28

Kilitler (Locks)

Girişten eşzamanlılığa kadar, eşzamanlı programlamadaki temel sorunlardan birini gördük: atomik olarak bir dizi talimat uygulamak istiyoruz, ancak tek bir işlemcide (veya aynı anda birden fazla işlemcide yürütülen birden fazla iş parçacığında kesintilerin olması nedeniyle) yapamadık. Bu bölümde, kilit olarak adlandırılan bir şeyin ortaya çıkmasıyla bu soruna doğrudan saldırıyoruz. Programcılar kaynak koduna kilitlerle açıklama ekler, bunları kritik bölümlerin etrafına koyarlar ve böylece bu tür kritik bölümlerin sanki günahkâr bir atomik talimatmış gibi yürütülmesini sağlarlar.

28.1 Kilitler: Temel Fikir (Locks: The Basic Idea)

Örnek olarak, kritik bölümümüzün, paylaşılan bir değişkenin kanonik güncellemesi gibi göründüğünü varsayalım:

```
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 ekliyoruz:

```
1  lock_t mutex; // some globally-allocated lock '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 **kilitsiz** 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 (Pthread Locks)

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; exits on failure
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 (Building A Lock)

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 (Evaluating Locks)

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** (**mutual exclusion**) sağlamak olan temel görevini yerine getirip getirmediğidir. Temel olarak kilit, birden fazla iş parçacığının kritik bir bölüme girmesini önleyerek çalışıyor mu?

İkincisi **adalettir** (**fairness**). 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** (**starve**) ve bu nedenle onu asla elde edemez mi?

Son kriter **performanstır** (performance), 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 (Controlling Interrupts)

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 muhtemelen 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.

```
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** (**First Attempt: A Simple Flag**)

Bu nedenlerden dolayı, kesintileri kapatmak yalnızca sınırlı bağlamlarda karşılıklı dışlama ilkel olarak kullanılır. Örneğin, bazı durumlarda, bir işletim sisteminin kendisi, kendi veri yapılarına erişirken atomikliği garanti etmek veya en azından belirli dağınık kesme işleme durumlarının ortaya çıkmasını önlemek için kesme maskelemesini kullanır. Bu kullanım, bir şekilde ayrıcalıklı işlemleri gerçekleştirmek için kendisine her zaman güvenen işletim sistemi içinde güven sorunu ortadan kalktığı için mantıklıdır.

28.6 Başarısız Bir Deneme: Yalnızca Yükleri/Mağazaları Kullanmak (A Failed Attempt: Just Using Loads/Stores)

Kesintiye dayalı tekniklerin ötesine geçmek için, uygun bir kilit oluşturmak için CPU donanımına ve onun bize sağladığı talimatlara güvenmemiz gerekecek. Önce tek bir bayrak değişkeni kullanarak basit bir kilit oluşturmaya çalışalım. Bu başarısız girişimde, bir kilit oluşturmak için gereken bazı temel fikirleri göreceğiz ve (umarız) sadece tek bir değişken kullanmanın ve ona normal yükler ve depolar aracılığıyla erişmenin neden yetersiz olduğunu göreceğiz.

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** (**tests**) (bu durumda değildir) ve ardından iş parçacığının artık kilidi **tuttuğunu** (**holds**) belirtmek için bayrağı 1'e **ayarlayan** (**sets**) 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. Şekil 28.2'deki kodun araya girdiğini hayal edin; Başlamak için flag=0 varsayın.

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

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 Test Et ve Ayarla Çalışan Döndürme Kilitleri Oluşturma (Building Working Spin Locks with Test-And-Set)

Kesintileri devre dışı bırakmak birden çok işlemcide çalışmadığından ve yükleri ve depoları kullanan basit yaklaşımlar (yukarıda gösterildiği gibi) işe yaramadığından, 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.

Anlaşılması gereken en basit donanım desteği parçası, **test et ve ayarla** (**test-and-set**) (veya **atomik değişim**[1] (**atomic exchange**[1])) talimatı olarak bilinir. Test et ve ayarla komutunun ne yaptığını aşağıdaki C kod parçacığı aracılığıyla tanımlarız:

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

KENARA: **DEKKER VE PETERSON'UN ALGORİTMALARI**
(DEKKER'S AND PETERSON'S ALGORITHMS)

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 kez daha, sadece yükler ve depolar kullanılır ve buradaki fikir, iki iş parçacığının asla aynı anda kritik bir bölüme girmemesini sağlamaktır. İşte Peterson'ın algoritması (iki iş parçacığı için); anlayabiliyor musun bir bak.

```
  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 revaçta oldu ve teori tiplerine 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 (gevşek bellek tutarlılığı modelleri nedeniyle), bu nedenle onları eskisinden daha az kullanışlı hale getirir. Yine de tarihin çöplüğüne atılan daha fazla araştırma...

```
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: **Test Et ve Ayarla Basit Bir Döndürme Kilidi**
(A Simple Spin Lock Using Test-and-set)

Test et ve ayarla komutunun yaptığı şey aşağıdaki gibidir. Eski 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; Şimdi Şekil 28.3'te incelediğimiz gibi, bu biraz daha güçlü komut basit bir **döndürmeli kilit** (**spin lock**)oluşturmak için yeterlidir. Ya da daha iyisi: önce kendin çöz!

Bu kilidin neden çalıştığını anladığımızdan emin olalım. Önce bir iş parçacığının lock() işlevini çağırdığı ve başka hiçbir iş parçacığının şu anda kilidi tutmadığı bir durumu hayal edin; bu nedenle bayrak 0 olmalıdır. İş parçacığı TestAndSet(flag, 1) öğesini çağırdığında, yordam bayrağın eski değeri olan 0'ı döndürür; bu nedenle, bayrağın değerini test eden çağıran iş parçacığı, while döngüsünde dönerken yakalanmayacak ve kilidi alacaktır. İş parçacığı ayrıca değeri atomik olarak 1'e ayarlayacaktır, böylece kilidin artık tutulduğunu gösterir. İş parçacığı kritik bölümüyle bittiğinde, bayrağı sıfıra ayarlamak için unlock() öğesini çağırır.

Hayal edebileceğimiz ikinci durum, bir iş parçacığının kilidi zaten tuttuğunda ortaya çıkar (yani bayrak 1'dir). Bu durumda, bu iş parçacığı lock() öğesini çağıracak ve ardından TestAndSet(flag, 1) öğesini de çağıracaktır. Bu sefer, TestAndSet() aynı anda tekrar 1'e ayarlarken (kilit tutulduğu için) bayraktaki 1 olan eski değeri döndürür. Kilit başka bir iş parçacığı tarafından tutulduğu sürece, TestAndSet() art arda 1 döndürür ve böylece bu iş parçacığı 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 (eski kilit değerinin) **testini** (**test**) hem de (yeni değerin) **setini** (**set**) tek bir atomik işlem yaparak, yalnızca bir iş parçacığının kilidi almasını sağlıyoruz. Çalışan bir karşılıklı dışlama ilkesi bu şekilde oluşturulur!

İPUCU: EŞ ZAMANLILIK HAKKINDA KÖTÜ BİR PLANLAYICI OLARAK DÜŞÜNÜN
Bu örnekten, eşzamanlı yürütmeyi anlamak için izlemeniz gereken yaklaşım hakkında bir fikir edinebilirsiniz. Yapmaya çalışmanız gereken şey, senkronizasyon ilkellerini oluşturmaya yönelik zayıf girişimlerini boşa çıkarmak için dizileri en uygun olmayan zamanlarda kesen **kötü niyetli bir programlayıcıymışsınız** (**malicious scheduler**) gibi davranmaktır. Ne kadar kötü bir planlayıcısın! Kesintilerin tam sırası pek olası olmasa da, bu mümkündür ve belirli bir yaklaşımın işe yaramadığını göstermek için ihtiyacımız olan tek şey budur. Kötü niyetli düşünmek faydalı olabilir! (en azından bazen)

Artık bu tür bir kilide neden genellikle **döndürmeli kilit** (spin lock) dendiğini de anlayabilirsiniz. Oluşturulması en basit kilit türüdür ve kilit kullanılabilir hale gelene kadar CPU döngülerini kullanarak basitçe döner. Tek bir işlemci üzerinde düzgün çalışması için, **önleyici bir programlayıcı** (**preemptive scheduler**) 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 (Evaluating Spin Locks)

Temel döndürme kilidimiz göz önüne alındığında, daha önce açıklanan eksenlerimiz boyunca ne kadar etkili olduğunu şimdi değerlendirebiliriz. Bir kilidin en önemli yönü **doğruluktur** (correctness): 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 **adalettir** (**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).

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

Figure 28.4: **Compare-and-swap**

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. 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 28.4'te bulunur.

Temel fikir, ptr tarafından belirtilen adresteki değerin beklenen değere eşit olup olmadığını test etmek için karşılaştır ve değiştir; öyleyse, ptr ile işaret edilen hafıza konumunu yeni değerle güncelleyin. Değilse, hiçbir şey yapmayın. Her iki durumda da, orijinal değeri o bellek konumuna döndürün, böylece karşılaştırma ve takas kodunu çağıran kodun başarılı olup olmadığını bilmesine izin verin.

Karşılaştır ve değiştir talimatıyla, test et ve ayarla 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; basitçe bayrağın 0 olup olmadığını kontrol eder ve eğer öyleyse, atomik olarak 1'de yer değiştirir ve böylece kilidi elde eder. Kilit tutulurken kilidi almaya çalışan iplikler, kilit nihayet serbest bırakılana kadar dönmeye devam edecek.

Karşılaştır ve değiştir özelliğinin C tarafından çağrılabilir bir x86 sürümünü gerçekten nasıl yapacağınızı görmek istiyorsanız, kod dizisi ([S05]'ten) yararlı olabilir[2].

Son olarak, sezmiş olabileceğiniz gibi, karşılaştır ve değiştir, test et ve ayarla'dan daha güçlü bir talimattır. Gelecekte kilitsiz senkronizasyon [H91] gibi konulara kısaca değindiğimizde bu gücü biraz kullanacağız. Bununla birlikte, onunla basit bir döndürmeli kilit oluşturursak, davranışı yukarıda analiz ettiğimiz döndürmeli kilitle aynıdır.

28.10 Yük Bağlantılı ve Mağaza Koşullu (Load-Linked and Store-Conditional)

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 28.5'te bulunan gibidir. Alpha, PowerPC ve ARM benzer talimatlar sağlar[W09].

Yük bağlantılı, tipik bir yükleme talimatı gibi çalışır ve basitçe bellekten bir değer alır ve onu bir kayda yerleştirir. En önemli fark, yalnızca adrese araya giren bir depolama gerçekleşmemişse başarılı olan (ve az önce yük bağlantısının yapıldığı adreste depolanan değeri güncelleyen) mağaza koşuluyla gelir. Başarı durumunda, saklama 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 28.6'dadır.

Lock() kodu, tek ilginç parçadır. İlk olarak, bayrağın 0'a ayarlanmasını bekleyen bir iplik döner (ve böylece kilidin tutulmadığını gösterir). Bunu yaptıktan sonra, iş parçacığı, mağaza koşullu aracılığıyla kilidi elde etmeye çalışır; başarılı olursa, iş parçacığı atomik olarak bayrağın değerini 1 olarak değiştirir ve böylece kritik bölüme geçebilir.

Mağaza koşulunun başarısızlığının nasıl ortaya çıkabileceğine dikkat edin. Bir iş parçacığı, lock() öğesini çağırır ve kilit tutulmadığı için 0 döndürerek yük bağlantılı işlemi yürütür. Koşullu saklama girişiminde bulunmadan önce, kesilir ve başka bir iş parçacığı kilit kodunu girerek, aynı zamanda yük bağlantılı talimatı yürütür,

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

Figure 28.5: **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 28.6: **Using LL/SC To Build A Lock**

ve ayrıca 0 almak ve devam etmek. Bu noktada, iki iş parçacığının her biri yük bağlantılı işlemi gerçekleştirmiştir ve her biri depolama koşullu işlemini denemek üzeredir. Bu talimatların temel özelliği, bu iş parçacıklarından yalnızca birinin bayrağı 1'e güncellemeyi başarması ve böylece kilidi almasıdır; 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) ||
3             !StoreC  onditional(&lock->flag, 1))
4          ; // spin
5  }
```

28.1 Fetch-And-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  }
```

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** (**Lauer's Law**) 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.

Bu örnekte, Mellor-Crummey ve Scott [MS91] tarafından tanıtıldığı gibi, daha ilginç bir **bilet kilidi** (ticket lock) oluşturmak için getir ve ekle yöntemini kullanacağız. Kilitleme ve kilit açma kodu Şekil 28.7'de (sayfa 14) bulunur.

Bu çözüm, tek bir değer yerine, bir kilit oluşturmak için bilet ve dönüş değişkenini birlikte kullanır. Temel işlem oldukça basittir: Bir iş parçacığı bir kilit 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 `kilit->dönüş` daha sonra hangi iş parçacığının sırası olduğunu belirlemek için kullanılır; belirli bir iş parçacığı için (`myturn == dönüş`), kritik bölüme girme sırası o iş parçacığınındır. Kilit 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 programlanacaktır (öndekiler kritik bölümü geçip kilidi açtıktan sonra). Daha önceki denemelerimizde böyle bir garanti yoktu; test ve ayarla (örneğin) üzerinde dönen bir iş parçacığı, diğer iş parçacıkları kilidi alıp serbest bıraksa bile sonsuza kadar dönebilir.

28.2 Çok Fazla Döndürme: Şimdi Ne Olacak? (Too Much Spinning: What Now?)

Basit donanım tabanlı kilitlerimiz 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 (bir dahaki sefere çalıştığında, söyle)

```
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       lock->turn = lock->turn + 1;
19   }
```

Figure 28.7: **Ticket Locks**

iplik 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 kilidi açması beklenir. Ve böylece, bir sonraki sorunumuz:

EN ÖNEMLİ NOKTA: DÖNÜŞTEN NASIL KAÇINILIR
CPU üzerinde dönerek gereksiz yere zaman kaybetmeyen bir kilidi nasıl geliştirebiliriz?

Donanım desteği tek başına sorunu çözemez. Bunun için OF desteğine ihtiyacımız olacak! Şimdi bunun nasıl işe yarayabileceğini anlayalım.

28.3 A Simple Approach: Just Yield, Baby

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, kesintiye uğramış (kilit tutma) iş parçacığının tekrar çalıştırılmasını bekleyerek sonsuz bir şekilde dönmeye başladığında ne yapmalıyız?

İlk denememiz basit ve arkadaşça bir yaklaşım: spin atacağınız zaman bunun yerine CPU'yu başka bir thread'e bırakın. Al Davis'in dediği gibi, "sadece boyun eğ bebeğim!" [D91]. Şekil 28.8 (sayfa 15) yaklaşımı göstermektedir.

```
1  void unlock() {
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 }
```

typedef struct __lock_t {

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

Bu yaklaşımda, bir iş parçacığının CPU'dan vazgeçmek ve başka bir iş parçacığının çalışmasına izin vermek istediğinde çağırabileceği bir işletim sistemi ilkel verim() varsayıyoruz. Bir iş parçacığı üç durumdan birinde olabilir (çalışıyor, hazır veya engellenmiş); verim, çağıranı çalışma durumundan hazır durumuna taşıyan ve böylece başka bir iş parçacığının çalışmasını destekleyen bir sistem çağrısıdır. Böylece, akan iş parçacığı esas olarak kendi kendini programdan çıkarır.

Bir CPU'da iki iş parçacığı olan örneği düşünün; bu durumda, verime dayalı yaklaşımımız oldukça iyi çalışıyor. Bir iş parçacığı, lock() öğesini çağırır ve tutulan bir kilit bulursa, yalnızca CPU'yu verir ve böylece diğer iş parçacığı çalışır ve kritik bölümünü tamamlar. Bu basit durumda, verimli yaklaşım iyi çalışır.

Şimdi bir kilit için art arda yarışan birçok iş parçacığının (diyelim ki 100) olduğu durumu ele alalım. Bu durumda, bir iş parçacığı kilidi alır ve serbest bırakmadan önce önlenirse, diğer 99'un her biri lock()'u çağırır, tutulan kilidi bulur ve CPU'yu verir. Bir tür döngüsel zamanlayıcı varsayarsak, 99'un her biri, kilidi tutan iş parçacığı tekrar çalışmaya başlamadan önce bu çalıştır ve ver modelini yürütecektir. Döndürme yaklaşımımızdan daha iyi olsa da (bu, döndürmede 99 zaman dilimini boşa harcar), bu yaklaşım yine de maliyetlidir; bir bağlam anahtarının maliyeti önemli olabilir ve bu nedenle çok fazla israf olur.

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.

28.4 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.

```
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
33                                    // (for next thread!)
34    m->guard = 0;
35 }
```

Figure 28.9: **Kuyruklar, Test Et ve Ayarla, Verim ve Uyandırma ile Kilitle**

Bu nedenle, mevcut tutucu kilidi serbest bıraktıktan sonra hangi iş parçacığının kilidi alacağı konusunda açıkça bir miktar kontrol uygulamalıyız. Bunu yapmak için, biraz daha fazla işletim sistemi desteğine ve kilidi almak için hangi iş parçacıklarının 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, tutulan bir kilidi almaya çalışırsa arayanı uyku moduna geçiren ve kilit serbest kaldığında onu uyandıran bir kilit oluşturmak için art arda kullanılabilir. Bu tür ilkellerin olası bir kullanımını anlamak için Şekil 28.9'daki koda bakalım.

YANİ: DÖNÜŞTEN KAÇINMAK İÇİN DAHA FAZLA NEDEN: ÖNCELİKLİ TERS ÇEVİRME

Döndürme kilitlerinden kaçınmanın iyi bir nedeni performanstır: ana metinde açıklandığı gibi, bir kilit tutulurken bir iş parçacığı kesintiye uğrarsa, döndürme kilitlerini kullanan diğer iş parçacıkları, kilidin kullanılabilir hale gelmesini beklemek için büyük miktarda CPU zamanı harcar. Ancak, bazı sistemlerde dönme kilitlerinden kaçınmak için başka ilginç bir neden daha olduğu ortaya çıktı: doğruluk. Dikkat edilmesi gereken sorun, Dünya [M15] ve Mars'ta [R97] meydana gelen, ne yazık ki galaksiler arası bir bela olan öncelik tersine çevirme olarak bilinir!

Bir sistemde iki iş parçacığı olduğunu varsayalım. İş Parçacığı 2'nin (T2) yüksek bir zamanlama önceliği vardır ve İş Parçacığı 1'in (T1) daha düşük önceliği vardır. Bu örnekte, eğer gerçekten ikisi de çalıştırılabilir ise, CPU zamanlayıcısının her zaman T2'yi T1 üzerinden çalıştıracağını varsayalım; T1, yalnızca T2 bunu yapamadığı zaman çalışır (örn. T2, G/Ç'de bloke edildiğinde).

Şimdi, sorun. T2'nin bir nedenden dolayı bloke olduğunu varsayalım. Böylece T1 çalışır, bir döndürme kilidi alır ve kritik bir bölüme girer. T2 artık engellenmemiş hale gelir (belki bir G/Ç tamamlandığından dolayı) ve CPU programlayıcı onu hemen programlar (böylece T1'in zamanlamasını kaldırır). T2 şimdi kilidi almaya çalışır ve başaramadığı için (T1 kilidi tutar), sadece dönmeye devam eder. Kilit bir döndürme kilidi olduğundan, T2 sonsuza kadar döner ve sistem askıda kalır.

Döndürme kilitlerinin kullanımından kaçınmak ne yazık ki ters çevirme sorununu ortadan kaldırmaz (ne yazık ki). T3'ün en yüksek önceliğe ve T1'in en düşük önceliğe sahip olduğu T1, T2 ve T3 olmak üzere üç iş parçacığını hayal edin. Şimdi T1'in bir kilit aldığını hayal edin. T3 daha sonra başlar ve T1'den daha yüksek önceliğe sahip olduğu için hemen çalışır (T1'i önleyerek). T3, T1'in tuttuğu kilidi almaya çalışır, ancak beklerken takılıp kalır çünkü T1 hala onu elinde tutar. T2 çalışmaya başlarsa, T1'den daha yüksek önceliğe sahip olacak ve bu nedenle çalışacaktır. T2'den daha yüksek önceliğe sahip olan T3, artık T2 çalıştığı için hiç çalışmayabilecek olan T1'i beklerken takılıp kaldı. Düşük seviyedeki T2 işlemciyi kontrol ederken kudretli T3'ün çalışamaması üzücü değil mi? Yüksek önceliğe sahip olmak eskisi gibi değil.

Bu örnekte birkaç ilginç şey yapıyoruz. İlk olarak, daha verimli bir kilit oluşturmak için eski test et ve ayarla fikrini açık bir kilit garsonları kuyruğuyla birleştiriyoruz. İkincisi, bir sonraki kilidi kimin alacağını kontrol etmeye yardımcı olmak ve böylece açlıktan kaçınmak için bir kuyruk kullanırız.

Koruyucunun nasıl kullanıldığını (Şekil 28.9, sayfa 16), temel olarak bayrağın etrafında dönen bir kilit ve kilidin kullandığı kuyruk manipülasyonları olarak fark edebilirsiniz. Dolayısıyla bu yaklaşım, spin-beklemeden tamamen kaçınmaz; kilidi alırken veya serbest bırakırken bir iş parçacığı kesintiye uğrayabilir ve bu nedenle

diğer iş parçacıklarının bunun tekrar çalışması için dönmesini beklemesine neden olabilir. Bununla birlikte, döndürmek

için harcanan zaman oldukça sınırlıdır (kullanıcı tanımlı kritik bölüm yerine kilitleme ve kilit açma kodunun içinde yalnızca birkaç talimat) ve bu nedenle bu yaklaşım makul olabilir.

Ayrıca, lock() içinde, bir thread kilidi alamadığında (zaten tutuluyor), kendimizi bir kuyruğa eklemeye dikkat ettiğimizi de gözlemleyebilirsiniz (gettid() fonksiyonunu çağırarak mevcut thread ID'sini almak için). iş parçacığı), korumayı 0'a ayarlayın ve CPU'yu verin. Okuyucu için bir soru: Koruma kilidinin serbest bırakılması parktan() sonra gelseydi ve daha önce olmasaydı ne olurdu? İpucu: kötü bir şey.

Ayrıca, başka bir iş parçacığı uyandığında bayrağın 0'a geri ayarlanmadığını da tespit edebilirsiniz. Bu neden? Eh, bu bir hata değil, bir zorunluluktur! Bir iş parçacığı uyandırıldığında, sanki park();'dan dönüyormuş gibi olacaktır; ancak, korumayı kodun o noktasında tutmaz ve bu nedenle bayrağı 1'e ayarlamaya bile çalışamaz. Böylece, kilidi doğrudan iş parçacığından kilidi serbest bırakarak onu alan bir sonraki iş parçacığına geçiririz; bayrak arada 0'a ayarlı değil.

Son olarak, park() çağrısından hemen önce çözümde algılanan yarış koşulunu fark edebilirsiniz. Sadece yanlış zamanlama ile, artık kilit tutulana kadar uyuması gerektiğini varsayarsak, bir iş parçacığı park etmek üzere olacaktır. O sırada başka bir iş parçacığına (örneğin, kilidi tutan bir iş parçacığına) geçiş, örneğin bu iş parçacığının kilidi serbest bırakması durumunda soruna yol açabilir. İlk iş parçacığının yanındaki sonraki park, daha sonra (potansiyel olarak) sonsuza kadar uyur, bu sorun bazen uyanma/bekleme yarışı olarak adlandırılır.

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.

28.5 Different OS, Different Support

Şimdiye kadar bir iş parçacığı kitaplığında daha verimli bir kilit oluşturmak için bir işletim sisteminin sağlayabileceği bir tür destek gördük. Diğer işletim sistemleri benzer destek sağlar; ayrıntılar değişir.

Örneğin, Linux, Solaris arayüzüne benzer, ancak daha fazla çekirdek içi işlevsellik sağlayan bir futex sağlar. Spesifik olarak, her bir futex, kendisiyle belirli bir fiziksel bellek konumunun yanı sıra bir

```
1  void mutex_lock (int *mutex) {
2    int v;
3    /* Bit 31 was clear, we got the mutex (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 waitFirst make sure the futex value
13          we are monitoring is truly negative (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 counter results in 0 if and
23      only if 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);
30 }
```

Figure 28.10: **Linux-based Futex Locks**

çekirdek başına kuyruk. Arayanlar, gerektiğinde uyumak ve uyanmak için futex çağrılarını (aşağıda açıklanmıştır) kullanabilir.

Özellikle, iki arama mevcuttur. futex wait(address,beklenen) çağrısı, adresteki değerin beklenen değere eşit olduğunu varsayarak çağıran iş parçacığını uyku moduna alır. Eşit değilse, arama hemen geri döner. Rutin futex uyandırma(adres) çağrısı, kuyrukta bekleyen bir iş parçacığını uyandırır. Bu çağrıların bir Linux muteksinde kullanımı Şekil 28.10'da (sayfa 19) gösterilmiştir.

nptl kitaplığındaki (gnu libc kitaplığının bir parçası) [L09] lowlevellock.h'den alınan bu kod pasajı birkaç nedenden dolayı ilgi çekicidir. İlk olarak, hem kilidin tutulup tutulmadığını (tamsayının yüksek biti) hem de kilitteki garsonların sayısını (diğer tüm bitler) izlemek için tek bir tamsayı kullanır. Böylece kilit negatif ise tutulur (çünkü yüksek bit set edilir ve o bit tamsayının işaretini belirler).

İkincisi, kod parçacığı ortak durum için nasıl optimize edileceğini gösterir,

özellikle kilit için bir çekişme olmadığında; bir kilidi alan ve serbest bırakan tek bir iş parçacığı ile çok az iş yapılır (kilitlemek için atomik bit test-ve-set ve kilidi serbest bırakmak için bir atomik ekleme).

Nasıl çalıştığını anlamak için bu "gerçek dünya" kilidinin geri kalanını çözüp çözemeyeceğinize bakın. Bunu yapın ve bir Linux kilitleme ustası olun ya da en azından bir kitap size bir şey yapmanızı söylediğinde sizi dinleyen biri olun [3].

28.6 İki Fazlı Kilitler

Son bir not: Linux yaklaşımı, en azından 1960'ların başlarında [M82] Dahm Locks'a kadar uzanan, yıllardır aralıklarla kullanılan eski bir yaklaşımın tadına sahiptir ve şimdi iki- faz kilidi. İki fazlı bir kilit, özellikle kilit serbest bırakılmak üzereyse döndürmenin faydalı olabileceğini fark eder. Böylece ilk aşamada kilit, kilidi elde edebileceğini umarak bir süre döner.

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 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.

28.7 Su

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 kod tabanlarına göz atın; büyüleyici bir okuma [L09, S09]. Ayrıca, modern çok işlemcili [D+13] üzerindeki kilitleme stratejilerinin bir karşılaştırması için David ve diğerlerinin mükemmel çalışmasına bakın.

[3] Like buy a print copy of OSTEP! Even though the book is available for free online, wouldn't you just love a hard cover for your desk? Or, better yet, ten copies to share with friends and family? And maybe one extra copy to throw at an enemy? (the book *is* heavy, and thus chucking it is surprisingly effective)

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 Wood- lands 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 semi- nal 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. Be- cause 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/MarsPathfinder/AuthoritativeAccount.html. *A descrip- tion 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, al- though 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 infor- mation there and it felt wrong not to. Further, it was useful, listing the instructions for the different ar- chitectures: ldll/stlc 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 In- ternational, 2000. http://www.sparc.org/standards/SPARCV9.pdf. *See developers. sun.com/solaris/articles/atomicsparc/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. Flag.s incele. Bu kod, tek bir bellek bayrağıyla kilitlemeyi "uygular". Assembly anlayabiliyor musun?

flag.s'ye inceledim. Evet, assembly dili anlayabiliyorum.
./x86.py -p flag.s

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p flag.s
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
ARG regtrace
ARG cctrace False
ARG printstats False
ARG verbose False


      Thread 0                Thread 1

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

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?

Varsayılanlarla çalıştırdığımda, flag.s düzgün çalışıyor. -M, -R ve -c bayraklarını kullanarak flag.s:

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

```
nazmul@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
```

tahmin ettiğim gibi değeri 0 ile bitiyor.

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?

bir iş parçacığı için
./x86.py -p flag.s -t 1 -a bx=2 -M flag,count -R ax,bx -c

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p flag.s -t 1 -a bx=2 -M flag,count -R ax,bx -c
ARG seed 0
ARG numthreads 1
ARG program flag.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched
ARG argv 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

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

iki iş parçacığı için
./x86.py -p flag.s -t 2 -a bx=2 -M flag,count -R ax,bx -c

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p flag.s -t 2 -a bx=2 -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 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
    0     1       0     1   1000 mov  flag, %ax
    0     1       0     1   1001 test $0, %ax
    0     1       0     1   1002 jne  .acquire
    1     1       0     1   1003 mov  $1, flag
    1     1       1     1   1004 mov  count, %ax
    1     1       2     1   1005 add  $1, %ax
    1     2       2     1   1006 mov  %ax, count
    0     2       2     1   1007 mov  $0, flag
    0     2       2     0   1008 sub  $1, %bx
    0     2       2     0   1009 test $0, %bx
    0     2       2     0   1010 jgt .top
    0     2       2     0   1011 halt
    0     2       0     2   ----- Halt;Switch -----  ----- Halt;Switch -----
    0     2       0     2                           1000 mov  flag, %ax
    0     2       0     2                           1001 test $0, %ax
    0     2       0     2                           1002 jne  .acquire
    1     2       0     2                           1003 mov  $1, flag
    1     2       2     2                           1004 mov  count, %ax
    1     2       3     2                           1005 add  $1, %ax
    1     3       3     2                           1006 mov  %ax, count
    0     3       3     2                           1007 mov  $0, flag
    0     3       3     1                           1008 sub  $1, %bx
    0     3       3     1                           1009 test $0, %bx
    0     3       3     1                           1010 jgt .top
    0     3       0     1                           1000 mov  flag, %ax
    0     3       0     1                           1001 test $0, %ax
    0     3       0     1                           1002 jne  .acquire
    1     3       0     1                           1003 mov  $1, flag
    1     3       3     1                           1004 mov  count, %ax
    1     3       4     1                           1005 add  $1, %ax
    1     4       4     1                           1006 mov  %ax, count
    0     4       4     1                           1007 mov  $0, flag
    0     4       4     0                           1008 sub  $1, %bx
    0     4       4     0                           1009 test $0, %bx
    0     4       4     0                           1010 jgt .top
    0     4       4     0                           1011 halt
```

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?

kötü sonuçlar
$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=15,bx=15 -i 1-10,12,13,14,17

iyi sonuçlar
$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=15,bx=15 -i 11,15,16

kötü sonuç örneği:
./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=15,bx=15 -i 1

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=15,bx=15 -i 1
ARG seed 0
ARG numthreads 2
ARG program flag.s
ARG interrupt frequency 1
ARG interrupt randomness False
ARG procsched
ARG argv bx=15,bx=15
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    15
    0     0       0    15   1000 mov  flag, %ax
    0     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    0     0       0    15                           1000 mov  flag, %ax
    0     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    0     0       0    15   1001 test $0, %ax
    0     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    0     0       0    15                           1001 test $0, %ax
    0     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    0     0       0    15   1002 jne  .acquire
    0     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    0     0       0    15                           1002 jne  .acquire
    0     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    1     0       0    15   1003 mov  $1, flag
    1     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    1     0       0    15                           1003 mov  $1, flag
    1     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    1     0       0    15   1004 mov  count, %ax
    1     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    1     0       0    15                           1004 mov  count, %ax
    1     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    1     0       1    15   1005 add  $1, %ax
    1     0       0    15   ------ Interrupt ------  ------ Interrupt ------
    1     0       1    15                           1005 add  $1, %ax
    1     0       1    15   ------ Interrupt ------  ------ Interrupt ------
    1     1       1    15   1006 mov  %ax, count
```

iyi sonuç örneği:
./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=15,bx=15 -i 11

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p flag.s -M flag,count -R ax,bx -c -a bx=15,bx=15 -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=15,bx=15
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    15
    0     0       0    15   1000 mov  flag, %ax
    0     0       0    15   1001 test $0, %ax
    0     0       0    15   1002 jne  .acquire
    1     0       0    15   1003 mov  $1, flag
    1     0       0    15   1004 mov  count, %ax
    1     0       1    15   1005 add  $1, %ax
    1     1       1    15   1006 mov  %ax, count
    0     1       1    15   1007 mov  $0, flag
    0     1       1    14   1008 sub  $1, %bx
    0     1       1    14   1009 test $0, %bx
    0     1       1    14   1010 jgt .top
    0     1       0    15   ------ Interrupt ------  ------ Interrupt ------
    0     1       0    15                           1000 mov  flag, %ax
    0     1       0    15                           1001 test $0, %ax
    0     1       0    15                           1002 jne  .acquire
    1     1       0    15                           1003 mov  $1, flag
    1     1       1    15                           1004 mov  count, %ax
    1     1       2    15                           1005 add  $1, %ax
    1     2       2    15                           1006 mov  %ax, count
    0     2       2    15                           1007 mov  $0, flag
    0     2       2    14                           1008 sub  $1, %bx
    0     2       2    14                           1009 test $0, %bx
    0     2       2    14                           1010 jgt .top
    0     2       1    14   ------ Interrupt ------  ------ Interrupt ------
```

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

```
nazmul@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?

./x86.py -p test-and-set.s -M mutex,count -R ax,bx -c -i 1

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


mutex count      ax    bx          Thread 0                Thread 1

    0     0       0     0
    0     0       1     0   1000 mov  $1, %ax
    0     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    0     0       1     0                           1000 mov  $1, %ax
    0     0       1     0   ------ Interrupt ------  ------ Interrupt ------
    1     0       0     0   1001 xchg %ax, mutex
    1     0       1     0   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     0                           1001 xchg %ax, mutex
    1     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    1     0       0     0   1002 test $0, %ax
    1     0       1     0   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     0                           1002 test $0, %ax
    1     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    1     0       0     0   1003 jne  .acquire
    1     0       1     0   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     0                           1003 jne  .acquire
    1     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    1     0       0     0   1004 mov  count, %ax
    1     0       1     0   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     0                           1000 mov  $1, %ax
    1     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     0   1005 add  $1, %ax
    1     0       1     0   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     0                           1001 xchg %ax, mutex
    1     0       1     0   ------ Interrupt ------  ------ Interrupt ------
    1     1       1     0   1006 mov  %ax, count
```

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

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


mutex count      ax    bx          Thread 0                Thread 1

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

./x86.py -p test-and-set.s -M mutex,count -R ax,bx -c -i 10

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


mutex count      ax    bx          Thread 0                Thread 1

    0     0       0     0
    0     0       1     0   1000 mov  $1, %ax
    1     0       0     0   1001 xchg %ax, mutex
    1     0       0     0   1002 test $0, %ax
    1     0       0     0   1003 jne  .acquire
    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, mutex
    0     1       1    -1   1008 sub  $1, %bx
    0     1       1    -1   1009 test $0, %bx
    0     1       0     0   ------ Interrupt ------  ------ Interrupt ------
    0     1       1     0                           1000 mov  $1, %ax
    1     1       0     0                           1001 xchg %ax, mutex
    1     1       0     0                           1002 test $0, %ax
    1     1       0     0                           1003 jne  .acquire
    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, mutex
    0     2       2    -1                           1008 sub  $1, %bx
    0     2       2    -1                           1009 test $0, %bx
    0     2       1    -1   ------ Interrupt ------  ------ Interrupt ------
    0     2       1    -1   1010 jgt .top
    0     2       1    -1   1011 halt
    0     2       2    -1   ----- Halt;Switch -----  ----- Halt;Switch -----
    0     2       2    -1                           1010 jgt .top
    0     2       2    -1                           1011 halt
```

Evet kod her zaman çalışır beklenildiği gibi. Mutex 0'dır.

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 101001

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p test-and-set.s -M mutex,count -R ax,bx -c -a bx=10,bx=10 -P 101001
ARG seed 0
ARG numthreads 2
ARG program test-and-set.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched 101001
ARG argv bx=10,bx=10
ARG load address 1000
ARG memsize 128
ARG memtrace mutex,count
ARG regtrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False


mutex count      ax    bx          Thread 0                Thread 1

    0     0       0    10
    0     0       1    10                             1000 mov  $1, %ax
    0     0       0    10   ------ Interrupt ------  ------ Interrupt ------
    0     0       1    10   1000 mov  $1, %ax
    0     0       1    10   ------ Interrupt ------  ------ Interrupt ------
    1     0       0    10                             1001 xchg %ax, mutex
    1     0       1    10   ------ Interrupt ------  ------ Interrupt ------
    1     0       1    10   1001 xchg %ax, mutex
    1     0       1    10   1002 test $0, %ax
    1     0       0    10   ------ Interrupt ------  ------ Interrupt ------
    1     0       0    10                             1002 test $0, %ax
    1     0       0    10                             1003 jne  .acquire
    1     0       1    10   ------ Interrupt ------  ------ Interrupt ------
    1     0       1    10   1003 jne  .acquire
    1     0       0    10   ------ Interrupt ------  ------ Interrupt ------
    1     0       0    10                             1004 mov  count, %ax
    1     0       1    10   ------ Interrupt ------  ------ Interrupt ------
    1     0       1    10   1000 mov  $1, %ax
    1     0       1    10   1001 xchg %ax, mutex
    1     0       0    10   ------ Interrupt ------  ------ Interrupt ------
    1     0       1    10                             1005 add  $1, %ax
    1     1       1    10                             1006 mov  %ax, count
    1     1       1    10   ------ Interrupt ------  ------ Interrupt ------
```

Evet kod sorunsuz çalışıyor.

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

./x86.py -p peterson.s

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p peterson.s
ARG seed 0
ARG numthreads 2
ARG program peterson.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 lea flag, %fx
1001 mov %bx, %cx
1002 neg %cx
1003 add $1, %cx
1004 mov $1, 0(%fx,%bx,4)
1005 mov %cx, turn
1006 mov 0(%fx,%cx,4), %ax
1007 test $1, %ax
1008 jne .fini
1012 mov count, %ax
1013 add $1, %ax
1014 mov %ax, count
1015 mov $0, 0(%fx,%bx,4)
1016 mov %cx, turn
1017 halt
----- Halt;Switch -----  ----- Halt;Switch -----
                         1000 lea flag, %fx
                         1001 mov %bx, %cx
                         1002 neg %cx
                         1003 add $1, %cx
                         1004 mov $1, 0(%fx,%bx,4)
                         1005 mov %cx, turn
                         1006 mov 0(%fx,%cx,4), %ax
                         1007 test $1, %ax
                         1008 jne .fini
                         1012 mov count, %ax
                         1013 add $1, %ax
                         1014 mov %ax, count
                         1015 mov $0, 0(%fx,%bx,4)
                         1016 mov %cx, turn
                         1017 halt
```

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 5

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p peterson.s -M count,flag,turn -R ax,cx -a bx=0,bx=1 -c -i 5
ARG seed 0
ARG numthreads 2
ARG program peterson.s
ARG interrupt frequency 5
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   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     1       0     1   ------ Interrupt ------  ------ Interrupt ------
    0     1     1       0     1   1005 mov %cx, turn
    0     1     1       1     1   1006 mov 0(%fx,%cx,4), %ax
    0     1     1       1     1   1007 test $1, %ax
    0     1     1       1     1   1008 jne .fini
    0     1     1       1     1   1009 mov turn, %ax
    0     1     1       0     0   ------ Interrupt ------  ------ Interrupt ------
    0     1     0       0     0                           1005 mov %cx, turn
    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     0                           1008 jne .fini
    0     1     0       0     0                           1009 mov turn, %ax
    0     1     0       1     1   ------ Interrupt ------  ------ Interrupt ------
    0     1     0       1     1   1010 test %cx, %ax
    0     1     0       1     1   1011 je .spin1
    0     1     0       1     1   1006 mov 0(%fx,%cx,4), %ax
    0     1     0       1     1   1007 test $1, %ax
    0     1     0       1     1   1008 jne .fini
```

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 11000

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p peterson.s -M count,flag,turn -R ax,cx -a bx=0,bx=1 -c -P 11000
ARG seed 0
ARG numthreads 2
ARG program peterson.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched 11000
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     1                           1001 mov %bx, %cx
    0     0     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    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   ------ Interrupt ------  ------ Interrupt ------
    0     0     0       0    -1                           1002 neg %cx
    0     0     0       0     0                           1003 add $1, %cx
    0     0     0       0     0   ------ Interrupt ------  ------ Interrupt ------
    0     0     0       0     1   1003 add $1, %cx
    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       0     1   1006 mov 0(%fx,%cx,4), %ax
    0     1     0       1     1   1007 test $1, %ax
    0     1     0       1     1   1008 jne .fini
    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       0     1   1009 mov turn, %ax
    0     1     0       0     1   1010 test %cx, %ax
    0     1     0       0     1   1011 je .spin1
    0     1     0       1     0   ------ Interrupt ------  ------ Interrupt ------
    0     1     0       1     0                           1008 jne .fini
    0     1     0       0     0                           1009 mov turn, %ax
    0     1     0       0     1   ------ Interrupt ------  ------ Interrupt ------
    0     1     0       0     1   1012 mov count, %ax
    0     1     0       1     1   1013 add $1, %ax
```

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
Count is correct. Yes.

```
kazmul@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
    1     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
    2     2     2       1   998     1   1011 test $0, %bx
    2     2     2       1   998     1   1012 jgt .top
    2     2     2       1   998     1   1000 mov $1, %ax
    2     3     2       2   998     1   1001 fetchadd %ax, ticket
    2     3     2       2   998     2   1002 mov turn, %cx
    2     3     2       2   998     2   1003 test %cx, %ax
    2     3     2       2   998     2   1004 jne .tryagain
    2     3     2       3   998     2   1005 mov  count, %ax
    2     3     2       3   998     2   1006 add  $1, %ax
```

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

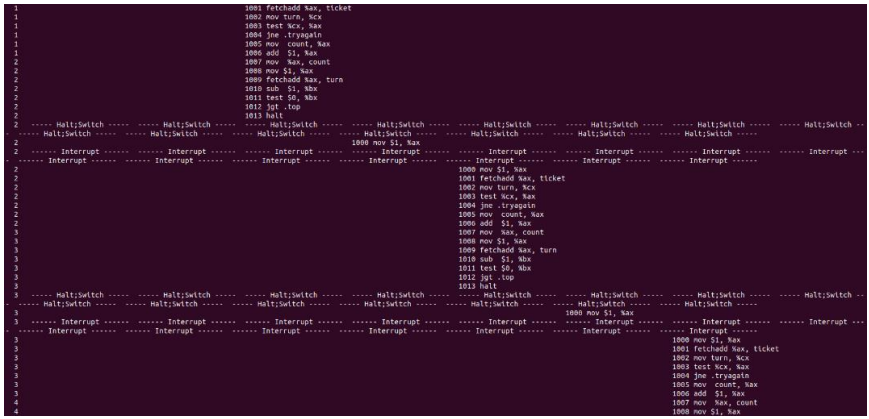
Burada biz 15 iş parçacığı ekledik, Thread 0'dan Thread 14'e kadar. Farklı iş parçacıkları için farklı farklı değerler gösteriyor.

./x86.py -p ticket.s -M count -t 15 -c -i 15

```
kazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p ticket.s -M count -t 15 -c -i 15
\ARG seed 0
ARG numthreads 15
ARG program ticket.s
ARG interrupt frequency 15
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              Thread 5              Thread 6              Thread 7
    Thread 8              Thread 9              Thread 10             Thread 11             Thread 12             Thread 13             Thread 14

    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   1005 mov  count, %ax
    0   1006 add  $1, %ax
    1   1007 mov  %ax, count
    1   1008 mov $1, %ax
    1   1009 fetchadd %ax, turn
    1   1010 sub  $1, %bx
    1   1011 test $0, %bx
    1   1012 jgt .top
    1   1013 halt
    1   ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch --
--- ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----  ----- Halt;Switch -----
    1                   1000 mov $1, %ax
    1   ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ---
--- ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------  ------ Interrupt ------
    1                   1000 mov $1, %ax
    1                   1001 fetchadd %ax, ticket
```



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=3,bx=3 -c -i 10

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p test-and-set.s -M count,mutex -R ax,bx -a bx=3,bx=3 -c -i 10
ARG seed 0
ARG numthreads 2
ARG program test-and-set.s
ARG interrupt frequency 10
ARG interrupt randomness False
ARG procsched
ARG argv bx=3,bx=3
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     3
    0     0      1     3   1000 mov  $1, %ax
    0     1      0     3   1001 xchg %ax, mutex
    0     1      0     3   1002 test $0, %ax
    0     1      0     3   1003 jne  .acquire
    0     1      0     3   1004 mov  count, %ax
    0     1      1     3   1005 add  $1, %ax
    1     1      1     3   1006 mov  %ax, count
    1     0      1     3   1007 mov  $0, mutex
    1     0      1     2   1008 sub  $1, %bx
    1     0      1     2   1009 test $0, %bx
    1     0      0     3   ------ Interrupt ------  ------ Interrupt ------
    1     0      1     3                           1000 mov  $1, %ax
    1     1      0     3                           1001 xchg %ax, mutex
    1     1      0     3                           1002 test $0, %ax
    1     1      0     3                           1003 jne  .acquire
    1     1      1     3                           1004 mov  count, %ax
    1     1      2     3                           1005 add  $1, %ax
    2     1      2     3                           1006 mov  %ax, count
    2     0      2     3                           1007 mov  $0, mutex
    2     0      2     2                           1008 sub  $1, %bx
    2     0      2     2                           1009 test $0, %bx
    2     0      1     2   ------ Interrupt ------  ------ Interrupt ------
    2     0      1     2   1010 jgt .top
    2     0      1     2   1000 mov  $1, %ax
    2     1      0     2   1001 xchg %ax, mutex
    2     1      0     2   1002 test $0, %ax
```

./x86.py -p yield.s -M count,mutex -R ax,bx -a bx=3,bx=3 -c -i 10

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p yield.s -M count,mutex -R ax,bx -a bx=3,bx=3 -c -i 10
ARG seed 0
ARG numthreads 2
ARG program yield.s
ARG interrupt frequency 10
ARG interrupt randomness False
ARG procsched
ARG argv bx=3,bx=3
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     3
    0     0       1     3   1000 mov  $1, %ax
    0     1       0     3   1001 xchg %ax, mutex
    0     1       0     3   1002 test $0, %ax
    0     1       0     3   1003 je .acquire_done
    0     1       0     3   1006 mov  count, %ax
    0     1       1     3   1007 add  $1, %ax
    1     1       1     3   1008 mov  %ax, count
    1     0       1     3   1009 mov  $0, mutex
    1     0       1     2   1010 sub  $1, %bx
    1     0       1     2   1011 test $0, %bx
    1     0       0     3   ------ Interrupt ------  ------ Interrupt ------
    1     0       1     3                           1000 mov  $1, %ax
    1     1       0     3                           1001 xchg %ax, mutex
    1     1       0     3                           1002 test $0, %ax
    1     1       0     3                           1003 je .acquire_done
    1     1       1     3                           1006 mov  count, %ax
    1     1       2     3                           1007 add  $1, %ax
    2     1       2     3                           1008 mov  %ax, count
    2     0       2     3                           1009 mov  $0, mutex
    2     0       2     2                           1010 sub  $1, %bx
    2     0       2     2                           1011 test $0, %bx
    2     0       1     2   ------ Interrupt ------  ------ Interrupt ------
    2     0       1     2   1012 jgt .top
    2     0       1     2   1000 mov  $1, %ax
    2     1       0     2   1001 xchg %ax, mutex
    2     1       0     2   1002 test $0, %ax
```

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

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

```
nazmul@ubuntu:~/ostep-homework/threads-locks$ ./x86.py -p test-and-test-and-set.s
ARG seed 0
ARG numthreads 2
ARG program test-and-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  mutex, %ax
1001 test $0, %ax
1002 jne .acquire
1003 mov  $1, %ax
1004 xchg %ax, mutex
1005 test $0, %ax
1006 jne .acquire
1007 mov  count, %ax
1008 add  $1, %ax
1009 mov  %ax, count
1010 mov  $0, mutex
1011 sub  $1, %bx
1012 test $0, %bx
1013 jgt .top
1014 halt
----- Halt;Switch -----  ----- Halt;Switch -----
                        1000 mov  mutex, %ax
                        1001 test $0, %ax
                        1002 jne .acquire
                        1003 mov  $1, %ax
                        1004 xchg %ax, mutex
                        1005 test $0, %ax
                        1006 jne .acquire
                        1007 mov  count, %ax
                        1008 add  $1, %ax
                        1009 mov  %ax, count
                        1010 mov  $0, mutex
```

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

```
nazmul@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
```