29

Kilit Tabanlı Eşzamanlı Veri Yapıları

Kilitlerin ötesine geçmeden önce, önce bazı yaygın veri yapılarında kilitlerin nasıl kullanılacağını açıklayacağız. Bir veri yapısına kilitler ekleyerek iş parçacıkları tarafından kullanılabilir hale getirilmesi, yapı **iş parçacığını (thread safe)güvenli** hale getirir. Tabii ki, bu tür kilitlerin tam olarak nasıl eklendiği, veri yapısının hem doğruluğunu hem de performansını belirler . Ve böylece, meydan okumamız:

CRUX: HOW TO ADD LOCKS TO DATA STRUCTURES
Belirli bir veri yapısı verildiğinde, doğru çalışmasını sağlamak için ona nasıl kilitler eklemeliyiz ? Ayrıca, veri yapısının yüksek performans göstermesini ve birçok iş parçacığının yapıya aynı anda, yani aynı anda erişmesini sağlayacak **şekilde** kilitleri nasıl ekleyebiliriz?

Tabii ki, tüm veri yapılarını veya eşzamanlılık eklemek için tüm yöntemleri kapsamakta zorlanacağız , çünkü bu, yıllardır çalışılan bir konudur ve (kelimenin tam anlamıyla) binlerce araştırma makalesi yayınlanmıştır. Bu nedenle, gerekli düşünme türüne yeterli bir giriş yapmayı ve sizi daha fazlası için bazı iyi malzeme kaynaklarına yönlendirmeyi umuyoruz. kendi başınıza soruşturma. 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. Şekil 29.1'de eş zamanlı olmayan basit bir sayaç tanımlıyoruz.

Basit Ama Ölçeklenebilir Değil

Gördüğünüz gibi, senkronize edilmemiş sayaç, uygulanması için çok az miktarda kod gerektiren önemsiz bir veri yapısıdır. Şimdi bir sonraki zorluğumuz var: Bu kod iş parçacığını nasıl güvenli 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: **Kilitsiz 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 : veri yapısını manipüle eden bir rutini çağırırken akıtılan ve çağrıdan dönerken serbest bırakılan tek bir kilit ekler . Bu şekilde , monitörlerle [BH73] oluşturulmuş bir veri yapısına benzer; burada kilitler, nesne yöntemlerini çağırdığınızda ve geri döndüğünüzde otomatik olarak alınır ve serbest bırakılır.

Bu noktada, çalışan bir eşzamanlı veri yapınız vardır. Sahip olabileceğiniz olasılık performanstır. Veri yapınız çok yavaşsa , sadece tek bir kilit eklemekten daha fazlasını yapmak istersiniz; bu tür optimizasyonlar, gerekirse, 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 maliyetlerinize performans maliyetlerini azaltmak için, her iş parçacığının tek bir paylaşılan sayacı sabit sayıda güncelleştirdiği bir karşılaştırma ölçütü çalıştırırız; daha sonra iş parçacığı sayısını değiştiririz. Şekil 29.5, bir ila dört iş parçacığı etkin olacak şekilde geçen toplam süreyi göstermektedir; her iş parçacığı sayacı bir milyon kez güncelleştirir. Bu deneme, dört Intel 2,7 GHz i5 CPU'ya sahip bir iMac üzerinde gerçekleştirildi; Daha fazla CPU etkinken, birim zaman başına daha fazla toplam iş yapmayı umuyoruz.

Şekildeki en üst satırdan ("Hassas" etiketli), senkronize sayacın performansının zayıf ölçeklendiğini görebilirsiniz . Tek bir iş parçacığı milyon sayaç güncelleştirmesini kısa bir sürede (kabaca 0,03 saniye) tamamlayabilirken, her biri sayacı bir milyon kez güncelleyen iki iş parçacığının olması büyük bir yavaşlamaya yol açar (5 saniyeden fazla sürer!). Sadece daha fazla iplikle daha da kötüleşir.

```
1  typedef Yapı___counter_t {
2        Int              value;
3        pthread_mutex_t kilit;
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 artışı(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, çok uçlu işlemcilerde iş parçacıklarının, tek iş parçacığının bir tanesinde olduğu kadar hızlı bir şekilde tamamlandığını görmek istersiniz . Bu amaca ulaşmak **mükemmel ölçeklendirme (perfect scaling)** olarak adlandırılır ; Daha fazla iş yapılmasına rağmen, paralel olarak yapılır ve bu nedenle görevi tamamlamak için harcanan süre artmaz.

Ölçeklenebilir Sayım

Şaşırtıcı bir şekilde, araştırmacılar yıllardır daha ölçeklenebilir ülkelerin nasıl oluşturulacağını araştırdılar [MS04]. Daha da şaşırtıcı olanı, işletim sistemi performans analizindeki son çalışmaların gösterdiği gibi ölçeklenebilir ülkelerin önemli olmasıdır [B + 10]; ölçeklenebilir sayım olmadan, Linux'ta çalışan bazı iş yükleri çok çekirdekli makinelerde ciddi ölçeklenebilirlik sorunlarından mustariptir .

Bu soruna saldırmak için birçok teknik geliştirilmiştir . Yaklaşık **sayaç** [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. Özellikle, dört CPU'lu bir makinede, dört tane vardır

Saat	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 → 0	1	3	4	5 (L_1'den itibaren)
7	0	2	4	5 → 0	10 (L_4'ten itibaren)

Şekil 29.3: **Yaklaşık Sayaçların İzini Sürme**

yerel sayaçlar ve bir küresel sayaç. Bu sayaçlara ek olarak, kilitler de vardır: her yerel sayaç [1] için bir tane ve küresel sayaç için bir tane .

Yaklaşık saymanın temel fikri aşağıdaki gibidir. Bir iş parçacığı olduğunda
Belirli bir çekirdek üzerinde çalışan sayacı artırmak ister, yerel sayacını arttırır; bu yerel sayaca erişim, corre-sponding yerel kilidi aracılığıyla senkronize edilir . Her CPU'nun kendi yerel sayacı olduğundan , CPU'lardaki iş parçacıkları yerel ülkeleri çekişme olmadan güncelleyebilir ve böylece sayacın güncellenmesi ölçeklenebilir.

Ancak, global sayacı güncel tutmak için (bir iş parçacığının değerini okumak istemesi durumunda), yerel değerler, global kilit alınarak ve yerel sayacın değeri artırılarak periyodik olarak global sayaca aktarılır ; yerel sayaç daha sonra sıfırlanır.

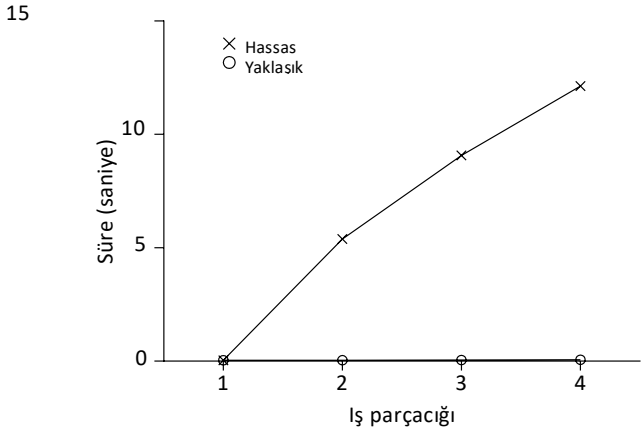
Bu yerelden küresele aktarımın ne sıklıkta gerçekleştiği, harman eski bir S tarafından belirlenir. S ne kadar küçük olursa, sayaç yukarıdaki ölçeklenebilir olmayan sayaç gibi o kadar çok davranır ; S ne kadar büyükse, sayaç o kadar ölçeklenebilir olur , ancak küresel değer gerçek sayımdan o kadar uzak olabilir. Kesin bir değer elde etmek için tüm yerel kilitleri ve küresel kilidi (belirli bir sırayla, kilitlenmeyi önlemek için) basit bir şekilde elde edebilirsiniz, ancak bu ölçeklenebilir değildir.

Bunu açıklığa kavuşturmak için, bir örneğe bakalım (Şekil 29.3). Bu örnekte , eşik S 5 olarak ayarlanır ve yerel sayaçlarını güncelleyen dört CPU'nun her birinde iş parçacıkları vardır L_1 ... L_4. Küresel sayaç değeri
(G) de izde gösterilir, zaman aşağı doğru artar. Her zaman adımında, yerel bir sayaç artırılabilir; yerel değer S eşiğine ulaşırsa, yerel değer genel sayaca aktarılır ve yerel sayaç sıfırlanır.

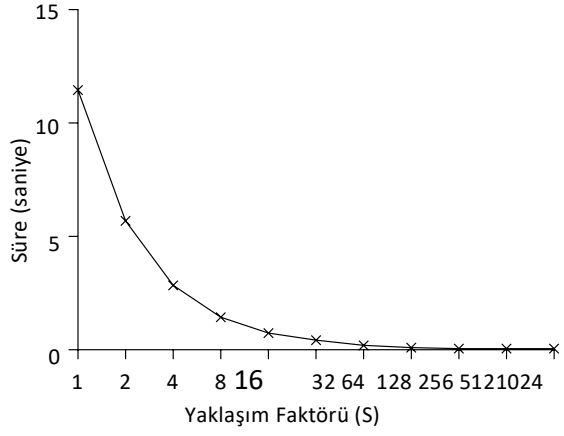
Şekil 29.5'teki alt çizgi (sayfa 6'da 'Yaklaşık' olarak etiketlenmiştir), S eşiği 1024 olan yaklaşık sayaçların performansını gösterir. Performans ı mükemmel; sayacı dört işlemcide dört milyon kez güncellemek için geçen süre , bir işlemcide bir milyon kez güncellemek için geçen süreden daha yüksek değildir.

[1] Yerel kilitlere ihtiyacımız var çünkü her çekirdekte birden fazla iş parçacığı olabileceğini varsayıyoruz. Bunun yerine, her çekirdekte yalnızca bir iş parçacığı çalıştırılırsa, yerel kilit gerekmez.

```
typedef Yapı counter_t {
    int             global;         // global count
    pthread_mutex_t glock;          // global lock
    int             local[NUMCPUS]; // per-CPU count
    pthread_mutex_t llock[NUMCPUS]; // ... and locks
    int             threshold;      // update frequency
} counter_t;

// init: record threshold, init locks, init values
//       of all local counts and global count
void init(counter_t *c, int threshold) {
    c->threshold = threshold;
    c->global = 0;
    pthread_mutex_init(&c->glock, NULL);
    int i;
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;
        pthread_mutex_init(&c->llock[i], NULL);
    }
}

// update: usually, just grab local lock and update
// local amount; once local count has risen 'threshold',
// grab global lock and transfer local values to it
void update(counter_t *c, int threadID, int amt) {
    int cpu = threadID % NUMCPUS;
    pthread_mutex_lock(&c->llock[cpu]);
    c->local[cpu] += amt;
    if (c->local[cpu] >= c->threshold) {
        // transfer to global (assumes amt>0)
        pthread_mutex_lock(&c->glock);
        c->global += c->local[cpu];
        pthread_mutex_unlock(&c->glock);
        c->local[cpu] = 0;
    }
    pthread_mutex_unlock(&c->llock[cpu]);
}

// get: just return global amount (approximate)
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val; // only approximate!
}
```



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



Şekil 29.6: **Yaklaşık Sayaçları Ölçeklendirme**

Şekil 29.6, her biri dört CPU'da sayacı 1 milyon kez artıran dört iş parçacığı ile *S* eşik değerinin önemini göstermektedir. *S* düşükse, performans zayıftır (ancak küresel sayı her zaman oldukça doğrudur); *S* yüksekse, performans mükemmeldir, ancak genel sayı gecikir (en fazla CPU sayısının S ile çarpımı). Bu doğruluk/performans dengesi , yaklaşık sayaçların etkinleştirdiği şeydir.

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

TIP: MORE CONCURRENCY ISN'T NECESSARILY FASTER

Tasarladığınız şema çok fazla ek yük ekliyorsa (örneğin, kilitleri bir kez yerine sık sık alıp serbest bırakarak), daha eşzamanlı olması önemli olmayabilir. Basit şemalar, özellikle pahalı rutinleri nadiren kullanıyorlarsa, iyi çalışma eğilimindedir. Daha fazla kilit ve karmaşıklık eklemek sizin düşüşünüz olabilir. Tüm bunları gerçekten bilmenin bir yolu var: her iki alternatifi de (basit ama daha az eşzamanlı ve karmaşık ama daha eşzamanlı) inşa edin ve nasıl yaptıklarını ölçün. Sonunda,

29.2 Eşzamanlı Bağlı Listeler

Daha sonra daha karmaşık bir yapıyı, bağlantılı listeyi inceliyoruz. Bir kez daha temel bir yaklaşımla başlayalım. Basitlik için, böyle bir listenin sahip olacağı bariz rutinlerden bazılarını atlayacağız ve sadece aynı fikirde olan eklemeye odaklanacağız; arama, silme vb. hakkında düşünmeyi okuyucuya bırakacağız . Şekil 29.7, bu temel veri yapısının kodunu göstermektedir.

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

Bu tür istisnai kontrol akışının oldukça hataya eğilimli olduğu gösterilmiştir; Linux çekirdek yamaları üzerine yapılan yakın tarihli bir çalışma, nadiren alınan kod yollarında hataların büyük bir kısmının (yaklaşık%40) bulunduğunu bulmuştur (aslında, bu gözlem kendi araştırmamızın bir kısmını tetikledi, bellek arızası yapan tüm yolları Linux dosya sisteminden kaldırdık ve daha sağlam bir sistem [S+11]).

Bu nedenle, bir zorluk: eşzamanlı ekleme altında yeniden düzeltmek için ekleme ve arama rutinlerini yeniden yazabilir miyiz, ancak hata yolunun kilidi açmak için çağrıyı eklememizi gerektirdiği durumlardan kaçınabilir miyiz?

Bu durumda cevap evet. Özellikle, kodu biraz yeniden düzenleyebiliriz, böylece kilit ve serbest bırakma yalnızca ekleme kodundaki gerçek kritik bölümü çevreler ve arama kodunda ortak bir çıkış yolu kullanılır. İlki işe yarar çünkü kesici ucun parçasının aslında kilitlenmesi gerekmez ; malloc() öğesinin kendisinin iş parçacığı açısından güvenli olduğunu varsayarsak, her iş parçacığı yarış koşulları veya diğer eşzamanlılık hatalarından endişe duymadan onu çağırabilir. Yalnızca paylaşılan listeyi güncellerken bir kilidin tutulması gerekir. Bu değişikliklerin ayrıntıları için Şekil 2 9.8'e bakın.

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

```
// basic node structure
typedef struct __node_t {
    int                 key;
    struct __node_t         *next;
} node_t;

// basic list structure (one used per list)
typedef struct   list_t {
    node_t                *head;
    pthread_mutex_t     lock;
} list_t;

13  void List_Init(list_t *L) {
14      L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}

int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        pthread_mutex_unlock(&L->lock);
        return -1; // fail
    }
    new->key  = key;
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
    return 0; // success
}

int List_Lookup(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            pthread_mutex_unlock(&L->lock);
            return 0; // success
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return -1; // failure
}
```

Figure 29.7: **Concurrent Linked List**

```
  }
```

Şekil 29.7: **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
```

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

Bağlı Listeleri Ölçeklendirme

Yine temel bir eşzamanlı bağlantılı listemiz olmasına rağmen, bir kez daha özellikle iyi ölçeklenmediği bir durumdayız. Araştırmacıların bir liste içinde daha fazla eşzamanlılık sağlamak için araştırdıkları bir teknik, **elden ele kilitleme(hand-over-hand locking)** (diğer adıyla **kilit bağlantısı**) [MS04] adı verilen bir şeydir. Fikir oldukça basit. Listenin tamamı için tek bir kilit yerine, listenin düğümü başına bir kilit eklersiniz. Listede gezinirken, kod önce bir sonraki düğümün kilidini yakalar ve ardından geçerli olanı serbest bırakır düğümün kilidi (bu da el ele ismine ilham verir).

TIP: BE WARY OF LOCKS AND CONTROL FLOW
Eşzamanlı kodda ve başka yerlerde yararlı olan genel bir tasarım ipucu, işlev dönüşlerine, çıkışlara veya bir işlevin yürütülmesini durduran diğer benzer hata koşullarına yol açan kontrol akışı değişikliklerine karşı dikkatli olmaktır. Birçok işlev bir kilit alarak, bazı bellekler ayırarak veya diğer benzer durum bilgisi olan işlemleri yaparak başladığından, hatalar ortaya çıktığında, kodun geri dönmeden önce tüm durumu geri alması gerekir, bu da hataya açıktır. Bu nedenle, bu kalıbı en aza indirmek için ev yapılandırmak içinde 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 kilidin bir pir olayı haline getirmesinden daha hızlı hale getirmek zordur, çünkü bir liste geçişinin her düğümü için kilitleri edinme ve serbest bırakma ek yükleri engelleyicidir. Çok büyük listeler ve çok sayıda iş parçacığı olsa bile, birden fazla devam eden geçişe izin vererek sağlanan eşzamanlılığın, tek bir kilidi yakalamaktan, bir işlem gerçekleştirmekten ve serbest bırakmaktan daha hızlı olması olası değildir. Belki de bir tür hibrid (her düğümde yeni bir kilit kaptığınız) araştırmaya değer olacaktır.

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 kuyruk için, bunu çözebileceğinizi varsayarak bu yaklaşımı atlarız.

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

Bu kodu dikkatlice incelerseniz, biri kuyruğun başı ve diğeri kuyruk için olmak üzere iki kilit olduğunu fark edeceksiniz. Bu iki kilidin amacı, kuyruk oluşturma ve kuyruktan çıkarma işlemlerinin eşzamanlılığını etkinleştirmektir. Genel durumda, kuyruk yordamı yalnızca kuyruk kilidine erişir ve yalnızca kafa kilidini sıraya alır.

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

Kuyruklar, çok iş parçacıklı uygulamalarda yaygın olarak kullanılır. Bununla birlikte, burada kullanılan kuyruk türü (sadece kilitlerle) genellikle bu tür programların ihtiyaçlarını tam olarak karşılamaz. Kuyruğun boş veya aşırı dolu olması durumunda bir iş parçacığının beklemesini sağlayan daha gelişmiş bir sınırlı kuyruk, koşul değişkenleri üzerine bir sonraki bölümde yoğun çalışmamızın konusudur. İzleyin !

```
1  typedef struct___    _t {
   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 }
```

Figure 29.9: **Michael and Scott Concurrent Queue**

```
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 }
```

Figure 29.10: **A Concurrent Hash Table**

29.4 Eşzamanlı Karma Tablo

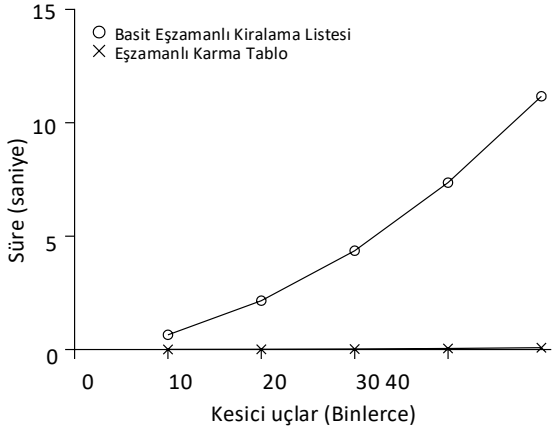
Tartışmamızı basit ve yaygın olarak uygulanabilir eşzamanlı bir veri yapısı olan karma tablo ile sonlandırıyoruz. Yeniden boyutlandırılmayan basit bir karma tabloya odaklanacağız; okuyucu için bir alıştırma olarak bıraktığımız yeniden boyutlandırmayı ele almak için biraz daha fazla çalışma gerekiyor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

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

Şekil 29.11 , eş zamanlı güncellemeler altında karma tablosunun performansını göstermektedir (dört CPU'lu aynı iMac'te , dört iş parçacığının her birinden 10.000 ila 50.000 eşzamanlı güncelleme). Ayrıca, karşılaştırma uğruna, bağlantılı bir listenin performansı (tek bir kilitle) gösterilmiştir. Grafikten de görebileceğiniz gibi, bu basit eşzamanlı karma tablo muhteşem bir şekilde ölçeklenir; bağlantılı liste, aksine, bunu yapmaz.

29.5 Özet

Sayaçlardan, listelere ve kuyruklara ve son olarak her yerde bulunan ve yoğun olarak kullanılan karma tabloya kadar eşzamanlı veri yapılarının bir örneklemesini kullanıma sunduk. Yol boyunca birkaç önemli ders aldık: kontrol akışı etrafındaki kilitlerin edinilmesine ve serbest bırakılmasına dikkat etmek



Şekil 29.11: **Karma Tabloları Ölçeklendirme**

Değişiklik; daha fazla eşzamanlılık sağlamanın mutlaka performansı artırmayacağı; performans sorunlarının ancak var olduklarında düzeltilmesi gerektiğini. **Erken optimizasyondan** kaçınmanın bu son noktası, performans odaklı herhangi bir geliştirici için merkezdir; Bunu yapmak, genel performansını artırmayacaksa, daha hızlı bir şey yapmanın bir değeri yoktur. uygulama.

Tabii ki, yüksek performanslı yapıların yüzeyini yeni çizdik. Daha fazla bilgi ve diğer kaynaklara bağlantılar için Moir ve Shavit'in mükemmel anketine bakın [MS04]. Özellikle, diğer yapılarda (B ağaçları gibi) etkileşime girebilirsiniz; bu bilgi için, bir veri tabanı sınıfı en iyi bahistir. Geleneksel kilitleri hiç kullanmayan teknikleri de merak ediyor olabilirsiniz; bu tür **engellemeyen veri** yapıları, yaygın eşzamanlılık hataları bölümünde tadacağımız bir şeydir, ancak açıkçası bu konu daha fazlasını gerektiren tüm bir bilgi alanıdır. Bu mütevazı kitapta mümkün olandan daha fazla çalışma. İsterseniz kendi başınıza daha fazla bilgi edinin (her zaman olduğu gibi!).

TIP: AVOID PREMATURE OPTIMIZATION (KNUTH'S LAW)

Eşzamanlı bir veri yapısı oluştururken, senkronize erişim sağlamak için tek bir büyük kilit eklemek olan en temel approach ile başlayın. Bunu yaparak, doğru bir kilit oluşturmanız muhtemeldir ; Daha sonra performans sorunlarından mustarip olduğunu tespit ederseniz, onu hassaslaştırabilir, böylece yalnızca gerekirse hızlı hale getirebilirsiniz. **Knuth** ünlüce belirttiği gibi, "Erken optimizasyon tüm kötülüklerin köküdür."

Birçok işletim sistemi, Sun OS ve Linux da dahil olmak üzere 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 bir seferde yalnızca tek bir aktif iş parçacığına izin vermek bir performans darboğazı haline geldi. Böylece, nihayet bu sistemlere geliştirilmiş eşzamanlılığın optimizasyonunu eklemenin zamanı gelmişti . Linux içinde daha basit bir yaklaşım benimsendi: bir kilidi birçok kilitle değiştirin. Su'nda daha radikal bir karar alındı: Solaris olarak bilinen, ilk günden itibaren eşzamanlılığı daha fazla temel nitelik içeren yepyeni bir işletim sistemi inşa etmek. Bu büyüleyici sistemler hakkında daha fazla bilgi için Linux ve Solaris çekirdek kitaplarını okuyun [BC05, MM00].

Referanslar

[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 Shared-memory 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)

In this homework, you'll gain some experience with writing concurrent code and measuring its performance. Learning to build code that performs well is a critical skill and thus gaining a little experience here with it is quite worthwhile.

Soru

1. Bu bölümdeki ölçümleri yeniden yaparak başlayacağız. Programınızdaki zamanı ölçmek için gettimeofday() çağrısını kullanın. Bu zamanlayıcı ne kadar doğru? Ölçebileceği en küçük aralık nedir? Çalışmalarında güven kazanın, çünkü sonraki tüm sorularda ona ihtiyacımız olacak . Ayrıca, rdtsc talimatı aracılığıyla x86'da bulunan döngü sayacı gibi diğer zamanlayıcılara da bakabilirsiniz.

he gettimeofday() işlevi, bir program içindeki zamanı ölçmenin bir yoludur. Unix döneminden bu yana geçen saniye ve mikrosaniye sayısı olarak ifade edilen geçerli zamanı alan bir sistem çağrısıdır (1 Ocak 1970'te 00:00:00 UTC). Bu zamanlayıcı, sisteme bağlı olarak genellikle birkaç mikrosaniye içinde doğrudur.

gettimeofday() öğesinin ölçebileceği en küçük aralık bir mikrosaniyedir, çünkü işlev zamanı mikrosaniye cinsinden döndürür. Bununla birlikte, zamanlayıcının gerçek çözünürlüğü, temel alınan sistem donanımına ve işletim sistemine bağlı olarak çok daha kaba olabilir.

Zamanı daha doğru bir şekilde ölçmek için x86 döngü sayacı gibi diğer zamanlayıcıları kullanmak da mümkündür. RDTSC yönergesi, işlemcinin son sıfırlanmasından bu yana geçen saat döngülerinin sayısını sayan döngü sayacını okumak için kullanılabilir. Bu, zamanın daha doğru bir ölçümünü sağlayabilir, ancak yalnızca belirli işlemci türlerinde kullanılabilir ve diğer platformlara taşınabilir olmayabilir.

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

Eşzamanlı bir sayaç oluşturmak için, önce birden çok iş parçacığı tarafından erişilebilen paylaşılan bir sayaç değişkeni tanımlamamız gerekir. Bu, sayaca erişimin düzgün bir şekilde eşitlendiğinden ve her iş parçacığının doğru bir sayım aldığından emin olmak için bir muteks veya başka bir

eşitleme ilkeli kullanılarak yapılabilir.

Sayacı artırmanın ne kadar sürdüğünü ölçmek için, iş parçacıklarını başlatma ve artışları tamamlama arasındaki geçen süreyi ölçmek için gettimeofday() işlevi gibi bir zamanlayıcı kullanmamız gerekir. Daha sonra testi birden çok kez çalıştırmamız ve sayacı artırmak için gereken sürenin daha doğru bir ölçüsünü elde etmek için sonuçların ortalamasını almamız gerekir.

İş parçacıkları işletim sistemi tarafından zamanlanacağından ve aynı CPU'da çalışmayabileceğinden, sistemde kullanılabilen CPU sayısının ölçümleri etkilemesi gerekmez. Ancak, sistemde çok sayıda CPU varsa ve iş parçacıklarını bunlar arasında etkili bir şekilde dağıtabiliyorsa, birden çok iş parçacığının farklı CPU'larda aynı anda çalışmasına izin vererek artışları hızlandırabilir.

3. Ardından, özensiz sayacın bir sürümünü oluşturun. Bir kez daha, iplik sayısı ve harman eskisi değiştikçe performansını ölçün. Sayılar bölümde gördüklerinizle eşleşiyor mu?

Özensiz sayaç, paylaşılan sayaç değişkenine daha hızlı erişim sağlamak için rahat bir bellek sıralama modeli kullanan bir eşzamanlı sayaç türüdür. Bu, sayacın performansını artırabilir, ancak aynı zamanda yarış koşulları ve diğer senkronizasyon sorunları için potansiyel de sunar.

Özensiz bir sayaç oluşturmak için, paylaşılan sayaç değişkenini rahat bir bellek sıralama modeli kullanarak tanımlamamız gerekir. Bu, C++'daki std::atomik sınıf kullanılarak yapılabilir, bu da atomik işlemleri rahat bellek sıralaması ile sağlar. Daha sonra, sayacı birden fazla iş parçacığından artırmak için fetch_add() gibi atomik işlemleri kullanmamız gerekir.

Özensiz sayacın performansını ölçmek için, sayacı belirli sayıda artırmak için gereken geçen süreyi ölçmek üzere gettimeofday() gibi bir zamanlayıcı kullanmamız gerekir. Daha doğru bir ölçüm elde etmek için testi birden çok kez çalıştırmamız ve sonuçların ortalamasını almamız gerekir.

Özensiz sayacın performansı, iş parçacığı sayısı, rahat bellek modeline erişim eşiği ve kullanılan donanım ve işletim sistemi gibi bir dizi faktöre bağlı olacaktır. Sayılar, belirli uygulamaya ve kodun çalıştırıldığı sisteme bağlı olarak değişebileceğinden, bölümde görülenlerle eşleşmesi gerekmeyebilir.

4. Bağlantılı bir listenin, bölümde belirtildiği gibi, el ele kilitleme [MS04] kullanan bir sürümünü oluşturun. Nasıl çalıştığını anlamak için önce makaleyi okumalı ve sonra uygulamalısınız. Performansını ölçün. Teslim listesi, bölümde gösterildiği gibi standart bir listeden ne zaman daha iyi çalışır?

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

Uygulanacak olası veri yapılarından biri bir B-ağacıdır. B ağacı, sıralanmış verileri koruyan ve verimli ekleme, silme ve arama işlemlerine olanak tanıyan kendi kendini dengeleyen bir ağaç veri yapısıdır. Genellikle veri tabanlarında ve dosya sistemlerinde büyük miktarda veriyi depolamak ve almak için kullanılır.

Bir B-ağacını uygulamak için, öncelikle tipik olarak bir dizi anahtar ve bir dizi alt düğümden oluşacak bir B-ağacı düğümünün yapısını tanımlamamız gerekir. Daha sonra ekleme, silme ve arama gibi bir B ağacında gerçekleştirilebilecek çeşitli işlemleri uygulamamız gerekir.

Basit bir kilitleme stratejisiyle başlamak için, B ağacına erişimi korumak için tek bir muteks kullanabiliriz. Bu, aynı anda yalnızca bir iş parçacığının B ağacına erişebilmesini sağlayarak yarış koşullarını ve diğer eşitleme sorunlarını önler. Ancak bu, birden çok iş parçacığı aynı anda erişmeye çalıştığında B ağacının performansını da sınırlar.

B ağacının performansını tek bir kilitle ölçmek için, B ağacında belirli sayıda işlem gerçekleştirmek için gereken geçen süreyi ölçmek üzere gettimeofday() gibi bir zamanlayıcı kullanmamız gerekir. Daha doğru bir ölçüm elde etmek için testi birden çok kez çalıştırmamız ve sonuçların ortalamasını almamız gerekir.

Eşzamanlı iş parçacığı sayısı arttıkça, tek bir kilidi olan B ağacının performansı muhtemelen düşecektir, çünkü daha fazla iş parçacığı mutekse erişmek için mücadele edecek ve B ağacına erişmek için sıralarını beklemek zorunda kalacaktır. Bu, ek yükün artmasına ve performansın yavaşlamasına neden olur. B ağacının eşzamanlı bir ortamda performansını artırmak için, B ağacına daha fazla eşzamanlı erişim sağlamak üzere ince taneli kilitleme veya kilitsiz algoritmalar gibi

daha karmaşık bir kilitleme stratejisi kullanmamız gerekir.

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

Bir B ağacı için olası bir kilitleme stratejisi, B ağacındaki her düğümün kendi muteksi tarafından korunduğu ince taneli kilitleme kullanmaktır. Bu, birden fazla iş parçacığının B ağacının farklı düğümlerine aynı anda erişmesine izin verir, muteks için çekişme miktarını azaltır ve B ağacının performansını artırır.

Bir B-ağacı için ince taneli kilitleme uygulamak için, her B-ağacı düğümüne bir muteks üyesi eklememiz gerekir. Daha sonra, o düğüme erişmemiz gerektiğinde bir düğüm için muteks edinmemiz ve işimiz bittiğinde serbest bırakmamız gerekir. Bu, belirli bir düğüme aynı anda yalnızca bir iş parçacığının erişebilmesini sağlarken, diğer iş parçacıklarının aynı anda diğer düğümlere erişmesine izin verir.

B ağacının performansını ince taneli kilitleme ile ölçmek için, B ağacında belirli sayıda işlem gerçekleştirmek için gereken geçen süreyi ölçmek üzere gettimeofday() gibi bir zamanlayıcı kullanmamız gerekir. Daha doğru bir ölçüm elde etmek için testi birden çok kez çalıştırmamız ve sonuçların ortalamasını almamız gerekir.

B-ağacının ince taneli kilitleme ile performansı, B ağacına daha fazla eşzamanlı erişime izin vereceği ve muteks için çekişme miktarını azaltacağından, B ağacının tek bir kilitle olan performansından daha iyi olacaktır. Bununla birlikte, ince taneli kilitleme ile B ağacının performansı, iş parçacığı sayısı, kullanılan donanım ve işletim sistemi ve B ağacının özel uygulaması ve kilitleme stratejisi gibi bir dizi faktöre bağlı olacaktır.