32

Ortak Eşzamanlılık Sorunları

Araştırmacılar, uzun yıllar boyunca eşzamanlılık hatalarını araştırmak için çok fazla zaman ve çaba harcadılar. İlk çalışmaların çoğu, önceki bölümlerde değindiğimiz ancak şimdi derinlemesine inceleyeceğimiz bir konu olan **kilitlenme(dead-lock)** 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. Ve böylece bu bölümdeki ana konumuz:

İpucu: ORTAK EŞ ZAMANLILIK HATALARINI NASIL ELE ALINIR?

Eşzamanlılık hataları, çeşitli ortak desenlerde bulunma yönleri. Hangilerine dikkat edildiğini bilmek, daha sağlam, doğru hızla 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. Pratikte 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 (tanınmış bir web sunucusu), Mozilla (ünlü web tarayıcısı) ve OpenOffice (MS Office'in ücretsiz bir sürümü). bazı insanların gerçekten kullandığı süit). 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 toplam 105 hata olduğunu görebilirsiniz.

Uygulama	Ne işe yarar?	Kilitlenme Olmayan	Çıkmaz
MySQL	Veritabanı Sunucusu	14	9
Apache	Web sunucusu	13	4
Mozilla	Web tarayıcısı	41	16
OpenOffice	Office Paketi	6	2
Toplam		74	31

Şekil 32.1: **Modern Uygulamalardaki Hatalar**

kilitlenme değildi (74); kalan 31 tanesi kilitlenme hatalarıydı. 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 arkadaşları tarafından bulunan kilitlenme olmayan iki ana hata türünü tartışacağız: atomiklik ihlali hataları(**atomicity violation**) ve düzen 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!

```
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 (`atomicity.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 daha sonra fputs çağrısından önce kesilirse, ikinci iş parçacığı arada çalışabilir ve böylece 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 çok bellek erişimi arasında istenen seri hale getirilebilirlik ihlal edildi (yani, bir kod bölgesinin atomik olması amaçlanıyor, ancak atomiklik yürütme sırasında zorunlu kılınmıyor). Yukarıdaki örneğimizde, kodun, işlem bilgisinin NULL olmayan olup olmadığının kontrolü ve fputs() çağrısında işlem bilgisinin kullanımı hakkında bir atomiklik varsayımı vardır (Lu'nun sözleriyle); 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 işlem bilgi alanına eriştiğinde, bir kilidin (proc info lock) tutulmasını sağlıyoruz. 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 (atomicity fixed.c)**

Sipariş İhlali Hataları

Lu ve arkadaşları tarafından bulunan başka bir yaygın kilitlenme dışı hata türü. 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  }
```

Şekil 32.4: **Sipariş Hatası (`ordering.c`)**

Muhtemelen anladığınız gibi, Thread 2'deki kod mThread değişkeninin zaten başlatılmış olduğunu (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: **Sıralama İhlalini Düzeltme (`Ordering fixed.c`)**

ancak, Thread 2 oluşturulduktan hemen sonra çalışırsa, Thread değeri, Thread 2'deki main() içinden erişildiğinde ayarlanmayacak ve muhtemelen bir NULL-işaretçi başvurusuyla çökecektir. mThread değerinin başlangıçta NULL olduğunu varsaydığımızı unutmayın; değilse, İş Parçacığı 2'deki referans 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, 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 değişkeninin (mtCond) ve karşılık gelen kilidin (mtLock) yanı sıra bir durum değişkeni ekledik (`mtInit`). Başlatma kodu çalıştığında, mtInit'in durumunu 1'e ayarlar ve bunu yaptığını bildirirİş 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

ayarlanmış) 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

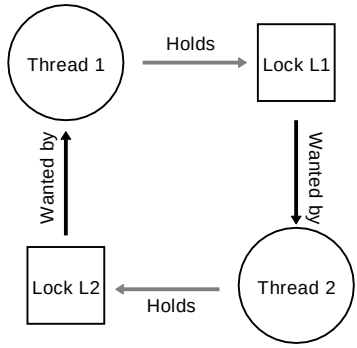
Yukarıda belirtilen eşzamanlılık hatalarının ötesinde, karmaşık kilitleme protokollerine sahip birçok eşzamanlı sistemde ortaya çıkan klasik bir sorun kilitlenme olarak bilinir. Kilitlenme, örneğin, bir iş parçacığı (diyelim ki İş Parçacığı bir kilidi tutuyor (L1) ve diğerini bekliyor (L2)), ne yazık ki, L2 kilidini tutan iş parçacığı (İplik 2) L1'in serbest bırakılmasını bekliyor. böyle bir potansiyel 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 (`deadlock.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 gösterimi 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?



Şekil 32.7: **Kilitlenme Bağımlılık Grafiği**

CRUX: KİLİTLENMEYLE 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 bloğu sayfalamak 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, kapsüllemenin doğasından kaynaklanmaktadır. yazılım olarak geliştiriciler, bize uygulamaların ayrıntılarını gizlememiz ve böylece yazılımı modüler bir şekilde oluşturmayı kolaylaştırmamız öğretildi. Ne yazık ki, bu tür bir modülerlik, kilitleme ile pek örtüşmez. Jula ve ark. [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 neredeyse aynı anda v2.AddAll(v1) öğesini çağırırsa, tümünü çağıran uygulamadan oldukça gizli bir şekilde kilitlenme potansiyeline sahibiz.

Kilitlenme Koşulları

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

- **Karşılıklı dışlama (Mutual exclusion):** Konular, ihtiyaç duydukları kaynakların özel kontrolünü talep eder (örneğin, bir iş parçacığı bir kilidi alır).
- **Tut ve bekle (Hold-and-wait):** Konular, ek kaynakları (örneğin, elde etmek istedikleri kilitler) beklerken kendilerine tahsis edilen kaynakları (ör. halihazırda edinmiş oldukları kilitler) tutar.
- **Ön alım yok (No preemption):** Kaynaklar (ör. kilitler) onları tutan iş parçacıklarından zorla kaldırılamaz.
- **Dairesel Bekleme (Circular wait):** 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.

Korunma

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 alımında **tam bir sıralama (total ordering)** 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, **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 kodun üst kısmındaki yorum, basit olanlar da dahil olmak üzere on farklı kilit edinme emri grubunu ortaya koymaktadır.

İpucu: KİLİT ADRESİNE GÖRE KİLİT SİPARİŞİNİ UYGULA

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: bir şey yap(mutex t *m1, mutex t *m2). Kod her zaman m2'den önce m1'i alırsa (veya her zaman m1'den önce m2'yi alırsa), kilitlenebilir, çünkü bir iş parçacığı bir şey yap(L1, L2) diyebilirken, başka bir iş parçacığı bir şey yap(L2, L1) diyebilir.

Bu özel sorundan kaçınmak için akıllı programcı, her bir kilidin adresini, kilit edinimi sipariş etmenin bir yolu olarak kullanabilir. Kilitleri yüksekten düşüğe veya düşükten yükseğe adres sırasına göre alarak bir şey(), hangi sırada geçirilirlerse geçsinler kilitleri her zaman aynı sırada almasını garanti edebilir. Bu:

```
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 mmap rwsem'den önce i mutex" ve "i sayfalar kilitlenmeden önce takas kilidinden önce özel kilitten önce i mmap rwsem" gibi daha karmaşık emirler. 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 gelenektir 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.

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" , "Deadlock" kısaltılmasıdır.

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 duyulduğu zaman yerine erken (bir kerede) alınması gerektiğinden, bu tekniğin eşzamanlılığı azaltması da olasıdır.

Önlem 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: **canlı kilit (live lock)**. 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 çalıştırır (ve dolayısıyla bu bir kilitlenme değildir), ancak ilerleme kaydedilmez, bu nedenle canlı kilit adı verilir. Canlı kilit sorununa da çözümler 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: deneme kilidi yaklaşımı kullanmanın zor kısımlarının etrafından dolanır. 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 almışsa (L1 dışında)

yol boyunca onları da dikkatlice 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 engelleme sağlamadığını (kilidi ona sahip olan bir iş parçacığından zorla alma eylemi) eklemediğini, bunun yerine bir geliştiricinin kilit sahipliğinden geri adım atmasına izin vermek için try lock yaklaşımını kullandığını da fark edebilirsiniz (yani, kendi sahipliklerini önleyin) 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 (lock-free)** (ve ilgili **beklemesiz (wait-free)**) 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 edin ve bırak 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 kullanıyoruz2. Bunun yerine, sadece karşılaştır ve değiştir talimatını kullanarak bu eklemeyi kilitsiz bir şekilde gerçekleştirmeye çalışalım. İşte 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]Zeki okuyucu, insert();'e girerken neden kilidi 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 **kaçınma(avoidance)** 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 zamanlar.

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

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

Böylece akıllı bir planlayıcı, T1 ve T2 aynı anda çalıştırılmadığı sürece hiçbir kilitlenmenin olmayacağını hesaplayabilir. İşte böyle bir zamanlama:

(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.:

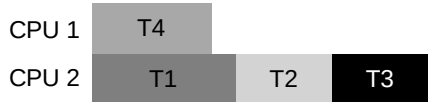
	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

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. Üzücü bir olay nadiren meydana geliyorsa, özellikle kötü bir 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 olan, hangisini ne zaman yapacağını bilmektir, biraz içgörü ancak deneyim ve eldeki göreve kendini adama yoluyla kazanılır.

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



Gördüğünüz gibi, statik zamanlama, T1, T2 ve T3'ün hepsinin aynı işlemci üzerinde çalıştırıldığı ve bu nedenle işleri tamamlamak için gereken toplam sürenin önemli ölçüde arttığı muhafazakar 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 birçok benzer 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 kullanışlıdırlar. Ayrıca, yukarıdaki ikinci örnekte gördüğümüz gibi, bu tür yaklaşımlar eşzamanlılığı sınırlayabilir. Bu nedenle, zamanlama yoluyla çıkmazdan kaçınmak, yaygın olarak kullanılan genel amaçlı bir çözüm değildir.

Algılama ve Kurtarma

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 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 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 eksikliği ve yeni bir beklemesiz veri yapısı geliştirmenin 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 olmaksızın belirli paralel hesaplama türlerini tanımlayabilirler. Kilitler doğası gereği sorunludur; belki de gerçekten mecbur kalmadıkça onları kullanmaktan kaçınmalıyız.

References

[B+87] "Concurrency Control and Recovery in Database Systems" by Philip A. Bernstein, Vas- sos 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. Interna- tional 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 Her- lihy. 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 ver-

sion). *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 Sur- veys, 19:4, December 1987. *An excellent overview of deadlock detection in distributed database sys- tems. 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 Wal- fish (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…*

Ödev (Kod)

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

Sorular

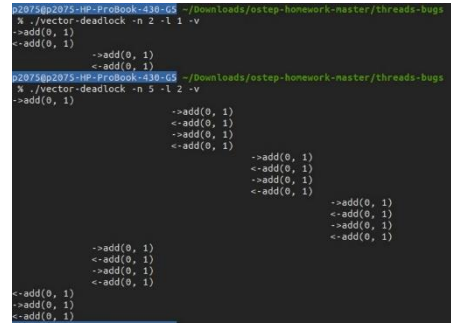
1. Öncelikle programların genel olarak nasıl çalıştığını ve bazı temel seçenekleri anladığınızdan emin olalım. main-common.c ve ilgili dosyaların yanı sıra vector-deadlock.c'deki kodu inceleyin. Şimdi, her biri bir vektör toplaması (-l 1) yapan iki iş parçacığını (-n 2) başlatan ve bunu ayrıntılı modda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

./vector-deadlock -n 2 -l 1 -v çalıştırıldığında, her biri bir çift vektör üzerinde bir vektör toplama işlemi gerçekleştirecek olan iki iş parçacığı başlatılır. -v bayrağı, vektör ekleme işleminin ilerleyişi hakkında ayrıntılı bilgileri yazdıracak olan ayrıntılı modu etkinleştirir.

Bu komutun çıktısı, iş parçacıklarının işletim sistemi tarafından programlanma sırasına bağlı olarak büyük olasılıkla çalıştırmadan çalıştırmaya farklılık gösterecektir. Bununla birlikte, çıktı genellikle vektörler üzerindeki kilitleri alırken, vektör ekleme işlemini gerçekleştirirken ve kilitleri serbest bırakırken her iş parçacığının ilerlemesini gösterecektir.

Örneğin, çıktı şöyle görünebilir:

Thread 0: starting vector add
Thread 1: starting vector add
Thread 0: acquired lock on vector 0
Thread 0: acquired lock on vector 1
Thread 0: performed vector add
Thread 0: released locks
Thread 1: acquired lock on vector 0
Thread 1: acquired lock on vector 1
Thread 1: performed vector add
Thread 1: released locks



Bu örnekte, iki iş parçacığı vektörler üzerindeki kilitleri alabilir ve herhangi bir kilitlenme olmadan aynı anda vektör ekleme işlemini gerçekleştirebilir. Bununla birlikte, iş parçacıklarının kilitleri aldığı ve işlemi gerçekleştirdiği tam sıra çalıştırmadan çalıştırmaya değişebilir.

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

-d bayrağı, programın bir kilitlenme durumu tespit etmesi durumunda bir mesaj yazdırmasına neden olacak olan kilitlenme tespitini etkinleştirir. Bu bayrak kullanıldığında, -l için daha yüksek bir değerle ./vector-deadlock -n 2 -l <n> -d komutunun çalıştırılması, işletim tarafından iş parçacıklarının programlandığı sıraya bağlı olarak kodun potansiyel olarak kilitlenmesine neden olur. sistem.

Örneğin, kod ./vector-deadlock -n 2 -l 2 -d ile çalıştırılırsa, aşağıdaki çıktıyı üretebilir:

Thread 0: starting vector add
Thread 1: starting vector add
Thread 0: acquired lock on vector 0
Thread 0: acquired lock on vector 1
Thread 1: acquired lock on vector 1
Thread 1: acquired lock on vector 0
DEADLOCK!

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-deadlock -n 4 -l 2 -v -d
->add(0, 1)
<-add(0, 1)
->add(0, 1)
<-add(0, 1)
                                        ->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(1, 0)
             <-add(1, 0)
             ->add(1, 0)
             <-add(1, 0)
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-deadlock -n 1 -l 2 -v -d
->add(0, 1)
<-add(0, 1)
->add(0, 1)
<-add(0, 1)
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-deadlock -n 2 -l 2 -v -d
->add(0, 1)
<-add(0, 1)
->add(0, 1)
<-add(0, 1)
             ->add(1, 0)
             <-add(1, 0)
             ->add(1, 0)
             <-add(1, 0)
```

Bu durumda, iki iş parçacığı vektörlerdeki kilitleri farklı sıralarda aldığından, kod kilitlendi, bu da her iş parçacığının diğer iş parçacığının kilidini açmasını beklediği bir duruma yol açıyor. Bu, kilitlemeyle ilgili yaygın bir sorundur ve kilitlenme durumu olarak bilinir.

-d ve -l için daha yüksek bir değer içeren kodun tüm çalıştırmaları, iş parçacıklarının işletim sistemi tarafından programlandığı belirli sıraya bağlı olduğundan, kilitlenmeye neden olmaz. Bununla birlikte, -l değerinin arttırılması, iş parçacıklarının farklı sıralardaki vektörler üzerindeki kilitleri elde etmek için daha fazla fırsatı olacağından, kilitlenme olasılığını artıracaktır.

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

İş parçacığı sayısının (-n) değiştirilmesi, vektörlerdeki kilitler için yarışan iş parçacığı sayısını etkileyeceğinden programın sonucunu değiştirecektir. İş parçacığı sayısını artırmak, kilitlenme olasılığını artıracaktır, çünkü vektörlerdeki kilitleri farklı sıralarda alabilecek daha fazla iş parçacığı olacaktır.

Kilitlenme oluşumu, iş parçacıklarının işletim sistemi tarafından programlandığı belirli sıraya bağlı olduğundan, kilitlenme olmamasını sağlayabilecek hiçbir -n değeri yoktur. Bununla birlikte, iş parçacığı sayısını artırmak kilitlenme olasılığını artıracaktır, bu nedenle -n için daha küçük bir değer kullanmak kilitlenme olasılığını azaltabilir.

Örneğin, ./vector-deadlock -n 1 -l 2 -d öğesinin tek iş parçacığı ile çalıştırılması kilitlenmeyle sonuçlanmaz çünkü vektörlerdeki kilitleri alabilecek yalnızca bir iş parçacığı vardır. Ancak aynı komutu -n için daha yüksek bir değerle çalıştırmak, vektörlerdeki kilitler için yarışan daha fazla iş parçacığı olacağından kilitlenmeye neden olabilir. Genel olarak, -n için daha küçük bir değer kullanmak, kilitlenme olasılığını azaltabilir, ancak tamamen ortadan kaldıramaz.

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-deadlock -n 5 -l 2 -v
->add(0, 1)
<-add(0, 1)
->add(0, 1)
<-add(0, 1)
              ->add(0, 1)
                                                      ->add(0, 1)
                                                      <-add(0, 1)
                                                      ->add(0, 1)
                                                      <-add(0, 1)
              <-add(0, 1)
              ->add(0, 1)
              <-add(0, 1)
                                        ->add(0, 1)
                                        <-add(0, 1)
                                        ->add(0, 1)
                                        <-add(0, 1)
                            ->add(0, 1)
                            <-add(0, 1)
                            ->add(0, 1)
                            <-add(0, 1)
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-deadlock -n 3 -l 2 -v
->add(0, 1)
<-add(0, 1)
->add(0, 1)
<-add(0, 1)
              ->add(0, 1)
              <-add(0, 1)
              ->add(0, 1)
              <-add(0, 1)
                            ->add(0, 1)
                            <-add(0, 1)
                            ->add(0, 1)
                            <-add(0, 1)
```

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

Kod, adres sırasına göre vektörler üzerinde kilitler alarak kilitlenmeyi önler. Bu, iki iş parçacığının birbirine iki vektör eklemeye çalışması durumunda kilitleri kilitlenmeden aynı sırayla almalarını sağlar.

Vector_add yordamında kaynak ve hedef vektörlerin aynı olduğu özel bir durum vardır, çünkü bu durumda fonksiyonun vektör üzerinde iki kez yerine yalnızca bir kez kilit alması gerekir. Vektör üzerinde iki kez kilit elde etmek, iş parçacığının kilitlenmesine neden olur. Kaynak ve hedef vektörlerin aynı olduğu özel bir duruma sahip olarak, fonksiyon bu potansiyel kilitlenmeyi önleyebilir.

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?

Kod -t -n 2 -l 100000 -d bayraklarıyla çalıştırıldığında, tamamlanması biraz zaman alacaktır. Kodu tamamlamak için gereken toplam süre, içinde çalıştığı belirli donanım ve yazılım ortamına bağlı olacaktır, ancak muhtemelen saniyeler veya dakikalar mertebesinde olacaktır.

Döngü sayısını (-l) artırmak, iş parçacıklarının gerçekleştirmesi gereken iş miktarını artıracak ve bu da kodu tamamlamak için gereken toplam süreyi artıracaktır. Benzer şekilde, iş parçacığı sayısını (-n) artırmak, vektörlerdeki kilitler için yarışan iş parçacığı sayısını artıracak ve bu da kodu tamamlamak için gereken toplam süreyi artıracaktır. Genel olarak, bu faktörlerin her ikisinin de kodu tamamlamak için gereken toplam süre üzerinde önemli bir etkisi olacaktır.

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-global-order -t -n 2 -l 100000 -d
Time: 0.06 seconds
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-global-order -t -n 5 -l 100000 -d
Time: 0.47 seconds
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-global-order -t -n 5 -l 1000000 -d
Time: 4.75 seconds
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-global-order -t -n 15 -l 1000000 -d
Time: 17.38 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?

Paralellik bayrağı (-p) açıldığında, her iş parçacığı vektörleri diğer iş parçacıklarıyla paylaşmak yerine kendi vektör çiftinde çalışacaktır. Bu, vektörlerdeki kilitler için rekabet etmek zorunda kalmadan iş parçacıklarının eşzamanlı olarak çalışmasına izin verdiği için performansı artırabilir.

Genel olarak, -p bayrağının açılması programın performansını artırmalıdır çünkü bu, iş parçacıklarının farklı vektör çiftleri üzerinde aynı anda çalışmasına izin verir. Kesin performans artışı, iş parçacığı sayısına ve vektör ekleme işleminin iş yüküne bağlı olacaktır, ancak genel olarak, -p bayrağının kullanılması, kullanılmamasına kıyasla programın performansını artırmalıdır.

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-global-order -t -n 15 -l 1000000 -d -p
Time: 0.57 seconds
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-global-order -t -n 5 -l 1000000 -d -p
Time: 0.47 seconds
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-global-order -t -n 5 -l 100000 -d -p
Time: 0.07 seconds
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/threads-bugs
 % ./vector-global-order -t -n 2 -l 100000 -d -p
Time: 0.02 seconds
```

7. Şimdi vector-try-wait.c'yi inceleyelim. Öncelikle kodu anladığınızdan emin olun. pthread mutex trylock()'a yapılan ilk çağrı 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 pthread_mutex_trylock() çağrısına gerek yok gibi görünüyor, çünkü kod başarısız olursa bir sonraki adımda kilidi hemen tekrar almaya çalışır. Kod, kilitlenmeyi önlemek amacıyla iki vektör üzerindeki kilitleri belirli bir sırayla elde etmek için bir yeniden deneme mekanizması uyguluyor gibi görünüyor.

Kodu çalıştırmadan, global sipariş yaklaşımına kıyasla ne kadar hızlı çalışacağını söylemek zor. Bununla birlikte, vektörler üzerindeki kilitler için daha fazla rekabet olacağından, iş parçacığı sayısı arttıkça yeniden deneme sayısının artması muhtemeldir. Bu potansiyel olarak, yeniden deneme mekanizmasının önlemesi amaçlanan daha fazla kilitlenme örneğine yol açabilir.

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, iki vektör üzerindeki kilitleri belirli bir sırada elde etmek için küresel bir kilit kullanmasıdır; bu, vektörlere erişimi etkili bir şekilde seri hale getirir. Bu, vektörler üzerinde aynı anda yalnızca bir iş parçacığı çalışabileceğinden, birden çok iş parçacığı kullanma amacını ortadan kaldırır.

Bu yaklaşımın performansının, -p bayrağı olan ve olmayan diğer sürümlerden daha kötü olması muhtemeldir. -p olmadan çalışırken, kod yalnızca bir iş parçacığı kullanacaktır, dolayısıyla birden çok iş parçacığı kullanmanın performans avantajı olmayacaktır. -p ile çalışırken, global kilit birden fazla iş parçacığının vektörler üzerinde aynı anda çalışmasını engelleyecek, böylece kod paralellikten yararlanamayacaktır. Her iki durumda da, küresel kilidin kullanılması büyük olasılıkla performansı düşürecek önemli bir ek yük getirecektir.

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

Bu sürüm, diğer sürümlerle aynı semantiği sağlamaz çünkü karşılıklı dışlamayı sağlamak ve vektörlere eşzamanlı erişimi engellemek için kilitler kullanmaz. Bu, birden çok iş parçacığının aynı anda vektörler üzerinde çalışabileceği ve bunun da yarış koşullarına ve yanlış sonuçlara yol açabileceği anlamına gelir.

fetch_and_add() işlevi, v_src'deki değeri v_dst'deki karşılık gelen öğeye eklemek için atomik bir komut kullanır. Ancak bu, işlemin yalnızca vektörlerin tek bir elemanı için atomik olmasını sağlar. Yarış koşullarına ve yanlış sonuçlara yol açabilecek diğer iş parçacığı tarafından vektörlerin diğer öğelerine eşzamanlı erişimi engellemez.

Buna karşılık, diğer sürümler, karşılıklı dışlamayı sağlamak ve vektörlere eşzamanlı erişimi önlemek için kilitler kullanır. Bu, vektörler üzerinde aynı anda yalnızca bir iş parçacığının çalışabilmesini sağlar, bu da yarış koşullarını önler ve sonuçların doğru olmasını sağlar.

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?

Kodu çalıştırmadan, bu kilitsiz sürümün diğer sürümlere kıyasla tam olarak nasıl performans göstereceğini söylemek zor. Ancak, her iki durumda da performansının daha kötü olması muhtemeldir.

-p bayrağı olmadan çalışırken, tüm iş parçacıkları aynı iki vektör üzerinde çalışıyor olacaktır. Kod, karşılıklı dışlamayı sağlamak ve vektörlere eşzamanlı erişimi engellemek için kilit kullanmadığından, bu muhtemelen yarış koşullarına ve yanlış sonuçlara neden olacaktır. Bu, potansiyel olarak önemli performans düşüşüne yol açabilir.

-p bayrağıyla çalışırken, her iş parçacığı ayrı vektörler üzerinde çalışacaktır. Bu durumda, kod bir dereceye kadar paralellikten faydalanabilir, ancak karşılıklı dışlamayı sağlamak ve vektörlere eşzamanlı erişimi önlemek için kilitler kullanan diğer sürümlerden daha az performans göstermesi muhtemeldir. Bunun nedeni, fetch_and_add() işlevinin vektörlerin yalnızca tek bir öğesine atomik erişim sağlaması, diğer sürümlerin ise vektörlerde aynı anda yalnızca bir iş parçacığının çalışabilmesini sağlamak için kilitler kullanmasıdır. Bu, potansiyel olarak yarış koşullarına ve performansı düşürebilecek yanlış sonuçlara yol açabilir.