

Kilit Tabanlı Eş Zamanlı Veri Yapıları

Kilitlerin ötesine geçmeden önce, bazı genel veri yapılarında kilitlerin nasıl kullanılacağını açıklayacağız. Bir veri yapısına, Thread 'ler tarafından kullanılabilir hale getirmek için kilitler eklemek, yapıyı thread 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, kilitleri nasıl eklemeliyiz doğru çalışması için mi? Ayrıca, bu tür kilitleri nasıl ekleriz veri yapısının yüksek performans sağlayarak birçok thread lere olanak tanıması yapıya bir kerede, yani aynı anda erişmek için mi?

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.

29.1 Eşzamanlı Sayaçlar

En basit veri yapılarından biri sayaçtır. Yaygın olarak kullanılan ve basit bir arayüze sahip bir yapıdır. Şekil 29.1'de basit bir eşzamanlı olmayan sayaç tanımlıyoruz.

Basit Ama Ölçeklendirilemez .

Gördüğünüz gibi, senkronize olmayan sayaç, uygulanması için çok az miktarda kod gerektiren önemsiz bir veri yapısıdır. Şimdi sıradaki görevimiz var: Bu kodu nasıl thread safe hale getirebiliriz? Şekil 29.2 bunu nasıl yaptığımızı göstermektedir.

```

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: Kilitli Bir Sayaç

Bu eşzamanlı sayaç basittir ve düzgün çalışır. Aslında, en basit ve en temel eşzamanlı veri yapılarında ortak olan bir tasarım modelini takip eder: basitçe, veri yapısını değiştiren bir rutini çağırırken elde edilen ve çağrıdan dönerken serbest bırakılan tek bir kilit ekler. Bu şekilde, siz nesne metodlarını çağırdığınızda ve geri döndüğünüzde kilitlerin otomatik olarak alındığı ve serbest bırakıldığı monitörler(monitors) [BH73] ile oluşturulmuş bir veri yapısına benzer.

Bu noktada, çalışan bir eşzamanlı veri yapınız var. Karşılaşılabileceğiniz sorun performans olabilir. Veri yapınız çok yavaşsa, tek bir kilit eklemekten daha fazlasını yapmanız gerekir; Gerekirse bu tür optimizasyonlar bu nedenle bölümün geri kalanının konusudur. Veri yapısı çok yavaş değilse, işinizin bittiğini unutmayın! Basit bir şey işe yarayacaksa, süslü bir şey yapmanıza gerek yok.

Basit yaklaşımın performans maliyetlerini anlamak için, her thread'in tek bir paylaşılan sayacı sabit sayıda güncellediği; daha sonra thread sayısını değiştiririz. Şekil 29.3, bir ila dört thread 'in aktif olduğu toplam süreyi göstermektedir; her thread sayacı bir milyon kez günceller. Bu deney, dört adet Intel 2,7 GHz i5 CPU'lu bir iMac üzerinde gerçekleştirildi; daha fazla CPU aktif olduğunda, birim zamanda daha fazla toplam iş yapılmasını umuyoruz.

Şekildeki en üst satırdan (kesin olarak etiketlenmiş), senkronize sayacın performansının kötü ölçeklendiğini görebilirsiniz. Tek bir thread 'in milyon sayaç güncellemesini çok kısa bir sürede (kabaca 0,03 saniye) tamamlayabilirken, iki thread 'in her birinin sayacı bir milyon kez aynı anda güncellemesi çok büyük bir yavaşlamaya yol açar (5 saniyeden fazla sürer!). Sadece daha fazla thread 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, thread 'lerin birden çok işlemcide, tek thread'in bir işlemcide yaptığı kadar hızlı tamamlandığını görmek istersiniz. Bu amaca ulaşmak, mükemmel ölçeklendirme (perfect scaling) olarak adlandırılır; daha fazla iş yapılsa bile paralel olarak yapılır ve dolayısıyla görevi tamamlamak için harcanan süre artmaz.

Ölçeklenebilir Sayım

Şaşırtıcı bir şekilde, araştırmacılar yıllardır [MS04] daha ölçeklenebilir sayaçların nasıl oluşturulacağını araştırıyorlar. Daha da şaşırtıcı olanı, ölçeklenebilir sayaçların işletim sistemi performans analizinde yapılan son çalışmalar gösterilen [B+10]; 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) olarak bilinen bir yaklaşımı açıklayacağız [C06].

Özensiz 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 dört yerel sayaç ve bir küresel sayaç vardır. Bu sayaçlara ek olarak, her bir yerel sayaç için bir tane ve global sayaç için bir tane olmak üzere kilitler de vardır.

Time	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 (from L_1)
7	0	2	4	$5 \rightarrow 0$	10 (from L_4)

Şekil 29.3: Yaklaşık Sayaçların İzlenmesi

Özensiz saymanın temel fikri aşağıdaki gibidir. Belirli bir çekirdek üzerinde çalışan bir thread'in 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 threadler yerel sayaçları çekişme olmadan güncelleyebilir ve bu nedenle sayaç güncellemeleri ölçeklenebilir.

Bununla birlikte, global sayacı güncel tutmak için (bir thread 'in değerini okumak istemesi durumunda), yerel değerler, küresel 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 gerçekleşeceği, burada S dediğimiz (özensizlik için) bir eşik tarafından belirlenir. S ne kadar küçük olursa, 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 (Tablo 29.1). Bu örnekte, S eşiği 5 olarak ayarlanmıştır ve dört CPU'nun her birinde yerel sayaçlarını $L_1 \dots L_4$ güncelleyen threadler vardır. Küresel sayaç değeri (G) de izlemeye gösterilir ve zaman aşağı doğru artar. Her zaman adımında, bir yerel sayaç arttırılabilir; yerel değer S eşğine ulaşırsa, yerel değer global sayaca aktarılır ve yerel sayaç sıfırlanır.

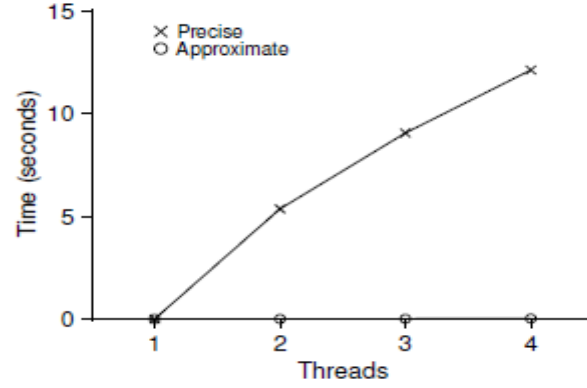
Şekil 29.5'teki alt satır (sayfa 6'da 'Yaklaşık' olarak etiketlenmiştir), S eşiği 1024 olan yaklaşık sayaçların performansını göstermektedir. Performans mükemmel; Sayacı dört işlemcide dört milyon kez güncellemek için geçen süre, bir işlemcide bir milyon kez güncellemek için geçen süreden neredeyse hiç yüksek değil.

```

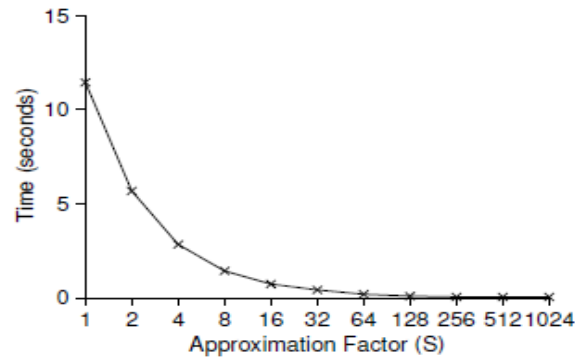
1  typedef struct __counter_t {
2      int          global;          // global count
3      pthread_mutex_t glock;        // global lock
4      int          local[NUMCPUS]; // per-CPU count
5      pthread_mutex_t llock[NUMCPUS]; // ... and locks
6      int          threshold;       // update frequency
7  } counter_t;
8
9  // init: record threshold, init locks, init values
10 //      of all local counts and global count
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: usually, just grab local lock and update
23 // local amount; once local count has risen 'threshold',
24 // grab global lock and transfer local values to it
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: just return global amount (approximate)
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; // only approximate!
45 }

```

Şekil 29.4: Yaklaşık Sayaç Uygulaması



Şekil 29.5: Geleneksel ve Yaklaşık Sayaçların Performansı



Şekil 29.6: Yaklaşık Sayaçların Ölçeklendirilmesi

Şekil 29.6, eşik değerinin önemini göstermektedir S, her biri sayacı dört cpu'da 1 milyon kez artıran dört thread ile. S düşükse, performans düşüktür (ancak genel sayım her zaman oldukça doğrudur); S yüksekse, performans mükemmeldir, ancak genel sayım geride kalır (neredeyse CPU sayısı S ile çarpılır). Bu doğruluk / performans değiş tokuşu, yaklaşık sayaçların etkinleştirdiği şeydir.

Yaklaşık bir sayacın kaba bir versiyonu Şekil 29.4'te (sayfa 5) bulunmaktadır. Nasıl çalıştığını daha iyi anlamak için okuyun veya daha iyisi, bazı deneylerde kendiniz çalıştırın.

İPUCU: DAHA FAZLA EŞ ZAMANLILIK GEREKLİ OLARAK DAHA HIZLI DEĞİLDİR

Tasarladığınız şema çok fazla ek yük getiriyorsa (örneğin, kilitleri bir kez yerine sık sık alıp bırakarak), daha eşzamanlı olması 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

Daha sonra bağlantılı liste olan daha karmaşık bir yapıyı inceliyoruz. Bir kez daha temel bir yaklaşımla başlayalım. Basitlik için, böyle bir listenin sahip olacağı bariz rutinlerden bazılarını atlayacağız ve yalnızca eşzamanlı eklemeye odaklanacağız; arama, silme vb. Hakkında düşünmesi için okuyucuya bırakacağız. Şekil 29.7, bu temel 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 çalışma, bu tür nadiren alınan kod yollarında çok büyük bir hata oranının (yaklaşık %40) bulunduğunu ortaya çıkardı (aslında bu gözlem, tüm bellek arızalı yolları bir bilgisayardan kaldırdığımız kendi araştırmamızın bazılarını ateşledi. Linux dosya sistemi, daha sağlam bir sistemle sonuçlanır [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 aramanın bir kısmının aslında kilitlenmesi gerekmez; malloc()'un kendisinin thread bakımından güvenli olduğunu varsayarsak, her thread 'in onu yarış koşulları veya diğer eşzamanlılık hataları endişesi olmadan çağırabilir. Yalnızca paylaşılan liste güncellenirken bir kilidin tutulması gerekir. Bu değişikliklerin ayrıntıları için bkz. Şekil 29.7.

Arama yordamına 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 // basic node structure
2 typedef struct __node_t {
3     int                key;
4     struct __node_t    *next;
5 } node_t;
6
7 // basic list structure (one used per list)
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; // fail
25     }
26     new->key = key;
27     new->next = L->head;
28     L->head = new;
29     pthread_mutex_unlock(&L->lock);
30     return 0; // success
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; // success
40         }
41         curr = curr->next;
42     }
43     pthread_mutex_unlock(&L->lock);
44     return -1; // failure
45 }

```

Şekil 29.7: Eşzamanlı 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     // synchronization not needed
8     node_t *new = malloc(sizeof(node_t));
9     if (new == NULL) {
10         perror("malloc");
11         return;
12     }
13     new->key = key;
14
15     // just lock critical section
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; // now both success and failure
35 }

```

Şekil 29.8: Eşzamanlı Bağlantılı Liste: Yeniden Yazıldı

Bağlantılı Listeleri Ölçeklendirme

Yine temel bir eşzamanlı bağlantılı listeye sahip olsakda, bir kez daha özellikle iyi ölçeklenmediği bir durumdayız. Araştırmacıların bir liste içinde daha fazla eş zamanlılık sağlamak için keşfettikleri bir teknik, elle kilitleme (a.k.a. kilit bağlantısı) [MS04] olarak adlandırılan bir tekniktir.

Fikir oldukça basit. Tüm liste için tek bir kilide sahip olmak yerine, bunun yerine listenin düğümü başına bir kilit eklersiniz. Listeyi geçerken, kod önce bir sonraki düğümün kilidini alır ve ardından geçerli düğümün kilidini serbest bırakır (bu, adı elden ele geçirir).

İPUCU: KİLİTLERE VE KONTROL AKIŞINA DİKKAT EDİN

Eşzamanlı kodda ve aynı zamanda yararlı olan genel bir tasarım ipucu başka bir yerde, bir işlevin yürütülmesini durduran işlev dönüşlerine, çıkışlarına veya diğer benzer hata koşullarına yol açan kontrol akışı değişikliklerine karşı dikkatli olunmalıdır. Çünkü birçok fonksiyon bir kilit elde etmekle başlayacak, tahsis etmek biraz bellek veya diğer benzer durum bilgisi olan işlemler yaparken, Hatalar ortaya çıkarsa, kodun geri dönmeden önce tüm durumu geri alması gerekir ki bu hataya açıktır. Bu nedenle, bu modeli en aza indirmek 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 thread 'de bile, birden çok devam eden geçişe izin vererek etkinleştirilen eşzamanlılığın, tek bir kilidi kapmak, bir işlem gerçekleştirmek ve onu serbest bırakmaktan daha hızlı olması pek olası değildir. Belki de bir tür melez (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. Bir sıra için, bunu anlayabileceğinizi varsayarak bu yaklaşımı atlayacağız.

Bunun yerine, Michael ve Scott [MS98] tarafından tasarlanan biraz daha eşzamanlı bir sıraya bakacağız. Bu sıra için kullanılan veri yapıları ve kod aşağıdaki sayfada Şekil 29.9'da bulunmaktadır.

Bu kodu dikkatlice incellerseniz, biri sıranın başı, diğeri kuyruk için olmak üzere iki kilit olduğunu fark edeceksiniz. Bu iki kilidin amacı, enqueue ve dequeue işlemlerinin eşzamanlılığını sağlamaktır. Genel durumda, kuyruğa alma yordamı yalnızca kuyruk kilidine erişecek ve yalnızca baş kilidini kuyruktan çıkaracaktır.

Michael ve Scott tarafından kullanılan bir numara, sahte bir düğüm eklemektir (kuyruk başlatma kodunda tahsis edilmiştir); bu kukla, baş ve kuyruk işlemlerinin ayrılmasını sağlar. Kodu inceleyin veya daha iyisi yazın, nasıl derinlemesine çalıştığını anlamak için çalıştırın ve ölçün.

Kuyruklar genellikle çok thread içeren uygulamalarda kullanılır. Yine de burada kullanılan sıra türü (yalnızca kilitlerle) genellikle bu tür programların ihtiyaçlarını tam olarak karşılamaz. Sıranın boş veya aşırı dolu olması durumunda bir thread 'in beklemesini sağlayan daha eksiksiz geliştirilmiş sınırlı bir sıra, koşul değişkenleri ile ilgili bir sonraki bölümdeki yoğun çalışmamızın 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; // queue was empty
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 Eş Zamanlı 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: Bir Eşzamanlı Hash Tablosu

29.4 Eşzamanlı Hash Tablosu

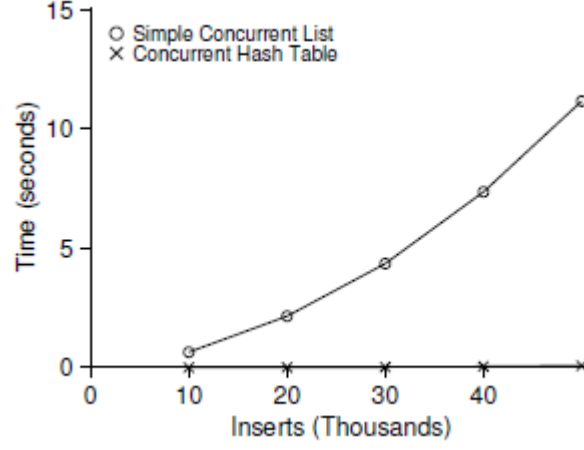
Tartışmamızı basit ve yaygın olarak uygulanabilir bir eşzamanlı veri yapısı olan karma tablo ile sonlandırıyoruz. Yeniden boyutlandırılmayan basit bir karma tabloya odaklanacağız; Yeniden boyutlandırmayı işlemek için okuyucuya bir alıştırma olarak bıraktığımız biraz daha çalışma gerekiyor (üzgünüz!).

Bu eşzamanlı karma tablo (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, karma kova başına bir kilit kullanmasıdır (her biri bir liste ile temsil edilir). Bunu yapmak, birçok eşzamanlı işlemin gerçekleşmesini sağlar.

Şekil 29.11, karma tablonun eşzamanlı güncelleştirmeler altındaki performansını göstermektedir (dört işlemcili aynı imac'te dört thread 'in her birinden 10.000 ila 50.000 eşzamanlı güncelleştirme). Karşılaştırma uğruna, bağlantılı bir listenin (tek bir kilit) 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 kuyruklara ve son olarak her yerde bulunan ve yoğun olarak kullanılan karma tabloya eşzamanlı veri yapılarının bir örneklemesini sunduk. Yol boyunca birkaç önemli ders aldık: kontrol akışı değişikliklerinin etrafındaki kilitlerin alınmasına ve serbest bırakılmasına dikkat etmek; Daha fazla eşzamanlılık sağlamanın performansı artırması gerekmediğini; performans sorunlarının yalnızca var olduklarında giderilmesi gerektiğini.



Şekil 29.11: Karma Tabloların Ölçeklendirilmesi

Erken optimizasyondan kaçınmanın bu son noktası, performans odaklı herhangi bir geliştiricinin merkezinde yer alır; Bunu yapmak, uygulamanın genel performansını iyileştirmezse, bir şeyi daha hızlı hale getirmenin bir değeri yoktur.

Tabii ki, sadece yüksek performanslı yapıların yüzeyini çizdik. Daha fazla bilgi ve diğer kaynaklara bağlantılar için Moir ve Shavit'in mükemmel anketine bakın [MS04]. Özellikle, diğer yapılarla (B ağaçları gibi) ilgilenebilirsiniz; bu bilgi için en iyi seçeneğiniz bir veritabanı sınıfıdır. Geleneksel kilitleri hiç kullanmayan teknikleri de merak ediyor olabilirsiniz; bu tür engellenmeyen veri yapıları, ortak eşzamanlılık hataları bölümünde tadacağımız bir şeydir, ancak açıkçası bu konu, bu mütevazı kitapta mümkün olandan daha fazla çalışma gerektiren tüm bir bilgi alanıdır. Dilerseniz kendi başınıza daha fazlasını öğrenin (her zaman olduğu gibi!).

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, doğru bir kilit oluşturmanız muhtemeldir; Daha sonra performans sorunları yaşadığınızı fark ederseniz, onu hassaslaştırabilir, böylece yalnızca gerektiğinde hızlı hale getirebilirsiniz. Knuth'un ünlü bir şekilde belirttiği gibi, "Erken optimizasyon bütün kötülüklerin anası."

Birçok işletim sistemi, SUNOS ve Linux dahil olmak üzere çok 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 sistemlerinin norm haline gelmesiyle, bir seferde çekirdekte yalnızca tek bir etkin thread 'e izin verilmesi bir performans darboğazı haline geldi. Böylece, nihayet bu sistemlere iyileştirilmiş eşzamanlılığın optimizasyonunu eklemenin zamanı gelmişti. Linux'ta daha basit bir yaklaşım benimsendi: bir kilidi birçok kilitle değiştirin. Sun içinde daha radikal bir karar alındı: Solaris olarak bilinen ve ilk günden itibaren eşzamanlılığı daha temelden içeren yepyeni bir işletim sistemi oluşturun. Bu büyüleyici sistemler hakkında daha fazla bilgi için Linux ve Solaris çekirdek kitaplarını okuyun [BC05, MM 00].

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 Zeldovich . 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. Available: <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 through every single patch by hand in order to understand what they did.

[MS98] “Nonblocking Algorithms and Preemption-safe Locking on by Multiprogrammed Sharedmemory Multiprocessors. ” M. Michael, M. Scott. Journal of Parallel and Distributed Computing, 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 missing some 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 memory before 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 onunla biraz deneyim kazanmak oldukça değerlidir.

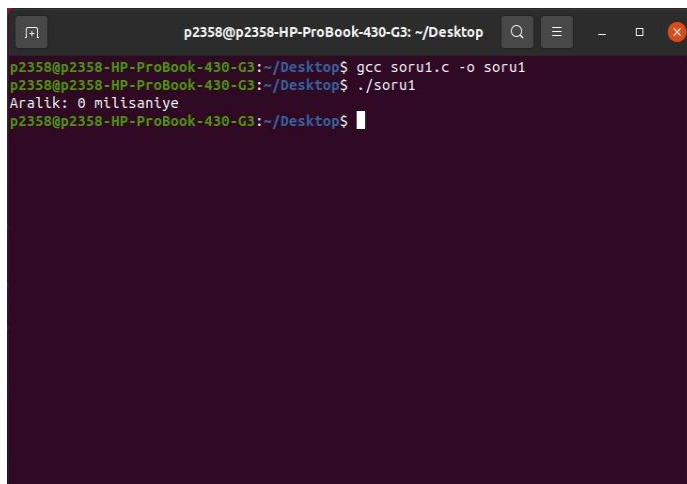
Sorular

1. Bu bölümdeki ölçümleri yeniden yaparak başlayacağız. Programınızdaki 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() fonksiyonu bir sistem çağrısıdır ve genellikle bir sistemin güncel zamanını ölçmek için kullanılır. Ölçebileceği en küçük zaman aralığı, sistemin zaman dilimini belirleyen bir parametredir ve genellikle bir mikrosaniye cinsinden ayarlanır.

x86 işlemcilerinde bulunan döngü sayacı (RDTSC talimatı) da bir zamanlayıcı olarak kullanılabilir. Ancak bu sayaç, işlemcinin çalışma hızına bağlı olarak değişebileceğinden, gettimeofday() fonksiyonundan daha az hassas olabilir.

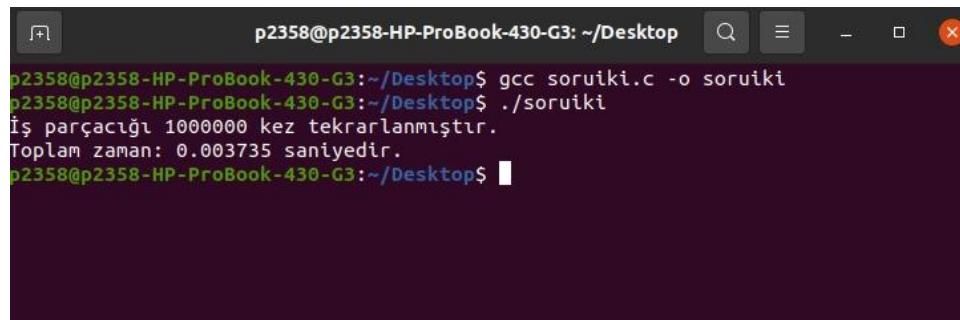
```
1 #include <stdio.h>
2 #include <sys/time.h>
3
4 int main() {
5     struct timeval start, end;
6
7     gettimeofday(&start, NULL);
8
9     gettimeofday(&end, NULL);
10
11     long int elapsed_time = (end.tv_sec - start.tv_sec) * 1000 + (end.tv_usec - start.tv_usec) / 1000;
12
13     printf("Aralık: %ld milisaniye\n", elapsed_time);
14
15     return 0;
16 }
17 }
```



```
p2358@p2358-HP-ProBook-430-G3: ~/Desktop
p2358@p2358-HP-ProBook-430-G3:~/Desktop$ gcc soru1.c -o soru1
p2358@p2358-HP-ProBook-430-G3:~/Desktop$ ./soru1
Aralık: 0 milisaniye
p2358@p2358-HP-ProBook-430-G3:~/Desktop$
```


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

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main() {
5     clock_t start = clock();
6
7
8     int num_iterations = 1000000;
9
10
11     for (int i = 0; i < num_iterations; i++) {
12         int counter = 0;
13         counter++;
14     }
15
16
17     clock_t end = clock();
18
19
20     double time_elapsed = (end - start) / (double)CLOCKS_PER_SEC;
21
22     printf("İş parçacığı %d kez tekrarlanmıştır.\n", num_iterations);
23     printf("Toplam zaman: %f saniyedir.\n", time_elapsed);
24
25
26     return 0;
27 }
28
29
```



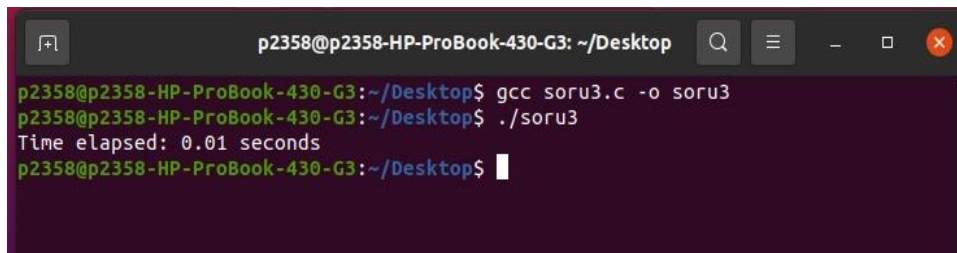
```
p2358@p2358-HP-ProBook-430-G3: ~/Desktop
p2358@p2358-HP-ProBook-430-G3:~/Desktop$ gcc soruiki.c -o soruiki
p2358@p2358-HP-ProBook-430-G3:~/Desktop$ ./soruiki
İş parçacığı 1000000 kez tekrarlanmıştır.
Toplam zaman: 0.003735 saniyedir.
p2358@p2358-HP-ProBook-430-G3:~/Desktop$
```

Bu koda, öncelikle başlangıç zamanı kaydediliyor thread sayısı tutuluyor, thread sayısı arttıkça sayacı artıracak döngü yazılıyor bitiş zamanı kaydediliyor bitiş ve başlangıç zamanı farkı hesaplanıyor ve ekrana yazdırılıyor.

Bu ölçümler kullanılan sistemdeki CPU sayısının ne olursa olsun aynı sonuçları verir. Ancak, eğer sistemde birden fazla CPU varsa ve thread ler bu CPU'lara dağıtılsa, işlemlerin daha hızlı tamamlanması sağlanır. Bu durumda, ölçümlerinizin süresi daha kısa olabilir.

3. Ardından, özensiz sayacın bir versiyonunu oluşturun. İş parçacığı sayısı eşiğin yanı sıra değiştikçe performansını bir kez daha ölçün. Sayılar bölümde gördüklerinizle eşleşiyor mu?

```
1#include <stdio.h>
2#include <time.h>
3#include <sys/time.h>
4
5int main()
6{
7
8    struct timeval start;
9    gettimeofday(&start, NULL);
10
11
12    int threshold = 1000;
13    int counter = 0;
14    for (int i = 0; i < 1000000; i++)
15    {
16
17        if (counter >= threshold)
18        {
19            counter = 0;
20        }
21
22        counter++;
23    }
24
25    struct timeval end;
26    gettimeofday(&end, NULL);
27
28
29    double time_elapsed = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1000000.0;
30    printf("Time elapsed: %.2f seconds\n", time_elapsed);
31
32    return 0;
33 }
```

A terminal window with a dark purple background. The title bar shows 'p2358@p2358-HP-ProBook-430-G3: ~/Desktop'. The prompt is 'p2358@p2358-HP-ProBook-430-G3:~/Desktop\$'. The user enters 'gcc soru3.c -o soru3'. The prompt changes to 'p2358@p2358-HP-ProBook-430-G3:~/Desktop\$./soru3'. The output is 'Time elapsed: 0.01 seconds'. The prompt returns to 'p2358@p2358-HP-ProBook-430-G3:~/Desktop\$'.

Bu koddaki, gettimeofday() fonksiyonu başlangıç ve bitiş zamanını ölçmek için kullanılmaktadır. Thread sayısı arttıkça, sayacın değeri artırılmakta ve bir eşik değeri belirlenmektedir. Eğer sayacın değeri bu eşik değerini aşarsa, sayacın sıfırlanmaktadır. Son olarak, başlangıç ve bitiş zamanı arasındaki fark alınarak ölçümlerin ne kadar sürdüğü bulunup ekrana yazdırılmıştır. Kod çalıştırıldığında ise başlangıç ve bitiş zamanı arasındaki fark görülmüştür.

4. Elden ele kilitlemeyi kullanan bağlantılı bir listenin sürümünü oluşturun [MS04], bölümde belirtildiği gibi. Önce gazeteyi okumalısın. nasıl çalıştığını anlamak ve sonra uygulamak. Ölçmek onun performansı. Elden ele liste, bölümde gösterildiği gibi standart bir listeden ne zaman daha iyi çalışır?

--Hiçbirzaman

5. B ağacı veya biraz daha ilginç başka bir yapı gibi favori veri yapınızı seçin. Uygulayın ve tek bir kilit gibi basit bir kilitleme stratejisiyle başlayın. Eşzamanlı thread sayısı arttıkça performansını ölçün.

Bir B ağacı, ikili arama ağacı veri yapısının bir varyasyonudur ve verileri anahtarlarına göre sıralı bir şekilde saklar. B ağacı, her düğümün birden fazla anahtar içerebilen bir veri yapısıdır ve bu nedenle, ikili arama ağacından daha yüksek bir eşleşme oranı sağlar.

Eşzamanlı thread sayısı arttıkça B ağacının performansı da değişebilir. Basit bir kilitleme stratejisi kullanıldığında, thread 'lerin aynı anda aynı B ağacına erişme çalışmalarını engellediğinden, bu tür bir veri yapısı çok thead 'li bir ortamda daha iyi performans gösterebilir. Ancak bu kilitleme stratejisi, kilitleme işlemleri nedeniyle daha yavaş bir performans gösterebilir.

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

Daha ilginç bir kilitleme stratejisi, veri yapısının farklı bölümlerine ayrı ayrı kilitleme uygulayarak daha hassas bir kontrol sağlayabilir. Bu sayede, thread 'lerin erişimleri daha etkin bir şekilde yönetilebilir ve bu da veri yapısının performansını artırabilir. Ancak bu kilitleme stratejisi, daha fazla "kilitle" değişkeni kullandığı için daha fazla kaynak tüketebilir. Bu nedenle, hangi kilitleme stratejisinin daha iyi performans göstereceği, kullanım koşullarına bağlı olarak değişebilir.