32

Yaygın Eşzamanlılık Sorunları

Araştırmacılar uzun yıllar boyunca eşzamanlılık hatalarını incelemek için büyük zaman ve çaba harcadılar. İlk çalışmaların çoğu, geçmiş bölümlerde değindiğimiz ancak şimdi derinlemesine inceleyeceğimiz bir konu olan **deadlock(kilitlenme)** üzerine odaklanmıştır [C+71]. Daha yeni çalışmalar, diğer yaygın eşzamanlılık hatalarını incelemeye odaklanmaktadır (i.e., non-deadlock bugs). 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 temel sorunumuz:

püf noktası: YAYGIN EŞZAMANLILIK HATALARININ NASIL ELE ALINACAĞI

Eşzamanlılık hataları çeşitli ortak modellerde ortaya çıkma eğilimindedir. Hangilerine dikkat edilmesi 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 soruya genel olarak cevap vermek zordur, ancak neyse ki başkaları bizim için bu işi yapmıştır. Özellikle, Lu ve diğerleri tarafından yapılan bir çalışmaya dayanıyoruz. [L+08], Uygulamada ne tür hataların ortaya çıktığını anlamak için bir dizi popüler eşzamanlı uygulamayı ayrıntılı olarak analiz ediyor.

Çalışma dört büyük ve önemli açık kaynak uygulamasına odaklanmaktadır: MySQL (popüler bir veritabanı yönetim sistemi), Apache (iyi bilinen bir web sunucusu), Mozilla (ünlü web tarayıcısı) ve OpenOffice (bazı insanların gerçekten kullandığı MS Office paketinin ücretsiz bir versiyonu). 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 üzerinde çalıştığı hataların bir özetini göstermektedir. Şekilden, toplam 105 hata olduğunu görebilirsiniz, bunların çoğu

Uygulama	Ne işe yarar	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: **Modern Uygulamalardaki Hatalar**

kilitlenme değildi (74); geri kalan 31 tanesi kilitlenme hatasıydı. Ayrıca, her uygulamadan incelenen hataların sayısını görebilirsiniz; OpenOffice'te sadece 8 toplam eşzamanlılık hatası varken, Mozilla'da yaklaşık 60 hata vardı.

Şimdi bu farklı hata sınıflarını (non-deadlock, dead- lock) biraz daha derinlemesine inceleyeceğiz. Kilitlenmeyen hataların ilk sınıfı için, tartışmamızı yönlendirmek üzere çalışmadan örnekler kullanıyoruz. İkinci sınıf kilitlenme hataları için, kilitlenmenin önlenmesi, kaçınılması veya ele alınması konusunda yapılan uzun çalışmaları tartışıyoruz.

32.2 Non-Deadlock Hatalar

Lu'nun çalışmasına göre, deadlock olmayan hatalar eşzamanlılık hatalarının çoğunluğunu oluşturuyor. Peki bunlar ne tür hatalardır? Nasıl ortaya çıkarlar? Onları nasıl düzeltebiliriz? Şimdi Lu ve arkadaşları tarafından bulunan iki ana kilitlenme dışı hata türünü tartışıyoruz.: **atomicity violation** hataları ve **order violation** hataları.

Atomicity-Violation (atomiklik ihlali) Hataları

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

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

Figure 32.2: **Atomicity Violation (`atomicity.c`)**

Örnekte, iki farklı thread thd yapısındaki proc info_ alanına erişmektedir. İlk thread değerin NULL olup olmadığını kontrol eder ve ardından değerini yazdırır; ikinci thread değeri NULL olarak ayarlar. Açıkça görüldüğü gibi, ilk thread kontrolü gerçekleştirir ancak daha sonra fputs çağrısından önce kesilirse, ikinci thread arada çalışabilir ve böylece işaretçiyi NULL olarak ayarlayabilir; ilk thread devam ettiğinde, NULL işaretçi fputs tarafından dereferenced edileceğinden çökecektir.

Lu ve arkadaşlarına göre atomiklik ihlalinin daha resmi tanımı şudur: "Birden fazla bellek erişimi arasında istenen serileştirilebilirlik ihlal edilir (yani bir kod bölgesinin atomik olması amaçlanır, ancak yürütme sırasında atomiklik uygulanmaz)." Yukarıdaki örneğimizde, kodun proc info'nun NULL olup olmadığının kontrolü ve proc info'nun fputs() çağrısında kullanımı hakkında (Lu'nun ifadesiyle) bir atomiklik varsayımı vardır; varsayım yanlış olduğunda, kod istendiği gibi çalışmayacaktır.

Bu tür bir sorun için bir çözüm bulmak genellikle (ancak her zaman değil) basittir. Yukarıdaki kodu nasıl düzeltebileceğinizi düşünebiliyor musunuz?

Bu çözümde (Şekil 32.3), paylaşılan değişken referanslarının etrafına kilitler ekleyerek, herhangi bir thread proc info alanına eriştiğinde bir kilide sahip olmasını sağlıyoruz (proc info kilidi). Elbette, yapıya erişen diğer tüm kodlar da bunu yapmadan önce bu kilidi edinmelidir.

```
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);
```

Figure 32.3: **Atomicity Violation Düzeltildi (`atomicity fixed.c`)**

Order-Violation(Düzen İhlal) Hataları

Lu ve arkadaşları tarafından bulunan bir başka yaygın kilitlenme dışı hata türü. order **violation** olarak bilinir. İşte bir başka basit örnek; bir kez daha, aşağıdaki kodda neden bir hata olduğunu bulup bulamayacağınızı görün.

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

Figure 32.4: **Ordering Hata (`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 }
```

Figure 32.5: **Ordering Violation Düzeltme (`ordering fixed.c`)**

Bununla birlikte, Thread 2 oluşturulduktan sonra hemen çalışırsa, Thread 2'de mMain() içinde erişildiğinde mThread değeri ayarlanmayacak ve muhtemelen bir NULL-pointer dereference ile çökecektir. mThread değerinin başlangıçta NULL olduğunu varsaydığımızı unutmayın; aksi takdirde, Thread 2'deki dereferans yoluyla rastgele bellek konumlarına erişildiğinde daha da garip şeyler olabilir.

Düzen ihlalinin daha resmi tanımı aşağıdaki gibidir: "İki (grup) bellek erişimi arasında istenen sıra tersine çevrilir (yani, A her zaman B'den önce yürütülmelidir, ancak yürütme sırasında sıra uygulanmaz)" [L+08].

Bu tür bir hatanın düzeltilmesi genellikle sıralamayı zorlamaktır. Daha önce tartışıldığı gibi, **condition variables (koşul değişkenleri)** kullanmak bu tarz bir 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 koşul değişkeni (mtCond) ve buna karşılık gelen kilit (mtLock) ile bir durum değişkeni ekledik

(`mtInit`). Başlatma kodu çalıştığında, mtInit'in durumunu 1'e ayarlar ve bunu yaptığını bildirir. Thread 2 bu noktadan önce çalışmışsa, bu sinyali ve ilgili durum değişikliğini bekleyecektir; daha sonra çalışırsa, durumu kontrol edecek ve başlatmanın zaten gerçekleştiğini görecektir (yani, mtInit 1'e ayarlanmıştır) ve böylece uygun şekilde devam edecektir. mThread'i durum değişkeninin kendisi olarak kullanabileceğimizi, ancak burada basitlik adına bunu yapmadığımızı unutmayın. Thread'ler arasında sıralama söz konusu olduğunda, koşul değişkenleri (veya semaforlar) imdada yetişebilir.

Non-Deadlock Hatalar: Özet

Lu ve arkadaşları tarafından incelenen deadlock olmayan hataların büyük bir kısmı (%97) ya atomiklik ya da düzen ihlalidir. Dolayısıyla, programcılar bu tür hata kalıpları hakkında dikkatlice düşünerek muhtemelen bunlardan kaçınmak için daha iyi bir iş çıkarabilirler. Dahası, daha otomatik kod kontrol araçları geliştirildikçe, dağıtım sırasında bulunan kilitlenme dışı hataların büyük bir kısmını oluşturdukları için muhtemelen bu iki hata türüne odaklanmalıdırlar.

Ne yazık ki, tüm hatalar yukarıda incelediğimiz örnekler kadar kolay düzeltilemez. Bazılarının düzeltilmesi için programın ne yaptığının daha iyi anlaşılması ya da daha fazla miktarda kod veya veri yapısının yeniden düzenlenmesi gerekir. Daha fazla ayrıntı için Lu ve arkadaşlarının mükemmel (ve okunabilir) makalesini okuyun.

32.3 Deadlock Hatalar

Yukarıda bahsedilen eşzamanlılık hatalarının ötesinde, karmaşık kilitleme protokollerine sahip birçok eşzamanlı sistemde ortaya çıkan klasik bir sorun **deadlock(kilitlenme)** olarak bilinir. Örneğin, bir thread (Thread 1 diyelim) bir kilidi (L1) tutuyor ve başka bir kilidi (L2) bekliyorsa; ne yazık ki, L2 kilidini tutan thread (Thread 2) L1'in serbest bırakılmasını bekliyorsa kilitlenme meydana gelir. İşte 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);
```

Figure 32.6: **Basit Deadlock (`deadlock.c`)**

Bu kod çalışırsa, kilitlenmenin mutlaka meydana gelmeyeceğini unutmayın; bunun yerine, örneğin Thread 1 L1 kilidini alırsa ve ardından Thread 2'ye bir bağlam geçişi olursa, kilitlenme meydana gelebilir. Bu noktada, Thread 2 L2'yi alır ve L1'i almaya çalışır. Böylece, her bir thread diğerini beklediği ve ikisi de çalışamadığı için bir kilitlenme meydana gelir. Grafiksel gösterim için Şekil 32.7'ye bakın; grafikte bir döngünün varlığı kilitlenmenin göstergesidir.

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

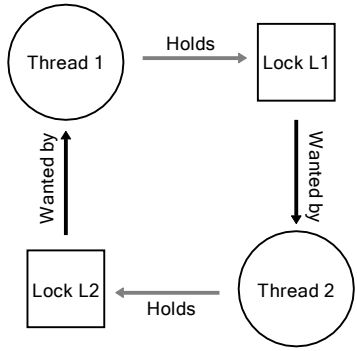


Figure 32.7: **Deadlock Bağımlılık Grafiği**

CRUX: HOW TO DEAL WITH DEADLOCK
How should we build systems to prevent, avoid, or at least detect and recover from deadlock? Is this a real problem in systems today?

Deadlock'lar(Kilitlenmeler) Neden Oluşur??

Düşündüğünüz gibi, yukarıdaki gibi basit kilitlenmeler kolaylıkla önlenebilir görünmektedir. Örneğin, Thread 1 ve 2'nin her ikisi de kilitleri aynı sırada kaptığından emin olsaydı, kilitlenme asla ortaya çıkmazdı. Peki neden kilitlenmeler meydana gelir?

Bunun bir nedeni, büyük kod tabanlarında bileşenler arasında karmaşık bağımlılıkların ortaya çıkmasıdır. Örneğin işletim sistemini ele alalım. Sanal bellek sisteminin diskten bir bloğu çağırmak için dosya sistemine erişmesi gerekebilir; dosya sistemi daha sonra bloğu okumak için bir bellek sayfasına ihtiyaç duyabilir ve böylece sanal bellek sistemiyle bağlantı kurabilir. Bu nedenle, büyük sistemlerde kilitleme stratejilerinin tasarımı, kodda doğal olarak oluşabilecek döngüsel bağımlılıklar durumunda kilitlenmeyi önlemek için dikkatli bir şekilde yapılmalıdır.

Bir başka neden de **encapsulation(kapsülleme)** doğasından kaynaklanmaktadır. Yazılım geliştiriciler olarak bize uygulamaların ayrıntılarını gizlememiz ve böylece yazılımı modüler bir şekilde oluşturmayı kolaylaştırmamız öğretilir. Ne yazık ki bu tür bir modülerlik kilitleme ile iyi bir uyum göstermez. Jula ve diğerleri gibi. [J+08], görünüşte zararsız olan bazı arayüzlerin neredeyse sizi kilitlenmeye davet ettiğini belirtmektedir. Örneğin, Java Vector sınıfını ve AddAll() metodunu ele alalım. Bu rutin aşağıdaki gibi çağrılacaktır:

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

Dahili olarak, yöntemin multi-thread güvenli olması gerektiğinden, hem eklenen vektör (v1) hem de parametre (v2) için kilitlerin edinilmesi gerekir. Program, v2'nin içeriğini v1'e eklemek için söz konusu kilitleri rastgele bir sırayla (örneğin v1 sonra v2) alır. Başka bir thread v2.AddAll(v1)'i neredeyse aynı anda çağırırsa, çağıran uygulamadan oldukça gizli bir şekilde kilitlenme potansiyeline sahip oluruz.

Deadlock(Kilitlenme) için Koşullar

Bir kilitlenmenin meydana gelmesi için dört koşulun gerçekleşmesi gerekir [C+71]:

- **Mutual exclusion(Karşılıklı dışlama):** Thread'ler ihtiyaç duydukları kaynakların özel kontrolünü talep ederler (örneğin, bir thread bir kilidi ele geçirir).
- **Hold-and-wait(Tut ve bekle):** Thread'ler kendilerine tahsis edilen kaynakları (örn.ek yeniden kaynakları (örneğin, edinmek istedikleri kilitler) beklerken) zaten edinmiş oldukları).
- **No preemption(Önlem yok):** Kaynaklar (örneğin kilitler) zorla kaldırılamaz onları tutan parçalardan.
- **Circular wait(Dairesel bekleme):** Her bir thread'in zincirdeki bir sonraki thread tarafından yeniden aranan bir veya daha fazla kaynağı (örn. kilitler) tuttuğu dairesel bir thread zinciri vardır.

Bu dört koşuldan herhangi biri karşılanmazsa kilitlenme meydana gelemez. Bu nedenle, ilk olarak kilitlenmeyi önleme tekniklerini inceleyeceğiz; bu stratejilerin her biri yukarıdaki koşullardan birinin ortaya çıkmasını önlemeye çalışır ve bu nedenle kilitlenme sorununu ele almak için bir yaklaşımdır.

Prevention (Önleme)

Circular Wait (Dairesel Bekleme)

Muhtemelen en pratik önleme tekniği (ve kesinlikle sıklıkla kullanılan) kilitleme kodunuzu asla döngüsel bir beklemeye neden olmayacak şekilde yazmaktır. Bunu yapmanın en basit yolu, kilit alımında **total ordering (toplam sıralama)** sağlamaktır. Örneğin, sistemde yalnızca iki kilit varsa (L1 ve L2), L1'i her zaman L2'den önce alarak kilitlenmeyi önleyebilirsiniz. Böylesine katı bir sıralama, döngüsel beklemenin ortaya çıkmamasını sağlar; dolayısıyla kilitlenme olmaz.

Elbette, daha karmaşık sistemlerde, ikiden fazla kilit mevcut olacaktır ve bu nedenle toplam kilit sıralamasını elde etmek zor olabilir (ve belki de gereksizdir). Bu nedenle, **partial ordering (kısmi sıralama)** kilitlenmeyi önlemek amacıyla kilit edinimini yapılandırmak için kullanışlı bir yol olabilir. Kısmi kilit sıralamasının mükemmel bir gerçek örneği Linux [T+94] (v5.2) bellek eşleme kodunda görülebilir; kaynak kodun üst kısmındaki yorum, basit kilitleme de dahil olmak üzere on farklı kilitleme sırası grubunu ortaya koymaktadır.

TIP: KILIT SIRALAMASINI KILIT ADRESINE GÖRE ZORLAR

Bazı durumlarda, bir fonksiyonun iki (veya daha fazla) kilit alması gerekir; bu nedenle, dikkatli olmamız gerektiğini biliyoruz, aksi takdirde kilitlenme ortaya çıkabilir. Aşağıdaki şekilde çağrılan bir fonksiyon düşünün: do something(mutex t *m1, mutex t *m2). Kod her zaman m1'i m2'den önce alırsa (veya her zaman m2'yi m1'den önce alırsa), kilitlenme olabilir, çünkü bir thread do something(L1, L2) çağrısı yaparken başka bir thread do something (L2, L1) çağrısı yapabilir. Bu özel sorundan kaçınmak için, akıllı programcı her kilidin adresini kilit edinimini sıralamanın bir yolu olarak kullanabilir. Do something(), kilitleri yüksek-düşük veya düşük-yüksek adres sırasına göre alarak, hangi sırayla iletildiklerine bakılmaksızın kilitleri her zaman aynı sırayla alacağını garanti edebilir. Kod aşağıdaki gibi görünecektir:

```
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)
```

Bir programcı bu basit tekniği kullanarak çoklu kilit ediniminin basit ve verimli bir şekilde kilitlenmeden uygulanmasını sağlayabilir.

"i mutex before i mmap rwsem" gibi olanlar ve "i mmap rwsem before private lock before swap lock before i pages lock" gibi daha karmaşık sıralamalar.

Tahmin edebileceğiniz gibi, hem toplam hem de kısmi sıralama, kilitleme stratejilerinin dikkatli bir şekilde tasarlanmasını gerektirir ve büyük bir özenle inşa edilmelidir. Ayrıca, sıralama sadece bir kuraldır ve özensiz bir programcı kilitleme protokolünü kolayca göz ardı edebilir ve potansiyel olarak kilitlenmeye neden olabilir. Son olarak, kilit sıralaması kod tabanının derinlemesine anlaşılmasını ve değişkenlerin nasıl

çok sayıda rutin çağrılır; tek bir hata "D" kelimesiyle sonuçlanabilir.

Hold-and-wait (Tut ve bekle)

Kilitlenme için bekletme ve bekleme gereksinimi, tüm kilitlerin bir kerede atomik olarak alınmasıyla önlenebilir. Pratikte bu şu şekilde gerçekleştirilebilir:

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

İpucu: "D" " Deadlock" anlamına gelmektedir.

Bu kod, ilk olarak kilit önlemeyi yakalayarak, kilit ediniminin ortasında zamansız thread geçişinin gerçekleşmemesini garanti eder ve böylece deadlock bir kez daha önlenebilir. Elbette, herhangi bir thread bir kilidi ele geçirdiğinde, önce global önleme kilidini elde etmesini gerektirir. Örneğin, başka bir thread L1 ve L2 kilitlerini farklı bir sırada almaya çalışsaydı, bunu yaparken önleme kilidini elinde tutacağı için sorun olmazdı.

Bu çözümün bir dizi 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 talep etmemizi gerektirir. Bu teknik aynı zamanda, tüm kilitlerin gerçekten ihtiyaç duyuldukları zaman yerine erken (bir kerede) alınması gerektiğinden eşzamanlılığı azaltabilir.

No Preemption (Önlem Yok)

Kilitleri genellikle unlock çağrısı yapılana kadar tutulan kilitler olarak gördüğümüz için, çoklu kilit edinimi genellikle başımızı belaya sokar çünkü bir kilidi beklerken başka bir kilidi tutuyoruz. Birçok thread kütüphanesi, bu durumdan kaçınmaya yardımcı olmak için daha esnek bir dizi arayüz sağlar. Özellikle, pthread mutex trylock() rutini ya kilidi alır (mevcutsa) ve başarı döndürür ya da kilidin tutulduğunu belirten bir hata kodu döndürür; ikinci durumda, bu kilidi almak istiyorsanız daha sonra tekrar deneyebilirsiniz.

Böyle bir arayüz, kilitlenmesiz, sıralama açısından sağlam bir kilit edinme protokolü oluşturmak için aşağıdaki şekilde 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 thread'in aynı protokolü takip edebileceğini ancak kilitleri diğer sırada (L2 sonra L1) alabileceğini ve programın yine de kilitlerden arınmış olacağını unutmayın. Ancak yeni bir sorun ortaya çıkıyor: **livelock(canlı kilit)**. İki thread'in de bu diziyi tekrar tekrar denemesi ve her iki kilidi de elde etmekte başarısız olması mümkündür (belki de olası değildir). Bu durumda, her iki sistem de bu kod dizisini tekrar tekrar çalıştırmaktadır (ve bu nedenle bir kilitlenme değildir), ancak ilerleme kaydedilmemektedir, bu nedenle livelock adı verilir. Livelock sorununa da çözümler vardır: örneğin, geri dönüp her şeyi tekrar denemeden önce rastgele bir gecikme eklenebilir, böylece rakip thread'ler arasında tekrarlanan müdahale olasılığı azaltılabilir.

Bu çözümle ilgili bir nokta: kullanımın zor kısımlarının etrafından dolaşıyor bir trylock yaklaşımı. Muhtemelen yine var olacak ilk sorun kapsülleme nedeniyle ortaya çıkar: bu kilitlerden biri çağrılan bir rutinin içine gömülürse, başlangıca geri atlamanın uygulanması daha karmaşık hale gelir. Kod bazı kaynaklar edinmiş olsaydı (L1 dışında)

Yol boyunca, bunları da dikkatli bir şekilde serbest bıraktığından emin olmalıdır; örneğin, L1'i aldıktan sonra kod bir miktar bellek ayırmışsa, tüm diziyi tekrar denemek için başa dönmeden önce L2'yi almadaki başarısızlık üzerine bu belleği serbest bırakması gerekecektir. Bununla birlikte, sınırlı durumlarda (örneğin, daha önce bahsedilen Java vektör yöntemi), bu tür bir yaklaşım iyi çalışabilir.
Ayrıca bu yaklaşımın gerçekten önlem almadığını da fark edebilirsiniz (bir kilidi ona sahip olan bir thread'den zorla alma eylemi), bunun yerine bir geliştiricinin kilit sahipliğinden geri çekilmesine (yani, sahipliklerini önleme) zarif bir şekilde izin vermek için trylock yaklaşımını kullanır. Ancak, bu pratik bir yaklaşımdır ve bu nedenle, bu konudaki kusurlarına rağmen buraya dahil ediyoruz.

Mutual Exclusion (Karşılıklı Dışlama)

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

Herlihy, çeşitli veri yapılarının kilitler olmadan da tasarlanabileceği fikrini ortaya atmıştır [H91, H93]. Buradaki bu **lock-free(Kilitsiz)** (ve ilgili **wait-free(beklemesiz)**) yaklaşımların arkasındaki fikir basittir: güçlü donanım talimatlarını kullanarak, veri yapılarını açık kilitleme gerektirmeyecek şekilde oluşturabilirsiniz.

Basit bir örnek olarak, hatırlayabileceğiniz gibi donanım tarafından sağlanan ve aşağıdakileri yapan atomik bir komut olan bir karşılaştırma ve takas yapısına sahip olduğumuzu 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, compare-and-swap(karşılaştır ve değiştir) kullanarak bir değeri atomik olarak belirli bir miktarda artırmak istediğimizi düşünün. Bunu aşağıdaki basit fonksiyonla 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 edinmek, güncellemeyi yapmak ve ardından serbest bırakmak yerine, değeri tekrar tekrar yeni miktara güncellemeye çalışan ve bunu yapmak için karşılaştır ve değiştir özelliğini kullanan bir yaklaşım geliştirdik. Bu şekilde,

kilit kazanılmaz ve kilitlenme ortaya çıkmaz (ancak livelock hala bir olasılıktır ve bu nedenle sağlam bir çözüm yukarıdaki basit kod parçasından daha karmaşık olacaktır).

Biraz daha karmaşık bir örneği ele alalım: liste ekleme. İşte bir listenin başına ekleme yapan 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 işlemi gerçekleştirir, ancak "aynı anda" birden fazla thread tarafından çağrılırsa bir yarış koşulu oluşur. Nedenini bulabilir misiniz? (her zaman olduğu gibi kötü niyetli bir zamanlama aralığı olduğunu varsayarak, iki eşzamanlı ekleme gerçekleşirse bir listeye ne olabileceğinin bir resmini çizin). Elbette, bu kodu bir kilit alma ve bırakma ile çevreleyerek bunu çözebiliriz:

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

Bu çözümde, kilitleri klasik şekilde kullanıyoruz. Bunun yerine, bu ekleme işlemini sadece compare-and-swap komutunu kullanarak 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 }
```

Dikkatli bir okuyucu kilidi neden insert() işlevine girerken değil de bu kadar geç yakaladığımızı sorabilir; siz, dikkatli okuyucu, bunun neden doğru olabileceğini bulabilir misiniz? Kod, örneğin malloc() çağrısı hakkında ne gibi varsayımlarda bulunuyor?

Buradaki kod, bir sonraki işaretçiyi mevcut head'i gösterecek şekilde günceller ve ardından yeni oluşturulan düğümü listenin yeni head'i olarak konumlandırmaya çalışır. Ancak, bu arada başka bir thread yeni bir head'i başarıyla takas etmişse bu başarısız olur ve bu thread'in yeni head ile tekrar denemesine neden olur.

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

Deadlock Avoidance via Scheduling(zamanlama Yoluyla Kilitlenmeden Kaçınma)

Kilitlenmeyi önlemek yerine, bazı senaryolarda kilitlenmeden **Avoidance** (kaçınmak) tercih edilir. Kaçınma, çeşitli thread'lerin yürütme sırasında hangi kilitleri ele geçirebileceğine dair bazı global bilgiler gerektirir ve daha sonra söz konusu thread'leri hiçbir kilitlenmenin meydana gelmeyeceğini garanti edecek şekilde planlar.

Örneğin, iki işlemcimiz ve bunlar üzerinde programlanması gereken dört thread'imiz olduğunu varsayalım. Ayrıca Thread 1'in (T1) L1 ve L2 kilitlerini aldığını (bazı sıralarda, yürütme sırasında bir noktada), T2'nin L1 ve L2'yi de aldığını, T3'ün sadece L2'yi aldığını ve T4'ün hiçbir kilidi almadığını bildiğimizi varsayalım. Thread'lerin bu kilit edinme taleplerini tablo şeklinde gösterebiliriz:

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

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

(T3 ve T1) veya (T3 ve T2)'nin çakışmasında bir sorun olmadığını unutmayın. T3, L2 kilidini ele geçirse bile, yalnızca bir kilidi ele geçirdiği için diğer thread'lerle birlikte çalışarak asla kilitlenmeye neden olamaz.

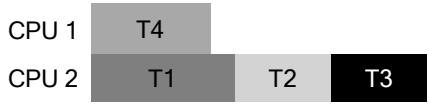
Bir örneğe daha bakalım. Bu örnekte, aşağıdaki çekişme tablosunda gösterildiği gibi, aynı kaynaklar (yine L1 ve L2 kilitleri) için daha fazla çekişme vardır:

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

öneri: HER ZAMAN MÜKEMMEL YAPMAYIN (TOM WEST YASASI)
Bilgisayar endüstrisinin klasik kitabı Soul of a New Machine'in [K81] yazarı Tom West'in meşhur bir sözü vardır: "Yapmaya değer her şey iyi yapmaya değer değildir", ki bu müthiş bir mühendislik özdeyişidir. Kötü bir şey nadiren meydana geliyorsa, özellikle de kötü şeyin meydana gelmesinin maliyeti düşükse, bunu önlemek için kesinlikle çok fazla çaba harcamamak gerekir. Öte yandan, bir uzay mekiği inşa ediyorsanız ve bir şeylerin ters gitmesinin bedeli uzay mekiğinin havaya uçmasıysa, belki de bu tavsiyeyi göz ardı etmelisiniz.

Bazı okuyucular itiraz ediyor: "Çözüm olarak vasatlığı öneriyormuşsunuz gibi geliyor!" Belki de haklılar, böyle bir tavsiyede bulunurken dikkatli olmalıyız. Bununla birlikte, deneyimlerimiz bize mühendislik dünyasında, son teslim tarihlerinin ve diğer gerçek dünya kaygılarının zorlamasıyla, bir sistemin hangi yönlerinin iyi inşa edileceğine ve hangilerinin başka bir güne bırakılacağına her zaman karar vermek zorunda kalacağımızı söylüyor. İşin zor kısmı, hangisinin ne zaman yapılacağını bilmektir; bu da ancak deneyim ve eldeki işe adanmışlıkla kazanılan bir içgörüdür.

Özellikle, T1, T2 ve T3 thread'lerinin hepsinin yürütme sırasında bir noktada L1 ve L2 kilitlerinin her ikisini de alması gerekir. İşte hiçbir kilitlenmenin meydana gelmeyeceğini 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 işlerin tamamlanması için gereken toplam süre önemli ölçüde uzar. Bu görevleri eş zamanlı olarak çalıştırmak mümkün olsa da, kilitlenme korkusu bunu yapmamızı engelliyor ve maliyeti performans oluyor.

Bu tür bir yaklaşımın ünlü bir örneği Dijkstra'nın Banker Algoritmasıdır [D64] 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 kilitlenmenin önlenmesi yaygın olarak kullanılan genel amaçlı bir çözüm değildir.

Detect and Recover(Algıla ve Kurtar)

Son bir genel strateji, zaman zaman kilitlenmelerin meydana gelmesine izin vermek ve ardından böyle bir kilitlenme tespit edildiğinde bazı önlemler almaktır. Örneğin,

Eğer bir işletim sistemi yılda bir kez donsaydı, onu yeniden başlatır ve mutlu (ya da huysuz) bir şekilde işinize devam ederdiniz. Eğer kilitlenmeler nadiren yaşanıyorsa, böyle bir çözümsüzlük gerçekten de oldukça faydalıdır.

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. Veri yapılarının daha karmaşık bir şekilde onarılması gerekiyorsa, süreci kolaylaştırmak için bir insan dahil edilebilir.

Veritabanı eşzamanlılığı, kilitlenme ve ilgili konular hakkında daha fazla ayrıntı başka bir yerde bulunabilir [B+87, K87]. Bu zengin ve ilginç konu hakkında daha fazla bilgi edinmek için bu çalışmaları okuyun ya da daha iyisi veritabanları üzerine bir kurs alın.

32.4 Summary (Özet)

Bu bölümde, mevcut programlarda ortaya çıkan hata türlerini inceledik. İlk tür, kilitlenmeyen hatalar şaşırtıcı derecede yaygındır, ancak genellikle düzeltilmesi daha kolaydır. Bunlar arasında, birlikte yürütülmesi gereken bir dizi talimatın birlikte yürütülmediği atomiklik ihlalleri ve iki thread arasında gerekli sıranın uygulanmadığı sıra ihlalleri yer alır.

Ayrıca kilitlenme konusunu da kısaca ele aldık: neden ortaya çıkar ve bu konuda ne yapılabilir. Bu sorun eşzamanlılık kadar eskidir ve konu hakkında yüzlerce makale yazılmıştır. Pratikte en iyi çözüm, dikkatli olmak, bir kilit edinme sırası geliştirmek ve böylece kilitlenmenin ilk etapta meydana gelmesini önlemektir. Bekleme gerektirmeyen yaklaşımlar da umut vaat etmektedir, çünkü bazı bekleme gerektirmeyen veri yapıları artık Linux dahil olmak üzere yaygın olarak kullanılan kütüphanelerde ve kritik sistemlerde yer almaktadır. Bununla birlikte, genellikten yoksun olmaları ve yeni bir bekleme gerektirmeyen veri yapısı geliştirmenin karmaşıklığı, bu yaklaşımın genel kullanımını muhtemelen sınırlayacaktır. Belki de en iyi çözüm yeni eşzamanlı programlama modelleri geliştirmektir: MapReduce (Google'dan) [GD02] gibi sistemlerde, programcılar belirli paralel hesaplama türlerini herhangi bir kilit olmadan tanımlayabilirler. Kilitler doğaları gereği sorunludur; belki de gerçekten mecbur kalmadıkça onları kullanmaktan kaçınmalıyız.

References

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

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

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

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

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

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

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

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

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

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

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

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

Homework (Code)(Ödev)

Bu ödev, kilitlenen (veya kilitlenmeyi önleyen) bazı gerçek kodları keşfetmenizi sağlar. Kodun farklı versiyonları, basitleştirilmiş bir vektör add() rutininde kilitlenmeyi önlemeye yönelik farklı yaklaşımlara karşılık gelir. Bu programlar ve ortak alt yapıları hakkında ayrıntılı bilgi için README'ye bakın.

Questions(Sorular)

1. 1.Öncelikle programların genel olarak nasıl çalıştığını ve bazı temel seçenekleri anladığınızdan emin olalım. `vector-deadlock.c`'nin yanı sıra `main-common.c` ve ilgili dosyalardaki kodu inceleyin.
   Şimdi, `./vector-deadlock -n 2 -l 1 -v` komutunu çalıştırın; bu komut her biri bir vektör ekleme (`-l 1`) yapan iki thread (`-n 2`) başlatır ve bunu ayrıntılı modda (`-v`) yapar. Çıktıyı anladığınızdan emin olun. Çıktı çalışmadan çalışmaya nasıl değişir?

   Thread'lerin çalışma sırası çalışmadan çalışmaya değişir.

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

   Kilitlenme asla gerçekleşmez. Tek çekirdekli ve çok çekirdekli bir makinede -l için daha yüksek değerlerle denendi.

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

   Kilitlenme -n üçten fazla olduğunda daha sık görülür.

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 bu `vektör add()` rutininde neden özel bir durum vardır?

   Kod, kilitlerin elde edilmesinde global order kullanarak kilitlenmeyi önler, böylece döngüsel bekleme olmaz. İki vektör aynı olduğunda, aynı kilidi iki kez beklemek kilitlenmeye neden olacaktır. Bu nedenle özel bir durum söz konusudur.

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

   ./vector-global-order -t -n 10 -l 100000 -d yaklaşık .05s sürer. Artan döngüler zamanı biraz doğrusal olarak artırır. Artan thread sayısı da zamanı doğrusal olarak artırır.

$ ./vector-global-order -t -n 2 -l 100000 -d
Time: 0.02 seconds

$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.05 seconds

$ ./vector-global-order -t -n 4 -l 50000 -d
Time: 0.12 seconds

6. Paralellik bayrağını (`-p`) etkinleştirirseniz ne olur? Her bir thread farklı vektörler eklemeye çalışırken (`-p` bunu sağlar) aynı vektörler üzerinde çalıştığında performansın ne kadar değişmesini beklersiniz?

Thread'ler kilit almak için beklemek zorunda olmadığından performans beklendiği gibi artar.

$ ./vector-global-order -t -n 2 -l 100000 -d -p
Time: 0.01 seconds

$ ./vector-global-order -t -n 2 -l 200000 -d -p
Time: 0.02 seconds

$ ./vector-global-order -t -n 4 -l 50000 -d -p
Time: 0.01 seconds

7. Şimdi `vector-try-wait.c` üzerinde çalışalım. Öncelikle kodu anladığınızdan emin olun. İlk `pthread_mutex_trylock()` çağrısı gerçekten gerekli mi? Şimdi kodu çalıştırın. Küresel düzen yaklaşımına kıyasla ne kadar hızlı çalışıyor? Kod tarafından sayılan yeniden deneme sayısı, thread sayısı arttıkça nasıl değişiyor?

pthread_mutex_trylock'a yapılan ilk çağrı gerçekten gerekli değildi. Normal bir pthread_mutex_lock beklerken uyuyacağı için muhtemelen performansı bile düşürür. Ama bu atlamaya devam edecek. Performans global order yaklaşımından daha kötüdür ve tekrar deneme sayısı katlanarak artar.

$ ./vector-try-wait -t -n 2 -l 100000 -d
Retries: 2173947
Time: 0.32 seconds

$ ./vector-try-wait -t -n 2 -l 200000 -d
Retries: 3775065
Time: 0.53 seconds

$ ./vector-try-wait -t -n 4 -l 50000 -d
Retries: 1341960
Time: 0.38 seconds

$ ./vector-try-wait -t -n 2 -l 100000 -d -p
Retries: 0
Time: 0.01 seconds

$ ./vector-try-wait -t -n 2 -l 200000 -d -p
Retries: 0
Time: 0.01 seconds

$ ./vector-try-wait -t -n 4 -l 50000 -d -p
Retries: 0
Time: 0.01 seconds

8. Şimdi `vector-avoid-hold-and-wait.c` dosyasına bakalım. Bu yaklaşımla ilgili temel sorun nedir? Hem `-p` ile hem de `-p` olmadan çalıştırıldığında performansı diğer sürümlerle karşılaştırıldığında nasıldır?

Bu durumda, sorun yalnızca -p bayrağı kullanıldığında ortaya çıkar. Tüm thread'ler farklı vektörler üzerinde çalışıyor olsalar bile global kilit için beklemek zorunda olduklarından performans düşer.

$ ./vector-avoid-hold-and-wait -t -n 2 -l 100000 -d
Time: 0.03 seconds

$ ./vector-avoid-hold-and-wait -t -n 2 -l 200000 -d
Time: 0.06 seconds

$ ./vector-avoid-hold-and-wait -t -n 4 -l 50000 -d
Time: 0.66 seconds

$ ./vector-avoid-hold-and-wait -t -n 2 -l 100000 -d -p
Time: 0.04 seconds

$ ./vector-avoid-hold-and-wait -t -n 2 -l 200000 -d -p
Time: 0.07 seconds

$ ./vector-avoid-hold-and-wait -t -n 4 -l 50000 -d -p
Time: 0.04 seconds

9. Son olarak `vector-nolock.c`'ye bakalım. Bu sürümde kilitler kullanılmaz hepsi; diğer versiyonlarla tam olarak aynı mantığı mı sağlıyor? Neden ya da neden değil?

   Lock yerine fetch ve add kullanır.
   Kilitleri hiç kullanmamak, programı aynı arayüzle çok daha basit hale getirir.

10. Şimdi hem thread'ler aynı iki vektör üzerinde çalışırken (no `-p`) hem de her thread ayrı vektörler üzerinde çalışırken (`-p`) performansını diğer versiyonlarla karşılaştırın. Bu kilitsiz sürüm nasıl bir performans sergiliyor?

    $ ./vector-nolock -t -n 2 -l 100000 -d
    Time: 0.39 seconds

    $ ./vector-nolock -t -n 2 -l 200000 -d
    Time: 0.83 seconds

    $ ./vector-nolock -t -n 4 -l 50000 -d
    Time: 0.34 seconds

    $ ./vector-nolock -t -n 2 -l 100000 -d -p
    Time: 0.06 seconds

    $ ./vector-nolock -t -n 2 -l 200000 -d -p
    Time: 0.11 seconds

    $ ./vector-nolock -t -n 4 -l 50000 -d -p
    Time: 0.04 seconds

    Ancak şaşırtıcı bir şekilde diğer çözümlerden çok daha hızlı değil. Global order yaklaşımından daha yavaş.