27

Interlude: Thread API

Bu bölüm, iş parçacığı API'sinin ana bölümlerini kısaca kapsar. API'nin nasıl kullanılacağını gösterdiğimiz için her bölüm sonraki bölümlerde daha ayrıntılı olarak açıklanacaktır. Daha fazla ayrıntı çeşitli kitaplarda ve çevrimiçi kaynaklarda bulunabilir [B89, B97, B+96, K+96]. Sonraki bölümlerin, birçok örnekle birlikte, kilitler ve koşul değişkenleri kavramlarını daha yavaş tanıttığını belirtmeliyiz; bu bölüm bu nedenle daha iyi bir referans olarak kullanılır.

CRUX: İPLİKLER NASIL OLUŞTURULUR VE KONTROL EDİLİR

İşletim sistemi, iş parçacığı oluşturma ve kontrol için hangi arabirimleri sunmalıdır? Bu arayüzler, kullanım kolaylığının yanı sıra fayda sağlamak için nasıl tasarlanmalıdır?

27.1 Thread Oluşum

Çok iş parçacıklı bir program yazmak için yapmanız gereken ilk şey yeni iş parçacıkları oluşturmaktır ve bu nedenle bir tür iş parçacığı oluşturma arabirimi olmalıdır. POSIX'te kolaydır:

```
#include <pthread.h>
int
pthread_create(pthread_t       *thread,
         const pthread_attr_t *attr,
               void           *(*start_routine)(void*),
               void           *arg);
```

Bu bildirim biraz karmaşık görünebilir (özellikle C'de işlev işaretçileri kullanmadıysanız), ancak aslında o kadar da kötü değil. Dört argüman vardır: thread, attr, start yordamı ve argüman. Birincisi, iş parçacığı, pthread t türünde bir yapıya işaretçidir; bu yapıyı bu thread ile etkileşimde bulunmak için kullanacağız ve bu yüzden onu başlatmak için onu pthread create()'e geçirmemiz gerekiyor.

İkinci argüman, attr, bu iş parçacığının sahip olabileceği nitelikleri belirtmek için kullanılır. Bazı örnekler, yığın boyutunu ayarlamayı veya iş parçacığının zamanlama önceliği hakkındaki bilgileri içerir. Bir öznitelik, pthread attr init(); işlevine ayrı bir çağrıyla başlatılır; ayrıntılar için kılavuz sayfasına bakın. Ancak, çoğu durumda, varsayılanlar iyi olacaktır; bu durumda, NULL değerini içeri ileteceğiz.

Üçüncü argüman en karmaşık olanıdır, ancak gerçekten sadece şunu soruyor: Bu iş parçacığı hangi fonksiyonda çalışmaya başlamalı? C'de, buna bir **işlev işaretçisi (function pointer)** diyoruz ve bu bize aşağıdakilerin beklendiğini söylüyor: void * türünde tek bir argüman geçirilen bir işlev adı (başlangıç rutini), (başlangıç rutininden sonra parantez içinde belirtildiği gibi) ve void * türünde bir değer döndürür (yani, bir **void işaretçisi (void pointer**)).

Eğer bu rutin, bunun yerine bir tamsayı argümanı gerektiriyorsa işaretçi, bildirim şöyle görünür:

```
int pthread_create(..., // first two args are the same
                   void *(*start_routine)(int),
                   int  arg);
```

Bunun yerine, rutin bir argüman olarak geçersiz bir işaretçi aldıysa, ancak bir tamsayı döndürdüyse, şöyle görünür:

```
int pthread_create(..., // first two args are the same
                   int (*start_routine)(void *),
                   void *arg);
```

Son olarak, dördüncü argüman, arg, tam olarak iş parçacığının yürütülmeye başladığı fonksiyona iletilecek argümandır. Şunu sorabilirsiniz: neden bu boşluk işaretçilerine ihtiyacımız var? Cevap oldukça basit: fonksiyon başlatma rutininin argümanı olarak bir void işaretçisine sahip olmak, bu herhangi bir argüman türüne geçmemize izin verir; onu bir dönüş değeri olarak almak, iş parçacığının herhangi bir türde sonuç döndürmesine izin verir.

Şekil 27.1'deki bir örneğe bakalım. Burada sadece kendi tanımladığımız (myarg t) tek bir tipte paketlenmiş iki argüman iletilen bir iş parçacığı yaratırız. İş parçacığı bir kez oluşturulduktan sonra, argümanını beklediği türe çevirebilir ve böylece argümanları istenildiği gibi açabilir.

Ve işte burada! Bir iş parçacığı oluşturduğunuzda, programdaki mevcut tüm iş parçacıklarıyla aynı adres alanı içinde çalışan, kendi çağrı yığınıyla tamamlanmış, gerçekten başka bir canlı yürütme varlığınız olur. Eğlence böylece başlıyor!

27.2 Thread Tamamlama

Yukarıdaki örnek, bir iş parçacığının nasıl oluşturulacağını gösterir. Ancak, bir iş parçacığının tamamlanmasını beklemek isterseniz ne olur? Tamamlanmasını beklemek için özel bir şey yapmanız gerekiyor; özellikle, pthread join() rutinini çağırmalısınız.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  typedef struct {
5      int a;
6      int b;
7  } myarg_t;
8
9  void *mythread(void *arg) {
10     myarg_t *args = (myarg_t *) arg;
11     printf("%d %d\n", args->a, args->b);
12     return NULL;
13 }
14
15 int main(int argc, char *argv[]) {
16     pthread_t p;
17     myarg_t args = { 10, 20 };
18
19     int rc = pthread_create(&p, NULL, mythread, &args);
20     ...
21 }
```

Figure 27.1: **Creating a Thread**

Bu rutin iki argüman alır. Birincisi pthread t tipindedir ve hangi iş parçacığının bekleneceğini belirtmek için kullanılır. Bu değişken, iş parçacığı oluşturma rutini tarafından başlatılır (pthread create()'ye bir argüman olarak bir işaretçi ilettiğinizde); Etrafta tutarsanız, o iş parçacığının sona ermesini beklemek için kullanabilirsiniz.

İkinci argüman, geri almayı beklediğiniz dönüş değerinin bir göstergesidir. Rutin herhangi bir şeyi döndürebildiğinden, bir işaretçiyi void'e döndürmek için tanımlanmıştır; pthread join() yordamı argümanda iletilen değerin değerini değiştirdiğinde, yalnızca değerin kendisine değil, o değere bir işaretçi iletmeniz gerekir.

Başka bir örneğe bakalım (Şekil 27.2, sayfa 4). Kodda, tekrar tek bir iş parçacığı oluşturulur ve myarg t yapısı aracılığıyla birkaç argüman iletilir. Değerleri döndürmek için myret t tipi kullanılır. İş parçacığının çalışması bittiğinde, bekleyen ana iş parçacığı pthread join() rutin1'in içinde, sonra döner ve iş parçacığından döndürülen değerlere, yani myret t'de ne varsa erişebiliriz.

Bu örnek hakkında dikkat edilmesi gereken birkaç şey. İlk olarak, çoğu zaman tüm bu acı verici argümanları paketlemek ve açmak zorunda değiliz. Örneğin, argümansız bir iş parçacığı oluştursak, iş parçacığı oluşturulduğunda NULL'u bir argüman olarak iletebiliriz. Benzer şekilde, eğer dönüş değerini önemsemiyorsak, NULL'u pthread join()'e geçirebiliriz.

1Burada sarmalayıcı işlevleri kullandığımıza dikkat edin; özellikle, Malloc(), Pthread join() ve Pthread create()'i çağırırız, bunlar sadece benzer isimli küçük harf sürümlerini çağırır ve rutinlerin beklenmeyen bir şey döndürmediğinden emin olur.

```
typedef struct { int a; int b; } myarg_t;
typedef struct { int x; int y; } myret_t;

void *mythread(void *arg) {
    myret_t *rvals = Malloc(sizeof(myret_t));
    rvals->x = 1;
    rvals->y = 2;
    return (void *) rvals;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myret_t *rvals;
    myarg_t args = { 10, 20 };
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &rvals);
    printf("returned %d %d\n", rvals->x, rvals->y);
    free(rvals);
    return 0;
}
```

Figure 27.2: **Waiting for Thread Completion**

İkincisi, eğer sadece tek bir değer veriyorsak (örneğin, bir uzun int), onu bir argüman olarak paketlememiz gerekmez. Şekil 27.3 (sayfa 5) bir örnek gösterir. Bu durumda, argümanları paketlemek ve yapıların içinde değerler döndürmek zorunda olmadığımız için hayat biraz daha basittir.

Üçüncüsü, bir iş parçacığından değerlerin nasıl döndürüldüğü konusunda son derece dikkatli olunması gerektiğine dikkat etmeliyiz. Spesifik olarak, asla iş parçacığının çağrı yığınında tahsis edilen bir şeye atıfta bulunan bir işaretçi döndürmeyin. Eğer yaparsan, ne olacağını düşünüyorsun? (bir düşünün!) İşte Şekil 27.2'deki örnekten değiştirilmiş tehlikeli bir kod parçası örneği.

```
void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    myret_t oops; // ALLOCATED ON STACK: BAD!
    oops.x = 1;
    oops.y = 2;
    return (void *) &oops;
}
```

Bu durumda, oops değişkeni, mytread yığınına atanır. Bununla birlikte, geri döndüğünde, değer otomatik olarak serbest bırakılır (bu yüzden yığının kullanımı bu kadar kolaydır!) ve böylece bir işaretçiyi geri iletir.

```
void *mythread(void *arg) {
    long long int value = (long long int) arg;
    printf("%lld\n", value);
    return (void *) (value + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    long long int rvalue;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &rvalue);
    printf("returned %lld\n", rvalue);
    return 0;
}
```

Figür 27.3: **Bir Konuya Daha Basit Argüman Aktarma**

Artık serbest bırakılmış bir değişkene her türlü kötü sonuca yol açacaktır. Elbette, döndürdüğünü düşündüğünüz değerlerin çıktısını aldığınızda, muhtemelen (ama zorunlu olarak değil!) şaşıracaksınız. Deneyin ve kendiniz öğrenin2!

Son olarak, bir iş parçacığı oluşturmak için pthread_create() ve ardından hemen pthread_join() çağrısının kullanılmasının, bir iş parçacığı oluşturmanın oldukça garip bir yolu olduğunu fark edebilirsiniz. Aslında bu görevi tam olarak yerine getirmenin daha kolay bir yolu var; buna prosedür çağrısı denir. Açıkçası, genellikle birden fazla iş parçacığı oluşturup tamamlanmasını bekleyeceğiz, aksi takdirde iş parçacıklarını kullanmanın pek bir amacı yoktur Çok iş parçacıklı tüm kodların birleştirme yordamını kullanmadığına dikkat etmeliyiz. Örneğin, çok iş parçacıklı bir web sunucusu bir dizi çalışan iş parçacığı oluşturabilir ve ardından istekleri kabul etmek ve bunları süresiz olarak işçilere iletmek için ana iş parçacığını kullanabilir. Bu tür uzun ömürlü programların bu nedenle katılması gerekmeyebilir. Bununla birlikte, belirli bir görevi (paralel olarak) yürütmek için iş parçacıkları oluşturan bir paralel program, çıkmadan veya bir sonraki hesaplama aşamasına geçmeden önce tüm bu tür çalışmaların tamamlandığından emin olmak için büyük olasılıkla birleştirmeyi kullanır.

27.3 Kilitler

İş parçacığı oluşturma ve birleştirmenin ötesinde, muhtemelen POSIX iş parçacığı kitaplığı tarafından sağlanan bir sonraki en kullanışlı işlevler kümesi, kilitler aracılığıyla kritik bir bölüme karşılıklı dışlama sağlamaya yönelik işlevlerdir. Bu amaç için kullanılacak en temel rutin çifti aşağıdakiler tarafından sağlanır:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

2Neyse ki, böyle bir kod yazdığınızda derleyici gcc muhtemelen şikayet edecektir, bu da derleyici uyarılarına dikkat etmeniz için başka bir nedendir.

Rutinlerin anlaşılması ve kullanılması kolay olmalıdır. Kritik bir bölüm olan ve bu nedenle doğru çalışmayı sağlamak için korunması gereken bir kod bölgeniz olduğunda, kilitler oldukça kullanışlıdır. Muhtemelen kodun nasıl göründüğünü hayal edebilirsiniz:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // veya kritik bölümünüz ne olursa olsun
pthread_mutex_unlock(&lock);
```

Kodun amacı şudur: pthread mutex lock() çağrıldığında kilidi başka bir iş parçacığı tutmuyorsa, iş parçacığı kilidi alacak ve kritik bölüme girecektir. Başka bir iş parçacığı gerçekten kilidi tutuyorsa, kilidi almaya çalışan iş parçacığı, kilidi elde edene kadar aramadan geri dönmeyecektir (kilidi tutan iş parçacığının, kilit açma çağrısı yoluyla onu serbest bıraktığı anlamına gelir). Tabii ki, belirli bir zamanda kilit edinme fonksiyonu içinde bekleyen birçok iş parçacığı sıkışmış olabilir; ancak, yalnızca kilidin elde edildiği iş parçacığı, kilidi açmayı çağırmalıdır.

Ne yazık ki, bu kod iki önemli şekilde bozulabilir. İlk sorun, uygun başlatma eksikliğidir. Başlamak için doğru değerlere sahip olduklarını garanti etmek ve böylece kilitleme ve kilit açma çağrıldığında istendiği gibi çalışmasını sağlamak için tüm kilitler düzgün bir şekilde başlatılmalıdır.

POSIX iş parçacıkları ile kilitleri başlatmanın iki yolu vardır. Bunu yapmanın bir yolu, aşağıdaki gibi PTHREAD MUTEX INITIALIZER kullanmaktır

```
:pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Bunu yapmak, kilidi varsayılan değerlere ayarlar ve böylece kilidi kullanılabilir hale getirir. Bunu yapmanın dinamik yolu (yani çalışma zamanında), aşağıdaki gibi pthread mutex init() çağrısı yapmaktır:

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Bu rutinin ilk argümanı kilidin adresidir, ikincisi ise isteğe bağlı bir nitelikler kümesidir. Nitelikler hakkında daha fazlasını okuyun; NULL'u geçmek sadece varsayılanları kullanır. Her iki şekilde de çalışır, ancak genellikle dinamik (ikinci) yöntemi kullanırız. Kilitle işiniz bittiğinde, pthread mutex destroy() işlevine karşılık gelen bir çağrının da yapılması gerektiğini unutmayın; tüm ayrıntılar için kılavuz sayfasına bakın.

Yukarıdaki kodla ilgili ikinci sorun, kilitleme ve kilit açma çağrılırken hata kodlarının kontrol edilememesidir. UNIX sisteminde çağırdığınız hemen hemen her kütüphane rutini gibi, bu rutinler de başarısız olabilir! Kodunuz hata kodlarını düzgün bir şekilde kontrol etmezse, hata sessizce gerçekleşir ve bu durumda kritik bir bölüme birden fazla iş parçacığının girmesine izin verebilir. Asgari olarak, Şekil 27.4'te (sayfa 7) gösterildiği gibi rutinin başarılı olduğunu iddia eden sarmalayıcılar kullanın; bir şeyler ters gittiğinde basitçe çıkamayan daha karmaşık (oyuncak olmayan) programlar, hata olup olmadığını kontrol etmeli ve bir çağrı başarılı olmadığında uygun bir şey yapmalıdır.

```
// Kodu temiz tutar; başarısızlık durumunda yalnızca exit() OK
kullanın void Pthread_mutex_lock(pthread_mutex_t *mutex)
{
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

Figure 27.4: **Örnek Bir Sarıcı**

Kilitleme ve kilit açma rutinleri, sistem içindeki tek rutinler değildir.

kilitlerle etkileşim için pthreads kitaplığı. Diğer iki ilgi rutin:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

Bu iki çağrı, kilit alımında kullanılır. Kilit zaten tutulmuşsa, trylock sürümü başarısız olur; Kilit edinmenin zaman ayarlı kilit sürümü, hangisi önce gerçekleşirse, bir zaman aşımından sonra veya kilidi aldıktan sonra geri döner. Böylece, sıfır zaman aşımına sahip zaman aşımı, trylock durumunda dejenere olur. Bu sürümlerin her ikisinden de genellikle kaçınılmalıdır; ancak, sonraki bölümlerde göreceğimiz gibi.

27.4 Koşul Değişkenleri

Herhangi bir iş parçacığı kitaplığının diğer ana bileşeni ve kesinlikle POSIX iş parçacıklarında olduğu gibi, bir koşul değişkeninin varlığıdır. Koşul değişkenleri, bir iş parçacığı devam etmeden önce diğerinin bir şey yapmasını bekliyorsa, iş parçacıkları arasında bir tür sinyalleşmenin gerçekleşmesi gerektiğinde yararlıdır. Bu şekilde etkileşimde bulunmak isteyen programlar tarafından iki temel rutin kullanılır:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

Bir koşul değişkenini kullanmak için, ek olarak bu koşulla ilişkili bir kilidin olması gerekir. Yukarıdaki rutinlerden herhangi birini çağırırken, bu kilit tutulmalıdır.

İlk rutin, pthread cond wait(), çağıran iş parçacığını uyku moduna geçirir ve bu nedenle, genellikle programda şu anda uyuyan iş parçacığının umursadığı bir şey değiştiğinde, başka bir iş parçacığının sinyal vermesini bekler. Tipik bir kullanım şöyle görünür:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

Bu kodda, ilgili kilit ve koşul3'ün başlatılmasından sonra bir iş parçacığı, hazır değişkenin henüz sıfırdan farklı bir şeye ayarlanıp ayarlanmadığını kontrol eder. Değilse, iş parçacığı başka bir iş parçacığı onu uyandırana kadar uyumak için bekleme rutinini çağırır.

Başka bir iş parçacığında çalışacak bir iş parçacığını uyandırma kodu şöyle görünür:

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

Bu kod dizisi hakkında dikkat edilmesi gereken birkaç şey. İlk olarak, sinyal verirken (ve global değişkeni hazır hale getirirken), her zaman kilidin tutulduğundan emin oluruz. Bu, kodumuza yanlışlıkla bir yarış koşulu eklemememizi sağlar.

İkinci olarak, bekleme çağrısının ikinci parametresi olarak bir kilit aldığını, oysa sinyal çağrısının yalnızca bir koşul aldığını fark edebilirsiniz. Bu farkın nedeni, bekleme çağrısının, çağrı dizisini uyku moduna geçirmenin yanı sıra, söz konusu çağrıyı uyku moduna geçirirken kilidi serbest bırakmasıdır. Olmadığını hayal edin: diğer iş parçacığı kilidi nasıl ele geçirebilir ve uyanması için sinyal verebilir? Ancak, uyandırıldıktan sonra geri dönmeden önce, pthread cond wait() kilidi yeniden alır, böylece bekleme dizisinin başlangıcındaki kilit edinme ile sonundaki kilit serbest bırakma arasında bekleyen iş parçacığının her zaman çalıştığından emin olur, kilidi tutar.

Son bir tuhaflık: bekleyen iş parçacığı, basit bir if ifadesi yerine bir while döngüsündeki koşulu yeniden kontrol eder. Gelecekteki bir bölümde koşul değişkenlerini incelerken bu konuyu ayrıntılı olarak tartışacağız, ancak genel olarak bir süre döngüsü kullanmak basit ve güvenli bir şeydir. Durumu yeniden kontrol etmesine rağmen (belki biraz ek yük ekleyerek), bekleyen bir iş parçacığını sahte bir şekilde uyandırabilecek bazı pthread uygulamaları vardır; böyle bir durumda, tekrar kontrol etmeden bekleyen iş parçacığı, durumun değişmediği halde değiştiğini düşünerek devam edecektir. Bu nedenle, uyanmayı mutlak bir gerçek yerine bir şeylerin değişmiş olabileceğine dair bir ipucu olarak görmek daha güvenlidir.

Bazen bir koşul değişkeni ve ilişkili kilit yerine iki iş parçacığı arasında sinyal göndermek için basit bir bayrak kullanmanın cazip olduğunu unutmayın. Örneğin, bekleme kodunda daha çok şöyle görünmesi için yukarıdaki bekleme kodunu yeniden yazabiliriz:

```
while (ready == 0)
    ; // spin
```

İlişkili sinyalizasyon kodu şöyle görünür:

```
ready = 1;
```

3Biri statik başlatıcı PTHREAD COND INITIALIZER yerine pthread cond init() (ve pthread cond destroy()) kullanabilir. Bu daha fazla iş gibi mi geliyor?

Aşağıdaki nedenlerden dolayı bunu asla yapmayın. İlk olarak, çoğu durumda düşük performans gösterir (uzun süre döndürmek yalnızca CPU döngülerini boşa harcar). İkincisi, hataya açıktır. Son araştırmaların [X+10] gösterdiği gibi, threadler arasında senkronizasyon için bayrakları (yukarıdaki gibi) kullanırken hata yapmak şaşırtıcı derecede kolaydır; bu çalışmada, bu ad hoc senkronizasyonların kabaca yarısı hatalıydı! Tembel olmayın; Bunu yapmadan kurtulabileceğinizi düşündüğünüzde bile koşul değişkenlerini kullanın.

Koşul değişkenleri kafa karıştırıcı geliyorsa, çok fazla endişelenmeyin (henüz) - sonraki bölümde bunları ayrıntılı olarak ele alacağız. O zamana kadar onların var olduğunu bilmek ve nasıl ve neden kullanıldıklarına dair bir fikir sahibi olmak yeterli olacaktır.

27.5 Derleme ve Çalıştırma

Bu bölümdeki tüm kod örneklerinin kurulumu ve çalıştırılması nispeten kolaydır. Bunları derlemek için kodunuza pthread.h başlığını eklemelisiniz. Bağlantı satırında, -pthread bayrağını ekleyerek pthreads kitaplığıyla da açıkça bağlantı kurmalısınız.

Örneğin, çok iş parçacıklı basit bir programı derlemek için yapmanız gereken tek şey şudur:

```
prompt> gcc -o main main.c -Wall -pthread
```

main.c, pthreads başlığını içerdiği sürece, eşzamanlı bir programı başarıyla derlediniz. Her zamanki gibi çalışıp çalışmadığı tamamen farklı bir konudur.

27.6 Özet

İplik oluşturma, kilitler aracılığıyla karşılıklı dışlama oluşturma ve koşul değişkenleri aracılığıyla sinyalleme ve bekleme dahil olmak üzere pthread kitaplığının temellerini tanıttık. Sabır ve büyük özen dışında, sağlam ve verimli çok iş parçacıklı kod yazmak için fazla bir şeye ihtiyacınız yok!

Şimdi bölümü, çok iş parçacıklı kod yazarken işinize yarayabilecek bir dizi ipucu ile bitiriyoruz (ayrıntılar için sonraki sayfaya bakın). API'nin ilginç olan başka yönleri de vardır; Daha fazla bilgi istiyorsanız, tüm arayüzü oluşturan yüzden fazla API'yi görmek için bir Linux sisteminde man -k pthread yazın. Bununla birlikte, burada tartışılan temel bilgiler, karmaşık (ve umarım, doğru ve performanslı) çok iş parçacıklı programlar oluşturmanıza olanak sağlamalıdır. İş parçacıklarının zor kısmı API'ler değil, eşzamanlı programları nasıl oluşturduğunuza dair karmaşık mantıktır. Daha fazlasını öğrenmek için okumaya devam edin.

ASIDE: THREAD API YÖNERGELERİ

Çok iş parçacıklı bir program oluşturmak için POSIX iş parçacığı kitaplığını (veya gerçekten herhangi bir iş parçacığı kitaplığını) kullanırken hatırlamanız gereken birkaç küçük ama önemli şey vardır. Bunlar:

- **Basit tutun.** Her şeyden önce, iş parçacıkları arasında kilitlenecek veya sinyal verilecek herhangi bir kod mümkün olduğunca basit olmalıdır. Zor iş parçacığı etkileşimleri hatalara yol açar.
- • **İş parçacığı etkileşimlerini en aza indirin.** İş parçacıklarının etkileşime girme yollarının sayısını minimumda tutmaya çalışın. Her etkileşim dikkatli bir şekilde düşünülmeli ve denenmiş ve doğru yaklaşımlarla oluşturulmalıdır (bunların birçoğunu sonraki bölümlerde öğreneceğiz).
- • **Kilitleri ve koşul değişkenlerini başlatın**. Bunu yapmamak, bazen çalışan ve bazen çok garip şekillerde başarısız olan koda yol açacaktır.
- • **İade kodlarınızı kontrol edin.** Elbette, yaptığınız herhangi bir C ve UNIX programlamasında, her bir dönüş kodunu kontrol ediyor olmalısınız ve bu burada da geçerlidir. Bunu yapmamak, sizi (a) çığlık atmanıza, (b) saçınızın bir kısmını çekmenize veya (c) her ikisini birden yapmanıza neden olacak şekilde tuhaf ve anlaşılması zor davranışlara yol açacaktır.
- • **Dizilere nasıl argüman ilettiğiniz ve dizilerden nasıl değerler döndürdüğünüz konusunda dikkatli olun.** Özellikle, yığında tahsis edilmiş bir değişkene bir başvuru ilettiğinizde, muhtemelen yanlış bir şey yapıyorsunuzdur.
- • **Her iş parçacığının kendi yığını vardır.** Yukarıdaki nokta ile ilgili olarak, lütfen her iş parçacığının kendi yığınına sahip olduğunu unutmayın. Bu nedenle, bir iş parçacığının yürüttüğü bir işlevin içinde yerel olarak tahsis edilmiş bir değişkeniniz varsa, esasen o iş parçacığına özeldir; başka hiçbir iş parçacığı (kolayca) erişemez. İş parçacıkları arasında veri paylaşmak için, değerlerin yığında veya başka bir şekilde genel olarak erişilebilir bir yerel ayarda olması gerekir**.**
- **İplikler arasında sinyal vermek için her zaman koşul değişkenlerini kullanın**. Basit bir bayrak kullanmak çoğu zaman cazip gelse de, bunu yapmayın.
- **Kılavuz sayfalarını kullanın**. Özellikle Linux'ta, pthread kılavuz sayfaları oldukça bilgilendiricidir ve burada sunulan nüansların çoğunu, genellikle daha da ayrıntılı olarak tartışır. Onları dikkatlice okuyun!

References

[B89] "An Introduction to Programming with Threads" by Andrew D. Birrell. DEC Technical Report, January, 1989. Available: https://birrell.org/andrew/papers/035-Threads.pdf *A classic but older introduction to threaded programming. Still a worthwhile read, and freely available.*

[B97] "Programming with POSIX Threads" by David R. Butenhof. Addison-Wesley, May 1997. *Another one of these books on threads.*

[B+96] "PThreads Programming: by A POSIX Standard for Better Multiprocessing. " Dick Buttlar, Jacqueline Farrell, Bradford Nichols. O'Reilly, September 1996 *A reasonable book from the excellent, practical publishing house O'Reilly. Our bookshelves certainly contain a great deal of books from this company, including some excellent offerings on Perl, Python, and Javascript (particularly Crockford's "Javascript: The Good Parts".)*

[K+96] "Programming With Threads" by Steve Kleiman, Devang Shah, Bart Smaalders. Prentice Hall, January 1996. *Probably one of the better books in this space. Get it at your local library. Or steal it from your mother. More seriously, just ask your mother for it – she'll let you borrow it, don't worry.*

[X+10] "Ad Hoc Synchronization Considered Harmful" by Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, Zhiqiang Ma. OSDI 2010, Vancouver, Canada. *This paper shows how seemingly simple synchronization code can lead to a surprising number of bugs. Use condition variables and do the signaling correctly!*

Ödev (Kod)

Bu bölümde, bazı basit çok kanallı programlar yazacağız ve bu programlardaki sorunları bulmak için helgrind adlı özel bir araç kullanacağız.

Programların nasıl oluşturulacağı ve helgrind'in nasıl çalıştırılacağı ile ilgili ayrıntılar için ödev indirme dosyasındaki README'yi okuyun.

Questions

1. Önce main-race.c'yi oluşturun. Koddaki (umarız bariz) veri yarışını görebilmek için kodu inceleyin. Şimdi helgrind'i çalıştırın (valgrind yazarak--tool=helgrind main-race) komutlarını, yarışı nasıl rapor ettiğini görmek için yazın. Doğru kod satırlarını gösteriyor mu? Size başka hangi bilgileri veriyor?

   Valgrind, kullanıcı kodunda veri yarışı olmadığı zamanlar da dahil, pthread kitaplıklarındaki veri yarışlarını bildirir. Aşağıdaki komutlar, veri yarışını tanımlayan bir uyarıyı ekrana vermek içindir:

   clang main-race.c -o main-race -fsanitize=thread -fPIE -g -Wall

   ./main-race

Valgrind ayrı kilit sıralama sorunlarından kaynaklanan olası kilitlenmeleri ve POSIX , API'sinin kötüye kullanımını raporlar

```
==33101== Helgrind, a thread error detector
==33101== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==33101== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==33101== Command: ./main-race
==33101==
==33101== ---Thread-Announcement------------------------------------------
==33101==
==33101== Thread #1 is the program's root thread
==33101==
==33101== ---Thread-Announcement------------------------------------------
==33101==
==33101== Thread #2 was created
==33101==    at 0x49AE122: clone (clone.S:71)
==33101==    by 0x48732EB: create_thread (createthread.c:101)
==33101==    by 0x4874E0F: pthread_create@@GLIBC_2.2.5 (pthread_create.c:817)
==33101==    by 0x48475DA: pthread_create_WRK (hg_intercepts.c:445)
==33101==    by 0x4848AF5: pthread_create@* (hg_intercepts.c:478)
==33101==    by 0x109209: main (in /home/berkay/Desktop/ostep-homework/threads-api/main-race)
==33101==
==33101== ----------------------------------------------------------------
==33101==
==33101== Possible data race during read of size 4 at 0x10C014 by thread #1
==33101== Locks held: none
==33101==    at 0x10922D: main (in /home/berkay/Desktop/ostep-homework/threads-api/main-race)
==33101==
==33101== This conflicts with a previous write of size 4 by thread #2
==33101== Locks held: none
==33101==    at 0x1091BE: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-race)
==33101==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==33101==    by 0x4874608: start_thread (pthread_create.c:477)
==33101==    by 0x49AE132: clone (clone.S:95)
==33101==  Address 0x10c014 is 0 bytes inside data symbol "balance"
==33101==
==33101== ----------------------------------------------------------------
==33101==
==33101== Possible data race during write of size 4 at 0x10C014 by thread #1
==33101== Locks held: none
==33101==    at 0x109236: main (in /home/berkay/Desktop/ostep-homework/threads-api/main-race)
==33101==
==33101== This conflicts with a previous write of size 4 by thread #2
==33101== Locks held: none
==33101==    at 0x1091BE: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-race)
==33101==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==33101==    by 0x4874608: start_thread (pthread_create.c:477)
==33101==    by 0x49AE132: clone (clone.S:95)
==33101==  Address 0x10c014 is 0 bytes inside data symbol "balance"
==33101==
==33101==
==33101== Use --history-level=approx or =none to gain increased speed, at
==33101== the cost of reduced accuracy of conflicting-access information
==33101== For lists of detected and suppressed errors, rerun with: -s
==33101== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

```
==35691== Helgrind, a thread error detector
==35691== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==35691== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==35691== Command: ./main-race
==35691==
==35691== ---Thread-Announcement------------------------------------------
==35691==
==35691== Thread #1 is the program's root thread
==35691==
==35691== ---Thread-Announcement------------------------------------------
==35691==
==35691== Thread #2 was created
==35691==    at 0x49AE122: clone (clone.S:71)
==35691==    by 0x48732EB: create_thread (createthread.c:101)
==35691==    by 0x4874E0F: pthread_create@@GLIBC_2.2.5 (pthread_create.c:817)
==35691==    by 0x48475DA: pthread_create_WRK (hg_intercepts.c:445)
==35691==    by 0x4848AF5: pthread_create@* (hg_intercepts.c:478)
==35691==    by 0x109209: main (in /home/berkay/Desktop/ostep-homework/threads-api/main-race)
==35691==
==35691== ----------------------------------------------------------------
==35691==
==35691== Possible data race during read of size 4 at 0x10C014 by thread #1
==35691== Locks held: none
==35691==    at 0x10922D: main (in /home/berkay/Desktop/ostep-homework/threads-api/main-race)
==35691==
==35691== This conflicts with a previous write of size 4 by thread #2
==35691== Locks held: none
==35691==    at 0x1091BE: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-race)
==35691==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==35691==    by 0x4874608: start_thread (pthread_create.c:477)
==35691==    by 0x49AE132: clone (clone.S:95)
==35691==  Address 0x10c014 is 0 bytes inside data symbol "balance"
==35691==
==35691== ----------------------------------------------------------------
==35691==
==35691== Possible data race during write of size 4 at 0x10C014 by thread #1
==35691== Locks held: none
==35691==    at 0x109236: main (in /home/berkay/Desktop/ostep-homework/threads-api/main-race)
==35691==
==35691== This conflicts with a previous write of size 4 by thread #2
==35691== Locks held: none
==35691==    at 0x1091BE: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-race)
==35691==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==35691==    by 0x4874608: start_thread (pthread_create.c:477)
==35691==    by 0x49AE132: clone (clone.S:95)
==35691==  Address 0x10c014 is 0 bytes inside data symbol "balance"
==35691==
==35691==
==35691== Use --history-level=approx or =none to gain increased speed, at
==35691== the cost of reduced accuracy of conflicting-access information
==35691== For lists of detected and suppressed errors, rerun with: -s
==35691== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

2. Rahatsız edici kod satırlarından birini kaldırdığınızda ne olur? Şimdi, paylaşılan değişkendeki güncellemelerden birinin etrafına ve ardından her ikisinin çevresine bir kilit ekleyin. Helgrind bu vakaların her birinde ne rapor eder?

Yukarıdaki gibi Valgrind kullanmamız gerekir. Threadlerin birinden erişimin kaldırılması ve TSan'ın çalışması hiçbir çıktı üretmez ve 0 ile geri döner.

Bir kilit ekledikten sonra, değişkeni güncellediğimizde ve bu kilidin thredlerden biri tarafından tuttulduğunu tanımlayan bir uyarı ekran verir. Bu durumda TSan 134 çıkış kodu ile geri döner.

3. Now Şimdi main-deadlock.c'ye bakalım. Kodu inceleyin. Bu kodun **kilitlenme (deadlock)** olarak bilinen bir sorunu var (gelecek bir bölümde çok daha derinlemesine tartışacağız). Ne gibi bir sorunu olabileceğini görebiliyor musun?

Threadler, ilk kilitlerini aynı anda alırlar ise, bir deadlock durumu oluşur. Her ikisi de sahip olamadıkları kilitleri beklerken deadlock duruma düşerler

```
==36720== Helgrind, a thread error detector
==36720== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==36720== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==36720== Command: ./main-deadlock
==36720==
==36720== ---Thread-Announcement------------------------------------------
==36720==
==36720== Thread #3 was created
==36720==    at 0x49AE122: clone (clone.S:71)
==36720==    by 0x48732EB: create_thread (createthread.c:101)
==36720==    by 0x4874E0F: pthread_create@@GLIBC_2.2.5 (pthread_create.c:817)
==36720==    by 0x48475DA: pthread_create_WRK (hg_intercepts.c:445)
==36720==    by 0x4848AF5: pthread_create@* (hg_intercepts.c:478)
==36720==    by 0x10939F: main (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock)
==36720==
==36720== ----------------------------------------------------------------
==36720==
==36720== Thread #3: lock order "0x10C040 before 0x10C080" violated
==36720==
==36720== Observed (incorrect) order is: acquisition of lock at 0x10C080
==36720==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==36720==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==36720==    by 0x109269: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock)
==36720==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==36720==    by 0x4874608: start_thread (pthread_create.c:477)
==36720==    by 0x49AE132: clone (clone.S:95)
==36720==
==36720==  followed by a later acquisition of lock at 0x10C040
==36720==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==36720==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==36720==    by 0x109298: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock)
==36720==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==36720==    by 0x4874608: start_thread (pthread_create.c:477)
==36720==    by 0x49AE132: clone (clone.S:95)
==36720==
==36720== Required order was established by acquisition of lock at 0x10C040
==36720==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==36720==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==36720==    by 0x10920B: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock)
==36720==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==36720==    by 0x4874608: start_thread (pthread_create.c:477)
==36720==    by 0x49AE132: clone (clone.S:95)
==36720==
==36720==  followed by a later acquisition of lock at 0x10C080
==36720==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==36720==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==36720==    by 0x10923A: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock)
==36720==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==36720==    by 0x4874608: start_thread (pthread_create.c:477)
==36720==    by 0x49AE132: clone (clone.S:95)
==36720==
==36720==  Lock at 0x10C040 was first observed
==36720==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==36720==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==36720==    by 0x10920B: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock)
==36720==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==36720==    by 0x4874608: start_thread (pthread_create.c:477)
==36720==    by 0x49AE132: clone (clone.S:95)
==36720==  Address 0x10c040 is 0 bytes inside data symbol "m1"
==36720==
==36720==  Lock at 0x10C080 was first observed
==36720==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==36720==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==36720==    by 0x10923A: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock)
==36720==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==36720==    by 0x4874608: start_thread (pthread_create.c:477)
==36720==    by 0x49AE132: clone (clone.S:95)
==36720==  Address 0x10c080 is 0 bytes inside data symbol "m2"
==36720==
==36720==
==36720==
==36720== Use --history-level=approx or =none to gain increased speed, at
==36720== the cost of reduced accuracy of conflicting-access information
==36720== For lists of detected and suppressed errors, rerun with: -s
==36720== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 7 from 7)
```

4. Şimdi bu kod üzerinde helgrind'i çalıştırın. Helgrind ne rapor ediyor(main-deadlock.c dosyasında)?

   TSan'I çalıştırdığımızdaaşağıdaki gibi bir rapor vercektir:

   make main-deadlock

   TSAN_OPTIONS=detect_deadlocks=1 ./main-deadlock

   ----verdiği rapor görüntüsü

WARNING: ThreadSanitizer: lock-order-inversion (potential deadlock) (pid=71488)
Cycle in lock order graph: M23 (0x0001001f4088) => M25 (0x0001001f40c8) => M23

Ayrıca kiltilerin nerede alındığının ve threadlerin oluşturulduğu konumu da raporlar.

5. Şimdi helgrind'I, main-deadlock-global.c üzerinde çalıştırın. Kodu inceleyin; main-deadlock.c ile aynı sorunu yaşıyor mu? Helgrind aynı hatayı bildirmeli mi? Bu size helgrind gibi araçlar hakkında ne söylüyor?

   Aynı sorunu içermez. Global kilit, olası bir ölümcül kilitlenmeyi engeller. Ama TSan ekrana şuna benzer bir uyarıyı verir:

   WARNING: ThreadSanitizer: lock-order-inversion (potential deadlock) (pid=71488)
   Cycle in lock order graph: M23 (0x0001001f4088) => M25 (0x0001001f40c8) => M23

```
==37886== Helgrind, a thread error detector
==37886== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==37886== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==37886== Command: ./main-deadlock-global
==37886==
==37886== ---Thread-Announcement------------------------------------------
==37886==
==37886== Thread #3 was created
==37886==    at 0x49AE122: clone (clone.S:71)
==37886==    by 0x48732EB: create_thread (createthread.c:101)
==37886==    by 0x4874E0F: pthread_create@@GLIBC_2.2.5 (pthread_create.c:817)
==37886==    by 0x48475DA: pthread_create_WRK (hg_intercepts.c:445)
==37886==    by 0x4848AF5: pthread_create@* (hg_intercepts.c:478)
==37886==    by 0x1093FD: main (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock-global)
==37886==
==37886== ----------------------------------------------------------------
==37886==
==37886== Thread #3: lock order "0x10C080 before 0x10C0C0" violated
==37886==
==37886== Observed (incorrect) order is: acquisition of lock at 0x10C0C0
==37886==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==37886==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==37886==    by 0x109298: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock-global)
==37886==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==37886==    by 0x4874608: start_thread (pthread_create.c:477)
==37886==    by 0x49AE132: clone (clone.S:95)
==37886==
==37886==  followed by a later acquisition of lock at 0x10C080
==37886==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==37886==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==37886==    by 0x1092C7: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock-global)
==37886==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==37886==    by 0x4874608: start_thread (pthread_create.c:477)
==37886==    by 0x49AE132: clone (clone.S:95)
==37886==
==37886== Required order was established by acquisition of lock at 0x10C080
==37886==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==37886==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==37886==    by 0x10923A: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock-global)
==37886==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==37886==    by 0x4874608: start_thread (pthread_create.c:477)
==37886==    by 0x49AE132: clone (clone.S:95)
==37886==
==37886==  followed by a later acquisition of lock at 0x10C0C0
==37886==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==37886==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==37886==    by 0x109269: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock-global)
==37886==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==37886==    by 0x4874608: start_thread (pthread_create.c:477)
==37886==    by 0x49AE132: clone (clone.S:95)
==37886==
==37886==  Lock at 0x10C080 was first observed
==37886==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==37886==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==37886==    by 0x10923A: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock-global)
==37886==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==37886==    by 0x4874608: start_thread (pthread_create.c:477)
==37886==    by 0x49AE132: clone (clone.S:95)
==37886==  Address 0x10c080 is 0 bytes inside data symbol "m1"
==37886==
==37886==  Lock at 0x10C0C0 was first observed
==37886==    at 0x48445BD: mutex_lock_WRK (hg_intercepts.c:942)
==37886==    by 0x4848EF2: pthread_mutex_lock (hg_intercepts.c:958)
==37886==    by 0x109269: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-deadlock-global)
==37886==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==37886==    by 0x4874608: start_thread (pthread_create.c:477)
==37886==    by 0x49AE132: clone (clone.S:95)
==37886==  Address 0x10c0c0 is 0 bytes inside data symbol "m2"
==37886==
==37886==
==37886==
==37886== Use --history-level=approx or =none to gain increased speed, at
==37886== the cost of reduced accuracy of conflicting-access information
==37886== For lists of detected and suppressed errors, rerun with: -s
==37886== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 7 from 7)
```

6. Şimdi main-signal.c'ye bakalım. Bu kod, çocuğun bittiğini ve ebeveynin artık devam edebileceğini belirtmek için bir değişken kullanır. Bu kod neden verimsizdir? (özellikle alt threadin tamamlanması uzun zaman alıyorsa, ebeveyn zamanını ne yaparak geçiriyor?)

Ebeveyn threadi, çocuk threadinin işini bitirmesine kadar geçen sürede koşul değişkenini kontrol ederek bekler. Bu kontrol işlemleri işlemci tarafından boşuna bir yer israfı olarak görülür

```
==38625== Helgrind, a thread error detector
==38625== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==38625== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==38625== Command: ./main-signal
==38625==
this should print first
==38625== ---Thread-Announcement------------------------------------------
==38625==
==38625== Thread #1 is the program's root thread
==38625==
==38625== ---Thread-Announcement------------------------------------------
==38625==
==38625== Thread #2 was created
==38625==    at 0x49AE122: clone (clone.S:71)
==38625==    by 0x48732EB: create_thread (createthread.c:101)
==38625==    by 0x4874E0F: pthread_create@@GLIBC_2.2.5 (pthread_create.c:817)
==38625==    by 0x48475DA: pthread_create_WRK (hg_intercepts.c:445)
==38625==    by 0x4848AF5: pthread_create@* (hg_intercepts.c:478)
==38625==    by 0x109214: main (in /home/berkay/Desktop/ostep-homework/threads-api/main-signal)
==38625==
==38625== ----------------------------------------------------------------
==38625==
==38625== Possible data race during read of size 4 at 0x10C014 by thread #1
==38625== Locks held: none
==38625==    at 0x109239: main (in /home/berkay/Desktop/ostep-homework/threads-api/main-signal)
==38625==
==38625== This conflicts with a previous write of size 4 by thread #2
==38625== Locks held: none
==38625==    at 0x1091C5: worker (in /home/berkay/Desktop/ostep-homework/threads-api/main-signal)
==38625==    by 0x48477D2: mythread_wrapper (hg_intercepts.c:406)
==38625==    by 0x4874608: start_thread (pthread_create.c:477)
==38625==    by 0x49AE132: clone (clone.S:95)
==38625==  Address 0x10c014 is 0 bytes inside data symbol "done"
==38625==
this should print last
==38625==
==38625== Use --history-level=approx or =none to gain increased speed, at
==38625== the cost of reduced accuracy of conflicting-access information
==38625== For lists of detected and suppressed errors, rerun with: -s
==38625== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 62 from 37)
```

7. Şimdi bu programda helgrind'i çalıştırın. Ne rapor ediyor? Kod doğru mu?

Helgrind bir veri yarışı olduğunu ve bir thread sızıntısı olduğunu bildirir. TSan bu raporu doğrular,

ancak talimatların çıkmadan önce yapıldığı varsayıldığında, programın doğruluğunu etkilemez.

8. Şimdi main-signal-cv.c'de bulunan kodun biraz değiştirilmiş versiyonuna bakın. Bu sürüm, sinyali (ve ilgili kilidi) yapmak için bir koşul değişkeni kullanır. Bu kod neden önceki sürüme tercih ediliyor? Doğruluk yönünden dolayı mı, performansından dolayı mı, yoksa her ikisi mi?

Önemli programlamarda iyi bir çözümdür çünkü sinyalleşmeye yazılan her bir kod yüzünden eşzamanlılık hatalarına neden olabilir. Koşul değişkeni, planlayıcıya o threadin durdurulduğunu ve başka bir yerden sinyal gelene kadar o threadin ele alınmamasını söyler.Ancak bu şekilde yapılır ise bu hatalara düşme ihtimali de azalmış olur ve birbirleri ile etkileşim süresi azalırsa daha iyi bir performans görülür.

9. Once again run `helgrind` on `main-signal-cv`. Does it report any errors?
   Helgrind, veri yarışı için bir hata vermez ama yinede thread sızıntısı olma ihtimalinden Dolayı hata verebilir

```
==38974== Helgrind, a thread error detector
==38974== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==38974== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==38974== Command: ./main-signal-cv
==38974==
this should print first
this should print last
==38974==
==38974== Use --history-level=approx or =none to gain increased speed, at
==38974== the cost of reduced accuracy of conflicting-access information
==38974== For lists of detected and suppressed errors, rerun with: -s
==38974== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 7 from 7)
```