28

Kilitler

Eşzamanlı çalışmaya girişten itibaren, temel özelliklerden birini gördük.eşzamanlı programlamadaki problemler: bir dizi yürütmek istiyoruz.talimatların atomik olarak, ancak tek bir komutta kesintilerin varlığından dolayı.işlemci (veya aynı anda birden çok işlemcide çalışan birden çok iş parçacığı) (Rently), yapamadık. Bu bölümde, bu sorunu doğrudan Kilit olarak adlandırdığımız bir şeyle çözümlüyoruz. Programcılar:kaynak koduna kilitlerle açıklama ekleyin, bunları kritik bölümlerin etrafına koyun ve böylece bu tür kritik bölümlerin suç işlemiş gibi yürütülmesini sağlayın.

,

28.1 Kilitler: Temel Fikir

Örneğin, kritik kısmın, paylaşılan bir değişkenin kanuni güncellemesi gibi göründüğünü düşünelim:

```
balance = balance + 1;
```

Tabii ki, bağlantılı bir listeye bir öğe eklemek veya paylaşılan yapılara daha karmaşık güncellemeler gibi diğer kritik bölümler mümkündür, ama şimdilik bu basit örneğe devam edeceğiz. Kilit kullanmak için kritik bölümün etrafına şu şekilde bir kod yazıyoruz:

```
1      lock_t mutex; // bazı küresel olarak tahsis edilmiş kilit 'mutex'
2      ...
3      lock(&mutex);
4      balance = balance + 1;
5      unlock(&mutex);
```

Bir kilit yalnızca bir değişkendir ve bu nedenle birini kullanmak için bir tür **kilit değişkeni** (**lock variable)** bildirmeniz gerekir (yukarıdaki mutex gibi). Bu kilit değişkeni (veya kısaca "kilit") herhangi bir anda kilidin durumunu tutar. Ya mevcuttur (ya da kilidi açılmış ya da serbesttir) ve bu nedenle hiçbir iş

parçacığı kilidi tutamaz ya da edinilmiş (ya da kilitli ya da tutulmuş), ve böylece tam olarak bir iş parçacığı kilidi tutar ve muhtemelen kritik bir bölümdedir. Hangi iş parçacığının kilidi tuttuğu veya kilit edinimi sipariş etmek için bir kuyruk gibi başka bilgileri de veri türünde depolayabiliriz, ancak bunun gibi bilgiler kilidin kullanıcısından gizlenir.

Lock() ve unlock() rutinlerinin semantiği basittir. Arama rutin lock() kilidi almaya çalışır; başka bir iş parçacığı tutmazsa kilit (yani serbesttir), iş parçacığı kilidi alacak ve kritik bölüme girecektir; bu iş parçacığının bazen kilidin sahibi olduğu söylenir. Başka bir iş parçacığı daha sonra aynı kilit değişkeninde (bu örnekte muteks) kilit()'i çağırırsa, kilit başka bir iş parçacığı tarafından tutulurken geri dönmeyecektir; bu sayede kilidi tutan ilk thread oradayken diğer threadlerin kritik bölüme girmesi engellenir.

Kilidin sahibi unlock() öğesini çağırdığında, kilit artık tekrar kullanılabilir (ücretsiz). Kilidi bekleyen başka bir iş parçacığı yoksa (yani, başka hiçbir iş parçacığı lock()'u çağırmadı ve orada takılıp kalmadıysa), kilidin durumu basitçe serbest olarak değiştirilir. Bekleyen ileti dizileri varsa (kilitte () sıkışmış), bunlardan biri (eninde sonunda) kilidin durumundaki bu değişikliği fark edecek (veya bundan haberdar edilecek), kilidi alacak ve kritik bölüme girecektir.

Kilitler, programcılara zamanlama üzerinde minimum miktarda kontrol sağlar. Genel olarak, dizileri programcı tarafından oluşturulan ancak işletim sisteminin seçtiği herhangi bir şekilde işletim sistemi tarafından programlanan varlıklar olarak görürüz. Kilitler, bu kontrolün bir kısmını programlayıcıya geri verir; programcı, kodun bir bölümünün çevresine bir kilit koyarak, o kod içinde birden fazla iş parçacığının etkin olamayacağını garanti edebilir. Böylece kilitler, geleneksel işletim sistemi planlaması olan kaosu daha kontrollü bir etkinliğe dönüştürmeye yardımcı olur.

28.2 Pthread Kilitleri

POSIX kitaplığının bir kilit için kullandığı ad **muteks**tir(**mutex**), çünkü iş parçacıkları arasında **karşılıklı dışlama** (**mutual exclusion**) sağlamak için kullanılır, yani bir iş parçacığı kritik bölümdeyse, bölümü tamamlayana kadar diğerlerini girmekten dışlar. Bu nedenle, aşağıdaki POSIX iş parçacığı kodunu gördüğünüzde, bunun yukarıdakiyle aynı şeyi yaptığını anlamalısınız (yine kilitleme ve açma sırasında hataları kontrol eden sarmalayıcılarımızı kullanıyoruz):

```
1    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3    Pthread_mutex_lock(&lock);     // wrapper for pthread_mutex_lock()
4    balance = balance + 1;
5    Pthread_mutex_unlock(&lock);
```

Farklı değişkenleri korumak için farklı kilitler kullanıyor olabileceğimizden, burada POSIX sürümünün kilitlemek ve kilidi açmak için bir değişkeni geçtiğini de fark edebilirsiniz. Bunu yapmak eşzamanlılığı artırabilir: herhangi bir kritik bölüme erişildiğinde kullanılan büyük bir kilit yerine (kaba-taneli (**coarse-grained**) bir kilitleme stratejisi), genellikle farklı veriler ve veri yapıları farklı kilitlerle korunur ve böylece daha fazla iş parçacığının girmesine izin verilir. aynı anda kilitli kod (daha ayrıntılı (**fine-grained**) bir yaklaşım).

28.3 Bir Kilit Oluşturmak

Şimdiye kadar, bir programcının bakış açısından bir kilidin nasıl çalıştığını biraz anlamış olmalısınız. Ama nasıl bir kilit inşa etmeliyiz? Hangi donanım desteğine ihtiyaç var? Hangi işletim sistemi desteği? Bu bölümün geri kalanında ele alacağımız soru dizisi budur.

İşin püf noktası: KİLİT NASIL YAPILIR
Verimli bir kilidi nasıl inşa edebiliriz? Verimli kilitler, düşük maliyetle karşılıklı dışlama sağladı ve ayrıca aşağıda tartışacağımız birkaç başka özelliği de elde edebilir. Hangi donanım desteğine ihtiyaç var? Hangi işletim sistemi desteği?

Çalışan bir kilit oluşturmak için eski dostumuz olan donanımdan ve iyi dostumuz olan işletim sisteminden biraz yardıma ihtiyacımız olacak. Yıllar geçtikçe, çeşitli bilgisayar mimarilerinin komut setlerine bir dizi farklı donanım ilkelleri eklendi; bu talimatların nasıl uygulandığını incelemeyecek olsak da (sonuçta bu, bir bilgisayar mimarisi dersinin konusudur), kilit gibi bir karşılıklı dışlama ilkelini oluşturmak için bunların nasıl kullanılacağını inceleyeceğiz. Resmi tamamlamak için işletim sisteminin nasıl dahil olduğunu da inceleyeceğiz ve karmaşık bir kilitleme kitaplığı oluşturmamızı sağlayacağız.

28.4 Kilitleri Değerlendirmek

Herhangi bir kilit oluşturmadan önce, hedeflerimizin ne olduğunu anlamalıyız ve bu nedenle belirli bir kilit uygulamasının etkinliğini nasıl değerlendireceğimizi sormalıyız. Bir kilidin çalışıp çalışmadığını (ve iyi çalıştığını) değerlendirmek için önce bazı temel kriterler oluşturmalıyız. Birincisi, kilidin karşılıklı dışlamayı sağlamak olan temel görevini yerine getirip getirmediğidir. Temel olarak kilit, birden fazla iş parçacığının kritik bir bölüme girmesini önleyerek çalışıyor mu?

İkincisi adalettir. Kilit için yarışan her iş parçacığı, ücretsiz olduğunda onu edinme konusunda adil bir şans elde ediyor mu? Buna bakmanın başka bir yolu, daha aşırı bir durumu incelemektir: kilit için yarışan herhangi bir iş parçacığı, bunu yaparken aç kalır ve bu nedenle onu asla elde edemez mi?

Son kriter performanstır, özellikle kilit kullanılarak eklenen zaman giderleridir. Burada dikkate alınmaya değer birkaç farklı durum var. Biri, çekişme olmaması durumudur; tek bir iş parçacığı çalışırken ve kilidi tutup serbest bıraktığında, bunu yapmanın ek yükü nedir? Bir diğeri, birden çok iş parçacığının tek bir CPU'daki kilit için yarıştığı durumdur; bu durumda, performans endişeleri var mı? Son olarak, birden fazla CPU söz konusu olduğunda ve her birinde kilit için yarışan iş parçacığı olduğunda kilit nasıl çalışır? Bu farklı senaryoları karşılaştırarak, aşağıda açıklandığı gibi çeşitli kilitleme teknikleri kullanmanın performans etkisini daha iyi anlayabiliriz.

28.5 Kesintileri Kontrol Etme

Karşılıklı dışlama sağlamak için kullanılan en eski çözümlerden biri, kritik bölümler için kesintileri devre dışı bırakmaktı; bu çözüm, tek işlemcili sistemler için icat edilmiştir. Kod şöyle görünür:

```
1    void lock() {
2        DisableInterrupts();
3    }
4    void unlock() {
5        EnableInterrupts();
6    }
```

Böyle tek işlemcili bir sistem üzerinde çalıştığımızı varsayalım. Kritik bir bölüme girmeden önce kesintileri kapatarak (bir tür özel donanım talimatı kullanarak), kritik bölüm içindeki kodun kesintiye uğramamasını ve böylece atomikmiş gibi çalışmasını sağlıyoruz. Bitirdiğimizde, kesintileri yeniden etkinleştiririz (yine bir donanım komutu aracılığıyla) ve böylece program her zamanki gibi devam eder.

Bu yaklaşımın ana olumlu yönü basitliğidir. Bunun neden işe yaradığını anlamak için kesinlikle kafanızı çok fazla kaşımanıza gerek yok. Kesintisiz bir iş parçacığı yürüttüğü kodun çalışacağından ve başka hiçbir iş parçacığının buna müdahale etmeyeceğinden emin olabilir.

Negatifler, ne yazık ki, çoktur. İlk olarak, bu yaklaşım, herhangi bir çağıran iş parçacığının ayrıcalıklı bir işlem gerçekleştirmesine (kesmelerin açılıp kapatılmasına) izin vermemizi ve dolayısıyla bu özelliğin kötüye kullanılmadığına güvenmemizi gerektirir. Bildiğiniz gibi, keyfi bir programa güvenmemiz gerektiğinde, muhtemelen başımız belaya girer. Burada, sorun çeşitli şekillerde kendini gösterir: açgözlü bir program çalıştırmanın başında lock()'u çağırabilir ve böylece işlemciyi tekelleştirebilir; daha da kötüsü, hatalı veya kötü niyetli bir program lock()'u çağırabilir ve sonsuz bir döngüye girebilir. Bu son durumda, işletim sistemi hiçbir zaman sistemin kontrolünü yeniden kazanamaz ve tek bir başvuru yolu vardır: sistemi yeniden başlatmak. Kesmeyi devre dışı bırakmanın genel amaçlı bir eşitleme çözümü olarak kullanılması, uygulamalara çok fazla güvenilmesini gerektirir.

İkincisi, yaklaşım çoklu işlemcilerde çalışmaz. Farklı CPU'larda birden fazla iş parçacığı çalışıyorsa ve her biri aynı kritik bölüme girmeye çalışıyorsa, kesmelerin devre dışı bırakılıp bırakılmadığı önemli değildir; threadler diğer işlemcilerde çalışabilecek ve böylece kritik bölüme girebilecektir. Çoklu işlemciler artık sıradan olduğundan, genel çözümümüzün bundan daha iyisini yapması gerekecek.

Üçüncüsü ve en az önemlisi, bu yaklaşım verimsiz olabilir. Normal komut yürütmeyle karşılaştırıldığında, kesintileri maskeleyen veya maskesini kaldıran kod, modern CPU'lar tarafından daha yavaş yürütülme eğilimindedir.

Bu nedenlerden dolayı, kesmeleri kapatmak yalnızca sınırlı bağlamlarda karşılıklı dışlama ilkel olarak kullanılır.

Örneğin, bazı durumlarda, bir işletim sisteminin kendisi bazen kendi veri yapılarına erişirken atomikliği garanti etmek için veya en azından belirli dağınık kesme işleme durumlarının ortaya çıkmasını önlemek için kesme maskeleme kullanır.

Bu kullanım, bir şekilde ayrıcalıklı işlemleri gerçekleştirmek için kendisine her zaman güvenen işletim sistemi içinde güven sorunu ortadan kalktığı için mantıklıdır.

KENARA: **DEKKER VE PETERSON'UN ALGORİTMALARI**

1960'larda Dijkstra, eşzamanlılık problemini arkadaşlarına sordu ve onlardan biri, Theodorus Jozef Dekker adlı bir matematikçi bir çözüm buldu [D68]. Burada tartıştığımız, özel donanım yönergeleri ve hatta işletim sistemi desteği kullanan çözümlerin aksine, Dekker'in yaklaşımı yalnızca yükleri ve depoları kullanır (birbirlerine göre atomik olduklarını varsayarsak, bu erken donanımda geçerliydi).

Dekker'in yaklaşımı daha sonra burada gösterilen Peterson [P81] (ve dolayısıyla "Peterson'ın algoritması") tarafından geliştirildi. Bir daha açıklayalım, sadece yükler kullanılır ve buradaki düşünce, iki iş parçacığının asla aynı anda kritik bir yere girmemesini sağlamaktır. İşte Peterson'ın algoritması (iki iş parçacığı için);

```
  int flag[2]; int turn;

  void init() {
  flag[0] = flag[1] = 0;      // 1->thread wants to grab lock turn = 0;
                              // whose turn? (thread 0 or 1?)
  }
  void lock() {
  flag[self] = 1;             // self: thread ID of caller turn = 1 - self;
                              // make it other thread's turn while
((flag[1-self] == 1) && (turn == 1 - self))
  ; // spin-wait
  }
  void unlock() {
  flag[self] = 0;             // simply undo your intent
  }
```

Nedense, özel donanım desteği olmadan çalışan kilitler geliştirmek bir süre çok gündemde oldu ve teori tiplerinin üzerinde çalışılacak pek çok sorun verdi. Tabii ki, insanlar biraz donanım desteği almanın çok daha kolay olduğunu fark ettiklerinde (ve gerçekten de bu destek çoklu işlemenin en eski günlerinden beri vardı) tüm bunlar oldukça yararsız hale geldi. Ayrıca, yukarıdakiler gibi algoritmalar modern donanım üzerinde çalışmaz , bu nedenle onları eskisinden daha az kullanışlı hale getirir.

28.6 Test Et ve Ayarla

Kesintileri devre dışı bırakmak birden çok işlemcide çalışmadığından, sistem tasarımcıları kilitleme için donanım desteği icat etmeye başladılar. 1960'ların başında [M82] Burroughs B5000 gibi en eski çok işlemcili sistemler böyle bir desteğe sahipti; günümüzde tüm sistemler, tek CPU sistemleri için bile bu tür bir destek sağlamaktadır.

```
1   typedef struct   lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 -> lock is available, 1 -> held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)  // TEST the flag
10          ; // spin-wait (do nothing)
11      mutex->flag = 1;          // now SET it!
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

Şekil 28.1: **İlk Deneme: Basit Bir Bayrak**

Anlaşılması gereken en basit donanım desteği, **atomik değişim** (**atomic exchange**) olarak da bilinen, **test et ve ayarla talimatı** (**test-and-set instruction**) olarak bilinen şeydir. Test et ve ayarla'nın nasıl çalıştığını anlamak için önce onsuz basit bir kilit oluşturmaya çalışalım. Bu başarısız girişimde, kilidin tutulup tutulmadığını belirtmek için basit bir bayrak değişkeni kullanıyoruz.

Bu ilk denemede (Şekil 28.1), fikir oldukça basittir: bazı iş parçacığının bir kilide sahip olup olmadığını belirtmek için basit bir değişken kullanın. Kritik bölüme giren ilk iş parçacığı, bayrağın 1'e eşit olup olmadığını test eden (bu durumda değildir) ve ardından iş parçacığının artık kilidi tuttuğunu belirtmek için bayrağı 1'e ayarlayan lock()'u çağırır. Kritik bölüm bittiğinde, iş parçacığı unlock() öğesini çağırır ve bayrağı temizler, böylece kilidin artık tutulmadığını gösterir.

İlk iş parçacığı kritik bölümdeyken başka bir iş parçacığı lock()'u çağırırsa, o iş parçacığının unlock()'u çağırması ve bayrağı temizlemesi için basitçe while döngüsünde **döner-bekler** (**spin-wait**). İlk iş parçacığı bunu yaptığında, bekleyen iş parçacığı while döngüsünden düşecek, bayrağı kendisi için 1'e ayarlayacak ve kritik bölüme ilerleyecektir.

Ne yazık ki, kodun iki sorunu var: biri doğruluk, diğeri performans. Eşzamanlı programlama hakkında düşünmeye alıştığınızda, doğruluk problemini görmek kolaydır. Kodun Tablo 28.1'de araya girdiğini hayal edin (başlamak için bayrak=0 kabul edin).

Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()` while (flag == 1) flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Tablo 28.1: **İzleme: Karşılıklı Dışlama Yok** (**Trace: No Mutual Exclusion**)

İPUCU: EŞ ZAMANLILIK HAKKINDA KÖTÜ ZAMANLAYICI OLARAK DÜŞÜNÜN
Bu örnekten ayrıca, eşzamanlı yürütmeyi anlamaya çalışırken almamız gereken yaklaşımın bir anlamı var. Gerçekte yapmaya çalıştığınız şey, kötü niyetli bir planlayıcı olduğunuzu, senkronizasyon ilkellerini oluşturmaya yönelik zayıf girişimlerini boşa çıkarmak için en uygun olmayan zamanlarda iş parçacıklarını kesen bir planlayıcı olduğunuzu iddia etmektir. Kesintilerin tam sırası ihtimal dışı olsa da, bu mümkündür ve belirli bir yaklaşımın işe yaramadığını göstermek için göstermemiz gereken tek şey budur.

Bu serpiştirmeden de görebileceğiniz gibi, zamanında (zamansız?) kesintilerle, her iki iş parçacığının da bayraklarını 1'e ayarladığı ve böylece her iki iş parçacığının da kritik bölüme girebildiği bir durumu kolayca üretebiliriz. Bu kötü! Açıkça en temel şartı sağlamada başarısız olduk: Karşılıklı dışlamayı sağlamak.

Daha sonra ele alacağımız performans sorunu, bir iş parçacığının zaten tutulan bir kilidi elde etmek için bekleme şeklidir: dönme beklemesi (**spin-waiting**) olarak bilinen bir teknik olan bayrağın değerini durmadan kontrol eder. Döndürme bekleme, başka bir iş parçacığının bir kilidi serbest bırakmasını bekleyerek zaman harcar. Garsonun beklediği iş parçacığının çalışamadığı (en azından bir bağlam değişikliği gerçekleşene kadar) tek işlemcili bir işlemcide israf son derece yüksektir! Bu nedenle, ilerlerken ve daha sofistike çözümler geliştirirken, bu tür israfı önlemenin yollarını da düşünmeliyiz.

28.7 Çalışan Bir Döndürme Kilidi Oluşturma

Yukarıdaki örneğin arkasındaki fikir iyi olsa da, donanımdan biraz destek almadan uygulanması mümkün değildir. Neyse ki, bazı sistemler bu konsepte dayalı olarak basit kilitlerin oluşturulmasını desteklemek için bir talimat sağlar. Bu daha güçlü talimatın farklı adları vardır - SPARC'ta bu, yükleme/depolama işaretsiz bayt talimatıdır (ldstub), oysa x86'da atomik değişim talimatıdır (xchg) - ancak temel olarak platformlar arasında aynı şeyi yapar ve genellikle genellikle **test et ve ayarla**(**test-and-set**) olarak adlandırılır. **Test et ve ayarla**(**test-and-set**) komutunun ne yaptığını aşağıdaki C kod parçacığı ile tanımlarız:

```
1 int TestAndSet(int *ptr, int new) {
2     int old = *ptr; // fetch old value at ptr
3     *ptr = new;     // store 'new' into ptr
4     return old;     // return the old value
5 }
```

Test et ve ayarla(**test-and-set**) komutunun yaptığı şey aşağıdaki gibidir. Ptr tarafından işaret edilen eski değeri döndürür ve aynı anda söz konusu değeri yeni olarak günceller. Anahtar, elbette, bu işlem dizisinin **atomik olarak**(**atomically**.) gerçekleştirilmesidir. "Test et ve ayarla" olarak adlandırılmasının nedeni, aynı anda bellek konumunu yeni bir değere "ayarlarken" eski değeri (döndürülen budur) "test etmenizi" sağlamasıdır;

```
1   typedef struct   lock_t {
2       int flag;
3   } lock_t;
4
5   void init(lock_t *lock) {
6       // 0 indicates that lock is available, 1 that it is held
7       lock->flag = 0;
8   }
9
10  void lock(lock_t *lock) {
11      while (TestAndSet(&lock->flag, 1) == 1)
12          ; // spin-wait (do nothing)
13  }
14
15  void unlock(lock_t *lock) {
16      lock->flag = 0;
17  }
```

Figure 28.2: **A Simple Spin Lock Using Test-and-set**

Görünüşe göre, bu biraz basit bir döndürme kilidi oluşturmak için daha güçlü talimat yeterlidir, şimdi Şekil 28.2'de incelediğimiz gibi.

Bunun neden işe yaradığını anladığımızdan emin olalım. Önce bir iş parçacığının lock() işlevini çağırdığı ve başka hiçbir iş parçacığının şu anda kilidi tutmadığı bir durumu hayal edin; bayrak bu nedenle, `flag` 0 olmalıdır. İş parçacığı daha sonra `TestAndSet(flag,1)` olarak adlandırdığında, rutin 0 olan eski `flag` değerini döndürür; bu nedenle, `flag` değerini test eden çağrı ipliği, while döngüsünde dönerken yakalanmayacak ve kilidi alacaktır. İplik ayrıca değeri atomik olarak 1 olarak ayarlar, böylece kilidin şimdi tutulduğunu gösterir. İş parçacığı kritik bölümü ile bittiğinde, flag'ı sıfıra ayarlamak için kilidini ( ) olarak çağırır.

Hayal edebileceğimiz ikinci durum, bir iş parçacığı zaten kilidi tuttuğunda ortaya çıkar (`flag` 1). Bu durumda, bu iş parçacığı lock'u arayacak ve sonra TestAndSet (flag, 1) öğesini de çağırın. Bu kez, TestAndSet() aynı anda yeniden 1'e ayarlarken(çünkü kilit tutulduğu için), flag'daki eski değeri 1 olarak döndürür. Kilit başka bir iş parçacığı tarafından tutulduğu sürece, TestAndSet ( ) tekrar tekrar 1 döndürür ve böylece bu ipl iş parçacığı ik kilit nihayet serbest bırakılana kadar döner ve döner. Bayrak en sonunda başka bir iş parçacığı tarafından 0'a ayarlandığında, bu iş parçacığı `TestAndSet()`'i yeniden çağıracak ve bu, değeri atomik olarak 1'e ayarlarken şimdi 0 döndürecek ve böylece kilidi elde edecek ve kritik bölüme girecektir.

Hem test (eski kilit değerinin) hem de set (yeni değer) tek bir atomik işlem, yalnızca bir iş parçacığının kilidi almasını sağlıyoruz. Çalışan bir karşılıklı dışlama ilkesi bu şekilde oluşturulur!

Artık bu tür kilitlerin neden genellikle spin kilidi olarak adlandırıldığını da anlayabilirsiniz. Kilit kullanılabilir hale gelene kadar CPU döngüleri kullanarak inşa etmek ve basitçe döndürmek için en basit kilit türüdür. Tek bir işlemci üzerinde düzgün çalışması için, önleyici bir programlayıcı gerektirir (yani, zaman zaman farklı bir iş parçacığı çalıştırmak için bir iş parçacığını bir zamanlayıcı aracılığıyla kesecek olan). Önlem olmadan, bir CPU üzerinde dönen bir iş parçacığı onu asla terk etmeyeceğinden, döndürme kilitleri tek bir CPU üzerinde pek bir anlam ifade etmez.

28.8 Döndürme Kilitlerini Değerlendirme

Temel döndürme kilidimiz göz önüne alındığında, daha önce açıklanan eksenlerimiz boyunca ne kadar etkili olduğunu şimdi değerlendirebiliriz. Bir kilidin en önemli yönü doğruluktur: Karşılıklı dışlanma sağlar mı? Buradaki cevap kesinlikle evet: döndürme kilidi, bir seferde yalnızca tek bir iş parçacığının kritik bölüme girmesine izin verir. Böylece doğru bir kilide sahibiz.

Bir sonraki eksen **eşitliktir** (**fairness**). Bekleyen bir iş parçacığına döndürme kilidi ne kadar adil? Bekleyen bir ileti dizisinin kritik bölüme gireceğini garanti edebilir misiniz? Buradaki cevap maalesef kötü haber: Döndürmeli kilitler herhangi bir adalet garantisi sağlamıyor. Gerçekten de, çekişme altında dönen bir iplik sonsuza kadar dönebilir. Döndürme kilitleri adil değildir ve açlığa yol açabilir.

Son eksen **performanstır** (**performance**.). Döndürmeli kilit kullanmanın maliyeti nedir? Bunu daha dikkatli bir şekilde analiz etmek için birkaç farklı durumu düşünmenizi öneririz. İlkinde, iş parçacıklarının tek bir işlemcide kilit için yarıştığını hayal edin; ikincisinde, iş parçacıklarını birçok işlemciye yayılmış olarak düşünün.

Döndürme kilitleri için, tek CPU durumunda, performans genel giderleri oldukça sancılı olabilir; Kilidi tutan ipliğin kritik bir bölüm içinde önceden boşaltıldığı durumu hayal edin. Zamanlayıcı daha sonra, her biri kilidi elde etmeye çalışan diğer her iş parçacığını çalıştırabilir (N − 1 tane daha olduğunu hayal edin). Bu durumda, bu iş parçacıklarının her biri, CPU döngülerini boşa harcayarak CPU'dan vazgeçmeden önce bir zaman dilimi boyunca dönecektir.

Bununla birlikte, birden çok CPU'da döndürme kilitleri oldukça iyi çalışır (iş parçacığı sayısı kabaca CPU sayısına eşitse). Düşünce şu şekildedir: Her ikisi de bir kilit için yarışan CPU 1'deki Thread A'yı ve CPU 2'deki Thread B'yi hayal edin. A İş parçacığı (CPU 1) kilidi alırsa ve B İş Parçacığı bunu denerse, B döner (CPU 2'de). Bununla birlikte, muhtemelen kritik bölüm kısadır ve bu nedenle kilit yakında kullanılabilir hale gelir ve İplik B tarafından alınır. Başka bir işlemcide tutulan bir kilidi beklemek için döndürme, bu durumda çok fazla döngü kaybetmez ve bu nedenle kilitlenebilir. oldukça etkili.

28.9 Karşılaştır ve Değiştir (Compare-And-Swap)

Bazı sistemlerin sağladığı diğer bir ilkel donanım, karşılaştır ve değiştir talimatı (örneğin SPARC'ta adı verildiği gibi) veya karşılaştır ve değiştir (x86'da çağrıldığı gibi) olarak bilinir. Bu tek talimat için C sözde kodu Şekil 28.3'te bulunur.

Temel fikir, ptr tarafından belirtilen adresteki değerin beklenen degree

```
1      int CompareAndSwap(int *ptr, int expected, int new) {
2  int actual = *ptr;
3  if (actual == expected)
4      *ptr = new;
5  return actual;
6      }
```

Figure 28.3: **Compare-and-swap**

eşit olup olmadığını test etmek için karşılaştır ve değiştir yöntemidir; öyleyse, ptr ile işaret edilen hafıza konumunu yeni değerle güncelleyin. Değilse, hiçbir şey yapmayın. Her iki durumda da, bu bellek konumundaki gerçek değeri döndürün, böylece karşılaştırma ve takas kodunun başarılı olup olmadığını bilmesini sağlayın.

Compare-and-swap talimatıyla, test-and-set komutuna oldukça benzer bir şekilde bir kilit oluşturabiliriz. Örneğin, yukarıdaki lock() yordamını aşağıdakiyle değiştirebiliriz:

```
1    void lock(lock_t *lock) {
2        while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3            ; // spin
4    }
```

Kodun geri kalanı, yukarıdaki test et ve ayarla örneğiyle aynıdır. Bu kod oldukça benzer şekilde çalışır; sadece bayrağın 0 olup olmadığını kontrol eder ve eğer böylece, atomik olarak 1'de yer değiştirir ve böylece kilidi elde eder. denenen konular basılıyken kilidi alın, kilit açılıncaya kadar dönerken takılıp kalır sonunda serbest bırakıldı.

Compare-and-swap'in C tarafından çağrılabilir bir x86 sürümünü gerçekten nasıl yapacağınızı görmek istiyorsanız, bu kod dizisi yararlı olabilir ([S05]'ten):

```
1    char CompareAndSwap(int *ptr, int old, int new) {
2        unsigned char ret;
3
4        // Note that sete sets a 'byte' not the word
5        asm     volatile   (
                            lock\n"
                            cmpxchgl
                            sete %0\n"
                            "=q" (ret),
                  : "r" (new), "m"
                  : "memory");
                  return ret;
```

Son olarak, sezmiş olabileceğiniz gibi, compare-and-swap, test-and-set'den daha güçlü bir talimattır. Gelecekte beklemesiz senkronizasyona (**wait-free synchronization**)[H91] kısaca değindiğimizde bu gücün bir kısmını kullanacağız. Bununla birlikte, sadece onunla basit bir spin kilidi oluştururursak, davranışı yukarıda analiz ettiğimiz spin kilidi ile aynıdır.

28.10 Yük Bağlantılı ve Mağaza Koşullu

Bazı platformlar, kritik bölümler oluşturmaya yardımcı olmak için birlikte çalışan bir çift talimat sağlar. MIPS mimarisinde [H93], örneğin, yüke bağlı (**load-linked**) ve depo koşullu (**store-conditional**) talimatlar, kilitler ve diğer eşzamanlı yapılar oluşturmak için birlikte kullanılabilir. Bu talimatlar için C sözde kodu, Şekil 28.4'te bulunan gibidir. Alpha, PowerPC ve ARM benzer talimatlar sağlar.

```
1      int LoadLinked(int *ptr) {
2      return *ptr;
3      }
4
5      int StoreConditional(int *ptr, int value) {
6      if (no one has updated *ptr since the LoadLinked to this address)
{
7   *ptr = value;
8   return 1; // success!
9      } else {
10  return 0; // failed to update
11     }
12     }
```

Figure 28.4: **Load-linked And Store-conditional**

```
1      void lock(lock_t *lock) {
2      while (1) {
3   while (LoadLinked(&lock->flag) == 1)
4      ; // spin until it's zero
5   if (StoreConditional(&lock->flag, 1) == 1)
6      return; // if set-it-to-1 was a success: all done
7              // otherwise: try it all over again
8      }
9      }
10
11     void unlock(lock_t *lock) {
12     lock->flag = 0;
13     }
```

Figure 28.5: **Using LL/SC To Build A Lock**

Yük bağlantılı, tipik bir yükleme talimatı gibi çalışır ve basitçe bellekten bir değer alır ve onu bir kayda yerleştirir. En önemli fark, yalnızca adrese aralıklı bir depolama yapılmadıysa başarılı olan (ve az önce yük bağlantısının yapıldığı adreste depolanan değeri güncelleyen) mağaza koşuluyla gelir. Başarı durumunda, depolama koşulu 1 döndürür ve ptr'deki değeri değer olarak günceller; başarısız olursa, ptr'deki değer güncellenmez ve 0 döndürülür.

Kendinize bir meydan okuma olarak, yük bağlantılı ve mağaza koşullu kullanarak nasıl kilit oluşturacağınızı düşünün. Ardından, işiniz bittiğinde, basit bir çözüm sağlayan aşağıdaki koda bakın. Yap! Çözüm Şekil 28.5'te.

Lock() kodu, tek ilginç parçadır. İlk olarak, flag'ın 0'a ayarlanmasını bekleyen bir thread döner (ve böylece lock'in tutulmadığını gösterir). Bir kez böyle olunca, thread, mağaza koşullu yoluyla lock elde etmeye çalışır; başarılı olursa, iş parçacığı atomik olarak flag değerini 1 olarak değiştirir ve böylece kritik bölüme geçebilir.

Mağaza koşulunun başarısızlığının nasıl ortaya çıkabileceğine dikkat edin. Bir iş parçacığı lock()'u çağırır ve lock tutulmadığı için 0 döndürerek load-linked'i yürütür. Koşullu saklama girişiminde bulunmadan önce kesilir ve başka bir iş parçacığı lock kodunu girer, ayrıca yük bağlantılı talimatı yürütür ve ayrıca bir 0 alır ve devam eder. Bu noktada, iki iş parçacığının her biri load-linked'i yürüttü ve her biri store-conditional'ı denemek üzere. Bu talimatların temel özelliği, bu iş parçacıklarından yalnızca birinin flag 1'e güncellemeyi başarması ve böylece lock'yı almasıdır;

İPUCU: DAHA AZ KOD DAHA İYİ KODDUR (LAUER YASASI)

(TIP: LESS CODE IS BETTER CODE (LAUER'S LAW))

Programcılar, bir şeyi yapmak için ne kadar kod yazdıklarıyla övünme eğilimindedir. Bunu yapmak temelden bozulur. Kişinin övünmesi gereken şey, daha ziyade, belirli bir görevi gerçekleştirmek için ne kadar az kod yazdığıdır. Kısa, özlü kod her zaman tercih edilir; anlaşılması muhtemelen daha kolaydır ve daha az hata içerir. Hugh Lauer'in Pilot işletim sisteminin yapımından bahsederken söylediği gibi: "Aynı kişilerin iki kat daha fazla zamanı olsaydı, kodun yarısı kadar iyi bir sistem üretebilirlerdi." [L81] Biz buna Lauer Yasası diyeceğiz ve hatırlamaya değer. Bir dahaki sefere ödevi bitirmek için ne kadar kod yazdığınızla övünürseniz, tekrar düşünün veya daha iyisi, geri dönün, yeniden yazın ve kodu olabildiğince açık ve öz yapın.

depo koşulunu deneyen ikinci iş parçacığı başarısız olacaktır (çünkü diğer iş parçacığı yük bağlantılı ve depo koşullu arasındaki bayrağın değerini güncellemiştir) ve bu nedenle kilidi yeniden elde etmeyi denemek zorunda kalacaktır.

Birkaç yıl önce, lisans öğrencisi David Capel, kısa devre boolean koşullu ifadelerden hoşlananlarınız için yukarıdakilerin daha özlü bir biçimini önerdi. Neden eşdeğer olduğunu anlayabilecek misin bir bak. Kesinlikle daha kısa!

```
1    void lock(lock_t *lock) {
2      while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
3        ; // spin
4    }
```

28.11 Fetch-Ve-Add

Son bir ilkel donanım, belirli bir adreste eski değeri döndürürken bir değeri atomik olarak artıran getir ve ekle talimatıdır. Getir ve ekle talimatı için C sözde kodu şöyle görünür:

```
1    int FetchAndAdd(int *ptr) {
2        int old = *ptr;
3        *ptr = old + 1;
4        return old;
5    }
```

Bu örnekte, Mellor-Crummey ve Scott [MS91] tarafından tanıtıldığı gibi, daha ilginç bir bilet kilidi oluşturmak için getir ve ekle yöntemini kullanacağız. Kilitleme ve kilit açma kodu, Şekil 28.6'da gördüğünüz gibi görünür.

Bu çözüm, tek bir değer yerine, bir lock oluşturmak için bilet ve dönüş değişkenini birlikte kullanır. Temel işlem oldukça basittir: Bir iş parçacığı bir lock elde etmek istediğinde, önce bilet değeri üzerinde atomik bir getir ve ekle işlemi gerçekleştirir; bu değer artık bu konunun "dönüşü" (myturn) olarak kabul edilir. Küresel olarak paylaşılan lock->turn daha sonra sıranın hangi iş parçacığına geldiğini belirlemek için kullanılır; ne zaman (myturn == turn) belirli bir iş parçacığı için, kritik bölüme girme sırası o iş parçacığındadır.

```
1      typedef struct   lock_t {
2   int ticket;
3   int turn;
4      } lock_t;
5
6      void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn    = 0;
9      }
10
11     void lock(lock_t *lock) {
12  int myturn = FetchAndAdd(&lock->ticket);
13  while (lock->turn != myturn)
14      ; // spin
15     }
16
17     void unlock(lock_t *lock) {
18  FetchAndAdd(&lock->turn);
19     }
```

Figure 28.6: **Ticket Locks**

Lock açma, sıradaki bekleyen iş parçacığının (eğer varsa) şimdi kritik bölüme girebilmesi için dönüşü artırarak gerçekleştirilir.

Önceki denemelerimize kıyasla bu çözümle ilgili önemli bir farka dikkat edin: tüm ileti dizileri için ilerleme sağlar. Bir iş parçacığına bilet değeri atandığında, gelecekte bir noktada planlanacaktır. (önündekiler kritik bölümü geçip lock açtıktan sonra). Daha önceki denemelerimizde böyle bir garanti yoktu; test ve ayarla (örneğin) üzerinde dönen bir iş parçacığı, diğer iş parçacıkları lock'yi alıp serbest bıraksa bile sonsuza kadar dönebilir.

28.12 Özet: Çok Fazla Dönen

Donanım tabanlı basit lock'lerimiz basittir (yalnızca birkaç satır kod) ve çalışırlar (hatta isterseniz biraz kod yazarak bunu kanıtlayabilirsiniz), bunlar herhangi bir sistemin veya kodun iki mükemmel özelliğidir. Ancak bazı durumlarda bu çözümler oldukça verimsiz olabiliyor. Tek bir işlemcide iki iş parçacığı çalıştırdığınızı hayal edin. Şimdi bir iş parçacığının (0 iş parçacığı) kritik bir bölümde olduğunu ve bu nedenle kilitlendiğini ve ne yazık ki kesintiye uğradığını hayal edin. İkinci iş parçacığı (iş parçacığı 1) şimdi kilidi almaya çalışır, ancak tutulduğunu bulur. Böylece dönmeye başlar ve döndür. Sonra biraz daha dönüyor. Ve son olarak, bir zamanlayıcı kesintisi söner, iş parçacığı 0 tekrar çalıştırılır, bu da kilidi serbest bırakır ve son olarak (diyelim ki bir dahaki sefere çalıştığında), iş parçacığı 1'in çok fazla dönmesi gerekmeyecek ve kilidi elde edebilecektir. . Böylece, ne zaman bir iş parçacığı böyle bir durumda dönerken yakalanırsa, değişmeyecek bir değeri kontrol etmekten başka hiçbir şey yapmadan bütün bir zaman dilimini boşa harcar! Bir kilit için yarışan N iş parçacığı ile sorun daha da kötüleşir; N − 1 zaman dilimi benzer bir şekilde boşa harcanabilir, basitçe döndürülür ve tek bir iş parçacığının lock'yı açması beklenir. Ve böylece, bir sonraki sorunumuz:

EN ÖNEMLİ NOKTA: DÖNÜŞTEN NASIL KAÇINILIR

(THE CRUX: HOW TO AVOID SPINNING)
CPU'da gereksiz yere zaman kaybetmeyen bir lock'yi nasıl geliştirebiliriz?

Donanım desteği tek başına sorunu çözemez. OS desteğine de ihtiyacımız olacak! Şimdi bunun nasıl işe yarayabileceğini anlayalım.

28.13 Basit Bir Yaklaşım: Sadece Verim, Bebek

Donanım desteği bizi oldukça ileri götürdü: çalışan kilitler ve hatta (bilet kilidi durumunda olduğu gibi) kilit edinmede adalet. Bununla birlikte, yine de bir sorunumuz var: kritik bir bölümde bir bağlam değişikliği meydana geldiğinde ve iş parçacıkları, kesme (lock-holding) iş parçacığının tekrar çalışmasını bekleyerek sonsuz bir şekilde dönmeye başladığında ne yapmalıyız?

İlk denememiz basit ve arkadaşça bir yaklaşımdır: döneceğiniz zaman , bunun yerine CPU'yu başka bir iş parçacığına bırakın. Ya da Al Davis'in dediği gibi, "sadece teslim ol, bebeğim!" [D91]. Şekil 28.7 yaklaşımı sunmaktadır.

Bu yaklaşımda, bir iş parçacığının CPU'dan vazgeçmek ve diğer bir iş parçacığının çalışmasına izin vermek istediğinde çağırabileceği bir işletim sistemi ilkel verimi () olduğunu varsayıyoruz . Bir iş parçacığı üç durumdan birinde (çalışıyor, hazır veya engellenmiş) olabileceğinden, bunu arayanı **running (çalışan)** durumdan **hazır (ready)** durumuna taşıyan ve böylece başka bir iş parçacığını çalışmaya teşvik eden bir işletim sistemi çağrısı olarak düşünebilirsiniz.

Bir CPU'da iki iş parçacığı olan örneği düşünün; Bu durumda, verime dayalı yaklaşımımız oldukça iyi çalışır. Bir iş parçacığı lock() öğesini çağırır ve tutulan bir lock bulursa, CPU'yu verir ve böylece diğer iş parçacığı çalışır ve kritik bölümünü bitirir. Bu basit durumda, verim yaklaşımı iyi çalışır.

Şimdi bir kilit için art arda mücadele eden birçok iş parçacığının olduğu durumu ele alalım. Bu durumda, bir iş parçacığı lock'u alırsa ve serbest bırakmadan önce önlenirse , diğer 99'un her biri lock()'u çağırır, tutulan lock'ubulur ve CPU'yu verir.

```
1    void init() {
2        flag = 0;
3    }
4
5    void lock() {
6        while (TestAndSet(&flag, 1) == 1)
7            yield(); // give up the CPU
8    }
9
10   void unlock() {
11       flag = 0;
12   }
```

Figure 28.7: **Lock With Test-and-set And Yield**

Bir tür varsayarsak round-robin zamanlayıcının her biri, 99'dan her biri bu çalıştır ve ver işlemini yürütecek lock'yi tutan iplik tekrar çalışmaya başlamadan önceki desen. daha iyi iken döndürme yaklaşımımızdan (99 zaman dilimini döndürerek boşa harcar), bu yaklaşım hala maliyetlidir; bir bağlam anahtarının maliyeti önemli olabilir, ve bu nedenle bol miktarda atık var.

Daha da kötüsü, açlık sorununu hiç çözemedik. Diğer iş parçacıkları tekrar tekrar kritik bölüme girip çıkarken bir iş parçacığı sonsuz bir verim döngüsüne yakalanabilir. Açıkça bu sorunu doğrudan ele alan bir yaklaşıma ihtiyacımız olacak.

28.14 Kuyrukları Kullanmak: Dönmek Yerine Uyumak

Önceki yaklaşımlarımızla ilgili asıl sorun, çok fazla şeyi şansa bırakmalarıdır. Zamanlayıcı, hangi iş parçacığının daha sonra çalışacağını belirler; zamanlayıcı kötü bir seçim yaparsa, kilidi beklerken dönmesi (ilk yaklaşımımız) veya CPU'yu hemen vermesi (ikinci yaklaşımımız) gereken bir iş parçacığı çalışır. Her iki durumda da, israf potansiyeli vardır ve açlığın önlenmesi mümkün değildir.

Bu nedenle, mevcut sahibi serbest bıraktıktan sonra lock'yi kimin alacağı üzerinde açıkça bir miktar kontrol uygulamalıyız. Bunu yapmak için, biraz daha fazla işletim sistemi desteğine ve lock'ye girmek için hangi konuların beklediğini takip etmek için bir kuyruğa ihtiyacımız olacak.

Basit olması için Solaris tarafından sağlanan desteği iki çağrı açısından kullanacağız: çağıran bir iş parçacığını uyku moduna geçirmek için park() ve iş parçacığı kimliği tarafından belirlenen belirli bir iş parçacığını uyandırmak için unpark(threadID). Bu iki rutin, bekletilen bir lock'yi almaya çalışırsa arayanı uyku moduna geçiren ve lock boş olduğunda onu uyandıran bir lock oluşturmak için art arda kullanılabilir. Bu tür ilkellerin olası bir kullanımını anlamak için Şekil 28.8'deki koda bakalım.

Bu örnekte birkaç ilginç şey yapıyoruz. İlk olarak, daha verimli bir lock yapmak için eski test ve set fikrini açık bir lock garson kuyruğu ile birleştiriyoruz. İkincisi, lock'yi kimin alacağını kontrol etmek ve böylece açlıktan kaçınmak için bir kuyruk kullanıyoruz.

Korumanın nasıl kullanıldığını, temel olarak bayrağın etrafında bir döndürme kilidi (spin-lock) ve lock'un kullandığı kuyruk manipülasyonları olarak fark edebilirsiniz. Dolayısıyla bu yaklaşım, spin-beklemeden tamamen kaçınmaz; lock'yi alırken veya serbest bırakırken bir iş parçacığı kesintiye uğrayabilir ve bu nedenle diğer iş parçacıklarının bunun tekrar çalışması için dönmesini beklemesine neden olabilir. Ancak, eğirme için harcanan zaman oldukça sınırlıdır (kullanıcı tanımlı kritik bölüm yerine kilitleme ve kilit açma kodunun içinde yalnızca birkaç talimat) ve dolayısıyla bu yaklaşım makul olabilir.

İkinci olarak, lock() içinde, bir iş parçacığı lock'yi (zaten tutuluyor) elde edemediğinde, kendimizi bir kuyruğa (gettid()'i çağırarak) eklemeye, korumayı 0'a ayarlamaya ve CPU'yu vermeye dikkat ettiğimizi fark edebilirsiniz. Okuyucu için bir soru: Koruma kilidinin serbest bırakılması parktan() sonra gelseydi ve daha önce olmasaydı ne olurdu? İpucu: kötü bir şey.

```
1   typedef struct   lock_t {
2       int flag;
3       int guard;
4       queue_t *q;
5   } lock_t;
6
7   void lock_init(lock_t *m) {
8       m->flag  = 0;
9       m->guard = 0;
10      queue_init(m->q);
11  }
12
13  void lock(lock_t *m) {
14      while (TestAndSet(&m->guard, 1) == 1)
15          ; //acquire guard lock by spinning
16      if (m->flag == 0) {
17          m->flag = 1; // lock is acquired
18          m->guard = 0;
19      } else {
20          queue_add(m->q, gettid());
21          m->guard = 0;
22          park();
23      }
24  }
25
26  void unlock(lock_t *m) {
27      while (TestAndSet(&m->guard, 1) == 1)
28          ; //acquire guard lock by spinning
29      if (queue_empty(m->q))
30          m->flag = 0; // let go of lock; no one wants it
31      else
32          unpark(queue_remove(m->q)); // hold lock (for next thread!)
33      m->guard = 0;
34  }
```

Figure 28.8: **Lock With Queues, Test-and-set, Yield, And Wakeup**

Başka bir iş parçacığı uyandığında bayrağın tekrar 0'a ayarlanmadığı ilginç gerçeğini de fark edebilirsiniz. Bunun nedeni nedir? Eh, bu bir hata değil, bir zorunluluktur! Bir iş parçacığı uyandırıldığında, park() işlevinden dönüyormuş gibi olacaktır; ancak, koruyucuyu kodun o noktasında tutmaz ve bu nedenle flag'yi 1 olarak ayarlamaya bile çalışamaz. Böylece, lock'yi direkt olarak lock'yi serbest bırakan iş parçacığından onu alan bir sonraki iş parçacığına geçiriyoruz; flag arada 0 olarak ayarlanmadı.

Son olarak, park() çağrısından hemen önce çözümde algılanan yarış koşulunu fark edebilirsiniz. Sadece yanlış zamanlamayla, lock artık tutulmayana kadar uyuması gerektiğini varsayarak bir ileti dizisi park etmek üzere olacaktır. O sırada başka bir iş parçacığına geçiş (örneğin, lock'yi tutan bir iş parçacığı), örneğin o iş parçacığı daha sonra lock'yi serbest bırakırsa soruna yol açabilir. İlk iş parçacığı tarafından sonraki park, daha sonra (potansiyel olarak) sonsuza kadar uyur. Bu sorun bazen **uyanma/bekleme yarışı**(**wakeup/waiting race**) olarak adlandırılır; bundan kaçınmak için biraz fazladan çalışmamız gerekiyor.

Solaris bu sorunu üçüncü bir sistem çağrısı ekleyerek çözer: setpark(). Bu rutini çağırarak, bir iş parçacığı park etmek üzere olduğunu gösterebilir. Daha sonra kesintiye uğrarsa ve park fiilen çağrılmadan önce başka bir iş parçacığı parktan çıkar çağrısı yaparsa, sonraki park uyumak yerine hemen geri döner.

Lock() içindeki kod değişikliği oldukça küçüktür:

```
1       queue_add(m->q, gettid());
2       setpark(); // new code
3       m->guard = 0;
```

Farklı bir çözüm, korumayı çekirdeğe geçirebilir. Bu durumda çekirdek, kilidi atomik olarak serbest bırakmak ve çalışan iş parçacığını kuyruğundan çıkarmak için önlemler alabilir.

28.15 Farklı İşletim Sistemi, Farklı Destek

We have thus far seen one type of support that an OS can provide in order to build a more efficient lock in a thread library. Other OS's provide similar support; the details vary.

For example, Linux provides something called a **futex** which is simi- lar to the Solaris interface but provides a bit more in-kernel functionality. Specifically, each futex has associated with it a specific physical mem- ory location; associated with each such memory location is an in-kernel queue. Callers can use futex calls (described below) to sleep and wake as need be.

Specifically, two calls are available. The call to `futex wait(address, expected)` puts the calling thread to sleep, assuming the value at `address` is equal to `expected`. If it is *not* equal, the call returns immediately. The call to the routine `futex wake(address)` wakes one thread that is wait- ing on the queue. The usage of these in Linux is as found in 28.9.

This code snippet from `lowlevellock.h` in the nptl library (part of the gnu libc library) [L09] is pretty interesting. Basically, it uses a single integer to track both whether the lock is held or not (the high bit of the integer) and the number of waiters on the lock (all the other bits). Thus, if the lock is negative, it is held (because the high bit is set and that bit determines the sign of the integer). The code is also interesting because it shows how to optimize for the common case where there is no contention: with only one thread acquiring and releasing a lock, very little work is done (the atomic bit test-and-set to lock and an atomic add to release the lock). See if you can puzzle through the rest of this "real-world" lock to see how it works.

28.16 İki Fazlı Kilitler

Son bir not: Linux yaklaşımı, en azından 1960'ların [M82] başındaki Dahm Locks'a kadar uzanan, yıllardır ara sıra kullanılan eski bir yaklaşımın tadına sahiptir, ve şimdi iki fazlı kilit olarak anılıyor. İki fazlı bir lock, özellikle lock serbest bırakılmak üzereyse döndürmenin yararlı olabileceğini fark eder. Böylece ilk aşamada, lock'yi elde edebileceğini umarak lock bir süre döner.

```
1    void mutex_lock (int *mutex) {
2      int v;
3      /* Bit 31 was clear, we got the mutex (this is the fastpath)  */
4      if (atomic_bit_test_set (mutex, 31) == 0)
5        return;
6      atomic_increment (mutex);
7      while (1) {
8          if (atomic_bit_test_set (mutex, 31) == 0) {
9              atomic_decrement (mutex);
10             return;
11         }
12         /* We have to wait now. First make sure the futex value
13            we are monitoring is truly negative (i.e. locked). */
14         v = *mutex;
15         if (v >= 0)
16           continue;
17         futex_wait (mutex, v);
18     }
19   }
20
21   void mutex_unlock (int *mutex) {
22     /* Adding 0x80000000 to the counter results in 0 if and only if
23       there are not other interested threads */
24     if (atomic_add_zero (mutex, 0x80000000))
25       return;
26
27     /* There are other threads waiting for this mutex,
28        wake one of them up.  */
29     futex_wake (mutex);
```

Figure 28.9: **Linux-based Futex Locks**

Ancak, ilk döndürme aşamasında kilit elde edilmezse, arayanın uykuya daldığı ve ancak kilit daha sonra serbest kaldığında uyandığı ikinci bir aşamaya geçilir. Yukarıdaki Linux kilidi, böyle bir kilidin bir biçimidir, ancak yalnızca bir kez döner; bunun genelleştirilmesi, uyumak için **futex** desteğini kullanmadan önce sabit bir süre boyunca bir döngüde dönebilir.

İki fazlı kilitler, iki iyi fikri birleştirmenin gerçekten de daha iyi bir fikir verebileceği **hibrit** (**hybrid**) yaklaşımın bir başka örneğidir. Tabii ki, donanım ortamı, iş parçacığı sayısı ve diğer iş yükü ayrıntıları dahil olmak üzere pek çok şeye güçlü bir şekilde bağlıdır. Her zaman olduğu gibi, tüm olası kullanım durumları için iyi olan tek bir genel amaçlı kilit yapmak oldukça zorlu bir iştir.

28.17 Özet

Yukarıdaki yaklaşım, günümüzde gerçek kilitlerin nasıl oluşturulduğunu göstermektedir: biraz donanım desteği (daha güçlü bir talimat biçiminde) artı bazı işletim sistemi desteği (örneğin, Solaris veya futex üzerinde park() ve unpark() ilkelleri biçiminde) Linux'ta). Elbette ayrıntılar farklılık gösterir ve bu tür bir kilitlemeyi gerçekleştirecek kesin kod genellikle yüksek düzeyde ayarlanmıştır. Daha fazla ayrıntı görmek istiyorsanız Solaris veya Linux açık kaynak kodu temellerine göz atın; büyüleyici bir okuma [L09, S09].

References

[D91] "Just Win, Baby: Al Davis and His Raiders" by Glenn Dickey. Harcourt, 1991. *The book about Al Davis and his famous quote. Or, we suppose, the book is more about Al Davis and the Raiders, and not so much the quote. To be clear: we are not recommending this book, we just needed a citation.*

[D+13] "Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask" by Tudor David, Rachid Guerraoui, Vasileios Trigonakis. SOSP '13, Nemacolin Woodlands Resort, Pennsylvania, November 2013. *An excellent paper comparing many different ways to build locks using hardware primitives. Great to see how many ideas work on modern hardware.*

[D68] "Cooperating sequential processes" by Edsger W. Dijkstra. 1968. Available online here: `http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF`. *One of the early seminal papers. Discusses how Dijkstra posed the original concurrency problem, and Dekker's solution.*

[H93] "MIPS R4000 Microprocessor User's Manual" by Joe Heinrich. Prentice-Hall, June 1993. Available: http://cag.csail.mit.edu/raw/documents/R4400_Uman_book_Ed2.pdf. *The old MIPS user's manual. Download it while it still exists.*

[H91] "Wait-free Synchronization" by Maurice Herlihy. ACM TOPLAS, Volume 13: 1, January 1991. *A landmark paper introducing a different approach to building concurrent data structures. Because of the complexity involved, some of these ideas have been slow to gain acceptance in deployment.*

[L81] "Observations on the Development of an Operating System" by Hugh Lauer. SOSP '81, Pacific Grove, California, December 1981. *A must-read retrospective about the development of the Pilot OS, an early PC operating system. Fun and full of insights.*

[L09] "glibc 2.9 (include Linux pthreads implementation)" by Many authors.. Available here: `http://ftp.gnu.org/gnu/glibc`. *In particular, take a look at the nptl subdirectory where you will find most of the pthread support in Linux today.*

[M82] "The Architecture of the Burroughs B5000: 20 Years Later and Still Ahead of the Times?" by A. Mayer. 1982. Available: `www.ajwm.net/amayer/papers/B5000.html`. *"It (RDLK) is an indivisible operation which reads from and writes into a memory location." RDLK is thus test-and-set! Dave Dahm created spin locks ("Buzz Locks") and a two-phase lock called "Dahm Locks."*

[M15] "OSSpinLock Is Unsafe" by J. McCall. `mjtsai.com/blog/2015/12/16/osspinlock-is-unsafe`. *Calling OSSpinLock on a Mac is unsafe when using threads of different priorities – you might spin forever! So be careful, Mac fanatics, even your mighty system can be less than perfect...*

[MS91] "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors" by John M. Mellor-Crummey and M. L. Scott. ACM TOCS, Volume 9, Issue 1, February 1991. *An excellent and thorough survey on different locking algorithms. However, no operating systems support is used, just fancy hardware instructions.*

[P81] "Myths About the Mutual Exclusion Problem" by G.L. Peterson. Information Processing Letters, 12(3), pages 115–116, 1981. *Peterson's algorithm introduced here.*

[R97] "What Really Happened on Mars?" by Glenn E. Reeves. `research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html`. *A description of priority inversion on Mars Pathfinder. Concurrent code correctness matters, especially in space!*

[S05] "Guide to porting from Solaris to Linux on x86" by Ajay Sood, April 29, 2005. Available: `http://www.ibm.com/developerworks/linux/library/l-solar/`.

[S09] "OpenSolaris Thread Library" by Sun.. Code: `src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libc/port/threads/synch.c`. *Pretty interesting, although who knows what will happen now that Oracle owns Sun. Thanks to Mike Swift for the pointer.*

[W09] "Load-Link, Store-Conditional" by Many authors.. `en.wikipedia.org/wiki/Load-Link/Store-Conditional`. *Can you believe we referenced Wikipedia? But, we found the information there and it felt wrong not to. Further, it was useful, listing the instructions for the different architectures: `ldl_l/stl_c` and `ldq_l/stq_c` (Alpha), `lwarx/stwcx` (PowerPC), `ll/sc` (MIPS), and `ldrex/strex` (ARM). Actually Wikipedia is pretty amazing, so don't be so harsh, OK?*

[WG00] "The SPARC Architecture Manual: Version 9" by D. Weaver, T. Germond. SPARC International, 2000. `http://www.sparc.org/standards/SPARCV9.pdf`. *See `developers.sun.com/solaris/articles/atomic_sparc/` for more on atomics.*

Ödev (Simülasyon)

Bu program, x86.py, farklı iş parçacığı ara ayrılmalarının nasıl yarış koşullarına neden olduğunu veya bunlardan nasıl kaçındığını görmenizi sağlar. Programın nasıl çalıştığıyla ilgili ayrıntılar için BENİOKU'ya bakın ve aşağıdaki soruları yanıtlayın.

Sorular

1. İncele: flag.s. Bu kod, tek bir bellek bayrağıyla kilitlemeyi "uygular". Montajı anlayabiliyor musun?

2. Varsayılanlarla çalıştırdığınızda flag.s çalışıyor mu? Değişkenleri ve kayıtları izlemek için -M ve -R flag'lerini kullanın (ve değerlerini görmek için -c'yi açın). flag'de hangi değerin biteceğini tahmin edebilir misiniz?

./x86.py -p flag.s -M flag,count -R ax,bx -c

```
yigit@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p flag.s -M flag,count -R ax,bx -c
ARG seed 0
ARG numthreads 2
ARG program flag.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace flag,count
ARG regtrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False


 flag count      ax    bx          Thread 0                Thread 1

    0     0       0     0
    0     0       0     0   1000 mov  flag, %ax
    0     0       0     0   1001 test $0, %ax
    0     0       0     0   1002 jne  .acquire
    1     0       0     0   1003 mov  $1, flag
    1     0       0     0   1004 mov  count, %ax
    1     0       1     0   1005 add  $1, %ax
    1     1       1     0   1006 mov  %ax, count
    0     1       1     0   1007 mov  $0, flag
    0     1       1    -1   1008 sub  $1, %bx
    0     1       1    -1   1009 test $0, %bx
    0     1       1    -1   1010 jgt .top
    0     1       1    -1   1011 halt
    0     1       0     0   ----- Halt;Switch -----  ----- Halt;Switch -----
    0     1       0     0                           1000 mov  flag, %ax
    0     1       0     0                           1001 test $0, %ax
    0     1       0     0                           1002 jne  .acquire
    1     1       0     0                           1003 mov  $1, flag
    1     1       1     0                           1004 mov  count, %ax
    1     1       2     0                           1005 add  $1, %ax
    1     2       2     0                           1006 mov  %ax, count
    0     2       2     0                           1007 mov  $0, flag
    0     2       2    -1                           1008 sub  $1, %bx
    0     2       2    -1                           1009 test $0, %bx
    0     2       2    -1                           1010 jgt .top
    0     2       2    -1                           1011 halt
```

3-a flag ile %bx kaydının değerini değiştirin (örneğin, yalnızca iki iş parçacığı çalıştırıyorsanız -a bx=2,bx=2). Kod ne yapar? Yukarıdaki soruya verdiğiniz cevabı nasıl değiştirir?

$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=2,bx=2

```
yigit@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=2,bx=2
ARG seed 0
ARG numthreads 2
ARG program flag.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched
ARG argv bx=2,bx=2
ARG load address 1000
ARG memsize 128
ARG memtrace flag,count
ARG regtrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False


 flag count      ax    bx          Thread 0                Thread 1

    0     0       0     2
    0     0       0     2   1000 mov  flag, %ax
    0     0       0     2   1001 test $0, %ax
    0     0       0     2   1002 jne  .acquire
    1     0       0     2   1003 mov  $1, flag
    1     0       0     2   1004 mov  count, %ax
    1     0       1     2   1005 add  $1, %ax
    1     1       1     2   1006 mov  %ax, count
    0     1       1     2   1007 mov  $0, flag
    0     1       1     1   1008 sub  $1, %bx
    0     1       1     1   1009 test $0, %bx
    0     1       1     1   1010 jgt .top
```

4-bx'i her iş parçacığı için yüksek bir değere ayarlayın ve ardından farklı kesme frekansları oluşturmak için -i bayrağını kullanın; Hangi değerler kötü sonuçlara yol açar? Hangisi iyi sonuçlara yol açar?

```
// bad outcomes
$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=10,bx=10 -i 1-10,12,13,14,17

// god outcomes
$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=10,bx=10 -i 11,15,16
```

```
rigit@ubuntu:~/ostep-homework/threads-locks$  ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=10,bx=10 -i 11
ARG seed 0
ARG numthreads 2
ARG program flag.s
ARG interrupt frequency 11
ARG interrupt randomness False
ARG procsched
ARG argv bx=10,bx=10
ARG load address 1000
ARG memsize 128
ARG memtrace flag,count
ARG regtrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False


 flag count      ax    bx          Thread 0                Thread 1

    0     0       0    10
    0     0       0    10   1000 mov  flag, %ax
    0     0       0    10   1001 test $0, %ax
    0     0       0    10   1002 jne  .acquire
    1     0       0    10   1003 mov  $1, flag
    1     0       0    10   1004 mov  count, %ax
    1     0       1    10   1005 add  $1, %ax
    1     1       1    10   1006 mov  %ax, count
    0     1       1    10   1007 mov  $0, flag
    0     1       1     9   1008 sub  $1, %bx
    0     1       1     9   1009 test $0, %bx
    0     1       1     9   1010 jgt .top
    0     1       0    10   ------ Interrupt ------  ------ Interrupt ------
    0     1       0    10                           1000 mov  flag, %ax
    0     1       0    10                           1001 test $0, %ax
    0     1       0    10                           1002 jne  .acquire
    1     1       0    10                           1003 mov  $1, flag
    1     1       1    10                           1004 mov  count, %ax
    1     1       2    10                           1005 add  $1, %ax
    1     2       2    10                           1006 mov  %ax, count
    0     2       2    10                           1007 mov  $0, flag
    0     2       2     9                           1008 sub  $1, %bx
    0     2       2     9                           1009 test $0, %bx
    0     2       2     9                           1010 jgt .top
    0     2       1     9   ------ Interrupt ------  ------ Interrupt ------
    0     2       0     9   1000 mov  flag, %ax
    0     2       0     9   1001 test $0, %ax
    0     2       0     9   1002 jne  .acquire
    1     2       0     9   1003 mov  $1, flag
    1     2       2     9   1004 mov  count, %ax
    1     2       3     9   1005 add  $1, %ax
    1     3       3     9   1006 mov  %ax, count
    0     3       3     9   1007 mov  $0, flag
    0     3       3     8   1008 sub  $1, %bx
    0     3       3     8   1009 test $0, %bx
    0     3       3     8   1010 jgt .top
    0     3       2     9   ------ Interrupt ------  ------ Interrupt ------
    0     3       0     9                           1000 mov  flag, %ax
    0     3       0     9                           1001 test $0, %ax
    0     3       0     9                           1002 jne  .acquire
```

5-Şimdi test-and-set.s programına bakalım. Öncelikle, basit bir kilitleme ilkel oluşturmak için xchg komutunu kullanan kodu anlamaya çalışın. Lock kazanımı nasıl yazılır? Lock yayınına ne dersiniz?

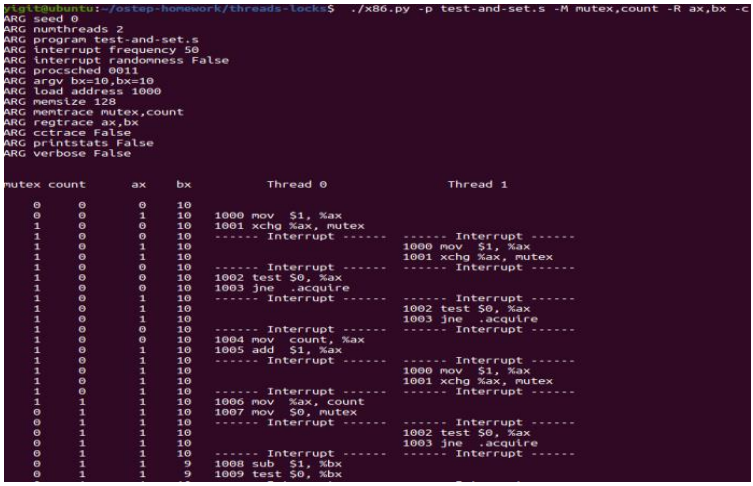
./x86.py -p test-and-set.s

```
yigit@ubuntu:~/ostep-homework/threads-locks$  ./x86.py -p test-and-set.s
ARG seed 0
ARG numthreads 2
ARG program test-and-set.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace
ARG regtrace
ARG cctrace False
ARG printstats False
ARG verbose False


       Thread 0                Thread 1

1000 mov  $1, %ax
1001 xchg %ax, mutex
1002 test $0, %ax
1003 jne  .acquire
1004 mov  count, %ax
1005 add  $1, %ax
1006 mov  %ax, count
1007 mov  $0, mutex
1008 sub  $1, %bx
1009 test $0, %bx
1010 jgt .top
1011 halt
----- Halt;Switch -----  ----- Halt;Switch -----
                        1000 mov  $1, %ax
                        1001 xchg %ax, mutex
                        1002 test $0, %ax
                        1003 jne  .acquire
                        1004 mov  count, %ax
                        1005 add  $1, %ax
                        1006 mov  %ax, count
                        1007 mov  $0, mutex
                        1008 sub  $1, %bx
                        1009 test $0, %bx
                        1010 jgt .top
                        1011 halt
```

6-Şimdi kesme aralığının (-i) değerini tekrar değiştirerek ve birkaç kez döngü yaptığınızdan emin olarak kodu çalıştırın. Kod her zaman beklendiği gibi çalışıyor mu? Bazen CPU'nun verimsiz kullanımına yol açıyor mu? Bunu nasıl ölçebilirsin?

7- Kilitleme kodunun belirli testlerini oluşturmak için -P flag'yi kullanın. Örneğin, ilk iş parçacığında lock'yi yakalayan, ancak ikincisinde onu almaya çalışan bir zamanlama çalıştırın. Doğru olan olur mu? Başka ne test etmelisin?

$ ./x86.py -p test-and-set.s -M mutex,count -R ax,bx -c -a bx=10,bx=10 -P 0011



8-Şimdi Peterson'ın algoritmasını uygulayan (metindeki bir kenar çubuğunda bahsedilen) peterson.s'deki koda bakalım. Kodu inceleyin ve anlamlandırıp anlamlandıramayacağınıza bakın.

9-Şimdi farklı -i değerleri ile kodu çalıştırın. Ne tür farklı davranışlar görüyorsunuz? Kodun varsaydığı gibi, iş parçacığı kimliklerini uygun şekilde (örneğin -a bx=0,bx=1 kullanarak) ayarladığınızdan emin olun.

$ ./x86.py -p peterson.s -M count,flag,turn -R ax,cx -a bx=0,bx=1 -c -i 2

```
yigit@ubuntu:~/ostep-homework/threads-locks$  ./x86.py -p peterson.s -M count,flag,turn -R ax,cx -a bx=0,bx=1 -c -i 2
ARG seed 0
ARG numthreads 2
ARG program peterson.s
ARG interrupt frequency 2
ARG interrupt randomness False
ARG procsched
ARG argv bx=0,bx=1
ARG load address 1000
ARG memsize 128
ARG memtrace count,flag,turn
ARG regtrace ax,cx
ARG cctrace False
ARG printstats False
ARG verbose False


count  flag  turn      ax    cx          Thread 0               Thread 1

    0     0     0       0     0
    0     0     0       0     0   1000 lea flag, %fx
    0     0     0       0     0   1001 mov %bx, %cx
    0     0     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    0     0     0       0     0                           1000 lea flag, %fx
    0     0     0       0     1                           1001 mov %bx, %cx
    0     0     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    0     0     0       0     0   1002 neg %cx
    0     0     0       0     1   1003 add $1, %cx
    0     0     0       0     1   ------ Interrupt ------  ------ Interrupt ------
    0     0     0       0    -1                           1002 neg %cx
    0     0     0       0     0                           1003 add $1, %cx
    0     0     0       0     1   ------ Interrupt ------  ------ Interrupt ------
    0     1     0       0     1   1004 mov $1, 0(%fx,%bx,4)
    0     1     1       0     1   1005 mov %cx, turn
    0     1     1       0     0   ------ Interrupt ------  ------ Interrupt ------
    0     1     1       0     0                           1004 mov $1, 0(%fx,%bx,4)
    0     1     0       0     0                           1005 mov %cx, turn
    0     1     0       0     1   ------ Interrupt ------  ------ Interrupt ------
    0     1     0       1     1   1006 mov 0(%fx,%cx,4), %ax
    0     1     0       1     1   1007 test $1, %ax
    0     1     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    0     1     0       1     0                           1006 mov 0(%fx,%cx,4), %ax
    0     1     0       1     0                           1007 test $1, %ax
    0     1     0       1     1   ------ Interrupt ------  ------ Interrupt ------
    0     1     0       1     1   1008 jne .fini
    0     1     0       0     1   1009 mov turn, %ax
```

10-Kodun çalıştığını "kanıtlamak" için zamanlamayı (-P flag ile) kontrol edebilir misiniz? Beklemede göstermeniz gereken farklı durumlar nelerdir? Karşılıklı dışlama ve kilitlenmeden kaçınmayı düşünün.

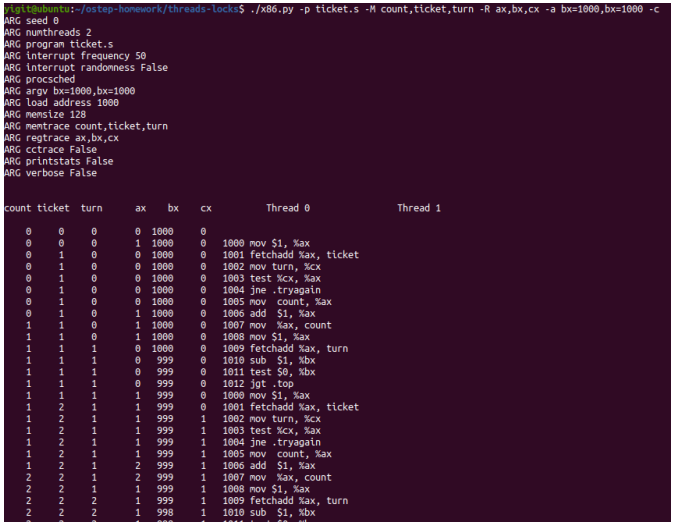
$ ./x86.py -p peterson.s -M count,flag,turn -R ax,cx -a bx=0,bx=1 -c -P 0000011111

```
yigit@ubuntu:~/ostep-homework/threads-locks$  ./x86.py -p peterson.s -M count,flag,turn -R ax,cx -a bx=0,bx=1 -c -P 0000011111
ARG seed 0
ARG numthreads 2
ARG program peterson.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched 0000011111
ARG argv bx=0,bx=1
ARG load address 1000
ARG memsize 128
ARG memtrace count,flag,turn
ARG regtrace ax,cx
ARG cctrace False
ARG printstats False
ARG verbose False


count  flag  turn      ax    cx          Thread 0               Thread 1

    0     0     0       0     0
    0     0     0       0     0   1000 lea flag, %fx
    0     0     0       0     0   1001 mov %bx, %cx
    0     0     0       0     0   1002 neg %cx
    0     0     0       0     1   1003 add $1, %cx
    0     1     0       0     1   1004 mov $1, 0(%fx,%bx,4)
    0     1     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    0     1     0       0     0                           1000 lea flag, %fx
    0     1     0       0     1                           1001 mov %bx, %cx
    0     1     0       0    -1                           1002 neg %cx
    0     1     0       0     0                           1003 add $1, %cx
    0     1     0       0     0                           1004 mov $1, 0(%fx,%bx,4)
    0     1     0       0     1   ------ Interrupt ------  ------ Interrupt ------
```

11- Şimdi ticket.s'deki lock biletinin kodunu inceleyin. Bölümdeki kodla eşleşiyor mu? Ardından şu flag'lerle çalıştırın: -a bx=1000,bx=1000 (her iş parçacığının kritik bölüm boyunca 1000 kez dönmesine neden olur). Ne olduğunu izle; iplikler lock'yi döndürmek için çok fazla zaman harcıyor

mu?

```
$ ./x86.py -p ticket.s -M count,ticket,turn -R ax,bx,cx -a bx=1000,bx=1000 -c
```



```
git@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p ticket.s -M count,ticket,turn -R ax,bx,cx -a bx=1000,bx=1000 -c
ARG seed 0
ARG numthreads 2
ARG program ticket.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched
ARG argv bx=1000,bx=1000
ARG load address 1000
ARG memsize 128
ARG memtrace count,ticket,turn
ARG regtrace ax,bx,cx
ARG cctrace False
ARG printstats False
ARG verbose False


count ticket  turn      ax    bx    cx          Thread 0              Thread 1

    0     0     0       0  1000     0
    0     0     0       1  1000     0   1000 mov $1, %ax
    0     1     0       0  1000     0   1001 fetchadd %ax, ticket
    0     1     0       0  1000     0   1002 mov turn, %cx
    0     1     0       0  1000     0   1003 test %cx, %ax
    0     1     0       0  1000     0   1004 jne .tryagain
    0     1     0       0  1000     0   1005 mov  count, %ax
    0     1     0       1  1000     0   1006 add  $1, %ax
    1     1     0       1  1000     0   1007 mov  %ax, count
    1     1     0       1  1000     0   1008 mov $1, %ax
    1     1     1       0  1000     0   1009 fetchadd %ax, turn
    1     1     1       0   999     0   1010 sub  $1, %bx
    1     1     1       0   999     0   1011 test $0, %bx
    1     1     1       0   999     0   1012 jgt .top
    1     1     1       1   999     0   1000 mov $1, %ax
    1     2     1       1   999     0   1001 fetchadd %ax, ticket
    1     2     1       1   999     1   1002 mov turn, %cx
    1     2     1       1   999     1   1003 test %cx, %ax
    1     2     1       1   999     1   1004 jne .tryagain
    1     2     1       1   999     1   1005 mov  count, %ax
    1     2     1       2   999     1   1006 add  $1, %ax
    2     2     1       2   999     1   1007 mov  %ax, count
    2     2     1       1   999     1   1008 mov $1, %ax
    2     2     2       1   999     1   1009 fetchadd %ax, turn
    2     2     2       1   998     1   1010 sub  $1, %bx
```

- 12-Siz daha fazla iş parçacığı ekledikçe kod nasıl davranır?

```
$ ./x86.py -p ticket.s -M count -t 10 -c -i 5
```

```
rkgit@ubuntu:~/ostep-homework/threads-locks$  ./x86.py -p ticket.s -M count -t 10 -c -i 5
ARG seed 0
ARG numthreads 10
ARG program ticket.s
ARG interrupt frequency 5
ARG interrupt randomness False
ARG procsched
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace count
ARG regtrace
ARG cctrace False
ARG printstats False
ARG verbose False


count         Thread 0              Thread 1              Thread 2              Thread 3              Thread 4

    0
    0   1000 mov $1, %ax
    0   1001 fetchadd %ax, ticket
    0   1002 mov turn, %cx
    0   1003 test %cx, %ax
    0   1004 jne .tryagain
    0   ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------
    0                           1000 mov $1, %ax
    0                           1001 fetchadd %ax, ticket
    0                           1002 mov turn, %cx
    0                           1003 test %cx, %ax
    0                           1004 jne .tryagain
    0   ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------
    0                                                     1000 mov $1, %ax
    0                                                     1001 fetchadd %ax, ticket
    0                                                     1002 mov turn, %cx
    0                                                     1003 test %cx, %ax
    0                                                     1004 jne .tryagain
    0   ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------
    0                                                                           1000 mov $1, %ax
    0                                                                           1001 fetchadd %ax, ticket
    0                                                                           1002 mov turn, %cx
    0                                                                           1003 test %cx, %ax
    0                                                                           1004 jne .tryagain
    0   ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------
    0
```

13- Şimdi, bir verim komutunun bir iş parçacığının CPU'nun kontrolünü sağlamasına izin verdiği verim.s'yi inceleyin (gerçekçi olarak, bu bir işletim sistemi ilkel olacaktır, ancak basit olması için, görevi bir talimatın yerine getirdiğini varsayıyoruz). Test-ve-set.s'nin dönme döngülerini boşa harcadığı, ancak verim.s'nin boşa harcamadığı bir senaryo bulun. Kaç talimat kaydedilir? Bu tasarruflar hangi senaryolarda ortaya çıkıyor?

./x86.py -p test-and-set.s -M count,mutex -R ax,bx -a bx=5,bx=5 -c -i 7

```
ogt1@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p test-and-set.s -M count,mutex -R ax,bx -a bx=5,bx=5 -c -i 7
ARG seed 0
ARG numthreads 2
ARG program test-and-set.s
ARG interrupt frequency 7
ARG interrupt randomness False
ARG procsched
ARG argv bx=5,bx=5
ARG load address 1000
ARG memsize 128
ARG memtrace count,mutex
ARG regtrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False


count mutex      ax    bx          Thread 0                Thread 1

    0     0       0     5
    0     0       1     5   1000 mov  $1, %ax
    0     1       0     5   1001 xchg %ax, mutex
    0     1       0     5   1002 test $0, %ax
    0     1       0     5   1003 jne  .acquire
    0     1       0     5   1004 mov  count, %ax
    0     1       1     5   1005 add  $1, %ax
    1     1       1     5   1006 mov  %ax, count
    1     1       0     5   ------ Interrupt ------  ------ Interrupt ------
    1     1       1     5                           1000 mov  $1, %ax
    1     1       1     5                           1001 xchg %ax, mutex
    1     1       1     5                           1002 test $0, %ax
    1     1       1     5                           1003 jne  .acquire
    1     1       1     5                           1000 mov  $1, %ax
    1     1       1     5                           1001 xchg %ax, mutex
    1     1       1     5                           1002 test $0, %ax
    1     1       1     5   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     5   1007 mov  $0, mutex
    1     0       1     4   1008 sub  $1, %bx
    1     0       1     4   1009 test $0, %bx
    1     0       1     4   1010 jgt .top
    1     0       1     4   1000 mov  $1, %ax
    1     1       0     4   1001 xchg %ax, mutex
    1     1       0     4   1002 test $0, %ax
    1     1       1     5   ------ Interrupt ------  ------ Interrupt ------
```

$ ./x86.py -p yield.s -M count,mutex -R ax,bx -a bx=5,bx=5 -c -i 7

```
ogt1@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p yield.s -M count,mutex -R ax,bx -a bx=5,bx=5 -c -i 7
ARG seed 0
ARG numthreads 2
ARG program yield.s
ARG interrupt frequency 7
ARG interrupt randomness False
ARG procsched
ARG argv bx=5,bx=5
ARG load address 1000
ARG memsize 128
ARG memtrace count,mutex
ARG regtrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False


count mutex      ax    bx          Thread 0                Thread 1

    0     0       0     5
    0     0       1     5   1000 mov  $1, %ax
    0     1       0     5   1001 xchg %ax, mutex
    0     1       0     5   1002 test $0, %ax
    0     1       0     5   1003 je .acquire_done
    0     1       0     5   1006 mov  count, %ax
    0     1       1     5   1007 add  $1, %ax
    1     1       1     5   1008 mov  %ax, count
    1     1       0     5   ------ Interrupt ------  ------ Interrupt ------
    1     1       1     5                           1000 mov  $1, %ax
    1     1       1     5                           1001 xchg %ax, mutex
    1     1       1     5                           1002 test $0, %ax
    1     1       1     5                           1003 je .acquire_done
    1     1       1     5                           1004 yield
    1     1       1     5   ------ Interrupt ------  ------ Interrupt ------
```

14-Son olarak, test-ve-test-and-set.s'yi inceleyin. Bu lock ne işe yarar? Test-and-set.s ile karşılaştırıldığında ne tür tasarruflar sağlar?