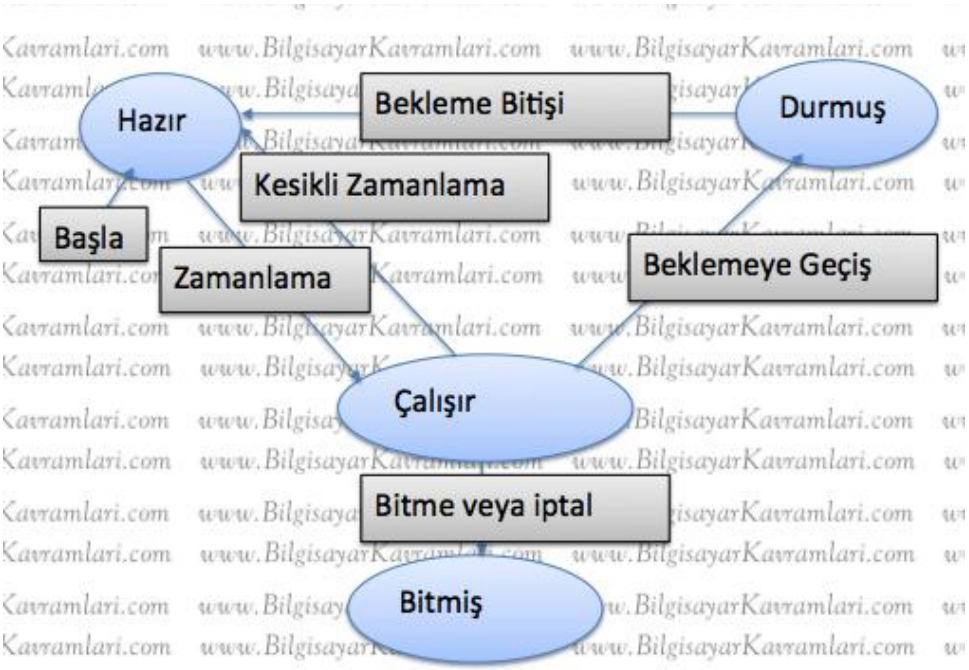
POSIX Thread pthread kütüphanesi – Bilgisayar Kavramları

Yazan : Şadi Evren ŞEKER

Bilgisayar bilimlerinde geçen lif (thread, iplik, sicim) (http://www.bilgisayarkavramlari.com/2010/03/22/thread-iplik/) kavramının C dili ile kodlanabilmesi için genellikle UNIX türevi işletim sistemelerinde geliştirilen programlama kütüphanesidir. Kütüphane UNIX ortamında (ve dolayısıyla LINUX ortamında da) POSIX kütüphanesi olarak geçmektedir ve bu kütüphanenin baş harfi olan P harfi ile thread kelimesinin birleşmesinden türemiştir (pthread).

Kütüphane kullanılarak temel lif işlemleri yapılabilir. Aşağıda, kütüphanede kullanılan bazı temel fonksiyonlar, tanımları ile birlikte verilecektir. Ardından meşhur, üretici / tüketici (producer consumer) (http://www.bilgisayarkavramlari.com/2012/03/05/producer-consumer-problem-uretici-tuketici-problemi/) örneği üzerinden nasıl uygulandığı gösterilecektir



Yukarıdaki şekilde 4 farklı durum arsındaki geçişler gösterilmiştir. Buna göre bir lif (thread) (http://www.bilgisayarkavramlari.com/2010/03/22/thread-iplik/) başlatıldıktan sonra, hazır durumuna geçer. Zamanlama algoritmasına bağlı olarak (scheduling algorithm) (http://www.bilgisayarkavramlari.com/2008/11/19/islemci-zamanlama-cpu-scheduling/), çalışır duruma geçebilir. Çalışma işlemi sırasında lif işini bitirirse veya iptal edilirse (kendi kendini iptal edebilir veya farklı bir lif tarafından iptal edilebilir) bitmiş duruma geçer. Çalışır durumdayken bir sebeple bekleme durumuna geçebilir. Örneğin farklı bir lifi bekleyebilir veya bir sistem kaynağına erişmek isteyebilir veya belirli bir süre için uyutulmuş olabilir. Durma sebebi ortadan kalktıktan sonra, hazır duruma geri geçer ve hazır sırasında (ready queue) (http://www.bilgisayarkavramlari.com/2008/11/19/bekleme-sirasi-ready-queue/) beklemeye başlar.

Konuyu örnek bir kod ile anlamaya çalışalım:

Örnek 1:

```
#include <pthread.h>
#include <stdio.h>
void *thread_routine(void* arg){
   printf("Yeni üretilen lif n");
}
void main(){
  pthread_tthread_id; void *thread_result;
  pthread_create( &thread_id, NULL, thread_routine, NULL );
  printf("Ana lifte n");
  pthread_join( thread_id, &thread_result);
}
```

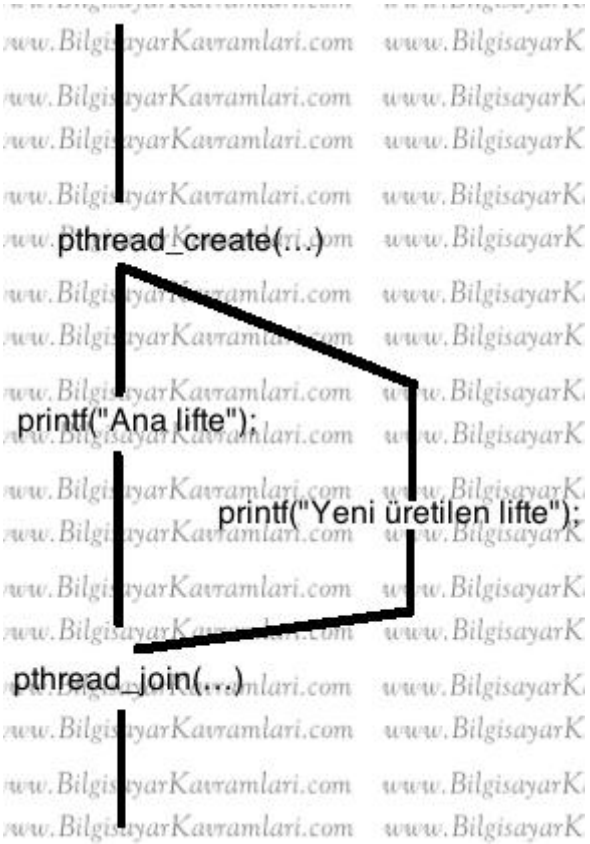
Yukarıdaki örnek kodun çalışması aşağıda verilmiştir.

```
SADIs-MacBook-Air:yedekler sadievrenseker$ gcc p.c -
lpthread
SADIs-MacBook-Air:yedekler sadievrenseker$ ./a.out
Ana lifte
Yeni üretilen lif
```

Yukarıdaki çalışma örneğinde görüldüğü üzere, derleme işlemi sırasında gcc derleyicisine -lpthread parametresi verilmiştir. Bunun sebebi derleme sırasında phtread kütüphanesinin bağlanması (link) gereğidir. Ardından derlenen kod çalıştırılmış ve ekrana sırasıyla “Ana lifte” ve “Yeni ğretilen lif” mesajlarını basmıştır.

Kodun çalışması sırasında main fonksiyonu içerisinde sırasıyla, önce bir lif bilgisini tutabilecek ve pthread_t yapısından (struct) (http://www.bilgisayarkavramlari.com/2007/11/08/olusum-composition-ve-struct-yapilar/) üretilen bir değişken tanımlanmıştır. Bu değişkeni daha sonraki örneklerde life doğrudan erişmek için kullanacağız. Bu örneğimizin en önemli satırı olan pthread_create fonksiyonunun ilk parametresidir. Bu parametre atıf ile çağırma (call by reference) (http://www.bilgisayarkavramlari.com/2009/01/12/atif-ile-cagirma-call-by-reference/) işlemi olarak düşünülmelidir ve aslında yapılan lif üretimi sırasında (pthread_create) parametre vererek, üretilen yeni lifin bilgisini almaktır.

Ardından ekrana “Ana lifte” mesajı basılmıştır. Burada dikkat edilecek husus, aslında yeni bir lif (thread) (http://www.bilgisayarkavramlari.com/2010/03/22/thread-iplik/) üretilirken en az iki lif bulunduğudur. Birisi yeni üretilen lif (thread) (http://www.bilgisayarkavramlari.com/2010/03/22/thread-iplik/) diğeri ise bu lifi üreten lif olan ana liftir (main thread). Bu durum aşağıdaki şekilde ifade edilebilir:



Temsili resimcikte görüldüğü üzere, program akışı içerisinde çağrılan bir pthread_create fonksiyonu ile hafızada iki farklı lif (thread) (http://www.bilgisayarkavramlari.com/2010/03/22/thread-iplik/) üretilmiş ve bunlar aynı anda çalışmaya devam etmiştir. Ana lifteki çalışma bittikten sonra, pthread_join ile, yeni üretilen lifin (thread) bitmesi beklenmiştir.

Hafızada anlık olarak iki lifin bulunduğu durum, yukarıdaki şekilde gösterilmiştir. Bu gösterimden anlaşılacağı üzere ana lif ve yeni lif için iki ayrı yığın (stack) alanı bulunmaktadır. Bu alanlarda liflerin kendisine özel bilgileri tutulmaktadır. Örneğin anlık olarak hangi fonksiyonun hangi satırını çalıştırdıkları, bu

fonksiyonu bitirdikten sonra hangi fonksiyona geri dönecekleri gibi bilgiler durur. Buna karşılık, lifin üretilmesi esnasında tanımlı olan bütün değişkenler, iki fonksiyon arasında paylaşılmıştır. Bu değerlere de paylaşılmış değişkenler (shared variable) ismi verilir.

Örnek 2:

```
#include <pthread.h>

#include <stdio.h>

#include <string.h>

void *thread_routine(void* arg){

   printf(“Inside newly created thread n”);

   return (void*) strdup(“Bir dizgi geliyor”);

}

void main(){

   pthread_tthread_id;

   void *thread_result=0; pthread_create( & thread_id, NULL,
thread_routine, NULL );

   printf(“Ana liftenn”);

   pthread_join( thread_id,&thread_result);

   if ( thread_result!= 0 )

      printf(“Ana life gelen %sn”, thread_result);

}
```

Yukarıdaki yeni kodda, pthread_join fonksiyonu, bir önceki örnekte olduğu gibi yeni üretilen lifi (thread) beklemek için kullanılmıştır. Bu kullanımda ikinci bir parametre de üretilen yeni liften gelen veriyi almak için atıf ile çağırma (call by reference) (http://www.bilgisayarkavramlari.com/2009/01/12/atif-ile-cagirma-call-by-reference/) şeklinde verilmiştir.

Kodda yeni olan ve koyu renkler ile ifade edilen kısımlara bakılacak olursa, thread_routine isimli fonksiyon, bir önceki koddan farklı olarak bir değeri döndürmektedir (return). Ayrıca bu fonksiyonun döndürdüğü değer, pthread_join fonksiyonunun ikinci parametresinde yakalanmıştır. Bu yakalama işlemi sırasında kullanılan değişkenin tipine dikkat edilirse void * olduğu görülür. Bu değişken tipi, C dilinde tipin belirsiz olması durumunda tercih edilir. Kısaca void * tipi, tipin belirsizliği anlamındadır. Gelen değer, if kontrolünden geçirilerek bir değer döndürülmesi halinde ekrana ana liften basılmaktadır.

Yukarıda kullanımlarından bahsettiğimiz fonksiyonların detayları aşağıda verilmiştir:

```
int pthread_create( pthread_t *tid, cons
pthread_attr_t *attr, void* funk, void * arg);
```

Yeni bir lif (thread) (http://www.bilgisayarkavramlari.com/2010/03/22/thread-iplik/) oluşturmak için kullanılan fonksiyondur. 4 parametre alır. Bunlar sırasıyla, oluşturulacak olan yeni lifin (thread) bilgisini tutan tid, lif oluşumu sırasında verilecek olan özellikler (attr), lifin çalıştıracağı fonksiyon (buradaki parametre bir fonksiyon göstericisi (function pointer) olarak verilmektedir), fonksiyon göstericisi olarak alınan parametreye verilecek olan parametreler. Yani lif, bir fonksiyonu çalıştırırken bu fonksiyona geçirilecek olan parametreler şeklindedir.

Fonksiyon değer olarak bir int döndürür ve şayet int değeri 0 ise başarılı, diğer durumlarda ise hata değerini taşımaktadır.

```
int pthread_exit(pthread_t *tid);
```

Parametre olarak aldığı lif bilgisini bitirir. Lifin çalışması sonlandırılarak sistemden kaldırılır.

```
int pthread_equal(pthread_t *tid1, pthread_t *tid2);
```

Parametre olarak aldığı iki lifi (thread) karşılaştırır ve sonuçta şayet eşitlerse 0 değilse eşitlik durumuna göre 0 dışında bir değer döndürür. Buradaki karşılaştırma iki lifin içeriğine bakılarak yapılmaktadır ve derin karşılaşıtrma (deep compare) olarak kabul edilebilir. Sığ karşılaştırma (shallow compare) (http://www.bilgisayarkavramlari.com/2011/03/19/nesne-kopyalama/) için == işlemi (operator) kullanılabilir.

```
int pthread_join(pthread_t tid, void ** ptr);
```

Verilen parametreli lif bitene kadar bekler (join) ve bu lif tarafından döndürülen değeri ikinci parametresi olan ptr ile alır. Buradaki değeri alma işlemi atıf ile çağırma (call by reference) (http://www.bilgisayarkavramlari.com/2009/01/12/atif-ile-cagirma-call-by-reference/).

```
int pthread_detach (pthread_t tid);
```

parametre olarak aldığı lifi hafızadan kaldırır. Bu fonksiyon ile lif sonlandırılmaz sadece hafızadaki kopyası silinir. Bu şekildeki lifler iptal edilemez (cancel) veya beklenemez (join).

```
int pthread_cancel(pthread_t tid);
```

parametre olarak aldığı lifi iptal eder (cancel). Bu işlem sayesinde o anda çalışan life artık ihtiyaç kalmadığı anlatılır ve o anda çalışan liflerin paylaşılmış durumlarını ayarlamaları sağlanır.

```
pthread_t pthread_self();
```

Bu fonksiyon çağrıldığında, değer olarak o andaki çalışan lifin bilgisi döndürülür.

```
int sched_yield();
```

Bu fonksiyon, zamanlayıcıya (scheduler) (http://www.bilgisayarkavramlari.com/2008/11/19/islemci-zamanlama-cpu-scheduling/) içinden çağrıldığı lifin çalışma durumundan alınarak bekleme durumuna geçirilmesini söyler. Bu sayede o anda beklemekte olan başka bir lif çalışabilecektir.

Örnek 3:

```
void *thread_routine(void* arg){
   printf("Lif olustu n");
   sleep( 30 );
   printf("Uykudan sonra n");
}
void main(){
   pthread_tthread_id; void *thread_result=0;
```

```
    pthread_create( & thread_id, NULL, thread_routine, NULL );

    sleep(3);

   printf("Ana lifn");

   pthread_cancel( thread_id);

   printf("Sonn");

}
```

Yukarıdaki yeni kodda, ana lif içerisinde bulunan 3 mili saniyelik uyuma fonksiyonuna karşılık, üretilen lifin çalıştırdığı fonksiyon içerisinde 30 mili saniyelik uyuma fonksiyonu bulunmaktadır. Ayrıca ana lifte bir pthread_cancel fonksiyonu çağrılmıştır. Bunun anlamı, ana lifin, 3 saniyelik uyuma işleminden sonra ekrana "Ana lif" mesajını basması ve ardından da oluşturulan yeni lifi iptal etmesidir (cancel). Bu esnada üretilen lif, "lif olustu" mesajını basacak ve 30 mili saniye uyuyacaktır. Tahminen ana lif, oluşturulan liften önce uyanacak ve oluşturulan lif henüz ekrana "uykudan sonra" yazamadan araya girecek ve bu lifi sonlandıracaktır. Neticede ekrana bir son yazısı yazılacak ama " uykudan sonra" yazısı yazılamayacaktır.

Örnek 4:

Bu örnekte klasik bir problem olan üretici / tüketici (producer /consumer) probleminin (http://www.bilgisayarkavramlari.com/2012/03/05/producer-consumer-problem-uretici-tuketici-problemi/) eş farklılık (mutual exclusion) (http://www.bilgisayarkavramlari.com/2011/01/05/birbirini-dislama-mutually-exclusive/) kısmını çözeceğiz. Amacımız, birden fazla üretici veya tüketici olması durumunda bu üretici veya tüketicilerden sadece birisinin çalışmasını sağlamaktır.

Üretici / tüketici modelinde iki temel problem olduğunu hatırlayınız (mutex ve progress (ilerleme)), bu kodda, bu problemlerden sadece bir tanesi çözülecektir.

```
pthread_mutex_tmutex=PTHREAD_MUTEX_INITILIZER;

intshared_data=1;

void *consumer(void* arg) {

    for(intI =0; I < 30 ; I ++ ){

        pthread_mutex_lock( &mutex);

        shared_data–; /* Critical Section. */

        pthread_mutex_unlock( &mutex);

    }

    printf(“Returning from Comsumer=%dn”, shared_data);

}

void main() {

    pthread_tthread_id;

    pthread_create( & thread_id, NULL, consumer, NULL );

    for(intI =0; I < 30 ; I ++ ){

        pthread_mutex_lock( &mutex);

        shared_data++; /* Producer Critical Section. */

        pthread_mutex_unlock( &mutex);

    } /*pthread_exit(0); /* Return from main thread. */
```

```
  printf("End of main =%dn", shared_data);

}
```

Yukarıdaki kodda, pthread kütüphanesinde bulunan mutex fonksiyonları kullanılmıştır. Bu fonksiyonlar, kritik alana (critical section) erişimi kontrol altına almışlar ve shared_data isimli değişkenin arttırım ve azaltım işlemlerini bölünemez (atomic) (http://www.bilgisayarkavramlari.com/2009/03/30/atomluluk-atomicity/) hale getirmişlerdir. Bu sayede 30 üretim ve 30 tüketim işleminin tamamı bölünmeden (atomic) (http://www.bilgisayarkavramlari.com/2009/03/30/atomluluk-atomicity/) gerçekleşmiş ve değişken değerleri hatasız olarak değiştirilmiştir.

Kodda görüldüğü üzere mutex kullanımı pthread kütüphanesi içerisinde 3 adımdan oluşur:

- başlangıç (init)
- kilit (lock)
- açma (unlock)

Öncelikle pthread_mutex_t yapısından (struct) (http://www.bilgisayarkavramlari.com/2007/11/08/olusum-composition-ve-struct-yapilar/) bir değişken tanımlanır. Yukarıdaki örnekte bu değişkene mutex ismi verilmiştir. Ardından istenildiği kadar bu değişken lock veya unlock fonksiyonları içerisinde çağırılabilir. Kilit fonksiyonunun (lock) çağrılması durumunda artık açma (unlock) fonksiyonu çağırılana kadar başka bir lifin (thread) çalışması engellenir. Bu sayede anlık olarak ilgili satırlar, tek bir lif (thread) (http://www.bilgisayarkavramlari.com/2010/03/22/thread-iplik/) tarafından çalıştırılmış olur.

Mutex için pthread kütüphanesinde kullanılan fonksiyonlara bakacak olursak aşağıdaki gibi bir liste çıkarılabilir:

```
pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attr);
```

Bu fonksiyon, yeni bir mutex oluşturulurken kullanılır ve verilen değişken ile ifade edilen ve ikinci parametrede verilen özellikleri taşıyan yeni bir lif mutex oluştururu. Bu değişken daha sonraki kilitleme ve açma fonksiyonlarına da parametre olacaktır.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Verilen parametredeki mutex kilidini siler ve yok eder.

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

Verilen parametredeki mutex değişkenine dayanarak bir kilitleme işlemi başlatır. Bu kilitleme işlemi sırasında, aynı değişkeni kullanan tek bir lif (thread) (http://www.bilgisayarkavramlari.com/2010/03/22/thread-iplik/) çalışabilir. Aynı değişkenin kilitli olması şartı bulunan diğer bütün lifler (thread) bekletilir. Ancak aynı anda birden fazla değişken ismi kullanılabilir.

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

Bu fonksiyonda bulunan parametredeki mutex değişkeni, daha önceden pthread_mutex_lock ile kilitlenen kilidi açılır. Böylelikle o anda bu kilidi bekleyen liflerden sadece bir tanesi daha kilidi geçerek kritik alana (critical section) girebilir.

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

Yukarıdaki fonksiyon, kilit koymadan kilit konulup konulamayacağını sorgular. Örneğin kilitlenme ihtimali olmayan bir mutex için kilit konulur ve kilit açılana kadar (unlock) lif işlemeye devam eder. Öte yandan zaten kilitli bir lif için kilitleme işlemi yapılmaksızın meşgul mesajı alınır (EBUSY).

Bu fonksiyon aslında oldukça kullanışlıdır. Şöyle bir durum düşünün ki kritik bir iş yapmamız gerekiyor ancak o anda krtik işlem için beklediğimiz mutex kilitlenmiş. Bu durumda kritik işimizi erteleyerek lifimizde bulunan diğer işleri yapıp sonra geri dönerek kilidin açılıp açılmadığını kontrol edebiliriz. Şayet kilit açıksa kritik alana girer şayet açık değilse yine başka işleri yapmaya devam edebiliriz.

Örnek 5:

```
pthread_mutex_tread_mutex=PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_twrite_mutex=PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_tqempty_cond_mutex=PTHREAD_MUTEX_INIT
IALIZER;

pthread_cond_tq_notempty_cond=PTHREAD_COND_INITIALIZ
ER;

pthread_mutex_tqfull_cond_mutex=
PTHREAD_MUTEX_INITIALIZER;

pthread_cond_tq_notfull_cond=
PTHREAD_COND_INITIALIZER;

void *consumer(void* arg) {

  int i;
```

```
  n_consumer++;
  for (i=0; i< ITERATIONS; i++) {
    pthread_mutex_lock( &read_mutex);
    while ( queue_is_empty() ){
      pthread_cond_wait(&q_notempty_cond,
&qempty_cond_mutex);
    } /*read from queue[ in ] */
    in = (in +1) % QUEUE_SIZE;
    pthread_mutex_unlock( &read_mutex);
    pthread_cond_signal(&q_notfull_cond);
  }
  printf("Returning from Comsumern"); n_consumer–;
}
void *producer(void* arg) {
  int i ;
  for (i=0; n_consumer; i++) {
    pthread_mutex_lock( &write_mutex);
    while ( queue_is_full() ){
      pthread_cond_wait(&q_notfull_cond,
&qfull_cond_mutex);
    } /* write to queue[out] */
```

```
      out = (out +1) % QUEUE_SIZE;
      pthread_mutex_unlock( &write_mutex);
      pthread_cond_signal(&q_notempty_cond);
    }
    printf("Returning from Producern");
}
int queue_is_empty(){
    if ( in == out ) return 1;
    else return 0 ;
}
int queue_is_full(){
    if ( in == (out+1 %QUEUE_SIZE) ) return 1;
    else return 0 ;
}
void main() {
    pthread_tthread_id;
    pthread_create(&thread_id,NULL, consumer, NULL);
    pthread_create(&thread_id,NULL, consumer, NULL);
    sleep(5);
    pthread_create(&thread_id,NULL, producer, NULL);
```

```
    pthread_create(&thread_id,NULL, producer, NULL);

    pthread_exit(0);

}
```

Yukarıdaki yeni kodda, üretici tüketici problemi (producer consumer problem) (http://www.bilgisayarkavramlari.com/2012/03/05/producer-consumer-problem-uretici-tuketici-problemi/) iki açıdan da çözülmüştür. Yukarıdaki kodda bulunan mutex fonksiyonları ile aynı anda en fazla tek bir lif (thread) (http://www.bilgisayarkavramlari.com/2010/03/22/thread-iplik/) çalışması garantilenirken, koyu renkte gösterilen ve yeni gelen koşullu değişken (conditional variable) uygulaması ile ilerleme (progress) problemi de çözülmüştür. Buna göre problemde şayet üretim için ayrılan alan kalmadıysa üreticilerin durup tüketicilerden en az birisinin bir tüketim yapmasını ve yeni alan açılmasını beklemesi gerekir. Benzer şekilde şayet hiç ürün yoksa, bu durumda da tüketicilerin durup en az bir üreticinin bir ürün üreterek hatta koymasını beklemesi gerekir. İşte bu problemin çözümü için yukarıda görülen boşluk ve doluluk kontrollerini yapan queue_is_empty ve queue_is_full fonksiyonları çağrılmıştır. Ayırca mutex problemi oluşturmak için birden fazla üretici ve tüketici lifi (thread) main fonksiyonunda üretilmiştir.

Son olarak mutex kontrollerinden bağımsız olarak üretim hattının dolu olması halinde üretici lifleri durduran bir koşullu değişken (conditional variable) ve hattın boş olması durumunda tüketici lifleri durduran bir koşullu değişken (conditional variable) kodlamaya eklenmiştir.

Kodda dikkat edileceği üzere, in ve out değişkenleri hem mutex hem de cond kormuması altındadır.

Bu kodda kullanılan cond fonksiyonları ise aşağıda açıklanmıştır:

```
int pthread_cond_init(pthread_cond_t *cond, const
pthread_contattr_t *attr);
```

Yukarıdaki fonksiyon sayesinde yeni bir koşullu değişken (conditional variable) tanımı yapılır ve bundan sonraki fonksiyonlarda kullanılabilir.

```
int pthread_cond_destroy(pthread_cont_t *cond);
```

Yukarıdaki fonksiyon sayesinde, daha önceden tanımlanmış olan koşullu değişken (conditional variable) sistemden kaldırılarak yok edilir.

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

Yukarıdaki fonksiyon, basitçe lifin beklemesini sağlayan ve o andaki çalışmayı durdurarak, signal fonksiyonu çağrılana kadar bekleten fonksiyondur.

```
int pthread_cont_signal(pthread_cond_t *cond);
```

Bu fonksiyonla, o anda beklemekte olan liflere sinyal yollanır ve uyanmaları sağlanır. Burada önemli bir not, aynı anda çeşitli zamanlama algoritmalarına göre (örneğin FIFO veya RR) beklemekte olan lifler (thread) bulunuyorsa, bu liflerden en yüksek öneme sahip (priority) lif uyandırılırken, şayet bütün bekleyen lifler aynı öneme sahipse bu durumda rast gelen bir lif uyandırılır.

```
int pthread_cond_timedwait(pthread_cond_t *cond,
pthread_mutex_t *mutex, const struct timespec
*abstime);
```

Klasik wait fonksiyonundan farklı olarak bir zaman bilgisini parametre alan fonksiyondur. Bu fonksiyonun uyanması için bir sinyal gelmesi veya belirtilen sürenin dolması yeterlidir. Genelde gerçek zamanlı sistemlerde (real time systems) vaz geçilmez fonksiyonlardan birisidir çünkü işlerin ilerlemesi için zaman bilgisi kritik önem taşır.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Bu fonksiyonla o anda parametre olarak alınan bütün beklemeler uyandırılır.