32

Ortak Eşzamanlılık Sorunları

Araştırmacılar, uzun yıllar boyunca sahte para hatalarını araştırmak için çok zaman ve çaba harcadılar. İlk çalışmaların çoğu kilitlenmeye(deadlock) odaklandı, geçmiş bölümlerde değindiğimiz ancak şimdi derinlemesine dalacağımız bir konu şimdi derinlere dalın [C + 71]. Daha yeni çalışmalar, diğer yaygın eşzamanlılık hataları türlerini incelemeye odaklanmaktadır (yani, kilitlenmeyen hatalar) İçinde bu bölümde, bazı örnek eşzamanlılık sorunlarına kısa bir göz atacağız hangi sorunlara dikkat etmeniz gerektiğini daha iyi anlamak için gerçek kod tabanlarında bulunur için. Ve böylece bu bölüm için ana sorunumuz:

CRUX: ORTAK EŞZAMANLILIK HATALARI NASIL ELE ALINIR
Eşzamanlılık hataları, çeşitli ortak kalıplarda gelme eğilimindedir. Hangilerine dikkat etmeniz gerektiğini bilmek, daha sağlam ve doğru eşzamanlı kod yazmanın ilk adımıdır.

32.1 Ne Tür Hatalar Var?

İlk ve en belirgin soru şudur: karmaşık, eşzamanlı programlarda ne tür eşzamanlılık hataları ortaya çıkar? Bu soruyu genel olarak cevaplamak zor ama neyse ki işi bizim için başkaları da yaptı. Özellikle, Lu ve arkadaşlarının yaptığı bir araştırmaya güveniyoruz. Uygulamada ne tür hataların ortaya çıktığını anlamak için bir dizi popüler eşzamanlı uygulamayı ayrıntılı olarak analiz eden [L + 08].

Çalışma dört ana ve önemli açık kaynak uygulamasına odaklanmaktadır: MySQL (popüler bir veritabanı yönetim sistemi), Apache (iyi bilinen bir web sunucusu), Mozilla (ünlü web tarayıcısı) ve OpenOffice (ücretsiz bir sürümü) Bazı kişilerin gerçekten kullandığı MS Office paketi). 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 göstermektedir. Şekilden, çoğu toplam 105 hata olduğunu görebilirsiniz.

Uygulama	Ne yapar	Kilitlenme Yok	Kilitlenme
MySQL	Veritabanı Sunucusu	14	9
Apache	Web Sunucusu	13	4
Mozilla	Web Tarayıcı	41	16
OpenOffice	Ofis Paketi	6	2
Total		74	31

Figure 32.1: **Modern Uygulamalardaki Hatalar**

kilitlenme değildi (74); geriye kalan 31 kilitlenme hatasıydı. Ayrıca, her uygulamadan incelenen hata sayısını görebilirsiniz; openoffice'de yalnızca 8 toplam eşzamanlılık hatası varken, Mozilla'da yaklaşık 60 hata vardı. Şimdi bu farklı böcek sınıflarına (kilitlenmeyen, kilitlenmeyen) biraz daha derine dalıyoruz. Kilitlenmeyen hataların birinci sınıfı için, tartışmamızı yönlendirmek için çalışmadan örnekler kullanıyoruz. Kilitlenme hatalarının ikinci sınıfı için, kilitlenmeyi önleme, önleme veya ele alma konusunda yapılan uzun çalışma serisini tartışıyoruz.

32.2 Kilitlenmeyen Hatalar

Kilitlenmeyen hatalar, Lu'nun çalışmasına göre eşzamanlılık hatalarının çoğunu oluşturur. .Ama bunlar ne tür hatalar? Nasıl ortaya çıkıyorlar? Onları nasıl düzeltebiliriz? Şimdi Lu ve arkadaşları tarafından bulunan iki ana kilitlenmeyen hata türünü tartışıyoruz.: atomiklik ihlali(atomicity violation) hataları ve sipariş ihlali (order violation)hataları.

Atomiklik-İhlal Hataları

Karşılaşılan ilk sorun türü atomiklik ihlali(atomicity violation) olarak adlandırılır. İşte mysql'de bulunan basit bir örnek. Açıklamayı okumadan önce hatanın ne olduğunu bulmaya çalışın. Yap şunu!

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

Şekil 32.2: Atomiklik İhlali (atomiklik.orta)

Örnekte, iki farklı iş parçacığı bu yapıdaki işlem bilgisi alanına erişir. İlk iş parçacığı, değerin NULL olup olmadığını denetler ve ardından değerini yazdırır; ikinci iş parçacığı onu NULL olarak ayarlar. Açıkça, ilk iş parçacığı denetimi gerçekleştirir, ancak fputs çağrısından önce kesintiye uğrarsa, ikinci iş parçacığı aralarında çalışabilir ve böylece işaretçiyi NULL olarak ayarlayabilir; ilk iş parçacığı devam ettiğinde, fputs tarafından NULL bir işaretçinin başvurusu kaldırılacağından çökecektir.

Lu ve arkadaşlarına göre, bir atomiklik ihlalinin daha resmi tanımı şudur: "Çoklu bellek erişimleri arasında istenen seri hale getirilebilirlik ihlal edilmiştir (yani bir kod bölgesinin atomik olması amaçlanmıştır, ancak atomiklik yürütme sırasında zorlanmaz)." Yukarıdaki örneğimizde, kodun, NULL olmayan proc bilgisinin kontrolü ve fputs () çağrısında proc bilgisinin kullanımı hakkında bir atomiklik varsayımı (Lu'nun sözleriyle) vardır; Varsayım yanlış olduğunda, kod istenildiği gibi çalışmaz.

Bu tür bir sorun için bir düzeltme bulmak genellikle (ancak her zaman değil) basittir. Yukarıdaki kodu nasıl düzelteceğinizi düşünebiliyor musunuz? Bu çözümde (Şekil 32.3), paylaşılan değişken referanslarının etrafına kilitler ekleyerek, her iki iş parçacığı da proc bilgi alanına eriştiğinde bir kilidin tutulmasını sağlarız (proc bilgi kilidi). Tabii ki, yapıya erişen diğer kodlar da bunu yapmadan önce bu kilidi almalıdır.

```
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);
```

Şekil 32.3: Atomiklik İhlali Düzeltildi (atomiklik düzeltildi.orta)

Sipariş İhlali Hataları

Lu ve arkadaşları tarafından bulunan bir başka yaygın kilitlenme dışı hata türü. Sipariş ihlali(order violation) olarak bilinir. İşte başka bir basit örnek; Bir kez daha, aşağıdaki kodun neden içinde bir hata olduğunu anlayıp anlayamayacağınıza 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  }
```

Şekil 32.4: Sipariş Hatası (sipariş.orta)

Muhtemelen anladığınız gibi, iş Thread 2'deki kod, değişken iş parçacığının 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 }
```

Şekil 32.5: Sipariş İhlalinin Düzeltilmesi (sipariş düzeltildi.orta)

ancak, iş Thread 2 oluşturulduktan hemen sonra çalışırsa, İş Threadı 2'de mMain() içinde erişildiğinde mThread değeri ayarlanmayacak ve büyük olasılıkla bir boş işaretçi dereferansı ile çökecektir. İş parçacığının değerinin başlangıçta BOŞ olduğunu varsaydığımızı unutmayın; değilse, İş Parçacığı 2'deki derefer- ence aracılığıyla rasgele bellek konumlarına erişildiğinden daha garip şeyler olabilir.

Bir sipariş ihlalinin daha resmi tanımı şudur: "İki (grup) bellek erişimi arasında istenen sipariş çevrilir (yani, A her zaman B'den önce yürütülmelidir, ancak sipariş yürütme sırasında zorlanmaz)" [L + 08].

Bu tür bir hatanın düzeltilmesi genellikle siparişi zorlamaktır. Daha önce tartışıldığı gibi, koşul değişkenlerini(condition variables) kullanmak, bu senkronizasyon stilini modern kod tabanlarına eklemenin kolay ve sağlam bir yoludur. Yukarıdaki örnekte, kodu Şekil 32.5'te görüldüğü gibi yeniden yazabiliriz.

Bu sabit kod dizisinde, bir durum değişkeni (cond) ve karşılık gelen kilit (matlock) ile bir durum değişkeni ekledik

(mtInıt). Başlatma kodu çalıştığında, mt İnit durumunu 1 olarak ayarlar ve bunu yaptığını bildirir. Eğer Thread2 bu noktadan önce çalışmışsa, bu sinyali ve buna karşılık gelen durum değişikliğini bekliyor olacaktır; Daha sonra çalışırsa, durumu kontrol edecek ve başlatmanın zaten occurred olduğunu görecektir (yani, mtInit 1 olarak ayarlanmıştır) ve böylece uygun şekilde devam edecektir. mThread durum değişkeninin kendisi olarak kullanabileceğimizi unutmayın, ancak burada basitlik uğruna bunu yapmayın. Konular arasında sipariş verirken, durum değişkenleri (veya semaforlar) kurtarmaya gelebilir.

Kilitlenmeyen Hatalar: Özet

Kilitlenmeyen hataların büyük bir kısmı (% 97) Lu ve ark. atomiklik veya düzen ihlalleridir. Böylelikle, bu tür hata kalıplarını dikkatlice düşünerek, programcılar muhtemelen onlardan kaçınmak için daha iyi bir iş yapabilirler. Dahası, daha otomatik kod kontrol araçları geliştikçe, dağıtımda bulunan kilitlenmeyen hataların bu kadar büyük bir bölümünü oluşturdukları için muhtemelen bu iki tür hataya odaklanmaları gerekir.

Ne yazık ki, tüm hatalar yukarıda incelediğimiz örnekler kadar kolay düzeltilemez. Bazıları, programın ne yaptığını daha iyi anlamayı veya düzeltmek için daha büyük miktarda kod veya veri yapısını yeniden düzenlemeyi gerektirir. Lu ve ark.daha fazla ayrıntı için mükemmel (ve okunabilir) kağıt.

32.3 Kilitlenme Hataları

Yukarıda bahsedilen eşzamanlılık hatalarının ötesinde, karmaşık kilitleme protokollerine sahip birçok eşzamanlı sistemde ortaya çıkan klasik bir sorun, kilitlenme(deadlock) olarak bilinir. Kilitlenme, örneğin, bir iş parçacığı (örneğin, Thread 1) bir kilidi (L1) tutarken ve başka birini (L2) beklerken oluşur; ne yazık ki, L2 kilidini tutan iş parçacığı (Thread 2) L1'in serbest bırakılmasını bekliyor. İşte böyle olası bir kilitlenmeyi gösteren bir kod parçacığı:

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

Şekil 32.6: Basit Kilitlenme (kilitlenme.orta)

Bu kod çalışırsa, kilitlenme mutlaka oluşmaz unutmayın; bunun yerine, örneğin, iş parçacığı 1 kilit L1 kapmak ve sonra Thread 2 için bir içerik anahtarı oluşursa oluşabilir. Bu noktada, Thread 2 L2'yi alır ve L1'i edinmeye çalışır. Bu nedenle, her iş parçacığı diğerini beklediğinden ve ikisi de çalışamayacağından bir kilitlenmemiz var. Grafiksel bir tasvir için Şekil 32.7'ye bakın; Grafikte bir döngünün varlığı çıkmazın göstergesidir.

Şekil sorunu netleştirmelidir. Programcılar kilitlenmeyi bir şekilde ele almak için nasıl kod yazmalıdır?

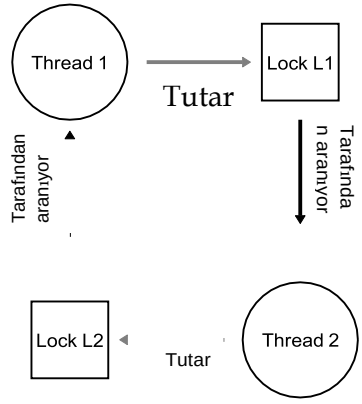


Figure 32.7: **The Deadlock Dependency Graph**

CRUX: KİLİTLENME İLE NASIL BAŞA ÇIKILIR
Çıkmazı önlemek, önlemek veya en azından tespit etmek ve bu çıkmazdan kurtulmak için sistemleri nasıl oluşturmalıyız? Bu, günümüzde sistemlerde gerçek bir sorun mu?

Kilitlenmeler Neden Oluşur?

Düşündüğünüz gibi, yukarıdaki gibi basit kilitlenmeler kolayca önlenebilir görünüyor. Örneğin, Thread 1 ve 2'nin her ikisi de kilitleri aynı sırada tuttuğundan emin olsaydı, kilitlenme asla ortaya çıkmazdı. Peki neden kilitlenmeler oluyor?

Bunun bir nedeni, büyük kod tabanlarında bileşenler arasında karmaşık bağımlılıkların ortaya çıkmasıdır. Örneğin işletim sistemini ele alalım. Sanal bellek sisteminin diskten bir blokta sayfa açmak için dosya sistemine erişmesi gerekebilir; Dosya sistemi daha sonra bloğu okumak ve böylece sanal bellek sistemiyle iletişim kurmak için bir sayfa belleğe ihtiyaç duyabilir. Bu nedenle, kodda doğal olarak oluşabilecek döngüsel bağımlılıklar durumunda kilitlenmeyi önlemek için büyük sistemlerde kilitleme stratejilerinin tasarımı dikkatli bir şekilde yapılmalıdır. Diğer bir neden ise kapsüllemenin(encapsulation) doğasından kaynaklanmaktadır. Yazılım geliştiriciler olarak, uygulamaların ayrıntılarını gizlememiz ve böylece yazılımın modüler bir şekilde oluşturulmasını kolaylaştırmamız öğretilir. Ne yazık ki, bu modülerlik kilitleme ile iyi bir şekilde örtüşmemektedir. Jula ve ark. [j + 08] 'e dikkat edin, görünüşte zararsız bazı arayüzler sizi neredeyse kilitlenmeye davet ediyor. Örneğin, Java Vektör sınıfını ve AddAll() yöntemini ele alalım. Bu rutin şu şekilde çağrılır:

```
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 (örneğin v1 sonra v2) alır. Başka bir iş parçacığı v2'yi çağırırsa.AddAll (v1) hemen hemen aynı zamanda, çağıran uygulamadan tamamen gizlenmiş bir şekilde kilitlenme potansiyeline sahibiz.

Kilitlenme Koşulları

- Bir kilitlenmenin meydana gelmesi için dört koşulun beklemesi gerekir [C + 71]:
- Karşılıklı dışlama(Mutual exclusion): İş parçacıkları, ihtiyaç duydukları kaynakların münhasır kontrolünü talep eder (örneğin, bir iş parçacığı bir kilidi yakalar).
- **Tutma ve bekle(**Hold-and-wait**):** İş parçacıkları, ek yeniden kaynaklar (örneğin, edinmek istedikleri kilitler) beklerken kendilerine tahsis edilen kaynakları (örneğin, daha önce edindikleri kilitler) tutar.
- **Önleme yok(**No preemption**):** Kaynaklar (ör. Kilitler), onları tutan iş parçacıklarından zorla kaldırılamaz.
- **Dairesel bekleme(Circular wait):** Dairesel bir iş parçacığı zinciri vardır, öyle ki her iş parçacığı zincirdeki bir sonraki iş parçacığı tarafından yeniden sorgulanan bir veya daha fazla kaynağı (örneğin kilitler) tutar.

* Bu dört koşuldan herhangi biri karşılanmazsa, kilitlenme meydana gelemez. Bu nedenle, önce kilitlenmeyi önlemeye yönelik teknikleri araştırıyoruz; Bu stratejilerin her biri, yukarıdaki koşullardan birinin ortaya çıkmasını önlemeye çalışır ve bu nedenle kilitlenme sorununu ele almak için bir yaklaşımdır.

Önleme

Dairesel Bekleme

Muhtemelen en pratik önleme tekniği (ve kesinlikle sıklıkla kullanılan), kilitleme kodunuzu asla dairesel bir beklemeye neden olmayacak şekilde yazmaktır. Bunu yapmanın en kolay yolu, kilit edinimi için toplam sipariş(total ordering) vermektir. Örneğin, sistemde yalnızca iki kilit varsa (L1 ve L2), L2'den önce her zaman L1'i edinerek kilitlenmeyi önleyebilirsiniz. Böyle katı bir düzen, döngüsel bir bekleyişin ortaya çıkmamasını sağlar; dolayısıyla kilitlenme yok.

Tabii ki, daha karmaşık sistemlerde ikiden fazla kilit mevcut olacaktır ve bu nedenle toplam kilit sırasının elde edilmesi zor olabilir (ve belki de yine de gereksizdir). Bu nedenle, kısmi bir sıralama(partial ordering), kilitlenmeyi önlemek için kilit alımını yapılandırmanın yararlı bir yolu olabilir . Kısmi kilit sıralamasının mükemmel bir gerçek örneği, Linux'taki bellek eşleme kodunda görülebilir [T + 94] (v5.2); Kaynak kodun en üstündeki yorum, basit olanlar da dahil olmak üzere on farklı kilit edinme sırası grubunu ortaya koymaktadır

İPUCU: KİLİT ADRESİNE GÖRE KİLİT SIRALAMASINI ZORLA

Bazı durumlarda, bir işlev iki (veya daha fazla) kilit tutmalıdır; Bu nedenle dikkatli olmamız gerektiğini biliyoruz, aksi takdirde kilitlenme meydana gelebilir. Aşağıdaki gibi adlandırılan bir işlev düşünün: `do something(mutex t *m1, mutex t *m2)`. Kod her zaman m2'den önce m1'i (veya m1'den önce her zaman m2'yi) alırsa, kilitlenebilir, çünkü bir thread arayabilir `do something(L1, L2)` başka bir iş parçacığı arayabilirken `do something(L2, L1)`.

Bu özel sorunu önlemek için, akıllı programcı her kilidin adresini kilit edinimi siparişi vermenin bir yolu olarak kullanabilir.Yüksekten düşüğe veya düşükten yükseğe adres sırasına göre kilitler edinerek, do something(), hangi sırayla iletildiklerine bakılmaksızın kilitleri her zaman aynı sırada almasını garanti edebilir. Kod böyle bir şeye benzeyecekti:

```
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)
```

By using this simple technique, a programmer can ensure a simple and efficient deadlock-free implementation of multi-lock acquisition.

”mmap rwsem'den önce muteks yapıyorum“ gibi olanlar ve ”sayfalar kilitlenmeden önce takas kilidinden önce özel kilitlemeden önce mmap rwsem" gibi daha karmaşık siparişler.

Tahmin edebileceğiniz gibi, hem tam hem de kısmi sıralama, kilitleme stratejilerinin dikkatli bir şekilde tasarlanmasını gerektirir ve büyük bir özenle inşa edilmelidir. Ayrıca, sipariş sadece bir kuraldır ve özensiz bir programcı kilitleme protokolünü kolayca görmezden gelebilir ve potansiyel olarak kilitlenmeye neden olabilir. Son olarak, kilit sıralaması, kod tabanının derinlemesine anlaşılmasını ve kod tabanının nasıl değiştirileceğini gerektirir.-

kullanıcı yordamları çağrılır; Yalnızca bir hata ”D" sözcüğüyle sonuçlanabilir1.

Tut ve Bekle:

Kilitlenme için bekle ve bekle gereksinimi, tüm kilitleri atomik olarak bir kerede alarak önlenebilir. Uygulamada, bu aşağıdaki gibi başarılabilir:

```
1   pthread_mutex_lock(prevention);   // satın almaya başla
2   pthread_mutex_lock(L1);
3   pthread_mutex_lock(L2);
4   ...
5   pthread_mutex_unlock(prevention); // son
```

[1]Hint: “D” stands for “Deadlock”.

Bu kod, önce kilit önlemeyi kaparak, kilit alımının ortasında zamansız bir iplik anahtarının oluşamayacağını ve böylece kilitlenmenin bir kez daha önlenebileceğini garanti eder. Tabii ki, herhangi bir iş parçacığı bir kilidi her kaptığında, önce küresel önleme kilidini edinmesini gerektirir. Örneğin, başka bir iş parçacığı L1 ve L2 kilitlerini farklı bir sırada yakalamaya çalışıyorsa, bunu yaparken önleme kilidini tutacağı için sorun olmaz.

Çözümün birkaç nedenden dolayı sorunlu olduğunu unutmayın. Daha önce olduğu gibi, kapsülleme bize karşı çalışır: bir rutini çağırırken, bu yaklaşım tam olarak hangi kilitlerin tutulması gerektiğini bilmemizi ve bunları önceden sorgulamamızı gerektirir. Bu tekniğin, tüm kilitlerin gerçekten ihtiyaç duyulduklarından ziyade erken (bir kerede) edinilmesi gerektiğinden, eşzamanlılığı da azaltması muhtemeldir.

Önleme Yok

Kilit açma çağrılana kadar kilitleri genellikle tutulan olarak gördüğümüzden, çoklu kilit edinimi genellikle başımızı belaya sokar çünkü bir kilidi beklerken diğerini tutuyoruz. Birçok thread kitaplığı, bu durumdan kaçınmaya yardımcı olmak için daha esnek bir arabirim kümesi sağlar. Özellikle, rutin pthread mutex trylock () ya kilidi alır (varsa) ve başarıyı döndürür ya da kilidin tutulduğunu belirten bir hata kodu döndürür; ikinci durumda, bu kilidi almak istiyorsanız daha sonra tekrar deneyebilirsiniz.

Böyle bir arayüz, kilitlenmeyen, siparişe dayanıklı bir kilit alma 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 sırada (L2 sonra L1) alabileceğini ve programın hala kilitlenmeyeceğini unutmayın. Ancak yeni bir sorun ortaya çıkıyor: canlı kitit (livelock). İki iş parçacığının her ikisinin de bu diziyi tekrar tekrar denemesi ve her iki kilidi de tekrar tekrar alamaması mümkündür (belki de olası değildir). Bu durumda, her iki sistem de bu kod dizisinden tekrar tekrar geçiyor (ve bu nedenle bir kilitlenme değil), ancak ilerleme kaydedilmiyor, dolayısıyla livelock adı. Livelock sorununa da çözümler var: örneğin, geri dönmeden önce rastgele bir gecikme eklenebilir ve her şeyi tekrar deneyebilir, böylece rakip iş parçacıkları arasında tekrarlanan müdahale olasılığı azaltılabilir.

Bu çözümle ilgili bir nokta: deneme kilidi yaklaşımını kullanmanın zor kısımlarının etrafını sarar. 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 bazı kaynaklar edinmiş olsaydı (L1 dışında)

yol boyunca, onları da dikkatlice serbest bıraktığından emin olmalıdır; Örneğin, kod L1'i aldıktan sonra bir miktar bellek ayırmış olsaydı, tüm diziyi tekrar denemek için en üste atlamadan önce, L2'yi edinemediğinde bu belleği serbest bırakması gerekirdi. Bununla birlikte, sınırlı durumlarda (örneğin, daha önce bahsedilen Java vektör yöntemi), bu tür bir yaklaşım iyi sonuç verebilir.

Ayrıca, bu yaklaşımın gerçekten önleme eklemediğini (kendisine sahip olan bir iş parçacığından bir kilit almanın zorunlu eylemi) değil, geliştiricinin kilit sahipliğinden vazgeçmesine izin vermek için trylock yaklaşımını kullandığını da fark edebilirsiniz. kendi sahipliğini) zarif bir şekilde. Ancak pratik bir yaklaşımdır ve bu nedenle bu konudaki kusuruna rağmen onu buraya dahil ediyoruz.

Karşılıklı Dışlanma

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

Herlihy, kilitsiz çeşitli veri yapıları tasarlayabileceği fikrine sahipti [H91, H93]. Buradaki bu kilitsiz(lock-free) (ve ilgili beklemesiz(wait-free)) yaklaşımların arkasındaki fikir basittir: güçlü donanım talimatlarını kullanarak, açık kilitlemeyi yeniden gerektirmeyecek şekilde veri yapıları oluşturabilirsiniz. Basit bir örnek olarak, hatırlayabileceğiniz gibi, aşağıdakileri yapan donanım tarafından sağlanan atomik bir talimat 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, bunun yerine değeri tekrar tekrar yeni miktara güncellemeye çalışan ve bunu yapmak için karşılaştır ve değiştir özelliğini kullanan bir yaklaşım geliştirdik. Bu şekilde,

hiçbir kilit elde edilmez ve hiçbir kilitlenme oluşamaz (ancak 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 eklenen 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 fazla iş parçacığı tarafından çağrılırsa, bir yarış koşuluna sahiptir. Nedenini anlayabiliyor musun? (her zaman olduğu gibi, kötü niyetli bir zamanlama serpiştirmesi varsayarak, iki eşzamanlı ekleme gerçekleşirse bir listeye ne olabileceğinin bir resmini çizin). Tabii ki, bunu bu kodu bir kilit alma ve bırakma ile çevreleyerek 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, geleneksel şekilde2 kilitler kullanıyoruz. Bunun yerine, sadece karşılaştır ve değiştir komutunu 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 }
```

2Akıllı okuyucu, insert()'e girerken kilidi neden doğru değil de bu kadar geç tuttuğumuzu soruyor olabilir; Zeki okur, bunun neden doğru olduğunu anlayabilir misin? Kod, örneğin malloc() çağrısı hakkında hangi varsayımları yapar?

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

Elbette, kullanışlı bir liste oluşturmak, yalnızca bir liste eklemesinden fazlasını gerektirir ve şaşırtıcı olmayan bir şekilde, içine ekleyebileceğiniz, silebileceğiniz ve üzerinde arama yapabileceğiniz bir liste oluşturmak, kilitsiz bir şekilde önemsiz değildir.Daha fazla bilgi edinmek için kilitsiz ve beklemesiz senkronizasyonla ilgili zengin literatürü okuyun [H01, H91, H93].

Zamanlama ile Kilitlenmeden Kaçınma

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

Örneğin, üzerinde programlanması gereken iki işlemcimiz ve dört iş parçacığımız olduğunu varsayalımAyrıca, İplik 1'in (T1) L1 ve L2 kilitlerini (bir sırayla, yürütülmesi sırasında bir noktada), T2'nin L1 ve L2'yi de tuttuğunu, T3'ün sadece L2'yi tuttuğunu ve T4'ün hiç kilit tutmadığını bildiğimizi varsayalım. Dişlerin bu kilit alma taleplerini tablo halinde gösterebiliriz:

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

Akıllı bir zamanlayıcı böylece T1 ve T2 aynı anda çalıştırılmadığı sürece hiçbir kilitlenmenin oluşamayacağını hesaplayabilir. İşte böyle bir program:

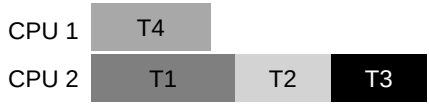
(T3 ve T1) veya (T3 ve T2) için üst üste binmenin uygun olduğunu unutmayın. T3, L2 kilidini tutsa da, yalnızca bir kilidi kaptığı için diğer iş parçacıklarıyla aynı anda çalışarak hiçbir zaman kilitlenmeye neden olamaz.

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

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

İPUCU: BUNU HER ZAMAN MÜKEMMEL YAPMAYIN (TOM WEST YASASI)
Klasik bilgisayar endüstrisi kitabının konusu olarak ünlü Tom West Yeni Bir Makinenin Ruhu [K81], ünlü bir şekilde şöyle diyor: "Yapmaya değer her şey iyi yapmaya değmez", ki bu müthiş bir mühendislik özdeyişidir. Kötü bir şey nadiren olursa, özellikle meydana gelen kötü şeyin maliyeti küçükse, bunu önlemek için kesinlikle çok fazla çaba harcamamalısınız. Öte yandan, bir uzay mekiği inşa ediyorsanız ve bir şeylerin ters gitmesinin bedeli uzay mekiğinin havaya uçması ise, belki de bu tavsiyeyi görmezden gelmelisiniz.
Bazı okuyucular itiraz ediyor: "Bu, çözüm olarak sıradanlığı öneriyormuşsunuz gibi geliyor!" Belki de bunun gibi tavsiyelere dikkat etmemiz gerektiği konusunda haklılar. Bununla birlikte, deneyimlerimiz bize mühendislik dünyasında, son teslim tarihlerine ve diğer gerçek dünya endişelerine bağlı olarak, bir sistemin hangi yönlerinin iyi inşa edileceğine ve hangilerinin bir gün daha bir kenara bırakılacağına her zaman karar verilmesi gerektiğini söylüyor. Zor kısım, hangisinin ne zaman yapılacağını bilmek, yalnızca eldeki göreve olan deneyim ve özveri ile kazanılan biraz içgörü.

Özellikle, T1, T2 ve T3 iş parçacıklarının hepsinin yürütülmesi sırasında bir noktada hem L1 hem de L2 kilitlerini tutması gerekir. İşte hiçbir kilitlenmenin meydana gelemeyeceğini garanti eden olası bir program:



Gördüğünüz gibi, statik zamanlama, T1, T2 ve T3'ün hepsinin aynı işlemci üzerinde çalıştırıldığı muhafazakar bir yaklaşıma yol açar ve bu nedenle işleri tamamlamak için toplam süre önemli ölçüde uzar. Bu görevleri aynı anda yürütmek mümkün olsa da, kilitlenme korkusu bunu yapmamızı engelliyor ve maliyet performanstır.

Bunun gibi bir yaklaşımın ünlü bir örneği Dijkstra'nın Banker's Algorithm'idir [D64] ve literatürde birçok benzer yaklaşım tanımlanmıştır. Ne yazık ki, yalnızca çok sınırlı ortamlarda, örneğin, çalıştırılması gereken tüm görevler kümesinin ve ihtiyaç duydukları kilitlerin tam bilgi birikimine sahip olduğu gömülü bir sistemde kullanışlıdırlar. Ayrıca, bu tür yaklaşımlar, yukarıdaki ikinci örnekte gördüğümüz gibi eşzamanlılığı sınırlayabilir. Bu nedenle, zamanlama yoluyla çıkmazdan kaçınmak, yaygın olarak kullanılan genel amaçlı bir çözüm değildir.

Algıla ve Kurtar

Son bir genel strateji, zaman zaman kilitlenmelerin meydana gelmesine izin vermek ve ardından böyle bir kilitlenme tespit edildikten sonra biraz harekete geçmektir.

Örneğin, bir işletim sistemi yılda bir kez donarsa, onu yeniden başlatır ve işinize mutlu (veya huysuzca) devam edersiniz. Kilitlenmeler nadirse, böyle bir çözümsüzlük gerçekten oldukça pragmatiktir.

Birçok veritabanı sistemi kilitlenme algılama ve kurtarma teknikleri kullanır. Bir kilitlenme detektörü periyodik olarak çalışır, bir kaynak grafiği oluşturur ve döngüleri kontrol eder. Bir döngü (kilitlenme) durumunda, sistemin yeniden başlatılması gerekir. Önce veri yapılarının daha karmaşık onarımına ihtiyaç duyulursa, süreci kolaylaştırmak için bir insan dahil olabilir. 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 kurs alın.

32.3 Özet

Bu bölümde, eşzamanlı programlarda meydana gelen hata türlerini inceledik. İlk tür, kilitlenmeyen hatalar şaşırtıcı derecede yaygındır, ancak çoğu zaman düzeltilmesi daha kolaydır. Bunlar, elde etmek için yürütülmesi gereken bir dizi talimatın uygulanmadığı atomiklik ihlallerini ve iki iş parçacığı arasında gerekli düzenin uygulanmadığı düzen ihlallerini içerir.

Kilitlenmeyi de kısaca tartıştık: neden ortaya çıkıyor ve bu konuda ne yapılabilir. Sorun eşzamanlılığın kendisi kadar eskidir ve konuyla ilgili yüzlerce makale yazılmıştır. Uygulamada en iyi çözüm dikkatli olmak, bir kilit alma emri geliştirmek ve böylece ilk etapta kilitlenmenin oluşmasını önlemektir. Bazı beklemesiz veri yapıları artık Linux da dahil olmak üzere yaygın olarak kullanılan kitaplıklara ve kritik sistemlere girmenin yolunu bulduğundan, beklemesiz yaklaşımların da umut vaat ediyor. Bununla birlikte, genellik eksikliği ve yeni bir beklemesiz veri yapısı geliştirme karmaşıklığı, muhtemelen bu yaklaşımın genel faydasını sınırlayacaktır. Belki de en iyi çözüm, yeni eşzamanlı programlama modelleri geliştirmektir: MapReduce (Google'dan) [GD02] gibi sistemlerde programcılar, herhangi bir kilit olmadan belirli paralel hesaplama türlerini tanımlayabilirler. Kilitler doğası gereği sorunludur; belki de gerçekten gerekmedikçe onları kullanmaktan kaçınmalıyız.

Referanslar

[B +87] Philip A. Bernstein, Vas- sos Hadzilacos, Nathan Goodman tarafından "Veritabanı Sistemlerinde Eşzamanlılık Kontrolü ve Kurtarma". Addison-Wesley, 1987. Veritabanı yönetim sistemlerinde eşzamanlılık üzerine klasik metin. Söyleyebileceğiniz gibi, veritabanları dünyasındaki eşzamanlılık, kilitlenme ve diğer konuları anlamak başlı başına bir dünyadır. Çalış ve kendin öğren.

[C +71] E.G. Coffman, M.J. Elphick, A. Shoshani'nin "Sistem Kilitlenmeleri". ACM Hesaplama Araştırmaları, 3: 2, Haziran 1971. Kilitlenme koşullarını ve bununla nasıl başa çıkabileceğinizi özetleyen klasik makale. Bu konuyla ilgili kesinlikle daha önceki bazı makaleler var; Ayrıntılar için bu makaledeki referanslara bakın.

[D64] Edsger Dijkstra tarafından "Een algorithme ter voorkoming van de dodelijke omarming". 1964. Mevcut: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF . Gerçekten de Dijkstra, kilitlenme sorununa yalnızca bir dizi çözüm bulmakla kalmadı, varlığını en azından yazılı olarak ilk not eden oydu. Ancak, buna (neyse ki) yetişmeyen "ölümcül kucaklama" adını verdi.

[GD02] "MapReduce: Büyük Kümelerde Basitleştirilmiş Veri İşleme", Sanjay Ghemawhat, Jeff Dean. OSDİ '04, San Francisco, CA, Ekim 2004. MapReduce makalesi, büyük ölçekli veri işleme çağını başlattı ve bu tür hesaplamaları genellikle güvenilmez makine kümeleri üzerinde gerçekleştirmek için bir çerçeve önerdi.

[H01] Tim Harris'in "Engellenmeyen Bağlantılı listelerin Pragmatik Bir Uygulaması". Uluslararası Dağıtılmış Bilgi İşlem Konferansı (DISC), 2001. Kilitsiz eşzamanlı bağlantılı bir liste kadar basit bir şey oluşturmanın zorluklarına nispeten modern bir örnek.

[H91] Maurice Herlihy'den "Beklemesiz Senkronizasyon" . ACM TOPLAS, 13:1, Ocak 1991. Herlihy'nin çalışmaları, eşzamanlı programlar yazmaya yönelik beklemesiz yaklaşımların ardındaki fikirlere öncülük ediyor. Bu yaklaşımlar karmaşık ve zor olma eğilimindedir, genellikle kilitleri doğru kullanmaktan daha zordur ve muhtemelen gerçek dünyadaki başarılarını sınırlar.

[H93] Maurice Her- lihy'nin "Yüksek Oranda Eşzamanlı Veri Nesnelerini Uygulamak için Bir Metodoloji". ACM TOPLAS, 15:5, Kasım 1993. Kilitsiz ve beklemesiz yapılara güzel bir genel bakış. Her iki yaklaşım da kilitlerden kaçınır, ancak eşzamanlı bir yapı üzerindeki herhangi bir işlemin sonlu sayıda adımda sona ermesini sağlamaya çalıştıkları için beklemesiz yaklaşımların gerçekleştirilmesi daha zordur (örneğin, sınırsız döngü yok).

[J + 08] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea'nın "Kilitlenme Bağışıklığı: Sistemlerin Kilitlenmelere Karşı Savunmasını Sağlamak". OSDI '08, San Diego, CA, Aralık 2008. Kilitlenmeler ve belirli bir sistemde tekrar tekrar aynı kilitlenmelere yakalanmaktan nasıl kaçınılacağına dair mükemmel bir makale.

[K81] Tracy Kidder'ın "Yeni Bir Makinenin Ruhu". Back bay Kitapları, 2000 (1980 ver.-

sion). Tom West liderliğindeki Data General (DG) içindeki bir ekibin "yeni bir makine" üretmek için nasıl çalıştığının ilk günlerini ayrıntılarıyla anlatan herhangi bir sistem üreticisi veya mühendisi için okunması gereken bir kitap." Kidder'ın diğer kitapları da "Dağların Ötesindeki Dağlar " da dahil olmak üzere mükemmel." Ya da belki bizimle aynı fikirde değilsin, virgül?

[K87] Edgar Knapp tarafından "Dağıtılmış Veritabanlarında Kilitlenme Tespiti". ACM Bilgi İşlem Araştırmaları, 19: 4, Aralık 1987. Dağıtılmış veritabanı sistemlerinde kilitlenme algılamaya mükemmel bir genel bakış. Ayrıca bir dizi başka ilgili esere de işaret eder ve bu nedenle okumaya başlamak için iyi bir yerdir.

[L +08] Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou'nun "Hatalardan Öğrenme - Gerçek Dünya Eşzamanlılık Hatası Özellikleri Üzerine Kapsamlı Bir Çalışma". ASPLOS '08, Mart 2008, Seattle, Washington. Gerçek yazılımdaki eşzamanlılık hatalarının ilk derinlemesine incelenmesi ve bu bölümün temeli. Böceklerle ilgili daha birçok ilginç makale için Y.Y. Zhou'nun veya Shan Lu'nun web sayfalarına bakın.

[T + 94] Linus Torvalds ve diğerleri tarafından "Linux Dosya Bellek Haritası Kodu". Çevrimiçi olarak şu adresten temin edilebilir: http://lxr.free-electrons.com/source/mm/filemap.c . Bu değerli örneğe işaret ettiği için Michael Walfish'e (NYU) teşekkürler. Bu dosyada gördüğünüz gibi gerçek dünya, ders kitaplarında bulunan basit netlikten biraz daha karmaşık olabilir...

Ödev (Kod)

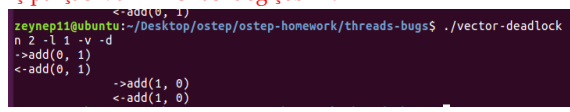
Bu ödev, kilitlenmeyen (veya kilitlenmeyi önleyen) bazı gerçek kodları keşfetmenizi sağlar. Kodun farklı sürümleri, basitleştirilmiş bir vector add () yordamında kilitlenmeyi önlemek için farklı yaklaşımlara karşılık gelir. Bu programlar ve bunların ortak alt katmanları hakkında ayrıntılar için BENİOKU sayfasına bakın.

Sorular

1. Öncelikle, programların genel olarak nasıl çalıştığını ve bazı temel seçenekleri anladığınızdan emin olalım. Vektör kilitlenmesindeki kodu inceleyin.c, yanı sıra ana- ortak.c ve ilgili dosyalar.

   Şimdi run ./vector-deadlock -n 2 -l 1 -v, her biri bir vektör ekleyen iki thread(-n 2) başlatır (-l 1) ve bunu ayrıntılı modda yapar (-v). Çıktıyı anladığınızdan emin olun. Çıktı çalıştırmadan çalıştırmaya nasıl değişir?

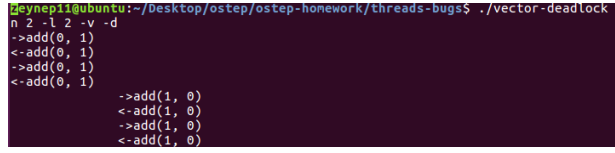
   İş parçacıklarının sırası değişebilir.



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

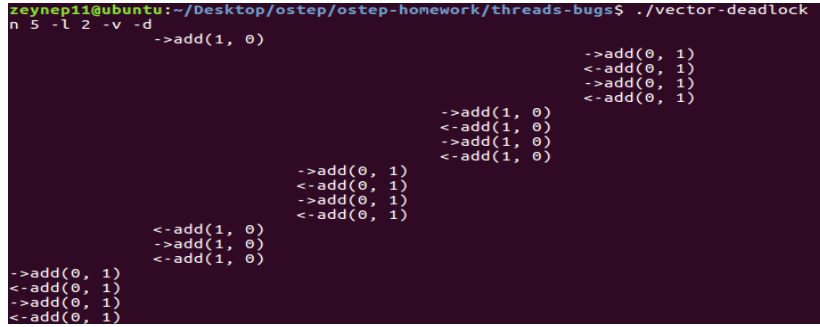
   Her zaman değil, ama bazen. (-l)ye 1 eklenmiş hali. Addler çoğalır.

   Kod her zaman kilitlenmez. Aslında, kodu 1.000 döngü (-l 1000) ile çalıştırana kadar kilitlenemedim ve o zaman bile olmadığı kadar sık sona erecekti.



3. İş parçacığı sayısını (-n) değiştirmek programın sonucunu nasıl değiştirir? Herhangi bir kilitlenme oluşmamasını sağlayan -n değerleri var mı?

   Evet. Set -n 1. Diğer tüm değerler kilitleniyor. 0,1



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

Kilitlerin alındığı sıra, vektör yapısının sanal bellek adresiyle tanımlanan toplam bir sıra olduğundan, kod kilitlenmeyi önler.
Kaynak ve hedefin aynı olduğu özel bir durum vardır, çünkü bu durumda yalnızca bir kilidin alınması gerekir
Bu özel durum olmadan, kod zaten tuttuğu bir kilidi almaya çalışarak bir kilitlenmeyi garanti ederdi.
Aynı kilidi iki kez kilitlememeli.

```
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ gcc -o vector-globa
l-order vector-global-order.c -Wall -pthread -O
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$
```

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

n ve l arttıkça saniyeler artar.

```
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-global-ord
er -t -n 2 -l 200000 -d
Time: 0.04 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-global-ord
er -t 2 -n 2 -l 100000 -d
Time: 0.03 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-global-ord
er -t 2 -n 2 -l 200000 -d
Time: 0.04 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-global-ord
er -t 3 -n 2 -l 200000 -d
Time: 0.06 seconds
```

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

Artık ipliklerin aynı iki kilidi beklemesine gerek yok.
Programın daha çabuk bitmesini bekliyorum ama ne kadar çabuk biteceğini bilmiyorum. Paralelliği etkinleştirdikten sonra çalışma süresinin döngü uzunluğu ile doğrusal olarak ölçeklenmeye devam etmesini bekliyorum. Daha ilginç bir şekilde, mantıksal işlemci sayısına kadar iş parçacığı sayısına göre değişmez olmasını ve daha sonra doğrusal olarak (mantıksal işlemci sayısının katları olarak) ölçeklenmesini bekliyorum. bu noktadan sonra.

```
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-global-ord
er -t 3 -n 2 -l 200000 -d -p
Time: 0.03 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-global-ord
er -t 2 -n 2 -l 200000 -d -p
Time: 0.03 seconds
```

7. Şimdi vector-try-wait.c. Önce kodu anladığınızdan emin olun. Pthread mutex trylock() öğesine yapılan ilk çağrıya gerçekten ihtiyaç var mı? Şimdi kodu çalıştır. Küresel düzen yaklaşımına kıyasla ne kadar hızlı çalışır? Kod tarafından sayılan yeniden deneme sayısı, iş parçacığı sayısı arttıkça nasıl değişir?

Deneme arttıkça süreler artar.

```
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ gcc -o vector-try-wait vector-try-wait.c -Wall -pthread -O
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-try-wait
Retries: 3
```

```
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-try-wait
Retries: 1
```

```
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-try-wait
Retries: 2
```

8. Şimdi vector-avoid-hold-and-wait'e bakalım.c. Bu yaklaşımın temel sorunu nedir? Hem -p ile hem de onsuz çalışırken performansı diğer sürümlerle nasıl karşılaştırılır?

Aynı anda yalnızca bir iş parçacığının eklenmesine izin verilir. Bu yaklaşımla ilgili temel sorun, çok kaba olmasıdır: küresel kilit (diğer tüm kilitlerin alınmasını koruyan), her iş parçacığı tarafından manipüle edilen vektörler farklı olsa bile çekişme altında olacaktır.

```
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -l 100000 -d -p
Time: 0.03 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -l 200000 -d -p
Time: 0.06 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -l 300000 -d -p
Time: 0.09 seconds
```

```
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -l 100000 -d
Time: 0.04 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -l 200000 -d
Time: 0.07 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -l 400000 -d
Time: 0.13 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -l 800000 -d
Time: 0.25 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$
```

9. Son olarak, vector-nolock.c. Bu sürüm hiç kilit kullanmıyor; diğer sürümlerle aynı anlambilimi sağlıyor mu? Neden ya da neden olmasın?
   Hayır. Her giriş çifti değil, yalnızca bir çift giriş ekleme açısından atomiktir. Kilit yerine getir ve ekle'yi kullanır.

10. Şimdi, hem iş parçacıkları aynı iki vektör üzerinde çalışırken(no -p) hem de her iş parçacığı ayrı vektörler üzerinde çalışırken (-p) performansını diğer sürümlerle karşılaştırın. Bu kilitsiz sürüm nasıl performans gösteriyor?
    -p eklediğimiz zaman daha hızlı çalışıyor.

```
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-global-ord
er -t -l 200000 -d
Time: 0.06 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-global-ord
er -t -l 400000 -d
Time: 0.09 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-global-ord
er -t -l 200000 -d -p
Time: 0.03 seconds
zeynep11@ubuntu:~/Desktop/ostep/ostep-homework/threads-bugs$ ./vector-global-ord
er -t -l 400000 -d -p
Time: 0.06 seconds
```