ÇOK İŞLEMCİLİ ZAMANLAMA

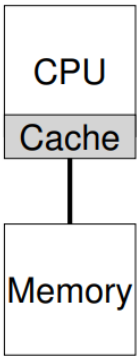
Bu bölüm, **çok işlemcili zamanlamanın (multiprocessor scheduling)** temellerini tanıtacaktır. Bu konu nispeten ileri düzey olduğundan, eşzamanlılık konusuna biraz detaylı çalıştıktan sonra ele almak daha iyi olabilir (yani, kitabın ikinci ana "kolay parçası").

Yıllarca bilgi işlem yelpazesinin sadece en üst seviyesinde var olduktan sonra, **çok işlemcili (multiprocessor)** sistemler giderek yaygınlaştı ve masaüstlerinde, dizüstü bilgisayarlarda ve hatta mobil cihazlarda kendine yer buldu. Bu çoğalmanın kaynağı, birden çok CPU çekirdeğinin tek bir çipte toplandığı **çok çekirdekli (multicore)** işlemcinin yükselişidir; bilgisayar mimarları çok fazla güç kullanmadan tek bir CPU'yu çok daha hızlı yapmakta zorlandıklarından bu çipler popüler hale geldi. Ve böylece hepimiz için birkaç CPU var, bu iyi bir şey, değil mi?

Elbette, birden fazla CPU'nun varlığıyla ortaya çıkan birçok zorluk var. Birincil olarak, tipik bir uygulamanın (yani, yazdığınız bazı C programları) yalnızca tek bir CPU kullanmasıdır; daha fazla CPU eklemek o uygulamanın daha hızlı çalışmasını sağlamaz. Bu sorunu gidermek için uygulamanızı **paralel (parallel)** olarak çalışacak şekilde, belki de **iş parçacıklarını (threads)** kullanarak yeniden yazmanız gerekir (bu kitabın ikinci bölümünde ayrıntılı olarak tartışıldığı gibi). Çok iş parçacıklı uygulamalar, işi birden çok CPU'ya yayabilir ve böylece daha fazla CPU kaynağı verildiğinde daha hızlı çalışabilir.

AYRICA: İLERİ SEVİYELİ BÖLÜMLER

İleri seviyeli bölümler, kitabın geniş bir alanından materyalin gerçekten anlaşılmasını gerektirirken, söz konusu önkoşul materyallerinden daha önceki bir bölüme mantıksal olarak sığar. Örneğin, çok işlemcili zamanlama hakkındaki bu bölüm, ilk olarak eşzamanlılık bölümünün ortasındaki kısmı okuduysanız daha anlamlı olacaktır; ancak, mantıksal olarak kitabın sanallaştırma ve CPU zamanlaması ile ilgili bölümüne uyuyor. Bu nedenle. Bu tür bölümlerin sıra dışı ele alınması önerilir; bu durumda, kitabın ikinci parçasından sonra.



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

Uygulamaların ötesinde, işletim sistemi için ortaya çıkan yeni bir sorun (şaşırtıcı olmayan bir şekilde!) **çok işlemcili zamanlama (multiprocessor scheduling)** sorunudur. Şimdiye kadar tek işlemcili zamanlamanın ardındaki bir dizi ilkeyi tartıştık; bu fikirleri birden fazla CPU üzerinde çalışacak şekilde nasıl genişletebiliriz? Hangi yeni sorunların üstesinden gelmeliyiz? Ve böylece sorunumuz:

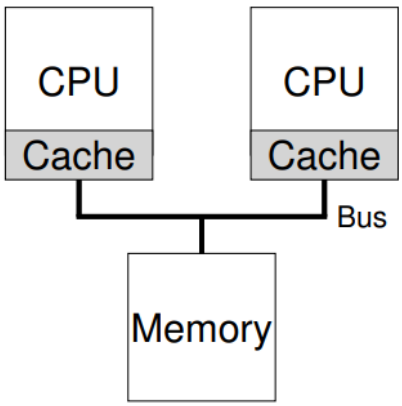
CRUX: ÇOKLU CPU'LARDA İŞLER NASIL PLANLANIR?

İşletim sistemi birden çok CPU'daki işleri nasıl planlamalıdır? Hangi yeni sorunlar ortaya çıkar? Eski teknikler işe yarar mı yoksa yeni fikirler mi gerekir?

10.1 Arkaplan: Çok İşlemcili Mimari

Çok işlemcili zamanlamayı kapsayan yeni sorunları anlamak için, tek CPU donanımı ile çoklu CPU donanımı arasındaki yeni ve temel farkı anlamamız gerekir. Bu fark,i donanım **önbelleklerinin (caches)** kullanımına ve verilerin birden fazla işlemci arasında nasıl paylaşıldığına odaklanır. Şimdi bu konuyu daha üst düzeyde tartışabiliriz. Ayrıntılar başka bir yerde [CSG99], özellile üst düzey veya yüksek lisans bilgisayar mimarisi kursunda mevcuttur.

Tek CPU'lu bir sistemde, genel olarak işlemcinin programları daha hızlı çalıştırmasına yardımcı olan bir donanım **önbellek (hardware caches**) hiyerarşisi vardır. Önbellekler (genelde) sistemin ana belleğinde bulunan popüler verilerin kopyalarını tutan küçük, hızlı belleklerdir. Ana bellek, aksine, tüm verileri tutar ama bu daha büyük olan 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)

Örneğin, 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 (64KB diyelim) ve büyük bir ana belleği var. 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 CPU önbelleğine koyar. Daha sonra program aynı veri öğesini tekrar getirirse, CPU önce onu önbellekte kontrol eder; orada bulursa veriler çok daha hızlı getirilir (yalnızca birkaç nanosaniye diyebiliriz) ve böylece program daha hızlı çalışır.

Bu nedenle önbelekler; iki tür **yerellik (locality)** kavramına dayanır: **zamanda yerellik (temporal locality)** ve **uzayda yerellik (spatial locality)**. Zamanda yerelliğin arkasındaki fikir, bir veri parçasına erişildiğinde, ona yakın gelecekte

tekrar erişilmesinin muhtemel olmasıdır; değişkenlere ve hatta yönergelere bir döngü içinde tekrar tekrar erişildiğini düşünün. Uzayda yerelliğin arkasındaki fikir, eğer bir program x adresindeki bir veri öğesine erişirse, x yakınlarındaki veri öğelerine de erişme olasılığının yüksek olmasıdır; burada bir dizi boyunca akan bir programı veya birbiri ardına yürütülen talimatları düşünün. Pek çok programda bu tür yerellikler bulunduğundan, donanım sistemleri hangi verilerin önbelleğe alınacağı konusunda iyi tahminler yapabilir ve bu nedenle iyi çalışır.

Şimdi zor kısma geçelim: Şekil 10.2'de gördüğümüz gibi, tek bir ana belleğin paylaşıldığı sistemde birden çok işlemciye sahip olduğunuzda ne olur?

Anlaşıldığı üzere; birden çok CPU ile önbelleğe alma işlemi daha karmaşıktır. Örneğin, CPU 1'de çalışan bir programın A adresindeki bir veri öğesini (D değerine sahip) okuduğunu hayal edin; veriler CPU 1'deki önbellekte olmadığı için sistem onu ana bellekten getirir 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 (genelde) bunu sonra yapar. Ardından, işletim sisteminin programı çalıştırmayı durdurmaya ve CPU 2'ye taşımaya karar verdiğini varsayalım. Program daha sonra A adresindeki değeri yeniden okur; CPU 2'nin önbelleğinde böyle bir veri yoktur ve bu nedenle sistem değeri ana bellekten alır ve doğru D' değeri yerine eski D değerini alır. Oops!

Bu genel sorun, önbellek tutarlılığı (cache coherence) sorunu olarak adlandırılır ve sorunun çözülmesiyle 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 (veya üç) alın.

Temel çözüm donanım tarafından sağlanır: donanım, bellek erişimlerini izleyerek temelde "doğru olanın" olmasını ve tek bir paylaşılan belleğin görünümünün korunmasını sağlayabilir. Bunu veri yolu tabanlı bir sistemde yapmanın bir yolu (yukarıda açıklandığı gibi), **veri yolu gözetleme (bus snooping)** [G83] olarak bilinen eski bir tekniği kullanmaktır; her önbellek, onları 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 **güncelleyecek (update)** (yani yeni değeri önbelleğine koyacaktır). Geri yazma önbellekleri, yukarıda ima edildiği

gibi, bunu daha karmaşık hale getirir (çünkü ana belleğe yazma daha henüz görünür değildir), ancak temel şemanın nasıl çalıştığını hayal edebilirsiniz.

10.2 Senkronizasyonu Unutmayın

Tutarlılık sağlamak için önbelleklerin tüm bu işi yaptığı göz önüne alındığında, programlar (veya işletim sisteminin kendisi) paylaşılan verilere eriştiklerinde herhangi bir şey için endişelenmek zorunda mı? Yanıt ne yazık ki evet ve bu kitabın eş zamanlılık konusundaki ikinci bölümünde ayrıntılı olarak açıklanmıştır. Burada ayrıntılara girmeyecek olsak da 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 muhtemelen karşılıklı dışlama ilkelleri (kilitler gibi) kullanılmalıdır (**kilitsiz (lock-free)** veri yapıları oluşturmak gibi diğer yaklaşımlar karmaşıktır ve yalnızca ayrıntılar için eşzamanlılık ile ilgili parçadaki kilitlenme ile ilgili 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, temeldeki tutarlılık protokolleriyle bile beklendiği gibi çalışmaz; veri yapısını yeni durumuna atomik olarak güncellemek için kilitlere ihtiyaç vardır.

Bunu daha somut hale getirmek için, Şekil 10.3'te gördüğümüz gibi, paylaşılan bir bağlantılı listeden bir öğeyi kaldırmak için kullanılan bu kod dizisini ele alın. 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 yürütürse, tmp değişkeninde depolanan geçerli kafa değerine sahip olacaktır; Eğer İş Parçacığı 2 daha sonra ilk satırı da yürütürse, kendi özel tmp değişkeninde depolanan aynı kafa değerine sahip olacaktır (yığın üzerinde tmp tahsis edilmiştir ve böylece her iş parçacığının kendisi için kendi özel deposu olacaktır). Bu nedenle, her iş parçacığı listenin başından bir öğeyi kaldırmak yerine, aynı baş öğeyi çıkarmaya çalışacak ve bu da her türlü soruna yol açacaktır (örneğin, 10. Satırdaki baş öğeden iki kez kurtulma girişimi; potansiyel olarak aynı veri değerini iki kez döndürerek).

Çözüm, elbette, bu tür rutinleri **kilitleme (locking)** yoluyla düzeltmektir. Bu durumda, basit bir muteks (örneğin, pthread_mutex_t m;) tahsis etmek ve ardından rutinin başına bir lock(&m) ve sonuna bir unlock(&m) eklemek sorunu çözecektir, kod istenildiği gibi çalışacaktır. Maalesef, göreceğimiz gibi, böyle bir yaklaşım, özellikle performans açısından sorunsuz değildir. Özellikle, CPU sayısı arttıkça, senkronize edilmiş bir paylaşılan veri yapısına erişim oldukça yavaşlar.

10.3 Son Bir Sorun: Önbellek Yakınlığı

Çok işlemcili bir önbellek planlayıcı oluştururken son bir sorun ortaya çıkıyor; **önbellek yakınlığı (cache affinity)** [TTG95]. Bu kavram basittir: bir işlem, belirli bir CPU üzerinde çalıştırıldığında, CPU'nun önbelleklerinde (ve TLB'lerde) oldukça fazla durum oluşturur. İşlemin bir sonraki çalışma zamanında, aynı CPU'da çalıştırmak genellikle avantajlıdır, çünkü durumunun bir kısmı o CPU'daki önbelleklerde zaten mevcutsa daha hızlı çalışacaktır. Bunun yerine, her seferinde farklı bir CPU'da bir işlem çalıştırılırsa, her çalıştırıldığında durumu yeniden yüklemek zorunda kalacağı için işlemin performansı daha kötü olacaktır (donanımın önbellek tutarlılık protokolleri sayesinde farklı bir CPU'da düzgün çalışacağını unutmayın). Bu nedenle, çok işlemcili bir zamanlayıcı, zamanlama kararlarını verirken önbellek yakınlığını dikkate almalı, belki de mümkünse bir işlemi aynı CPU'da tutmayı tercih etmelidir.

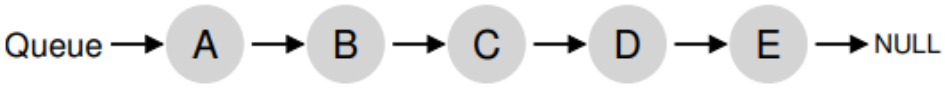
10.4 Tek Kuyruklu Zamanlama

Arka plan hazır olduğunda, 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; buna **tek kuyruklu çok işlemcili zamanlama (single-queue multiprocessor scheduling)** veya kısaca **SQMS** diyoruz. Bu yaklaşımın basitlik avantajı vardır; 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).

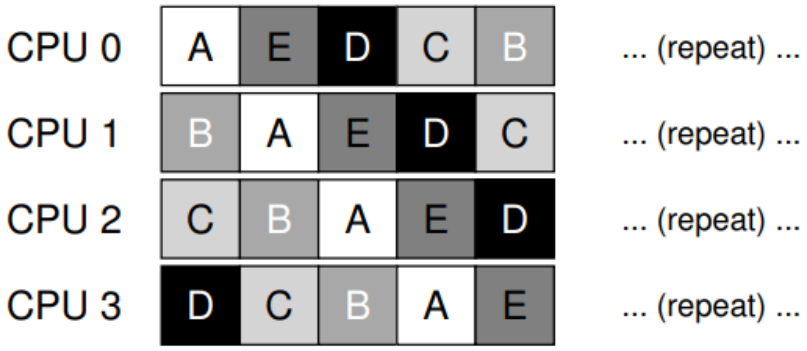
Yine de SQMS'nin bariz eksiklikleri vardır. İlk sorun, **ölçeklenebilirlik (scalability)** eksikliğidir. Zamanlayıcının birden fazla CPU üzerinde doğru şekilde çalışmasını sağlamak için, geliştiriciler yukarıda açıklandığı gibi koda bir tür **kilitleme (locking)** eklerler. Kilitler, SQMS kodu tek kuyruğa 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 yüküne giderek daha fazla zaman harcıyor ve sistemin yapması gereken işi yapmak için daha az zaman harcıyor (not: bir gün bunun gerçek bir ölçümünü buraya dahil etmek harika olurdu).

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

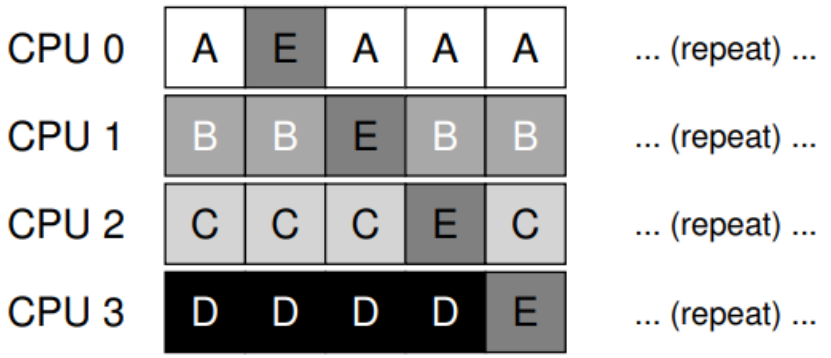


Zamanla, her işin bir zaman diliminde çalıştığını ve ardından başka bir işin seçildiğini varsayarsak, burada CPU'lar arasında olası bir iş çizelgesi var:



Her bir CPU, küresel olarak paylaşılan sıradan çalıştırılacak bir sonraki işi seçtiğinden, her iş CPU'dan CPU'ya sıçrayarak sona erer ve 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. Özellikle, biri bazı işler için yakınlık sağlayabilir, ancak yükü dengelemek için 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 (migrating)** ve böylece çoğu için yakınlık 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.

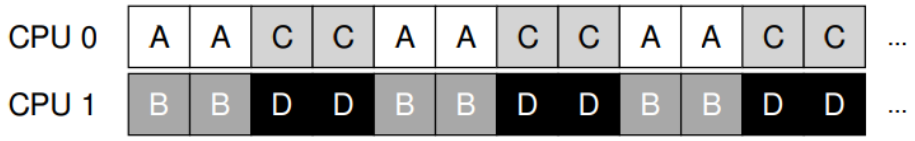
10.5 Çok Kuyruklu Zamanlama

Tek kuyruklu programlayıcıların neden olduğu sorunlar nedeniyle, bazı sistemler, örneğin CPU başına bir tane olmak üzere birden çok sırayı tercih eder. Bu yaklaşıma **çok kuyruklu çok işlemcili çizelgeleme (multi-queue multiprocessor scheduling) (veya MQMS)** diyoruz.

MQMS'de, temel zamanlama çerçevemiz birden fazla zamanlama kuyruğundan oluşur. Her bir sıra, büyük olasılıkla, hepsini bir kez deneme gibi belirli bir zamanlama disiplinini izleyecektir, ancak elbette herhangi bir algoritma kullanılabilir. Bir iş sisteme girdiğinde, bazı buluşsal yöntemlere göre (örneğin, rastgele veya diğerlerinden daha az işi olan birini seçmek) tam olarak bir zamanlama kuyruğuna yerleştirilir. Ardından, temelde bağımsız olarak programlanır, böylece tek sıra yaklaşımında bulunan bilgi paylaşımı ve senkronizasyon sorunlarından kaçınılır.

Örneğin, yalnızca iki CPU'nun (CPU 0 ve CPU 1 olarak etiketlenmiş) bulunduğu bir sistemimiz olduğunu ve sisteme bir miktar işin 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:

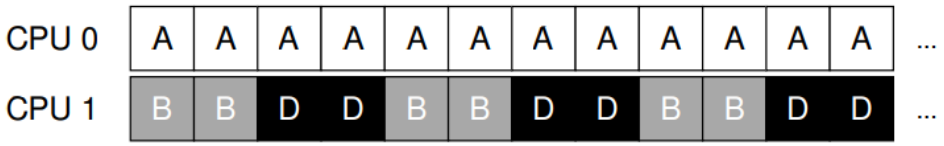
Kuyruk zamanlama ilkesine bağlı olarak, artık her CPU'nun neyin çalıştırılacağına karar verirken aralarından seçim yapabileceği iki görevi vardır. Örneğin, **hepsini bir kez deneme (round robin)** ile sistem şuna benzeyen bir zamanlama oluşturabilir:



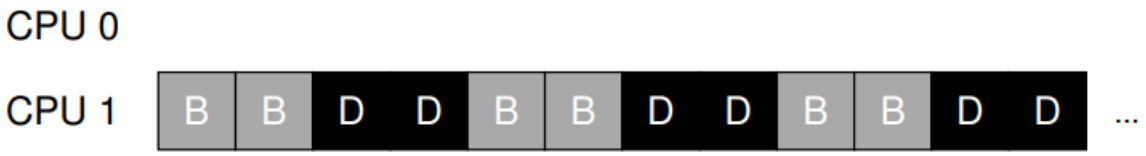
MQMS, doğası gereği daha ölçeklenebilir olması gerektiği için SQMS'nin belirgin bir avantajına sahiptir. CPU sayısı arttıkça sıra sayısı da artar ve bu nedenle kilit ve önbellek çekişmesi merkezi bir sorun haline gelmemelidir. Ek olarak, MQMS özünde önbellek benzerliği sağlar; işler aynı CPU'da kalır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

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

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



Bu şemadan da görebileceğiniz gibi, A, B ve D'den iki kat daha fazla CPU alıyor ki istediğimiz sonuç bu değil. Daha da kötüsü hem A hem de C'nin bittiğini ve sistemde yalnızca B ve D işlerini bıraktığını düşünelim. İki zamanlama kuyruğu ve sonuçta ortaya çıkan zaman çizelgesi şöyle görünecektir:



Ne kadar korkunç – CPU 0 boşta! (buraya dramatik ve uğursuz müzik ekleyin) Ve bu nedenle CPU kullanım zaman çizelgemiz oldukça üzücü görünüyor.

Öyleyse, zayıf çok sıralı çok işlemcili zamanlayıcı ne yapmalıdır? Nasıl sinsi yük dengesizliği probleminin üstesinden gelebiliriz ve Decepticon'ların şeytani güçlerini yenebiliriz... Bu harika kitapla pek alakalı olmayan sorular sormayı nasıl durdurabiliriz?

CRUX: YÜK DENGESİZLİĞİYLE NASIL BAŞA ÇIKILIR

Çok kuyruklu çok işlemcili bir programlayıcı, istenen programlama hedeflerine daha iyi ulaşmak için yük dengesizliğini nasıl ele almalıdır?

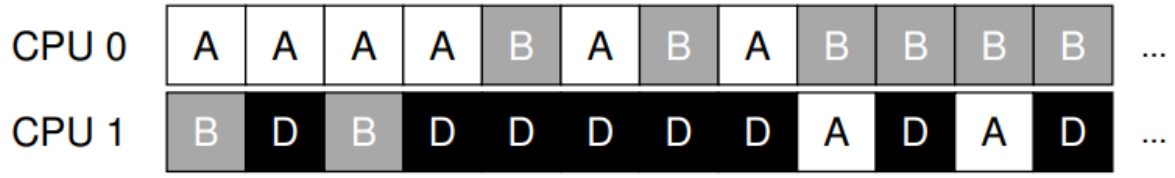
Bu sorunun bariz cevabı, (bir kez daha) **geçiş (migration)** olarak adlandırdığımız bir teknik olan işleri hareket ettirmektir. Bir işi bir CPU'dan diğerine geçirerek gerçek yük dengesi elde edilebilir.

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, eşit şekilde dengelenmiş bir yüktür ve herkes mutludur.

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

Bu durumda, tek bir geçiş sorunu çözmez. Bu durumda ne yapardınız? Cevap, ne yazık ki, 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 bulunur. Birkaç zaman diliminden sonra B, CPU 0'da A ile rekabet etmek için taşınırken, D, CPU 1'de tek başına birkaç zaman diliminin keyfini çıkarır. Ve böylece yük dengelenir:



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

Temel yaklaşımlardan biri, **iş çalma (work stealing)** [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, ilk etapta çoklu sıra planlamasını uygulamanın tüm amacı buydu! 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 Zamanlayıcıları

İlginç bir şekilde, Linux topluluğunda, çok işlemcili bir zamanlayıcı oluşturmak için ortak bir çözüme yaklaşılmadı. Zamanla üç farklı programlayıcı ortaya çıktı: O(1) zamanlayıcı, Tamamen Adil Zamanlayıcı (CFS) ve BF Zamanlayıcı (BFS). Bahsedilen zamanlayıcıların [M11] güçlü ve zayıf yönlerine dair mükemmel bir

genel bakış için Meehean'ın tezine bakın; burada sadece birkaç temel bilgiyi 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 zamanla 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

Çok işlemcili zamanlamaya yönelik çeşitli yaklaşımlar gördük. Tek kuyruklu yaklaşım (SQMS), yükü iyi bir şekilde oluşturmak ve dengelemek için oldukça basittir, ancak doğası gereği birçok işlemciye ve önbellek yakınlığına ölçeklendirmede zorluk yaşar. Çoklu kuyruk yaklaşımı (MQMS) daha iyi ölçeklenir ve önbellek yakınlığını iyi yönetir, ancak yük dengesizliği sorunu vardır ve daha karmaşıktır. Hangi yaklaşımı benimserseniz seçin basit bir cevap yoktur: küçük kod değişiklikleri büyük davranışsal farklılıklara yol açabileceğinden, genel amaçlı bir zamanlayıcı oluşturmak göz korkutucu bir görev olmaya devam etmektedir. Sadece tam olarak ne yaptığınızı biliyorsanız veya en azından bunun için büyük miktarda para alıyorsanız böyle bir egzersizi yapın.

Referanslar

[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, Nickolai 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-stealing

paradigm.

[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 Claypool 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 Proportional Share Resource Allocation” by Ion Stoica and Hussein Abdel-Wahab. Technical Report 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, and

many other things.

[TTG95] “Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors” by Josep Torrellas, Andrew Tucker, Anoop Gupta.

Journal of Parallel and Distributed 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 queuingbased analysis papers.

Ödev (Simülasyon)

Bu ödevde, çok işlemcili bir CPU programlayıcıyı simüle etmek için multi.py kullanacağız ve bazı ayrıntılarını öğreneceğiz. Simülatör ve seçenekleri hakkında daha fazla bilgi için ilgili README dosyasını okuyun.

Sorular

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'da şu şekilde çalıştırın: ./multi.py -n 1 -L a:30:200. Tamamlanması ne kadar sürer? Son yanıtı görmek için -c bayrağını ve işin adım adım izini ve nasıl programlandığını görmek için -t bayrağını açın.

Bir CPU ve bir işle, 30 birim zamanda bitmeli.

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

İlk 10 döngü aynı olacaktır, eğer -r 2 ise sonraki 20 döngü on birim sürer.

3. multi.py ile ilgili harika bir şey vardır ki, farklı izleme bayraklarıyla neler olup bittiği hakkında 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 onay çalıştırıldıktan sonra o işin ne kadar çalışma zamanı kaldığını gösterir. İkinci sütunun nasıl azaldığı konusunda ne fark ettiniz?

CPU ısındıktan sonra her tikte -r kat daha hızlı azalır.

4. Şimdi -C bayrağıyla her iş için her bir CPU önbelleğinin durumunu göstermek için 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 süresi parametresini (-w) varsayılandan daha düşük veya daha yüksek değerlere değiştirdiğinizde ne olur?

Beklendiği gibi çalışır, -w işaretlerinden sonra olur.

5. Bu noktada, simülatörün tek bir CPU üzerinde çalışan tek bir iş için nasıl çalıştığı hakkında iyi bir fikriniz olmalıdır. Ama hey, bu çok işlemcili bir CPU planlama bölümü değil mi? Ah evet! O halde birden fazla işle çalışmaya başlayalım. Spesifik olarak, aşağıdaki üç işi iki CPU'lu bir sistemde çalıştıralım (ör. ./multi.py -n 2 -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 olup olmadığınızı görmek için -c'yi kullanın ve ardından adım adım görmek için -t ile ayrıntılara inin ve ardından -C ile bu işler için önbelleklerin etkili bir şekilde ısınıp ısınmadığını görün. Neyi fark ettiniz?

Önbellekler, çalışan kümelerin üçünü birden tutamaz. Üstelik kuantum, ısınma süresiyle aynıdır. CPU ısınır ısınmaz iş değiştirilir.

6. Şimdi, bölümde açıklandığı gibi, önbellek yakınlığını incelemek için bazı açık denetimler uygulayacağız. Bunu yapmak için -A bayrağına ihtiyacınız olacak. Bu bayrak, zamanlayıcının belirli bir işi yerleştirebileceği CPU'ları sınırlamak için kullanılabilir. Bu durumda, 'a'yı CPU 0 ile sınırlandırırken, 'b' ve 'c' işlerini CPU 1'e yerleştirmek için kullanalım. Bu sihir ./multi.py -n 2 -L a:100 :100,b:100:50, c:100:50 -A a:0,b:1,c:1 yazarak gerçekleştirilir; gerçekte neler olduğunu görmek için çeşitli izleme seçeneklerini açmayı unutmayın! Bu sürümün ne kadar hızlı çalışacağını tahmin edebilir misiniz? Neden daha iyi yapar? İki işlemcideki diğer 'a', 'b' ve 'c' kombinasyonları daha hızlı mı yoksa daha yavaş mı çalışacak?

Bence bu kombinasyon olabildiğince iyi. a, büyük çalışma seti olduğu için bir CPU'daki diğer işlerle birlikte yerleştirilemez.

7. Çoklu işlemcileri önbelleğe almanın ilginç bir yönü, birden çok CPU (ve bunların önbellekleri) kullanıldığında, işleri tek bir işlemcide çalıştırmaya kıyasla

beklenenden daha iyi hızlandırma fırsatıdır. Spesifik olarak, N CPU üzerinde çalıştığınızda, bazen N faktöründen daha fazla hızlandırabilirsiniz, bu durum süper lineer hızlanma olarak adlandırılır. Bunu denemek için, küçük bir önbellekle (-M 50) buradaki iş tanımını (-L a:100:100,b:100:100,c:100:100) kullanarak üç iş oluşturun. 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 ölçeklendikçe performans hakkında ne fark ediyorsunuz? Tahminlerinizi doğrulamak için -c'yi ve daha da derine inmek için diğer izleme işaretlerini kullanın.

Bir işi yalnızca bir performansta çalıştırabilirsek, daha iyi önbellek kullanımı nedeniyle çok daha fazla artış olur.

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? "Pek aralığını" (-P) daha düşük veya daha yüksek değerlere değiştirdiğinizde performans nasıl değişir? Bu CPU başına yaklaşım, CPU sayısı ölçeklenirken nasıl çalışır?

Daha iyi CPU kullanımı nedeniyle manuel yakınlıktan daha iyi çalışır; -P'nin daha düşük değerleri ile performans artışı olur.

9. Son olarak, rastgele iş yükleri oluşturmaktan çekinmeyin ve farklı işlemci sayıları, önbellek boyutları ve zamanlama seçenekleri üzerindeki performanslarını tahmin edip edemeyeceğinize bakın. Bunu yaparsanız, kısa sürede çok işlemcili bir planlama ustası olacaksınız ki bu oldukça harika bir şey. İyi şanslar!