29

Kilit Tabanlı Eşzamanlı Veri Yapıları

Kilitlerin ötesine geçmeden önce, bazı yaygın veri yapılarında kilitlerin nasıl kullanılacağını açıklayacağız. İş parçacıkları tarafından kullanılabilir hale getirmek için bir veri yapısına kilit eklemek yapıyı iş parçacığı (**thread)** güvenli hale getirir. Elbette, bu tür kilitlerin tam olarak nasıl eklendiği, veri yapısının hem doğruluğunu hem de performansını belirler. Ve böylece, bizim challengemiz:

VERI YAPILARINA NASIL KILIT EKLENIR

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 eşzamanlı(**concurrently**) olarak erişmesini sağlayacak şekilde kilitleri nasıl ekleriz?

Elbette, tüm veri yapılarını veya eşzamanlılık eklemek için tüm yöntemleri ele almakta zorlanacağız, çünkü bu yıllardır üzerinde çalışılan bir konudur ve bu konuda (kelimenin tam anlamıyla) binlerce araştırma makalesi yayınlanmıştır. Bu nedenle, gerekli düşünce türüne yeterli bir giriş sağlamayı ve sizi kendi başınıza daha fazla araştırma yapmanız için bazı iyi materyal kaynaklarına yönlendirmeyi umuyoruz. Moir ve Shavit'in araştırmasının harika bir bilgi kaynağı olduğunu 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 basit bir eşzamanlı olmayan sayaç tanımlıyoruz.

Basit Ama Ölçeklenebilir Değil (Simple But Not Scalable)

Gördüğünüz gibi, senkronize olmayan sayaç önemsiz bir veri yapısıdır ve uygulanması için çok az miktarda kod gerektirir. Şimdi bir sonraki zorluğumuz: bu kodu nasıl iş parçacığı(**thread**) güvenli hale getirebiliriz? Şekil 29.2 bunu nasıl yapacağımızı 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 }
```

Figure 29.1: **A Counter Without Locks**

Bu eşzamanlı sayaç basittir ve doğru şekilde çalışır. Aslında, en basit ve en temel eşzamanlı veri yapılarında yaygın olan bir tasarım modelini takip eder: veri yapısını manipüle eden bir rutin çağrıldığında edinilen ve çağrıdan dönüldüğünde serbest bırakılan tek bir kilit ekler. Bu şekilde, kilitlerin nesne yöntemlerini çağırırken ve bunlardan dönerken otomatik olarak edinildiği ve serbest bırakıldığı monitörlerle(**monitors**)[BH73] oluşturulmuş bir veri yapısına benzer.

Bu noktada, çalışan bir eşzamanlı veri yapısına sahip olursunuz. Karşılaşabileceğiniz sorun performanstır. Veri yapınız çok yavaşsa, tek bir kilit eklemekten daha fazlasını yapmanız gerekecektir; gerekirse bu tür optimizasyonlar bölümün geri kalanının konusudur. Veri yapısı çok yavaş değilse, işinizin bittiğini unutmayın! Basit bir şey işe yarayacaksa süslü bir şey yapmaya gerek yoktur.

Basit yaklaşımın performans maliyetlerini anlamak için, her iş parçacığının tek bir paylaşılan sayacı sabit sayıda güncellediği bir kıyaslama çalıştırıyoruz; daha sonra iş parçacığı sayısını değiştiriyoruz. Şekil 29.5, bir ila dört iş parçacığı etkinken geçen toplam süreyi göstermektedir; her iş parçacığı sayacı bir milyon kez güncellemektedir. Bu deney dört adet Intel 2.7 GHz i5 CPU'ya sahip bir iMac üzerinde gerçekleştirilmiştir; daha fazla CPU aktif olduğunda birim zamanda daha fazla toplam iş yapılacağını umuyoruz.

Şekildeki en üst satırdan ('Precise' etiketli), senkronize sayacın performansının zayıf bir şekilde ölçeklendiğini görebilirsiniz. Tek bir iş parçacığı bir milyon sayaç güncellemesini çok küçük bir sürede (yaklaşık 0,03 saniye) tamamlayabilirken, iki iş parçacığının her birinin sayacı

eş zamanlı olarak bir milyon kez güncellemesi büyük bir yavaşlamaya yol açıyor (5 saniyeden fazla sürüyor!). Daha fazla iş parçacığı ile durum daha da kötüleşir.

```
    int             value;
    pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value--;
    Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    int rc = c->value;
    Pthread_mutex_unlock(&c->lock);
    return rc;
}
```

İdeal olarak, iş parçacıklarının çoklu işlemcilerde tek bir iş parçacığında olduğu kadar hızlı bir şekilde tamamlandığını görmek istersiniz. Bu sonuca ulaşmaya mükemmel ölçekleme(**perfect scalling**) denir; daha fazla iş yapılsa da, iş parçacıkları eşit olarak yapılır ve bu nedenle görevi tamamlamak için geçen süre artmaz.

Ölçeklenebilir Sayım(Scalable Counting)

Şaşırtıcı bir şekilde, araştırmacılar yıllardır daha ölçeklenebilir sayıcıların nasıl oluşturulacağı üzerinde çalışmaktadır [MS04]. Daha da şaşırtıcı olan, işletim sistemi performans analizindeki son çalışmaların gösterdiği gibi ölçeklenebilir sayıcıların önemli olduğu gerçeğidir [B+10]; ölçeklenebilir sayıcı olmadan, Linux üzerinde çalışan bazı iş yükleri çok çekirdekli makinelerde ciddi ölçeklenebilirlik sorunlarından muzdariptir.

Bu sorunun üzerine gitmek i için birçok teknik geliştirilmiştir. Yaklaşık sayaç(**approximate counter**)olarak bilinen bir yaklaşımı açıklayacağız [C06].

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

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)

yerel sayaçlar ve bir küresel sayaç. Bu sayaçlara ek olarak, kilitler de vardır: her yerel sayaç1 için bir tane ve küresel sayaç için bir tane.

Yaklaşık saymanın temel fikri aşağıdaki gibidir. Bir iş parçacığı

Belirli bir çekirdek üzerinde çalışan iş parçacığı sayacı artırmak isterse, yerel sayacını artırır; bu yerel sayaca erişim, ilgili yerel kilit aracılığıyla senkronize edilir. Her CPU kendi yerel sayacına sahip olduğundan, CPU'lardaki iş parçacıkları yerel sayaçları çekişme olmadan güncelleyebilir ve böylece sayaç güncellemeleri ölçeklenebilir.

Bununla birlikte, global sayacı güncel tutmak için (bir iş parçacığının değerini okumak istemesi durumunda), yerel değerler periyodik olarak global kilidin alınması ve yerel sayacın değeri kadar artırılmasıyla global sayaca aktarılır; yerel sayaç daha sonra sıfırlanır.

Bu yerelden globale transferin ne sıklıkta gerçekleşeceği bir eşik değer S ile belirlenir. S ne kadar küçükse sayaç o kadar yukarıdaki ölçeklenemeyen sayaç gibi davranır; S ne kadar büyükse sayaç o kadar ölçeklenebilir, ancak global değer gerçek sayımdan o kadar uzak olabilir. Kesin bir değer elde etmek için tüm yerel kilitleri ve global kilidi (kilitlenmeyi önlemek için belirli bir sırayla) almak yeterlidir, ancak bu ölçeklenebilir değildir.

Bunu açıklığa kavuşturmak için bir örneğe bakalım (Şekil 29.3). Bu örnekte, S eşiği 5 olarak ayarlanmıştır ve dört CPU'nun her birinde yerel sayaçlarını güncelleyen iş parçacıkları vardır L1 ... L4. Global sayaç değeri

(G) de zaman aşağı doğru artarken izde gösterilir. Her zaman adımında, yerel bir sayaç artırılabilir; yerel değer S eşiğine ulaşırsa, yerel değer global sayaca aktarılır ve yerel sayaç sıfırlanır.

Şekil 29.5'teki alt satır ('Yaklaşık' etiketli, sayfa 6) 1024 S eşikli yaklaşık sayaçların performansını göstermektedir. Performans mükemmeldir; sayacı dört işlemci üzerinde dört milyon kez güncellemek için geçen süre, bir işlemci üzerinde bir milyon kez güncellemek için geçen süreden neredeyse hiç daha yüksek değildir.

```
    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!
}
```

Figure 29.4: **Approximate Counter Implementation**

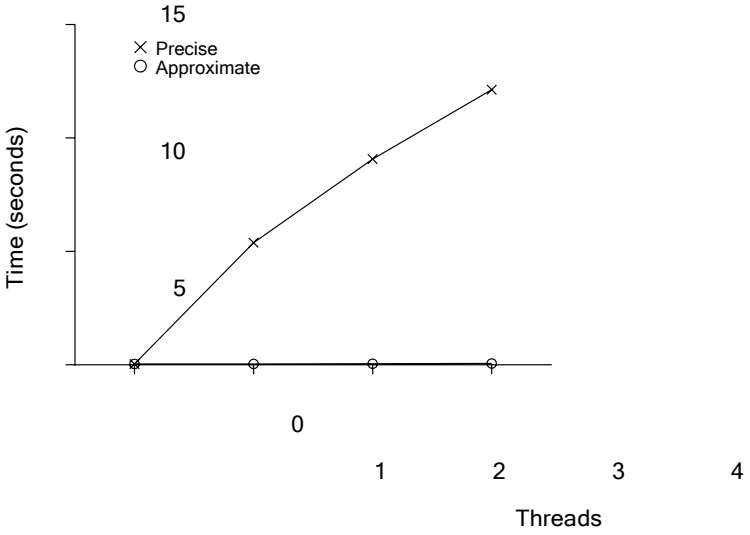

Figure 29.5: **Performance of Traditional vs. Approximate Counters**

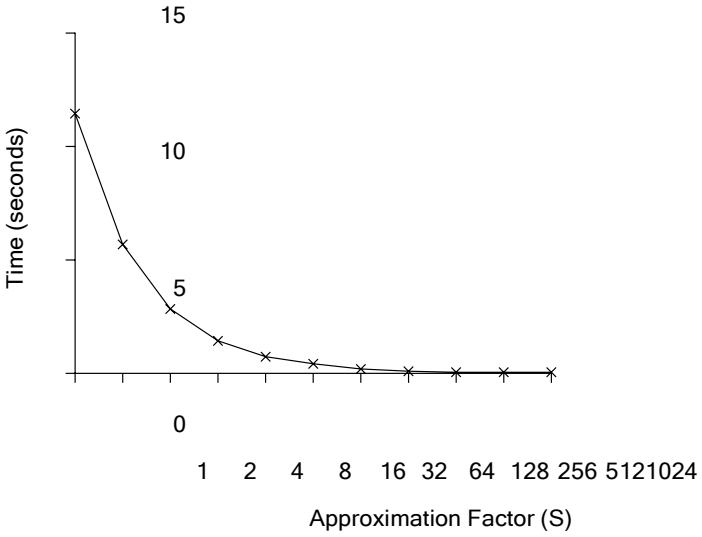

Figure 29.6: **Scaling Approximate Counters**

Şekil 29.6, her biri dört CPU üzerinde sayacı 1 milyon kez artıran dört iş parçacığı ile eşik değeri S'nin önemini göstermektedir. S düşükse, performans düşüktür (ancak global sayım her zaman oldukça doğrudur); S yüksekse, performans mükemmeldir, ancak global sayım gecikir (en fazla CPU sayısının S ile çarpımı kadar). Bu doğruluk/performans değiş tokuşu yaklaşık sayaçların mümkün kıldığı şeydir.

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

IPUCU: DAHA FAZLA EŞZAMANLILIK DAHA HIZLI OLMAK ZORUNDA DEĞILDIR

Tasarladığınız şema çok fazla ek yük getiriyorsa (örneğin, kilitleri bir kez yerine sık alıp bırakarak), daha eşzamanlı olması önemli olmayabilir. Basit şemalar, özellikle maliyetli rutinleri nadiren kullanıyorlarsa, iyi çalışma eğilimindedir. Daha fazla kilit ve karmaşıklık eklemek sizin çöküşünüz olabilir. Tüm bunlarla birlikte, gerçekten bilmenin bir yolu vardır: her iki alternatifi de (basit ama daha az eşzamanlı ve karmaşık ama daha fazla eşzamanlı) oluşturun ve nasıl yaptıklarını ölçün. Sonuçta, performans konusunda hile yapamazsınız; fikriniz ya daha hızlıdır ya da değildir.

29.2 **Eşzamanlı Bağlı Listeler(Concurrent Linked Lists)**

Şimdi daha karmaşık bir yapı olan bağlı listeyi inceleyeceğiz. Bir kez daha temel bir yaklaşımla başlayalım. Basitlik için, böyle bir listenin sahip olacağı bazı bariz rutinleri atlayacağız ve sadece concur- rent insert'e odaklanacağız; lookup, delete ve benzerleri hakkında düşünmeyi okuyucuya bırakacağız. Şekil 29.7 bu ilkel veri yapısı için kodu göstermektedir.

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

Bu tür istisnai kontrol akışının hataya oldukça yatkın olduğu gösterilmiştir; Linux çekirdek yamaları üzerinde yapılan yeni bir çalışma, hataların büyük bir kısmının (yaklaşık %40) bu tür nadiren kullanılan kod yollarında bulunduğunu ortaya koymuştur (aslında bu gözlem, bir Linux dosya sisteminden tüm bellek başarısız yollarını kaldırdığımız ve daha sağlam bir sistemle sonuçlanan kendi araştırmamızın bir kısmını tetiklemiştir [S+11]).

Bu nedenle, bir zorluk: ekleme ve arama rutinlerini eşzamanlı ekleme altında yeniden ana doğru olacak şekilde yeniden yazabilir miyiz, ancak hata yolunun aynı zamanda kilit açma çağrısını eklememizi gerektirdiği durumdan kaçınabilir miyiz?

Bu durumda cevap evettir. Özellikle, 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 işe yarar çünkü ekleme işleminin bir kısmının aslında kilitlenmesi gerekmez; malloc()'un kendisinin iş parçacığı güvenli olduğunu varsayarsak, her iş parçacığı yarış koşulları veya diğer eşzamanlılık hataları endişesi olmadan onu çağırabilir. Yalnızca paylaşılan liste güncellenirken bir kilit tutulması gerekir. Bu değişikliklerin ayrıntıları için Şekil 29.8'e bakın.

Arama rutinine gelince, ana arama döngüsünden tek bir dönüş yoluna atlamak basit bir kod dönüşümüdür. Bunu yapmak koddaki kilit alma/bırakma noktalarının sayısını yeniden azaltır

ve böylece koda yanlışlıkla hata ekleme (geri dönmeden önce kilidi açmayı unutmak gibi) olasılığını azaltır.

```
// 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;

void List_Init(list_t *L) {
    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**

```
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 }
```

Figure 29.8: **Concurrent Linked List: Rewritten**

Bağlı Listeleri Ölçekleme(Scalling Linked Lists)

Yine temel bir eşzamanlı bağlı listeye sahip olsak da, bir kez daha özellikle iyi ölçeklenmediği bir durumdayız. Araştırmacıların bir liste içinde daha fazla eşzamanlılık sağlamak için keşfettikleri bir teknik, elden ele kilitleme (diğer adıyla kilit bağlama) olarak adlandırılan bir şeydir [MS04]. Fikir oldukça basittir. Tüm liste için tek bir kilide sahip olmak yerine, listenin her düğümü için bir kilit eklersiniz. Listede gezinirken, kod önce bir sonraki düğümün kilidini alır ve ardından mevcut kilidi serbest bırakır. düğümünün kilidi (elden ele ismine ilham veren).

akışı değişikliklerine karşı dikkatli olmaktır. Birçok fonksiyon bir kilit edinerek, bir miktar bellek ayırarak ya da diğer benzer durumsal işlemleri yaparak başlayacağından, hatalar ortaya çıktığında, kodun geri dönmeden önce tüm durumu geri alması gerekir ki bu da hataya açıktır. Bu nedenle, en iyisi kodu bu modeli en aza indirecek şekilde yapılandırmaktır.

Kavramsal olarak, elden ele bağlı liste bir anlam ifade eder; liste işlemlerinde yüksek derecede eşzamanlılık sağlar. Ancak, pratikte böyle bir yapıyı basit tek kilit yaklaşımından daha hızlı hale getirmek zordur çünkü liste geçişinin her bir düğümü için kilit alma ve bırakma ek yükleri engelleyicidir. Çok büyük listelerde ve çok sayıda iş parçacığında bile, birden fazla devam eden geçişe izin vererek sağlanan eşzamanlılığın, tek bir kilidi kapmak, bir işlem gerçekleştirmek ve serbest bırakmaktan daha hızlı olması pek olası değildir. Belki de bir tür hy- brid (her düğümde yeni bir kilit aldığınız) araştırmaya değer olabilir.

29.3 **Eşzamanlı Kuyruklar(Concurrent Queues)**

Şimdiye kadar bildiğiniz gibi, eş zamanlı bir veri yapısı oluşturmak için her zaman standart bir yöntem vardır: büyük bir kilit eklemek. Bir kuyruk için, bunu 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 dikkatle incelerseniz, biri kuyruğun başı diğeri kuyruğu için olmak üzere iki kilit olduğunu fark edeceksiniz. Bu iki kilidin amacı, enqueue ve dequeue işlemlerinin eşzamanlılığını sağlamaktır. Genel durumda, enqueue rutini yalnızca kuyruk kilidine erişecek ve dequeue yalnızca baş kilidine erişecektir.

Michael ve Scott tarafından kullanılan bir hile, kuyruk başlatma koduna kukla bir düğüm eklemektir; bu kukla, baş ve kuyruk işlemlerinin ayrılmasını sağlar. Nasıl

çalıştığını derinlemesine anlamak için kodu inceleyin ya da daha iyisi yazın, çalıştırın ve ölçün.

Kuyruklar genellikle çok iş parçacıklı uygulamalarda kullanılır. Ancak, burada kullanılan kuyruk türü (sadece kilitlerle) genellikle bu tür programların ihtiyaçlarını tam olarak karşılamaz. Kuyruk boş ya da aşırı dolu olduğunda **TIP**: KİLİTLERE VE KONTROL AKIŞINA DİKKAT EDİN

Eşzamanlı kodda olduğu kadar başka yerlerde de faydalı olan genel bir tasarım ipucu, bir fonksiyonun yürütülmesini durduran fonksiyon dönüşlerine, çıkışlarına veya diğer benzer hata koşullarına yol açan kontrol

bir iş parçacığının beklemesini sağlayan daha tam gelişmiş sınırlı bir kuyruk, koşul değişkenleri ile ilgili bir sonraki bölümde yoğun çalışmamızın konusudur. Bunun için izleyin!

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;

typedef struct __queue_t {
    node_t              *head;
    node_t              *tail;
    pthread_mutex_t     head_lock, tail_lock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}

void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    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**

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;

    for (i = 0; i < BUCKETS; i++)
        List_Init(&H->lists[i]);
}

int Hash_Insert(hash_t *H, int key) {
    return List_Insert(&H->lists[key % BUCKETS], key);
}

int Hash_Lookup(hash_t *H, int key) {
    return List_Lookup(&H->lists[key % BUCKETS], key);
}
```

Figure 29.10: **A Concurrent Hash Table**

29.4 **Eşzamanlı Karma Tablo(Concurrent Hash Table)**

Tartışmamızı basit ve yaygın olarak uygulanabilir bir eşzamanlı veri yapısı olan hash tablosu ile sonlandırıyoruz. Yeniden boyutlandırılmayan basit bir hash tablosuna odaklanacağız; yeniden boyutlandırmayı ele almak için biraz daha fazla çalışma gerekiyor, bunu okuyucu için bir alıştırma olarak bırakıyoruz (üzgünüm!).

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

Şekil 29.11, eş zamanlı güncellemeler altında hash tablosunun performansını göstermektedir (dört CPU'lu aynı iMac üzerinde dört iş parçacığının her birinden 10.000 ila 50.000 eş zamanlı güncelleme). Ayrıca karşılaştırma amacıyla bağlantılı listenin (tek kilitli) performansı da gösterilmiştir. Grafikten de görebileceğiniz gibi, bu basit eşzamanlı hash tablosu muhteşem bir şekilde ölçekleniyor; buna karşın bağlantılı liste ölçeklenmiyor.

29.5 **Özet**

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

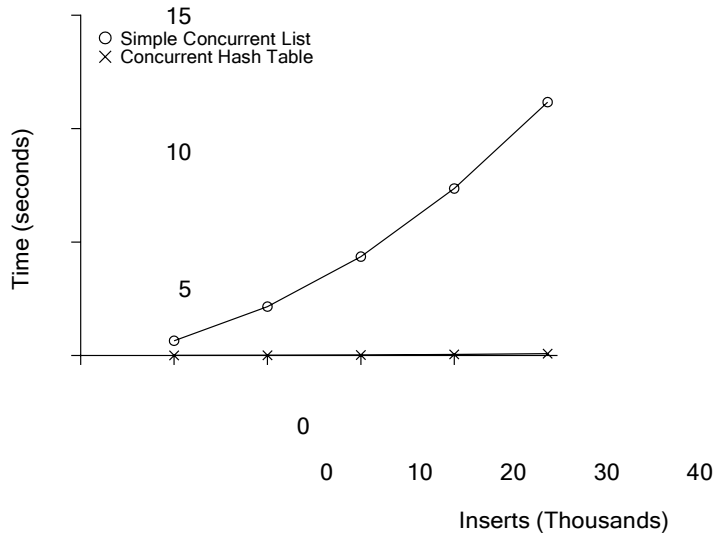

Figure 29.11: **Scaling Hash Tables**

değişiklikler; daha fazla eşzamanlılık sağlamanın performansı mutlaka artırmayacağı; performans sorunlarının yalnızca var olduklarında giderilmesi gerektiği. Bu son nokta, yani erken optimizasyondan(**premature optimization**) kaçınmak, performansa önem veren her geliştirici için çok önemlidir; uygulamanın genel performansını artırmayacaksa bir şeyi daha hızlı yapmanın hiçbir değeri yoktur.

Elbette, yüksek performanslı yapıların sadece yüzeyini çizdik. Daha fazla bilgi için Moir ve Shavit'in mükemmel araştırmasına ve diğer kaynaklara bağlantılara bakın [MS04]. Özellikle, diğer yapılarla (B-ağaçları gibi) ilgileniyor olabilirsiniz; bu bilgi için bir veritabanı sınıfı en iyi seçeneğinizdir. Ayrıca geleneksel kilitleri hiç kullanmayan teknikleri de merak ediyor olabilirsiniz; bu tür kilitlemesiz veri yapıları, yaygın eşzamanlılık hataları bölümünde tadına bakacağımız bir şeydir, ancak açıkçası bu konu, bu mütevazı kitapta mümkün olandan daha fazla çalışma gerektiren bütün bir bilgi alanıdır. Arzu ederseniz 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, muhtemelen doğru bir kilit oluştursunuz; daha sonra performans sorunlarından muzdarip olduğunu fark ederseniz, onu iyileştirebilir, böylece yalnızca gerekirse hızlı hale getirebilirsiniz. **Knuth**'un da belirttiği gibi, "Erken optimizasyon tüm kötülüklerin anasıdır."

Sun OS ve Linux da dahil olmak üzere birçok işletim sistemi çoklu işlemcilere ilk geçişte tek bir kilit kullanmıştır. Sonuncusunda bu kilidin bir adı bile vardı: büyük çekirdek kilidi(**big kernel lock**) (**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 darboğazı haline geldi. Böylece, nihayet bu sistemlere gelişmiş eşzamanlılık optimizasyonunu eklemenin zamanı gelmişti. Linux içinde, daha basit bir yaklaşım benimsendi: bir kilidi birçok kilitle değiştirmek. Sun'da ise daha radikal bir karar verildi: Solaris olarak bilinen ve ilk günden itibaren eşzamanlılığı daha temel olarak içeren yepyeni bir işletim sistemi inşa edildi. Bu büyüleyici sistemler hakkında daha fazla bilgi için Linux ve Solaris kernel kitaplarını okuyun [BC05, MM00].

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 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 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 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

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

Sorular

1. Bu bölümdeki ölçümleri yeniden yaparak başlayacağız. Programınızda zamanı ölçmek için gettimeofday() çağrısını kullanın. Bu zamanlayıcı ne kadar doğrudur? Ölçebildiği en küçük aralık nedir? Daha sonraki tüm sorularda ihtiyaç duyacağımız için bu zamanlayıcının işleyişi konusunda güven kazanın. Ayrıca x86'da rdtsc komutu aracılığıyla kullanılabilen döngü sayacı gibi diğer zamanlayıcılara da bakabilirsiniz.

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

3. Ardından, özensiz sayacın bir versiyonunu oluşturun. Bir kez daha, iş parçacığı sayısı değiştikçe performansını ve eşik değerini ölçün. Rakamlar bölümde gördüklerinizle uyuşuyor mu?

4. Bölümde atıfta bulunulduğu gibi elden ele kilitleme [MS04] kullanan bir bağlı liste versiyonu oluşturun. Nasıl çalıştığını anlamak için önce makaleyi okumalı ve sonra uygulamalısınız. Performansını ölçün. Bir elden ele liste, 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 kilit gibi basit bir kilitleme stratejisi ile başlayın. Eşzamanlı iş parçacığı sayısı arttıkça performansını ölçün.

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

1. **SORU**: İşlev '**gettimeofday()'**, geçerli zamanı saniye ve mikrosaniye cinsinden almak için kullanılabilen bir sistem çağrısıdır. Yüksek çözünürlüklü zamanlama bilgisi sağladığı için genellikle programlardaki zaman aralıklarını ölçmek için kullanılır. Doğruluğu '**gettimeofday()',** altta yatan işletim sistemine ve donanıma bağlıdır, ancak genel olarak oldukça doğrudur ve mikro saniyelik bir çözünürlüğe sahiptir.

Ölçülebilen en küçük aralık ‘**gettimeofday()’,** sistem saatinin çözünürlüğü ile belirlenir. Bu genellikle mikrosaniye mertebesindedir, ancak bazı sistemlerde nanosaniye kadar düşük olabilir.

komutuna alternatif olarak , CPU tarafından yürütülen döngü sayısını ölçmek için x86 tabanlı sistemlerde yönergeyi ‘**gettimeofday()**’de kullanabilirsiniz . ‘**rdtsc’** Bu, çok küçük aralıkları yüksek hassasiyetle ölçmek için kullanılabilir, ancak farklı mimarilerde taşınabilir değildir ve CPU frekans ölçeklendirmesi ve sıra dışı yürütme gibi çeşitli faktörlerden etkilenebilir.

‘gettimeofday()’ Genel olarak , çoğu amaç için gibi sistem tarafından sağlanan zamanlayıcı işlevlerini kullanmak en iyisidir . **‘rdtsc’** Ancak, çok yüksek hassasiyette zamanlama gerekiyorsa, talimat gibi özel teknikler kullanmanız gerekebilir .

2. **SORU:** Basit bir eşzamanlı sayaç oluşturmak için std::atomicsınıfı C++'da kullanabilirsiniz. Bu sınıf, açık senkronizasyona ihtiyaç duymadan eşzamanlı programlarda kullanılabilen, bir sayacı artırma gibi paylaşılan bir değişken üzerinde atomik işlemler sağlar. Aşağıdakileri kullanan bir eşzamanlı sayaç örneği std::atomic:

```cpp
#include <atomic>
#include <thread>

std::atomic<int> counter(0);

void increment_counter()
{
    for (int i = 0; i < 100000; ++i)
    {
        ++counter;
    }
}

int main()
{
    // Create 10 threads that will each increment the counter
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i)
    {
        threads.push_back(std::thread(increment_counter));
    }

    // Wait for all threads to finish
    for (auto& thread : threads)
    {
        thread.join();
    }

    // Print the final value of the counter
    std::cout << "Counter value: " << counter << std::endl;

    return 0;
}
```

Artan iş parçacığı sayıları ile sayacı artırmanın ne kadar sürdüğünü gettimeofday()ölçmek için, geçen süreyi ölçme işlevini kullanabilirsiniz. İşte bunun nasıl yapılacağına dair bir örnek:

```cpp
#include <sys/time.h>
#include <thread>

std::atomic<int> counter(0);

void increment_counter()
{
    for (int i = 0; i < 100000; ++i)
    {
        ++counter;
    }
}

int main()
{
    // Measure elapsed time
    struct timeval start, end;
    gettimeofday(&start, nullptr);

    // Create 10 threads that will each increment the counter
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i)
    {
        threads.push_back(std::thread(increment_counter));
    }

    // Wait for all threads to finish
    for (auto& thread : threads)
    {
        thread.join();
    }

    // Measure elapsed time
    gettimeofday(&end, nullptr);

    // Print elapsed time in seconds
    double elapsed_time = (end.tv_sec - start.tv_sec) + (end.tv_usec -
start.tv_usec) / 1000000.0;
    std::cout << "Elapsed time: " << elapsed_time << " seconds" << std::endl;

    return 0;
}
```

Sistemde bulunan CPU sayısı, aynı anda kaç iş parçacığının çalışabileceğini belirlediği için ölçümleri etkileyebilir. İş parçacığı sayısı kullanılabilir CPU sayısını aşarsa, iş parçacıkları aynı CPU'larda çalışacak şekilde programlanacak ve bu da çekişmeye ve düşük performansa neden olabilir. ‘**std::thread::hardware_concurrency()’** Sistemdeki kullanılabilir CPU sayısını belirlemek için işlevi kullanabilirsiniz .

Genel olarak, eşzamanlı programları çalıştırırken kullanılabilir CPU sayısına eşit sabit sayıda iş parçacığı kullanmak en iyisidir. Bu, her iş parçacığının üzerinde çalışacak kendi CPU'suna sahip olmasını sağlar, bu da performansı artırabilir ve çekişmeyi azaltabilir

3. SORU:Özensiz bir sayaç oluşturmak için '**std::mutex**', paylaşılan sayaç değişkenini korumak için a kullanabilirsiniz. Bu, birden çok iş parçacığının aynı anda sayacı artırmasına izin verir, ancak aynı anda yalnızca bir iş parçacığı sayaca erişebilir. Aşağıda, aşağıdakileri kullanan özensiz bir sayaç örneği verilmiştir **'std::mutex'**:

```cpp
#include <mutex>
#include <thread>

std::mutex counter_mutex;
int counter = 0;

void increment_counter()
{
    for (int i = 0; i < 100000; ++i)
    {
        std::lock_guard<std::mutex> lock(counter_mutex);
        ++counter;
    }
}

int main()
{
    // Create 10 threads that will each increment the counter
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i)
    {
        threads.push_back(std::thread(increment_counter));
    }

    // Wait for all threads to finish
    for (auto& thread : threads)
    {
        thread.join();
    }

    // Print the final value of the counter
    std::cout << "Counter value: " << counter << std::endl;

    return 0;
```

Özensiz sayacın performansını ölçmek için '**gettimeofday()',** önceki yanıtta açıklanan işlevi kullanabilirsiniz. Ardından, sayaç performansını nasıl etkilediklerini görmek için iş parçacığı

sayısını ve eşiği (yani, muteks serbest bırakılmadan önce sayacın artırılması gereken sayı) değiştirebilirsiniz.

'std::mutex' Özensiz sayacın performansının, kullanılabilir CPU'ların sayısı, iş parçacıklarının iş yükü ve sınıfın özel uygulaması gibi çeşitli faktörlere bağlı olacağına dikkat etmek önemlidir. Bu nedenle, ölçümlerinizde gördüğünüz rakamlar, bölümde gösterilen rakamlarla tam olarak eşleşmeyebilir. Ayrıca, özensiz sayacın genellikle atomik sayaçtan daha az verimli olduğunu belirtmek gerekir, çünkü her bir artış işlemi için muteksi edinme ve serbest bırakma ek yüküne neden olur.

4. **SORU**: Hand-over-hand kilitleme, birden çok iş parçacığının aynı anda listenin farklı bölümlerine erişmesine izin veren eşzamanlı bağlantılı listeleri uygulamaya yönelik bir tekniktir. Bu, iş parçacıkları arasındaki çekişme miktarını azalttığından, tek bir genel kilit kullanan standart bir listeye kıyasla bağlantılı listenin performansını artırabilir.

Elden teslim kilitli bir bağlantılı liste uygulamak için, düğüm yapısını her düğüm için bir muteks içerecek şekilde değiştirmeniz gerekir. Bu muteks, düğümün verilerini ve bağlantılarını korumak için kullanılacak ve düğüme her eriştiğinde bir iş parçacığı tarafından alınacaktır. Aşağıda, C++'da elden ele kilitli bağlantılı bir liste örneği verilmiştir:

```
#include <mutex>

struct Node
{
    int data;
    std::mutex node_mutex;
    Node* next;
};

class LinkedList
{
public:
    LinkedList()
    {
        head = new Node;
        head->next = nullptr;
    }

    void insert(int data)
    {
        Node* new_node = new Node;
        new_node->data = data;
        new_node->next = nullptr;

        // Lock the head node
        std::lock_guard<std::mutex> head_lock(head->node_mutex);

        // Find the last node in the list
        Node* current_node = head;
        while (current_node->next != nullptr)
        {
            current_node = current_node->next;

            // Lock the next node before moving to it
            std::lock_guard<std::mutex> next_lock(current_node->next-
>node_mutex);
        }

        // Insert the new node at the end of the list
        current_node->next = new_node;
    }

private:
    Node* head;
```

Elden ele kilitli bağlantılı listenin performansını ölçmek için, **'gettimeofday()'** işlevi önceki yanıtlarda açıklanan şekilde kullanabilirsiniz. Ardından, listenin performansını nasıl etkilediklerini görmek için iş parçacığı sayısını ve listenin iş yükünü (yani ekleme sayısını) değiştirebilirsiniz.

Genel olarak, elden ele kilitli bir bağlantılı liste, başlıklar arasında yüksek derecede çekişme olduğunda standart bir bağlantılı listeden daha iyi performans gösterir. Bunun nedeni, elden teslim tekniğinin birden çok iş parçacığının listenin farklı bölümlerine aynı anda erişmesine izin vererek kilitlerin alınmasını beklerken harcanan süreyi azaltmasıdır. Ancak, elden teslim tekniğinin, birden fazla kilit alma ve serbest bırakma ihtiyacı nedeniyle ek yük getirdiğini belirtmekte fayda var, bu nedenle belirli iş yüküne ve donanıma bağlı olarak her zaman en iyi seçim olmayabilir.

5. SORU:Favori bir veri yapısı olarak bir B-ağacı uygulayacağım. Bir B-ağacı, verimli eklemelere, silmelere ve aramalara izin veren, kendi kendini dengeleyen bir ağaç veri yapısıdır. Genellikle veritabanlarında ve dosya sistemlerinde büyük miktarda veriyi indekslemek ve depolamak için kullanılır.

Bir B ağacını uygulamak için, düğüm yapısını ve B ağacı sınıfını tanımlamanız gerekir. Düğüm yapısı, bir dizi anahtar, bir dizi değer ve bir dizi alt işaretçi içermelidir. B-ağacı sınıfı, ağaçta anahtar ekleme, silme ve arama işlevleri içermelidir. İşte C++'da bir B-ağacı örneği:

```cpp
#include <algorithm>
#include <cstdint>
#include <iostream>

const int MAX_KEYS = 3;

struct Node
{
    int keys[MAX_KEYS + 1];
    int values[MAX_KEYS + 1];
    Node* children[MAX_KEYS + 2];
    int num_keys;

    Node()
    {
        num_keys = 0;
        std::fill(keys, keys + MAX_KEYS + 1, 0);
        std::fill(values, values + MAX_KEYS + 1, 0);
        std::fill(children, children + MAX_KEYS + 2, nullptr);
    }
};

class BTree
{
public:
    BTree()
    {
        root = new Node;
    }

    void insert(int key, int value)
    {
        Node* r = root;
        if (r->num_keys == MAX_KEYS)
        {
```

```
        {
            Node* s = new Node;
            root = s;
            s->children[0] = r;
            split_child(s, 0);
            insert_nonfull(s, key, value);
        }
        else
        {
            insert_nonfull(r, key, value);
        }
    }

    void remove(int key)
    {
        // TODO: implement remove function
    }

    int search(int key)
    {
        Node* r = root;
        while (r != nullptr)
        {
            int i = 0;
            while (i < r->num_keys && key > r->keys[i])
            {
                ++i;
            }
            if (i < r->num_keys && key == r->keys[i])
            {
                return r->values[i];
            }
            r = r->children[i];
        }
        return -1;
    }
```

```
            int i = 0;
            while (i < r->num_keys && key > r->keys[i])
            {
                ++i;
            }
            if (i < r->num_keys && key == r->keys[i])
            {
                return r->values[i];
            }
            r = r->children[i];
        }
        return -1;
    }

private:
    Node* root;

    void split_child(Node* parent, int index)
    {
        Node* child = parent->children[index];
        Node* sibling = new Node;

        sibling->num_keys = MAX_KEYS / 2;
        for (int j = 0; j < MAX_KEYS / 2; ++j)
        {
            sibling->keys[j] = child->keys[j + MAX_KEYS / 2 + 1];
            sibling->values[j] = child->values[j + MAX_KEYS / 2 + 1];
            sibling->children[j] = child->children[j + MAX_KEYS / 2 + 1];
        }
        sibling->children[MAX
```