# 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 kilit ve koşul değişkenleri kavramlarını birçok örnekle daha yavaş tanıttığını not etmeliyiz; Bu nedenle bu bölüm referans olarak daha iyi kullanılır.

CRUX: IŞ PARÇACIKLARI NASIL OLUŞTURULUR VE KONTROL EDILIR

İşletim sistemi iş parçacığı oluşturma ve kontrol için hangi arayüzleri sunmalıdır? Bu arayüzler, kullanım kolaylığının yanı sıra fayda sağlamak için 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 bu çok kolaydır: oluşturmaktır ve bu nedenle bir tür iş parçacığı oluşturma arabirimi bulunmalıdır. POSIX'te bu çok 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 çok da kötü değildir. Dört bağımsız değişken vardır: thread, attr, start routine ve arg. Birincisi, iş parçacığı, pthread t tipi bir yapıya işaret eder; bu yapıyı bu iş parçacığıyla etkileşim kurmak için kullanacağız ve bu nedenle başlatmak için pthread create() öğesine iletmemiz gerekiyor.

İkinci bağımsız değişken olan attr, bu iş parçacığının sahip olabileceği tüm öznitelikleri belirtmek için kullanılır. Bazı örnekler yığın boyutunu ayarlamayı veya belki de iş parçacığının zamanlama önceliği hakkında bilgi vermeyi içerir. Bir öznitelik, pthread attr init(); ayrıntılar için yönetim sayfasına bakın. Ancak, çoğu durumda, varsayılanlar iyi olacaktır; bu durumda, NULL değerini basitçe geçireceğiz.

Üçüncü argüman en karmaşık olanıdır, ancak gerçekten sadece şunu sormaktır: Bu iş parçacığı hangi işlevde çalışmaya başlamalıdır? C'de buna a diyoruz. **işlev işaretçisi** (**function pointer**) ve bu bize aşağıdakilerin beklendiğini söyler: void * türünde tek bir bağımsız değişken geçirilen (başlangıç rutininden sonra parantez içinde belirtildiği gibi) ve void türünde bir değer döndüren bir işlev adı (başlangıç yordamı) * (yani**, boşluk işaretçisi ( void pointer**).)).

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

Bunun yerine, rutin bir boşluk işaretçisini argüman olarak alsaydı, ancak bir tamsayı döndürseydi, şöyle görünürdü:

```
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 yürütülmeye başladığı işleve geçirilecek bağımsız değişkendir. Şunu sorabilirsiniz: Neden bu boşluk işaretçilerine ihtiyacımız var? Cevap oldukça basit: fonksiyon başlangıç rutinine bir argüman olarak bir boşluk işaretçisine sahip olmak, herhangi bir argüman türünde geçmemize izin verir; dönüş değeri olarak olması, iş parçacığının her türlü sonucu döndürmesini sağlar.

Şekil 27.1'deki bir örneğe bakalım. Burada sadece iki argüman iletilen, kendimizi tanımladığımız tek bir tipte paketlenmiş bir iş parçacığı oluşturuyoruz (myarg t). İş parçacığı, bir kez oluşturulduktan sonra, bağımsız değişkenini beklediği türe aktarabilir ve böylece bağımsız değişkenleri istediği gibi açabilir.

Ve işte burada! Bir iş parçacığı oluşturduktan sonra, gerçekten kendi çağrı yığınıyla tamamlanmış, programda mevcut olan tüm iş parçacıklarıyla aynı adres alanında çalışan başka bir canlı yürütücü varlığınız olur. Eğlence böylece başlar!

## 27.2 İplik Tamamlama

Yukarıdaki örnekte bir iş parçacığının nasıl oluşturulacağı gösterilmektedir. Ancak, bir iş parçacığının tamamlanmasını beklemek isterseniz ne olur? Tamamlanmasını beklemek için özel bir şey yapmanız gerekir; özellikle, rutin pthread join() öğesini çağırmanız gerekir.

```
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 }
```

Şekil 27.1: **İş parçacığı oluşturma (Creating a Thread)**

Bu rutin iki argüman gerektirir. Birincisi pthread t tipindedir ve hangi iş parçacığının bekleneceğini belirtmek için kullanılır. Bu değişken, iş parçacığı oluşturma yordamı tarafından başlatılır (pthread create()); etrafta tutarsanız, bu iş parçacığının sonlanmasını beklemek için kullanabilirsiniz.

İkinci bağımsız değişken, geri almayı beklediğiniz dönüş değerine bir işaretçidir. Rutin herhangi bir şeyi döndürebildiğinden, bir işaretçiyi boşluğa döndürmek için tanımlanır; pthread join() yordamı iletilen bağımsız değişkenin değerini değiştirdiğinden, yalnızca değerin kendisine değil, bu değere de bir işaretçi iletmeniz gerekir.

Başka bir örneğe bakalım (Şekil 27.2, sayfa 4). Kodda, tek bir iş parçacığı tekrar oluşturulur ve myarg t yapısı aracılığıyla birkaç bağımsız değişken geçirilir. Değerleri döndürmek için myret t türü kullanılır. İş parçacığının çalışması bittiğinde, bekleyen ana iş parçacığı

pthread join() routine1'in içinde, sonra geri döner ve iş parçacığından döndürülen değerlere, yani myret t'de ne varsa ona erişebiliriz.

Bu örnek hakkında dikkat edilmesi gereken birkaç nokta. Birincisi, çoğu zaman tüm bu acı verici paketlemeyi ve argümanların paketini açmak zorunda değiliz. Örneğin, bağımsız değişkeni olmayan bir iş parçacığı oluştursak, iş parçacığı oluşturulduğunda NULL'u bağımsız değişken olarak geçirebiliriz. Benzer şekilde, dönüş değerini umursamazsak NULL değerini pthread join() içine geçirebiliriz.

[1]1Not Burada sarmalayıcı işlevleri kullanıyoruz; özellikle, Malloc(), Pthread join() ve Pthread create() olarak adlandırıyoruz, bu da yalnızca benzer şekilde adlandırılmış küçük harfli sürümlerini çağırıyor ve

```
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: **Waiting for Thread Completion: (İplik Tamamlanmayı Bekliyor)**

İkincisi, sadece tek bir değerde (örneğin, uzun uzun bir int) geçiyorsak, bunu bir argüman olarak paketlememiz gerekmez. Şekil 27.3 (sayfa 5) bir örnek gösterir. Bu durumda, argümanları paketlemek ve yapıların içindeki değerleri döndürmek zorunda kalmadığımız için hayat biraz daha basittir.

Üçüncüsü, değerlerin bir iş parçacığından nasıl döndürüldüğü konusunda son derece dikkatli olunması gerektiğine dikkat etmeliyiz. Özellikle, hiçbir zaman iş parçacığının çağrı yığınında ayrılan bir şeye başvuran bir işaretçi döndürmeyin. Bunu yaparsanız, ne olacağını düşünüyorsunuz? (bir düşünün!) Burada, Şekil 27.2'deki örnekten değiştirilen tehlikeli bir kod parçası örneği verilmiştir.

```
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 ayrılır. Ancak, geri döndüğünde, değer otomatik olarak serbest bırakılır (sonuçta yığının kullanımı bu kadar kolaydır!) ve böylece bir işaretçiyi geri geçirir.

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: **Simpler Argument Passing to a Thread(Bir İş Parçacığına Giden Daha Basit Bağımsız Değişken)**

şimdi serbest bırakılmış bir değişkene her türlü kötü sonuca yol açacaktır. Kesinlikle, iade ettiğinizi düşündüğünüz değerleri yazdırdığınızda, muhtemelen (ama mutlaka değil!) şaşıracaksınız. Deneyin ve kendiniz öğrenin2!

Son olarak, bir thread oluşturmak için pthread create() öğesinin kullanılmasının ve ardından pthread join() öğesine anında çağrı yapılmasının bir thread oluşturmanın oldukça garip bir yolu olduğunu fark edebilirsiniz. Aslında, bu kesin görevi yerine getirmenin daha kolay bir yolu vardır; buna bir (**prosedür çağrısı**) **procedure call.** Açıkçası, genellikle birden fazla iş parçacığı oluşturacağız ve tamamlanmasını bekleyeceğiz, diğer-
bilge iplikleri kullanmanın pek bir amacı yoktur.

Çok iş parçacıklı olan tüm kodların birleştirme yordamını kullanmadığını unutmamalıyız. Örneğin, çok iş parçacıklı bir web sunucusu bir dizi çalışan iş parçacığı oluşturabilir ve ardından istekleri kabul etmek ve bunları süresiz olarak çalışanlara iletmek için ana iş parçacığını kullanabilir. Bu tür uzun ömürlü programların bu nedenle katılması gerekmeyebilir. Ancak, belirli bir görevi (paralel olarak) yürütmek için iş parçacıkları oluşturan paralel bir program, hesaplamanın bir sonraki aşamasından çıkmadan veya geçmeden önce tüm bu çalışmaların tamamlandığından emin olmak için büyük olasılıkla birleştirmeyi kullanacaktır.

## 27.1 Kilitler

İş parçacığı oluşturma ve birleştirmenin ötesinde, muhtemelen POSIX iş parçacığı kitaplığı tarafından sağlanan bir sonraki en kullanışlı işlev kümesi, **locks(kilitler)**. 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);
```

2Neyse ki derleyici gcc, böyle bir kod yazdığınızda muhtemelen şikayet edecektir, derleyici uyarılarına dikkat etmek için başka bir neden.

Rutinlerin anlaşılması ve kullanımı kolay olmalıdır. Bir kod bölgeniz olduğunda **critical section(kritik bölüm)**, ve bu nedenle doğru çalışmayı sağlamak için korunması gerekir, kilitler oldukça kullanışlıdır. Muhtemelen kodun neye benzediğini tahmin 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 kilidi başka hiçbir iş parçacığı tutmazsa, iş parçacığı kilidi alır ve kritik bölüme girer. Başka bir iş parçacığı gerçekten kilidi tutuyorsa, kilidi almaya çalışan iş parçacığı, kilidi alana kadar çağrıdan geri dönmez (kilidi tutan iş parçacığının kilit açma çağrısı yoluyla serbest bıraktığını ima eder). Tabii ki, belirli bir zamanda kilit alma işlevinin içinde bekleyen birçok iş parçacığı sıkışmış olabilir; Bununla birlikte, yalnızca kilidi alınmış olan iş parçacığı kilit açma çağrısında bulunmalıdır.

Ne yazık ki, bu kod iki önemli şekilde kırıldı. İlk sorun bir **lack of proper initialization(uygun başlatma eksikliği)**. Tüm kilitler, başlamak için doğru değerlere sahip olduklarından emin olmak için uygun şekilde başlatılmalı ve böylece kilit açma ve kilit açma çağrıldığında istenildiği gibi çalışmalıdır.

POSIX iş parçacıklarında, 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), pthread mutex init() öğesine aşağıdaki gibi bir çağrı yapmaktır:

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

Bu rutinin ilk argümanı kilidin kendisinin adresidir, ikincisi ise isteğe bağlı bir öznitelikler kümesidir. Nitelikler hakkında daha fazla bilgi edinin; NULL değerini iletmek yalnızca varsayılanları kullanır. Her iki şekilde de çalışır, ancak genellikle dinamik (ikincisi) yöntemi kullanırız. Kilitle işiniz bittiğinde pthread mutex destroy() öğesine karşılık gelen bir çağrının da yapılması gerektiğini unutmayın; Tüm ayrıntılar için kılavuz sayfasına bakın.

Yukarıdaki kodla ilgili ikinci sorun, kilit ve kilit açma çağırırken hata kodlarını kontrol edememesidir. Tıpkı bir UNIX sisteminde aradığınız hemen hemen her kütüphane rotası gibi, bu rutinler de başarısız olabilir! Kodunuz hata kodlarını düzgün bir şekilde denetlemezse, 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 sofistike (oyuncak olmayan) programlar, başarısızlığı kontrol etmeli ve bir çağrı başarılı olmadığında uygun bir şey yapmalıdır.

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

Şekil 27.4: **An Example Wrapper(Örnek bir sarmalayıcı)**

Kilitleme ve kilit açma rutinleri,

kilitlerle etkileşim kurmak için pthreads kütüphanesi. İlgilenilen diğer iki 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. Trylock sürümü, kilit zaten tutulmuşsa arızayı yeniden döndürür; bir kilidi edinmenin timedlock sürümü, zaman aşımından sonra veya kilidi aldıktan sonra (hangisi önce gerçekleşirse) geri döner. Böylece, sıfır zaman aşımına sahip timedlock, trylock kasasına dejenere olur. Bu versiyonların her ikisinden de genellikle kaçınılmalıdır; Bununla birlikte, bir kilit edinme rutininde sıkışıp kalmaktan kaçınmanın (belki de kesinlikle) gelecekteki bölümlerde göreceğimiz gibi (örneğin, çıkmazı incelediğimizde) yararlı olabileceği birkaç durum vardır.

## 27.2 Koşul Değişkenleri

Herhangi bir iş parçacığı kitaplığının diğer önemli bileşeni ve kesinlikle POSIX iş parçacıklarında olduğu gibi, bir **condition variable(koşul değişkeni)**. Kondisyon değişkenleri, bir tür sinyalleşmenin gerçekleşmesi gerektiğinde kullanışlıdır

iş parçacıkları arasında, bir iş parçacığı diğerinin bir şey yapmasını bekliyorsa, bu durum devam edebilir. 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 kilidin olması gerekir. Yukarıdaki rutinlerden herhangi birini ararken, bu kilit tutulmalıdır.

İlk rutin olan pthread cond wait(), çağıran iş parçacığını uyku moduna geçirir ve böylece genellikle programda şu anda uyuyan iş parçacığının umursayabileceği 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 başlatıldıktan sonra, bir iş parçacığı ready değişkeninin henüz sıfırdan başka 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 verirken (ve global değişkeni hazır olarak değiştirirken), her zaman kilidin tutulduğundan emin oluruz. Bu, yanlışlıkla kodumuza bir yarış durumu eklemememizi sağlar.

İkincisi, bekleme çağrısının ikinci parametresi olarak bir kilit aldığını, sinyal çağrısının ise yalnızca bir koşul aldığını fark edebilirsiniz. Bu farkın nedeni, bekleme çağrısının, çağrı iş parçacığını uykuya yatırmanın yanı sıra, söz konusu arayanı uyuturken kilidi serbest bırakmasıdır. Öyle olmadığını hayal edin: diğer iplik kilidi nasıl alabilir ve uyanması için sinyal verebilir?

Bununla birlikte, uyandırıldıktan sonra geri dönmeden önce, pthread cond wait() kilidi yeniden alır, böylece bekleme ipliği, bekleme sırasının başlangıcındaki kilit kazanımı ile sonunda kilit serbest bırakma arasında her çalıştığında, kilidi tutar.

Son bir tuhaflık: Bekleme iş parçacığı, basit bir if ifadesi yerine bir süre döngüsünde koşulu yeniden kontrol eder. Gelecekteki bir bölümde koşul değişkenlerini incelerken bu konuyu ayrıntılı olarak tartışacağız, ancak genel olarak, bir süre döngüsü kullanmak yapılacak basit ve güvenli bir şeydir. Koşulu yeniden denetlemesine rağmen (belki de biraz ek yük ekleyerek), bekleyen bir iş parçacığını sahte bir şekilde uyandırabilecek bazı pthread uygulamaları vardır; böyle bir durumda, tekrar kontrol etmeden, bekleme ipliği, durumun değişmemiş olmasına rağmen değiştiğini düşünmeye devam edecektir. Bu nedenle, uyanmayı mutlak bir gerçek olmaktan ziyade bir şeylerin değişmiş olabileceğine dair bir ipucu olarak görmek daha güvenlidir.

Bazen bir koşul değişkeni ve ilişkili kilit yerine iki iş parçacığını aralamak için basit bir bayrak kullanmanın cazip geldiğini unutmayın. Örneğin, bekleme kodunda daha fazla görünmesi için yukarıdaki bekleme kodunu yeniden yazabiliriz:

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

İlişkili sinyal kodu şöyle görünür:

```
ready = 1;
```

3Statik 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, birçok durumda kötü performans gösterir (uzun süre eğirmek sadece CPU döngülerini boşa harcar). İkincisi, hataya eğilimlidir. Son araştırmaların gösterdiği gibi [X + 10], iş parçacıkları arasında senkronize etmek için bayrakları (yukarıdaki gibi) kullanırken hata yapmak şaşırtıcı derecede kolaydır; Bu çalışmada, bu geçici senkronizasyonların kullanımlarının yaklaşık yarısı buggy! Tembel olmayın; koşul değişkenlerini kullanmadan kurtulabileceğinizi düşündüğünüzde bile 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 bir fikre sahip olmak yeterli olmalıdır.

## 27.3 Derleme ve Çalıştırma

Bu bölümdeki tüm kod örneklerinin çalıştırılması ve çalıştırılması nispeten kolaydır. Bunları derlemek için kodunuza pthread.h üstbilgisini eklemeniz gerekir. Bağlantı satırında, -pthread bayrağını ekleyerek pthreads kitaplığıyla da açıkça bağlantı kurmanız gerekir.

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

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

main.c pthreads üstbilgisini içerdiği sürece, artık eşzamanlı bir programı başarıyla derlediniz. İşe yarayıp yaramadığı, her zamanki gibi, tamamen farklı bir konudur.

## 27.4 Özet

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

Şimdi bölümü, çok iş parçacıklı kod yazarken sizin için yararlı olabilecek bir dizi ipucuyla 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, karmaşık (ve umarım, doğru ve performanslı) çok iş parçacıklı programlar oluşturmanıza olanak sağlamalıdır. İş parçacıklarının zor kısmı API'ler değil, eşzamanlı programları nasıl oluşturduğunuzun zor mantığıdır. Daha fazla bilgi edinmek için okumaya devam edin.

BIR KENARA: THREAD API YÖNERGELERI

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

**Keep it simple(Basit tutun).** Her şeyden önce, dişler arasında kilitlemek veya sinyal vermek için herhangi bir kod mümkün olduğunca basit olmalıdır. Zorlu iş parçacığı etkileşimleri hatalara yol açar

- **Minimize thread interaction.s(İş parçacığı etkileşimini en aza indirin )** Iş parçacıklarının etkileşim kurma yollarının sayısını minimumda tutmaya çalışın. Her etkileşim olmalıdır
  denenmiş ve gerçek apache'lerle dikkatlice düşünülmeli ve inşa edilmelidir (bunların çoğunu önümüzdeki bölümlerde öğreneceğiz).
- **Initialize locks and condition variables.( Kilitleri ve durum değişkenlerini başlatın)** Bunu yapmamak, bazen çalışan ve bazen çok garip bir şekilde başarısız olan koda yol açacaktır.
- **Check your return codes.( İade kodlarınızı kontrol edin)** Tabii ki, yaptığınız herhangi bir C ve UNIX programında, her bir dönüş kodunu kontrol etmelisiniz,

  ve burada da geçerli. Bunu yapmamak, tuhaf ve anlaşılması zor davranışlara yol açacak ve (a) çığlık atmanıza, (b) saçlarınızın bir kısmını çekmenize veya (c) her ikisini birden yapmanıza neden olacaktır

- **Be careful with how you pass arguments to, and return values from, threads(Bağımsız değişkenleri iş parçacıklarına nasıl geçirdiğinize ve iş parçacıklarından değer döndürdüğünüze dikkat edin)**
- **.** Özellikle, bir referansı ilettiğiniz her zaman
  yığın üzerinde ayrılmış bir değişken, muhtemelen yanlış bir şey yapıyorsunuzdur
- .**Each thread has its own stack.** . Yukarıdaki noktayla ilgili olarak, lütfen her iş parçacığının kendi yığınına sahip olduğunu unutmayın. Böylece, eğer bir
- Bir iş parçacığının yürüttüğü bazı işlevlerin içinde yerel olarak ayrılmış değişken, esasen bu iş parçacığına özeldir; başka hiçbir iş parçacığı (kolayca) ona erişemez. Iş parçacıkları arasında veri paylaşmak için, değerler **heap(yığın)** veya başka bir şekilde küresel olarak erişilebilen bir yerel ayar.
- **Always use condition variables to signal between threads(İş parçacıkları arasında sinyal vermek için her zaman koşul değişkenlerini kullanın).**

  Basit bir bayrak kullanmak genellikle cazip gelse de, bunu yapmayın.
- **Use the manual pages.( Kılavuz sayfalarını kullanın.)** Özellikle Linux'ta, pthread man sayfaları oldukça bilgilendiricidir ve önceki nüansların çoğunu tartışır.

  Buraya gönderildi, genellikle daha da ayrıntılı. Onları 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. Pren- tice 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!*

## Homework(ÖDEV) (Code)

Bu bölümde, bazı basit çok iş parçacıklı programlar yazacağız ve adı verilen belirli bir araç kullanacağız. **helgrind**, Bu programlardaki sorunları bulmak için.

Programların nasıl oluşturulacağı ve helgrind'in nasıl çalıştırılacağı hakkında ayrıntılar için ödev indirmesindeki README'yi okuyun.

### Questions(Sorular)

1. İlk inşa main-race.c. Kodu inceleyin, böylece koddaki (umarım açık) veri yarışını görebilirsiniz. Şimdi helgrind'i çalıştırın (valgrind yazarak--tool=helgrind main-race) yarışı nasıl rapor ettiğini görmek için. Doğru kod satırlarına işaret ediyor mu? Size başka hangi bilgileri veriyor?

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

```
$ DevToolsSecurity -enable
aksi takdirde macOS yetersiz izinlerden şikayet eder.
İkilinin TSan araçlarıyla oluşturulması ve ardından çalıştırılması gerekir:
$ clang main-race.c -o main-race -fsanitize=thread -fPIE -g -Wall
$ ./main-race
Bu, veri yarışını ve aşağıdakileri tanımlayan bir uyarı bildirir:
```

   paylaşılan belleğe tehlikeli bir şekilde erişen her iş parçacığındaki kod satırları

   paylaşılan belleğin sembol adı (balance)

   iş parçacığının oluşturulduğu kod satırı

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

   Yukarıdaki gibi, TSan'ı kullanacağız. İş parçacıklarından birinden racy erişimlerini kaldırmak ve TSan'ı çalıştırmak hiçbir çıktı üretmez ve çıkış kodu 0 ile geri döner.

   Paylaşılan değişkene yapılan güncelleştirmelerden yalnızca birinin etrafına bir kilit ekledikten sonra, TSan, değişkeni güncelleştirdiğinde iş parçacıklarından birinin kilit tuttuğunu ve bu kilidin nerede başlatıldığını da tanımlayan soru 1'deki gibi bir uyarı bildirir. Bu durumda, TSan çıkış kodu 134 ile döner.
   Paylaşılan değişkenlere yapılan her iki güncelleştirmenin etrafına bir kilit ekledikten sonra, TSan çıktı üretmez ve çıkış kodu 0 ile döner.

3. Şimdi main-deadlock.c'ye bakalım. Kodu inceleyin. Bu kodun kilitlenme olarak bilinen bir sorunu var (önümüzdeki bölümde çok daha derinlemesine tartışacağız). Hangi sorunu olabileceğini görebiliyor musunuz?

İplikler ilk kilitlerini aynı anda yakalarsa potansiyel bir kilitlenme vardır (yani, iplik 0 kapmak kilit 1 ve iplik 1 kapmak kilit 2, her ikisi de sahip olmadıkları kilidi beklerken sıkışıp kalır).

4. Şimdi bu kodda helgrind çalıştırın. Helgrind neyi rapor eder?

```
$ make main-deadlock
$ TSAN_OPTIONS=detect_deadlocks=1 ./main-deadlock
```

TSan aşağıdaki uyarıyı bildirir:

UYARI: İş Parçacığı Temizleyici: lock-order-inversion (potansiyel kilitlenme) (pid=71488)
Kilit sırasına göre döngü grafiği: M23 (0x0001001f4088) => M25 (0x0001001f40c8) => M23

5 Şimdi main-deadlock-global.c adresinde helgrind çalıştırın. Kodu inceleyin; main-deadlock.c'nin sahip olduğu aynı soruna sahip mi? Helgrind aynı hatayı bildirmeli mi? Bu size helgrind gibi aletler hakkında ne söylüyor?

Bunun yerine TSan'ı çalıştıracağım (bkz. soru 1). Yeni kod **main-deadlock-global.c**, **main-deadlock.c** ile aynı sorunlara sahip değildir: genel kilit bir kilitlenmenin oluşmasını önler. Bununla birlikte, TSan aşağıdaki uyarıyı bildirmektedir:

UYARI: ThreadSanitizer: lock-order-inversion (olası kilitlenme) (pid=73168)

Kilit sırasına göre döngü grafiği: M25 (0x0001013b40c8) => M27 (0x0001013b4108) => M25

Bu bize TSan'ın nispeten basit vakalar için bile yanlış pozitifler bildirdiğini söylüyor. Özellikle, algoritma sadece bir kilit grafiği oluşturmak ve içindeki döngüleri bulmak için basit bir işleme sahiptir.

6 Sonra main-signal.c'ye bakalım. Bu kod, çocuğun bittiğini ve ebeveynin artık devam edebileceğini belirtmek için bir değişken (bitti) kullanır. Bu kod neden verimsiz? (Ebeveyn, özellikle çocuk ipliğinin tamamlanması uzun zaman alıyorsa, zamanını ne yaparak geçirir?)

Ana spin, değişene kadar bitti değerini kontrol ederek bekler. Bu, başka bir işlem (veya zamanlayıcı her iki iş parçacığını da aynı işlemciye zamanlamaya çalışıyorsa çalışan iş parçacığı) tarafından kullanılabilecek işlemci zamanını boşa harcar.

7 . Şimdi bu programda helgrind çalıştırın. Neyi rapor ediyor? Kod doğru mu?

Bunun yerine TSan'ı çalıştıracağım (bkz. soru 1). TSan bir veri yarışı ve bir iş parçacığı sızıntısı bildirir (ana iş parçacığı Pthread_join()) olarak adlandırmaz). TSan doğrudur (bir veri yarışı vardır), ancak talimatların yeniden sıralanmadığı varsayılarak, programın doğruluğunu etkilemez, böylece ayar standart olarak yazdırmadan önce yapılır.

8 Şimdi main-signal-cv.c'de bulunan kodun biraz değiştirilmiş bir sürümüne bakın. Bu sürüm, sig-naling'i (ve ilişkili kilidi) yapmak için bir koşul değişkeni kullanır. Bu kod neden önceki sürüme tercih ediliyor? Doğruluk mu, performans mı, yoksa her ikisi de 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 arttığında daha iyi performans gösterir (koşul değişkeni beklemeyi veya verimi döndürmez, bunun yerine zamanlayıcıya iş parçacığının engellendiğini ve başka bir iş parçacığından bir sinyal alana kadar zamanlanmaması gerektiğini bildirir).

9 Helgrind'i bir kez daha main-signal-cv üzerinde çalıştırın. Herhangi bir hata bildiriyor mu?

Bunun yerine TSan'ı çalıştıracağım (bkz. soru 1). Bu kod, TSan'ın herhangi bir veri yarışı uyarısı atmasına neden olmaz (yine de önemsiz bir şekilde düzeltilebilir iş parçacığı sızıntısı uyarısını atar).