Çok İşlemcili Zamanlama (Gelişmiş)

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

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

Elbette, 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, o tek uygulamanın daha hızlı çalışmasını sağlamaz. Bu sorunu çözmek için, uygulamanızı **paralel** çalışacak şekilde, belki de dizileri kullanarak yeniden yazmanız gerekecek (bu kitabın ikinci bölümünde ayrıntılı olarak tartışıldığı gibi). Çok **iş parçacıklı(threads)** uygulamalar, işi birden çok CPU'ya yayabilir ve böylece daha fazla CPU kaynağı verildiğinde daha hızlı çalışabilir.

KENARA: GELİŞMİŞ BÖLÜMLER
İleri bölümler, kitabın geniş bir alanından materyaller gerektirir.

mantıksal olarak daha önceki bir bölüme sığarken gerçekten anlayın
söz konusu önkoşul malzemeleri seti. Örneğin, çok işlemcili zamanlama ile ilgili bu bölüm, önce ortadaki kısmı okuduysanız çok daha anlamlı olacaktır.
eşzamanlılık üzerine parça; ancak, mantıksal olarak kitabın bölümüne uyuyor
sanallaştırma (genel olarak) ve CPU zamanlaması (özel olarak). Böylece, o
bu tür bölümlerin sıra dışı ele alınması önerilir; bu durumda, sonra
kitabın ikinci parçası.

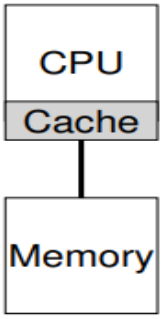


Figure 10.1: **Single CPU With Cache**

Uygulamaların ötesinde, işletim sistemi için ortaya çıkan yeni bir sorun (şaşırtıcı olmayan bir şekilde!) **çok işlemcili programlama (multiprocessor scheduling)** sorunudur. Şimdiye kadar tek işlemcili programlamanın ardındaki bir dizi ilkeyi tartıştık; bu fikirleri birden fazla CPU üzerinde çalışacak şekilde nasıl genişletebiliriz? Üstesinden gelmemiz gereken yeni sorunlar? Ve böylece sorunumuz:

CRUX: ÇOKLU CPU'LARDA İŞLER NASIL PLANLANIR

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

10.1 Arka Plan: Çok İşlemcili Mimari

Çok işlemcili zamanlamayı çevreleyen yeni sorunları anlamak için, çok işlemcili zamanlamayı çevreleyen yeni ve temel bir farkı anlamamız gerekir. Tek CPU donanımı ve çoklu CPU donanımı. Bu farklılık merkezleri donanım önbelleklerinin kullanımı etrafında (ör. Şekil 10.1) ve tam olarak nasıl veriler birden çok işlemci arasında paylaşılır. Şimdi bu konuyu daha üst düzeyde tartışıyoruz. Ayrıntılar başka bir yerde [CSG99] mevcuttur, özellikle bir üst düzey veya belki de yüksek lisans bilgisayar mimarisi kursunda. Tek CPU ya ait bir sistemde, bir donanım hiyerarşisi vardır. Genel olarak işlemcinin programları daha hızlı çalıştırmasına yardımcı olan önbellekler. Önbellekler popüler verilerin kopyalarını (genel olarak) tutan küçük, hızlı belleklerdir. Sistemin ana belleğinde bulunur. Ana bellek, aksine, tüm verileri tutar, ancak bu daha büyük belleğe erişim daha yavaştır. Sistem, sık erişilen verileri bir önbellekte tutarak büyük, yavaş bellek hızlıymış gibi görünür.

Örnek olarak, bellekten bir değer getirmek için açık bir yükleme talimatı veren bir programı ve yalnızca tek bir değere sahip basit bir sistemi düşünün.

İŞLEMCİ; CPU'nun küçük bir önbelleği (örneğin 64 KB) ve büyük bir ana belleği vardır.

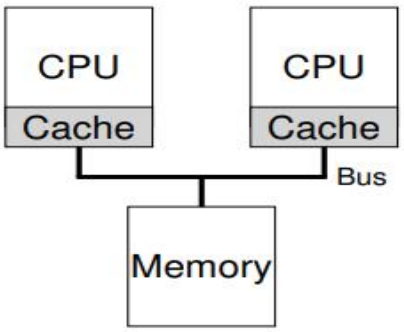


Figure 10.2: **Two CPUs With Caches Sharing Memory**

Bir program bu yükü ilk kez verdiğinde, veriler ana bellekte bulunur ve bu nedenle getirilmesi uzun zaman alır (belki onlarca nanosaniye, hatta yüzlerce). İşlemci, verilerin olabileceğini tahmin ederek yeniden kullanılabilir, yüklenen verilerin bir kopyasını CPU önbelleğine koyar. Program daha sonra bu aynı veri öğesini tekrar getirirse, CPU önce bunu kontrol eder.

Önbellek; orada bulduğu için veriler çok daha hızlı getirilir (Yalnızca birkaç nanosaniye) ve böylece program daha hızlı çalışır. Önbellekler bu nedenle yerellik kavramına dayalıdır; iki tür: zamansal yerellik ve mekânsal yerellik. Zamansal yerelliğin arkasındaki fikir, bir veri parçasına erişildiğinde, muhtemelen yakın gelecekte tekrar erişilen; değişkenleri ve hatta talimatları hayal edin kendilerine bir döngü içinde tekrar tekrar erişiliyor. Uzamsal yerelliğin ardındaki fikir, eğer bir program bir adresteki bir veri öğesine erişirse x, x yakınındaki veri öğelerine de erişme olasılığı yüksektir; burada bir program düşünün bir dizi üzerinden akış veya komutlardan sonra yürütülen komutlardır. Bu türlerin yerelliği birçok programda bulunduğundan, donanım sistemler önbelleğe hangi verilerin konulacağı konusunda iyi tahminler yapabilir ve böylece iyi çalışır.

Şimdi işin zor kısmına geçelim: Gördüğümüz gibi, tek bir paylaşılan ana belleğe sahip tek bir sistemde birden fazla işlemciniz olduğunda ne olur?

Şekil 10.2'de?

Birden fazla CPU ile önbelleğe almanın çok daha karmaşık olduğu ortaya çıktı. Örneğin, CPU 1'de çalışan bir programın okuduğunu hayal edin.

A adresinde bir veri öğesi (D değerine sahip); çünkü veriler ortada yok

CPU 1'deki önbellek, sistem onu ana bellekten alır ve

D değeri. Program daha sonra A adresindeki değeri değiştirir, yalnızca önbelleğini yeni D değeriyle günceller.

Verileri baştan sona yazmak ana belleğe geçiş yavaştır, bu nedenle sistem (genellikle) bunu daha sonra yapar. O zamanlar işletim sisteminin programı çalıştırmayı durdurmaya ve onu CPU'ya taşımaya karar verdiğini varsayalım

2. Program daha sonra A adresindeki değeri yeniden okur; böyle bir veri yok

CPU 2'nin önbelleği ve böylece sistem değeri ana bellekten alır ve doğru D değeri yerine eski D değerini alır.

Hata!

Bu genel soruna önbellek tutarlılığı sorunu denir ve birçok farklı inceliği açıklayan geniş bir araştırma literatur vardır.

[SHW11] sorunu çözmekle ilgili. Burada, hepsini atlayacağız. Nüans ve bazı önemli noktaları belirtin; bilgisayar mimarisi dersi almak (Veya üç) daha fazlasını öğrenmek için. Temel çözüm donanım tarafından sağlanır: donanım, bellek erişimlerini izleyerek temelde "doğru olanın" olmasını ve tek bir paylaşılan belleğin görünümünün korunmasını sağlayabilir. Tek yön bunu veri yolu tabanlı bir sistemde yapmak (yukarıda açıklandığı gibi) eski bir veri yolu gözetleme [G83] olarak bilinen teknik; her önbellek dikkat eder onları ana belleğe bağlayan veri yolunu gözlemleyerek bellek güncellemeleri. Bir CPU daha sonra önbelleğinde tuttuğu bir veri öğesi için bir güncelleme gördüğünde, değişikliği fark edecek ve kopyasını geçersiz kılacaktır (yani, onu kaldıracaktır). Yukarıda ima edildiği gibi geri yazma önbellekleri, bunu daha karmaşık hale getirir (Çünkü ana belleğe yazma işlemi daha sonra görünür değildir), ancak temel şemanın nasıl çalışabileceğini hayal edin.

Senkronizasyonu Unutmayın

Tutarlılık sağlamak için önbelleklerin tüm bu işi yaptığı göz önüne alındığında, programlar (veya işletim sisteminin kendisi) eriştiklerinde herhangi bir şey için endişelenmek zorunda mı? paylaşılan veriler? Yanıt, ne yazık ki, evet ve içinde belgelenmiştir. Eşzamanlılık konusunda bu kitabın ikinci parçasında büyük bir ayrıntı.

Burada ayrıntılara girmeyecek olsak da bazılarını çizeceğiz/inceleyeceğiz. Temel fikirler burada (eşzamanlılığa aşina olduğunuzu varsayarak). Paylaşılan veri öğelerine erişirken (ve özellikle güncellerken) veya CPU'lar arasındaki yapılar, karşılıklı dışlama ilkelleri (kilitler gibi) muhtemelen doğruluğu garanti etmek için kullanılmalıdır (kilitsiz veri yapıları oluşturmak gibi diğer yaklaşımlar karmaşıktır ve yalnızca ara sıra kullanılır; ayrıntılar için eşzamanlılık ile ilgili parçadaki kilitlenme ile ilgili bölüme bakın). İçin örneğin, birden çok ağ üzerinden erişilen paylaşılan bir kuyruğumuz olduğunu varsayalım. CPU'lar aynı anda. Kilitler olmadan, öğe ekleme veya çıkarma sıra aynı anda, temeldeki tutarlılık protokolleriyle bile beklendiği gibi çalışmaz; verileri atomik olarak güncellemek için kilitlere ihtiyaç vardır yapısı yeni haline getirildi.

Bunu daha somut hale getirmek için, kullanılan bu kod dizisini hayal edin.

Şekil 10.3'te gördüğümüz gibi, paylaşılan bir bağlantılı listeden bir öğeyi kaldırmak için.

İki CPU'daki iş parçacıklarının bu rutine aynı anda girdiğini hayal edin. Eğer iş parçacığı 1 ilk satırı yürütür, şu anki head değerine sahip olacaktır. Tmp değişkeninde saklanır; Eğer Thread 2 daha sonra ilk satırı da yürütürse, ayrıca kendi özel tmp'sinde depolanan aynı kafa değerine sahip olacaktır. Değişken (tmp yığında tahsis edilir ve böylece her iş parçacığının sahip olacağı onun için kendi özel deposu). Böylece, her iş parçacığını çıkarmak yerine listenin başından bir öğe varsa, her iş parçacığı onu kaldırmaya çalışır.

```
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 }
```

Figure 10.3: **Simple List Delete Code**

Şekil 10.3: Basit Liste Silme Kodu aynı kafa elemanı, her türlü soruna yol açar (örneğin, 4. satırda ana öğeden iki kat serbest kalmanın yanı sıra potansiyel olarak geri dönme aynı veri değeri iki kez).

Çözüm, elbette, bu tür rutinleri kilitleme yoluyla düzeltmektir. Bu durumda, basit bir muteks tahsis etmek (örneğin, pthread mutex tmp;) ve ardından rutinin başına bir kilit(&m) ekleyerek ve sonunda bir kilit açma(&m) sorunu çözerek kodun istenildiği gibi yürütülecektir. Ne yazık ki, göreceğimiz gibi, böyle bir yaklaşım özellikle performans açısından sorunsuz değil. Özellikle, CPU sayısı arttıkça, senkronize edilmiş paylaşılan bir veriye erişim yapı oldukça yavaşlar.

Son Bir Sorun: Önbellek Benzeşimi

Çok işlemcili bir önbellek planlayıcı oluştururken ortaya çıkan son bir sorun, önbellek yakınlığı olarak bilinir. Bu kavram basittir: bir işlem, bir bilgisayarda çalıştırıldığında belirli bir CPU, önbelleklerde (ve TLB'lerde) oldukça fazla durum oluşturur.

Süreç bir dahaki sefere çalıştığında, onu çalıştırmak genellikle avantajlıdır. Durumunun bir kısmı zaten mevcutsa daha hızlı çalışacağı için aynı CPU bu CPU'daki önbellekler. Bunun yerine, farklı bir CPU'da bir işlem çalıştırılırsa her seferinde, sürecin performansı daha kötü olacaktır. Böylece, birçok işlemcili zamanlayıcı, önbellek yakınlığını dikkate almalıdır.

10.4 Tek Sıralı Programlama

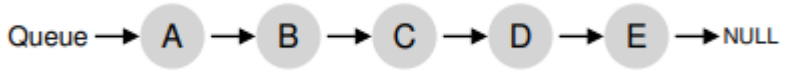
Bu arka plan hazır olduğunda, şimdi çok işlemcili bir sistem için bir programlayıcının nasıl oluşturulacağını tartışacağız. En temel yaklaşım basitçe tümünü koyarak, tek işlemci zamanlaması için temel çerçeveyi yeniden kullanın tek bir kuyruğa programlanması gereken işler; buna tek kuyruklu çok işlemcili çizelgeleme veya kısaca SQMS diyoruz. Bu yaklaşım basitlik avantajına sahiptir; almak için fazla çalışma gerektirmez. Bir sonraki çalıştırılacak en iyi işi seçen ve üzerinde çalışacak şekilde uyarlayan mevcut politika birden fazla CPU (varsa çalıştırılacak en iyi iki işi seçebileceği yer) örneğin iki CPU'dur).

Bununla birlikte, SQMS'nin bariz eksiklikleri vardır. İlk sorun bir eksiklik ölçeklenebilirlik. Zamanlayıcının birden çok CPU'da düzgün çalışmasını sağlamak için, geliştiriciler, koda bir tür kilitleme eklemiş olacaklar: Yukarıda tarif edilen kilitler, SQMS kodu tekliye eriştiğinde sıra (örneğin, çalıştırılacak bir sonraki işi bulmak için), uygun sonuç ortaya çıkar. Kilitler maalesef performansı büyük ölçüde azaltabilir, özellikle de sistemlerdeki CPU sayısı artıyor [A91]. Bunun için çekişme olarak tek bir kilit artar, sistem kilitte daha fazla zaman harcar sistemin yapması gereken işi yapmak için ek yük ve daha az zaman (not: bir gün bunun gerçek bir ölçümünü buraya dahil etmek harika olurdu).

SQMS ile ilgili ikinci ana sorun, önbellek yakınlığıdır. Örneğin,

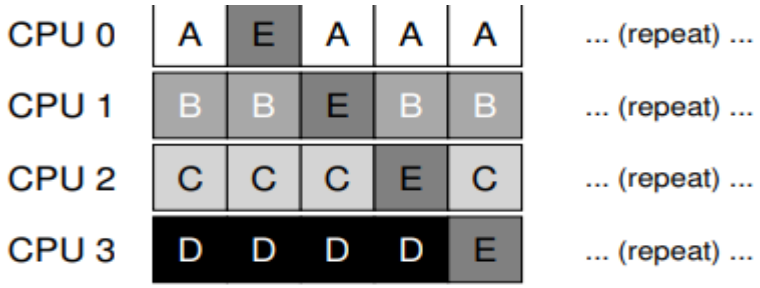
Çalıştırılacak beş işimiz (A, B, C, D, E) ve dört işlemcimiz olduğunu varsayalım.

Böylece planlama kuyruğumuz şöyle görünür:



Her bir CPU, evrensel olarak paylaşılan sıradan çalıştırılacak bir sonraki işi seçtiğinden, her iş CPU'dan CPU'ya sıçrayarak son bulur. Önbellek yakınlığı açısından mantıklı olanın tam tersini yapmak. Bu sorunun üstesinden gelmek için, çoğu SQMS zamanlayıcısı bir çeşit

içerir. Sürecin devam etmesini daha olası hale getirmeye çalışmak için yakınlık mekanizması mümkünse aynı CPU üzerinde çalışmak için. Spesifik olarak, biri bazı işler için yakınlık sağlayabilir, ancak yükü dengelemek için diğerlerini hareket ettirebilir. Örneğin, aynı beş işin aşağıdaki gibi 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 yakınlık korunur. Daha sonra, bir sonraki sefer farklı bir işe geçmeye karar verebilirsiniz. Zaman geçtikçe, böylece bir tür yakınlık adaletine de ulaşılır. Bununla birlikte, böyle bir planın uygulanması karmaşık olabilir. Böylece, SQMS yaklaşımının güçlü ve zayıf yönleri olduğunu görebiliriz. Mevcut bir tek CPU verildiğinde uygulanması kolaydır tanım gereği yalnızca tek bir kuyruğa sahip olan zamanlayıcı. Ancak, yapar iyi ölçeklenemez (uyum ek yükleri nedeniyle) ve kolayca önbellek yakınlığını koru.

Çok Kuyruklu Zamanlama

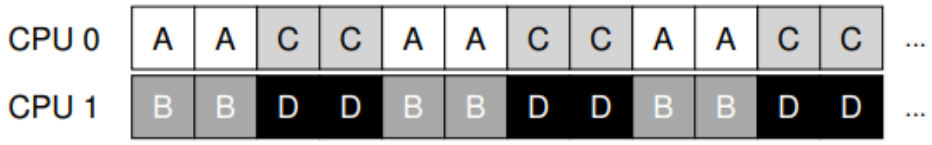
Tek kuyruklu programlayıcıların neden olduğu sorunlar nedeniyle, bazı sistemler, örneğin CPU başına bir tane olmak üzere birden çok sırayı tercih eder. Biz buna yaklaşım diyoruz çok kuyruklu çok işlemcili zamanlama (veya MQMS). MQMS'de, temel çizelgeleme çerçevemiz birden fazla çizelgeleme kuyruğundan 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, tam olarak bir çizelgeleme üzerine yerleştirilir. Kuyruğa, bazı buluşsal yöntemlere göre (örneğin, rastgele veya

diğerlerinden daha az iş). Daha sonra esasen bağımsız olarak planlanır, böylece bilgi paylaşımı ve uyum problemlerinden kaçınılır.

Tek kuyruklu yaklaşımda bulunur.

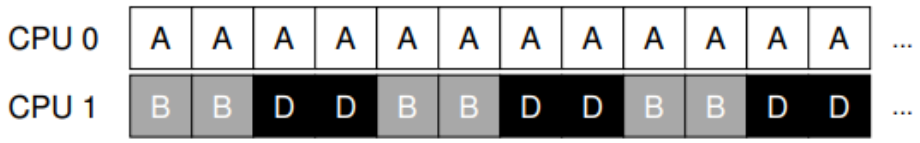
Örneğin, sadece iki CPU'nun olduğu bir sistemimiz olduğunu varsayalım. (CPU 0 ve CPU 1 olarak etiketlenmiştir) ve bazı işler sisteme girer: Örneğin A, B, C ve D. Her CPU'nun bir zamanlama kuyruğu olduğu göz önüne alındığında şimdi işletim sisteminin her işi hangi kuyruğa yerleştireceğine karar vermesi gerekiyor. Yapabilir bunun gibi bir şey:

Kuyruk zamanlama ilkesine bağlı olarak, artık her CPU'nun neyin çalıştırılacağına karar verirken seçilecek işler sırayla, sistem şuna benzeyen bir program üretebilir:

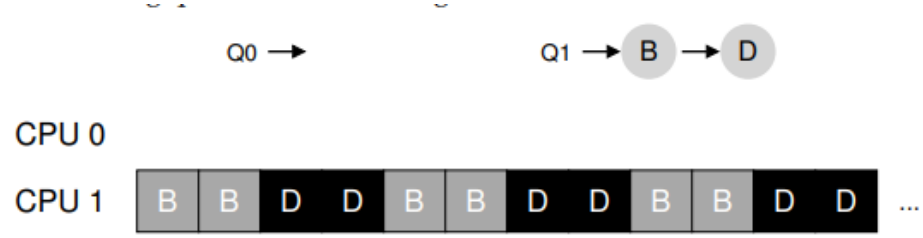


MQMS, doğası gereği daha ölçeklenebilir olması gerektiği için SQMS'nin belirgin bir avantajına sahiptir. CPU sayısı arttıkça sıra sayısı da artar ve bu nedenle kilit ve önbellek çekişmesi bir sorun haline gelmemelidir. Merkezi sorun. Ek olarak, MQMS özünde önbellek benzerliği sağlar; işler aynı CPU'da kalır ve böylece önbelleğe alınanları yeniden kullanma avantajından yararlanır içindekiler. Ancak, dikkat ettiyseniz, yeni bir özelliğimiz olduğunu görebilirsiniz. Çoklu kuyruğa dayalı yaklaşımda temel olan problem: yük dengesizlik Yukarıdakiyle aynı kuruluma sahip olduğumuzu varsayalım (dört iş, iki CPU), ancak sonra işlerden biri (C diyelim) biter. Şimdi elimizde aşağıdaki zamanlama kuyrukları:

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



Bu diyagramdan da görebileceğiniz gibi, A, B'nin iki katı CPU alır ve D, istenen sonuç değil. Daha da kötüsü, her ikisinin de olduğunu düşünelim. A ve C tamamlanır ve sistemde sadece B ve D işleri kalır. Zamanlama kuyruklar şöyle görünecek:



Sonuç olarak, CPU 0 boşta kalacak! Bu nedenle CPU kullanım zaman çizelgemiz üzücü görünüyor: Öyleyse, zayıf birçok kuyruklu çok işlemcili zamanlayıcı ne yapmalıdır? Soru sormayı nasıl durdurabiliriz?

CRUX: YÜK DENGESİZLİĞİYLE NASIL BAŞA ÇIKILIR

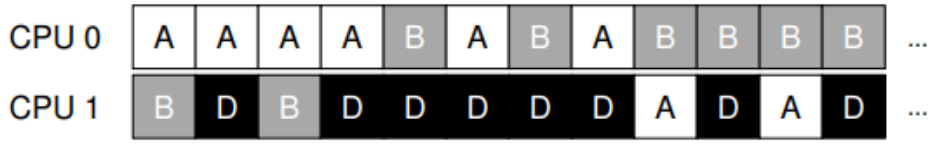
Çok kuyruklu çok işlemcili bir programlayıcı, istenen programlama hedeflerine daha iyi ulaşmak için yük dengesizliğini nasıl ele almalıdır?

Bu sorunun bariz cevabı, işleri hareket ettirmektir, bir teknik buna (bir kez daha) göç diyoruz. Bir işten bir işi taşıyarak CPU'dan diğerine, gerçek yük dengesi elde edilebilir. Biraz netlik katmak için birkaç örneğe bakalım. Bir kez daha biz bir CPU'nun boşta olduğu ve diğerinin bazı işleri olduğu bir duruma sahip olun. Bu durumda, istenen geçişin anlaşılması kolaydır:

İşletim sistemi B veya D'den birini CPU 0'a taşımanız yeterlidir. Bu tek iş geçişinin sonucu, yükün eşit şekilde dengelenmesidir ve herkes mutludur. A'nın kaldığı önceki örneğimizde daha zor bir durum ortaya çıkıyor. Sadece CPU 0'da ve B ve D, CPU 1'de değişiyordu: Bu durumda, tek bir geçiş sorunu çözmez.

Cevap, ne yazık ki, sürekli göçtür. Bir veya daha fazla işten Olası bir çözüm, iş değiştirmeye devam etmektir. Aşağıdaki zaman çizelgesinde görüyoruz. Şekilde, ilk A, CPU 0'da yalnızdır ve B ve D, CPU 1'de dönüşümlü olarak çalışır. Birkaç zaman diliminden sonra, B şuraya taşınır: CPU 0'da A ile rekabet ederken, D CPU'da tek başına birkaç zaman diliminin tadını çıkarır

1. Böylece yük dengelenir:



Tabii ki, başka birçok olası göç modeli mevcuttur. Ama şimdi için işin zor kısmı: sistem böyle bir taşımayı yasalaştırmaya nasıl karar vermeli?

Temel yaklaşımlardan biri, iş çalma olarak bilinen bir tekniği kullanmaktır.

[FLR98]. İş çalma yaklaşımıyla, düşük olan bir (kaynak) kuyruk 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 "çalacaktır". Elbette böyle bir yaklaşımda doğal bir gerilim vardır. Eğer bakarsan diğer kuyruklarda çok sık dolaşırsanız, yüksek ek yükten mustarip olursunuz ve Ölçeklendirmede sorun yaşıyorsanız, ilk etapta çoklu sıra zamanlamasını uygulamanın tüm

amacı buydu! Öte yandan, eğer diğer sıralara çok sık bakmayın, acı çekme tehlikesiyle karşı karşıyasınız. Şiddetli yük dengeleri. Yaygın olduğu gibi doğru eşiği bulmak kalır sistem politikası tasarımında kara bir sanat.

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)2 Meehean'ın tezine bakın söz konusu zamanlayıcıların güçlü ve zayıf yönlerine mükemmel bir genel bakış

[M11]; burada sadece birkaç temel bilgiyi özetliyoruz.

Hem O(1) hem de CFS birden çok sıra kullanırken, BFS tek bir kuyruk kullanır. Sıra, her iki yaklaşımın da başarılı olabileceğini gösteriyor. Tabii ki orada bu programlayıcıları ayıran diğer birçok ayrıntıdır. Örneğin, O(1) planlayıcı, önceliğe dayalı bir programlayıcıdır (daha önce tartışılan MLFQ'ya benzer), zaman içinde bir sürecin önceliğini değiştirir ve ardından çeşitli programlama hedeflerini karşılamak için en yüksek önceliğe sahip olanları çizelgeler; etkileşim özel bir odak noktasıdır. Buna karşılık CFS, deterministiktir. Orantılı paylaşım yaklaşımı (daha çok Stride zamanlaması gibi, tartışıldığı gibi) daha erken). Üçü arasındaki tek tek kuyruklu yaklaşım olan BFS, aynı zamanda orantılı paylaşım, ancak şu şekilde bilinen daha karmaşık bir şemaya dayalıdır: Önce En Erken Uygun Sanal Bitiş Tarihi (EEVDF) [SA96]. Hakkında devamını okuyup bu modern algoritmalar kendi başınıza; anlayabilmelisin

Şimdi nasıl çalışıyorlar!

10.7 Özet

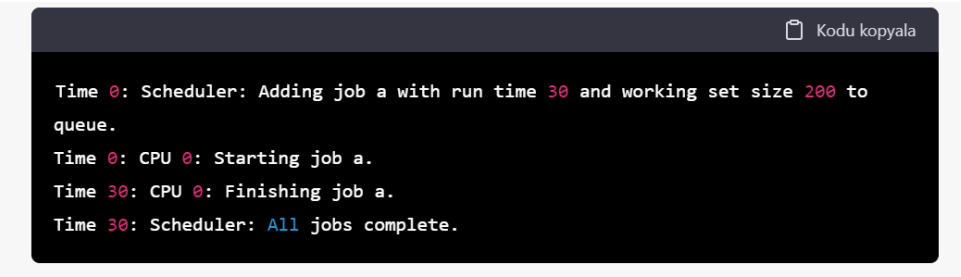
Çok işlemcili zamanlamaya yönelik çeşitli yaklaşımlar gördük. Bu tek kuyruklu yaklaşım (SQMS), yükü iyi bir şekilde oluşturmak ve dengelemek için oldukça basittir, ancak doğası gereği birçok işlemciye ve önbellek yakınlığına ölçeklendirmede zorluk yaşar. Çoklu kuyruk yaklaşımı (MQMS) ölçekleri daha iyidir ve önbellek yakınlığını iyi idare eder, ancak yük dengesizliği ile ilgili sorunları vardır ve daha karmaşıktır. Hangi yaklaşımı benimserseniz seçin, hiçbir basit cevap: genel amaçlı bir zamanlayıcı oluşturmak göz korkutucu olmaya devam ediyor küçük kod değişiklikleri büyük davranışsal farklılıklara yol açabileceğinden görev.

ÖDEV:

1. 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 iş çalıştıracaktır. Bu işi (burada iş 'a' olarak adlandırılır) simüle edilmiş bir CPU'da aşağıdaki gibi çalıştırın: -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.

Simülasyonu komutuyla çalıştırırsanız, **`./multi.py -n 1 -L a:30:200 -c -t`**'a' işi 30 çalışma süresine sahip olduğundan ve tek bir CPU üzerinde çalıştırıldığından, tamamlanması 30 zaman birimi alacaktır. Bayrak **`-c`**, simülasyonun nihai sonucunu gösterecek ve **`-t`**bayrak, işin adım adım izini ve nasıl programlandığını gösterecektir.

Bu simülasyonu çalıştırırken görebileceğiniz çıktıya bir örnek:



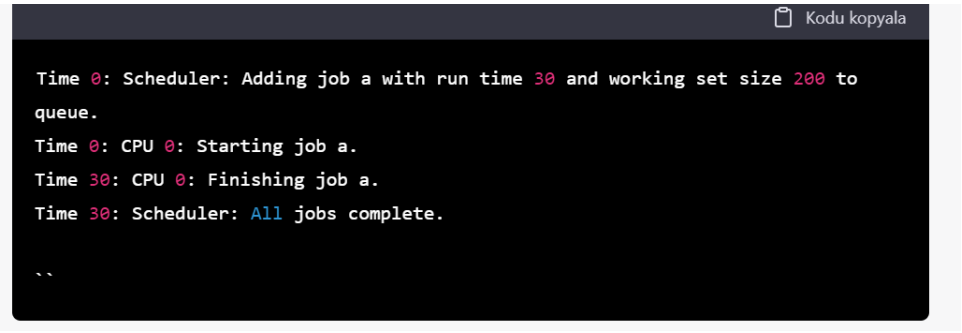
Çıktı, işin 0 zamanında kuyruğa eklendiğini, 0 zamanında CPU 0'da başladığını ve 30 zamanında bittiğini gösterir. Simülasyon daha sonra 30 zamanında sona erer çünkü tüm işler tamamlanmıştır.

2. Şimdi işin çalışma setini (boyut=200) önbelleğe (varsayılan olarak boyut=100'dür) sığdırmak için önbellek boyutunu artırın; örneğin, ./multi.py -n 1 -L a:30:200 -M 300 dosyasını çalıştırın. 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,**-M**bayrak**-r**bayrak**-r**bayrak

İle birlikte**-c**bayrak

İşte bir



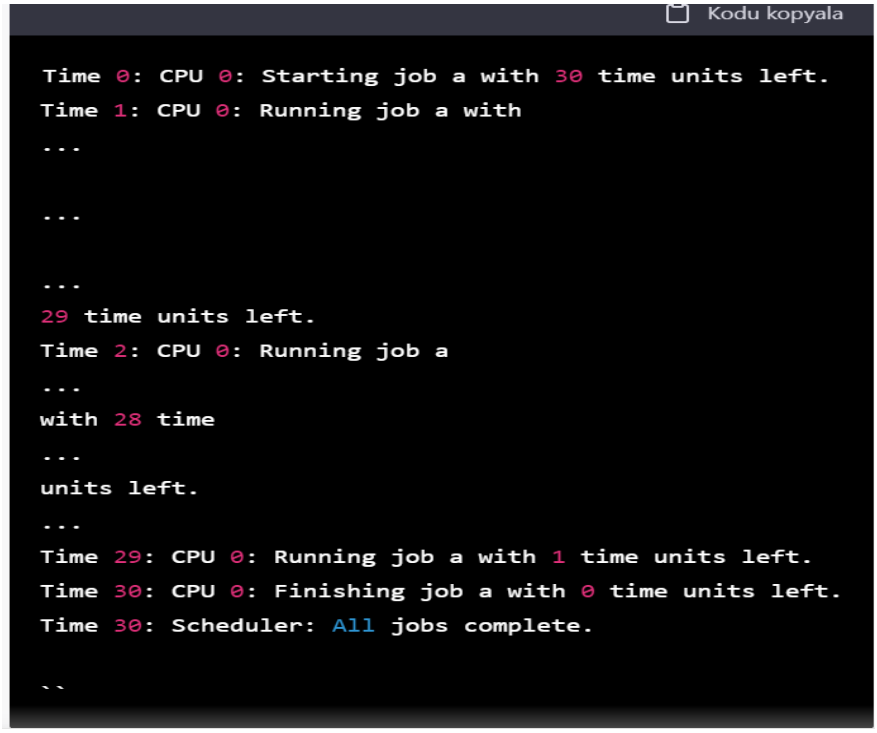
bu

Eğer**-r**bayrak**./multi.py -n 1 -L a:30:200 -M 300 -r 2 -c**,

3. multi.py ile ilgili harika bir şey, 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?

./multi.py -n 1 -L a:30:200 -M 300 -T,**-T**bayrak

İle birlikte



bu

Eğer**-r**bayrak**./multi.py -n 1 -L a:30:200 -M 300 -r 2 -T**,

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?

./multi.py -n 1 -L a:30:200 -M 300 -C,**-C**bayrak

İle birlikte

```
Time 0: Scheduler: Adding job a with run time 30 and working set size 200 to
queue.
Time 0: CPU 0: Starting job a with caches [''].
Time 1: CPU 0: Running job a with caches [''].
Time 2: CPU 0: Running job a with caches [''].
Time 3: CPU 0: Running job a with caches [''].
Time 4: CPU 0: Running job a with caches [''].
Time 5: CPU 0: Running job a with caches ['w'].
...
Time 30: CPU 0: Finishing job a with caches ['w'].
Time 30: Scheduler: All jobs complete.

``
```

bu**''**),**'w'**).

Eğer**-w**bayrak**./multi.py -n 1 -L a:30:200 -M 300 -C -w 10**,**./multi.py -n 1 -L a:30:200 -M 300 -C -w 1**,

5. Bu noktada, simülatörün tek bir CPU'da 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 (örn. ./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 yazın) Nasıl olacağını tahmin edebilir misiniz? döngüsel bir merkezi planlayıcı göz önüne alındığında, bu ne kadar sürer? 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. Ne farkettin?

./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -c,

İle birlikte

Sen**-t**bayrak**-C**bayrak

Buraya

```
Kodu kopyala

Time 0: Scheduler: Adding job a with run time 100 and working set size 100 to
queue.
Time 0: Scheduler: Adding job b with run time 100 and working set size 50 to
queue.
Time 0: Scheduler: Adding job c with run time 100 and working set size 50 to
queue.
Time 0: CPU 0: Starting job a.
Time 0: CPU 1: Starting job b.
Time 1: CPU 0: Running job a with 99 time units left.
Time 1: CPU 1: Running job b with 99 time units left.
Time 2: CPU 0: Running job a with 98 time units left.
Time 2: CPU 1: Running job b with 98 time units left.
...
Time 148: CPU 0: Running job a with 2 time units left.
Time 148: CPU 1: Running job b with 2 time units left.
Time 149: CPU 0: Running job a with 1 time units left.
Time 149: CPU 1: Running job b with 1 time units left.
Time 150: CPU 0: Finishing job a with 0 time units left.
Time 150: CPU 1: Finishing job b with 0 time units left.
Time 150: Scheduler: All jobs complete.

``
```

bu

Sen-**C**bayrak

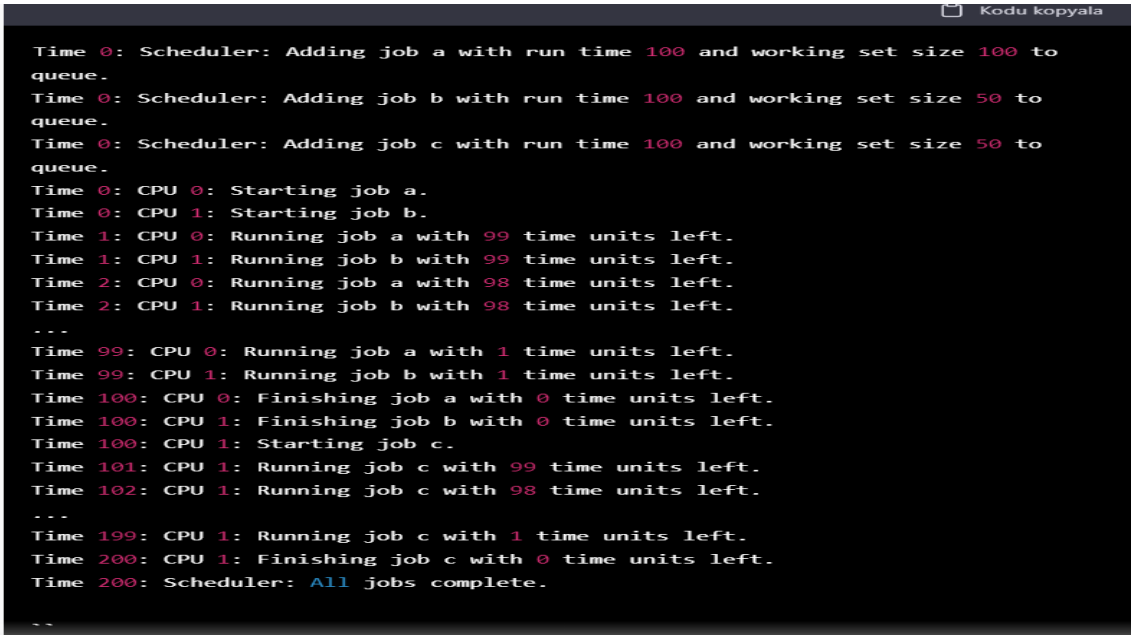
6. Şimdi, bölümde açıklandığı gibi, önbellek yakınlığını incelemek için bazı açık kontroller uygulayacağız. Bunu yapmak için -A bayrağına ihtiyacınız olacak. Bu bayrak, zamanlayıcının belirli bir işi yerleştirebileceği CPU'ları sınırlamak için kullanılabilir. Bu durumda, 'a'yı CPU 0 ile sınırlandırırken, 'b' ve 'c' işlerini CPU 1'e yerleştirmek için kullanalım. Bu sihir ./multi.py -n 2 -L a:100 yazarak gerçekleştirilir :100,b:100:50, c:100:50 -A a:0,b:1,c:1 ; 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 çalışır?

./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -A a:0,b:1,c:1 -c,**-A**bayrak

İle birlikte

Sen**-t**ve**-C**bayraklar

Buraya



bu**-A**bayrak

Diğer**./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -A a:1,b:0,c:0**,

7. Çok işlemcili önbelleğe almanın ilginç bir yönü, tek bir işlemci üzerinde çalışan işlerle karşılaştırıldığında birden çok CPU (ve bunların önbellekleri) kullanıldığında işleri 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.

./multi.py -n 1 -L a:100:100,b:100:100,c:100:100 -M 50 -c,

Eğer**./multi.py -n 2 -L a:100:100,b:100:100,c:100:100 -M 50 -c**,

Eğer**./multi.py -n 3 -L a:100:100,b:100:100,c:100:100 -M 50 -c**,

Şimdi**./multi.py -n 1 -L a:100:100,b:100:100,c:100:100 -M 100 -c**,

Eğer**./multi.py -n 2 -L a:100:100,b:100:100,c:100:100 -M 100 -c**,

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?

./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -p -c,**-p**bayrak

Tarafından

Sen**-P**bayrak**./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -p -P 5 -c**,

Sen

Olarak`-t`ve`-C`bayraklar

Etraflı

9. Son olarak, rastgele iş yükleri oluşturmaktan çekinmeyin ve bunların 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 harika bir şey. İyi şanlar!

Rastgele iş yükleri oluşturmak ve bunların farklı işlemci yapılandırmaları, önbellek boyutları ve zamanlama seçenekleri üzerindeki performanslarını tahmin etmeye çalışmak iyi bir fikirdir. Bu, farklı iş yüklerinin performans özelliklerini ve farklı donanım ve yazılım yapılandırmalarından nasıl etkilendiklerini öğrenmenin yararlı bir yolu olabilir.

İş yükleri oluşturulurken, gerçekleştirilen işin türünü ve işlemci sayısı, önbellek boyutu ve zamanlama seçenekleri gibi faktörlerden nasıl etkilenebileceğini dikkate almak önemlidir. Örneğin, önbellekte depolanan verilere büyük ölçüde bağımlı olan bir iş yükü, önbelleği daha küçük olan bir sistemle karşılaştırıldığında daha büyük bir önbelleğe sahip bir sistemde farklı performans gösterebilir. Benzer şekilde, yüksek oranda paralelleştirilebilen bir iş yükü, daha fazla sayıda işlemciye sahip bir sistemde çalışmaktan fayda sağlayabilirken, daha sıralı olan bir iş yükü, performansta o kadar fazla iyileşme görmeyebilir.

Performans analizi araçları, simülasyon araçları ve yük testi araçları dahil olmak üzere iş yüklerini oluşturmak ve analiz etmek için birçok araç ve teknik mevcuttur. Bu araçları kullanarak ve farklı donanım ve yazılım yapılandırmalarının performansı nasıl etkileyebileceğine

ilişkin bilginizi uygulayarak, farklı sistemlerdeki farklı iş yüklerinin performansını tahmin etme konusunda uzmanlaşabilirsiniz.