29

Kilit Tabanlı Eş Zamanlı Veri Yapıları

Kilitlerin ötesine geçmeden önce, bazı durumlarda ortak veri yapılarında kilitlerin nasıl kullanılacağını açıklayacağız. İş parçacığı tarafından kullanılabilir hale getirmek için bir veri yapısına kilitler eklemek, **yapı iş parçacığını güvenli(thread safe)** hale getirir. Elbette bu tür kilitlerin tam olarak nasıl eklendiği, veri yapısının hem doğruluğunu hem de performansını belirler. Ve böylece, meydan okumamız:

CRUX: VERİ YAPILARINA KİLİTLER NASIL EKLENİR

Belirli bir veri yapısı verildiğinde, doğru çalışması için ona nasıl kilitler eklemeliyiz? Ayrıca, veri yapısı yüksek performans sağlayacak ve birçok iş parçacığının yapıya aynı anda erişmesini sağlayacak şekilde nasıl kilitler ekleyeceğiz?

Elbette, eşzamanlılık eklemeye yönelik tüm veri yapılarını veya tüm yöntemleri ele almakta zorlanacağız, çünkü bu, (kelimenin tam anlamıyla) hakkında yayınlanan binlerce araştırma makalesiyle yıllardır üzerinde çalışılan bir konu. Bu nedenle, gerekli düşünme türüne yeterli bir giriş sunmayı ve kendi başınıza daha fazla araştırma yapmanız için sizi bazı iyi malzeme kaynaklarına yönlendirmeyi umuyoruz. Moir ve Shavit'in anketini harika bir bilgi kaynağı olarak gördük [MS04].

29.1 Eşzamanlı Sayaçlar

En basit veri yapılarından biri sayaçtır. Yaygın olarak kullanılan ve basit bir arayüze sahip bir yapıdır. Şekilde basit bir eşzamanlı olmayan sayaç tanımlıyoruz.

Basit Ama Ölçeklendirilemez

Gördüğünüz gibi, senkronize olmayan sayaç önemsiz bir veri yapısıdır, uygulamak için çok az miktarda kod gerektirir. Bizim sırada ki meydan okumamız: bu kod dizisini nasıl güvenli(**thread safe)** hale getiririz ? Şekil 29.2 bunu nasıl yapacağımızı gösterecek.

```
1  typedef struct __counter_t {
2      int value;
3  } counter_t;
4
5  void init(counter_t *c) {
6      c->value = 0;
7  }
8
9  void increment(counter_t *c) {
10     c->value++;
11 }
12
13 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

Şekil 29.1: **Kilitsiz Bir Sayaç**

Bu eşzamanlı sayaç basittir ve düzgün çalışır. Aslında, en basit ve en temel eşzamanlı tasarım modelinde ortak olan bir tasarım modelini izler. Veri yapıları: basitçe tek bir kilit ekler, bu, veri yapısını değiştiren bir rutini çağırırken elde edilir ve çağrıdan dönerken serbest bırakılır. Bu şekilde, siz nesne metotlarını çağırdıkça ve geri döndükçe kilitlerin otomatik olarak alındığı ve serbest bırakıldığı **monitörler(monitors)** [BH73] ile inşa edilmiş bir veri yapısına benzer.

Bu noktada, çalışan bir eşzamanlı veri yapınız var. Karşılaşabileceğiniz sorun performans olabilir. Veri yapınız çok yavaşsa tek bir kilit eklemekten fazlasını yapmanız gerekir; gerekirse bu tür optimizasyonlar bölümün geri kalanının konusudur. Veri yapısı çok yavaş *değilse*, işinizin bittiğini unutmayın! Basit bir şey işe yarayacaksa karmaşık bir şey yapmaya gerek yok.

Basit yaklaşımın performans maliyetlerini anlamak için, her iş parçacığının tek bir paylaşılan sayacı sabit sayıda güncellediği; daha sonra iş parçacığı sayısını değiştiririz. Şekil 29.5, bir ila dört iş parçacığının etkin olduğu toplam süreyi göstermektedir; her iş parçacığı sayacı bir milyon kez günceller. Bu deney, dört adet Intel 2,7 GHz i5 işlemcili bir iMac üzerinde gerçekleştirildi; daha fazla işlemci aktif olduğunda, birim zamanda daha fazla toplam iş yapılmasını umuyoruz.

Şeklin en üst satırından ('Kesin' olarak etiketlenmiştir), senkronize sayacın performansının düşük olduğunu görebilirsiniz. Tek bir iş parçacığı milyon sayaç güncellemesini çok kısa bir sürede (kabaca 0,03 saniye) tamamlayabilirken, iki iş parçacığının her birinin sayacı bir milyon kez aynı anda güncellemesi çok büyük bir yavaşlamaya yol açar (5 saniyeden fazla sürer!). Daha fazla iş parçacığı ile daha da kötüleşir.

```
1  typedef struct__counter_t {
2      int             value;
3      pthread_mutex_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
17 void decrement(counter_t *c) {
18     Pthread_mutex_lock(&c->lock);
19     c->value--;
20     Pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     Pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     Pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

Şekil 29.2 **Kilitli Bir Sayaç**

İdeal olarak, iş incelemenin birden fazla işlemcide tek iş parçacığının bir işlemcide yaptığı kadar hızlı tamamlandığını sahada. Bu amaca ulaşmak, **mükemmel ölçeklendirme (perfect scaling)** olarak adlandırılan; daha fazla iş yapılırsa bile paralel olarak yapılır ve bundan sonraki görevin tamamlanması için geçen süre artmaz.

Ölçeklenebilir Sayım

Şaşırtıcı bir şekilde, araştırmacılar yıllardır nasıl daha ölçeklenebilir sayaçlar inşa edeceklerini araştırıyorlar [MS04]. Daha da şaşırtıcı olanı, işletim sistemi performans analizinde yapılan son çalışmaların [B+10] gösterdiği gibi, ölçeklenebilir sayaçların önemli olduğu gerçeğidir; ölçeklenebilir sayım olmadan, Linux üzerinde çalışan bazı iş yükleri, çok çekirdekli makinelerde ciddi ölçeklenebilirlik sorunlarından muzdariptir.

Bu soruna saldırmak için birçok teknik geliştirilmiştir. **Yaklaşık sayaç** (**approximate counter**)[C06] olarak bilinen bir yaklaşımı açıklayacağız.

Yaklaşık sayaç, CPU çekirdeği başına bir tane olmak üzere çok sayıda yerel *fiziksel* sayaç ve tek bir genel sayaç aracılığıyla tek bir *mantıksal* sayacı temsil ederek çalışır. Spesifik olarak, dört CPU'lu bir makinede

Zaman	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	$5 \rightarrow 0$	1	3	4	5 (L_1 den)
7	0	2	4	$5 \rightarrow 0$	10 (L_4 den)

Şekil 29.3: **Yaklaşık Sayaçları İzleme**

Bu sayıcılara ek olarak, her bir yerel sayaç için birer ve global sayaç için birer tane olmak üzere kilitler de vardır. Yaklaşık saymanın temel fikri aşağıdaki gibidir. Belirli bir çekirdek üzerinde çalışan bir iş parçacığı sayacı artırmak istediğinde yerel sayacını artırır; bu yerel sayaca erişim, karşılık gelen yerel kilit aracılığıyla senkronize edilir. Her CPU'nun kendi yerel sayacı olduğundan, CPU'lar arasındaki iş parçacıkları yerel sayaçları çekişme olmadan güncelleyebilir ve bu nedenle sayaç güncellemeleri ölçeklenebilir.

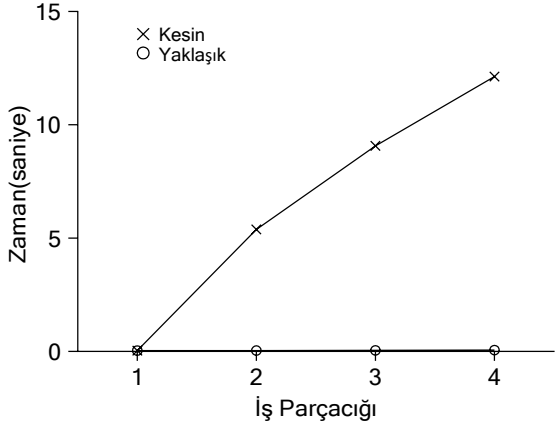
Bununla birlikte, global sayacı güncel tutmak için (bir iş parçacığının değerini okumak istemesi durumunda), yerel değerler, genel kilit elde edilerek ve yerel sayacın değeri kadar artırılarak, periyodik olarak küresel sayaca aktarılır; yerel sayaç daha sonra sıfırlanır. Bu yerelden küresele aktarımın ne sıklıkta gerçekleştiği bir eşik S tarafından belirlenir. S ne kadar küçükse, sayaç o kadar yukarıdaki ölçeklenemeyen sayaç gibi davranır; S ne kadar büyükse, sayaç o kadar ölçeklenebilir, ancak genel değer gerçek sayımdan o kadar uzak olabilir. Tam bir değer elde etmek için tüm yerel kilitler ve genel kilit (belirli bir sırayla, kilitlenmeyi önlemek için) elde edilebilir, ancak bu ölçeklenebilir değildir.

Bunu netleştirmek için bir örneğe bakalım (Şekil 29.3). Bu örnekte, S eşiği 5 olarak ayarlanmıştır ve dört CPU'nun her birinde yerel sayaçlarını L1 ... L4 güncelleyen iş parçacıkları vardır. Global sayaç değeri (G) de izde gösterilir ve süre aşağı doğru artar. Her zaman adımında, bir yerel sayaç artırılabilir; yerel değer S eşiğine ulaşırsa, yerel değer global sayaca aktarılır ve yerel sayaç sıfırlanır.

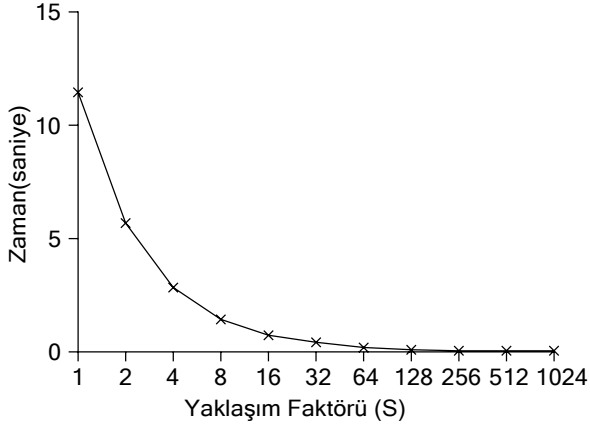
Şekil 29.5'teki alt satır ('Yaklaşık' olarak etiketlenmiştir, sayfa 6), 1024 S eşiğine sahip yaklaşık sayaçların performansını gösterir. Performans mükemmel; sayacın dört işlemcide dört milyon kez güncellenmesi için geçen süre, bir işlemcide bir milyon kez güncellenmesi için geçen süreden neredeyse daha yüksek değildir.

```
1  typedef struct___counter_t {
2      int             global;         // genel sayaç
3      pthread_mutex_t glock;          // genel kilit
4      int             local[NUMCPUS]; // CPU başına sayım
5      pthread_mutex_t llock[NUMCPUS]; // ... ve kilitler
6      int             threshold;      // güncelleme sıklıgı
7  } counter_t;
8
9  // init: kayıt eşiği, init kilitleri, init değerleri
10 //       tüm yerel sayaçların ve genel sayaçların
11 void init(counter_t *c, int threshold) {
12     c->threshold = threshold;
13     c->global = 0;
14     pthread_mutex_init(&c->glock, NULL);
15     int i;
16     for (i = 0; i < NUMCPUS; i++) {
17         c->local[i] = 0;
18         pthread_mutex_init(&c->llock[i], NULL);
19     }
20 }
21
22 // update: öncelikle,yerel kilidi alın ve güncelleyin
23 // yerel miktar; yerel sayım 'eşik' yükseldiğinde,
24 // genel kilidi alın ve yerel değerleri ona aktarın
25 void update(counter_t *c, int threadID, int amt) {
26     int cpu = threadID % NUMCPUS;
27     pthread_mutex_lock(&c->llock[cpu]);
28     c->local[cpu] += amt;
29     if (c->local[cpu] >= c->threshold) {
30         // transfer to global (assumes amt>0)
31         pthread_mutex_lock(&c->glock);
32         c->global += c->local[cpu];
33         pthread_mutex_unlock(&c->glock);
34         c->local[cpu] = 0;
35     }
36     pthread_mutex_unlock(&c->llock[cpu]);
37 }
38
39 // get: sadece genel miktarı yaklaşık değeri geri döndür
40 int get(counter_t *c) {
41     pthread_mutex_lock(&c->glock);
42     int val = c->global;
43     pthread_mutex_unlock(&c->glock);
44     return val; // sadece yaklaşık değer!
45 }
```

Şekil 29.4: **Yaklaşık Karşı Uygulaması**



Şekil 29.5: **Geleneksel ve Yaklaşık Sayaçların Performansı**



Şekil 29.6: **Yaklaşık Sayaçları Ölçeklendirme**

Şekil 29.6, dört CPU'da her biri sayacı 1 milyon kez artıran dört iş parçacığı ile S eşik değerinin önemini göstermektedir. S düşükse, performans zayıftır (ancak genel sayım her zaman oldukça doğrudur); S yüksekse, performans mükemmeldir, ancak genel sayım gecikir (en fazla CPU sayısı S ile çarpılır). Bu doğruluk/performans değiş tokuşu, yaklaşık sayaçların mümkün kıldığı şeydir.

Yaklaşık bir sayacın kaba bir versiyonu Şekil 29.4'te (sayfa 5) de bulunur. Okuyun veya daha iyisi, nasıl çalıştığını daha iyi anlamak için bazı deneylerde kendiniz çalıştırın.

İPUCU: FAZLA EŞZAMANLILIK HER ZAMAN HIZLI DEĞİLDİR
Tasarladığınız şema çok fazla ek yük getiriyorsa (örneğin, kilitleri bir kez yerine sık sık alıp bırakarak), eş zamanlı önemli olmayabilir. Basit şemalar, özellikle nadiren maliyetli rutinler kullanıyorlarsa, iyi çalışma eğilimindedir. Daha fazla kilit ve karmaşıklık eklemek sizin düşüşünüz olabilir. Bütün bunlar, gerçekten bilmenin bir yolu var: her iki alternatifi de oluşturun (basit ama daha az eşzamanlı ve karmaşık ama daha fazla eşzamanlı) ve nasıl yaptıklarını ölçün. Sonuç olarak performans konusunda kopya çekemezsiniz; fikriniz ya daha hızlıdır ya da değildir.

29.2 Eş Zamanlı Bağlantılı Listeler

Şimdi daha karmaşık bir yapı olan bağlantılı listeyi inceleyeceğiz. Bir kez daha temel bir yaklaşımla başlayalım. Basit olması için, böyle bir listenin sahip olabileceği bariz rutinlerden bazılarını atlayacağız ve sadece eşzamanlı eklemeye odaklanacağız; arama, silme vb. hakkında düşünmeyi okuyucuya bırakacağız. Şekil 29.7, bu ilkel veri yapısının kodunu göstermektedir.

Kodda görebileceğiniz gibi, kod girişte ekleme rutininde bir kilit alır ve çıkışta onu serbest bırakır. Malloc() başarısız olursa (nadir görülen bir durum) küçük bir zor sorun ortaya çıkar; bu durumda, kodun, ekleme başarısız olmadan önce kilidi de serbest bırakması gerekir.

Bu tür istisnai kontrol akışının oldukça hatalı olduğu gösterilmiştir. Yatkın; Linux çekirdeği yamaları üzerine yakın zamanda yapılan bir araştırma, hatalar (yaklaşık %40) bu tür nadiren alınan kod yollarında bulunur (aslında bu, gözlem, kendi araştırmamızın bir kısmını ateşledi; bir Linux dosya sisteminden bellek arızası olan yollar, daha sağlam bir sonuçla sonuçlanır sistemi [S+11]).Bu nedenle, bir zorluk: Ekleme ve arama yordamlarını eşzamanlı ekleme altında doğru kalacak şekilde yeniden yazabilir miyiz, ancak hata yolunun aynı zamanda kilidi açmak için aramayı eklememizi gerektirdiği durumu önleyebilir miyiz? Cevap, bu durumda, evet. Spesifik olarak, kodu biraz yeniden düzenleyebiliriz, böylece kilitleme ve serbest bırakma yalnızca ekleme kodundaki gerçek kritik bölümü çevreler ve arama kodunda ortak bir çıkış yolu kullanılır. İlki çalışır, çünkü ek parçanın bir kısmının aslında kilitlenmesi gerekmez; malloc()'un kendisinin iş parçacığı açısından güvenli olduğunu varsayarsak, her iş parçacığı onu yarış koşulları veya diğer eşzamanlılık hataları endişesi olmadan çağırabilir. Yalnızca paylaşılan liste güncellenirken bir kilidin tutulması gerekir. Bu değişikliklerin ayrıntıları için bkz. Şekil 29.8.

Arama rutinine gelince, ana arama döngüsünden tek bir dönüş yoluna atlamak basit bir kod dönüşümüdür. Bunu tekrar yapmak, koddaki kilit alma/serbest bırakma noktalarının sayısını azaltır ve böylece yanlışlıkla koda hatalar (geri dönmeden önce kilidi açmayı unutmak gibi) ekleme şansını azaltır.

```
1  // temel düğüm yapısı
2  typedef struct___node_t {
3      int                 key;
4      struct___node_t         *next;
5  } node_t;
6
7  // temel liste yapısı   (liste başına bir
                           tane kullanılır)
8  typedef struct___list_t {
9      node_t                *head;
10     pthread_mutex_t     lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14     L->head = NULL;
15     pthread_mutex_init(&L->lock, NULL);
16 }
17
18 int List_Insert(list_t *L, int key) {
19     pthread_mutex_lock(&L->lock);
20     node_t *new = malloc(sizeof(node_t));
21     if (new == NULL) {
22         perror("malloc");
23         pthread_mutex_unlock(&L->lock);
24         return -1; // başarısız
25     }
26     new->key  = key;
27     new->next = L->head;
28     L->head   = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0; // başarılı
31 }
32
33 int List_Lookup(list_t *L, int key) {
34     pthread_mutex_lock(&L->lock);
35     node_t *curr = L->head;
36     while (curr) {
37         if (curr->key == key) {
38             pthread_mutex_unlock(&L->lock);
39             return 0; // başarılı
40         }
41         curr = curr->next;
42     }
43     pthread_mutex_unlock(&L->lock);
44     return -1; // başarısız
45 }
```

Şekil 29.7: **Eş Zamanlı Bağlantılı Liste**

```
1  void List_Init(list_t *L) {
2      L->head = NULL;
3      pthread_mutex_init(&L->lock, NULL);
4  }
5
6  void List_Insert(list_t *L, int key) {
7      // senkronizasyon gerekli değil
8      node_t *new = malloc(sizeof(node_t));
9      if (new == NULL) {
10         perror("malloc");
11         return;
12     }
13     new->key = key;
14
15     // sadece kritik bölümü kilitle
16     pthread_mutex_lock(&L->lock);
17     new->next = L->head;
18     L->head   = new;
19     pthread_mutex_unlock(&L->lock);
20 }
21
22 int List_Lookup(list_t *L, int key) {
23     int rv = -1;
24     pthread_mutex_lock(&L->lock);
25     node_t *curr = L->head;
26     while (curr) {
27         if (curr->key == key) {
28             rv = 0;
29             break;
30         }
31         curr = curr->next;
32     }
33     pthread_mutex_unlock(&L->lock);
34     return rv; // hem başarı hem de başarısızlık
35 }
```

Şekil 29.8: **Eşzamanlı Bağlantılı Liste: Yeniden Yazıldı**

Bağlantılı Listeleri Ölçeklendirme

Yine temel bir eşzamanlı bağlantılı listemiz olmasına rağmen, bir kez daha bunun özellikle iyi ölçeklenemediği bir durumdayız. Araştırmacıların bir liste içinde daha fazla eş zamanlılık sağlamak için keşfettikleri bir teknik, **elle kilitleme**(**hand-over-hand locking**) (a.k.a. **kilit bağlantısı**(**lock coupling**)) [MS04] olarak adlandırılan bir tekniktir. Fikir oldukça basit. Tüm liste için tek bir kilide sahip olmak yerine, listenin düğümü başına bir kilit eklersiniz. Listede gezinirken, kod önce bir sonraki düğümün kilidini alır ve ardından mevcut düğümün kilidini serbest bırakır (bu, elden teslim ismine ilham verir).

İPUCU: KİLİTLERE VE KONTROL AKIŞINA DİKKAT EDİN
Eşzamanlı kodda ve başka yerlerde yararlı olan genel bir tasarım ipucu, işlev dönüşlerine, çıkışlarına veya bir işlevin yürütülmesini durduran diğer benzer hata koşullarına yol açan kontrol akışı değişikliklerine karşı dikkatli olmaktır. Birçok işlev bir kilit alarak, bir miktar bellek ayırarak veya diğer benzer durum bilgisi olan işlemler yaparak başlayacağından, hatalar ortaya çıktığında kodun geri dönmeden önce tüm durumu geri alması gerekir ki bu hataya açıktır. Bu nedenle, bu modeli en aza indirgemek için kodu yapılandırmak en iyisidir.

Kavramsal olarak, elden ele bağlantılı bir liste bir anlam ifade eder; liste işlemlerinde yüksek derecede eşzamanlılık sağlar. Bununla birlikte, pratikte böyle bir yapıyı basit tek kilit yaklaşımından daha hızlı yapmak zordur, çünkü bir liste geçişinin her düğümü için kilit alma ve serbest bırakma ek yükleri engelleyicidir. Çok büyük listelerde ve çok sayıda iş parçacığında bile, birden çok devam eden geçişe izin vererek etkinleştirilen eşzamanlılığın, tek bir kilidi kapmak, bir işlem gerçekleştirmek ve onu serbest bırakmaktan daha hızlı olması muhtemel değildir. Belki de bir tür hibrit (her düğümde yeni bir kilit aldığınız yer) araştırmaya değer olabilir.

29.3 Eşzamanlı Kuyruklar

Şimdiye kadar bildiğiniz gibi, eşzamanlı bir veri yapısı oluşturmak için her zaman standart bir yöntem vardır: büyük bir kilit ekleyin. Sıra için, çözebileceğinizi varsayarak bu yaklaşımı atlayacağız.

Bunun yerine, Michael ve Scott [MS98] tarafından tasarlanan biraz daha eşzamanlı bir kuyruğa göz atacağız. Bu kuyruk için kullanılan veri yapıları ve kod, bir sonraki sayfada Şekil 29.9'da bulunmaktadır.

Bu kodu dikkatlice incelerseniz, biri kuyruğun başı, diğeri kuyruk için olmak üzere iki kilit olduğunu fark edeceksiniz. Bu iki kilidin amacı, kuyruğa alma ve kuyruktan çıkarma işlemlerinin eşzamanlılığını sağlamaktır. Genel durumda, kuyruğa alma yordamı yalnızca kuyruk kilidine erişecek ve yalnızca kafa kilidini kuyruktan çıkaracaktır. Michael ve Scott tarafından kullanılan hilelerden biri, sahte bir düğüm eklemektir (kuyruk başlatma kodunda tahsis edilmiştir); bu kukla, baş ve kuyruk işlemlerinin ayrılmasını sağlar. Derinlemesine nasıl çalıştığını anlamak için kodu inceleyin veya daha iyisi yazın, çalıştırın ve ölçün. Kuyruklar genellikle çok iş parçacıklı uygulamalarda kullanılır. Ancak burada kullanılan sıra türü (sadece kilitlerle) genellikle bu tür programların ihtiyaçlarını tam olarak karşılamaz. Kuyruk boşsa veya aşırı doluysa bir iş parçacığının beklemesini sağlayan daha tam gelişmiş bir sınırlı sıra, koşul değişkenleri üzerine bir sonraki bölümde yapacağımız yoğun çalışmanın konusudur. Dikkat et!

```
1  typedef struct __node_t {
2      int                 value;
3      struct __node_t    *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t              *head;
8      node_t              *tail;
9      pthread_mutex_t     head_lock, tail_lock;
10 } queue_t;
11
12 void Queue_Init(queue_t *q) {
13     node_t *tmp = malloc(sizeof(node_t));
14     tmp->next = NULL;
15     q->head = q->tail = tmp;
16     pthread_mutex_init(&q->head_lock, NULL);
17     pthread_mutex_init(&q->tail_lock, NULL);
18 }
19
20 void Queue_Enqueue(queue_t *q, int value) {
21     node_t *tmp = malloc(sizeof(node_t));
22     assert(tmp != NULL);
23     tmp->value = value;
24     tmp->next  = NULL;
25
26     pthread_mutex_lock(&q->tail_lock);
27     q->tail->next = tmp;
28     q->tail = tmp;
29     pthread_mutex_unlock(&q->tail_lock);
30 }
31
32 int Queue_Dequeue(queue_t *q, int *value) {
33     pthread_mutex_lock(&q->head_lock);
34     node_t *tmp = q->head;
35     node_t *new_head = tmp->next;
36     if (new_head == NULL) {
37         pthread_mutex_unlock(&q->head_lock);
38         return -1; // kuyruk boştu
39     }
40     *value = new_head->value;
41     q->head = new_head;
42     pthread_mutex_unlock(&q->head_lock);
43     free(tmp);
44     return 0;
45 }
```

Şekil 29.9: **Michael ve Scott Eşzamanlı Kuyruğu**

```
1  #define BUCKETS (101)
2
3  typedef struct __hash_t {
4      list_t lists[BUCKETS];
5  } hash_t;
6
7  void Hash_Init(hash_t *H) {
8      int i;
9      for (i = 0; i < BUCKETS; i++)
10         List_Init(&H->lists[i]);
11 }
12
13 int Hash_Insert(hash_t *H, int key) {
14     return List_Insert(&H->lists[key % BUCKETS], key);
15 }
16
17 int Hash_Lookup(hash_t *H, int key) {
18     return List_Lookup(&H->lists[key % BUCKETS], key);
19 }
```

Şekil 29.10: **Eşzamanlı Hash Tablosu**

29.4 Eşzamanlı Hash Tablosu

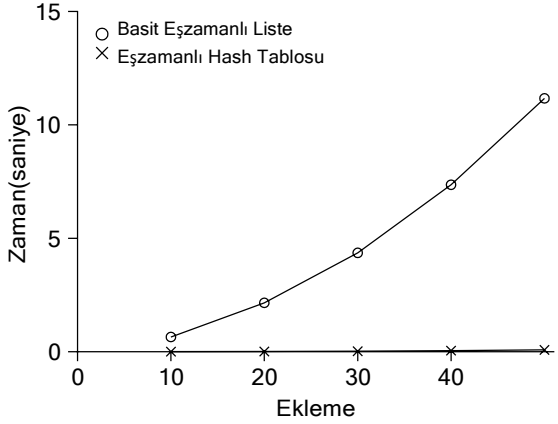
Tartışmamızı basit ve yaygın olarak uygulanabilir bir eşzamanlı veri yapısı olan hash tablosu ile bitiriyoruz. Yeniden boyutlandırmayan basit bir hash tablosuna odaklanacağız; yeniden boyutlandırmayı halletmek için biraz daha çalışma gerekiyor, bunu okuyucuya bir alıştırma olarak bırakıyoruz (üzgünüm!).

Bu eşzamanlı hash tablosu (Şekil 29.10) basittir, daha önce geliştirdiğimiz eşzamanlı listeler kullanılarak oluşturulmuştur ve inanılmaz derecede iyi çalışır. İyi performansının nedeni, tüm yapı için tek bir kilide sahip olmak yerine, hash kovası başına bir kilit kullanmasıdır (her biri bir liste ile temsil edilir). Bunu yapmak, birçok eşzamanlı işlemin gerçekleşmesini sağlar.

Şekil 29.11, eşzamanlı güncellemeler altında hash tablosunun performansını gösterir (dört CPU'lu aynı iMac'te, dört iş parçacığının her birinden 10.000 ila 50.000 eşzamanlı güncelleme). Karşılaştırma amacıyla, bağlantılı bir listenin (tek kilitli) performansı da gösterilmiştir. Grafikten de görebileceğiniz gibi, bu basit eşzamanlı hash tablosu muhteşem bir şekilde ölçekleniyor; bağlantılı liste ise aksine yapmaz.

29.5 Özet

Sayaçlardan listelere ve sıralara ve son olarak her yerde bulunan ve yoğun şekilde kullanılan hash tablosuna kadar eşzamanlı veri yapılarının bir örneğini tanıttık. Yol boyunca birkaç önemli ders öğrendik: kontrol çevresinde kilitlerin alınması ve serbest bırakılması konusunda dikkatli olmak akış



Şekil 29.11: **Hash Tablolarını Ölçeklendirme**

değişiklikler; daha fazla eşzamanlılığın etkinleştirilmesinin mutlaka performansı artırmadığı; performans sorunlarının yalnızca ortaya çıktıktan sonra çözülmesi gerektiğini. **Erken optimizasyondan (premature optimization)** kaçınmaya ilişkin bu son nokta, performansa önem veren herhangi bir geliştiricinin merkezinde yer alır; uygulamanın genel performansını iyileştirmeyecekse, bir şeyi daha hızlı yapmanın hiçbir değeri yoktur.

Tabii ki, yüksek performanslı yapıların yüzeyini çizdik. Daha fazla bilgi ve diğer kaynaklara bağlantıları için Moir ve Shavit'in mükemmel anketine bakın. [MS04] Özellikle, diğer yapılarla **(B-ağaçlar(B-trees) gibi)** ilgilenebilirsiniz; bu bilgi için, bir veritabanı sınıfı en iyi seçeneğinizdir. Geleneksel kilitleri hiç kullanmayan teknikleri de merak ediyor olabilirsiniz; bu tür **engellemeyen veri yapıları(non-blocking data structures)**, ortak eşzamanlılık hataları ile ilgili bölümde tadına bakacağımız bir şeydir, ancak açıkçası bu konu, bu mütevazı kitapta mümkün olandan daha fazla çalışma gerektiren eksiksiz bir bilgi alanıdır. Arzu ederseniz (her zaman olduğu gibi!) daha fazlasını kendi başınıza öğrenin.

İPUCU: ERKEN OPTİMİZASYONDAN KAÇININ (KNUTH YASASI)

Eşzamanlı bir veri yapısı oluştururken, senkronize erişim sağlamak için tek bir büyük kilit eklemek olan en temel yaklaşımla başlayın. Bunu yaparak, muhtemelen doğru bir kilit oluşturacaksınız; daha sonra performans sorunları yaşadığını fark ederseniz, düzeltebilir, böylece yalnızca gerektiğinde hızlı hale getirebilirsiniz. **Knuth**'un ünlü bir şekilde belirttiği gibi, "Erken optimizasyon tüm kötülüklerin köküdür."

Sun OS ve Linux dahil olmak üzere birçok işletim sistemi, çoklu işlemcilere ilk geçişte tek bir kilit kullandı. İkincisinde, bu kilidin bir adı bile vardı, **büyük çekirdek kilidi** (BKL). Uzun yıllar boyunca bu basit yaklaşım iyi bir yaklaşımdı, ancak çoklu CPU sistemleri norm haline geldiğinde, çekirdekte aynı anda yalnızca tek bir aktif iş parçacığına izin vermek bir performans darboğazına dönüştü. Böylece, nihayet bu sistemlere geliştirilmiş eş zamanlılık optimizasyonunu eklemenin zamanı gelmişti. Linux'ta daha basit bir yaklaşım izlendi: bir kilidi birçok kilitle değiştirin. Sun içinde daha radikal bir karar alındı: Solaris olarak bilinen ve ilk günden itibaren eşzamanlılığı daha temelden birleştiren yepyeni bir işletim sistemi oluşturmak. Bu büyüleyici sistemler [BC05, MM00] hakkında daha fazla bilgi için Linux ve Solaris çekirdek kitaplarını okuyun.

References

[B+10] "An Analysis of Linux Scalability to Many Cores" by Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zel- dovich . OSDI '10, Vancouver, Canada, October 2010. *A great study of how Linux performs on multicore machines, as well as some simple solutions. Includes a neat* ***sloppy counter*** *to solve one form of the scalable counting problem.*

[BH73] "Operating System Principles" by Per Brinch Hansen. Prentice-Hall, 1973. Available: `http://portal.acm.org/citation.cfm?id=540365`. *One of the first books on operating systems; certainly ahead of its time. Introduced monitors as a concurrency primitive.*

[BC05] "Understanding the Linux Kernel (Third Edition)" by Daniel P. Bovet and Marco Cesati. O'Reilly Media, November 2005. *The classic book on the Linux kernel. You should read it.*

[C06] "The Search For Fast, Scalable Counters" by Jonathan Corbet. February 1, 2006. Avail- able: `https://lwn.net/Articles/170003`. *LWN has many wonderful articles about the latest in Linux This article is a short description of scalable approximate counting; read it, and others, to learn more about the latest in Linux.*

[L+13] "A Study of Linux File System Evolution" by Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu. FAST '13, San Jose, CA, February 2013. *Our paper that studies every patch to Linux file systems over nearly a decade. Lots of fun findings in there; read it to see! The work was painful to do though; the poor graduate student, Lanyue Lu, had to look throughevery single patch by hand in order to understand what they did.*

[MS98] "Nonblocking Algorithms and Preemption-safe Locking on by Multiprogrammed Shared-memory Multiprocessors. " M. Michael, M. Scott. Journal of Parallel and Distributed Com- puting, Vol. 51, No. 1, 1998 *Professor Scott and his students have been at the forefront of concurrent algorithms and data structures for many years; check out his web page, numerous papers, or books to find out more.*

[MS04] "Concurrent Data Structures" by Mark Moir and Nir Shavit. In Handbook of Data Structures and Applications (Editors D. Metha and S.Sahni). Chapman and Hall/CRC Press, 2004. Available: `www.cs.tau.ac.il/~shanir/concurrent-data-structures.pdf`. *A short but relatively comprehensive reference on concurrent data structures. Though it is missingsome of the latest works in the area (due to its age), it remains an incredibly useful reference.*

[MM00] "Solaris Internals: Core Kernel Architecture" by Jim Mauro and Richard McDougall. Prentice Hall, October 2000. *The Solaris book. You should also read this, if you want to learn about something other than Linux.*

[S+11] "Making the Common Case the Only Case with Anticipatory Memory Allocation" by Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau . FAST '11, San Jose, CA, February 2011. *Our work on removing possibly-failing allocation calls from kernel code paths. By allocating all potentially needed memorybefore doing any work, we avoid failure deep down in the storage stack.*

Ödev (Kod)

Bu ödevde, eşzamanlı kod yazma ve performansını ölçme konusunda biraz deneyim kazanacaksınız. İyi performans gösteren kod oluşturmayı öğrenmek kritik bir beceridir ve bu nedenle burada biraz deneyim kazanmak oldukça değerlidir.

Sorular

1. Bu bölümdeki ölçümleri yeniden yaparak başlayacağız. Programınızda zamanı ölçmek için gettimeofday() çağrısını kullanın. Bu zamanlayıcı ne kadar doğru? Ölçebileceği en küçük aralık nedir? Sonraki tüm sorularda ihtiyacımız olacağından, işleyişine güven kazanın. Ayrıca, **rdtsc** komutu aracılığıyla x86'da bulunan döngü sayacı gibi diğer zamanlayıcılara da bakabilirsiniz.

   gettimeofday() Posix Epoch'tan beri geçen süreyi saniye + mikrosaniye çözünürlüğünde alabilmemizi sağlar. Yani ölçebildiği en küçük aralık mikrosaniye cinsindendir.

   Rdtsc: Zaman Damgası Sayacı Okuyucu (Read Time-Stamp Counter) İşlemcinin zaman damgası sayacının geçerli değerini EDX:EAX kayıtlarına yükler. İşlemci, zaman damgası sayacı MSR'yi her saat döngüsünde monoton bir şekilde artırır ve işlemci her sıfırlandığında bunu 0'a sıfırlar.

2. Şimdi, basit bir eşzamanlı sayaç oluşturun ve iş parçacığı sayısı arttıkça sayacı birçok kez artırmanın ne kadar sürdüğünü ölçün. Kullandığınız sistemde kaç tane CPU var? Bu sayı ölçümlerinizi hiç etkiliyor mu?

3. Ardından özensiz sayacın bir sürümünü oluşturun. Eşiğin yanı sıra iş parçacığı sayısı değiştikçe performansını bir kez daha ölçün. Sayılar bölümde gördüğünüzle eşleşiyor mu?

4. Bölümde belirtildiği gibi, elden ele kilitleme [MS04] kullanan bir bağlantılı liste sürümü oluşturun. Nasıl çalıştığını anlamak için önce makaleyi okumalı ve sonra onu uygulamalısınız. Performansını ölçün. Elden teslim listesi, bölümde gösterildiği gibi standart bir listeden ne zaman daha iyi çalışır?

5. B-ağacı**(B-tree**) veya diğer biraz daha ilginç yapı gibi favori veri yapınızı seçin. Bunu uygulayın ve tek kilit gibi basit bir kilitleme stratejisiyle başlayın. Eşzamanlı iş parçacığı sayısı arttıkça performansını ölçün.

   B-tree yapısı struct node veri yapısıyla tanımlanmış olan bir çeşit node'dur. İçerisinde veri tutan datadeğişkeni, adres bilgisi tutan left ve right isimli iki işaretçisi bulunmaktadır. Bununla birlikte iş parçacığı sayısı arttıkça B-tree performansının düştüğünü görebiliriz.

6. Son olarak, bu favori veri yapınız için daha ilginç bir kilitleme stratejisi düşünün. Uygulayın ve performansını ölçün. Basit kilitleme yaklaşımıyla nasıl karşılaştırılır?