28

Locks(Kilitler)

Eşzamanlılığa girişte, eşzamanlı programlamada temel problemlerin birini gördük: bir dizi talimatları atomik olarak çalıştırmak isteriz, ancak halihazırda tek işlemcide varolan meşguliyetler (veya çoklu iş parçacıklarının çoklu işlemcilerde eşzamanlı olarak yürütülmesi) sebebiyle, yapamıyoruz. Bu bölümde , direkt olarak bu problemi, **lock(kilit)** olarak adlandırılan şeyin tanımı ile ele alacağız. Programcılar kaynak kodunu kilitlerle açıklar, bunu kritik bölümlerin etrafına koyar, ve böylece herhangi bir kritik bölümün sanki tek bir atomik talimatmış gibi yürütülmesini sağlarlar.

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

Örnek olarak, kritik bölümümüzün bu şekilde göründüğünü varsayalım, paylaşılan bir değişkenin standart gelişimi:

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

Tabiki, bağlı listeye öğe eklemek veya paylaşılan yapılara diğer daha karmaşık değişimler yapılması gibi başka kritik bölümlerde mümkündür ama şimdilik bu basit örnek ile yetineceğiz. Kilit kullanımı hakkında kritik bölüme şu şekilde bir dizi kod ekliyoruz:

```
1 lock_t mutex; // some globally-allocated lock 'mutex'
2 ...
3 lock(&mutex);
4 balance = balance + 1;
5 unlock(&mutex);
```

Bir kilit sadece bir değişkendir, bu yüzden birini kullanmak için, bir tip **kilit değişkeni (lock variable)** tanımlanmalıdır (yukarıdaki `mutex` gibi). Bu kilit değişkeni (veya kısaca "kilit") herhangi bir zamandaki kilidin durumunu tutar. Bu durum **mevcut(available)** (yada **açık(unlocked), serbest (free)**) durumlarından biri yani herhangi bir iş parçacığı tarafından tutulmayan kilitler, veya **kazanılmış (acquired)** (yada **kilitli (locked), tutulan (held)**), ve böylece bir kilit muhtemel kritik bölümde tam olarak bir iş parçacığı tarafından tutulurç Diğer bilgileride veri tipinde saklayabilirdik, örneğin hangi iş parçacığının kilidi tuttuğunu, veya kilit oluşturulması için kullanılan kuyruğu ancak bu gibi bilgiler kilit kullanıcısından saklıdır.

`lock()` ve `unlock()` işlemlerinin anlambilimi basittir. `lock()` ifadesi çağırmak kilidi çağırmayı dener; eğer iş parçacıklarından herhangi biri kilit tutmuyorsa (yani boştaysa), iş parçacığı kilidi kazanır ve kritik bölüme girer; Bu gibi iş parçacıkları kilidin **sahip (owner)** iş parçacığı da denir. Sonradan farklı iş parçacığı `lock()` ifadesini aynı kilit değişkeni üzerinde çağırırsa (bu örenekte `mutex`), kilit başka bir iş parçacığı tarafından tutulurken dönüt olamaycaktır; bu yöntemle, kritik bölümdeki kilidi tutan iş parçacığının haricindeki iş parçacıklarının kritik bölüme girmesi engellenir.

Kilidin sahibi `unlock()` işlevini çağırdığında, kilit mevcut(boş) hale gelir.Eğer diğer iş parçacıkları kilit için beklemiyorsa (yani `lock()` ifadesini çağıran iş parçacığı yoksa), kilidin durumu basitçe serbest hale getirilir. Eğer bekleyen iş parçacıkları varsa (`lock()` işleminde bekleyen), biri (mutlaka) kilidin durumundaki bu değişikliği farkedecek (bilgilendirilecek), kilidi kazanıp kritik bölüme girecektir.

Kilitler programcılara minimal düzeyde planlamada kontrol sağlar. Genel olarak, iş parçacıklarını programcı tarafından yaratılan İS tarafından planlanan varlıklar olarak görürüz, tabi İS seçtiği tarzda.Kilitler bazı kontrolleri bölüm kodunun etrafına kilit koyarak geri programcıya verir; programcı da tekli iş parçacığı haricindeki iş parçacıklarının o kod içinde aktif olamayacağını garantiler. Böylece kilitler geleneksel İS planlamasını kaostan daha kontrollü bir aktiviteye dönüştürmeye yardımcı olur.

28.2 Pthread Kilitleri (Pthread Locks)

POSIX kütüphanesi kilit adının yerine **karşılıklı dışlama (mutex)** adını kullanır, bu da iş parçacıkları arasında **karşılıklı dışlama (mutual exclusion)** anlamına gelir, yani bir iş parcacığı kritik bölümde ise bölümü bitirinceye kadar diğer iş parçacıklarının bölüme girmesini engeller. Böylece aşağıdaki POSIX iş parçacığı kodunu gördüğünüzde yukarıdakiyle aynı şeyin yapıldığını anlamalısınız (kilitleme ve kilit açma sırasında hataları control eden sarmalayıcılarımızı tekrar 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);
```

Burda POSIX sürümünün kilitlemek ve kilit açmak için bir değişken geçirdiğini de fark edebilirsiniz, diğer değişkenleri korumak için *farklı* kilitler kullandığımızı fark edebilirsiniz. Bunu yapmak tutarlılığı arttırabilir: herhangi bir zamandan ve bölüme erişimdde kullanılan büyük tek bir kilidin yerine(**iri taneli (coarse-grained)** kilitleme yöntemi), bir tanesi sıkça farklı veri ve veri yapılarını farklı kilitler ile koruyacaktır, böylece tek bir seferde birden fazla iş parçacığı kod içinde kilitlenmiş olacaktır(**ince taneli (fine-grained)** yaklaşım).

28.3 Kilit Oluşturmak (Building A Lock)

Şimdiye kadar bir programcının gözünden kilidin nasıl çalıştığını yeterince anladınız. Peki ama kilidi nasıl oluştururuz ? Ne gibi bir donanım desteği gereklidir ? İşletim Sistemi ne destekler ? Bu bölümde bu sorulara yer vererek cevaplandırmaya çalışacağız.

Püf Noktası: Nasıl Kilit İnşa Edilir ?
Nasıl verimli bir kilit inşa ederiz? Verimli kilitler düşük maaliyetle karşılıklı dışlama sağlar, aşağıda tartışacağımız birkaç özelliği de elde etmemizi sağlar. Ne donanım desteği gereklidir? İşletim Sistemi ne destekler ?

Çalışan bir kilit oluşturmak için, eski bir dostumuzun, donanımın, aynı zamana iyi dostumuz, İşletim Sisteminin, yardımına ihtiyaç duyacağız. Yıllar geçtikçe, çeşitli bilgisayar mimarilerinin komut setlerine bir dizi farklı donanım ilkeleri eklenmiştir; tabi bu komut setlerinin nasıl uygulandığını ele almayacağız (sonuçta bu bilgisayar mimarisi dersinin konusudur), bunun yerine onları kullanarak kilitler gibi ilkel karşılıklı dışlamayı nasıl inşa edebileceğimizi çalıçacağız. İşletim Sİsteminin bütün resme nasıl dahil olarak bizim karmaşık kililtleme kütüphanesi oluşturduğumuza değineceğiz.

28.4 Kilitleri Değerlendirmek (Evaluating Locks)

Herhangi bir kilit oluşturmadan önce, hedeflerimizin ne olduğunu anlamalıyız. Böylece belirli bir kilit uygulaması etkinliğinin nasıl değerlendirebileceğimizi sormalıyız. Bir kilidin çalışıp çalışmadığını (hatta iyi çalışıp çalışmadığını) değerlendirmek için bazı temel kriterler oluşturmalıyız. Birincisi, kilidin **karşılıklı dışlamayı (mutual exclusion)** sağlamak olan temel görevini yerine getirip getirmediğidir. Basitçe, kilit çalışıyormu, çoklu iş parçacıklarının kritik bölüme girmesine engel olabiliyor mu ?
İkincisi ise **tarafsızlık (fairness)**.Her bir iş parçacığı kilitler boştayken onları tarafsız bir şekilde elde edebililyor mu ? Buna bakmanın bir başka yolu da daha uç durumu incelemektir: Herhangi bir iş parçacığı kilit için istekte bulunurken **açlık (starve)** durumunda kalıp etkisiz hale geliyor ve böylece istekte bulunabiliyor mu ?

Son kriter ise **performans (performance)** 'tır, özellikle eklenen zaman genel giderleri kilidi kullanarak. Burada tartışmaya değer birkaç farklı durum var. Bunlardan biri çekişme olmaması durumudur; Tek bir iş parçacığı çalışırken ve kilidi tutup serbest bıraktığında, bunu yapmanın yükü nedir? Bir diğeri, tek bir CPU'daki kilit için birden fazla iş parçacığının mücadele ettiği durumdur; Bu durumda performans endişeleri var mı? Genel olarak, birden fazla CPU söz konusu olduğunda ve her biri kilit için yarışan iş parçacıkları olduğunda kilit nasıl performans gösterir? Bu farklı senaryoları karşılaştırarak, aşağıda açıklandığı gibi çeşitli kilitleme tekniklerini kullanmanın performans etkisini daha iyi anlayabiliriz.

28.5 Kesmeleri Denetleme (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 edildi. Kod şöyle görünürdü::

```
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 bölümün içindeki kodun kesintiye uğramamasını ve böylece atomikmiş gibi yürütülmesini sağlarız. İşimiz bittiğinde, kesintileri yeniden etkinleştiririz (yine bir donanım yapısı aracılığıyla) ve böylece program her zamanki gibi ilerler.

Bu yaklaşımın ana olumlu yanı sadeliğidir. Bunun neden işe yaradığını anlamak için kesinlikle başınızı çok fazla çizmenize gerek yok. Kesintisiz olarak, bir iş parçacığı, yürüttüğü kodun yürütüleceğinden ve başka hiçbir iş parçacığının buna müdahale etmeyeceğinden emin olabilir.

Negatifler maalesef çoktur. İlk olarak, bu yaklaşım, herhangi bir çağıran iş parçacığının *ayrıcalıklı* bir işlem gerçekleştirmesine (kesintileri açıp kapatmasına) izin vermemizi ve böylece bu tesisin kötüye kullanılmadığına *güvenmemizi* gerektirir. Zaten bildiğiniz gibi, keyfi bir pro-gram'a güvenmemiz gerektiğinde, muhtemelen başımız belada. Burada sorun çeşitli şekillerde ortaya çıkıyor: açgözlü bir program, yürütmesinin başında `lock()` öğesini çağırabilir ve böylece işlemciyi tekeline alabilir; daha da kötüsü, hatalı veya mali açıdan kötü bir program `lock()` öğesini çağırabilir ve sonsuz bir döngüye girebilir. Bu ikinci durumda, işletim sistemi hiçbir zaman sistemin kontrolünü geri kazanmaz ve tek bir yöntem vardır: sistemi yeniden başlatmak. Kesme devre dışı bırakmayı genel amaçlı bir senkronizasyon çözümü olarak kullanmak, uygulamalara çok fazla güven gerektirir.

İkincisi, yaklaşım çok işlemcilerde çalışmaz. Birden çok iş parçacığı farklı CPU'larda çalışıyorsa ve her biri aynı kritik bölüme girmeye çalışıyorsa, kesmelerin devre dışı bırakılıp bırakılmadığı önemli değildir; iş parçacıkları diğer işlemcilerde çalışabilir ve bu nedenle kritik bölüme girebilir. Çoklu işlemciler artık yaygın olduğundan, genel çözümümüz bundan daha iyidir.

Üçüncüsü, kesintileri uzun süre kapatmak, kesintilerin kaybolmasına neden olabilir ve bu da ciddi sistem sorunlarına yol açabilir. Örneğin, cpu'nun bir disk aygıtının okuma isteğini tamamladığı gerçeğini kaçırdığını düşünün. İşletim sistemi, söz konusu okuma için bekleme sürecini uyandırmayı nasıl bilecek?

Son olarak ve muhtemelen en az önemlisi, bu yaklaşım verimsiz olabilir. Normal komut yürütme ile karşılaştırıldığında, kesmeleri maskeleyen veya maskesini kaldıran kod, modern CPU'lar tarafından yavaş yürütülme eğilimindedir.

```
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 metinlerde karşılıklı dışlama ilkeli olarak kullanılır. Örneğin, bazı durumlarda, bir işletim sisteminin kendisi, kendi veri yapılarına erişirken atomikliği garanti etmek veya en azından dağınık kesme işleme durumlarının ortaya çıkmasını önlemek için kesme maskelemesini kullanacaktır. Bu kullanım mantıklıdır, çünkü güven sorunu işletim sistemi içinde kaybolur ve bu da her zaman ayrıcalıklı işlemleri gerçekleştirmek için kendine güvenir.

28.6 Başarısız Bir Girişim: Yalnızca Yükler/Depolar Kullanma (A Failed Attempt: Just Using Loads/Stores)

Kesme tabanlı tekniklerin ötesine geçmek için, CPU donanımına ve uygun bir kilit oluşturmamız için bize sağladığı talimatlara güvenmemiz gerekecek. Önce tek bir bayrak değişkeni kullanarak basit bir kilit oluşturmaya çalışalım. Bu başarısız denemede, bir kilit oluşturmak için gereken bazı temel fikirleri göreceğiz ve (umarım) tek bir değişkeni kullanmanın ve normal yükler ve depolar aracılığıyla erişmenin neden yetersiz olduğunu göreceğiz.

Bu ilk denemede (Şekil 28.1) fikir oldukça basittir: bazı iş parçacıklarının bir kilide sahip olup olmadığını belirtmek için basit bir değişken (`flag`) kullanın. Kritik bölüme giren ilk iş parçacığı, bayrağın 1'e eşit olup olmadığını test eden (bu durumda değil) `lock()` öğesini çağırır ve ardından iş parçacığının artık kilidi tuttuğunu belirtmek için bayrağı 1 olarak ayarlar. Kritik bölümle işiniz bittiğinde, iş parçacığı `unlock()` işlevini çağırır ve bayrağı siler, böylece kilidin artık tutulmadığını gösterir.

İlk iş parçacığı kritik bölümdeyken başka bir iş parçacığı `lock()` işlevini çağırırsa, yalnızca **dön-bekle (spin-wait** döngüde bu iş parçacığının `unlock()` işlevini çağırması ve bayrağı temizlemesi için while döngüsünde bekleyin. Bu ilk iş parçacığı bunu yaptıktan sonra, bekleyen iş parçacığı while döngüsünden düşecek, bayrağı kendisi için 1 olarak ayarlayacak ve

kritik bölüme geçecektir.

Ne yazık ki, kodun iki sorunu var: biri doğruluk diğeri ise performans

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!)	

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

Eşzamanlı programlama hakkında düşünmeye alıştıktan sonra doğruluk problemini görmek basittir. Şekil 28.2'de serpiştirilmiş kodu hayal edin; başlamak için `flag=0` olduğunu varsayalım.

Bu serpiştirmeden görebileceğiniz gibi, zamanında (zamansız?) kesmeler, her iki iş parçacığının da bayrağı 1 olarak ayarladığı ve böylece *her iki* iş parçacığının da kritik bölüme girebileceği bir durumu kolayca üretebiliriz. Bu davranış, profesyonellerin "kötü" dediği şeydir – açıkçası en temel gereksinimi sağlayamadık: karşılıklı dışlanma sağlamak.

Daha sonra ele alacağımız performans sorunu, bir iş parçacığının halihazırda tutulan bir kilidi elde etmek için bekleme şeklinin şu olmasıdır: **Dönme-bekleme (spin-waiting)** olarak bilinen bir teknik olan bayrak değerini sonsuz bir şekilde kontrol eder. Döndürme beklemesi, başka bir iş parçacığının kilidi açmasını beklemek için zaman harcar. Garsonun beklediği iş parçacığının bile çalışamadığı (en azından bir bağlam anahtarı oluşana kadar) tek işlemcide atık son derece yüksektir! Bu nedenle, ilerledikçe ve daha karmaşık çözümler geliştirdikçe, bu tür israflardan kaçınmanın yollarını da düşünmeliyiz.

28.7 Test ve Set ile Çalışan Döner Kilitler 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) çalışmadığından, sistem tasarımcıları kilitleme için donanım desteği icat etmeye başladılar. 1960'ların başındaki [M82] Burroughs B5000 gibi en eski çok işlemcili sistemler böyle bir desteğe sahipti; Bugün tüm sistemler, tek CPU sistemleri için bile bu tür bir destek sağlıyor.

Anlaşılması gereken en basit donanım desteği şu şekilde bilinir:

test et ve ayarla (test-and-set) (veya **atomıik takas**) talimatı. Test et ve ayarla komutunun ne yaptığını aşağıdaki C kod parçacığıyla 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 }
```

[1]Each architecture that supports test-and-set calls it by a different name. On SPARC it is

called the load/store unsigned byte instruction (`ldstub`); on x86 it is the locked version of the atomic exchange (`xchg`).

BİR KENARA: DEKKER VE PETERSON'IN ALGORİTMALARI (ASIDE: DEKKER'S AND PETERSON'S ALGORITHMS)

1960'larda Dijkstra eşzamanlılık problemini arkadaşlarına yöneltti ve onlardan biri olan Theodorus Jozef Dekker adında bir matematikçi bir çözüm buldu [D68]. Burada tartıştığımız, özel donanım talimatları ve hatta işletim sistemi desteği kullanan çözümlerin aksine, **Dekker 'in Algoritması (Dekker's algorithm)** yalnızca yükler ve depolar kullanır (birbirlerine göre atomik olduklarını varsayarsak, bu erken donanımda geçerliydi).

Dekker'in yaklaşımı daha sonra Peterson tarafından rafine edildi [S81]. Bir kez daha, yalnızca yükler ve depolar kullanılır ve fikir, iki iş parçacığının aynı anda hiçbir zaman kritik bir bölüme girmemesini sağlamaktır. İşte **Peterson ' un Algoritması (Peterson's algorithm)** (iki iş parçacığı için); Kodu anlayıp anlayamayacağınıza bakın. `flag` ve dönüş değişkenleri ne için kullanılır?

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

void init() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}
void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while it's not your turn
}
void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

Bazı nedenlerden dolayı, özel donanım desteği olmadan çalışan kilitler geliştirmek bir süredir tüm öfke haline geldi ve teori türlerine üzerinde çalışılması gereken birçok sorun verdi. Tabii ki, insanlar biraz donanım desteği almanın çok daha kolay olduğunu fark ettiklerinde (ve aslında bu desteğin çok işlemin ilk günlerinden beri var olduğunu) bu iş kolu oldukça işe yaramaz hale geldi. Ayrıca, yukarıdakiler gibi algoritmalar, mod- ern donanımı üzerinde çalışmaz (rahat bellek tutarlılığı modelleri nedeniyle), bu nedenle onları eskisinden daha az kullanışlı hale getirir. Yine de daha fazla araştırma tarihin çöplüğüne düştü...

```
1  typedef struct__lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock 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 }
```

Şekil 28.3:**Test ve set Kullanarak Basit Bir Döndürme Kilidi (A Simple Spin Lock Using Test-and-set)**

Test ve set talimatının yaptığı şey aşağıdaki gibidir. Eski ptr'nin işaret ettiği `old ptr` döndürür ve aynı anda söz konusu değeri yeniye günceller. Anahtar, elbette, bu işlem dizisinin **atomik (atomically)** olarak gerçekleştirilmesidir.

"Test et ve ayarla" olarak adlandırılmasının nedeni, bellek konumunu aynı anda yeni bir değere "ayarlarken" eski değeri (döndürülen) "test etmenize" olanak sağlamasıdır; Anlaşıldığı üzere, bu biraz daha güçlü talimat, basit bir **döndürme kilidi (spin lock)** oluşturmak için yeterlidir, şimdi Şekil 28.3'te incelediğimiz gibi. Ya da daha iyisi: önce kendin bul!

Bu kilidin neden çalıştığını anladığımızdan emin olalım. Önce bir iş parçacığının `lock()` işlevini çağırdığı ve şu anda kilidi tutan başka bir iş parçacığının bulunmadığı durumu düşünün; Bu nedenle, bayrak 0 olmalıdır. İş parçacığı çağırdığında `TestAndSet (flag, 1)`, yordam eski değerini döndürür bayrak, hangisi 0; bu nedenle, değerini test eden çağıran iş parçacığı bayrak, while döngüsünde dönerken yakalanmayacak ve kilidi alacaktır. İplik aynı zamanda değeri 1 olarak ayarlayacaktır, böylece kilidin artık tutulduğunu gösterir. İş parçacığı kritik bölümü ile tamamlandığında, bayrağı sıfıra *set* için `unlock()` işlevini çağırır. `unlock() lock() TestAndSet() flag,1` *set*

Hayal edebileceğimiz ikinci durum, bir iş parçacığı zaten kilidi tuttuğunda ortaya çıkar (yani, bayrak 1'dir). Bu durumda, bu iş parçacığı `lock()`öğesini çağırır ve ardından `TestAndSet(flag, 1)` öğesini de çağırır. Bu kez `TestAndSet()` aynı anda 1'e yeniden ayarlarken, 1 olan (kilit tutulduğu için) bayraktaki 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 nihayet başka bir iş parçacığı tarafından 0'a ayarlandığında, bu iş parçacığı tekrar `TestAndSet()`öğesini çağırır ve şimdi değeri atomik olarak 1 olarak ayarlarken 0 değerini döndürür ve

böylece kilidi alır ve kritik bölüme girer.

İPUCU: EŞZAMANLILIĞI KÖTÜ AMAÇLI BİR ZAMANLAYICI OLARAK DÜŞÜNÜN(TIP: THINK ABOUT CONCURRENCY AS A MALICIOUS SCHEDULER)

Bu örnekten, eşzamanlı yürütmeyi anlamak için izlemeniz gereken yaklaşım hakkında bir fikir edinebilirsiniz. Yapmaya çalışmanız gereken şey, senkronizasyon ilkellerini oluşturmaya yönelik zayıf girişimlerini engellemek için iş parçacıklarını en uygunsuz zamanlarda kesen **kötü planlayıcı (malicious scheduler)** olduğunuzu iddia etmektir. Ne kadar kötü bir planlayıcısın! Kesin kesinti sırası olanaksız olsa da mümkündür ve belirli bir yaklaşımın işe yaramadığını göstermek için ihtiyacımız olan tek şey budur. Kötü niyetli düşünmek faydalı olabilir! (en azından bazen)

Hem testi (eski kilit değerinin) hem de seti (yeni değerin) tek bir atomik işlem yaparak, kilidi yalnızca bir iş parçacığının almasını sağlarız. Ve bu, çalışan bir karşılıklı dışlama ilkelinin nasıl inşa edileceğidir!

Artık bu tür bir kilidin neden genellikle **döndürme kilidi (spin lock)** olarak adlandırıldığını da anlayabilirsiniz. Oluşturması en basit kilit türüdür ve kilit kullanılabilir hale gelene kadar CPU döngülerini kullanarak basitçe döner. Tek bir işlemcide düzgün çalışması için **önleyici zamanlayıcı (preemptive scheduler)** gerekir (yani, zaman zaman farklı bir iş parçacığını çalıştırmak için bir iş parçacığını bir zamanlayıcı aracılığıyla kesecek olan). Bir CPU üzerinde dönen bir iplik onu asla bırakmayacağından, önleme olmadan, döndürme kilitleri tek bir cpu'da pek bir anlam ifade etmez.

28.8 Döndürme Kilitlerinin Değerlendirilmesi (Evaluating Spin Locks)

Temel döndürme kilidimiz göz önüne alındığında, şimdi daha önce açıklanan eksenlerimiz boyunca ne kadar etkili olduğunu değerlendirebiliriz. Bir kilidin en önemli yönü **doğruluktur (correctness)**: karşılıklı dışlama sağlar mı? Buradaki cevap evet: döndürme kilidi, bir seferde yalnızca tek bir iş parçacığının kritik bölüme girmesine izin verir. Böylece doğru bir kilidimiz var.

Bir sonraki eksen **tarafsızlık (fairness)**. Bekleyen bir iş parçacığına dönüş kilidi ne kadar adil? Bekleyen bir iş parçacığının kritik bölüme gireceğini garanti edebilir misiniz? Ne yazık ki buradaki cevap kötü haber: dönen kilitler herhangi bir adalet garantisi vermiyor. Gerçekten de, bir iplik eğirme çekişme altında sonsuza kadar dönebilir. Basit sıkma kilitleri (şimdiye kadar tartışıldığı gibi) adil değildir ve açlığa yol açabilir.

Son eksen **performanstır (performance)**. Döndürme kilidi kullanmanın maliyeti nedir? Bunu daha dikkatli analiz etmek için birkaç farklı durum hakkında düşünmenizi öneririz. İlkinde, tek bir işlemcide kilit için rekabet eden iş parçacıklarını hayal edin; ikincisinde, birçok CPU'ya yayılmış iş parçacıklarını düşünün.

Döndürme kilitleri için, tek CPU durumunda, performans genel giderleri oldukça acı verici olabilir; Kilidi tutan iş parçacığının kritik bir bölüm içinde ön plana çıktığı durumu hayal edin. Zamanlayıcı daha sonra her biri kilidi açmaya çalışan diğer tüm iş parçacıklarını çalıştırabilir (N − 1 diğerleri

olduğunu düşünün). Bu durumda, bu iş parçacıklarının her biri, cpu'dan vazgeçmeden önce bir zaman dilimi boyunca dönerek CPU döngülerini boşa harcar.

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 A iş Parçacığını ve CPU 2'deki B iş parçacığını hayal edin. İş parçacığı A (CPU 1) kilidi kapar ve ardından İş Parçacığı B denerse, B dönecektir (CPU 2'de). Bununla birlikte, muhtemelen kritik bölüm kısadır ve bu nedenle kısa süre sonra kilit kullanılabilir hale gelir ve İş Parçacığı B tarafından etkinleştirilir. Başka bir işlemcide tutulan bir kilidi beklemek için eğirme, bu durumda çok fazla döngü harcamaz ve bu nedenle etkili olabilir.

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

Şekil 28.4: **Karşılaştırma ve Yer değiştirme (Compare-and-swap)**

28.9 Karşılaştırma ve Yer Değiştirme (Compare-And-Swap)

Bazı sistemlerin sağladığı diğer bir donanım ilkesi **karşılaştır-yer değiştir (compare-and-swap)** talimatı (sparc'da örnek olarak çağrıldığı gibi) veya **karşılaştır ve değiştir (compare-and-exchange)** (x86'da çağrıldığı gibi) olarak bilinir. Bu tek talimat için C sözde kodu Şekil 28.4'te bulunmaktadır.

Temel fikir, `ptr` tarafından belirtilen adresteki değerin `expected`'a eşit olup olmadığını test etmek için karşılaştır ve değiştir; öyleyse, `ptr` tarafından işaret edilen bellek konumunu yeni değerle güncelleyin. Değilse, hiçbir şey yapmayın. Her iki durumda da, bu bellek konumundaki orijinal değeri döndürün, böylece başarılı olup olmadığını bilmek için karşılaştır ve değiştir'i çağıran kodun kullanılmasına izin verin.

Karşılaştır ve değiştir talimatıyla, test et ve ayarla talimatına oldukça benzer bir man-nerde bir kilit oluşturabiliriz. Örneğin, yukarıdaki `lock()` yordamını aşağıdakilerle 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; bayrağın 0 olup olmadığını kontrol eder ve eğer öyleyse, atomik olarak 1'de takas ederek kilidi alır. Kilidi tutulurken elde etmeye çalışan iplikler, kilit nihayet serbest bırakılana kadar dönerken sıkışacaktır.

Eğer gerçekten bir C-callable x86-versiyonunu nasıl yapacağınızı görmek istiyorsanız karşılaştır ve değiştir, kod dizisi ([S05] 'ten) yararlı olabilir[2]

Son olarak, hissetmiş olabileceğiniz gibi, karşılaştır ve değiştir, test et ve ayarla'dan daha güçlü bir talimattır. Bu gücü biraz kullanacağız[2]

[2] `github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-locks`

Kilit-serbest senkronizasyon (lock-free synchronization) gibi konulara kısaca girdiğimiz gelecek [H91]. Bununla birlikte, onunla basit bir döndürme kilidi oluşturursak, davranışı yukarıda analiz ettiğimiz döndürme kilidi ile aynıdır.

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

Bazı platformlar, kritik bölümlerin oluşturulmasına yardımcı olmak için birlikte çalışan bir çift talimat sağlar. MIPS mimarisinde [H93], örneğin, **yük-bağlı (load-linked)** ve **depo-şartlı (store-conditional)** yönergeler, kilitler ve diğer eşzamanlı yapılar oluşturmak için birlikte kullanılabilir. Bu talimatların 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 getirir ve onu bir kayıt defterine yerleştirir. Anahtar farkı, yalnızca adrese müdahale eden bir depo gerçekleşmemişse başarılı olan (ve yalnızca yük bağlantılı adreste depolanan değeri güncelleyen) depo koşuluyla birlikte gelir. Başarı durumunda, depo koşullu 1 döndürür ve `ptr` 'deki değeri değere günceller; başarısız olursa, `ptr` 'deki `value` güncellenmez ve 0 döndürülür.

Kendinize bir meydan okuma olarak, yük bağlantılı ve depo koşullu kullanarak bir kilidin nasıl oluşturulacağını düşünmeyi deneyin. Ardından, işiniz bittiğinde, basit bir çözüm sağlayan aşağıdaki koda bakın. Dene bakalım! 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). Bundan sonra, iş parçacığı kilidi depo koşulu aracılığıyla almaya çalışır; Başarılı olursa, iş parçacığı bayrağın değerini atomik olarak 1 olarak değiştirdi ve böylece kritik bölüme geçebilir.

Depo koşulunun başarısızlığının nasıl ortaya çıkabileceğine dikkat edin. Bir iş parçacığı `lock()` öğesini çağırır ve kilit tutulmadığı için 0 döndürerek yüke bağlı olanı yürütür. Depo koşulunu denemeden önce kesintiye uğrar ve kilit koduna başka bir iş parçacığı girerek yüke bağlı talimatı da 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 }
```

Şekil 28.5: **Yük-Bağlı ve Depo-Şartlı (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 }
```

Şekil 28.6: **YB/DŞ kullanarak Kilit Oluşturma (Using LL/SC To Build A Lock)**

ve ayrıca 0 alıyor ve devam ediyor. Bu noktada, her biri yüke bağlı iki iş parçacığı çalıştırdı ve her biri depo koşulunu denemek üzere. Bu yönergelerin temel özelliği, bu iş parçacıklarından yalnızca birinin bayrağı 1'e güncellemeyi başarması ve böylece kilidi edinmesidir; Depo koşullu girişimi için ikinci iş parçacığı başarısız olur (çünkü diğer iş parçacığı, yük bağlantılı ve depo koşullu arasındaki bayrak değerini güncelledi) ve böylece kilidi tekrar almaya çalışmalıyız.

Birkaç yıl önce sınıfta, lisans öğrencisi David Capel, kısa devre yapan boole koşullarından hoşlananlar için yukarıdakilerin daha özlü bir biçimini önerdi. Bakalım neden eşdeğer olduğunu bulabilecek misin. Kesinlikle daha kısa!

```
1  void lock(lock_t *lock) {
2      while (LoadLinked(&lock->flag) ||
3             !StoreConditional(&lock->flag, 1))
4          ; // spin
5  }
```

28.11 Getir ve Ekle (Fetch-And-Add)

Son bir donanım ilkeli, eski değeri kısmi bir adreste döndürürken bir değeri atomik olarak artıran **getir ve ekle (fetch-and-add)** talimatıdır. Getir ve ekle yönergesi için C sözde kodu şuna benzer:

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

Programcılar, bir şeyler yapmak için ne kadar kod yazdıkları konusunda övünme eğilimindedir. Bunu yapmak temelde bozuldu. Övünmesi gereken şey, kişinin belirli bir görevi yerine getirmek için ne kadar az kod yazdığıdır. Kısa, özlü kod her zaman tercih edilir; anlaşılması daha kolaydır ve daha az hataya sahiptir. Hugh Lauer'in dediği gibi, Pilot işletim sisteminin inşasını tartışırken: "Aynı insanlar iki kat daha fazla zamana sahip olsaydı, kodun yarısında bir sistem kadar iyi üretebilirlerdi." [L81] Buna Lauer Yasası diyeceğiz ve hatırlamakta fayda var. Bir dahaki sefere ödevi bitirmek için ne kadar kod yazdığınız konusunda övündüğünüzde, tekrar düşünün ya da daha iyisi, geri dönün, yeniden yazın ve kodu olabildiğince açık ve tutarlı hale getirin.

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

Tek bir değer yerine, bu çözüm bir kilit oluşturmak için birlikte bir bilet ve dönüş değişkeni kullanır. Temel işlem oldukça basittir: bir iş parçacığı bir kilit almak istediğinde, önce bilet değerine atomik bir getir ve ekle yapar; Bu değer şimdi bu iş parçacığının "dönüşü" (`myturn`) olarak kabul edilir. Genel olarak paylaşılan `lock->turn` daha sonra hangi iş parçacığının dönüşünün olduğunu belirlemek için kullanılır; ne zaman (`myturn == turn`) belirli bir iş parçacığı için kritik bölüme girme sırası o iş parçacığıdır. Kilit açma, bir sonraki bekleyen iş parçacığının (varsa) artık kritik bölüme girebileceği şekilde dönüşü artırarak gerçekleştirilir

Bu çözümle önceki denemelerimize kıyasla önemli bir farklılığa dikkat edin: tüm iş parçacıkları için ilerleme sağlar. Bir iş parçacığına bilet değeri atandığında, gelecekte bir noktada zamanlanacaktır (önündekiler kritik bölümden geçip kilidi bıraktıktan sonra). Önceki denemelerimizde böyle bir garanti mevcut değildi; Test ve sette dönen bir iplik (örneğin), diğer iplikler kilidi alıp serbest bıraksa bile sonsuza kadar dönebilir.

28.12 Aşırı Dönme Şimdi Ne Var ? (Too Much Spinning: What Now?)

Basit donanım tabanlı kilitlerimiz basittir (yalnızca birkaç satır kod) ve çalışırlar (isterseniz biraz kod yazarak bile kanıtlayabilirsiniz), bunlar herhangi bir sistemin veya kodun iki mükemmel özelliğidir. Ancak bazı durumlarda bu çözümler oldukça verimsiz olabilir. Tek bir işlemcide iki iş parçacığı çalıştırdığınızı düşünün. Şimdi bir iş parçacığının (iş parçacığı 0) kritik bir bölümde olduğunu ve bu nedenle bir kilidin tutulduğunu ve ne yazık ki kesintiye uğradığını hayal edin. İkinci iş parçacığı (iş parçacığı 1) şimdi kilidi almaya çalışır, ancak tutulduğunu bulur.

```
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**

Böylece dönmeye başlar. Ve dön. Sonra biraz daha dönüyor. Ve son olarak, bir zamanlayıcı kesintisi söner, iplik 0 tekrar çalıştırılır, bu da kilidi serbest bırakır ve son olarak (bir dahaki sefere çalıştığında, diyelim), iplik 1'in çok fazla dönmesi gerekmeyecek ve kilidi elde edebilecektir. Böylece, bir iş parçacığı böyle bir durumda dönerken yakalandığında, değişmeyecek bir değeri kontrol etmekten başka bir şey yapmadan tüm zaman dilimini boşa harcar! Bir kilit için yarışan N iş parçacığı ile sorun daha da kötüleşir; $N - 1$ zaman dilimleri benzer şekilde boşa harcanabilir, basitçe döndürülebilir ve tek bir iş parçacığının kilidi serbest bırakmasını bekleyebilir. Ve böylece, bir sonraki sorunumuz:

PÜF NOKTASI: EĞİRME NASIL ENGELLENİR? THE CRUX: HOW TO AVOID SPINNING

CPU üzerinde gereksiz yere zaman kaybetmeyen bir kilidi nasıl geliştirebiliriz?

Donanım desteği tek başına sorunu çözemez. İşletim sistemi desteğine de ihtiyacımız olacak! Şimdi bunun nasıl işe yarayacağını bulalım.

28.13 Basit Bir Yaklaşım: Sadece İzin Ver Bebeğim (A Simple Approach: Just Yield, Baby)

Donanım desteği bizi oldukça ileri götürdü: çalışan kilitler ve hatta (bilet kilidinde olduğu gibi) kilit alımında adalet. Ancak yine de bir sorunumuz var: kritik bir bölümde bir bağlam anahtarı oluştuğunda ve iş parçacıkları kesintisiz dönmeye başladığında, kesintiye uğramış (kilit tutma) iş parçacığının yeniden çalıştırılmasını beklerken ne yapmalı?

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

Şekil 28.8: **Test Set ve İzin ver ile Kilitler (Lock With Test-and-set And Yield)**

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

IBu yaklaşımda, bir iş parçacığının CPU' dan vazgeçmek ve başka bir iş parçacığının çalışmasına izin vermek istediğinde çağırabileceği bir işletim sistemi ilkel `yield()` varsayıyoruz. Bir iş parçacığı üç durumdan birinde olabilir (**çalışıyor (running)**, **hazır (ready)** veya **engellenmiş (blocked)**); verim, yalnızca arayanı çalışma durumundan hazır duruma taşıyan ve böylece başka bir iş parçacığını çalışmaya teşvik eden bir sistem çağrısıdır. Böylece, verimli iş parçacığı esas olarak kendini **plansızlaştırır (deschedules).**

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

Şimdi, tekrar tekrar bir kilit için yarışan birçok iş parçacığının (100 diyelim) olduğu durumu ele alalım. Bu durumda, bir iş parçacığı kilidi alır ve serbest bırakmadan önce engellenirse, diğer 99'un her biri `lock()` öğesini çağırır, tutulan kilidi bulur ve CPU'yu verir. Bir tür round-robin zamanlayıcısı varsayarsak, kilidi tutan iş parçacığı yeniden çalışmaya başlamadan önce 99'un her biri bu çalıştırma ve verme modelini yürütür. Eğirme yaklaşımımızdan daha iyi olsa da (ki bu 99 zaman dilimi eğirmeyi boşa harcar), bu yaklaşım hala maliyetlidir; Bir bağlam anahtarının maliyeti önemli olabilir ve bu nedenle bol miktarda atık vardır.

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

28.14 Kuyrukları Kullanmak: Döndürmek Yerine Uykuya Almak (Using Queues: Sleeping Instead Of Spinning)

Önceki yaklaşımlarımızla ilgili asıl sorun, şansa çok fazla şey bırakmalarıdır. Zamanlayıcı, hangi iş parçacığının daha sonra çalışacağını belirler; Zamanlayıcı kötü bir seçim yaparsa, kilidi bekleyerek 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 yoktur.

```
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 }
```

Şekil 28.9: **Kuyruklar, Test-set, İzin Verme ve Uyanma ile Kilitler (Lock With Queues, Test-and-set, Yield, And Wakeup)**

Böylece, mevcut tutucu serbest bıraktıktan sonra kilidi almak için hangi iş parçacığının daha sonra alacağı üzerinde açıkça bir miktar kontrol uygulamamız gerekir. Bunu yapmak için, hangi iş parçacıklarının kilidi almayı beklediğini takip etmek için biraz daha fazla işletim sistemi desteğine ve bir sıraya ihtiyacımız olacak.

Basit olması için Solaris tarafından sağlanan desteği iki çağrı açısından kullanacağız: çağıran bir iş parçacığını uyku moduna geçirmek için `park()` ve `threadID` tarafından belirlenen belirli bir iş parçacığını uyandırmak için `unpark(threadID)`. Bu iki diş, tutulan bir kilidi almaya çalışırsa arayanı uyutan ve kilit serbest olduğunda onu uyandıran bir kilit oluşturmak için birlikte kullanılabilir. Bu tür ilkellerin olası bir kullanımını anlamak için Şekil 28.9'daki koda bakalım.

KENARA: DÖNMEKTEN KAÇINMAK İÇİN DAHA FAZLA NEDEN: ÖNCELİKLİ TERS ÇEVİRME ASIDE: MORE REASON TO AVOID SPINNING: PRIORITY INVERSION

Döndürme kilitlerinden kaçınmanın iyi bir nedeni performanstır: ana metinde açıklandığı gibi, kilit tutulurken bir iş parçacığı kesintiye uğrarsa, döndürme kilitlerini kullanan diğer iş parçacıkları, kilidin kullanılabilir olmasını beklemek için büyük miktarda CPU zamanı harcar. Bununla birlikte, bazı sistemlerde dönen kilitlerden kaçınmanın başka bir karşılıklı nedeni olduğu ortaya çıktı: doğruluk. Dikkat edilmesi gereken prob- lem, ne yazık ki Dünya [M15] ve Mars [R97] 'da meydana gelen galaksiler arası bir bela olan **öncelikli ters çevirme (priority inversion)** olarak bilinir!

Bir sistemde iki iş parçacığı olduğunu varsayalım. İş parçacığı 2 (T2) yüksek zamanlama önceliğine sahiptir ve iş parçacığı 1 (T1) daha düşük önceliğe sahiptir. Bu örnekte, her ikisi de çalıştırılabiliyorsa, CPU zamanlayıcısının her zaman T2'yi T1 üzerinden çalıştıracağını varsayalım; T1 yalnızca T2 bunu yapamadığında çalışır (örneğin, G / Ç'de T2 engellendiğinde).

Şimdi, sorun. T2'nin bir nedenden dolayı engellendiğini varsayalım. Böylece T1 koşar, bir dönüş kilidi alır ve kritik bir bölüme girer. T2 artık engellenmez hale gelir (belki de bir G / Ç tamamlandığı için) ve CPU zamanlayıcı bunu hemen planlar (böylece T1'in zamanlamasını kaldırır). T2 şimdi kilidi almaya çalışıyor ve alamadığı için (T1 kilidi tutuyor) dönmeye devam ediyor. Kilit bir döndürme kilidi olduğundan, T2 sonsuza kadar döner ve sistem asılır.

Sadece sıkma kilitlerinin kullanılmasından kaçınmak, ne yazık ki, ters çevirme probleminden kaçınmaz (ne yazık ki). Üç iş parçacığı düşünün, T1, T2 ve T3, T3 en yüksek önceliğe ve T1 en düşük önceliğe sahip. Şimdi T1'in bir kilit tuttuğunu hayal edin. T3 daha sonra başlar ve T1'den daha yüksek öncelikli olduğu için hemen çalışır (T1'i engeller). T3, T1'in tuttuğu kilidi almaya çalışır, ancak beklemeye takılır, çünkü T1 hala onu tutar. T2 çalışmaya başlarsa, T1'den daha yüksek önceliğe sahip olacak ve böylece çalışacaktır. T2'den daha yüksek önceliğe sahip olan T3, T2 çalıştığına göre hiçbir zaman çalışmayabilecek olan T1'i beklerken sıkıştı. Düşük T2 CPU'yu kontrol ederken, güçlü T3'ün çalışamaması üzücü değil mi? Yüksek önceliğe sahip olmak eskisi gibi değil.

Öncelikli ters çevirme sorununu çeşitli şekillerde ele alabilirsiniz. Döndürme kilitlerinin soruna neden olduğu özel durumda, döndürme kilitlerini kullanmaktan kaçınabilirsiniz (aşağıda daha ayrıntılı olarak açıklanmıştır). Daha genel olarak, daha düşük öncelikli bir iş parçacığını bekleyen daha yüksek öncelikli bir iş parçacığı, alt iş parçacığının önceliğini geçici olarak artırabilir, böylece çalışmasını ve tersine çevirmenin üstesinden gelmesini sağlayabilir, bu da **öncelikli devralma (priority inheritance)** olarak bilinen bir tekniktir. Son bir çözüm en basittir: tüm iş parçacıklarının aynı önceliğe sahip olduğundan emin olun

Bu örnekte birkaç ilginç şey yapıyoruz. İlk olarak, daha verimli bir kilit yapmak için eski test et ve ayarla fikrini açık bir kilit garsonları kuyruğuyla birleştiriyoruz. İkincisi, kilidi kimin alacağını kontrol etmek ve böylece açlıktan kaçınmak için bir sıra kullanıyoruz.

Korumanın nasıl kullanıldığını fark edebilirsiniz (Şekil 28.9, sayfa 16), temel olarak bayrağın etrafında bir döndürme kilidi olarak ve kilidin kullandığı kuyruk manipülasyonları olarak. Bu nedenle bu yaklaşım, dönüş beklemesini tamamen engellemez; Kilidi alırken veya serbest bırakırken bir iş parçacığı kesintiye uğrayabilir ve böylece diğer iş parçacıklarının dönmesine neden olabilir - bunun tekrar çalışmasını bekleyin. Bununla birlikte, eğirme için harcanan zaman oldukça sınırlıdır (kullanıcı tanımlı kritik bölüm yerine kilit ve kilit açma kodunun içindeki birkaç talimat) ve bu nedenle bu yaklaşım makul olabilir.

Ayrıca şunu da gözlemleyebilirsiniz: `lock()`, bir iş parçacığı kilidi alamadığında (zaten tutulduğunda), kendimizi bir sıraya eklemeye özen gösteririz (çağırarak `gettid()` iş parçacığı kimliğini almak için işlev geçerli iş parçacığı), guard değerini 0 olarak ayarlayın ve verim CPU. Okuyucu için bir soru: Koruma kilidinin serbest bırakılması `park()`, sonra gelirse ve daha önce gelmezse ne olur? İpucu: kötü bir şey.

Ayrıca, başka bir iş parçacığı uyandığında bayrağın 0'a ayarlanmadığını da algılayabilirsiniz. Bu neden? Eh, bu bir hata değil, bir zorunluluktur! Bir iş parçacığı uyandığında, `park()`öğesinden yeniden dönüyormuş gibi olur; ancak, kodun o noktasında bekçiyi tutmaz ve bu nedenle bayrağı 1 olarak ayarlamaya bile çalışamaz. Bu nedenle, kilidi doğrudan kilidi serbest bırakan iplikten onu alan bir sonraki ipliğe geçiririz; bayrak arada 0 olarak ayarlanmadı.

Son olarak, çağrıdan hemen önce çözümde algılanan yarış durumunu fark edebilirsiniz. Sadece yanlış zamanlama ile, kilit artık tutulmayana kadar uyuması gerektiği varsayılarak bir iş parçacığı `park()` etmek üzere olacaktır. O sırada başka bir iş parçacığına geçiş (örneğin, kilidi tutan bir iş parçacığı), örneğin o iş parçacığı kilidi serbest bırakırsa soruna yol açabilir. İlk ipliğin yanındaki sonraki park daha sonra sonsuza kadar uyuyacaktı (potansiyel olarak), bazen **uyanma/bekleme (wakeup/waiting race)** yarışı olarak adlandırılan bir sorun.

Solaris, üçüncü bir sistem çağrısı ekleyerek bu sorunu çözer: `setpark()`. Bu yordamı çağırarak, bir iş parçacığı park etmek üzere olduğunu gösterebilir. Daha sonra kesintiye uğrarsa ve park gerçekten çağrılmadan önce başka bir iş parçacığı park'ı çağırırsa, 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 devre dışı bırakmak için önlemler alabilir.

28.15 Farklı İşletim Sistemi, Farklı Destek (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 de benzer destek sağlar; ayrıntılar değişir.

```
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**

Örneğin Linux, Solaris arayüzüne benzer ancak daha fazla iç işlevsellik sağlayan bir **futeks (futex)** sağlar. Spesifik olarak, her futex onunla belirli bir fiziksel bellek konumunun yanı sıra çekirdek içi futex başına bir sıra ilişkilendirmiştir. Arayanlar, gerektiğinde uyumak ve uyanmak için futex çağrılarını (aşağıda açıklanmıştır) kullanabilir.

Özellikle, iki arama mevcuttur. `futex wait(address, expected)` çağrısı, `address` değerin `expected` değerine eşit olduğunu varsayarak çağıran iş parçacığını uyku moduna geçirir. Eşit değilse, çağrı hemen geri döner. Rutin `futex wake(address)`, sırada bekleyen bir iş parçacığını uyandırır. Bu çağrıların bir Linux muteksinde kullanımı Şekil 28.10'da gösterilmiştir (sayfa 19). `lowlevellock.h`

Bu kod parçacığı `lowlevellock.h` kütüphanesindeki h (gnu libc kütüphanesinin bir parçası) [L09] birkaç nedenden dolayı ilginçtir. İlk olarak, hem kilidin tutulup tutulmadığını (tamsayının yüksek biti) hem de kilit üzerindeki garsonların sayısını (diğer tüm bitler) izlemek için tek bir tamsayı kullanır. Böylece, kilit negatifse tutulur (çünkü yüksek bit ayarlanır ve bu bit tamsayının işaretini belirler).

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

özellikle kilit için herhangi bir çekişme olmadığında; Yalnızca bir iş parçacığı bir kilit alıp serbest bıraktığında, çok az iş yapılır (atomik bit kilitlenecek şekilde test edilir ve ayarlanır ve kilidi serbest bırakmak için bir atomik ekleme).

özellikle kilit için herhangi bir çekişme olmadığında; Yalnızca bir iş parçacığı bir kilit alıp serbest bıraktığında, çok az iş yapılır (atomik bit kilitlenecek şekilde test edilir ve ayarlanır ve kilidi serbest bırakmak için bir atomik ekleme).

Nasıl çalıştığını anlamak için bu "gerçek dünya" kilidinin geri kalanını çözüp çözemeyeceğinizi görün. Bunu yapın ve Linux kilitleme ustası olun veya

en azından bir kitap sana bir şey yapmanı söylediğinde dinleyen biri[3].

28.16 İki Fazlı Kilitler (Two-Phase Locks)

Son bir not: Linux yaklaşımı, yıllardır açık ve kapalı olarak kullanılan, en azından 1960'ların başlarında Dahm Kilitlerine kadar uzanan eski bir yaklaşımın lezzetine sahiptir. [M82] ve şimdi **iki fazlı kilit (two-phase lock)** olarak anılıyor. İki fazlı bir kilit, özellikle kilit serbest bırakılmak üzereyse, döndürmenin faydalı olabileceğini fark eder. Böylece ilk aşamada kilit, kilidi alabileceğini umarak bir süre döner.

Ancak, ilk döndürme aşamasında kilit alınmazsa, arayanın uyutulduğu ikinci bir aşama girilir ve yalnızca kilit daha sonra serbest kaldığında uyanır. Yukarıdaki Linux kilidi böyle bir kilidin şeklidir, ancak yalnızca bir kez döner; Bunun bir genellemesi, uyumak için **futeks(futex)** desteğini kullanmadan önce sabit bir süre boyunca bir döngüde dönebilir.

İki fazlı kilitler, **hibrit (hybrid)** bir yaklaşımın başka bir örneğidir; iki iyi fikri birleştirmek gerçekten daha iyi bir fikir verebilir. Tabii ki, olup olmadığı, donanım ortamı, iş parçacığı sayısı ve diğer iş yükü ayrıntıları dahil olmak üzere birçok şeye büyük ölçüde bağlıdır. Her zaman olduğu gibi, olası tüm kullanım durumları için iyi olan tek bir genel amaçlı kilit yapmak oldukça zordur.

28.17 Özet (Summary)

Yukarıdaki yaklaşım, bugünlerde gerçek kilitlerin nasıl oluşturulduğunu göstermektedir: bazı donanım desteği (daha güçlü bir talimat şeklinde) artı bazı işletim sistemi desteği (örneğin, Solaris'te `park()` ve `unpark()` ilkelleri veya Linux'ta futex şeklinde). Tabii ki, detaylar farklıdır ve bu tür kilitlemeyi gerçekleştirmek için tam kod genellikle oldukça ayarlanmıştır. Daha fazla ayrıntı görmek istiyorsanız Solaris veya Linux kod tabanlarına göz atın; büyüleyici bir okuma [L09, S09]. David ve ark.modern çok işlemcilerdeki kilitleme stratejilerinin karşılaştırılması için mükemmel bir çalışma [D + 13].

[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 Woodlands Resort, Pennsylvania, November 2013. *An excellent paper comparing many different ways to build locks using hardware primitives. Great to see how many ideas work on modern hardware.*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ödev (Simülasyon) (Homework (Simulation))

Bu program, `x86.py`, farklı iplik ara yapraklarının yarış koşullarına nasıl neden olduğunu veya bunlardan kaçındığını görmenizi sağlar. Programın nasıl çalıştığına dair kuyrukları kaldırmak için README sayfasına bakın ve aşağıdaki soruları yanıtlayın.

Sorular (Questions)

1. `flag.s` inceleyin. Bu kod, tek bir bellek bayrağıyla kilitlemeyi “uygular". derlemeyi anlayabiliyor musunuz?

   Bu kod, bir işlem parçacığının bellekteki bir bayrağı (flag) kullanarak diğer parçacıklardan bu alanı kullanmalarını engellemeyi amaçlar. Bu işleme "kilitleme" (lock) denir. Kilitleme, işlemler arasında veri çakışmalarını önlemek için kullanılır. Örneğin, bir parçacık bir değişkeni değiştirirken, diğer parçacıklar bu değişkeni okumaya çalıştıklarında veri çakışması oluşabilir. Bu nedenle, değiştirme işlemini tamamlamak için parçacık kilitlemeyi kullanır ve diğer parçacıklar bu alana erişemezler. Kilitleme, işlemler arasında serileştirilmiş bir şekilde çalışmayı sağlar.

2. Varsayılanlarla çalıştırdığınızda, `flag.s` çalışır mı i? Değişkenleri ve kayıtları izlemek için `-M` ve `-R` bayraklarını kullanın (ve değerlerini görmek için `-c` 'yi açın). `flag` ‘ta hangi değerin biteceğini tahmin edebilir misiniz?

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

4. Her iş parçacığı için `bx` değerini 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?

5. Şimdi `test-and-set.s` programına bakalım. İlk olarak, basit bir kilitleme ilkesi oluşturmak için `xchg` yönergesini kullanan kodu anlamaya çalışın. Kilit edinme nasıl yazılır? Kilidi açmaya ne dersin?
6. Şimdi kodu çalıştırın, kesme aralığının (`-i`) değerini tekrar değiştirin ve birkaç kez döngü yaptığınızdan emin olun. Kod her zaman beklendiği gibi çalışıyor mu? Bazen cpu'nun verimsiz kullanımına yol açar mı? Bunu nasıl ölçebilirsin?
7. Kilitleme kodunun belirli testlerini oluşturmak için `-P` bayrağını kullanın. Örneğin, kilidi ilk iş parçacığında yakalayan, ancak ikincisinde almaya çalışan bir zamanlama çalıştırın. Doğru olan olur mu? Başka ne test etmelisin?

8. Şimdi `peterson.s` koda bakalım.Peterson'ın algoritmasını uygulayan (metinde bir kenar çubuğunda belirtilmiştir). Kodu inceleyin ve bir anlam ifade edip edemeyeceğinizi görün.

9. Şimdi kodu farklı değerlerle çalıştırın `-i`. Ne tür farklı değerler görüyorsunuz? Kodun varsaydığı gibi iş parçacığı kimliklerini uygun şekilde

ayarladığınızdan emin olun (örneğin `-abx=0,bx=1`kullanarak).

10. Kodun çalıştığını "kanıtlamak" için zamanlamayı (`-P` bayrağıyla) kontrol edebilir misiniz? Beklemeniz gereken farklı durumlar nelerdir? Karşılıklı dışlanma ve çıkmazdan kaçınmayı düşünün.
11. Şimdi `ticket.s`. bilet kilidinin kodunu inceleyin. Bölümdeki kodla eşleşiyor mu? Ardından aşağıdaki bayraklarla çalıştırın: `-a bx=1000,bx=1000` (her iş parçacığının kritik bölümden 1000 kez geçmesine neden olur). Ne olduğuna dikkat edin; iplikler kilidi beklemek için çok zaman harcıyor mu? `test-and-set.s`
12. Daha fazla iş parçacığı eklerken kod nasıl davranır?
13. Şimdi `yield.s`, inceleyin.bir verim talimatının bir iş parçacığının CPU'nun kontrolünü sağlamasına izin verdiği (gerçekçi olarak, bu bir işletim sistemi ilkeli olacaktır, ancak basitlik için bir talimatın görevi yerine getirdiğini varsayıyoruz). `test-and-set.s` yapıldığı bir senaryo bulun.dönen döngüleri boşa harcar, ancak `yield` yapmaz. Kaç talimat kaydedilir? Bu tasarruflar hangi senaryolarda ortaya çıkıyor ?
14. Son olarak, `test-and-test-and-set.s`inceleyin. Bu kilit ne işe yarıyor? `test-and-set` kıyasla ne tür bir tasarruf sağlar?