10

Çok İşlemcili Zamanlama (İleri Düzey)

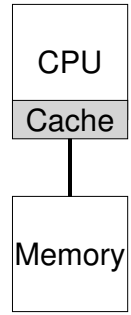
Bu bölüm, **çok işlemcili zamanlamanın (Multiprocessor Scheduling)** temellerini tanıtacaktır. Bu konu nispeten ileri düzeyde olduğundan, eşzamanlılık konusunu biraz ayrıntılı olarak (yani kitabın ikinci büyük "kolay parçası") çalıştıktan sonra ele almak en iyisi olabilir.

Yıllarca yalnızca bilgi işlem spektrumunun en üst noktasında var olduktan sonra, **çok işlemcili** sistemler 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 (multicore)** işlemcinin yükselişi, bu çoğalmanın kaynağıdır; bilgisayar mimarları çok fazla güç (çok) 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 (parallel)** çalışacak şekilde, belki de **çekirdekleri (threads)** kullanarak yeniden yazmanız gerekecek (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.

Bölüm: İleri Bölümler

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



Şekil 10.1: **Önbellekli Tek CPU**

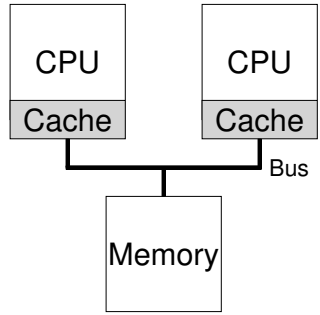
Uygulamaların ötesinde, işletim sistemi için ortaya çıkan yeni bir sorun (şaşırtıcı olmayan bir şekilde!) **çok işlemcili zamanlama** sorunudur. Şimdiye kadar tek işlemcili çizelgelemenin ardındaki bir dizi ilkeyi tartıştık; bu fikirleri birden fazla CPU üzerinde çalışacak şekilde nasıl genişletebiliriz?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

Kritik Nokta: 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 Arkaplan: Çok İşlemcili Mimari

Çok işlemcili zamanlamayı çevreleyen yeni sorunları anlamak için, tek CPU donanımı ile çoklu CPU donanımı arasındaki yeni ve temel farkı anlamamız gerekir. Bu fark, donanım **önbelleklerinin (caches)** kullanımı (ör. Şekil 10.1) ve verilerin birden fazla işlemci arasında tam olarak nasıl paylaşıldığı etrafında toplanıyor. Şimdi bu konuyu daha üst düzeyde tartışıyoruz. Ayrıntılar başka bir yerde [CSG99], özellikle üst düzey veya belki lisansüstü 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, (genel olarak) sistemin ana belleğinde bulunan *popüler* verilerin kopyalarını tutan küçük, hızlı belleklerdir. Ana bellek, aksine, *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 Önbellekli İki CPU**

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 nanosaniye, hatta yüzlerce). İşlemci, verilerin yeniden kullanılabileceğini tahmin ederek, yüklenen verilerin bir kopyasını CPU önbelleğine koyar. Program daha sonra bu aynı veri öğesini tekrar getirirse, CPU önce önbellekte onu kontrol eder; onu orada bulursa, veriler çok daha hızlı getirilir (örneğin, yalnızca birkaç nanosaniye) ve böylece program daha hızlı çalışır.

Bu nedenle önbellekler, iki tür olan **yerellik (locality)** kavramına dayanır: **zamansal yerellik (temporal locality)** ve **uzamsal yerellik (spatial locality)**. Zamansal yerelliğin ardındaki fikir, bir veri parçasına erişildiğinde, ona yakın gelecekte yeniden erişilmesinin muhtemel olmasıdır; değişkenlere ve hatta yönergelere bir döngü içinde tekrar tekrar erişildiğini hayal edin. Uzamsal 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 türlerin yerelliği 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 paylaşılan ana belleğe sahip tek bir sistemde birden çok işlemciye sahip olduğunuzda ne olur? Birden fazla CPU ile önbelleğe almanın çok daha karmaşık olduğu ortaya çıktı. 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 alır, "D"değerini de çeker.

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 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 değer D' yerine eski D değerini alır. Hata!

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) fazla). 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

Tutarlılık sağlamak için önbelleklerin tüm bu işi yaptığı göz önüne alındığında, programların (veya işletim sisteminin kendisinin) paylaşılan verilere eriştiklerinde herhangi bir şey için endişelenmeleri gerekir mi? Yanıt ne yazık ki evettir ve bu kitabın eş zamanlılık 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 muhtemelen karşılıklı dışlama ilkelleri (kilitler gibi) 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). 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 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 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**

Eğer Thread 1 ilk satırı yürütürse, tmp değişkeninde kayıtlı head'in geçerli 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 böylece her iş parçacığının bunun için kendi özel deposu olacaktır). Bu nedenle, 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. Satırdaki baş öğeden iki kez kurtulma girişimi gibi) potansiyel olarak aynı veri değerini iki kez döndürme olarak).

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 çözerek kodun çalışmasını sağlayacaktır. istediğiniz gibi. Ne yazık ki, 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 Benzeşimi

Önbellek benzerliği (cache affinity) [TTG95] olarak bilinen çok işlemcili bir önbellek planlayıcının oluşturulmasında son bir sorun ortaya çıkar. Bu kavram basittir: bir işlem, belirli bir CPU üzerinde çalıştırıldığında, CPU'nun önbelleklerinde (ve TLB'lerde) oldukça fazla durum oluşturur. İşlemin bir sonraki çalıştırılışında, 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 üzerinde bir işlem çalıştırılırsa, işlemin performansı daha kötü olacaktır, çünkü her çalıştırıldığında durumu yeniden yüklemek zorunda kalacaktır (dikkat edin, farklı bir CPU üzerinde doğru şekilde çalışacaktır. donanımın önbellek tutarlılık protokolleri). Bu nedenle, çok işlemcili bir programlayıcı, zamanlama kararlarını verirken önbellek yakınlığını dikkate almalı, belki de mümkünse bir işlemi aynı CPU'da tutmayı tercih etmelidir.

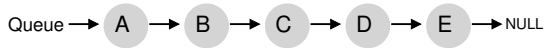
10.4 Tek Kuyruklu Zamanlama

Bu arka plan hazır olduğunda, çok işlemcili bir sistem için bir planlayıcının nasıl oluşturulacağını tartışacağız. En temel yaklaşım, zamanlanması gereken tüm işleri tek bir kuyruğa koyarak tek işlemcili zamanlama için temel çerçeveyi basitçe yeniden kullanmaktır; buna **tek kuyruklu çok işlemcili programlama (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) .

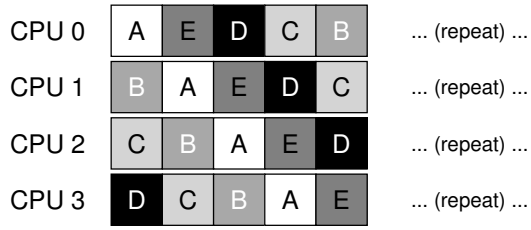
Bununla birlikte, SQMS'nin bariz eksiklikleri vardır. İlk sorun, **ölçeklenebilirlik (scalability)** eksikliğidir. Zamanlayıcının birden fazla CPU üzerinde doğru şekilde çalışmasını sağlamak için, geliştiriciler yukarıda açıklandığı gibi koda bir tür kilitleme eklemiş olacaklardır. Kilitler, SQMS kodu tek kuyruğa eriştiğinde (örneğin çalıştırılacak bir sonraki işi bulmak için) uygun 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 fakat 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:



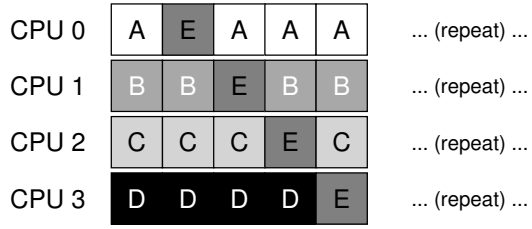
Zamanla, her işin bir zaman diliminde çalıştığını ve ardından başka bir işin seçildiğini varsayarsak, burada CPU'lar arasında olası bir iş çizelgesi var:



Her bir CPU, küresel olarak paylaşılan sıradan çalıştırılacak bir sonraki işi seçtiğinden, her iş CPU'dan CPU'ya sıçrayarak sona erer ve böylece önbellek benzeşimi açısından mantıklı olanın tam tersini yapar.

Bu sorunun üstesinden gelmek için, çoğu SQMS zamanlayıcısı, mümkünse işlemin aynı CPU üzerinde çalışmaya dwevam etmesini daha olası hale getirmeye çalışmak için bir tür yakınlık mekanizması içerir.

Spesifik olarak, kişi bazı işler için yakınlık sağlayabilir, ancak yükü dengelemek için diğerlerini hareket ettirebilir. Örneğin, aynı beş işin aşağıdaki şekilde planlandığını hayal edin:



Bu düzenlemeyle, A-D arasındaki işler işlemcilere taşınmaz ve yalnızca E işi CPU 'dan CPU' ya **geçiş** sağlar. Bu nedenle çoğu iş için benzerliği korumaz. Daha sonra farklı bir işi bir sonraki seferde geçirmeye karar verebilirsiniz, böylece bir tür benzeşme de elde edilebilir. Ancak böyle bir planın uygulanması karmaşık olabilir.

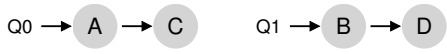
Böylece, SQMS yaklaşımının güçlü ve zayıf yönleri olduğunu görebiliriz. Tanımı gereği yalnızca tek bir kuyruğa sahip olan mevcut bir tek CPU planlayıcı verildiğinde uygulanması kolaydır. Ancak, (eşzamanlama ek yükleri nedeniyle) iyi ölçeklenemez ve önbellek yakınlığını hemen korumaz.

10.5 Çok Kuyruklu Zamanlama

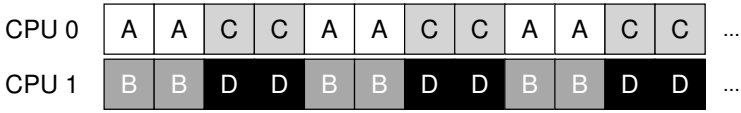
Tek kuyruklardaki zamanlayıcılarda ortaya çıkan sorunlar nedeniyle, bazı sistemler birden çok kuyruk (örneğin, CPU başına bir tane) tercih eder. Bu yaklaşıma **çok kuyruklu çoklu işlemci zamanlaması (multi-queue multiprocessor scheduling)** (or**MQMS**) adı verilir.

MQMS'de, temel çizelgeleme çerçevemiz birden çok çizelgeleme kuyruğundan oluşur. Elbette her algoritma kullanılabilir olsa da, her bir kuyruk büyük olasılıkla belirli bir zamanlama disiplinini takip edecektir, örneğin hepsini bir kez deneme gibi. 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, sadece iki CPU'nun (CPU 0 ve CPU 1 olarak etiketlenmiş) bir sistem olduğunu ve bazı işlerin sisteme girdiğini varsayalım: Örneğin A, B, C ve D. Her CPU'nun bir zamanlama kuyruğu olduğu düşünüldüğünde, işletim sistemi hangi kuyruğa hangi işi yerleştireceğine karar vermelidir. Bu şekilde yapabilir:



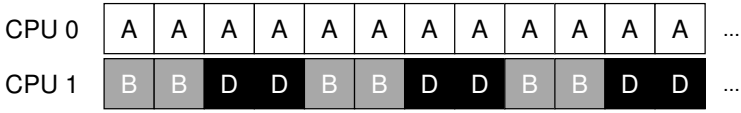
Kuyruk zamanlama ilkesine bağlı olarak, her CPU şimdi ne çalıştırılacağına karar verirken iki işten birini seçebilir. Örneğin, **round robin** ile sistem bu şekilde bir zamanlama üretebilir:



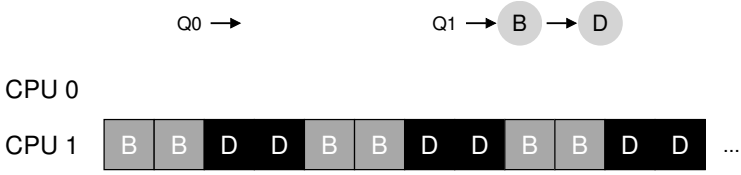
MQMS, daha fazla ölçeklenebilir olması gereken SQMS ' nin farklı bir avantajına sahiptir. CPU sayısı büyüdükçe, kuyrukların sayısı da çok fazla olur ve bu nedenle kilit ve önbellek çekişmesi merkezi bir sorun haline gelmemelidir. Buna ek olarak, MQMS 'de önbellek benzerliği sağlar; işler aynı CPU' da kalır ve bu nedenle, önbelleğe alınan içeriği yeniden kullanma avantajından yararlanır.

Ama, dikkat etmişseniz, yeni bir problemimiz olduğunu görebilirsiniz: **yük dengesizliği (load imbalance)**. Önceki kurulumu (dört iş, iki CPU) aynı şekilde varsayalım, ancak işlerden birinin (örneğin C) bitmesiyle birlikte aşağıdaki zamanlama kuyruklarına sahip oluruz:

Sistemdeki her kuyruğu round robin politikamızla çalıştırdığımızda, bu sonuç zamanlamasına bakacağız:



Bu şemadan 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 sonuçta ortaya çıkan zaman çizelgesi şöyle görünecektir:



Ne kadar korkunç-CPU 0 boşta! *(burada dramatik ve kötü müzik ekleyin)*
Ve böylece, CPU kullanımı zaman çizelgesi oldukça üzücü görünüyor.

Peki zavallı bir çoklu kuyruk çoklu işlemci zamanlayıcısı ne yapmalı? Yük dengesizliği ve Decepticons'ların kötü güçlerini nasıl yenebiliriz? Bu aksi takdirde harika kitabımızla ilgisi az olan soruları nasıl durdurabiliriz?

YÜK DENGESİ İLE NASIL BAŞA ÇIKILIR

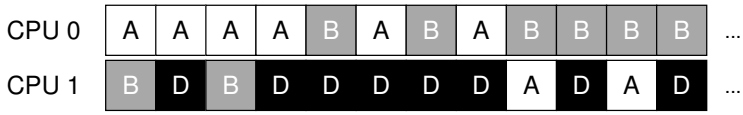
Çok kuyruklu çok işlemcili bir programlayıcı, istenen programlama hedeflerine daha iyi ulaşmak için yük dengesizliğini nasıl ele almalıdır?

Bu sorgunun en bariz cevabı, işleri taşımak, (bir kez daha) **geçiş yapmak (migration)** olarak adlandırdığımız bir teknik. Bir işin bir CPU ' dan başka bir işlemciye geçirilmesiyle, gerçek yük dengelemesi sağlanabilir.

Biraz açıklık eklemek için bir kaç örneğe bakalım. Bir kez daha, bir CPU ' un boşta olduğu ve diğerinin de bazı işleri olduğu bir durum ortaya çıktı.

Bu durumda, istenen geçişin anlaşılması kolaydır: İşletim sisteminin B veya D'den birini CPU 0'a taşıması yeterlidir. Bu tek iş geçişinin sonucu eşit dengeli 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 yer alır. Birkaç zaman diliminden sonra, B, CPU 0'da A ile rekabet etmek üzere hareket ederken, D, CPU 1'de birkaç zaman dilimini tek başına kullanır. Ve böylece yük dengelenir:



Tabii ki, birçok diğer olası taşıma deseni de mevcuttur. Ancak şimdi zor kısım:sistem bu tür bir taşımayı nasıl gerçekleştireceğine karar vermelidir?

[1] Az bilinen gerçek, Cybertron'un ana gezegeninin kötü CPU planlama kararları nedeniyle yok edildiğidir. Bu kitaptaki Transformers'a yapılan ilk ve son gönderme olsun, bunun için içtenlikle özür diliyoruz.

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, ki bu, çoklu kuyruk zamanlamasını ilk etapta uygulamanın tüm 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 İşlemcili Zamanlayıcıları

İlginçtir ki, Linux topluluğunda, çoklu işlemci zamanlayıcısını oluşturma konusunda ortak bir çözüm yoktur. Zaman içinde, üç farklı zamanlayıcı ortaya çıktı: O (1) zamanlayıcı, Tamamen Adil Zamanlayıcı (CFS) ve BF Zamanlayıcı (BFS) 2. Meehean'ın tezinde bu zamanlayıcıların güçlü ve zayıf yönlerine harika bir bakış bulabilirsiniz [M11]; burada sadece birkaç temel özetleyeceğiz.

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 zamanlayıcıları ayıran başka birçok detay var. Örneğin, O(1) zamanlayıcı, önceliğe dayalı bir zamanlayıcı (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

Çoklu işlemci zamanlama için çeşitli yaklaşımlar gördük. Tek kuyruk yaklaşımı (SQMS) oluşturmak için oldukça açık ve yükü iyi dengeler ancak özünde birçok işlemciye ölçekleme ve önbellek bağlılığı ile zorluk çeker. Çoklu kuyruk yaklaşımı (MQMS) daha iyi ölçekler ve önbellek bağlılığını iyi yönetir ancak yük dengesi ile zorlanır ve daha karmaşıktır. Hangi yaklaşımı seçerseniz seçin, basit bir cevap yoktur: genel amaçlı bir zamanlayıcı oluşturmak hala korkutucu bir görevdir, çünkü küçük kod değişiklikleri büyük davranış farklılıklarına neden olabilir. Bu tür bir egzersizi ancak tam olarak ne yaptığınızı ya da en azından bu iş için büyük bir miktarda para aldığınızı biliyorsanız yapın.

[2] BF'nin kendi başınıza neyi temsil ettiğine bakın; önceden uyarılmalıdır, kalbin zayıflığı için değildir

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, 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 queuing-based analysis papers.*

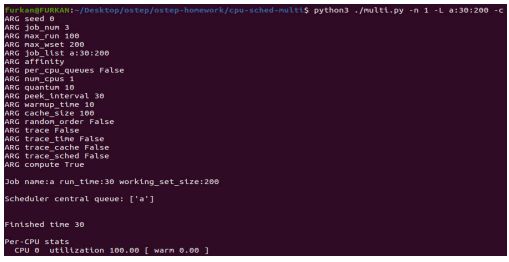
Ödev (Simülasyon)

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

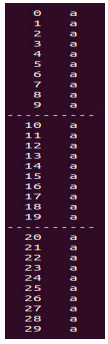
Sorular

1- İşe başlamak için, etkili bir çok işlemcili zamanlayıcının 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 aşağıdaki gibi çalıştırın: `./multi.py -n 1 -L a:30:200`. Tamamlanması ne kadar sürer? Son yanıtı görmek için `-c` işaretini, işin adım adım izini ve nasıl programlandığını görmek için `-t` işaretini açın.

c flagı ile komutumuzu çalıştırdığımızda 'Finished time 30' olarak görebiliriz. Yani 30 saniyede işlem tamamlanıyor.



-t flağı ile birlikte komutumuzu çalıştırdığımızda alacağımız ayrıntılı çıktı yandaki fotoğraftaki gibidir.



2- Şimdi çalışma kümesini (size = 200) önbelleğe sığdırmak için önbellek boyutunu artırın (varsayılan olarak size = 100); örneğin, ./multi.py -n 1 -L a:30:200 -M 300 çalıştırın. Önbellekte sığdığında işin nasıl hızlı çalışacağını tahmin edebilir misiniz? (ipucu: anahtar parametre olan sıcaklık oranını hatırlayın, bu -r bayrağı tarafından ayarlanır) Cevabınızı çözme bayrağını (-c) etkinleştirildiğinde çalıştırarak kontrol edin.

önbellek boyutunu arttırdığımızda programın daha hızlı çalıştığını görebiliyoruz. Aynı zamanda ısı değeri de değişmiş durumda.

```
furkan@FURKAN:~/Desktop/ostep/ostep-homework/cpu-sched-multi$ python3 ./multi.py -n 1 -L a:30:200 -M 300 -c
ARG seed 0
ARG job_num 3
ARG max_run 100
ARG max_wset 200
ARG job_list a:30:200
ARG affinity
ARG per_cpu_queues False
ARG num_cpus 1
ARG quantum 10
ARG peek_interval 30
ARG warmup_time 10
ARG cache_size 300
ARG random_order False
ARG trace False
ARG trace_time False
ARG trace_cache False
ARG trace_sched False
ARG compute True

Job name:a run_time:30 working_set_size:200

Scheduler central queue: ['a']


Finished time 20

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 50.00 ]
```

-t bayrağı ile bilrikte 'a' işinin detaylı çalışma şeklini gözlemleyebiliriz.

```
Job name:a run_time:30 working_set_size:200

Scheduler central queue: ['a']

   0   a
   1   a
   2   a
   3   a
   4   a
   5   a
   6   a
   7   a
   8   a
   9   a
----------
  10   a
  11   a
  12   a
  13   a
  14   a
  15   a
  16   a
  17   a
  18   a
  19   a

Finished time 20

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 50.00 ]
```

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

-T bayrağı ile birlikte hem her zaman adımında CPU'daki işi, hem de ne kadar çalışma süresi kaldığını görebiliriz.
Farkettiyseniz ikinci sütundan itibaren azalma 2'şerli olarak azalıyor.

```
  0   a [ 29]
  1   a [ 28]
  2   a [ 27]
  3   a [ 26]
  4   a [ 25]
  5   a [ 24]
  6   a [ 23]
  7   a [ 22]
  8   a [ 21]
  9   a [ 20]
---------------
 10   a [ 18]
 11   a [ 16]
 12   a [ 14]
 13   a [ 12]
 14   a [ 10]
 15   a [  8]
 16   a [  6]
 17   a [  4]
 18   a [  2]
 19   a [  0]
```

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?

-w parametresini varsayılandan daha düşük bir değer verdiğimizde önbellek daha erken ısınır, ısı değeri artar; daha yüksek verdiğimizde ise geç ve daha az ısınır.

python3 ./multi.py -n 1 -L a:30:200 -M 300 -c -C -w 2

```
   0   a cache[ ]
   1   a cache[w]
   2   a cache[w]
   3   a cache[w]
   4   a cache[w]
   5   a cache[w]
   6   a cache[w]
   7   a cache[w]
   8   a cache[w]
   9   a cache[w]
--------------------
  10   a cache[w]
  11   a cache[w]
  12   a cache[w]
  13   a cache[w]
  14   a cache[w]
  15   a cache[w]

Finished time 16

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 87.50 ]
```

python3 ./multi.py -n 1 -L a:30:200 -M 300 -c -C -w 18

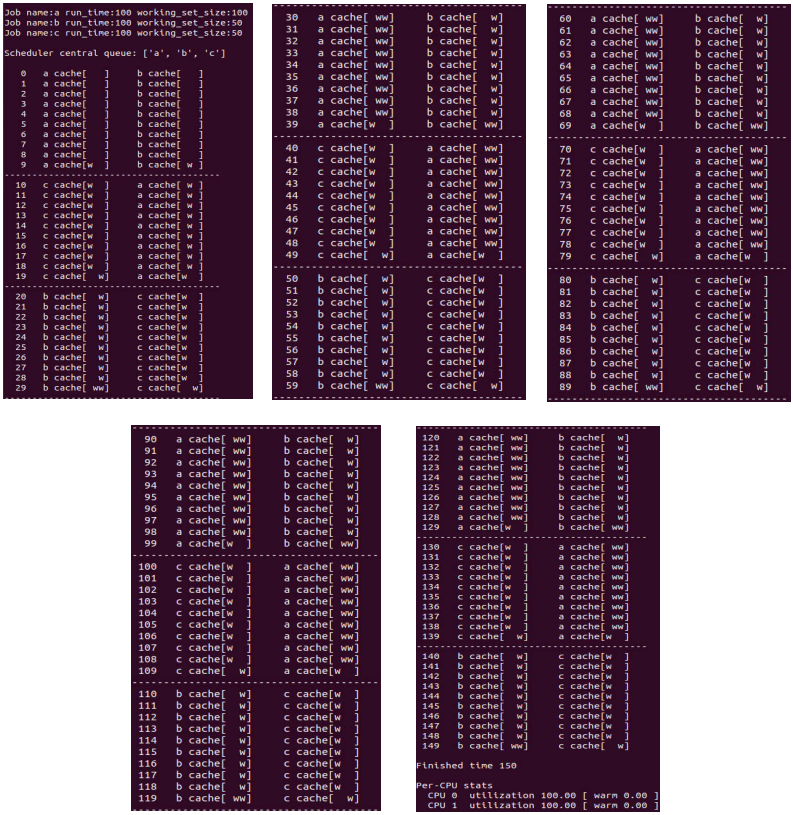
```
   0   a cache[ ]
   1   a cache[ ]
   2   a cache[ ]
   3   a cache[ ]
   4   a cache[ ]
   5   a cache[ ]
   6   a cache[ ]
   7   a cache[ ]
   8   a cache[ ]
   9   a cache[ ]
--------------------
  10   a cache[ ]
  11   a cache[ ]
  12   a cache[ ]
  13   a cache[ ]
  14   a cache[ ]
  15   a cache[ ]
  16   a cache[ ]
  17   a cache[w]
  18   a cache[w]
  19   a cache[w]
--------------------
  20   a cache[w]
  21   a cache[w]
  22   a cache[w]
  23   a cache[w]

Finished time 24

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 25.00 ]
```

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. Özellikle, aşağıdaki üç işi iki CPU'lu bir sistemde çalıştıralım (yani, `./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 edin? Haklı olup olmadığınızı görmek için `-c`'yi kullanın ve ardından adım adım görmek için `-t` ile ayrıntılara dalın ve ardından `-C` ile bu işler için önbelleklerin etkili bir şekilde ısıtılıp ısıtılmadığını görün. Ne farkettiniz?

python3 ./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -c -t -C
komutunu çalıştırdığımızda aldığımız çıktı bu şekilde olacaktır.



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 ./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?

python3 ./multi.py -n 2 -L a:100:100,b:100:50,c:100:50 -A a:0,b:1,c:1 -c
CPU sayısını arttırdığımız için daha hızlı çalıştığını gözlemiyoruz.

```
Job name:a run_time:100 working_set_size:100
Job name:b run_time:100 working_set_size:50
Job name:c run_time:100 working_set_size:50

Scheduler central queue: ['a', 'b', 'c']


Finished time 110

Per-CPU stats
  CPU 0  utilization 50.00 [ warm 40.91 ]
  CPU 1  utilization 100.00 [ warm 81.82 ]
```

7- Çoklu işlemcileri önbelleğe almanın ilginç bir yönü, tek bir işlemcide çalışan işlere kıyasla, birden fazla CPU (ve bunların önbellekleri) kullanılırken işleri beklenenden daha iyi hızlandırma fırsatıdır. Spesifik olarak, N CPU üzerinde çalıştığınızda, bazen N faktöründen daha fazla hızlandırabilirsiniz, bu durum süper lineer hızlanma (**super-linear speedup**) 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 arttıkça performans hakkında ne fark ediyorsunuz? Tahminlerinizi doğrulamak için -c'yi ve daha da derine dalmak için diğer izleme işaretlerini kullanın.

python3 ./multi.py -n 1 -n 2 -n 3 -L a:100:100,b:100:100,c:100:100 -M [] -A a:0,b:1,c:2 -c

M değerini 50 olarak verdiğimizde

```
Job name:a run_time:100 working_set_size:100
Job name:b run_time:100 working_set_size:100
Job name:c run_time:100 working_set_size:100

Scheduler central queue: ['a', 'b', 'c']


Finished time 100

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 0.00 ]
  CPU 1  utilization 100.00 [ warm 0.00 ]
  CPU 2  utilization 100.00 [ warm 0.00 ]
```

M değerini 100'e yükselttiğimizde

```
Job name:a run_time:100 working_set_size:100
Job name:b run_time:100 working_set_size:100
Job name:c run_time:100 working_set_size:100

Scheduler central queue: ['a', 'b', 'c']


Finished time 55

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 81.82 ]
  CPU 1  utilization 100.00 [ warm 81.82 ]
  CPU 2  utilization 100.00 [ warm 81.82 ]
```

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

"Gözetleme aralığı" (-P) seçeneği, bir iş parçacığının ne sıklıkla bir iş parçacığının performansını izlemesi gerektiğini belirtir. Daha düşük bir gözetleme aralığı, daha sık performans izleme anlamına gelir ve bu genellikle daha yüksek bir CPU yüküne neden olur. Daha yüksek bir gözetleme aralığı ise daha az sık performans izleme anlamına gelir ve bu genellikle daha düşük bir CPU yüküne neden olur.

CPU sayısı ölçeklendirilirken, daha düşük bir gözetleme aralığı genellikle daha iyi performans verir, ancak bu durum CPU yükünün artmasına neden olabilir. Bu, özellikle CPU sayısı çok yüksek olduğunda önemli olabilir, çünkü CPU'lar arasındaki yükü dengede tutmak için daha fazla çalışma yapılması gerekir. Daha yüksek bir gözetleme aralığı ise daha düşük bir CPU yüküne neden olabilir, ancak bu genellikle daha düşük bir performansa neden olur.

Bu durum, çok çekirdekli sistemlerde özellikle önemlidir, çünkü bu sistemlerde birden fazla CPU çekirdeği bulunur ve bu çekirdekler arasında yükü dengede tutmak gerekir. Bu nedenle, gözetleme aralığı seçiminin, çok çekirdekli sistemlerde performansı nasıl etkilediğine dikkat etmek önemlidir.

```
Finished time 90

Per-CPU stats
  CPU 0  utilization 94.44 [ warm 72.22 ]
  CPU 1  utilization 94.44 [ warm 72.22 ]
```

```
Finished time 110

Per-CPU stats
  CPU 0  utilization 68.18 [ warm 50.00 ]
  CPU 1  utilization 86.36 [ warm 68.18 ]
```

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 şanlar!

tek CPU'da 3 işlemi çalıştırdığımızda elde edeceğimiz sonuç:

```
Finished time 300

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 0.00 ]
```

3 işlemi iki CPU'ya dağıttığımızda elde ettiğimiz sonuç:

```
Finished time 150

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 0.00 ]
  CPU 1  utilization 100.00 [ warm 0.00 ]
```

Her işlem için birer CPU atadığımızda elde ettiğimiz sonuç ise:

```
Finished time 55

Per-CPU stats
  CPU 0  utilization 100.00 [ warm 81.82 ]
  CPU 1  utilization 100.00 [ warm 81.82 ]
  CPU 2  utilization 100.00 [ warm 81.82 ]
```

İşleri CPU lara dağıttığımızda aradaki performans farkının artışını görmemek mümkün değil. İşlerimizi mantıklı bir şekilde parçalara ayırdığımızda daha iyi performans sonuçları elde ediyoruz.