32

Yaygın Eş Zamanlılık Sorunları

Araştırmacılar, uzun yıllar boyunca eşzamanlı böcekleri araştırmak için çok fazla zaman ve çaba harcadılar. İlk çalışmaların çoğu, önceki bölümlerde değindiğimiz ancak şimdi derinlemesine inceleyeceğimiz bir konu olan **(deadlock) kilitlenme** üzerine odaklandı [C+71]. Daha yeni çalışmalar, diğer ortak eşzamanlılık hatalarını (yani, kilitlenme olmayan hataları) incelemeye odaklanmaktadır. Bu bölümde, hangi sorunlara dikkat edilmesi gerektiğini daha iyi anlamak için gerçek kod tabanlarında bulunan bazı örnek eşzamanlılık sorunlarına kısaca göz atacağız. Bu bölümdeki ana meselemiz:

CRUX: ORTAK EŞ ZAMANLILIK HATALARINI NASIL ELE ALINIR
Eşzamanlılık hataları, çeşitli ortak modellerde ortaya çıkma eğilimindedir. Hangilerine dikkat edileceğini bilmek, daha sağlam, doğru eşzamanlı kod yazmanın ilk adımıdır.

32.1 Ne Tür Hatalar Var?

İlk ve en bariz soru şudur: Karmaşık, eşzamanlı programlarda ne tür eşzamanlılık hataları ortaya çıkıyor? Bu soruyu genel olarak cevaplamak zordur, ancak neyse ki başkaları bu işi bizim için yaptı. Spesifik olarak, Lu ve diğerleri tarafından yapılan bir araştırmaya güveniyoruz. [L+08], pratikte ne tür hataların ortaya çıktığını anlamak için bir dizi popüler eşzamanlı uygulamayı ayrıntılı olarak analiz eder.

Çalışma, dört ana ve önemli açık kaynaklı uygulamaya odaklanmaktadır: MySQL (popüler bir veri tabanı yönetim sistemi), Apache (iyi bilinen bir web sunucusu), Mozilla (ünlü web tarayıcısı) ve OpenOffice (MS Office paketinin ücretsiz bir sürümü, bazı insanların gerçekte kullandığı). Çalışmada yazarlar, bu kod tabanlarının her birinde bulunan ve düzeltilen eşzamanlılık hatalarını inceleyerek, geliştiricilerin çalışmalarını nicel bir hata analizine dönüştürüyor; bu sonuçları anlamak, olgun kod tabanlarında gerçekte ne tür sorunların ortaya çıktığını anlamanıza yardımcı olabilir.

Şekil 32.1, Lu ve meslektaşlarının incelediği hataların bir özetini gösterir. Şekilden, çoğu kilitlenme olmayan toplam 105 hata olduğunu görebilirsiniz

Uygulama	Ne işe yarar	Kilitlenme Olmayan	Kilitlenme
MySQL	Veritabanı Sunucu	14	9
Apache	Web Sunucu	13	4
Mozilla	Web Tarayıcı	41	16
OpenOffice	Office Paketi	6	2
Toplam		74	31

Figür 32.2: **Modern Uygulamalardaki Hataları**

(74); kalan 31 tanesi kilitlenme hatasıydı. Ayrıca, her uygulamadan incelenen hataların sayısını görebilirsiniz; OpenOffice toplam yalnızca 8 eşzamanlılık hatasına sahipken, Mozilla'da yaklaşık 60 hata vardı.

Şimdi bu farklı hata sınıflarına (kilitlenme olmayan, kilitlenme) biraz daha derinden giriyoruz. Kilitlenme olmayan hataların birinci sınıfı için, tartışmamızı yönlendirmek için çalışmadan örnekler kullanıyoruz. İkinci sınıf kilitlenme hataları için, kilitlenmeyi önlemek, kaçınmak veya kilitlenmeyi ele almak için yapılan uzun çalışmaları tartışıyoruz.

32.2 Kilitlenme Olmayan Hatalar

Lu'nun araştırmasına göre, kilitlenmeyen hatalar eşzamanlılık hatalarının çoğunu oluşturuyor. Ancak bunlar ne tür hatalardır? Nasıl ortaya çıkıyorlar? Onları nasıl düzeltebiliriz? Şimdi Lu ve diğerleri tarafından bulunan kilitlenme olmayan iki ana hata türünü tartışacağız: **atomicity violation (atomiklik ihlali)** hataları ve **order violation (düzen ihlali)** hataları.

Atomiklik İhlal Hatalar

Karşılaşılan ilk sorun türü, atomiklik ihlali olarak adlandırılır. İşte MySQL'de bulunan basit bir örnek. Açıklamayı okumadan önce hatanın ne olduğunu bulmaya çalışın. Yap!

```
1 Thread 1::
2 if (thd->proc_info) {
3   fputs(thd->proc_info, ...);
4 }
5
6 Thread 2::
7 thd->proc_info = NULL;
```

Figür 32.2: **Atomiklik İhlal (`atomiklik.c`)**

Örnekte, iki farklı iş parçacığı `thd` yapısındaki `proc info` alanına erişir. İlk iş parçacığı, değerin NULL olup olmadığını kontrol eder ve ardından değerini yazdırır; ikinci iş parçacığı onu NULL olarak ayarlar. Açıktır ki, ilk iş parçacığı kontrolü gerçekleştirir ancak ardından `fputs` çağrısından önce kesilirse, ikinci iş parçacığı arada çalışabilir ve bu nedenle işaretçiyi NULL' a ayarlayabilir; ilk iş parçacığı devam ettiğinde, bir NULL işaretçisi `fputs` tarafından başvurulmayacağından çökecektir.

Lu ve diğerlerine göre atomiklik ihlalinin daha resmi tanımı şudur: "Birden fazla bellek erişimi arasında istenen seri hale getirilebilirlik ihlal edilmiştir (yani, bir kod bölgesinin atomik olması amaçlanır, ancak atomiklik yürütme sırasında uygulanmaz). Yukarıdaki örneğimizde, kod, `proc info`'nun NULL olmayan olup olmadığının kontrolü ve `fputs()` çağrısında `proc info`'nun kullanımı hakkında bir *atomiklik varsayımına* (Lu'nun sözleriyle) sahiptir; varsayım yanlış olduğunda, kod istenildiği gibi çalışmayacaktır.

Bu tür bir sorun için bir çözüm bulmak genellikle (ancak her zaman değil) basittir. Yukarıdaki kodu nasıl düzelteceğinizi düşünebilir misiniz?

Bu çözümde (Şekil 32.3), paylaşılan değişken referanslarının çevresine basitçe kilitler ekleyerek, her iki iş parçacığının `proc_info` alanına erişmesi durumunda bir kilidin (`proc_info` kilidi) tutulmasını sağlıyoruz. Tabii ki, yapıya erişen diğer herhangi bir kodun da bunu yapmadan önce bu kilidi alması gerekir.

```
1  pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Thread 1::
4  pthread_mutex_lock(&proc_info_lock);
5  if (thd->proc_info) {
6    fputs(thd->proc_info, ...);
7  }
8  pthread_mutex_unlock(&proc_info_lock);
9
10 Thread 2::
11 pthread_mutex_lock(&proc_info_lock);
12 thd->proc_info = NULL;
13 pthread_mutex_unlock(&proc_info_lock);
```

Figür 32.3: **Atomiklik İhlali Düzeltildi (atomicity fixed.c)**

Sipariş İhlali Hataları

Lu ve diğerleri tarafından bulunan başka bir yaygın kilitlenme dışı hata türü. **order violation (sipariş ihlali)** olarak bilinir. İşte başka bir basit örnek; Bir kez daha, aşağıdaki kodun neden bir hata içerdiğini anlayabilecek misiniz bir bakın.

```
1  Thread 1::
2  void init() {
3     mThread = PR_CreateThread(mMain, ...);
4  }
5
6  Thread 2::
7  void mMain(...) {
8      mState = mThread->State;
9  }
```

Figür 32.4: **Sıralama Hatası (ordering.c)**

Muhtemelen anladığınız gibi, Thread 2'deki kod, `mThread` değişkeninin zaten başlatıldığını (ve NULL olmadığını) varsayıyor gibi görünüyor;

```
1  pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
3  int mtInit             = 0;
4
5  Thread 1::
6  void init() {
7      ...
8      mThread = PR_CreateThread(mMain, ...);
9
10     // signal that the thread has been created...
11     pthread_mutex_lock(&mtLock);
12     mtInit = 1;
13     pthread_cond_signal(&mtCond);
14     pthread_mutex_unlock(&mtLock);
15     ...
16 }
17
18 Thread 2::
19 void mMain(...) {
20      ...
21      // wait for the thread to be initialized...
22      pthread_mutex_lock(&mtLock);
23      while (mtInit == 0)
24          pthread_cond_wait(&mtCond, &mtLock);
25      pthread_mutex_unlock(&mtLock);
26
27      mState = mThread->State;
28      ...
29 }
```

Figür 32.5: **Sıralama İhlalini Düzeltme (orderingfixed.c)**

ancak, Thread 2 oluşturulduktan hemen sonra çalışırsa, `mThread` 'in değeri, Thread 2'deki `mMain()` içinden erişildiğinde ayarlanmayacaktır ve büyük olasılıkla bir NULL işaretçi başvurusuyla çökecektir. `mThread` değerinin başlangıçta NULL olduğunu varsaydığımızı unutmayın; değilse, İş Parçacığı 2'deki başvuru yoluyla rastgele bellek konumlarına erişildiğinden daha garip şeyler olabilir.

Bir emir ihlalinin daha resmi tanımı şu şekildedir: "İki (grup) hafıza erişimi arasındaki istenen sıra tersine çevrilir (yani, A her zaman B'den önce yürütülmelidir, ancak emir yürütme sırasında uygulanmaz)" [L+ 08].

Bu tür bir hatanın düzeltilmesi genellikle sıralamayı zorunlu kılmak içindir. Daha önce tartışıldığı gibi, **condition variables** (**koşul değişkenlerini)** kullanmak, bu tarz senkronizasyonu modern kod tabanlarına eklemenin kolay ve sağlam bir yoludur. Yukarıdaki örnekte, kodu Şekil 32.5'te görüldüğü gibi yeniden yazabiliriz.

Bu düzeltilmiş kod dizisinde, bir durum değişkeni (`mtCond`) ve karşılık gelen kilit (`mtLock`) ile bir durum değişkeni (`mtInit`) ekledik. Başlatma

kodu çalıştığında, `mtInit`'in durumunu 1 olarak ayarlar ve öyle yaptığını bildirir. İş Parçacığı 2 bu noktadan önce çalışmışsa, bu sinyali ve karşılık gelen durum değişikliğini bekliyor olacaktır; daha sonra çalışırsa, durumu kontrol edecek ve başlatmanın zaten gerçekleştiğini görecek (yani, `mtInit` 1'e ayarlı) ve böylece uygun şekilde devam edecek. `mThread` 'i durum değişkeninin kendisi olarak kullanabileceğimize dikkat edin, ancak bunu burada basitlik adına yapmayın. Konuları iş parçacığı arasında sıralarken koşul değişkenleri (veya semaforlar) imdada yetişebilir.

Kilitlenme Olmayan Hatalar: Özet

Lu ve diğerleri tarafından incelenen kilitlenme olmayan hataların büyük bir bölümü (%97) atomiklik veya düzen ihlalleridir. Bu nedenle, bu tür hata kalıpları hakkında dikkatli bir şekilde düşünerek, programcılar muhtemelen onlardan kaçınmak için daha iyi bir iş çıkarabilirler. Ayrıca, daha otomatik kod denetleme araçları geliştikçe, dağıtımda bulunan kilitlenme olmayan hataların çok büyük bir kısmını oluşturdukları için muhtemelen bu iki tür hataya odaklanmalıdırlar.

Ne yazık ki, tüm hatalar yukarıda incelediğimiz örnekler kadar kolay düzeltilemez. Bazıları, programın ne yaptığına dair daha derin bir anlayış veya düzeltmek için daha büyük miktarda kod veya veri yapısının yeniden düzenlenmesini gerektirir. Daha fazla ayrıntı için Lu ve diğerlerinin mükemmel (ve okunabilir) makalesini okuyun.

32.3 Kilitlenme Hataları

Yukarıda belirtilen eşzamanlılık hatalarının ötesinde, karmaşık kilitleme protokollerine sahip birçok eşzamanlı sistemde ortaya çıkan klasik bir sorun **deadlock (kilitlenme)** olarak bilinir. Kilitlenme, örneğin, bir iş parçacığı (diyelim ki İş Parçacığı1) bir kilidi tutarken (`L1`) ve diğerini beklerken (`L2`) meydana gelir; ne yazık ki, `L2` kilidini tutan iş parçacığı (Thread 2) `L1`'in serbest bırakılmasını bekliyor. İşte böyle bir potansiyel kilitlenmeyi gösteren bir kod parçacığı:

Atomiklik-İhlal Hataları

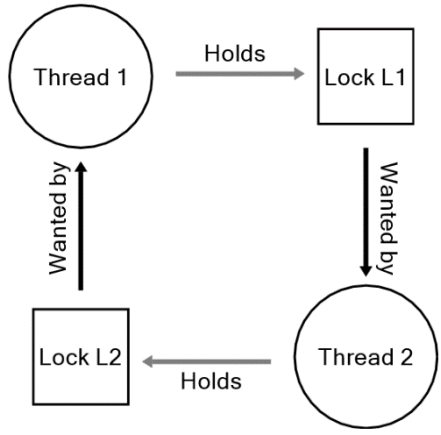
Karşılaşılan ilk sorun türü, **atomiklik ihlali** olarak adlandırılır. İşte MySQL'de bulunan basit bir örnek. Açıklamayı okumadan önce hatanın ne olduğunu bulmaya çalışın. Yap!

```
Thread 1:                       Thread 2:
pthread_mutex_lock(L1);         pthread_mutex_lock(L2);
pthread_mutex_lock(L2);         pthread_mutex_lock(L1);
```

Figür 32.6: **Atomiklik İhlali (atomicity.c)**

Bu kod çalışırsa kilitlenmenin mutlaka gerçekleşmediğini unutmayın; bunun yerine, örneğin İş Parçacığı 1, `L1` kilidini alırsa ve ardından İş Parçacığı 2'ye bir bağlam geçişi gerçekleşirse oluşabilir. Bu noktada, İş Parçacığı 2, `L2`'yi alır ve `L1`'i almaya çalışır. Böylece, her iş parçacığı

diğerini beklediği ve hiçbiri çalışamadığı için bir kilitlenme yaşarız. Grafik bir tasvir için bkz. Şekil 32.7; grafikte bir döngünün varlığı kilitlenmenin göstergesidir.
Şekil sorunu açıklığa kavuşturmalıdır. Kilitlenmeyi bir şekilde ele almak için programcılar nasıl kod yazmalıdır?



Figür 32.7: **Kilitlenme Bağımlılık Grafiği**

CRUX: KİTLENMEYLE NASIL BAŞA ÇIKILIR

Kilitlenmeyi önlemek, kaçınmak veya en azından tespit edip bu kilitlenmeden kurtulmak için nasıl sistemler kurmalıyız? Bu, günümüz sistemlerinde gerçek bir sorun mu?

Kilitlenmeler Neden Oluşur?

Düşünüyor olabileceğiniz gibi, yukarıdaki gibi basit kilitlenmeler kolayca önlenebilir görünüyor. Örneğin, İş Parçacığı 1 ve 2'nin her ikisi de kilitleri aynı sırayla kaptığından emin olsaydı, kilitlenme asla ortaya çıkmazdı. Öyleyse kilitlenmeler neden oluyor?

Bunun bir nedeni, büyük kod tabanlarında bileşenler arasında karmaşık bağımlılıkların ortaya çıkmasıdır. Örneğin işletim sistemini ele alalım. Sanal bellek sisteminin, diskten bir bloğa sayfa eklemek için dosya sistemine erişmesi gerekebilir; dosya sistemi daha sonra bloğu okumak ve böylece sanal bellek sistemiyle iletişim kurmak için bir bellek sayfası gerektirebilir. Bu nedenle, büyük sistemlerde kilitleme stratejilerinin tasarımı, kodda doğal olarak meydana gelebilecek döngüsel bağımlılıklar durumunda kilitlenmeyi önlemek için dikkatli bir şekilde yapılmalıdır.

Diğer bir neden, **encapsulation (kapsüllemenin)** doğasından kaynaklanmaktadır. Yazılım geliştiriciler olarak, uygulamaların ayrıntılarını gizlememiz ve böylece yazılımın modüler bir şekilde oluşturulmasını

kolaylaştırmamız öğretildi. Ne yazık ki, bu tür bir modülerlik, kilitleme ile pek örtüşmez. Jula ve diğerleri gibi. [J+08]'e dikkat edin, görünüşte zararsız görünen bazı arayüzler sizi neredeyse çıkmaza davet ediyor. Örneğin, Java Vector sınıfını ve `AddAll ()` yöntemini ele alalım. Bu rutin aşağıdaki gibi çağrılacaktır:

```
Vector v1, v2;
v1.AddAll(v2);
```

Dahili olarak, yöntemin çok iş parçacıklı güvenli olması gerektiğinden, hem (v1)'e eklenen vektör hem de (v2) parametresi için kilitlerin alınması gerekir. Rutin, v2'nin içeriğini v1'e eklemek için söz konusu kilitleri keyfi bir sırayla (v1 sonra v2 deyin) alır. Başka bir iş parçacığı `v2.AddAll (v1)` öğesini hemen hemen aynı anda çağırırsa, tümünü çağıran uygulamadan oldukça gizli bir şekilde kilitlenme potansiyeline sahibiz.

Kilitlenme Koşulları

Kilitlenmenin gerçekleşmesi için dört koşulun sağlanması gerekir [C+71]:

- **Mutual exclusion (Karşılıklı dışlama):** Konular, ihtiyaç duydukları kaynakların özel kontrolünü talep eder (örneğin, bir iş parçacığı bir kilidi kapar).
- **Hold-and-wai (Tut ve bekle):** İş parçacıkları, ek kaynakları (örneğin, elde etmek istedikleri kilitler) beklerken kendilerine tahsis edilen kaynakları (örneğin, halihazırda edinmiş oldukları kilitler) tutar.
- **No preemption (Önleme yok):** Kaynaklar (ör. kilitler) onları tutan iş parçacığından zorla kaldırılamaz.
- **Circular wait (Dairesel bekleme) :** Her iş parçacığının zincirdeki bir sonraki iş parçacığı tarafından talep edilen bir veya daha fazla kaynağı (örneğin kilitler) tuttuğu dairesel bir iş parçacığı zinciri vardır.

Bu dört koşuldan herhangi biri karşılanmazsa kilitlenme oluşamaz. Bu nedenle, önce kilitlenmeyi önleme tekniklerini araştırıyoruz; bu stratejilerin her biri, yukarıdaki koşullardan birinin ortaya çıkmasını engellemeye çalışır ve bu nedenle, kilitlenme sorununu ele almak için bir yaklaşımdır.

Önleme

Dairesel Bekleme

Muhtemelen en pratik önleme tekniği (ve kesinlikle sıklıkla kullanılan bir teknik), kilitleme kodunuzu asla döngüsel bir beklemeye neden olmayacak şekilde yazmaktır. Bunu yapmanın en basit yolu, kilit edinimi konusunda tam bir sıralama sağlamaktır. Örneğin, sistemde yalnızca iki kilit varsa (`L1` ve `L2`), her zaman `L1`'i `L2`'den önce alarak kilitlenmeyi önleyebilirsiniz. Bu tür katı sıralama, hiçbir döngüsel beklemenin ortaya çıkmamasını sağlar; dolayısıyla kilitlenme yok.

Tabii ki, daha karmaşık sistemlerde, ikiden fazla kilit bulunacaktır ve bu nedenle toplam kilit sıralamasını başarmak zor olabilir (ve belki de zaten gereksizdir). Bu nedenle, **partial ordering (kısmi bir sıralama)**, kilitlenmeyi önlemek için kilit edinimini yapılandırmak için yararlı bir yol olabilir. Kısmi kilit

sıralamasının mükemmel bir gerçek örneği, Linux [T+94] (v5.2)'deki bellek eşleme kodunda görülebilir; kaynak kodunun üst kısmındaki yorum,

İPUCU: KİLİT ADRESİNE GÖRE KİLİT SİPARİŞİNİ UYGULAYIN

Bazı durumlarda, bir işlev iki (veya daha fazla) kilit almalıdır; bu nedenle, dikkatli olmamız gerektiğini biliyoruz, aksi takdirde çıkmaz ortaya çıkabilir. Şu şekilde adlandırılan bir işlev düşünün: `do_something(mutex_t *m1, mutex_t *m2)`. Kod her zaman m1'i m2'den önce alırsa (veya her zaman m1'den önce m2'yi alırsa), bir iş parçacığı `do_something(L1, L2)`'yi çağırırken başka bir iş parçacığı `do_something (L1, L2)`'i çağırabileceğinden kilitlenebilir.

Bu özel sorundan kaçınmak için akıllı programcı, her bir kilidin *adresini*, kilit edinimi sipariş etmenin bir yolu olarak kullanabilir. `Do_something ()`, kilitleri yüksekten düşüğe veya düşükten yükseğe adres sırasına göre alarak, hangi sırada iletildiklerine bakmaksızın kilitleri her zaman aynı sırada almasını garanti edebilir. Kod şöyle görünür :

```
if (m1 > m2) { // grab in high-to-low address order
  pthread_mutex_lock(m1);
  pthread_mutex_lock(m2);
} else {
  pthread_mutex_lock(m2);
  pthread_mutex_lock(m1);
}
// Code assumes that m1 != m2 (not the same lock)
```

Bu basit tekniği kullanarak, bir programcı çoklu kilit ediniminin kilitlenmeden basit ve verimli bir şekilde uygulanmasını sağlayabilir.

" `i mutex` önce `i mmap rwsem` " gibi basit olanlar ve " `i mmap rwsem` önce `private lock` önce `i pages lock` 'tan önce `swap lock` " gibi daha karmaşık siparişler dahil olmak üzere on farklı kilit edinme emri grubunu ortaya koyuyor.

Tahmin edebileceğiniz gibi hem tam hem de kısmi sıralama, kilitleme stratejilerinin dikkatli bir şekilde tasarlanmasını gerektirir ve büyük bir dikkatle oluşturulmalıdır. Ayrıca, sıralama sadece bir kuraldır ve özensiz bir programcı kilitleme protokolünü kolayca görmezden gelebilir ve potansiyel olarak kilitlenmeye neden olabilir. Son olarak, kilit sıralaması, kod tabanının ve çeşitli rutinlerin nasıl adlandırıldığının derinlemesine anlaşılmasını gerektirir; sadece bir hata "D" kelimesiyle sonuçlanabilir[1].

Tut ve Bekle

Kilitlenme için tut ve bekle gereksinimi, tüm kilitleri aynı anda atomik olarak alarak önlenebilir. Uygulamada, bu şu şekilde elde edilebilir:

```
1  pthread_mutex_lock(prevention);   // begin acquisition
2  pthread_mutex_lock(L1);
3  pthread_mutex_lock(L2);
4  ...
5  pthread_mutex_unlock(prevention); // end
```

1 İpucu: "D", "Kilitlenme" anlamına gelir.

Bu kod, ilk önce kilit önlemeyi yakalayarak, kilit almanın ortasında zamansız bir iş parçacığı geçişinin meydana gelmeyeceğini ve böylece kilitlenmenin bir kez daha önlenebileceğini garanti eder. Tabii ki, herhangi bir iş parçacığının bir kilit alması durumunda, önce küresel önleme kilidini almasını gerektirir. Örneğin, başka bir iş parçacığı L1 ve L2 kilitlerini farklı bir sırada tutmaya çalışıyorsa, bunu yaparken önleme kilidini tutacağı için sorun olmaz.

Çözümün birkaç nedenden dolayı sorunlu olduğunu unutmayın. Daha önce olduğu gibi, kapsülleme bize karşı işliyor: Bir rutini çağırırken, bu yaklaşım tam olarak hangi kilitlerin tutulması gerektiğini bilmemizi ve onları önceden edinmemizi gerektiriyor. Tüm kilitlerin gerçekten ihtiyaç duyuldukları zaman yerine erken (bir kerede) edinilmesi gerektiğinden, bu tekniğin eşzamanlılığı da düşürmesi muhtemeldir.

Ön Alım Yok

Kilitleri genellikle kilit açma çağrılana kadar tutulmuş olarak gördüğümüz için, çoklu kilit edinimi çoğu zaman başımızı belaya sokar çünkü bir kilidi beklerken diğerini tutuyoruz. Birçok iş parçacığı kitaplığı, bu durumu önlemeye yardımcı olmak için daha esnek bir arabirim seti sağlar. Spesifik olarak, rutin `pthread_mutex_trylock()` ya kilidi alır (eğer varsa) ve başarıyı döndürür ya da kilidin tutulduğunu gösteren bir hata kodu döndürür; ikinci durumda, o kilidi almak istiyorsanız daha sonra tekrar deneyebilirsiniz.

Böyle bir arayüz, kilitlenme içermeyen, sıralı sağlam bir kilit edinme protokolü oluşturmak için aşağıdaki gibi kullanılabilir:

```
1 top:
2   pthread_mutex_lock(L1);
3   if (pthread_mutex_trylock(L2) != 0) {
4     pthread_mutex_unlock(L1);
5     goto top;
6   }
```

Başka bir iş parçacığının aynı protokolü izleyebileceğini, ancak kilitleri diğer sırayla (`L2` sonra `L1`) alabileceğini ve programın yine kilitlenmeden kurtulacağını unutmayın. Bununla birlikte, yeni bir sorun ortaya çıkıyor: **livelock.** İki iş parçacığının her ikisinin de bu sırayı tekrar tekrar denemesi ve her iki kilidi de tekrar tekrar elde edememesi mümkündür (muhtemelen olası değildir). Bu durumda, her iki sistem de bu kod dizisini tekrar tekrar çalıştırmaktadır (ve dolayısıyla bu bir kilitlenme değildir), ancak ilerleme kaydedilmemektedir, dolayısıyla adı livelock'tur. Livelock sorununun da çözümleri var: örneğin, geri döngüye girmeden ve her şeyi yeniden denemeden önce rastgele bir gecikme eklenebilir, böylece rakip ileti dizileri arasında tekrarlanan girişim olasılığı azaltılabilir.

Bu çözümle ilgili bir nokta: kullanmanın zor kısımlarını çevreliyor bir trylock yaklaşımı. Muhtemelen tekrar var olacak ilk sorun, kapsülleme nedeniyle ortaya çıkar: Bu kilitlerden biri çağrılan bir rutine gömülürse, başa geri

atlamanın uygulanması daha karmaşık hale gelir. Kod, yol boyunca bazı kaynaklar (L1 dışında) edinmişse, bunları da dikkatli bir şekilde serbest bıraktığından emin olmalıdır; örneğin, L1'i aldıktan sonra, kod bir miktar bellek ayırmışsa, tüm diziyi yeniden denemek için en başa atlamadan önce, L2'yi edinemediğinde bu belleği serbest bırakması gerekir. Bununla birlikte, sınırlı durumlarda (örneğin, daha önce bahsedilen Java vektör yöntemi), bu tür bir yaklaşım işe yarayabilir.

Ayrıca, bu yaklaşımın gerçekten önleme eklemediğini (kilidi ona sahip olan bir iş parçacığından zorla alma eylemi) eklemediğini, bunun yerine bir geliştiricinin kilit sahipliğinden geri adım atmasına izin vermek için trylock yaklaşımını kullandığını da fark edebilirsiniz (ör. kendi mülkiyeti) zarif bir şekilde. Ancak bu pratik bir yaklaşımdır ve bu bakımdan kusurlu olmasına rağmen onu buraya dahil ediyoruz.

Karşılıklı Dışlama

Nihai önleme tekniği, karşılıklı dışlama ihtiyacından tamamen kaçınmak olacaktır. Genel olarak bunun zor olduğunu biliyoruz çünkü çalıştırmak istediğimiz kodun gerçekten de kritik bölümleri var. Öyleyse ne yapabiliriz?

Herlihy, çeşitli veri yapılarının hiç kilit olmadan tasarlanabileceği fikrine sahipti [H91, H93]. Buradaki kilitsiz (ve ilgili **wait-free (beklemesiz)**) yaklaşımların arkasındaki fikir basittir: güçlü donanım yönergelerini kullanarak, açık kilitleme gerektirmeyen bir şekilde veri yapıları oluşturabilirsiniz.

Basit bir örnek olarak, hatırlayabileceğiniz gibi, aşağıdakileri yapan donanım tarafından sağlanan atomik bir komut olan bir karşılaştır ve değiştir talimatımız olduğunu varsayalım:

```
1  int CompareAndSwap(int *address, int expected, int new) {
2    if (*address == expected) {
3      *address = new;
4      return 1; // success
5    }
6    return 0; // failure
7  }
```

Şimdi karşılaştır ve değiştir özelliğini kullanarak bir değeri atomik olarak belirli bir miktarda artırmak istediğimizi hayal edin. Bunu aşağıdaki basit işlevle yapabiliriz:

```
1  void AtomicIncrement(int *value, int amount) {
2    do {
3      int old = *value;
4    } while (CompareAndSwap(value, old, old + amount) == 0);
5  }
```

Bir kilit almak, güncellemeyi yapmak ve ardından serbest bırakmak yerine, değeri tekrar tekrar yeni miktara güncellemeye çalışan ve bunu yapmak için

karşılaştır ve değiştir yöntemini kullanan bir yaklaşım geliştirdik. Bu şekilde, hiçbir kilit elde edilmez ve kilitlenme meydana gelmez (yine de canlı kilit hala bir olasılıktır ve bu nedenle sağlam bir çözüm, yukarıdaki basit kod parçacığından daha karmaşık olacaktır).

Biraz daha karmaşık bir örneği ele alalım: liste ekleme. İşte bir listenin başına ekleyen kod:

```
1 void insert(int value) {
2   node_t *n = malloc(sizeof(node_t));
3   assert(n != NULL);
4   n->value = value;
5   n->next  = head;
6   head     = n;
7 }
```

Bu kod basit bir ekleme gerçekleştirir, ancak "aynı anda" birden çok iş parçacığı tarafından çağrılırsa, bir yarış durumu vardır. Nedenini anlayabilir misin? (Her zaman olduğu gibi kötü amaçlı bir zamanlama serpiştirmesi varsayılarak, iki eşzamanlı ekleme gerçekleşirse bir listeye ne olabileceğinin bir resmini çizin). Elbette, bu kodu bir kilit alma ve bırakma ile çevreleyerek bunu çözebiliriz:

```
1 void insert(int value) {
2   node_t *n = malloc(sizeof(node_t));
3   assert(n != NULL);
4   n->value = value;
5   pthread_mutex_lock(listlock);   // begin critical section
6   n->next  = head;
7   head     = n;
8   pthread_mutex_unlock(listlock); // end critical section
9 }
```

Bu çözümde, kilitleri geleneksel şekilde kullanıyoruz[2]. Bunun yerine, sadece karşılaştır ve değiştir talimatını kullanarak bu eklemeyi kilitsiz bir şekilde gerçekleştirmeye çalışalım. İşte olası bir yaklaşım:

```
1 void insert(int value) {
2   node_t *n = malloc(sizeof(node_t));
3   assert(n != NULL);
4   n->value = value;
5   do {
6     n->next = head;
7   } while (CompareAndSwap(&head, n->next, n) == 0);
8 }
```

[2]Akıllı okuyucu, insert();'e girerken kilidi neden hemen değil de bu kadar geç kavradığımızı soruyor olabilir; zeki okuyucu, bunun neden muhtemelen doğru olduğunu anlayabilir misin? Kod, örneğin malloc() çağrısı hakkında hangi varsayımlarda bulunur?

Buradaki kod, bir sonraki işaretçiyi geçerli başlığa işaret edecek şekilde günceller ve ardından yeni oluşturulan düğümü listenin yeni başı olarak konuma getirmeye çalışır. Ancak, bu arada başka bir iş parçacığı yeni bir başlıkta başarılı bir şekilde değiştirilirse, bu iş parçacığının yeni başlıkla yeniden denenmesine neden olursa, bu başarısız olur.

Elbette, yararlı bir liste oluşturmak, yalnızca bir liste eklemekten daha fazlasını gerektirir ve kilitsiz bir şekilde içine ekleyebileceğiniz, silebileceğiniz ve üzerinde arama yapabileceğiniz bir liste oluşturmanın önemsiz olmaması şaşırtıcı değildir. Daha fazla bilgi edinmek için kilitsiz ve beklemesiz senkronizasyon hakkındaki zengin literatürü okuyun [H01, H91, H93].

Zamanlama Yoluyla Kilitlenmeden Kaçınma

Kilitlenme önleme yerine, bazı senaryolarda kilitlenmeden **avoidance (kaçınma)** tercih edilir. Kaçınma, çeşitli iş parçacıklarının yürütülürken hangi kilitleri tutabileceğine dair genel bir bilgi gerektirir ve daha sonra söz konusu iş parçacıklarını kilitlenme olmayacağını garanti edecek şekilde planlar.

Örneğin, iki işlemcimiz ve üzerlerinde programlanması gereken dört iş parçacığımız olduğunu varsayalım. Ayrıca, İş Parçacığı 1'in (T1) `L1`ve `L2`'yi (bir sırayla, yürütme sırasında bir noktada) kilitlediğini, T2'nin `L1` ve `L2`'yi de tuttuğunu, T3'ün yalnızca `L2`'yi tuttuğunu ve T4'ün hiç kilit tutmadığını bildiğimizi varsayalım. İş parçacıklarının bu kilit edinme taleplerini tablo halinde gösterebiliriz:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

Böylece akıllı bir programlayıcı, T1 ve T2 aynı anda çalıştırılmadığı sürece hiçbir kilitlenmenin ortaya çıkmayacağını hesaplayabilir. İşte böyle bir program:

(T3 ve T1) veya (T3 ve T2) için üst üste gelmenin uygun olduğunu unutmayın. T3, `L2` kilidini kapsa da yalnızca bir kilidi kaptığı için diğer iş parçacığı ile aynı anda çalışarak asla bir kilitlenmeye neden olamaz.

Bir örneğe daha bakalım. Bunda, aşağıdaki çekişme tablosunda gösterildiği gibi, aynı kaynaklar için (yine `L1` ve `L2` kilitleri) daha fazla çekişme vardır:

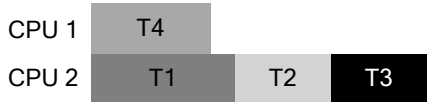
	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

İPUCU: HER ZAMAN MÜKEMMEL YAPMAYIN (TOM WEST YASASI)

Klasik bilgisayar endüstrisi kitabı *Soul of a New Machine'in* [K81] konusu olarak ünlenen Tom West, müthiş bir mühendislik özdeyişi olan "Yapmaya değer her şey iyi yapmaya değmez" der. Kötü bir şey nadiren oluyorsa, özellikle de kötü şeyin maliyeti küçükse, bunu önlemek için çok fazla çaba harcanmamalıdır. Öte yandan, bir uzay mekiği inşa ediyorsanız ve ters giden bir şeyin maliyeti uzay mekiğinin patlamasıysa, belki de bu tavsiyeyi göz ardı etmelisiniz.

Bazı okuyucular itiraz ediyor: "Sıradanlığı bir çözüm olarak öneriyor gibisin!" Belki de haklılar, bu tür tavsiyelerde dikkatli olmalıyız. Bununla birlikte, deneyimlerimiz bize mühendislik dünyasında, acil teslim tarihleri ve diğer gerçek dünya kaygılarıyla, bir sistemin hangi yönlerinin iyi inşa edileceğine ve hangilerinin başka bir güne bırakılacağına her zaman karar verilmesi gerektiğini söylüyor. Zor kısım, hangisini ne zaman yapacağını bilmektir, biraz içgörü ancak deneyim ve eldeki göreve kendini adama yoluyla kazanılır.

Özellikle, T1, T2 ve T3 iş parçacıklarının hepsinin, yürütülürken bir noktada hem `L1` hem de `L2` kilitlerini tutması gerekir. İşte hiçbir kilitlenmenin olmayacağını garanti eden olası bir program:



Gördüğünüz gibi, statik zamanlama, T1, T2 ve T3'ün hepsinin aynı işlemci üzerinde çalıştığı ve dolayısıyla işleri tamamlamak için gereken toplam sürenin önemli ölçüde uzadığı muhafazakâr bir yaklaşıma yol açar. Bu görevleri aynı anda yürütmek mümkün olsa da kilitlenme korkusu bizi bunu yapmaktan alıkoyar ve bunun maliyeti performanstır.

Bunun gibi bir yaklaşımın ünlü bir örneği Dijkstra'nın Banker Algoritması [D64]'dür ve literatürde benzer birçok yaklaşım tanımlanmıştır. Ne yazık ki, yalnızca çok sınırlı ortamlarda, örneğin çalıştırılması gereken tüm görevler ve ihtiyaç duydukları kilitler hakkında tam bilgiye sahip olunan gömülü bir sistemde yararlıdırlar. Ayrıca, yukarıdaki ikinci örnekte gördüğümüz gibi, bu tür yaklaşımlar eşzamanlılığı sınırlayabilir. Bu nedenle, zamanlama yoluyla kilitlenmeden kaçınmak, yaygın olarak kullanılan genel amaçlı bir çözüm değildir.

Tespit Et ve Kurtar

Son bir genel strateji, kilitlenmelerin ara sıra meydana gelmesine izin vermek ve ardından böyle bir kilitlenme algılandığında harekete geçmektir. Örneğin, bir işletim sistemi yılda bir kez donarsa, onu yeniden başlatır ve işinize mutlu (ya da

huysuz) devam edersiniz. Kilitlenmeler nadirse, böyle bir çözümsüzlük gerçekten oldukça pragmatiktir.

Birçok veritabanı sistemi, kilitlenme algılama ve kurtarma tekniklerini kullanır. Bir kilitlenme detektörü periyodik olarak çalışarak bir kaynak grafiği oluşturur ve döngüler için kontrol eder. Bir döngü (kilitlenme) durumunda, sistemin yeniden başlatılması gerekir. İlk önce veri yapılarının daha karmaşık onarımı gerekiyorsa, süreci kolaylaştırmak için bir insan dahil edilebilir.

Veritabanı eşzamanlılığı, kilitlenme ve ilgili sorunlar hakkında daha fazla ayrıntı başka bir yerde bulunabilir [B+87, K87]. Bu çalışmaları okuyun veya daha iyisi, bu zengin ve ilginç konu hakkında daha fazla bilgi edinmek için veritabanları üzerine bir kursa katılın.

32.4 Özet

Bu bölümde, eşzamanlı programlarda meydana gelen hata türlerini inceledik. İlk tip, kilitlenmeyen hatalar şaşırtıcı derecede yaygındır, ancak genellikle düzeltilmesi daha kolaydır. Bunlar, birlikte yürütülmesi gereken bir dizi talimatın uygulanmadığı atomiklik ihlallerini ve iki iş parçacığı arasında gerekli sıranın uygulanmadığı sıra ihlallerini içerir.

Kilitlenmeyi de kısaca tartıştık: neden oluşur ve bu konuda neler yapılabilir. Sorun, eşzamanlılığın kendisi kadar eskidir ve konu hakkında yüzlerce makale yazılmıştır. Uygulamadaki en iyi çözüm, dikkatli olmak, bir kilit alma emri geliştirmek ve böylece kilitlenmenin oluşmasını en başta önlemektir. Bazı beklemesiz veri yapıları artık yaygın olarak kullanılan kitaplıklara ve Linux da dahil olmak üzere kritik sistemlere girmeye başladığından, beklemesiz yaklaşımlar da umut vaat ediyor. Bununla birlikte, genellik eksiklikleri ve bekleme gerektirmeyen yeni bir veri yapısı geliştirmenin karmaşıklığı, bu yaklaşımın genel faydasını büyük olasılıkla sınırlayacaktır. Belki de en iyi çözüm, yeni eşzamanlı programlama modelleri geliştirmektir: MapReduce (Google'dan) [GD02] gibi sistemlerde, programcılar herhangi bir kilit olmaksızın belirli paralel hesaplama türlerini tanımlayabilirler. Kilitler doğası gereği sorunludur; belki de gerçekten mecbur kalmadıkça onları kullanmaktan kaçınmalıyız.

Referanslar

[B+87] "Concurrency Control and Recovery in Database Systems" by Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman. Addison-Wesley, 1987. *The classic text on concurrency in database management systems. As you can tell, understanding concurrency, deadlock, and other topics in the world of databases is a world unto itself. Study it and find out for yourself.*

[C+71] "System Deadlocks" by E.G. Coffman, M.J. Elphick, A. Shoshani. ACM Computing Surveys, 3:2, June 1971. *The classic paper outlining the conditions for deadlock and how you might go about dealing with it. There are certainly some earlier papers on this topic; see the references within this paper for details.*

[D64] "Een algorithme ter voorkoming van de dodelijke omarming" by Edsger Dijkstra. 1964. Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF. *Indeed, not only did Dijkstra come up with a number of solutions to the deadlock problem, he was the first to note its existence, at least in written form. However, he called it the "deadly embrace", which (thankfully) did not catch on.*

[GD02] "MapReduce: Simplified Data Processing on Large Clusters" by Sanjay Ghemawhat, Jeff Dean. OSDI '04, San Francisco, CA, October 2004. *The MapReduce paper ushered in the era of large-scale data processing, and proposes a framework for performing such computations on clusters of generally unreliable machines.*

[H01] "A Pragmatic Implementation of Non-blocking Linked-lists" by Tim Harris. International Conference on Distributed Computing (DISC), 2001. *A relatively modern example of the difficulties of building something as simple as a concurrent linked list without locks.*

[H91] "Wait-free Synchronization" by Maurice Herlihy . ACM TOPLAS, 13:1, January 1991. *Herlihy's work pioneers the ideas behind wait-free approaches to writing concurrent programs. These approaches tend to be complex and hard, often more difficult than using locks correctly, probably limiting their success in the real world.*

[H93] "A Methodology for Implementing Highly Concurrent Data Objects" by Maurice Herlihy. ACM TOPLAS, 15:5, November 1993. *A nice overview of lock-free and wait-free structures. Both approaches eschew locks, but wait-free approaches are harder to realize, as they try to ensure than any operation on a concurrent structure will terminate in a finite number of steps (e.g., no unbounded looping).*

[J+08] "Deadlock Immunity: Enabling Systems To Defend Against Deadlocks" by Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea. OSDI '08, San Diego, CA, December 2008. *An excellent recent paper on deadlocks and how to avoid getting caught in the same ones over and over again in a particular system.*

[K81] "Soul of a New Machine" by Tracy Kidder. Backbay Books, 2000 (reprint of 1980 version). *A must-read for any systems builder or engineer, detailing the early days of how a team inside Data General (DG), led by Tom West, worked to produce a "new machine." Kidder's other books are also excellent, including "Mountains beyond Mountains." Or maybe you don't agree with us, comma?*

[K87] "Deadlock Detection in Distributed Databases" by Edgar Knapp. ACM Computing Surveys, 19:4, December 1987. *An excellent overview of deadlock detection in distributed database systems. Also points to a number of other related works, and thus is a good place to start your reading.*

[L+08] "Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics" by Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou. ASPLOS '08, March 2008, Seattle, Washington. *The first in-depth study of concurrency bugs in real software, and the basis for this chapter. Look at Y.Y. Zhou's or Shan Lu's web pages for many more interesting papers on bugs.*

[T+94] "Linux File Memory Map Code" by Linus Torvalds and many others. Available online at: `http://lxr.free-electrons.com/source/mm/filemap.c`. *Thanks to Michael Walfish (NYU) for pointing out this precious example. The real world, as you can see in this file, can be a bit more complex than the simple clarity found in textbooks...*

Ev ödevi (Kod)

Bu ev ödevi, kilitlenen (veya kilitlenmeyi önleyen) bazı gerçek kodları keşfetmenizi sağlar. Farklı kod sürümleri, basitleştirilmiş bir `vector add()` rutininde kilitlenmeden kaçınmaya yönelik farklı yaklaşımlara karşılık gelir. Bu programlar ve ortak kaynakları hakkında ayrıntılar için BENİOKU'ya bakın.

Sorular

1. Öncelikle programların genel olarak nasıl çalıştığını ve bazı önemli seçenekleri anladığınızdan emin olalım. `vector-deadlock.c` ve ilgili dosyaların yanı sıra `main-common.c` deki kodu inceleyin.
   Şimdi, her biri bir vektör toplama (`-l 1`) yapan iki iş parçacığını (`-n 2`) başlatan ve bunu ayrıntılı modda (-v) yapan `./vector-deadlock -n 2 -l 1 -v` komutunu çalıştırın. Çıktıyı anladığınızdan emin olun. Çıktı çalıştırmadan çalıştırmaya nasıl değişir?

   Çıktı her çalıştırmada aynı sonucu vermektedir.

```
root@kali:~/Desktop/threads-bugs# ./vector-deadlock -n 2 -l 1 -v
                ->add(0, 1)
                <-add(0, 1)
->add(0, 1)
<-add(0, 1)
root@kali:~/Desktop/threads-bugs# ./vector-deadlock -n 2 -l 1 -v
                ->add(0, 1)
                <-add(0, 1)
->add(0, 1)
<-add(0, 1)
```

2. Şimdi `-d` bayrağını ekleyin ve döngü sayısını (`-l`) 1'den daha yüksek sayılara değiştirin. Ne oluyor? Kod (her zaman) kilitleniyor mu?

   Kod her zaman kilitlenmez. Aslında, 1.000 döngü (-l 1000) ile çalıştırana kadar kodu kilitleyemedim ve o zaman bile sona eriyordu.

```
root@kali:~/Desktop/threads-bugs# ./vector-deadlock -n 2 -l 3 -v -d
                ->add(1, 0)
                <-add(1, 0)
                ->add(1, 0)
                <-add(1, 0)
                ->add(1, 0)
                <-add(1, 0)
->add(0, 1)
<-add(0, 1)
->add(0, 1)
<-add(0, 1)
->add(0, 1)
<-add(0, 1)
```

3. İş parçacığı sayısını (`-n`) değiştirmek programın sonucunu nasıl değiştirir? Kilitlenme olmamasını sağlayan herhangi bir `-n` değeri var mı?

   Evet vardır (-n 1). Diğer tüm değerlerde kilitleniyor.

```
root@kali:~/Desktop/threads-bugs# ./vector-deadlock -n 1 -l 1 -v
->add(0, 1)
<-add(0, 1)
root@kali:~/Desktop/threads-bugs# ./vector-deadlock -n 3 -l 1 -v
                          ->add(0, 1)
                          <-add(0, 1)
             ->add(0, 1)
             <-add(0, 1)
->add(0, 1)
<-add(0, 1)
```

4. Şimdi `vector-global-order.c` içindeki kodu inceleyin. Öncelikle, kodun ne yapmaya çalıştığını anladığınızdan emin olun; kodun neden kilitlenmeyi önlediğini anlıyor musunuz? Ayrıca, kaynak ve hedef vektörler aynı olduğunda bu `vector_add()` rutininde neden özel bir durum var?

   Kod, kilitlenmeyi önler çünkü kilitlerin alınma sırası, vektör yapısının sanal bellek adresi tarafından tanımlanan toplam bir sıradır.

   Kaynak ve hedefin aynı olduğu özel bir durum vardır çünkü bu durumda yalnızca bir kilidin alınması gerekir. Bu özel durum olmadan, kod kilitlenmeyi garanti ederek halihazırda tuttuğu bir kilidi elde etmeye çalışırdı.

5. Şimdi kodu aşağıdaki bayraklarla çalıştırın: `-t -n 2 -l 100000 -d` . Kodun tamamlanması ne kadar sürer? Toplam süre nasıl değişir? Döngü sayısını veya iş parçacığı sayısını artırdığınızda toplam süre nasıl değişir?

   `./vector-global-order -t -n 2 -l 100000 -d` 0,1-0,2 saniyede tamamlanır ve dağılım 0,1'ye çok daha yakındır.

   -t -n 2 -d döngü uzunlukları 1e4, 1e5, 1e6 ve 1e7 çalıştırılması, çalışma süresinin döngü uzunluğuyla kabaca doğrusal olarak ölçeklendiğini gösterir.

   -t -l 100000 -d' yi iş parçacığı sayısı 2, 20 ve 99 ile çalıştırmak, çalışma süresinin iş parçacığı sayısıyla kabaca doğrusal olarak ölçeklendiğini gösterir.

```
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 2 -l 100000 -d
Time: 0.02 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 5 -l 100000 -d
Time: 0.04 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 2 -l 300000 -d
Time: 0.05 seconds
```

```
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 2 -l 100000 -d
Time: 0.02 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 20 -l 100000 -d
Time: 0.17 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 99 -l 100000 -d
Time: 0.82 seconds
```

6. Paralellik bayrağını (`-p`) açarsanız ne olur? Her iş parçacığı farklı vektörler eklemeye çalışırken (`-p`'nin sağladığı şey budur) aynı vektörler üzerinde çalışmaya kıyasla performansın ne kadar değişmesini beklersiniz?

   Programın daha hızlı tamamlanmasını bekliyorum ama ne kadar hızlı bilmiyorum. Paralelliği etkinleştirdikten sonra çalışma süresinin döngü uzunluğuyla doğrusal olarak ölçeklenmeye devam etmesini bekliyorum. Daha ilginç olanı, iş parçacığı

sayısına göre, mantıksal işlemci sayısına kadar değişmez olmasını ve ardından bu noktadan sonra doğrusal olarak (mantıksal işlemci sayısının katları halinde) ölçeklenmesini bekliyorum.

Bu rakamlar tahminlerime tam olarak uymuyor, ancak çok da uzak değiller. Doğrusal ölçeklendirme, 8 iş parçacığına ulaşana kadar tutulacak gibi görünmüyor. 5 veya daha az iş parçacığı için çalışma süresi sabite yakındır ve 8'den az iş parçacığı için ölçeklendirme alt doğrusaldır.

```
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 2 -l 100000 -p
Time: 0.01 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 3 -l 100000 -p
Time: 0.02 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 4 -l 100000 -p
Time: 0.03 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 5 -l 100000 -p
Time: 0.05 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 6 -l 100000 -p
Time: 0.05 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 7 -l 100000 -p
Time: 0.06 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 8 -l 100000 -p
Time: 0.07 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 9 -l 100000 -p
Time: 0.07 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 9 -l 100000
Time: 0.09 seconds
```

7. Şimdi `vector-try-wait.c` 'yi inceleyelim. Öncelikle kodu anladığınızdan emin olun. İlk `pthread_mutex_trylock()`-çağrısı gerçekten gerekli mi? Şimdi kodu çalıştırın. Global sipariş yaklaşımına kıyasla ne kadar hızlı çalışıyor? Kod tarafından sayılan yeniden deneme sayısı, iş parçacığı sayısı arttıkça nasıl değişir?

İlk aramaya gerçekten ihtiyaç var, hedef vektörü başka nasıl kilitleyeceksin?

Aşağıda, -t -l 100000 -d seçenekleriyle, iş parçacığı sayısı (-n) değişen çalışma süreleri verilmiştir:

```
root@kali:~/Desktop/threads-bugs# ./vector-try-wait -t -n 2 -l 100000 -d
Retries: 0
Time: 0.01 seconds
root@kali:~/Desktop/threads-bugs# ./vector-try-wait -t -n 4 -l 100000 -d
Retries: 3081990
Time: 0.18 seconds
root@kali:~/Desktop/threads-bugs# ./vector-try-wait -t -n 8 -l 100000 -d
Retries: 2328737
Time: 0.26 seconds
root@kali:~/Desktop/threads-bugs# ./vector-try-wait -t -n 16 -l 100000 -d
Retries: 48541726
Time: 8.31 seconds
```

Aşağıda, yüksek paralellik -t -l 100000 -d -p altında çalışma süreleri verilmiştir (yapı gereği, tüm çalıştırmaların sıfır yeniden denemesi vardır):

```
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 2 -l 100000 -d -p
Time: 0.02 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 4 -l 100000 -d -p
Time: 0.04 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 8 -l 100000 -d -p
Time: 0.07 seconds
root@kali:~/Desktop/threads-bugs# ./vector-global-order -t -n 16 -l 100000 -d -p
Time: 0.13 seconds
```

Yüksek çekişme altında hem harcanan süre hem de yeniden deneme sayısı süper doğrusal bir şekilde ölçeklenir (zaman, ikinci dereceden daha hızlı, ancak taban 2 ile üstelden daha yavaş ölçeklenir gibi görünüyor). Büyük O notasyonunda bu

ölçeklendirme, küresel kilit düzeni çözümünden çok daha kötü olsa da bu çözüm, az sayıda iş parçacığı için mutlak zamanda önemli ölçüde daha hızlıdır. 2 iş parçacığı için en az 5 kat daha hızlıdır ve küresel kilit düzeni çözümü yalnızca yaklaşık 11 iş parçacığını yakalar.

Yüksek paralellik altında (-p bayrağıyla) çalışma süreleri olağanüstüdür ve `vector-global-order` önce bir, sonra daha yüksek sayıda iş parçacığı için iki büyüklük sırası kadar geride bırakır.

8. Şimdi `vector-avoid-hold-and-wait.c` 'ye bakalım. Bu yaklaşımla ilgili temel sorun nedir? Hem `-p` ile hem de onsuz çalışırken performansı diğer sürümlerle nasıl karşılaştırılır?

Bu yaklaşımla ilgili temel sorun, çok kaba olmasıdır: (diğer tüm kilitlerin edinimini koruyan) küresel kilit, her iş parçacığı tarafından manipüle edilen vektörler farklı olsa bile çekişme altında olacaktır.

-t -l 100000 -d, değişken sayıda iş parçacığı -n:

```
root@kali:~/Desktop/threads-bugs# ./vector-avoid-hold-and-wait -t -n 2 -l 100000 -d
Time: 0.02 seconds
root@kali:~/Desktop/threads-bugs# ./vector-avoid-hold-and-wait -t -n 4 -l 100000 -d
Time: 0.04 seconds
root@kali:~/Desktop/threads-bugs# ./vector-avoid-hold-and-wait -t -n 8 -l 100000 -d
Time: 0.09 seconds
root@kali:~/Desktop/threads-bugs# ./vector-avoid-hold-and-wait -t -n 16 -l 100000 -d
Time: 0.16 seconds
```

-t -l 100000 -d -p (paralellik ile), değişen iş parçacığı sayısı -n:

```
root@kali:~/Desktop/threads-bugs# ./vector-avoid-hold-and-wait -t -n 2 -l 100000 -d -p
Time: 0.02 seconds
root@kali:~/Desktop/threads-bugs# ./vector-avoid-hold-and-wait -t -n 4 -l 100000 -d -p
Time: 0.04 seconds
root@kali:~/Desktop/threads-bugs# ./vector-avoid-hold-and-wait -t -n 8 -l 100000 -d -p
Time: 0.09 seconds
root@kali:~/Desktop/threads-bugs# ./vector-avoid-hold-and-wait -t -n 16 -l 100000 -d -p
Time: 0.16 seconds
```

Yoğun çekişme altındaki çalışma süreleri (-p bayrağı olmadan) `vector-global-order` için olanlara çok yakın. Yüksek paralellik (-p bayrağıyla) durumu için, `vector-avoid-hold-and-wait` yüksek çekişme altında olduğundan aşağı yukarı performans gösterir, ancak yüksek paralellik altında `vector-global-order` yaklaşık iki kat daha yavaştır. Sonuç olarak, `vector-global-order` ve `vector-try-wait` arasındaki karşılaştırma aşağı yukarı burada da geçerlidir.

9. Son olarak, `vector-nolock.c` 'ye bakalım. Bu sürüm hiç kilit kullanmaz; diğer sürümlerle tamamen aynı semantiği sağlıyor mu? Neden veya neden olmasın?

Hayır. Her giriş çiftini değil, bir çift giriş eklemeye göre yalnızca atomiktir.

10. Şimdi, hem iş parçacıkları aynı iki vektör üzerinde çalışırken (`-p` yok) hem de her iş parçacığı ayrı vektörler üzerinde çalışırken (`-p`) performansını diğer sürümlerle karşılaştırın. Bu kilitsiz sürüm nasıl performans gösteriyor?

Aynı iki vektör üzerinde çalışırken çalıştırma süreleri (yüksek çekişme):

2 threads: 0.4-0.5 secs

4 threads: ~0.4 secs

8 threads: 0.6-0.7 secs
16 threads: 1.3-1.4 secs
32 threads: 2.6-2.7 secs
Bu, `vector-global-order` 'dan yaklaşık iki kat daha hızlıdır.
Her iş parçacığının ayrı vektörler üzerinde çalıştığı çalışma süreleri (yüksek paralellik):
2 threads: ~0.06 secs
4 threads: ~0.12 secs
8 threads: ~0.24 secs
16 threads: ~0.49 secs
32 threads: ~0.97 secs
Bu neredeyse `vector-global-order` 'dan daha hızlı bir büyüklük sırasıdır.