorun, ölçeklenebilirliğin olmamasıdır. Zamanlayıcının birden fazla CPU üzerinde doğru şekilde çalışmasını sağlamak için, geliştiriciler yukarıda açıklandığı gibi koda bir tür kilitleme eklemiş olacaklardır. Kilitler, SQMS kodu tek kuyruğa eriştiğinde (örneğin çalıştırılacak bir sonraki işi bulmak için) doğru sonucun ortaya çıkmasını sağlar.

Maalesef kilitler, özellikle sistemlerdeki CPU sayısı arttıkça performansı büyük ölçüde azaltabilir [A91]. Böyle tek bir kilit için çekişme arttıkça, sistem kilit ek yüküne giderek daha fazla zaman harcar ve sistemin yapması gereken işi yapmak için daha az zaman harcar (not: bir gün bunun gerçek bir ölçümünü buraya dahil etmek harika olur).

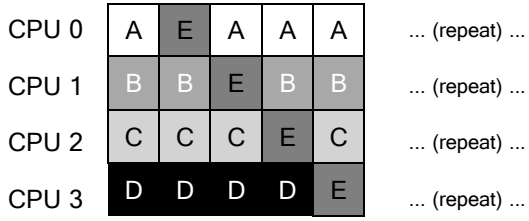
SQMS ile ilgili ikinci ana sorun, önbellek benzeşimidir. Örneğin,
beş işimiz (A, B, C, D, E) ve dört işlemcimiz olduğunu varsayalım. Böylece planlama kuyruğumuz şöyle görünür:



Zaman içinde, her işin bir zaman diliminde çalıştığını ve sonra başka bir işin seçildiğini varsayarsak, burada CPU'lar arasında olası bir iş çizelgesi yer alır:



Her bir CPU, küresel olarak paylaşılan kuyruktan çalıştırılacak bir sonraki işi seçtiği için, her iş CPU'dan CPU'ya sıçrayarak sona erer, böylece önbellek yakınlığı açısından mantıklı olanın tam tersini yapar. Bu sorunun üstesinden gelmek için, çoğu SQMS zamanlayıcısı, işlemin mümkünse aynı CPU üzerinde çalışmaya devam etmesini daha olası hale getirmeye çalışmak için bir tür yakınlık mekanizması içerir. Spesifik olarak, biri bazı işler için yakınlık sağlayabilir, ancak yükü dengelemekiçin diğerlerini hareket ettirebilir. Örneğin, aynı beş işin aşağıdaki şekilde planlandığını hayal edin:



Bu düzenlemede, A'dan D'ye kadar olan işler işlemciler arasında taşınmaz, yalnızca E işi CPU'dan CPU'ya taşınır ve böylece çoğu için benzeşim korunur. Daha sonra, bir dahaki sefere farklı bir işe geçmeye karar verebilirsiniz, böylece bir tür yakınlık adaleti de elde edebilirsiniz. Bununla birlikte, böyle bir planın uygulanması karmaşık olabilir.

Böylece, SQMS yaklaşımının güçlü ve zayıf yönleri olduğunu görebiliriz. Tanımı gereği yalnızca tek bir kuyruğa sahip olan mevcut bir tek CPU planlayıcı verildiğinde uygulanması kolaydır. Ancak, (eşzamanlama ek yükleri nedeniyle) iyi ölçeklenemez ve önbellek yakınlığını hemen korumaz.

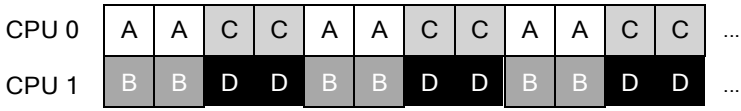
10.5 Çok Kuyruklu Zamanlama(Multi-Queue Scheduling)

Tek kuyruklu programlayıcıların neden olduğu problemler nedeniyle, bazı sistemler, örneğin CPU başına bir tane olmak üzere birden çok sırayı tercih eder. Bu yaklaşıma çok kuyruklu çok işlemcili programlama (veya MQMS) diyoruz.

MQMS'de, temel çizelgeleme çerçevemiz çoklu çizelgeleme kuyruklarından oluşur. Elbette her algoritma kullanılabilmesine rağmen, her bir sıra, büyük olasılıkla, hepsini bir kez deneme gibi belirli bir çizelgeleme disiplinini takip edecektir. Bir iş sisteme girdiğinde, bazı buluşsal yöntemlere göre (örneğin, rastgele veya diğerlerinden daha az iş içeren birini seçmek) tam olarak bir zamanlama kuyruğuna yerleştirilir). Ardından, esasen bağımsız olarak programlanır, böylece tek kuyruklu yaklaşımda bulunan bilgi paylaşımı ve senkronizasyon sorunlarından kaçınılır.

Örneğin, sadece iki CPU'nun (CPU 0 ve CPU 1 olarak etiketlenmiş) olduğu bir sistemimiz olduğunu ve sisteme bazı işlerin girdiğini varsayalım: örneğin A, B, C ve D. Artık her CPU'nun bir zamanlama kuyruğu olduğu göz önüne alındığında, işletim sisteminin her işi hangi kuyruğa yerleştireceğine karar vermesi gerekir. Bunun gibi bir şey yapabilir:

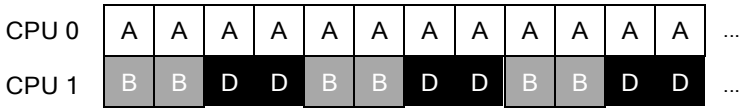
Kuyruk zamanlama ilkesine bağlı olarak, artık her CPU'nun neyin çalıştırılacağına karar verirken aralarından seçim yapabileceği iki işi vardır. Örneğin, hepsini bir kez deneme ile sistem şuna benzeyen bir çizelge oluşturabilir:



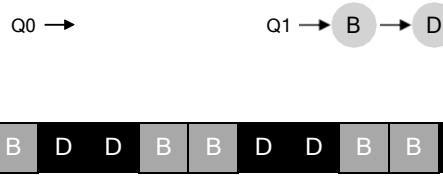
MQMS, doğası gereği daha ölçeklenebilir olması gerektiği için SQMS'nin belirgin bir avantajına sahiptir. CPU sayısı arttıkça sıra sayısı da artar ve bu nedenle kilit ve önbellek çekişmesi merkezi bir sorun haline gelmemelidir. Ek olarak, MQMS özünde önbellek benzerliği sağlar; işler aynı CPU'da kalır ve böylece burada önbelleğe alınmış içerikleri yeniden kullanma avantajından yararlanır.

Ancak, dikkat ettiyseniz, çoklu kuyruğa dayalı yaklaşımda temel olan yeni bir sorunla karşı karşıya olduğumuzu görebilirsiniz: yük dengesizliği. Yukarıdakiyle aynı kuruluma sahip olduğumuzu varsayalım (dört iş, iki CPU), ancak sonra işlerden biri (C diyelim) biter. Artık aşağıdaki zamanlama kuyruklarımız var:

Daha sonra sistemin her bir kuyruğunda hepsini bir kez deneme politikamızı çalıştırırsak, ortaya çıkan bu çizelgeyi göreceğiz:



Bu diyagramdan da görebileceğiniz gibi, A, B ve D'den iki kat daha fazla CPU alıyor ki bu istenen sonuç değil. Daha da kötüsü, hem A hem de C'nin bittiğini ve sistemde yalnızca B ve D işlerini bıraktığını düşünelim. İki zamanlama kuyruğu ve bunun sonucu olan zaman çizelgesi şu şekilde görünecektir:



Ne kadar korkunç – CPU 0 boşta! (https://www.youtube.com/watch?v=7GnbgQd9IQg) Ve bu nedenle CPU kullanım zaman çizelgemiz oldukça üzücü görünüyor.

Öyleyse, zayıf bir çok kuyruklu çok işlemcili zamanlayıcı ne yapmalıdır? Sinsi yük dengesizliği probleminin üstesinden nasıl gelebiliriz ve Decepticon'ların(1) kötü güçlerini nasıl yenebiliriz? Bu harika kitapla pek alakalı olmayan sorular sormayı nasıl durdurabiliriz?

Püf Nokta: YÜK DENGESİZLİĞİYLE NASIL BAŞA ÇIKILIR (HOW TO DEAL WITH LOAD IMBALANCE)
Çok kuyruklu çok işlemcili bir programlayıcı, istenen programlama hedeflerine daha iyi ulaşmak için yük dengesizliğini nasıl ele almalıdır?

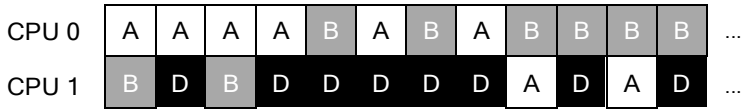
Bu sorunun bariz yanıtı, (bir kez daha) geçiş olarak adlandırdığımız bir teknik olan işleri taşımaktır. Bir işi bir CPU'dan diğerine taşıyarak, gerçek yük dengesi elde edilebilir.

Biraz netlik katmak için birkaç örneğe bakalım. Bir kez daha, bir CPU'nun boşta kaldığı ve diğerinin bazı işlere sahip olduğu bir durumla karşı karşıyayız.

Bu durumda, istenen geçişin anlaşılması kolaydır: İşletim sisteminin B veya D'den birini CPU 0'a taşıması yeterlidir. Bu tek iş geçişinin sonucu, yükün eşit şekilde dengelenmesidir ve herkes mutludur.

Önceki örneğimizde A'nın CPU 0'da tek başına bırakıldığı ve B ve D'nin CPU 1'de sırayla değiştiği daha çetrefilli bir durum ortaya çıkar:

Bu durumda, tek bir geçiş sorunu çözmez. Bu durumda ne yapardınız? Ne yazık ki cevap, bir veya daha fazla işin sürekli olarak taşınmasıdır. Muhtemel bir çözüm, aşağıdaki zaman çizelgesinde gördüğümüz gibi iş değiştirmeye devam etmektir. Şekilde, ilk A, CPU 0'da yalnızdır ve B ve D, CPU 1'de dönüşümlü olarak çalışır. Birkaç zaman diliminden sonra, B, CPU 0'da A ile rekabet etmek üzere hareket ettirilirken, D, CPU 1'de birkaç zaman diliminin keyfini tek başına çıkarır. Böylece yük dengelenmiş olur:



Tabii ki, başka birçok olası göç modeli mevcuttur. Ama şimdi zor olan kısma geçelim: sistem böyle bir geçişe karar vermeye nasıl karar vermeli?

[1] Az bilinen gerçek, Cybertron'un ana gezegeninin kötü CPU planlama kararları tarafından yok edildiğidir. Ve şimdi bu kitaptaki Transformers'a yapılan ilk ve son atıf bu olsun, içtenlikle özür dileriz.

Temel yaklaşımlardan biri, iş çalma [FLR98] olarak bilinen bir tekniği kullanmaktır. İş çalma yaklaşımıyla, işleri az olan bir (kaynak) kuyruğu, ne kadar dolu olduğunu görmek için ara sıra başka bir (hedef) kuyruğa göz atar. Hedef sıra (belirgin bir şekilde) kaynak sıradan daha doluysa, kaynak, yükü dengelemeye yardımcı olmak için hedeften bir veya daha fazla işi "çalar"

Elbette böyle bir yaklaşımda doğal bir gerilim vardır. Diğer kuyruklara çok sık bakarsanız, yüksek ek yükten muzdarip olursunuz ve ölçeklendirmede sorun yaşarsınız, ki bu da çoklu kuyruk zamanlamasını uygulamanın en baştaki amacıydı! Öte yandan, diğer kuyruklara çok sık bakmazsanız, ciddi yük dengesizlikleri yaşama tehlikesiyle karşı karşıya kalırsınız. Doğru eşiği bulmak, sistem politikası tasarımında yaygın olduğu gibi, kara bir sanat olarak kalır.

10.6 Linux Çok İşlemci Planlayıcılar(Linux Multiprocessor Schedulers)

İlginç bir şekilde, Linux topluluğunda, çok işlemcili bir planlayıcı oluşturmak için ortak bir çözüme yaklaşılmamıştır. Zamanla, üç farklı planlayıcı ortaya çıktı: O(1) planlayıcı, Tamamen Uygun Planlayıcı (CFS) ve BF Planlayıcı (BFS)2. Bahsedilen planlayıcıların güçlü ve zayıf yönlerine dair mükemmel bir genel bakış için Meehean'ın tezine bakın. [M11]; burada sadece bazı temel bilgileri özetliyoruz.

Hem O(1) hem de CFS birden çok kuyruk kullanırken, BFS tek bir sıra kullanır, bu da her iki yaklaşımın da başarılı olabileceğini gösterir. Tabii ki, bu programlayıcıları ayıran başka birçok detay var. Örneğin, O(1) programlayıcı, önceliğe dayalı bir programlayıcıdır (daha önce tartışılan MLFQ'ya benzer), bir sürecin önceliğini zaman içinde değiştirir ve ardından çeşitli programlama hedeflerini karşılamak için en yüksek önceliğe sahip olanları planlar. ; etkileşim özel bir odak noktasıdır. Buna karşılık CFS, deterministik bir orantılı paylaşım yaklaşımıdır (daha önce tartışıldığı gibi Stride programlamaya daha çok benzer). Üçü arasındaki tek tek kuyruklu yaklaşım olan BFS de orantılı paylaşımdır, ancak Önce En Erken Uygun Sanal Son Tarih (EEVDF) [SA96] olarak bilinen daha karmaşık bir şemaya dayanır. Bu modern algoritmalar hakkında daha fazlasını kendi başınıza okuyun; şimdi nasıl çalıştıklarını anlayabilmelisiniz!

10.7 Özet(Summary)

Çok işlemcili programlamaya çeşitli yaklaşımlar gördük. Tek kuyruklu yaklaşımın (SQMS) oluşturulması oldukça basittir ve yükü iyi dengeler, ancak doğası gereği birçok işlemciye ve önbellek yakınlığına ölçeklendirmede zorluk yaşar. Çoklu kuyruk yaklaşımı (MQMS) daha iyi ölçeklenir ve önbellek benzerliğini iyi idare eder, ancak yük dengesizliği ile ilgili sorunları vardır ve daha karmaşıktır. Hangi yaklaşımı seçerseniz seçin, basit bir cevap yoktur: küçük kod değişiklikleri büyük davranışsal farklılıklara yol açabileceğinden, genel amaçlı bir planlayıcı oluşturmak göz korkutucu bir görev olmaya devam etmektedir. Sadece tam olarak ne yaptığınızı biliyorsanız veya en azından bunu yapmak için büyük miktarda para alıyorsanız böyle bir egzersizi yapın.

[2] BF'nin kendi başına ne anlama geldiğine bakın; önceden uyarayım, kalbi zayıf olanlar için değildir.

Referanslar (References)

[A90] "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors" by Thomas E. Anderson. IEEE TPDS Volume 1:1, January 1990. *A classic paper on how different locking alternatives do and don't scale. By Tom Anderson, very well known researcher in both systems and networking. And author of a very fine OS textbook, we must say.*

[B+10] "An Analysis of Linux Scalability to Many Cores Abstract" by Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nick- olai Zeldovich. OSDI '10, Vancouver, Canada, October 2010. *A terrific modern paper on the difficulties of scaling Linux to many cores.*

[CSG99] "Parallel Computer Architecture: A Hardware/Software Approach" by David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. Morgan Kaufmann, 1999. *A treasure filled with details about parallel machines and algorithms. As Mark Hill humorously observes on the jacket, the book contains more information than most research papers.*

[FLR98] "The Implementation of the Cilk-5 Multithreaded Language" by Matteo Frigo, Charles E. Leiserson, Keith Randall. PLDI '98, Montreal, Canada, June 1998. *Cilk is a lightweight language and runtime for writing parallel programs, and an excellent example of the work-stealingparadigm.*

[G83] "Using Cache Memory To Reduce Processor-Memory Traffic" by James R. Goodman. ISCA '83, Stockholm, Sweden, June 1983. *The pioneering paper on how to use bus snooping, i.e., paying attention to requests you see on the bus, to build a cache coherence protocol. Goodman's research over many years at Wisconsin is full of cleverness, this being but one example.*

[M11] "Towards Transparent CPU Scheduling" by Joseph T. Meehean. Doctoral Dissertation at University of Wisconsin—Madison, 2011. *A dissertation that covers a lot of the details of how modern Linux multiprocessor scheduling works. Pretty awesome! But, as co-advisors of Joe's, we may be a bit biased here.*

[SHW11] "A Primer on Memory Consistency and Cache Coherence" by Daniel J. Sorin, Mark D. Hill, and David A. Wood. Synthesis Lectures in Computer Architecture. Morgan and Clay- pool Publishers, May 2011. *A definitive overview of memory consistency and multiprocessor caching. Required reading for anyone who likes to know way too much about a given topic.*

[SA96] "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Pro- portional Share Resource Allocation" by Ion Stoica and Hussein Abdel-Wahab. Technical Re- port TR-95-22, Old Dominion University, 1996. *A tech report on this cool scheduling idea, from Ion Stoica, now a professor at U.C. Berkeley and world expert in networking, distributed systems, andmany other things.*

[TTG95] "Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Mul- tiprocessors" by Josep Torrellas, Andrew Tucker, Anoop Gupta. Journal of Parallel and Dis- tributed Computing, Volume 24:2, February 1995. *This is not the first paper on the topic, but it has citations to earlier work, and is a more readable and practical paper than some of the earlier queuing-based analysis papers.*

Ödev (Homework (Simulation))

IBu ödevde, multi.py'yi çok işlemcili bir CPU programlayıcıyı simüle etmek için kullanacağız ve bazı ayrıntılarını öğreneceğiz. Simülatör ve seçenekleri hakkında daha fazla bilgi için ilgili BENİOKU'yu okuyun.

Sorular(Questions)

1.İşe başlamak için, etkili bir çok işlemcili programlayıcının nasıl oluşturulacağını incelemek için simülatörü nasıl kullanacağımızı öğrenelim. İlk simülasyon, çalışma zamanı 30 ve çalışma seti boyutu 200 olan tek bir işi çalıştıracaktır. Bu işi (burada iş 'a' olarak adlandırılır) simüle edilmiş bir CPU üzerinde aşağıdaki gibi çalıştırın: `./multi.py -n 1 -L a:30:200.` Ne kadar sürede tamamlanır? Son yanıtı görmek için -c bayrağını ve işin adım adım izini ve nasıl programlandığını görmek için -t bayrağını açın.

./multi.py -n 1 -L a:30:200 -c -t komutunu çalıştırırsanız, işin çalışma zamanı belirtildiği için 'a' işini tamamlamak 30 birim zaman alacaktır. 30 olarak. -c bayrağı son yanıtı yazdıracak ve -t bayrağı işin adım adım izini ve nasıl zamanlandığını yazdıracaktır. Örneğin, çıktı şöyle görünebilir:

Zaman 0: İş geldi (gerekli süre: 30, çalışma seti: 200)

Zaman 0: CPU 0'da çalışan bir iş

Zaman 1: CPU 0'da çalışan bir iş

Zaman 2: CPU 0'da çalışan bir iş

...

Zaman 29: CPU 0'da çalışan bir iş

Zaman 30: İş bitti

multi.py programı, işleri belirli sayıda CPU üzerinde planlayan çok görevli bir işletim sisteminin simülasyonudur. -n bayrağı, simülasyonda kullanılacak CPU sayısını belirtir ve -L bayrağı, isim:zaman:çalışma_kümesi_boyutu biçiminde çalıştırılacak işlerin bir listesini belirtir. Bu durumda, -L bayrağı, çalışma süresi 30 ve çalışma kümesi boyutu 200 olan 'a' adlı tek bir işi belirtir..

2. Şimdi işin çalışma kümesini (boyut=200) önbelleğe sığdırmak için önbellek boyutunu artırın (varsayılan olarak boyut=100'dür); örneğin, `run ./multi.py -n 1 -L a:30:200 -M 300.` İşin önbelleğe sığdığında ne kadar hızlı çalışacağını tahmin edebilir misiniz? (ipucu: -r bayrağı tarafından ayarlanan ısınma hızının anahtar parametresini unutmayın) Çözme bayrağı (-c) etkinken çalıştırarak cevabınızı kontrol edin.

./multi.py -n 1 -L a:30:200 -M 300 -c komutunu çalıştırırsanız, 'a' işi 300 boyutuna sahip önbelleğe sığar (-M bayrağıyla belirtilir)), çünkü işin çalışma seti boyutu önbellek boyutundan daha küçük olan 200'dür. İşin önbelleğe sığdıktan sonra çalışma hızı, -r bayrağıyla belirtilen ısınma hızına bağlı olacaktır. Isınma oranı, programın ana bellekten verilere erişmeye

kıyasla önbellekten verilere ne kadar hızlı erişebileceğini belirler. Isınma oranı yüksek ise program daha hızlı çalışacaktır çünkü önbellekten gelen verilere daha hızlı erişebilmektedir. önbellek kullanılmıyordu çünkü ısıtma hızı, önbellek erişim süresinin ana bellek erişim süresine oranını temsil ediyor ve ısıtma hızı 1 ise, önbellek erişim süresi ana bellek erişim süresiyle aynı. Isınma oranı 1'den küçük bir değere ayarlanırsa önbellek erişim süresi ana bellek erişim süresinden daha hızlı olduğu için program daha hızlı çalışacaktır. Isınma oranının işin yürütme süresi üzerindeki etkisini görmek için ./multi.py -n 1 -L a:30:200 -M 300 -c -r 0.5 komutunu çalıştırarak ısıtma oranını şu şekilde ayarlayabilirsiniz: 0,5. Önbellek erişim süresi, ana bellek erişim süresinin yarısı olduğu için bu, programın daha hızlı çalışmasına neden olacaktır. İşin yürütme süresini nasıl etkilediğini görmek için komutu farklı ısıtma hızı değerleri ile çalıştırmayı da deneyebilirsiniz.

3. multi.py ile ilgili harika bir şey, farklı izleme bayraklarıyla neler olup bittiğine dair daha fazla ayrıntı görebilmenizdir. Yukarıdakiyle aynı simülasyonu çalıştırın, ancak bu sefer kalan süre takibi etkinken (-T). Bu bayrak, hem her zaman adımında bir CPU'da programlanan işi hem de her bir onay çalıştırıldıktan sonra o işin ne kadar çalışma süresi kaldığını gösterir. Bu ikinci sütunun nasıl azaldığına dair ne fark ettiniz?

./multi.py -n 1 -L a:30:200 -M 300 -T komutunu çalıştırırsanız, program iş planlamasının izini ve her zaman adımında her işin ne kadar çalışma süresinin kaldığını yazdırır .

Çıktı şöyle bir şeye benzeyebilir:

Zaman 0: İş geldi (gerekli süre: 30, çalışma seti: 200)

Zaman 0: CPU 0'da çalışan bir iş (kalan süre: 30)

Zaman 1: CPU 0'da çalışan bir iş (kalan süre: 29)

Zaman 2: CPU 0'da çalışan bir iş (kalan süre: 28)

...

Zaman 29: CPU 0'da çalışan Job a (kalan süre: 1)

Süre 30: Job a tamamlandı (kalan süre: 0)

Her iş için kalan süreyi gösteren ikinci sütunun her zaman adımında 1 azaldığına dikkat etmelisiniz. Bunun nedeni, işin CPU üzerinde her zaman adımında 1 birim zaman çalışması ve işin çalışma süresinin 30 olmasıdır, bu nedenle işin tamamlanması 30 zaman adımı alacaktır.

4. Şimdi -C bayrağıyla her bir iş için her bir CPU önbelleğinin durumunu göstermek üzere bir izleme biti daha ekleyin. Her iş için, her önbellekte bir boşluk (önbellek o iş için soğuksa) veya bir 'w' (önbellek o iş için sıcaksa) gösterilir. Bu basit örnekte 'a' işi için önbellek hangi noktada ısınıyor? Isınma zamanı parametresini (-w) varsayılan değerden daha düşük veya daha yüksek değerlere değiştirdiğinizde ne olur?

./multi.py -n 1 -L a:30:200 -M 300 -T -C komutunu çalıştırırsanız, program iş planlamasının bir izini, her işin her seferinde ne kadar çalışma süresi kaldığını yazdırır. zaman adımı ve her iş için önbelleğin durumu.

Çıktı şöyle bir şeye benzeyebilir:

Zaman 0: İş geldi (gerekli süre: 30, çalışma seti: 200)

Zaman 0: CPU'da çalışan bir iş 0 (kalan süre: 30) | CPU 0 önbelleği:

Zaman 1: CPU 0'da çalışan bir iş (kalan süre: 29) | CPU 0 önbelleği:

Zaman 2: CPU 0'da çalışan bir iş (kalan süre: 28) | CPU 0 önbelleği:

...

Zaman 29: CPU 0'da çalışan bir iş (kalan süre: 1) | CPU 0 önbelleği:

Zaman 30: İş bitti (kalan süre: 0) | CPU 0 önbelleği:

Bu basit örnekte, önbellek 'a' işi için ilk adımda ısınır, çünkü işin çalışma kümesi boyutu 200'dür, bu önbellek boyutu 300'den küçüktür, bu nedenle çalışma kümesinin tamamı önbelleğe sığar .

-w bayrağıyla belirtilen ısınma süresi parametresini varsayılan değerden (1 olan) daha düşük bir değere değiştirirseniz, önbellek iş için daha hızlı ısınır çünkü ısınma süresi adım sayısını belirler. Bir iş çalışmaya başladıktan sonra önbelleğin ısınması için gereken süre. Isınma süresini daha yüksek bir değere ayarlarsanız, önbelleğin ısınması daha uzun sürer.

Örneğin, ./multi.py -n 1 -L a:30:200 -M 300 -T -C -w 2 komutunu çalıştırırsanız, çıktı, önbelleğin ısınmasının 2 zaman adımı sürdüğünü gösterecektir. 'a' işi için:

Zaman 0: İş geldi (gerekli süre: 30, çalışma seti: 200)

Zaman 0: CPU'da çalışan bir iş 0 (kalan süre: 30) | CPU 0 önbelleği:

Zaman 1: CPU 0'da çalışan bir iş (kalan süre: 29) | CPU 0 önbelleği:

Zaman 2: CPU 0'da çalışan bir iş (kalan süre: 28) | CPU 0 önbelleği: w

...

Zaman 29: CPU 0'da çalışan bir iş (kalan süre: 1) | CPU 0 önbelleği: w

Zaman 30: İş bitti (kalan süre: 0) | CPU 0 önbelleği: w

Isınma süresini daha yüksek bir değere ayarlarsanız, önbelleğin ısınması daha uzun sürer. Örneğin, ./multi.py -n 1 -L a:30:200 -M 300 -T -C -w 3 komutunu çalıştırırsanız, önbelleğin iş için ısınması 3 zaman adımı alacaktır' a':

Zaman 0: İş geldi (gerekli süre: 30, çalışma seti: 200)

Zaman 0: CPU'da çalışan bir iş 0 (kalan süre: 30) | CPU 0 önbelleği:

Zaman 1: CPU 0'da çalışan bir iş (kalan süre: 29) | CPU 0 önbelleği:

Zaman 2: CPU 0'da çalışan bir iş (kalan süre: 28) | CPU 0 önbelleği:

Zaman 3: CPU 0'da çalışan bir iş (kalan süre: 27) | CPU 0 önbelleği: w

...

Zaman 29: CPU 0'da çalışan bir iş (kalan süre: 1) | CPU 0 önbelleği: w

Zaman 30: İş bitti (kalan süre: 0) | CPU 0 önbelleği: w

5. Bu noktada, simülatörün tek bir CPU'da çalışan tek bir iş için nasıl çalıştığına dair iyi bir fikir edinmiş olmalısınız. Ama bu bir çok işlemcili CPU programlama bölümü değil mi? E tabi! Öyleyse birden çok işle çalışmaya başlayalım. Spesifik olarak, aşağıdaki üç işi iki CPU'lu bir sistemde çalıştıralım (ör./multi.py-n2-L a:100:100,b:100:50,c:100:50 yazın). Döngüsel bir merkezi planlayıcı göz önüne alındığında, bunun ne kadar süreceğini tahmin edebilir misiniz? Haklı olduğmuzu görmek için -c'yi ve ardından adımları görmek için -t ile ayrıntılara dalın ve ardından -C ile bu işler için önbelleklerin etkili bir şekilde ısıtılıp ısıtılmadığını görün. Ne farkettin?

6. Şimdi, bölümde açıklandığı gibi önbellek yakınlığını incelemek için bazı açık kontroller uygulayacağız. Bunu yapmak için -A bayrağına ihtiyacınız olacak. Bu bayrak, programlayıcının belirli bir işi hangi CPU'lara yerleştirebileceğini sınırlamak için kullanılabilir. Bu durumda, 'a'yı CPU 0 ile sınırlandırırken, 'b' ve 'c' işlerini CPU 1'e yerleştirmek için kullanalım. Bu sihir bunu yazarak gerçekleştirilir. `./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -A a:0,b:1,c:1`; gerçekte neler olup bittiğini görmek için çeşitli izleme seçeneklerini açmayı unutmayın! Bu versiyonun ne kadar hızlı çalışacağını tahmin edebiliyor musunuz? Neden daha iyi oluyor? İki işlemcideki diğer 'a', 'b' ve 'c' kombinasyonları daha hızlı mı yoksa daha yavaş mı çalışır?

7. Çoklu işlemcileri önbelleğe almanın ilginç bir yönü, tek bir işlemci üzerinde çalışan işlere kıyasla, birden çok CPU (ve bunların önbellekleri) kullanılırken, işlerin beklenenden daha iyi hızlanması fırsatıdır. Spesifik olarak, N CPU üzerinde çalıştığınızda, bazen N faktöründen daha fazla hızlandırabilirsiniz, bu durum süper lineer hızlanma olarak adlandırılır. Bunu denemek için, üç iş oluşturmak üzere küçük bir önbellekle (-M 50) buradaki iş tanımını (-L a:100:100,b:100:100,c:100:100) kullanın. Bunu 1, 2 ve 3 CPU'lu sistemlerde çalıştırın (-n 1, -n 2, -n 3). Şimdi aynısını yapın, ancak 100 boyutunda CPU başına daha büyük bir önbellekle. CPU sayısı arttıkça performans hakkında ne fark ediyorsunuz? Tahminlerinizi doğrulamak için -c'yi ve daha da derine

dalmak için diğer izleme işaretlerini kullanın.

8. Simülatörün incelenmeye değer diğer bir yönü, CPU başına zamanlama seçeneği olan -p bayrağıdır. Tekrar iki CPU ile çalıştırın ve bu üç iş yapılandırması (-L a:100:100,b:100:50,c:100:50). Yukarıda uyguladığınız elle kontrol edilen benzeşim sınırlarının aksine bu seçenek nasıl çalışır? "Gözetleme aralığını" (-P) daha düşük veya daha yüksek değerlere değiştirdiğinizde performans nasıl değişir? Bu CPU başına yaklaşım, CPU sayısı arttıkça nasıl çalışır?

9. Son olarak, rastgele iş yükleri oluşturmaktan çekinmeyin ve bunların farklı işlemci sayıları, önbellek boyutları ve zamanlama seçenekleri üzerindeki performanslarını tahmin edip edemeyeceğinizi görün. Bunu yaparsanız, kısa sürede çok işlemcili bir planlama ustası olacaksınız ki bu oldukça harika bir şey. Allah yardımcınız Olsun!