27

Interlude: Thread API

Bu bölüm, iş parçacığı(thread) API'nin 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 kilit ve koşul değişken kavramlarını birçok örnekle daha yavaş tanıttığını not etmeliyiz; Bu nedenle bu bölüm referans olarak daha iyi kullanılır.

CRUX: İŞ PARÇACIKLARI (THREADLER) NASIL OLUŞTURULUR VE KONTROL EDİLİR?

İşletim sistemi, iş parçacığı (thread) oluşturma ve control için hangi arabirimleri sunmalıdır? Bu arayüzler, kullanım kolanım kolaylığının yanı sıra fayda sağlayacak şekilde nasıl tasarlanmalıdır?

27.1 İŞ PARÇACIĞI OLUŞTURMA

Çok parçacıklı (multi-thread) bir program yasmak için yapabilmeniz gereken ilk şey, yeni şi parçacıkları oluşturmaktır ve bu nedenle bir tür iş parçacığı oluşturma arabirimi bulunmalıdır. POSIX'te bu kolaydır:

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

Bu açıklama biraz karmaşık görünebilir (özellikle C'de fonksiyon işaretçilerini kullanmadıysanız) ama aslında o kadar kötü değil. Dört argüman var: iş parçacığı (`thread`), `attr`, başlangıç rutini (`start routine`), ve `arg`. Öncelikle, `iş parçacığı`, pthread t tipinde bir yapıya işaretçidir ; bu yapıyı bu iş parçacığıyla etkileşim kurmak için kullanacağız, ve bu nedenle onu başlatmak için pthread create()'e geçirmemiz gerekiyor.

İkinci argüman, `attr`, bu iş parçacığının sahip olabieceği nitelikleri belirtmek için kullanılır. Bazı örnekler, yığın boyutunun ayarlanması veya belki de iş parçacığının zamanlama önceliği hakkında bilgiler içerebilir. Bir öznitelik, pthread attr init();'e ayrı bir çağrı ile başlatılır; ayrıntılar için kılavuz sayfasına bakın. Ancak, çoğu durumda varsayılanlar uygun olacaktır; bu durumda, basitçe NULL değerini ileteceğiz.

Üçüncü argüman en karmaşık olanı, ama sadece şu soruyu soruyor: bu iş parçacığı hangi fonksiyonda çalışmaya başlamalı? C'de buna fonksiyon işaretçisi (**function pointer**) diyoruz ve bu bize şunları bekleyeceğimizi söylüyor: `void *` tipinde tek seferlik birr fonksiyon adı (başlangıç rutininden sonra parantez içinde belirtildiği gibi), ve `void *` tipinde döndürülen bir değer(**void pointer**) bir void işaretçisi.

Eğer bu döngü void yerine bir tamsayı argümanı gerektiriyorsa; işaretçi şöyle bir bildirim verir:

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

If instead the routine took a void pointer as an argument, but returned an integer, it would look like this:

```
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 void işaretçilerine ihtiyacımız var? Cevap oldukça basit: fonksiyon başlatma rutininin argümanı olarak bir void işaretçisine sahip olmak, 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 iletilmiş bir iş parçacığı oluştururuz. İş parçacığı bir kez oluşturulduktan sonra, argümanını beklenen türe çevirebilir ve böylece argümanları istenen türde açabilir.

Ve işte! Bir kere oluşturulan iş parçacığı, 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 İş Parçacığı (Thread) Tamamlama

Yukarıdaki örnek, bir iş parçacığının nasıl oluşturulacağını gösterir. Peki iş parçacığının tamamlanmasını beklerken ne olur ? Tamamlanmasını beklemek için özel bir şey yapmanız gerekiyor; özellikle, `pthread join()` döngüsünü çağırmak gerekir.

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

```
#include <stdio.h>
#include <pthread.h>

typedef struct {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Figure 27.1: **İş Parçacığı (Thread) Oluşturma**

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()'e bir argüman olarak bir işaretçi ilettiğinizde); Elde tutarsanız, o iş parçacığının sona ermesini beklemek için kullanabilirsiniz.

İkinci bağımsız değişken, geri almayı beklediğiniz dönüş değerinin bir işaretçisidir. Rutin her şeyi döndürebildiğinden, boşluğa bir işaretçi döndürmek üzere tanımlanır; pthread birleştirme() yordamı, iletilen bağımsız değişkenin değerini değiştirdiğinden, yalnızca değerin kendisine değil, bu değere bir işaretçi iletmeniz gerekir..

Başka bir örneğe bakalım (Şekil 27.2, sayfa 4). Kodda, yeniden tek bir iş parçacığı oluşturulur ve myarg t yapısı aracılığıyla birkaç argüman geçirilir.. Değer döndürmek için `myret t` tipi kullanılır. İş parçacığı çalışması bittiğinde, bekleyen ana iş parçacığı `pthread join()` içinde rutin[1], ardından geri döner ve iş parçacığından döndürülen değerlere, yani myret t'de ne varsa erişebiliriz.

Bu örnek hakkında dikkat edilmesi gereken birkaç nokta. İlk olarak, çoğu zaman tüm bu sancılı argümanları paketlemek ve açmak zorunda değiliz. Örneğin, argümansız bir thread oluştursak, thread oluşturulduğunda NULL'u argüman olarak iletebiliriz. Benzer şekilde, dönüş değerini umursamıyorsak, NULL'u pthread join()'e iletebiliriz.

[1] Burada sarıcı işlevleri kullandığımızı unutmayın; 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, sadece tek bir değer (örneğin, uzun bir int) iletiyorsak, bunu bir bağımsız değişken olarak paketlememiz gerekmez. Şekil 27.3 (sayfa 5) bir örnek göstermektedir. Bu durumda, argümanları paketlememiz ve yapıların içinde değerler döndürmemiz gerekmediğinden, hayat biraz daha basittir.

Üçüncüsü, değerlerin bir iş parçacığından nasıl döndürüldüğü konusunda son derece dikkatli olunması gerektiğine dikkat etmeliyiz. Spesifik olarak, iş parçacığının çağrı yığınında tahsis edilen bir şeye atıfta bulunan bir işaretçiyi asla 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, efsane okuması yığınına tahsis edilir. Bununla birlikte, geri döndüğünde, değer otomatik olarak yeniden dağıtılır (bu nedenle yığının kullanımı bu kadar kolaydır!) ve bu nedenle,

```
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;
}
```

Figure 27.3: **Simpler Argument Passing to a Thread**

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 ileti dizisi oluşturmak için pthread create() kullanımının ve hemen ardından pthread birleştirme() çağrısının (**procedure call**) kullanılmasının, bir ileti dizisi oluşturmanın oldukça garip bir yolu olduğunu fark edebilirsiniz. Aslında tam da bu görevi başarmanın daha kolay bir yolu var; buna prosedür çağrısı denir. Açıkçası, genellikle birden fazla iş parçacığı oluşturacağız ve bunun tamamlanmasını bekleyeceğiz, aksi halde iş parçacıklarını kullanmanın pek bir amacı yok.

Çok iş parçacıklı olan tüm kodların birleştirme yordamını kullanmadığına dikkat etmeliyiz. Örneğin, çok iş parçacıklı bir web sunucusu bir dizi çalışan iş parçacığı oluşturabilir ve ardından ana iş parçacığını istekleri kabul etmek ve bunları çalışanlara süresiz olarak iletmek için 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 hesaplamanın bir sonraki aşamasına geçmeden önce bu tür tüm işlerin tamamlandığından emin olmak için büyük olasılıkla birleştirme özelliğini kullanacaktır.

27.1 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);
```

[2] Neyse ki derleyici gcc, böyle bir kod yazdığınızda 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 neye benzediğini hayal edebilirsiniz:

```
pthread_mutex_t lock; pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

Kodun amacı şu şekildedir: pthread mutex lock() çağrıldığında başka bir thread kilidi tutmazsa, thread kilidi alır ve kritik bölüme girer. Başka bir iş parçacığı gerçekten kilidi tutuyorsa, kilidi almaya çalışan iş parçacığı, kilidi elde edene kadar çağrıdan geri dönmeyecek (kilidi tutan iş parçacığının kilidi açma çağrısı yoluyla onu serbest bıraktığı anlamına gelir). Tabii ki, birçok iş parçacığı belirli bir zamanda kilit edinme işlevi içinde beklemede kalmış olabilir; Ancak, yalnızca edinilen kilide sahip iş parçacığı, kilidi açmalıdır.

Ne yazık ki, bu kod iki önemli şekilde kırılmıştır. İlk sorun, uygun başlatma eksikliğidir. Başlangıçta doğru değerlere sahip olduklarını ve böylece kilitle ve kilit aç çağrıldığında istenildiği gibi çalıştıklarını garanti etmek için tüm kilitler uygun şekilde başlatılmalıdır.

POSIX iş parçacıklarıyla, kilitleri başlatmanın iki yolu vardır. Bunu yapmanın bir yolu, PTHREAD MUTEX INITIALIZER'ı aşağıdaki gibi 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() öğesine bir çağrı 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 kendiniz okuyun; NULL'u geçmek, yalnızca varsayılanları kullanır. Her iki şekilde de çalışır, ancak genellikle dinamik (ikinci) yöntemi kullanırız. Kilitle işiniz bittiğinde, ilgili pthread mutex destroy() çağrısının da yapılması gerektiğini unutmayın; tüm ayrıntılar için kılavuz sayfasına bakın.

Yukarıdaki kodla ilgili ikinci sorun, kilitleme ve kilidi açma çağrılırken hata kodlarını kontrol edememesidir. Tıpkı bir UNIX sisteminde çağırdığınız hemen hemen her kitaplık yordamı gibi, bu yordamlar da başarısız olabilir! Kodunuz hata kodlarını düzgün bir şekilde kontrol etmezse, hata sessizce gerçekleşir ve bu durumda kritik bir bölüme birden çok iş parçacığının girmesine izin verebilir. En azından, Şekil 27.4'te (sayfa 7) gösterildiği gibi, rutinin başarılı olduğunu iddia eden paketleyiciler kullanın; bir şeyler ters gittiğinde kolayca çıkamayan daha karmaşık (oyuncak olmayan) programlar, arıza olup olmadığını kontrol etmeli ve bir arama başarılı olmadığında uygun bir şeyler yapmalıdır.

```
// Keeps code clean; only use if exit() OK upon failure
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 etkileşime geçmek için pthreads kitaplığı. Diğer iki ilgi rutini:

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

Bu iki çağrı, kilit ediniminde kullanılır. Kilit zaten tutulmuşsa, trylock versiyonu başarısızlık döndürür; Bir kilit edinmenin zamanlanmış kilit versiyonu, hangisi önce olursa, bir zaman aşımından sonra veya kilidi aldıktan sonra geri döner. Böylece, sıfır zaman aşımına sahip zamanlanmış kilit, trylock durumuna dejenere olur. Bu sürümlerin her ikisinden de genellikle kaçınılmalıdır; ancak, gelecek bölümlerde göreceğimiz gibi (örneğin kilitlenmeyi incelerken) göreceğimiz gibi, bir kilit edinme rutininde takılıp kalmaktan kaçınmanın (belki de kesinlikle) yararlı olabileceği birkaç durum vardır.

27.3 Durum Değişkenleri

Herhangi bir iş parçacığı kitaplığının diğer ana bileşeni ve kesinlikle POSIX iş parçacıklarında durum, bir koşul değişkeninin varlığıdır. Koşul değişkenleri, bir iş parçacığı devam etmeden önce diğerinin bir şeyler yapmasını bekliyorsa, iş parçacıkları arasında bir tür sinyalleşmenin gerçekleşmesi gerektiğinde kullanışlıdır. Bu şekilde etkileşim kurmak 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, kişinin ek olarak bu koşulla ilişkilendirilmiş bir kilide sahip 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 uykuda olan iş parçacığının umursayabileceği bir şey değiştiğinde, başka bir iş parçacığının buna 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);
```

3

Bu kodda, ilgili kilit ve koşul[3]'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 olan 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 nokta. İlk olarak, sinyal gönderirken (ayrıca global değişkeni hazır hale getirirken), kilidi her zaman basılı tuttuğumuzdan emin oluruz. Bu, kazara kodumuza 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, arayan diziyi uyku moduna geçirmeye ek olarak, söz konusu arayan kişiyi uyuttuğunda kilidi serbest bırakmasıdır. Olmadığını hayal edin: diğer iş parçacığı kilidi nasıl alabilir ve uyanması için ona sinyal verebilir? Bununla birlikte, uyandırıldıktan sonra geri dönmeden önce, pthread cond wait() kilidi yeniden alır, böylece bekleyen iş parçacığının, bekleme dizisinin başlangıcındaki kilit edinimi ile sonundaki kilidin serbest bırakılması arasında çalıştığı her an sağlanır. kilidi tutar.

Son bir tuhaflık: bekleyen iş parçacığı, basit bir if deyimi yerine bir süre döngüsünde durumu yeniden kontrol eder. Gelecekteki bir bölümde koşul değişkenlerini incelerken bu konuyu ayrıntılı olarak tartışacağız, ancak genel olarak, bir while döngüsü kullanmak yapılacak basit ve güvenli şeydir. Koşulu yeniden kontrol etse de (belki biraz ek yük ekleyerek), bekleyen bir diziyi sahte bir şekilde uyandırabilecek bazı pthread uygulamaları vardır; böyle bir durumda tekrar kontrol edilmeden bekleyen thread değişmediği halde durum değişti zannederek devam edecektir. Bu nedenle, uyanmayı mutlak bir gerçek yerine bir şeylerin değişmiş olabileceğine dair bir ipucu olarak görmek daha güvenlidir.Bazen iki iş parçacığı arasında sinyal vermek için bir koşul değişkeni ve ilgili kilit yerine basit bir işaret kullanmanın cazip geldiğini unutmayın. Örneğin, bekleme kodunda daha çok böyle görünmesi için yukarıdaki bekleme kodunu yeniden yazabiliriz:

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

The associated signaling code would look like this:

```
ready = 1;
```

[3] Statik başlatıcı PTHREAD_COND_INITIALIZER yerine pthread_cond_init() (ve pthread_cond_destroy()) kullanılabilir. Daha fazla iş gibi mi geliyor? Öyle.

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]'u gösterdiği gibi, iş parçacıkları arasında senkronizasyon için işaretler (yukarıdaki gibi) kullanılırken hata yapmak şaşırtıcı derecede kolaydır; o çalışmada, bu ad hoc senkronizasyonların kullanımlarının kabaca yarısı hatalıydı! tembel olmayın; bunu yapmadan kurtulabileceğinizi düşündüğünüzde bile koşul değişkenlerini kullanın. Koşul değişkenleri kafa karıştırıcı geliyorsa, (henüz) çok fazla endişelenmeyin; bunları bir sonraki bölümde ayrıntılı olarak ele alacağız. O zamana kadar var olduklarını bilmek ve nasıl ve neden kullanıldıklarına dair fikir sahibi olmak yeterli olacaktır.

27.4 Derleme ve Çalıştırma

Bu bölümdeki tüm kod örneklerini kurmak ve çalıştırmak nispeten kolaydır. Bunları derlemek için kodunuza pthread.h başlığını eklemelisiniz. Bağlantı satırında, -pthread işaretini ekleyerek açıkça pthreads kitaplığıyla bağlantı kurmalısınız.

Örneğin, basit bir multi-threaded programı derlemek için yapmanız gereken tek şey şudur:

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

main.c, pthreads başlığını içerdiği sürece, bir eşzamanlı programı başarıyla derlemiş olursunuz. Her zamanki gibi işe yarayıp yaramadığı tamamen farklı bir konu.

27.5 Özet

Konu oluşturma, kilitler aracılığıyla karşılıklı dışlama oluşturma ve koşul değişkenleri aracılığıyla sinyal gönderme ve bekleme dahil olmak üzere pthread kitaplığının temellerini tanıttık. Sağlam ve verimli çok iş parçacıklı kod yazmak için sabır ve büyük bir özen dışında başka bir şeye ihtiyacınız yok!

Şimdi, çok iş parçacıklı kod yazarken işinize yarayabilecek bir dizi ipucuyla bu bölümü bitiriyoruz (ayrıntılar için bir sonraki sayfada bir kenara bakın). API'nin ilginç olan başka yönleri de var; daha fazla bilgi istiyorsanız, arayüzün tamamını oluşturan yüzden fazla API görmek için bir Linux sisteminde man -k pthread yazın. Bununla birlikte, burada tartışılan temel bilgiler, karmaşık (ve umarız doğru ve performanslı) çok iş parçacıklı programlar oluşturmanıza olanak sağlamalıdır. İş parçacıklarıyla ilgili zor kısım, API'ler değil, eşzamanlı programları nasıl oluşturduğunuza ilişkin aldatıcı mantıktır. Daha fazlasını öğrenmek için okumaya devam edin.

Köşe Yazısı: THREAD API YÖNERGELERİ

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

- **Basit tutun.** Her şeyden önce, iş parçacıkları arasında kilitlenecek veya sinyal verecek herhangi bir kod mümkün olduğunca basit olmalıdır. Zor iş parçacığı etkileşimleri hatalara yol açar.
- **İş parçacığı etkileşimlerini en aza indirin.** Konuların etkileşimde bulunduğu yolların sayısını minimumda tutmaya çalışın. Her etkileşim dikkatlice düşünülmeli ve denenmiş ve doğru yaklaşımlarla (çoğu hakkında gelecek bölümlerde öğreneceğimiz) inşa edilmelidir.
- **Kilitleri ve koşul değişkenlerini başlatın.** Bunun yapılmaması, bazen çok garip şekillerde çalışan ve bazen başarısız olan kodlara yol açacaktır.
- **İade kodlarınızı kontrol edin.** Elbette, yaptığınız herhangi bir C ve UNIX programlamasında, her dönüş kodunu kontrol etmelisiniz ve bu burada da geçerli. Bunu yapmamak, tuhaf ve anlaşılması zor davranışlara yol açarak (a) çığlık atmanıza, (b) saçınızın bir kısmını yolmanıza veya (c) her ikisine birden yol açmanıza neden olur.
- **Dizilere bağımsız değişkenleri nasıl ilettiğinize ve ileti dizilerinden nasıl değer döndürdüğünüze dikkat edin.** Özellikle, yığında tahsis edilmiş bir değişkene referans ilettiğiniz her seferde muhtemelen yanlış bir şey yapıyorsunuzdur.
- **Her iş parçacığının kendi yığını vardır.** Yukarıdaki noktayla ilgili olarak, lütfen her iş parçacığının kendi yığını olduğunu unutmayın. Bu nedenle, bir iş parçacığının yürüttüğü bazı işlevlerin içinde yerel olarak tahsis edilmiş bir değişkeniniz varsa, bu aslında o iş parçacığına özeldir; başka hiçbir iş parçacığı ona (kolayca) erişemez. İş parçacıkları arasında veri paylaşmak için, değerlerin öbekte veya başka bir şekilde genel olarak erişilebilen bir yerel ayarda olması gerekir.
- **İş parçacıkları arasında sinyal vermek için her zaman koşul değişkenlerini kullanın.** Genellikle basit bir bayrak kullanmak 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.

Sorular

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 hata ayıkladığı için karşılaşacağı kilitlenme sorunlarını 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)
```

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?

   Döngü dengesi sağlanmadan artırım yapıldığı için artırımın yapıldığı satırlar düzeni bozar.

   Rahatsız edici satırları sırayla kaldırdığımızda çıktımız şu şekildedir:

```
==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)
```

3. 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?
   İş parçacıkları işi aynı anda alırlrsa birbirlerini bekleyecekleri bir durum oluşur ve bu deadlock'a dönüşür.
4. Şimdi bu kod üzerinde helgrind'i çalıştırın. Helgrind ne rapor ediyor(main-deadlock.c dosyasında)?
   Helgrind kilitleri ve iş parçacıklarının lokasyonlarını şu şekilde raporlar:

```
==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)
```

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 ?

   Main-deadlock-global main-deadlock'ı engeller ve şöyle bir rapor çıktısı oluşur:

```
==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?)
   Kod işlemcide gereksiz yer kapladığı için verimsizdir.
   Parent iş parçacığı bu süreyi koşulu kontrol ederek geçirir.

7. Şimdi bu programda helgrind'I çalıştırın. Ne rapor ediyor? Kod doğru mu?
   İş parçacığında açık olduğunu raporlar ancak kod doğrudur.

```
==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)
```

8. Şimdi main-signal-cv.c'de bulunan kodun biraz değiştirilmiş versiyonuna bakın. Bu sürüm, sinyali (ve ilgili kilidi) vermek için bir koşul değişkeni kullanır. Bu kod neden önceki sürüme tercih edilir? Doğruluk yönünden dolayı mı, performansından dolayı mı, yoksa her ikisi mi?
   Koşul değişkeni iş parçacıklarının birbirlerinin çalışma zamanına müdahale etmesini önlediği için hata ve sürelerde optimuma yaklaşılır. Her iki durum sebebiyle tercih edilir.

9. Main-signal-cv üzerinde helgrind'i bir kez daha çalıştırın. Herhangi bir hata bildiriyor mu?
   Veri akışından vermez ama iş parçacığından sızıntı hatası 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)
```