İleri İşletim Sistemleri

İplikler (Threads)

504071502 Ali Keleş
504041505 Burak Sekmen

Giriş

Modern bilgisayarlar aynı anda çok sayıda işlemi yerine getirmelidirler. Bir kullanıcı herhangi bir programı yürütülürken, diskten veri okunması ya da yazıcının kullanılması olağan durumlardır. Tek işlemcili mimarilerde işlemcinin aynı anda çalışan programlar arasında zaman paylaşımlı olarak kullanılması gerekmektedir. Çok işlemcili mimari dahi olsa, o an yürümekte olan program sayısının işlemci sayısından fazla olması beklenen bir durumdur. İşletim sistemi tasarımcıları paralel çalışmaya ilişkin gerçeklemeyi kolaylaştıracak bir *proses modeli* ortaya koymuşlardır. Yürütülecek tüm yazılımlar *sıralı prosesler* (ya da kısaca *prosesler*) halinde organize edilir. Proses, yürütülen programa ve program sayıcının, saklayıcıların(işlemci bağlamının), değişkenlerin geçerli değerlerine verilen isimdir. Gerçek işlemcide yürütülen proses belli zaman aralıklarıyla değiştirilir. Prosesin o an için yürütülüp yürütülmediğini *proses durumu* kavramı ile açıklamak istersek aşağıdaki durumlardan söz edilebilir:

- Ana Bellekte yüklü durumda olan prosesler:
  - Yürütülüyor(CPU'da o an o prosese ait bağlam yüklü)
  - Hazır durumda(yürütülebilir, fakat CPU'da o an için başka bir prosese ait bağlam yüklü)
  - Bloke durumda(yürütülemez durumda, harici bir olayın gerçekleşmesini bekliyor.)
- Sanal Bellekte yüklü durumda olan prosesler:
  - Hazır durumda(yürütülebilir fakat ana bellekte yer olmadığından dolayı sanal bellekte)
  - Bloke durumda(yürütülemez durumda ve ana bellekte yer olmadığından dolayı sanal bellekte)
- Yukarıdaki durumların dışında yeni yaratılmış prosesler için *yeni* sonlandırılmış prosesler için *sonlandırılmış* durumlarından söz edilebilir.

Her bir prosese ait, aşağıdaki kaynaklar bulunmaktadır:

- Proses ID, proses grubuID, kullanıcı ID, ve grup ID
- Değişkenler
- Çalışma Dizini.
- Program Sayıcı
- Saklayıcılar
- Yığın
- Heap
- Açık dosyalar
- Sinyaller
- Paylaşılan kütüphaneler
- IPC araçları (örneğin mesaj kuyrukları, pipe'lar, semaforlar, paylaşılan bellek).

Yukarıda sıralanmış kaynaklara bakıldığında proses yaratmanın işletim sistemi için çok hızlı yapılabilecek bir işlem olmadığı ortadadır. Paralel çalışabilecek fakat aynı adres uzayını kullanan iş parçacıklarının olduğu bir model verimli programların yazılmasına olanak sağlayabilecektir. Bu nedenle *thread(iplik)* modeli geliştirilmiştir.

İplikler

İplikler proses kaynakları içinde var olurlar ve proses kaynaklarını kullanırlar. Bir proses içerisindeki iplikler bağımsız paralel koşabilme yeteneğine sahiptirler.

Her bir ipliğin:

- Sırada yürütülecek komuta işaret eden program sayıcısı,
- O an kullandığı değişkenleri tuttuğu saklayıcıları,
- Fonksiyon çağrıları için yığını bulunmaktadır.

İplikler proseslere göre çok daha hızlı yaratılabilmektedirler, çünkü bir prosese göre ihtiyaç duydukları kaynaklar çok daha azdır. Prosesler tarafından yaratılırlar, yaratan proses sonlandığında iplikler de sonlanır. Bir proses içindeki iplikler aynı kaynakları paylaştığından bir ipliğin yaptığı değişiklikten diğer bir iplik etkilenebilir(örneğin bir dosyanın kapatılması). İki farklı işaretçi aynı veriye işaret edebilir. Aynı bellek alanı okunup yazılabilir, dolayısıyla programcının yarış durumlarına dikkat ederek geliştirme yapması gerekir.

İplik Gerçekleme Yöntemleri

Bu bölümde ipliklerin gerçeklenme yöntemleri anlatılacaktır. İplikler kullanıcı modunda, çekirdek modunda ve her ikisinin de kullanıldığı hibrid yöntemle gerçeklebilirler. İpliklerin bu modlarda gerçeklenmelerini anlatmadan önce kullanıcı modu ile çekirdek modu arasındaki farkı belirtmek faydalı olacaktır.

A) Çekirdek Modu

Çekirdek modunda tek bir adres uzayı vardır. İşlemcinin bazı özel komutları yalnızca çekirdek modunda çalışır. Kesmeleri kapatan komut bu özel komutlardan birisidir. İşletim sistemleri bu çekirdek modunda çalışırlar.

İşlemcilerin çekirdek modundan kullanıcı modlarını geçmeleri durum kütüklerinde saklanan bit değerleri ile sağlanır.

B) Kullanıcı Modu

Tek bir adres uzayı kullanılmasının getirdiği bazı zorluklar vardır. Bunlarda bir tanesi de farklı uygulamaların hatalı kod parçaları nedeniyle birbirlerini etkilemesidir. Örneğin C dilinde Java dilinde olduğu gibi dizilere erişilen indekslerin kontrolü yoktur. Bir kod hatası nedeniyle dizinin gerçekte var olan elemanından daha yüksek adresli bir alana yazma erişimi yapılabilir. Böyle bir durumda farklı bir uygulamanın bellek alanı bozulacaktır. Bu tür hataları engellemek nedeniyle işletim sistemleri sanal bellek kullanırlar. Prosesler ve iplikler de bu sanal bellekler üzerinde çalışırlar. Bu şekilde bir proses başka bir prosesin bellek alanına doğrudan erişemez. İplikler durum biraz daha farklıdır. İplikler yaratıldıkları prosesin adres uzayını kullandıkları için birbirlerinin bellek alanlarına erişebilirler. Farklı yazılım modülleri birbirleri ile iletişim içinde olmasalar dahi kod hataları nedeniyle birbirlerinin çalışmalarını bu şekilde etkileyebilirler.

Bu bilgiyi verdikten sonra, ipliklerin gerçekleme yöntemlerinden ilki olan kullanıcı seviyesinde iplik gerçekleme yöntemi anlatılabilir.

Kullanıcı Seviyesinde İplik Gerçekleme

İplikler işletim sistemi desteğinden bağımsız şekilde yönetilir. Bu sayede işletim sistemi iplik yapısını desteklemese dahi iplikler kullanılabilir. Bir iplik bloke olacak bir sistem çağrısı yaptığında, kullanıcı uzayında iplik değişimini yapacak olan fonksiyon çağrılır. İşlemci bağlamı değişimi çekirdek çağrısı yapılmadan bu fonksiyon ile gerçekleştirilir. İplikleri kullanıcı modunda gerçeklerken bazı zorunlar ortaya çıkacaktır. Aşağıda bu sorunlar ve önerilen çözümler anlatılmıştır:

a) İplik kendi isteği dışında işlemci zamanını harcamaya devam edebilir. İşletim sistemi periyodik zaman dilimlerinde oluşan kesmeler sayesinde görev değişimi "task switching" yapar. Önceliği yüksek olan prosesin çalışması bu kesme sayesinde gerçeklenir. Kullanıcı düzeyinde bu kesme yönetilemeyeceğinden, bir iplik kendi istemediği sürece prosesin zaman dilimini kullanmaya devam eder. Bu sorunda çözüm önerisi olarak şu yöntem verilebilir. İşletim sisteminden bir zamanlayıcı isteği yapılabilir. Sistem çağrıları yapılacağı için yavaşlamaya sebep olacaktır.
b) Bir iplik diğer tüm iplikleri bloke edebilir. Sistem çağrısı yapan iplik ile birlikte diğer tüm iplikler bloke olabilir. Sistem çağrıları üstünde bir yazılım katmanı oluşturulabilir. İpliği bloke edecek olan sistem çağrıları bu ara katman ile bekletilebilir.
c) İpliklerin oluşturacağı bir kesme nedeniyle proses ve dolayısıyla diğer tüm iplikler bloke olabilir. Sayfalama hatları: Erişilmek istenen bir verinin ya da kod alanının bulunduğu sayfa için bir TLB tanımlanmamışsa olası bir kesme hizmet programı çalışmaya başlar. Kullanıcı düzeyinde gerçeklenen ipliklere bu sorun için bir çözüm yöntemi yok.

Çekirdek Seviyesinde İplik Gerçekleme

İpliklerin bilgileri çekirdek düzeyinde tutulur. İpliklerin yönetimi için sistem çağrıları vardır. Kullanıcı uzayında görülen problemler ortadan kalkar. Sistem çağrıları kullanıldığı için kullanıcı uzayında gerçekleme yöntemine göre yavaştır. Windows işletim sistemlerinde bu gerçekleme yöntemi kullanılmaktadır.

Hibrid Yöntem ile İplik Gerçekleme

Çekirdek seviyesinde, kullanıcı seviyesindeki iplik isteklerini karşılamak için belirli sayıda iplik bulunur. Çoklu işlemcili sistemlerde çekirdek seviyesindeki bu iplikler farklı işlemciler koşabilir.

Hibrid gerçekleme yönteminde işletim sisteminin iplik desteği vermesi beklenmektedir. Çekirdek düzeyinde koşan N tane iplik üzerinde, kullanıcının isteğine bağlı olarak çok sayıda iplik gerçeklenebilir. Bu iplik gerçekleme yönteminin kullanıcı seviyesinde gerçeklenen iplik modeline göre bir avantajı da çoklu işlemcilerde çekirdek seviyesindeki ipliklerin her birinin farklı işlemcilere dağıtılabilmesidir.

Pthread Kitaplığı

İpliklerin gerçeklenmesinde geçmişte donanım geliştiren firmaların kendi gerçeklemeleri kullanılmaktaydı. Bu durumda her bir platform için farklı iplik gerçeklemesi kullanıldığından yazılımların taşınabilirliğini zorlaştırmaktaydı. İplik kullanımının avantajlarından tam olarak yararlanabilmek amacıyla, standart bir programlama arayüzüne ihtiyaç duyulmaktaydı. UNIX sistemler için bu standart arayüz IEEE POSIX 1003.1c standardı (1995) olarak kaydedildi. Bu standarda bağlı gerçeklemelerde kullanılan ipliklere POSIX iplikleri adı verildi. Şu an çok sayıda donanım geliştirici POSIX standardını desteklemektedir.

POSIX iplikleri Pthread API'si kullanılarak geliştirilebilir. GNU Linux'da Pthread API'si kullanarak geliştirilen bir program aşağıdaki şekilde derlenebilir:

gcc –pthread kaynak.c –o cikis

Pthread API'sinde yaklaşık 60 kadar fonksiyon bulunmaktadır. Bu fonksiyonlar genel olarak 3 grupta toplanabilir:

1. ***İplik yönetimi:*** Doğrudan doğruya iplikleri yaratan, birleştiren,ayıran özelliklerini tanımlayan fonksiyonların bulunduğu grup.
2. ***Mutex'ler:*** Senkronizasyona, karşılıklı dışlamalara dair fonksiyonların bulunduğu grup.
3. ***Koşul Değişkenleri:*** Bir mutex'i paylaşan iplikler arasındaki haberleşmeye dair fonksiyonların bulunduğu grup.

Pthread Adlandırma Konvansiyonu

Rutin Öncülü	Fonksiyonel Grup
pthread_	iplikler ve diğer genel fonksiyonlar
pthread_attr_	iplik özellik nesneleri
pthread_mutex_	mutex'ler
pthread_mutexattr_	mutex özellik nesneleri.
pthread_cond_	koşul değişkenleri
pthread_condattr_	koşul özellik nesneleri
pthread_key_	İpliğe özgü veri anahtarları

İplik Yönetimi

İplik Yaratma ve Sonlandırma

Sık kullanılan rutinler:

pthread_create (thread,attr,start_routine,arg)

pthread_exit (status)

pthread_attr_init (attr)

pthread_attr_destroy (attr)

İplik Yaratma

Başlangıç durumunda main() fonksiyonu tek ipliktir. `pthread_create` fonksiyonuyla yeni iplikler yaratılır. `pthread_create` şu argümanları alır:

- `thread`: `pthread_create` fonksiyonu tarafından iplik üzerinden yeni yaratılan iplik için bir benzersiz ID döndürülür.
- `attr`: İpliğin özelliklerinin kurulabileceği bir özellik objesidir. NULL değeri varsayılan özelliklerle yaratılmaya yol açar.
- `start_routine`: Yaratıldığı zaman ipliğin yürüteceği C rutini.
- `arg`: start_routine fonksiyonun bir adet void işaretçi üzerinden erişilebilen bir parametre yollanabilir. Parametre yollanmıyorsa arg NULL değerini almalıdır.

İpliklerin Sonlanması

Bir Pthread aşağıdaki durumlardan biriyle sonlanabilir:

- İplik start_routine'den geri döner.(ana rutin)
- İplik `pthread_exit` fonksiyonunu çağırır.
- İplik başka bir ipliğin `pthread_cancel` çağrısıyla sonlanır.
- İpliği çalıştıran proses sonlanır.

ÖRNEK KOD 1: İplik yaratma ve sonlandırma

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

ÖRNEK KOD 2: İpliğe parametre yollama

Aşağıdaki kodda yaratılacak her bir ipliğe benzersiz bir *taskid* parametresi gönderilir.

```
long *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
   taskids[t] = (long *) malloc(sizeof(long));
   *taskids[t] = t;
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
   ...
}
```

İpliklerin Birleştirilmesi ve Ayrılması

Sık kullanılan rutinler:

pthread_join (threadid,status)

pthread_detach (threadid,status)

pthread_attr_setdetachstate (attr,detachstate)

pthread_attr_getdetachstate (attr,detachstate)

- İpliklerin senkronizasyonu için birleştirme işlemi kullanılabilir. `pthread_join()` rutinini çağıran ipliği `threadid` parametresi ile belirtilen iplik sonlanıncaya kadar bloke eder.
- Hedef ipliğin sonlanma durumu, `pthread_exit()` fonksiyonunda belirtilmişse, programcı hedef ipliğin sonlanma durumundan haberdar olur..
- Bir iplik üzerinde sadece bir adet `pthread_join` olmalıdır. Birden fazla birleşme girişimi mantıksal hataya neden olacaktır.

İpliğin birleşebilir bir iplik olup olmadığı iplik özelliklerinde tanımlanmaktadır. Sadece birleşebilir ipliklerde birleşme işlemi yapılabilir, ayrık ipliklerde yapılamaz.

İpliğin ayrık yada birleşebilir türden yaratılması için aşağıdaki işlemler yapılmalıdır:

`pthread_create()` rutininin attr parametresi kullanılacaktır. 4 adımlı tipik bir prosedürle attr değeri uygun şekilde kurulabilir:

1. `pthread_attr_t` veri tipinden bir özellik değişkeni tanımlanır.
2. `pthread_attr_init()` fonksiyonuyla bu değişken ilklendirilir.
3. `pthread_attr_setdetachstate()` fonksiyonuyla ayrıklık durumu atanır.
4. İşlem bitince `pthread_attr_destroy()` fonksiyonuyla kaynaklar boşaltılır.

Birleşebilir özelikte yaratılmış bir thread `pthread_detach()` ile ayrık hale getirilebilir.

ÖRNEK KOD 3: Threadlerin birleştirilmesi

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS    4

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
   printf("Thread %ld starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n",tid, result);
   pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;

   /* İlklendir ve birleşibilir özelliği ata */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from pthread_create()
                is %d\n", rc);
         exit(-1);
         }
      }

   /* özelliği bırak ve diğer iplikleri bekle */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join()
                is %d\n", rc);
         exit(-1);
         }
      printf("Main: completed join with thread %ld having a status
            of %ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
```

```
pthread_exit(NULL);
}
```

Mutex Değişkenleri

Mutex'ler yarış durumlarını önlemek için kullanılan, ismini İngilizce karşılıklı dışlama ilkesinden alan(mutual exclusion) özel değişkenlerdir. Ortak değişkenlere erişim esnasında mutex kullanımı gerekebilmektedir.

Mutex kullanımında tipik olarak aşağıdaki işlemler yapılmaktadır

- Mutex değişkenini yaratılır ve ilklendirilir.
- Çok sayıda iplik mutex'i kilitlemeye çalışır.
- Sadece bir iplik mutex'i kilitler ve sahibi olur
- Mutex'e sahip iplik işlemlerini yapar
- Sahip iplik mutex'i bırakır
- Başka bir iplik mutex'i kilitler ve işlemlerini yapar
- Sonunda mutex yokedilir.

Mutex için yarışan fakat kaybeden iplik bloke durumu geçer. *lock* fonksiyonu yerine *trylock* fonksiyonu tercih edilirse iplik bloke duruma geçmez.

Mutex Yaratma ve yok etme

Sık kullanılan rutinler:

pthread_mutex_init (mutex,attr)

pthread_mutex_destroy (mutex)

pthread_mutexattr_init (attr)

pthread_mutexattr_destroy (attr)

`pthread_mutex_t` tipinden mutex değişkeni yaratılmalıdır ve `pthread_mutex_init mutex,attr)` fonksiyonuyla ilklendirilmelidir. Mutex değişkeni, deklarasyon esnasında statik olarak da ilklendirilebilir:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

Mutex ilk olarak serbest durumdadır. Mutex değişkeninin özelikleri `pthread_mutex_init()` fonksiyonunun attr parametresi kullanılarak kurulabilir. `pthread_mutex_destroy()` rutini ile bir daha kullanılmayacak olan mutex değişkeni yok edilebilir.

Mutex Kilitleme ve Serbest Bırakma

Sık kullanılan rutinler:

pthread_mutex_lock (mutex)

pthread_mutex_trylock (mutex)

pthread_mutex_unlock (mutex)

- The `pthread_mutex_lock()` rutini ipliğin mutex'i aldığı rutindir.Mtuex değişkeni başka bir iplik tarafından alınmış ise, istekte bulunan iplik bloklanır ve mutex'in boşalmasını bekler.
- `pthread_mutex_trylock()` mutex'i almaya çalışır. Mutex değişkeni başka bir iplik tarafından alınmış ise bloklanma olmaz.
- `pthread_mutex_unlock()` mutex'i diğer iplikler için serbest bırakır.

ÖRNEK KOD 4: MUTEX kullanımı

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


typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Global değişkenleri ve mutexi tanımla */

#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

/*
dotprod fonksiyonu iplik yaratıldığında koşmaya başlar. DOTDATA struct'i
tipinden giriş parametresi alır ve fonsiyonun çıkışı da bu struct'a
yazılır. Bu yaklaşımın çok iplikli programlamada şöyle bir faydası vardır:
Bir iplik yaratıldığında aktive edilenfonksiyona sadece bir adet parametre
gönderilir- genelde ipliğin numarası. Bunun dışındaki fonsiyonun ihtiyaç
duyabileceği tüm bilgiler global olarak erişilebilen bir struct'tan
aktarılır.
*/

void *dotprod(void *arg)
{

   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
```

```
   y = dotstr.b;

   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

   /*
   Mutexi kilitle, veri güncellendikten sonra kilidi bırak
   */
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum);

   pthread_exit((void*) 0);
}

/*
Ana program bütün işi yapacak olan iplikleri yaratır. İşlemler bittikten
sonra sonucu ekrana yazar. İplikleri yaratmadan önce giriş datası
yaratılır. Tüm iplikler paylaşılan bir veriyi güncellediğinden karşılıklı
dışlama için mutex kullanılır. Ana iplik diğer tüm ipliklerin bitmesini
beklemelidir. Bu yüzden birleşme kullanılır.
*/

int main (int argc, char *argv[])
{
   long i;
   double *a, *b;
   void *status;
   pthread_attr_t attr;

   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

   for (i=0; i<VECLEN*NUMTHRDS; i++)
    {
     a[i]=1.0;
     b[i]=a[i];
    }

   dotstr.veclen = VECLEN;
   dotstr.a = a;
   dotstr.b = b;
   dotstr.sum=0;

   pthread_mutex_init(&mutexsum, NULL);

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

        for(i=0; i<NUMTHRDS; i++)
        {

        pthread_create(&callThd[i], &attr, dotprod, (void *)i);
        }

        pthread_attr_destroy(&attr);

        /* diğer iplikleri bekle */
```

```
        for(i=0; i<NUMTHRDS; i++)
        {
          pthread_join(callThd[i], &status);
        }

   /* birleşmeden sonra sonucu ekrana bas, temizlik yap */
   printf ("Sum =  %f \n", dotstr.sum);
   free (a);
   free (b);
   pthread_mutex_destroy(&mutexsum);
   pthread_exit(NULL);
}
```

Koşul Değişkenleri

Koşul değişkenleri senkronizasyonda kullanılan başka bir yöntemdir. Mutexler ipliklerin verilere erişimini denetlerken koşul değişkenleri verinin o anki değerinin kullanıldığı bir senkronizasyon mekanizmasını sağlar. Koşul değişkenlerinin olmadığı bir gerçeklemede, ipliklerin belli koşulların gerçekleşip gerçekleşmediğini anlamalarının tek yolu taramalı çalışmadır. Bu da işlemci zamanının fazla tüketilmesine neden olacaktır. Koşul değişkenleri mutex kilitleriyle birlikte kullanılırlar. Koşul değişkenlerinin nasıl kullanıldığı aşağıdaki tabloda gösterilmektedir.

Ana İplik	
○ Senkronizasyon gerektiren global değişkenleri tanımla ilk değerlerini ver(örnek: count değişkeni) ○ Bir koşul değişkeni tanımla ve ilklendir. ○ Bir mutex tanımla ve ilklendir. ○ A ve B ipliklerini yarat	
A İpliği	**B İpliği**
○ Belirli bir koşul oluşuncaya kadar işlem yap (örnek: count değişkeninin belli bir değere gelmesi) ○ Mutex'i kilitle, global değişkenin değerini kontrol et ○ `pthread_cond_wait()` ile B ipliğinden gelecek sinyali bloklanarak bekle. pthread_cond_wait() çağrısı otomatik olarak mutex'i bırakır ve B ipliğinin kullanması sağlanır. ○ Sinyal gelince uyan, mutex sinyal gelince otomatik olarak kilitlendi ○ Mutex'i bırak ○ Devam Et	○ İşlem yap ○ Mutex'i kilitle ○ A ipliğinin beklediği global değişkenin değerini değiştir. ○ Global değişkenin değerini control et, beklenen değere geldiyse A ipliğini sinyalle ○ Mutex'i bırak ○ Devam et.

Ana İplik
Birleş/ Devam et

Koşul Değişkeni Yaratma ve Yok etme

Sık kullanılan rutinler:

```
pthread_cond_init (condition,attr)

pthread_cond_destroy (condition)

pthread_condattr_init (attr)

pthread_condattr_destroy (attr)
```

Koşul değişkenleri `pthread_cond_t` tipinden yaratılmalı ve ilklendirilmelidirler. Bu ilklendirme şu şekillerde yapılabilir

1. Deklarasyon esnasında statik olarak:
   `pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;`
2. `pthread_cond_init()` rutini ile dinamik olarak: attr parametresi ile koşul değişkeninin özelliği tanımlanabilir.

`pthread_cond_destroy()` ile kullanılmalarının ardından koşul değişkenleri yok edilirler.

Koşul Değişkenlerinde Bekleme ve Sinyalleme

Sık kullanılan rutinler:

pthread_cond_wait (condition,mutex)

pthread_cond_signal (condition)

pthread_cond_broadcast (condition)

`pthread_cond_wait()` çağıran ipliği koşula sinyal gelene kadar bloke eder. Bu rutin mutex kilitlenip çağrılmalıdır ve bekleme esnasında mutex'i otomatik olarak bırakır. Sinyal geldikten sonra iplik uyanır ve mutex otomatik olarak kilitlenir. Mutexteki kilidi işlemler bittikten sonra çözmek programcının sorumluluğundadır.

`pthread_cond_signal()` rutini koşul değişkeni üzerinden beklemekte olan bir ipliğe sinyal göndermek(veya uyandırmak) amacıyla kullanılır. Mutex kilitlendikten sonra çağrılmalıdır. Uyuyan ipliğin `pthread_cond_wait()` fonksiyonunu tamamlayabilmesi için mutex daha sonra bırakılmalıdır.

`pthread_cond_broadcast()` rutini birden fazla ipliğe sinyal göndermek için kullanılır.

ÖRNEK KOD 6: Koşul Değişkeni Kullanımı

Bu kodda birden fazla koşul değişkeninin kullanımı gösterilmiştir. Ana rutin 3 adet iplik yaratır. 2 iplik birtakım işler yapar ve *count* değişkenini günceller. Üçüncü iplik *count* değişkeninin belli bir değere gelmesini bekler.

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
  int j,i;
  double result=0.0;
  long my_id = (long)t;

  for (i=0; i<TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
     count un değerini kontrol et ve gerekli değere ulaşınca sinyal
gönderir.
    */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %ld, count = %d  Threshold reached.\n",
             *my_id, count);
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
           *my_id, count);
    pthread_mutex_unlock(&count_mutex);

    sleep(1);
    }
  pthread_exit(NULL);
}

void *watch_count(void *t)
{
  long my_id - (long)t;

  printf("Starting watch_count(): thread %ld\n", *my_id);

  /*
  Mutex'i kilitle ve sinyal bekle. Şayet count'un değeri buraya gelinmeden
COUNT_LIMIT'i geçtiyse burası direk geçilir.
  */
  pthread_mutex_lock(&count_mutex);
  if (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received.\n",
my_id);
```

```
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

   pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);

}
```

Kullanılan Kaynaklar

I. A. S. Tanenbaum, Modern Operating Systems 2nd Edition, Prentice Hall.
II. http://en.wikipedia.org/wiki/Threads, 2008
III. Blaise Barney, Lawrence Livermore National Laboratory, Posix Thread Programming, https://computing.llnl.gov/tutorials/pthreads/