27

Interlude: İş Parçacığı API'leri

Bu bölüm kısaca iş parçacığı API'sinin ana bölümlerini 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 bölümlerde kilitler ve koşul değişkenleri kavramlarının birçok örnekle birlikte daha yavaş tanıtıldığına dikkat etmeliyiz; bu bölüm dolayısıyla daha iyi bir referans olarak kullanılır.

Neler oluyor?: IŞ PARÇACIKLARI NASIL OLUŞTURULUR VE KONTROL EDILIR

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

27.1 İş Parçacığı Oluşturma

Çok iş parçacıklı bir program yazmak için yapabilmeniz gereken ilk şey, yeni iş parçacıkları oluşturmaktır ve bu nedenle bir tür iş parçacığı oluşturma arabirimi bulunmalıdır. 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) ama aslında o kadar da kötü değil. Dört argüman vardır: `thread`, `attr`, `start_routine` ve `arg`. İlki, `thread`, `pthread_t` tipindeki bir yapının işaretçisidir; bu yapıyı bu iş parçacığıyla etkileşim kurmak için kullanacağız ve bu nedenle onu başlatmak için onu `pthread_create()`'e geçirmemiz gerekiyor.

İkinci bağımsız değişken olan `attr`, bu iş parçacığının sahip olabileceği nitelikleri belirtmek için kullanılır. Bazı örnekler, yığın boyutunun ayarlanmasını veya belki de iş parçacığının zamanlama önceliği hakkında bilgileri içerir. 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 olanıdır, ancak sadece şunu soruyor: bu iş parçacığı hangi işlevde çalışmaya başlamalıdır? C'de buna **bir işlev işaretçisi(function pointer)** diyoruz ve bu bize şunun beklendiğini söylüyor: `void *` türünde tek bir argüman iletilen bir işlev adı (`start routine`) (başlangıç yordamından sonra parantez içinde belirtildiği gibi) ve bu, `void *` türünde bir değer döndürür (yani, bir **geçersiz işaretçi(void pointer)**).

Bunun yerine bu rutin bir boşluk yerine bir tamsayı bağımsız değişkeni gerektiriyorsa işaretçi bildirim şöyle görünür:

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

Bunun yerine rutin argüman olarak bir geçersiz işaretçi alırsa ancak bir tamsayı döndürürse şöyle görünür:

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

Son olarak, dördüncü bağımsız değişken olan `arg`, tam olarak iş parçacığının yürütmeye başladığı işleve iletilecek bağımsız değişkendirŞunu sorabilirsiniz: neden bu geçersiz işaretçilere ihtiyacımız var? Cevap oldukça basit: işlev `start routine` için bir argüman olarak bir boşluk işaretçisine sahip olmak, herhangi bir argüman türünü iletmemize izin verir; bir dönüş değeri olarak sahip olmak, iş parçacığının herhangi bir türde sonuç döndürmesine izin verir.

Şekil 27.1'deki bir örneğe bakalım. Burada, kendi tanımladığımız tek bir türde paketlenmiş, iki bağımsız değişkenin iletildiği bir ileti dizisi oluşturuyoruz. (`myarg t`). İş parçacığı bir kez oluşturulduktan sonra, bağımsız değişkenini beklediği türe kolayca aktarabilir ve böylece bağımsız değişkenleri istenildiği gibi açabilir.

Ve işte burada! Bir iş parçacığı oluşturduğunuzda, programda şu anda var olan tüm iş parçacıklarıyla aynı adres alanında çalışan, kendi çağrı yığınıyla tamamlanmış, gerçekten başka bir canlı yürütme varlığınız olur. Eğlence böylece başlar!

27.2 İş Parçacığı Tamamalama

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

```
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: **İş Parçacığı Oluşturma**

Bu rutin iki bağımsız değişken alır. İlki `pthread t` türündedir ve hangi iş parçacığının bekleneceğini belirtmek için kullanılır. Bu değişken, iş parçacığı oluşturma yordamı tarafından başlatılır (`pthread create()` işlevine bir argüman olarak ona bir işaretçi ilettiğinizde); etrafta 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 join()` rutini, 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, tekrar tek bir iş parçacığı oluşturulur ve `myarg t` yapısı aracılığıyla birkaç argüman iletilir. Değer döndürmek için `myret t` tipi kullanılır. İş parçacığı çalışması bittiğinde, bekleyen ana iş parçacığı `pthread join()` rutin1'in içinde, sonra geri döner ve iş parçacığından döndürülen değerlere, yani `myret t`'de ne varsa erişebiliriz.

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

Burada sarmalayıcı işlevleri kullandığımıza dikkat edin; özellikle, Malloc(), Pthread join() ve Pthread create() olarak adlandırırız, bunlar sadece benzer şekilde adlandırılan küçük harf sürümlerini çağırır.

```
1  typedef struct { int a; int b; } myarg_t;
2  typedef struct { int x; int y; } myret_t;
3
4  void *mythread(void *arg) {
5      myret_t *rvals = Malloc(sizeof(myret_t));
6      rvals->x = 1;
7      rvals->y = 2;
8      return (void *) rvals;
9  }
10
11 int main(int argc, char *argv[]) {
12     pthread_t p;
13     myret_t *rvals;
14     myarg_t args = { 10, 20 };
15     Pthread_create(&p, NULL, mythread, &args);
16     Pthread_join(p, (void **) &rvals);
17     printf("returned %d %d\n", rvals->x, rvals->y);
18     free(rvals);
19     return 0;
20 }
```

Figure 27.2 **İş Parçacığının Tamamlanması**

İkincisi, sadece tek bir değer (örneğin; `long long int`) iletiyorsak, onu bir bağımsız değişken olarak paketlememiz gerekmez. Şekil 27.3 (sayfa 5) bir örnek göstermektedir. 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, iş parçacığının çağrı yığınında tahsis edilen bir şeye atıfta bulunan bir işaretçiyi asla döndürmeyin. Yaparsan, ne olacağını düşünüyorsun? (bir düşünün!) İşte Şekil 27.2'deki örnekten değiştirilmiş tehlikeli bir kod parçası örneği.

```
1  void *mythread(void *arg) {
2      myarg_t *args = (myarg_t *) arg;
3      printf("%d %d\n", args->a, args->b);
4      myret_t oops; // ALLOCATED ON STACK: BAD!
5      oops.x = 1;
6      oops.y = 2;
7      return (void *) &oops;
8  }
```

Bu durumda, oops değişkeni `mythread` yığınına tahsis edilir. Bununla birlikte, geri döndüğünde, değer otomatik olarak yeniden dağıtılır (bu yüzden yığının kullanımı bu kadar kolaydır!) ve böylece bir işaretçi geri gönderilir.

rutinler beklenmedik bir şey döndürmedi.

```
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: Bir İş Parçacığına Daha Basit Argüman Geçirme

Şimdi serbest bırakılmış bir değişkene her türlü kötü sonuca yol açacaktır. Elbette, geri döndüğünüzü düşündüğünüz değerleri yazdırdığınızda, muhtemelen (ama zorunlu değil!) şaşıracaksınız. Deneyin ve kendiniz öğrenin!

Son olarak, bir ileti dizisi oluşturmak için `pthread create()` kullanımının ve hemen ardından `pthread join()` çağrısının kullanılmasının, bir ileti dizisi oluşturmanın oldukça garip bir yolu olduğunu fark edebilirsiniz. Aslında, tam olarak bu görevi yerine getirmenin daha kolay bir yolu var; buna **prosedür çağrısı(procedure call)** 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, büyük olasılıkla bu tür tüm işlerin, hesaplamanın bir sonraki aşamasına geçmeden veya çıkmadan önce tamamlandığından emin olmak için birleştirme özelliğini kullanacaktır.

27.3 Kilitler

Dizi oluşturma ve birleştirmenin ötesinde, muhtemelen POSIX dizi kitaplığı tarafından sağlanan bir sonraki en yararlı işlev grubu, kilitler yoluyla kritik bir bölüme karşılıklı dışlama sağlayanlardır. Bu amaçla kullanılacak en temel rutin çifti aşağıdakiler tarafından sağlanır:

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

Neyse 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(critical**

section) 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; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

Kodun amacı aşağıdaki gibidir: `pthread mutex lock()` çağrıldığında başka bir thread kilidi tutmuyorsa, 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(lack of proper initialization).** 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 geçirmek yalnızca varsayılanları kullanır. Her iki şekilde de çalışır, ancak genellikle dinamik (ikinci) yöntemi kullanırız Kilitle işiniz bittiğinde, ilgili bir `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 hemen 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 Sarmalayıcı

The lock and unlock routines are not the only routines within the pthreads library to interact with locks. Two other routines of interest:

```
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` sürümünü sona erdiriyor; Bir kilit gövdesinin `timedlock` versiyonu, hangisi daha önce olursa, bir zaman aşımından sonra veya kilit geri döndükten sonra döner. Böylece, sıfır zaman aşılamasına sahip zamanlanmış kilit, trylock yönünde dejenere olur. Bunların ikisinden de genellikle kaçınılmalıdır; Ancak, gelecek bölümde göreceğimiz gibi (örneğin kilitlenmeyi incelerken) göreceğimiz gibi, bir kilit deposu rutininde takılıp kalmaktan kaçınmanın geçerli olabileceği birkaç durum vardır.

27.4 Durum Değişkenleri

Herhangi bir iş parçacığı kütüphanesinin diğer önemli bileşeni ve kesinlikle POSIX iş parçacıklarında durum, bir **koşul değişkeninin(condition variables)** varlığıdır. İş parçacıkları arasında bir tür sinyalleşme gerçekleşmesi gerektiğinde, bir iş parçacığı devam etmeden önce diğerinin bir şey yapmasını bekliyorsa, durum değişkenleri 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, ek olarak bu koşulla ilişkili bir kilide sahip olmak gerekir. Yukarıdaki rutinlerden birini çağırırken, bu kilit tutulmalıdır.

İlk rutin olan `pthread cond wait()`, çağıran iş parçacığını uyutur ve böylece genellikle programda şu anda uyuyan iş parçacığının önemseyebileceği bir şey değiştiğinde başka bir iş parçacığının sinyal vermesini bekler. Tipik bir kullanım şu şekildedir:

```
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 koşul[3] başlatıldıktan sonra, bir iş parçacığı `ready` değişkeninin henüz sıfırdan başka bir değere ayarlanıp ayarlanmadığını kontrol eder. Aksi takdirde, iş parçacığı başka bir iş parçacığı onu uyandırana kadar uyumak için bekleme rutinini çağırır.

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

Bu kod dizisi hakkında dikkat edilmesi gereken birkaç şey var. İlk olarak, sinyal verirken (ve `ready` global değişkenini değiştirirken), her zaman kilidin tutulduğundan emin oluruz. Bu, kodumuza yanlışlıkla bir yarış koşulu eklememizi sağlar.

İkinci olarak, wait çağrısının ikinci parametre olarak bir kilit aldığını, oysa signal çağrısının yalnızca bir koşul aldığını fark edebilirsiniz. Bu farkın nedeni, wait çağrısının, çağrıyı yapan iş parçacığını uyutmanın yanı sıra, söz konusu çağrıyı yapanı uyuturken kilidi serbest bırakmasıdır.Bunu yapmadığını düşünün: diğer iş parçacığı kilidi nasıl alabilir ve uyanması için ona nasıl sinyal verebilir? Bununla birlikte, uyandırıldıktan sonra geri dönmeden önce, `pthread cond wait()` kilidi yeniden alır, böylece bekleyen iş parçacığının bekleme dizisinin başlangıcındaki kilit alımı ile sonundaki kilit serbest bırakma arasında çalıştığı her zaman kilidi tutmasını sağlar.

Son bir gariplik: Bekleyen iş parçacığı, basit bir if deyimi yerine bir while döngüsü içinde koşulu yeniden kontrol eder. Bu konuyu ilerideki bir bölümde koşul değişkenlerini incelerken ayrıntılı olarak tartışacağız, ancak genel olarak while döngüsü kullanmak yapılacak en basit ve güvenli şeydir. Her ne kadar koşulu tekrar kontrol etse de (belki biraz ek yük ekleyerek), bekleyen bir iş parçacığını yanlışlıkla uyandırabilecek bazı pthread uygulamaları vardır; böyle bir durumda, tekrar kontrol edilmeden, bekleyen iş parçacığı koşul değişmediği halde değiştiğini düşünmeye devam edecektir. Bu nedenle uyanmayı mutlak bir gerçek olarak görmek yerine bir şeylerin değişmiş olabileceğine dair bir ipucu olarak görmek daha güvenlidir. Bazen iki iş parçacığı arasında sinyal vermek için bir koşul değişkeni ve ilişkili kilit yerine basit bir bayrak kullanmanın cazip olduğunu unutmayın. Örneğin, yukarıdaki bekleme kodunu daha çok bekleme kodundaki gibi görünecek şekilde yeniden yazabiliriz:

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

İlgili sinyal kodu şu şekilde görünecektir:

```
ready = 1;
```

3Statik başlatıcı PTHREAD COND INITIALIZER yerine pthread cond init() (ve pthread cond destroy()) kullanılabilir. Daha fazla iş gibi mi geliyor? Evet öyle.

Aşağıdaki nedenlerden dolayı bunu asla yapmayın. Birincisi, birçok durumda kötü performans gösterir (uzun süre döndürmek sadece CPU döngülerini boşa harcar). İkincisi, hataya meyillidir. Yakın zamanda yapılan bir araştırmanın [X+10] gösterdiği gibi, iş parçacıkları arasında senkronizasyon sağlamak için bayrakları (yukarıdaki gibi) kullanırken hata yapmak şaşırtıcı derecede kolaydır; bu çalışmada, bu geçici senkronizasyonların kullanımlarının yaklaşık yarısı hatalıydı! Tembel olmayın; bunu yapmadan kurtulabileceğinizi düşündüğünüzde bile koşul değişkenlerini kullanın. Koşul değişkenleri kafa karıştırıcı geliyorsa, çok fazla endişelenmeyin (henüz) - bunları bir sonraki bölümde ayrıntılı olarak ele alacağız. O zamana kadar, var olduklarını bilmek ve nasıl ve neden kullanıldıkları hakkında biraz fikir sahibi olmak yeterli olacaktır.

27.4 Derleme ve Çalıştırma

Bu bölümdeki tüm kod örneklerinin çalıştırılması nispeten kolaydır. Bunları derlemek için kodunuza `pthread.h` başlığını eklemeniz gerekir. Bağlantı satırında, `-pthread` bayrağını ekleyerek pthreads kütüphanesiyle de açıkça bağlantı kurmalısınız. Örneğin, basit bir çok iş parçacıklı programı derlemek için tek yapmanız gereken aşağıdakileri yapmaktır:

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

`main.c` pthreads başlığını içerdiği sürece, artık eş zamanlı bir programı başarıyla derlediniz demektir. Her zamanki gibi çalışıp çalışmaması tamamen farklı bir konudur.

27.5 ÖZET

İş parçacığı oluşturma, kilitler aracılığıyla karşılıklı dışlama oluşturma ve koşul değişkenleri aracılığıyla sinyal verme ve bekleme dahil olmak üzere pthread kütüphanesinin temellerini tanıttık. Sağlam ve verimli çok iş parçacıklı kod yazmak için sabır ve büyük bir özen dışında başka bir şeye ihtiyacınız yoktur!

Bu bölümü, çok iş parçacıklı kod yazarken işinize yarayabilecek bir dizi ipucu ile bitiriyoruz (ayrıntılar için bir sonraki sayfadaki kenara 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, sofistike (ve umarım doğru ve performanslı) çok iş parçacıklı programlar oluşturmanızı sağlayacaktır. İş parçacıkları ile ilgili zor kısım API'ler değil, eşzamanlı programları nasıl oluşturduğunuzun zor mantığıdır. Daha fazlasını öğrenmek için okumaya devam edin.

ASIDE: **İŞ PARÇACIĞI API YÖNERGELERI**

- **Çok iş parçacıklı bir program oluşturmak için POSIX iş parçacığı kütüphanesini (veya herhangi bir iş parçacığı kütüphanesini) kullanırken hatırlamanız gereken birkaç küçük ama önemli şey vardır. Bunlar:**
- **- Basit tutun. Her şeyden önce, iş parçacıkları arasında kilitleme veya sinyal gönderme kodlarının mümkün olduğunca basit olması gerekir. Zorlu iş parçacığı etkileşimleri hatalara yol açar.**
- **- İş parçacığı etkileşimlerini en aza indirin. İş parçacıklarının etkileşime girdiği yolların sayısını minimumda tutmaya çalışın. Her etkileşim dikkatlice düşünülmeli ve denenmiş ve doğru yaklaşımlarla (birçoğunu ilerleyen bölümlerde öğreneceğiz) oluşturulmalıdır.**
- **- Kilitleri ve koşul değişkenlerini başlatın. Bunu yapmamak, bazen çalışan bazen de çok garip şekillerde başarısız olan kodlara yol açacaktır.**
- **Dönüş kodlarınızı kontrol edin. Elbette, yaptığınız tüm C ve UNIX programlarında, her bir dönüş kodunu kontrol etmeniz gerekir ve bu durum burada da geçerlidir. Bunu yapmamak tuhaf ve anlaşılması zor davranışlara yol açarak (a) çığlık atmanıza, (b) saçınızı başınızı yolmanıza ya da (c) her ikisine birden neden olabilir.**
- **İş parçacıklarına nasıl argüman aktardığınıza ve iş parçacıklarından nasıl değer döndürdüğünüze dikkat edin. Özellikle, yığın üzerinde ayrılmış bir değişkene referans aktardığınız her zaman, 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 ayrılmış bir değişkeniniz varsa, bu aslında o iş parçacığına özeldir; başka hiçbir iş parçacığı buna (kolayca) erişemez. İş parçacıkları arasında veri paylaşmak için, değerlerin heap'te ya da global olarak erişilebilen bir yerde olması gerekir.**
- **İş parçacıkları arasında sinyal vermek için her zaman koşul değişkenleri kullanın. Genellikle basit bir bayrak kullanmak cazip gelse de bunu yapmayın.**
- **Kılavuz sayfalarını kullanın. Özellikle Linux'ta, pthread man sayfaları oldukça bilgilendiricidir ve burada önceden gönderilen nüansların çoğunu, genellikle daha da ayrıntılı olarak tartışır. Bunları dikkatlice okuyun!**

.

References

[B89] "An Introduction to Programming with Threads" by Andrew D. Birrell. DEC Techni- cal 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!*

Ev Ödevi (Kod)

Bu bölümde, bazı basit çok iş parçacıklı 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 indirmesindeki README'yi okuyun.

Sorular

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

`Valgrind-3.12.0`Helgrind, El Capitan kurulumumda çalışmıyor . Kullanıcı kodunda veri yarışı olmadığı zamanlar da dahil olmak üzere, pthread kitaplıklarındaki veri yarışlarını bildirir.
Bunun yerine ThreadSanitizer (TSan) kullanacağım. TSan'ı kullanmak için önce koşmam gerekiyordu

```
$ DevToolsSecurity -enable
```

aksi takdirde macOS, yetersiz izinlerden şikayet eder.

İkili dosyanın TSan araçlarıyla oluşturulması ve ardından çalıştırılması gerekir:

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

Bu, veri yarışını tanımlayan bir uyarının yanı sıra:

- paylaşılan belleğe tehlikeli bir şekilde erişen her iş parçacığındaki kod satırları
- paylaşılan hafızanın sembol adı (`balance`)
- iş parçacığının oluşturulduğu kod satırı

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 ediyor?
   Yukarıdaki gibi TSan kullanacağız. İş parçacıklarının birinden müstehcen erişimlerin kaldırılması ve TSan'ın çalıştırılması hiçbir çıktı üretmez ve çıkış kodu 0 ile geri döner.TSan, paylaşılan değişkene yapılan güncellemelerden yalnızca birinin etrafına bir kilit ekledikten sonra, 1. sorudaki gibi, değişkeni güncellediğinde ve bu kilidin nerede başlatıldığı zaman iş parçacıklarından birinin bir kilit tuttuğunu da tanımlayan bir uyarı bildirir. Bu durumda TSan 134 çıkış kodu ile geri döner.

   Paylaşılan değişkenlere her iki güncellemenin etrafına bir kilit ekledikten sonra, TSan çıktı üretmez ve 0 çıkış koduyla geri döner.

3. Şimdi main-deadlock.c'ye bakalım. Kodu inceleyin. Bu kodun, kilitlenme olarak bilinen bir sorunu vardır (bunu daha sonraki bölümlerde tartışacağız).
İş parçacıkları aynı anda ilk kilitlerini alırsa potansiyel bir kilitlenme vardır (yani iş parçacığı 0, kilit 1'i ve iş parçacığı 1, kilit 2'yi yakalar ve her ikisi de sahip olmadıkları kilidi beklerken takılıp kalır).

4. Şimdi bu kod üzerinde helgrind'i çalıştırın. Helgrind ne rapor ediyor?
Bunun yerine TSan'ı çalıştıracağım (1. soruya bakın).

```
$ make main-deadlock
$ TSAN_OPTIONS=detect_deadlocks=1 ./main-deadlock
```
TSan aşağıdaki uyarıyı bildirir:

```
WARNING: ThreadSanitizer: lock-order-inversion (potential deadlock) (pid=71488)
  Cycle in lock order graph: M23 (0x0001001f4088) => M25 (0x0001001f40c8) => M23
```
Ayrıca, mutekslerin alındığı ve iş parçacıklarının nerede oluşturulduğu kod satırlarını da detaylandırır.

5. Şimdi main-deadlock-global.c üzerinde helgrind'i çalıştırın. Kodu inceleyin; main-deadlock.c ile aynı sorunu yaşıyor mu? Helgrind aynı hatayı bildirmeli mi? Bu size helgrind gibi araçlar hakkında ne söylüyor?
Bunun yerine TSan'ı çalıştıracağım (1. soruya bakın). Yeni kod `main-deadlock-global.c`aşağıdaki sorunlarla aynı değildir `main-deadlock.c`: genel kilit, bir kilitlenmenin oluşmasını engeller. Buna rağmen TSan şu uyarıyı yapıyor:
```
WARNING: ThreadSanitizer: lock-order-inversion (potential deadlock) (pid=73168)
  Cycle in lock order graph: M25 (0x0001013b40c8) => M27 (0x0001013b4108) => M25
```
Bu bize TSan'ın nispeten basit vakalar için bile yanlış pozitifler bildirdiğini söylüyor. Özellikle, algoritma yalnızca bir kilit grafiği oluşturmak ve içinde döngüler bulmak için basit bir işleme sahiptir.

6. Şimdi main-signal.c'ye bakalım. Bu kod, çocuğun bittiğini ve ebeveynin artık devam edebileceğini belirtmek için bir değişken (tamamlandı) kullanır. Bu kod neden verimsiz? (özellikle alt iş parçacığının tamamlanması uzun zaman alıyorsa, ebeveyn zamanını ne yaparak harcıyor?)
Ana dönüş, değişene kadar değerini kontrol ederek bekler . Bu, başka bir işlem (hatta zamanlayıcı her iki iş parçacığını aynı işlemcide programlamaya çalışıyorsa çalışan iş parçacığı) tarafından kullanılabilecek işlemci zamanını boşa harcar.

7. Şimdi bu programda helgrind'i çalıştırın. Ne rapor ediyor? Kod doğru mu?
Bunun yerine TSan'ı çalıştıracağım (1. soruya bakın). TSan bir veri yarışı ve bir iş parçacığı sızıntısı bildirir (ana iş parçacığı aramaz `Pthread_join()`). TSan doğrudur (bir veri yarışı vardır), ancak talimatların yeniden sıralanmadığı ve böylece ayarın donestandart çıkıştan önce gerçekleşmesi için 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 mu, performans mı, yoksa her ikisi mi?
Önemsiz olmayan programlar için daha iyi bir çözümdür, çünkü sinyal mekanizmasına özellikler eklendikçe eşzamanlılık hatalarına neden olma olasılığı daha düşüktür ve sinyaller arasındaki bekleme süresi arttığında daha iyi

performans gösterir (koşul değişkeni bekleme veya verim döndürmez, ancak planlayıcıya iş parçacığının engellendiğini ve başka bir iş parçacığından bir sinyal alana kadar programlanmaması gerektiğini bildirir).

9. Bir kez daha main-signal-cv üzerinde helgrind'i çalıştırın. Herhangi bir hata bildiriyor mu?
Bunun yerine TSan'ı çalıştıracağım (1. soruya bakın). Bu kod, TSan'ın herhangi bir veri yarışı uyarısı atmasına neden olmaz (hala önemsiz bir şekilde düzeltilebilir iş parçacığı sızıntısı uyarısı verir).