30

Koşul Değişkenleri

Şimdiye kadar bir kilit kavramını geliştirdik ve doğru donanım ve işletim sistemi desteği kombinasyonuyla nasıl düzgün bir şekilde oluşturulabileceğini gördük. Ne yazık ki, eşzamanlı programlar oluşturmak için gereken tek ilkel öğe kilitler değildir.

Özellikle, bir iş parçacığının yürütmeye devam etmeden önce bir **koşulun (condition)** doğru olup olmadığını kontrol etmek istediği birçok durum vardır. Örneğin, bir ana ileti dizisi devam etmeden önce bir alt ileti dizisinin tamamlanıp tamamlanmadığını kontrol etmek isteyebilir (buna genellikle `join()` adı verilir); böyle bir bekleme nasıl uygulanmalıdır? Şekil 30.1'e bakalım.

```
1  void *child(void *arg) {
2      printf("child\n");
3      // XXX işimizin bittiğini nasıl belirtiriz?
4      return NULL;
5  }
6
7  int main(int argc, char *argv[]) {
8      printf("parent: begin\n");
9      pthread_t c;
10     Pthread_create(&c, NULL, child, NULL); // child yarat
11     // XXX child için nasıl beklenir?
12     printf("parent: end\n");
13     return 0;
14 }
```

Şekil 30.1: **Child'ını Bekleyen Bir Parent (A Parent Waiting For Its Child)**

Burada görmek istediğimiz aşağıdaki çıktıdır:

```
parent: begin
child
parent: end
```

Şekil 30.2'de gördüğünüz gibi, paylaşılan bir değişken kullanmayı deneyebiliriz. Bu çözüm genellikle işe yarayacaktır,

```
1  volatile int done = 0;
2
3  void *child(void *arg) {
4      printf("child\n");
5      done = 1;
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     printf("parent: begin\n");
11     pthread_t c;
12     Pthread_create(&c, NULL, child, NULL); // child yarat
13     while (done == 0)
14         ; // döndür
15     printf("parent: end\n");
16     return 0;
17 }
```

Şekil 30.2: **Child'ı Bekleyen Parent: Döndürme Temelli Yaklaşım (Parent Waiting For Child: Spin based Approach)**

ancak parent dönüp CPU zamanını boşa harcadığı için son derece verimsizdir. Bunun yerine burada istediğimiz şey, beklediğimiz koşul (örneğin, child'ın yürütmesi bitene kadar) gerçekleşene kadar parent'ı uyutmanın bir yolunu bulmaktır.

ÖNEMLİ NOKTA: BİR ŞART İÇİN NASIL BEKLENİR

Çok iş parçacıklı programlarda, bir iş parçacığının ilerlemeden önce bazı koşulların gerçekleşmesini beklemesi genellikle yararlıdır. Koşul doğru olana kadar döndürme şeklindeki basit yaklaşım, büyük ölçüde verimsizdir ve CPU döngülerini boşa harcar ve bazı durumlarda yanlış olabilir. Böylece, bir iş parçacığı bir koşul için nasıl beklemelidir?

30.1 Tanım ve Rutinler

Bir koşulun gerçekleşmesini beklemek için bir iş parçacığı, **koşul değişkeni (condition veriable)** olarak bilinen şeyi kullanabilir. Bir **koşul değişkeni (condition veriable)**, bazı yürütme durumları (yani, bazı **koşullar (condition)**) istenildiği gibi olmadığında (koşul üzerinde **bekleyerek (waiting)**) iş parçacıklarının kendilerini koyabilecekleri açık bir sıradır; başka bir iş parçacığı, söz konusu durumu değiştirdiğinde, bekleyen iş parçalarından birini (veya daha fazlasını) uyandırabilir ve böylece (koşul üzerinde **sinyal (signaling)** vererek) devam etmelerine izin verebilir. Fikir, Dijkstra'nın "özel semaforlar" [D68] kullanımına kadar gider; benzer bir fikir daha sonra Hoare tarafından monitörler üzerine yaptığı çalışmada [H74] "durum değişkeni" olarak adlandırıldı.

Böyle bir koşul değişkenini bildirmek için, basitçe şöyle bir şey yazılır: c'yi bir koşul değişkeni olarak bildiren `pthread_cond_t c;` (not: uygun başlatma da gereklidir). Bir koşul değişkeninin kendisiyle ilişkilendirilmiş iki işlemi vardır: `wait()` ve `signal()`. `wait()` çağrısı, bir iş parçacığı uyku moduna geçmek istediğinde yürütülür; `signal()` çağrısı,

```
1  int done  = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

Şekil 30.3: **Child'ı Bekleyen Parent: Bir Koşul Değişkeni Kullanın (Parent Waiting For Child: A tion Variable)**

bir iş parçacığı programda bir şeyi değiştirdiğinde ve bu durumda bekleyen uykudaki bir iş parçacığını uyandırmak istediğinde yürütülür. Özellikle, POSIX çağrıları şöyle görünür:

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

Basitlik için bunlara genellikle `wait()` ve `signal()` olarak atıfta bulunacağız. `wait()` çağrısıyla ilgili fark edebileceğiniz bir şey de parametre olarak bir muteks almasıdır; `wait()` çağrıldığında bu muteksin kilitli olduğunu varsayar. `wait()`'in sorumluluğu kilidi serbest bırakmak ve çağıran thread'i (atomik olarak) uyku moduna geçirmektir; iş parçacığı uyandığında (başka bir iş parçacığı bunu işaret ettikten sonra), arayana geri dönmeden önce kilidi yeniden alması gerekir. Bu karmaşıklık, bir iş parçacığı kendini uyku moduna almaya çalışırken belirli yarış koşullarının

oluşmasını önleme arzusundan kaynaklanır. Bunu daha iyi anlamak için birleştirme sorununun çözümüne (Şekil 30.3) bir göz atalım.

Dikkate alınması gereken iki durum var. İlkinde, parent alt diziyi yaratır ama kendi kendine çalışmaya devam eder (yalnızca tek bir işlemcimiz olduğunu varsayalım) ve bu nedenle hemen child iş parçacığın tamamlanmasını beklemek için `thr_join()` işlevini çağırır. Bu durumda, kilidi alacak, child'ın yapılıp yapılmadığını kontrol edecek (değil) ve `wait()`'i çağırarak (dolayısıyla kilidi serbest bırakarak) kendini uykuya sokacaktır. Child en sonunda çalışacak, "child" mesajını yazdıracak ve üst diziyi uyandırmak için `thr_exit()` öğesini çağıracaktır; bu kod sadece kilidi alır, durum değişkenini done olarak ayarlar ve parent'e sinyal vererek onu uyandırır. Son olarak, parent çalışacak (tutulan kilit ile `wait()`'den dönerek), kilidi açacak ve "parent: end" final mesajını yazdıracaktır.

İkinci durumda, child oluşturulduktan hemen sonra çalışır, `done` 1'e ayarlanır, uyuyan bir iş parçacığını uyandırmak için sinyal çağırır (ancak hiçbiri yoktur, bu yüzden sadece geri döner) ve tamamlanır. Parent daha sonra çalışır, `thr_join()` aracılığıyla çağırır, `done`'nın 1 olduğunu görür ve bu nedenle beklemez ve geri döner.

Son bir not: Parent'ın , koşulu bekleyip beklememeye karar verirken yalnızca bir if ifadesi yerine bir `while` döngüsü kullandığını gözlemleyebilirsiniz. Programın mantığına göre bu kesinlikle gerekli görünmese de aşağıda göreceğimiz gibi her zaman iyi bir fikirdir.

`thr_exit()` ve `thr_join()` kodunun her bir parçasının önemini anladığınızdan emin olmak için, birkaç alternatif uygulama deneyelim. Öncelikle, durum değişkeninin yapılmasına ihtiyacımız olup olmadığını merak ediyor olabilirsiniz. Ya kod aşağıdaki örneğe benziyorsa? (Şekil 30.4)

Ne yazık ki bu yaklaşım bozuldu. Child'ın hemen çalıştığı ve `thr_exit()` çağırdığı bir durumu hayal edin; bu durumda, child sinyal verir, ancak koşulda uyuyan iş parçacığı yoktur. Parent çalıştığında, sadece beklemeyi çağıracak ve takılıp kalacak; hiçbir iş parçacığı onu uyandıramaz. Bu örnekten, done durum değişkeninin önemini takdir etmelisiniz; iş parçacıklarının bilmek istediği değeri kaydeder. Uyumak, uyanmak ve kilitlenmek onun etrafında inşa edilmiştir.

```
1  void thr_exit() {
2      Pthread_mutex_lock(&m);
3      Pthread_cond_signal(&c);
4      Pthread_mutex_unlock(&m);
5  }
6
7  void thr_join() {
8      Pthread_mutex_lock(&m);
9      Pthread_cond_wait(&c, &m);
10     Pthread_mutex_unlock(&m);
11 }
```

Şekil 30.4: **Parent Beklemede: Durum Değişkeni Yok (Parent Waiting: No State Variable)**

```
1 void thr_exit() {
2     done = 1;
3     Pthread_cond_signal(&c);
4 }
5
6 void thr_join() {
7     if (done == 0)
8         Pthread_cond_wait(&c);
9 }
```

Şekil 30.5: **Parent Beklemede: Kilit Yok (Parent Waiting: No Lock)**

Burada (Şekil 30.5) başka bir kötü uygulama var. Bu örnekte, sinyal vermek ve beklemek için kilit tutmaya gerek olmadığını hayal ediyoruz. Burada ne gibi bir sorun olabilir? Bir düşün[1]!

Buradaki sorun ince bir yarış koşulu. Spesifik olarak, parent, `thr_join()` öğesini çağırır ve ardından done değerini kontrol ederse, bunun 0 olduğunu görecek ve böylece uykuya dalmaya çalışacaktır. Ancak, uyumak için bekle çağrısından hemen önce, parent'ın sözü kesilir ve child çalışır. Child, `done` durum değişkenini 1 olarak değiştirir ve sinyal verir, ancak hiçbir iş parçacığı beklemez ve dolayısıyla hiçbir iş parçacığı uyandırılmaz. Parent tekrar çalıştığında sonsuza kadar uyur ki bu üzücü.

Umarım, bu basit birleştirme örneğinden, koşul değişkenlerini düzgün kullanmanın bazı temel gerekliliklerini görebilirsiniz. Anladığınızdan emin olmak için şimdi daha karmaşık bir örnek üzerinden geçiyoruz: **üretici/tüketici (producer/consumer)** veya **sınırlı arabellek (bounded-**

İPUCU: SİNYAL VERİRKEN DAİMA KİLİDİ TUTUN

Her durumda kesinlikle gerekli olmasa da koşul değişkenlerini kullanıp sinyal gönderirken kilidi tutmak muhtemelen basittir ve en iyisidir. Yukarıdaki örnek, doğruluk için kilidi tutmanız gereken bir durumu göstermektedir; ancak, yapmamanın muhtemelen uygun olduğu, ancak muhtemelen kaçınmanız gereken başka bazı durumlar da vardır. Bu nedenle, basit olması için, **sinyal çağırırken kilidi tutun (hold the lock when calling signal).**

Bu ipucunun tersi, yani, bekle çağrılırken kilidi tut, sadece bir ipucu değil, daha çok beklemenin semantiği tarafından zorunlu kılınmıştır, çünkü bekle her zaman (a) siz çağırdığınızda kilidin tutulduğunu varsayar, (b) serbest bırakır arayanı uyuturken söz konusu kilit ve (c) geri dönmeden hemen önce kilidi yeniden ele alır. Bu nedenle, bu ipucunun genellemesi doğrudur: **sinyal çağırırken kilidi tutun veya bekleyin (hold the lock when calling signal or wait)** ve her zaman iyi durumda olacaksınız.

[1] Bu örneğin "gerçek" kod olmadığına dikkat edin, çünkü `pthread_cond_wait()` çağrısı bir koşul değişkeninin yanı sıra her zaman bir muteks gerektirir; burada, olumsuz örnek adına arayüzün böyle yapmadığını varsayıyoruz.

```
1  int buffer;
2  int count = 0; // başlangıç, boş
3
4  void put(int value) {
5      assert(count == 0);
6      count = 1;
7      buffer = value;
8  }
9
10 int get() {
11     assert(count == 1);
12     count = 0;
13     return buffer;
14 }
```

Şekil 30.6: **Rutinleri Koy ve Al (v1) (The Put And Get Routines (v1))**

30.2 Üretici/Tüketici (Sınırlı Arabellek) Problemi

Bu bölümde karşılaşacağımız bir sonraki senkronizasyon problemi, ilk olarak Dijkstra [D72] tarafından ortaya atılan, **üretici/a (producer/consumer)** problemi veya bazen **sınırlı arabellek (bounded buffer)** problemi olarak bilinir. Aslında, Dijkstra ve iş arkadaşlarının genelleştirilmiş semaforu (kilit veya koşul değişkeni olarak kullanılabilen) [D01] icat etmelerine yol açan tam da bu producer /consumer sorunuydu; semaforlar hakkında daha sonra daha fazla şey öğreneceğiz.

Bir veya daha fazla producer iş parçacığı ve bir veya daha fazla consumer iş parçacığı düşünün. Producer'lar veri öğeleri oluşturur ve bunları bir buffer'a yerleştirir; consumer'lar söz konusu öğeleri buffer'dan alır ve bir şekilde tüketir

Bu düzenleme birçok gerçek sistemde gerçekleşir. Örneğin, çok iş parçacıklı bir web sunucusunda, bir producer HTTP isteklerini bir iş kuyruğuna (yani, buffer'a) koyar; consumer iş parçacıkları, istekleri bu kuyruktan alır ve işler.

Bir programın çıktısını diğerine aktardığınızda da sınırlı bir buffer kullanılır, örneğin, `grep foo file.txt | wc-1`. Bu örnek, aynı anda iki işlemi çalıştırır; `grep, file.txt` dosyasından, içinde `foo` dizesi bulunan satırları, standart çıktı olduğunu düşündüğü şeye yazar; UNIX kabuğu, çıktıyı UNIX borusu olarak adlandırılan şeye yönlendirir (**boru (pipe)** sistem çağrısı tarafından oluşturulur). Bu borunun diğer ucu, giriş akışındaki satır sayısını basitçe sayan ve sonucu yazdıran wc sürecinin standart girişine bağlıdır. Böylece `grep` işlemi producer; `wc` işlemi consumer'dir, aralarında çekirdekle sınırlı bir buffer bulunur; bu örnekte siz sadece mutlu kullanıcısınız.

Sınırlı buffer paylaşılan bir kaynak olduğundan, bir yarış durumu ortaya çıkmaması için[2] elbette ona senkronize erişim gerektirmeliyiz. Bu sorunu daha iyi anlamaya başlamak için bazı gerçek kodları inceleyelim.

İhtiyacımız olan ilk şey, bir producer'ın içine veri koyduğu ve bir consumer'ın içinden veri aldığı ortak bir buffer'dır. Basitlik için tek bir tamsayı kullanalım

[2] Burası size ciddi bir Eski İngilizce ve dilek kipi eklediğimiz yer.

```
1  void *producer(void *arg) {
2      int i;
3      int loops = (int) arg;
4      for (i = 0; i < loops; i++) {
5          put(i);
6      }
7  }
8
9  void *consumer(void *arg) {
10     while (1) {
11         int tmp = get();
12         printf("%d\n", tmp);
13     }
14 }
```

Şekil 30.7: **Üretici/Tüketici İş Parçacığı (v1) (Producer/Consumer Threads (v1))**

(bu yuvaya kesinlikle bir veri yapısına bir işaretçi yerleştirdiğinizi hayal edebilirsiniz) ve paylaşılan buffer'a bir değer koymak ve buffer'dan bir değer almak için iki iç rutini kullanalım. Ayrıntılar için bkz. Şekil 30.6 (sayfa 6). Oldukça basit, değil mi? `put()` rutini, buffer'ın boş olduğunu varsayar (ve bunu bir iddiayla kontrol eder) ve ardından paylaşılan buffer'a basitçe bir değer koyar ve `count`'ı 1 olarak ayarlayarak onu dolu olarak işaretler. `get()` rutini bunun tersini yapar, buffer'ı boşaltmak (yani, `count`'ı 0'a ayarlamak) ve değeri döndürmek. Bu paylaşılan buffer'ın yalnızca tek bir girişi olduğundan endişelenmeyin; daha sonra, göründüğünden daha eğlenceli olacak, birden fazla girişi tutabilen bir kuyruğa genelleştireceğiz.

Şimdi buffer'a veri koymak veya ondan veri almak için erişmenin ne zaman uygun olduğunu bilen bazı rutinler yazmamız gerekiyor. Bunun için koşullar açık olmalıdır: yalnızca `count` 0 olduğunda (yani, buffer boş olduğunda) buffer'a veri koyun ve yalnızca `count` 1 olduğunda (yani, buffer dolduğunda) buffer'dan veri alın. Senkronizasyon kodunu, bir producer'ın verileri dolu bir buffer'a koyacağı veya bir consumer'ın boş bir buffer'dan veri alacağı şekilde yazarsak, bir şeyi yanlış yapmış oluruz (ve bu kodda bir iddia ortaya çıkacaktır).

Bu iş iki tür iş parçacığı ile yapılacak, bunlardan birine **üretici (producer)** iş parçacığı, diğer takımına da **tüketici (consumer)** iş parçacığı diyeceğiz. Şekil 30.7, paylaşılan buffer `loop`'lara birkaç kez bir tamsayı koyan bir producer'ın ve paylaşılan buffer'dan çektiği veri öğesini her yazdırdığında verileri bu paylaşılan buffer'dan (sonsuza kadar) alan bir consumer'ın kodunu gösterir.

Karmaşık Bir Çözüm

Şimdi tek bir producer'ımız ve tek bir consumer'ımız olduğunu hayal edin. Açıkçası, `put()` ve `get()` rutinleri içlerinde kritik bölümleri vardır, çünkü `put()` buffer'ı günceller ve `get()` buffer'ı okur. Ancak, kodu kilitlemek işe yaramaz; daha fazlasına ihtiyacımız var.

```
1  int loops; // bir yerde başlatmalı...
2  cond_t  cond;
3  mutex_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          Pthread_mutex_lock(&mutex);            // p1
9          if (count == 1)                        // p2
10             Pthread_cond_wait(&cond, &mutex); // p3
11         put(i);                                // p4
12         Pthread_cond_signal(&cond);            // p5
13         Pthread_mutex_unlock(&mutex);          // p6
14     }
15 }
16
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex);            // c1
21         if (count == 0)                        // c2
22             Pthread_cond_wait(&cond, &mutex); // c3
23         int tmp = get();                       // c4
24         Pthread_cond_signal(&cond);            // c5
25         Pthread_mutex_unlock(&mutex);          // c6
26         printf("%d\n", tmp);
27     }
28 }
```

Şekil 30.8: **Üretici/Tüketici: Tek CV ve If Beyannamesi (Producer/Consumer: Single CV And If Statement)**

Şaşırtıcı olmayan bir şekilde, daha fazlası bazı durum değişkenleridir. Bu (bozuk) ilk denemede (Şekil 30.8), tek bir `cond` koşul değişkenine ve ilişkili kilit `mutex`'sine sahibiz.

Producer'lar ve consumer'lar arasındaki sinyal mantığını inceleyelim. Bir producer buffer'a sahip olmak istediğinde, boş olmasını bekler (p1 - p3). Consumer tam olarak aynı mantığa sahiptir, ancak farklı bir durumu bekler: dolgunluk (c1 - c3).

Sadece tek bir producer ve tek bir consumer ile Şekil 30.8'deki kod çalışır. Bununla birlikte, bu iş parçacıklarından birden fazlasına sahip olursak (örneğin, iki consumer), çözümün iki kritik problemi vardır. Onlar nelerdir? *.. (düşünmek için burada duraklayın) ...*

Beklemeden önceki if ifadesiyle ilgili olan ilk sorunu anlayalım. İki consumer (T_{c1} ve T_{c2}) ve bir producer (T_p) olduğunu varsayalım. Önce bir consumer (T_{c1}) çalışır; kilidi alır (c1), herhangi bir buffer'ın tüketime hazır olup olmadığını kontrol eder (c2) ve hiçbirinin hazır olmadığını görünce bekler (c3) (kilidi serbest bırakır).

Ardından producer (T_p) çalışır. Kilidi alır (p1), tüm buffer'ın dolu olup

T_{c1}	Durum	T_{c2}	Durum	T_p	Durum	Count	Yorum
c1	Çalışır		Hazır		Hazır	0	
c2	Çalışır		Hazır		Hazır	0	
c3	Uyur		Hazır		Hazır	0	Alacak bir şey yok
	Uyur		Hazır	p1	Çalışır	0	
	Uyur		Hazır	p2	Çalışır	0	
	Uyur		Hazır	p4	Çalışır	1	Buffer şu an dolu
	Hazır		Hazır	p5	Çalışır	1	T_{c1} uyandı
	Hazır		Hazır	p6	Çalışır	1	
	Hazır		Hazır	p1	Çalışır	1	
	Hazır		Hazır	p2	Çalışır	1	
	Hazır		Hazır	p3	Uyur	1	Buffer dolu: uyku
	Hazır	c1	Çalışır		Uyur	1	T_{c2} gizlice girer ...
	Hazır	c2	Çalışır		Uyur	1	
	Hazır	c4	Çalışır		Uyur	0	... ve veri yakalar
	Hazır	c5	Çalışır		Hazır	0	T_p uyandı
	Hazır	c6	Çalışır		Hazır	0	
	Çalışır		Hazır		Hazır	0	Oh! Veri yok

Şekil 30.9: **İş parçacığı İzleme: Bozuk Çözüm (v1) (Thread Trace: Broken Solution (v1))**

olmadığını kontrol eder (p2) ve durumun böyle olmadığını anlayarak devam eder ve buffer'ı doldurur (p4). Producer daha sonra bir buffer'ın dolduğunun sinyalini verir (p5). Kritik olarak, bu, ilk consumer'ı (T_{c1}) bir durum değişkeninde uyumaktan hazır kuyruğuna taşır; T_{c1} artık çalışabilir (ancak henüz çalışmıyor). Producer daha sonra buffer'ın dolduğunu anlayana kadar devam eder ve bu noktada uyku moduna geçer (p6, p1–p3).

Sorunun ortaya çıktığı yer burasıdır: başka bir consumer (T_{c2}) içeri sızar ve buffer'da var olan bir değeri (c1, c2, c4, c5, c6, b+uffer dolu olduğu için c3'teki beklemeyi atlayarak) tüketir. Şimdi T_{c1}'in çalıştığını varsayalım; beklemeden dönmeden hemen önce kilidi yeniden alır ve ardından geri döner. Daha sonra `get()` (c4) öğesini çağırır, ancak tüketilecek buffer yoktur! Bir iddia tetiklenir ve kod istendiği gibi çalışmaz. Açıkçası, T_{c1}'in tüketmeye çalışmasını bir şekilde engellemeliydik çünkü T_{c2} gizlice girdi ve üretilmiş olan buffer'daki tek değeri tüketti. Şekil 30.9, her iş parçacığının gerçekleştirdiği eylemi ve zaman içinde programlayıcı durumunu (Hazır, Çalışıyor veya Uyuyor) gösterir.

Sorun basit bir nedenle ortaya çıkıyor: Producer T_{c1}'i uyandırdıktan sonra, ancak T_{c1} hiç çalışmadan önce, sınırlandırılmış buffer'ın durumu değişti (T_{c2} sayesinde). Bir iş parçacığının sinyalini vermek onları yalnızca uyandırır; bu nedenle, dünyanın durumunun değiştiğine dair bir ipucudur (bu durumda, buffer'a bir değer yerleştirilmiştir), ancak uyandırılan iş parçacığı çalıştığında, durumun hala istendiği gibi olacağına dair bir garanti yoktur. Bir sinyalin ne anlama geldiğine ilişkin bu yoruma, bu şekilde bir koşul değişkeni oluşturan ilk araştırmadan sonra genellikle **Mesa anlambilimi (Mesa semantics)** denir [LR80]; **Hoare anlambilimi**

```
1  int loops;
2  cond_t  cond;
3  mutex_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          Pthread_mutex_lock(&mutex);            // p1
9          while (count == 1)                     // p2
10             Pthread_cond_wait(&cond, &mutex);  // p3
11         put(i);                                // p4
12         Pthread_cond_signal(&cond);            // p5
13         Pthread_mutex_unlock(&mutex);          // p6
14     }
15 }
16
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex);            // c1
21         while (count == 0)                     // c2
22             Pthread_cond_wait(&cond, &mutex);  // c3
23         int tmp = get();                       // c4
24         Pthread_cond_signal(&cond);            // c5
25         Pthread_mutex_unlock(&mutex);          // c6
26         printf("%d\n", tmp);
27     }
28 }
```

Şekil 30.10: **Üretici/Tüketici: Tek CV ve While (Producer/Consumer: Single CV And While)**

(Hoare semantics) olarak anılan karşıtlığın oluşturulması daha zordur, ancak uyandırılan iş parçacığının uyandırıldıktan hemen sonra çalışacağına dair daha güçlü bir garanti sağlar [H74]. Hemen hemen şimdiye kadar yapılmış her sistem Mesa semantiğini kullanır.

Daha İyi Ama Hala Karmaşık: While , Not If

Neyse ki, bu düzeltme kolaydır (Şekil 30.10): `if`'i `while` ile değiştirin. Bunun neden işe yaradığını düşünün; şimdi consumer T_{c1} uyanır ve (kilit tutularak) paylaşılan değişkenin (c2) durumunu hemen yeniden kontrol eder. Bu noktada buffer boşsa, consumer basitçe uykuya geri döner (c3). Sonuç `if` de producer'da `while` olarak değiştirilir (p2).

Mesa semantiği sayesinde, koşul değişkenleri ile hatırlanması gereken basit bir kural, **her zaman while döngüleri kullanmaktır (always use while loops)**. Bazen durumu yeniden kontrol etmeniz gerekmez, ancak bunu yapmak her zaman güvenlidir; sadece yap ve mutlu ol.

Ancak, bu kodda hala bir hata var, yukarıda belirtilen iki sorundan ikincisi. Bunu görebiliyor musun? Sadece bir koşul değişkeni olduğu gerçeğiyle bir ilgisi var. Devamını okumadan önce sorunun ne olduğunu anlamaya çalışın. YAP! *(düşünmek için duraklayın veya gözlerinizi kapatın...)*

T_{c1}	Durum	T_{c2}	Durum	T_p	Durum	Count	Yorum
c1	Çalışır		Hazır		Hazır	0	
c2	Çalışır		Hazır		Hazır	0	
c3	Uyur		Hazır		Hazır	0	Alacak bir şey yok
	Uyur	c1	Çalışır		Hazır	0	
	Uyur	c2	Çalışır		Hazır	0	
	Uyur	c3	Uyur		Hazır	0	Buffer şu an dolu
	Uyur		Uyur	p1	Çalışır	0	
	Uyur		Uyur	p2	Çalışır	0	
	Uyur		Uyur	p4	Çalışır	1	Buffer şu an dolu
	Hazır		Uyur	p5	Çalışır	1	Tc1 uyandı
	Hazır		Uyur	p6	Çalışır	1	
	Hazır		Uyur	p1	Çalışır	1	
	Hazır		Uyur	p2	Çalışır	1	
	Hazır		Uyur	p3	Uyur	1	Uyumak zorunda (ful)
c2	Çalışır		Uyur		Uyur	1	Durumu tekrar kontrol et
c4	Çalışır		Uyur		Uyur	0	T_{c1} verileri alır
c5	Çalışır		Hazır		Uyur	0	Ayy! T_{c2}'yi uyandırdı
c6	Çalışır		Hazır		Uyur	0	
c1	Çalışır		Hazır		Uyur	0	
c2	Çalışır		Hazır		Uyur	0	
c3	Uyur		Hazır		Uyur	0	Alacak bir şey yok
	Uyur	c2	Çalışır		Uyur	0	
	Uyur	c3	Uyur		Uyur	0	Herkes uyuyor

Şekil 30.11: **İş parçacığı İzleme: Bozuk Çözüm (v2) (Thread Trace: Broken Solution (v2))**

Doğru anladığınızı veya belki de şu anda uyanık olduğunuzu ve kitabın bu bölümünü okuduğunuzu onaylayalım. Sorun, iki consumer önce çalıştığında (T_{c1} ve T_{c2}) ve her ikisi de uykuya daldığında (c3) ortaya çıkar. Ardından, producer çalışır, buffer'a bir değer koyar ve consumer'lardan birini uyandırır (T_{c1} diyelim). Producer daha sonra geri döner (yol boyunca kilidi serbest bırakır ve yeniden alır) ve buffer'a daha fazla veri koymaya çalışır; buffer dolu olduğundan, producer bunun yerine koşulu bekler (dolayısıyla uyur). Şimdi, bir consumer çalışmaya hazır (T_{c1}) ve iki iş parçacığı bir koşulda uyuyor (T_{c2} ve T_p). Bir soruna neden olmak üzereyiz: işler heyecan verici bir hal alıyor!

T_{c1} consumer'ı daha sonra `wait()` (c3) işlevinden dönerek uyanır, koşulu yeniden kontrol eder (c2) ve buffer'ın dolu olduğunu bulunca değeri (c4) tüketir. Bu consumer daha sonra, kritik olarak, *yalnızca* uykuda olan bir iş parçacığını uyandıran koşul (c5) hakkında sinyal verir. Ancak, hangi iş parçacığını uyandırmalı?

Consumer buffer'ı boşalttığı için producer'ı açıkça uyandırmalıdır. Ancak, consumer T_{c2}'yi uyandırırsa (bekleme kuyruğunun nasıl yönetildiğine bağlı olarak bu kesinlikle mümkündür), bir sorunumuz var demektir. Spesifik olarak, consumer T_{c2} uyanacak ve buffer'ı boş bulacak (c2) ve uykuya geri

```
1  cond_t  empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == 1)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Şekil 30.12: **Üretici/Tüketici: İki CV ve While (Producer/Consumer: Two CVs And While)**

dönecektir (c3).Buffer'a girecek değeri olan producer T_p uykuda bırakılır. Diğer consumer iş parçacığı T_{c1} de uyku moduna geri döner. Üç iş parçacığının tümü uykuda bırakıldı, açık bir hata; bu korkunç felaketin acımasız adım adım ilerlemesi için Şekil 30.11'e bakın.

Sinyal vermeye açıkça ihtiyaç vardır, ancak daha fazla yönlendirilmelidir. Bir consumer diğer consumer'ları değil, sadece producer'ları uyandırmalıdır ve bunun tersi de geçerlidir.

Tek Buffer'lı Producer/Consumer Çözümü

Buradaki çözüm bir kez daha küçük: sistemin durumu değiştiğinde hangi iş parçacığının uyanması gerektiğini düzgün bir şekilde işaret etmek için bir yerine *iki* koşul değişkeni kullanın. Şekil 30.12 sonuç kodunu göstermektedir.

Kodda, producer iş parçacıkları **boş (empty)** durumda bekler ve sinyaller dolar. Tersine, consumer iş parçacıkları **dolduğunda (fill)** bekler ve **boş (empty)** sinyali verir. Bunu yaparak, tasarım sayesinde yukarıdaki ikinci sorundan kaçınılır: Bir consumer asla bir consumer'ı yanlışlıkla uyandıramaz ve bir producer da bir producer'ı asla kazara uyandıramaz.

```
1  int buffer[MAX];
2  int fill_ptr = 0;
3  int use_ptr  = 0;
4  int count    = 0;
5
6  void put(int value) {
7      buffer[fill_ptr] = value;
8      fill_ptr = (fill_ptr + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use_ptr];
14     use_ptr = (use_ptr + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

Şekil 3.13: **Doğru Koy ve Al Rutinleri (The Correct Put And Get Routines)**

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);            // p1
8          while (count == MAX)                   // p2
9              Pthread_cond_wait(&empty, &mutex); // p3
10         put(i);                                // p4
11         Pthread_cond_signal(&fill);            // p5
12         Pthread_mutex_unlock(&mutex);          // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);            // c1
20         while (count == 0)                     // c2
21             Pthread_cond_wait(&fill, &mutex);  // c3
22         int tmp = get();                       // c4
23         Pthread_cond_signal(&empty);           // c5
24         Pthread_mutex_unlock(&mutex);          // c6
25         printf("%d\n", tmp);
26     }
27 }
```

Şekil 30.14: **Doğru Üretici/Tüketici Senkronizasyonu (The Correct Producer/Consumer Synchronization)**

İPUCU: KOŞULLAR İÇİN WHILE (İF DEĞİL) KULLANIN

Çok iş parçacıklı bir programda bir koşulu kontrol ederken, bir `while` döngüsü kullanmak her zaman doğrudur; Bir `if` ifadesinin kullanılması, sinyallemenin semantiğine bağlı olarak yalnızca olabilir. Bu nedenle, her zaman `while` kullanın ve kodunuz beklendiği gibi davranacaktır.

Koşullu denetimler etrafında while döngülerinin kullanılması**, sahte uyandırmaların (spurious wakeups)** meydana geldiği durumu da ele alır. Bazı iş parçacığı paketlerinde, uygulamanın ayrıntıları nedeniyle, yalnızca tek bir sinyal gerçekleşmesine rağmen iki iş parçacığının uyanması mümkündür [L11]. Sahte uyanmalar, bir iş parçacığının beklediği durumu yeniden kontrol etmek için başka bir nedendir.

Doğru Producer/Consumer Çözümü

Artık tamamen genel olmasa da çalışan bir producer/consumer çözümümüz var. Yaptığımız son değişiklik, daha fazla eşzamanlılık ve verimlilik sağlamak; özellikle, daha fazla buffer yuvası ekliyoruz, böylece uyumadan önce birden çok değer üretilebilir ve benzer şekilde uyumadan önce birden çok değer tüketilebilir. Yalnızca tek bir producer ve consumer ile bu yaklaşım, içerik geçişlerini azalttığı için daha etkilidir; birden fazla producer veya consumer (veya her ikisi) ile, eşzamanlı üretim veya tüketimin gerçekleşmesine bile izin verir, böylece eşzamanlılığı artırır. Neyse ki, mevcut çözümümüzden küçük bir değişiklik.

Bu doğru çözüm için ilk değişiklik, buffer yapısının kendisinde ve karşılık gelen `put()` ve `get()` içindedir (Şekil 30.13). Producer'ların ve consumer'ların uyuyup uyumamak için kontrol ettikleri koşulları da biraz değiştiriyoruz. Ayrıca doğru bekleme ve sinyal verme mantığını da gösteriyoruz (Şekil 30.14). Bir producer yalnızca tüm buffer'lar o anda doluysa uyur (p2); benzer şekilde, bir consumer yalnızca tüm buffer'lar şu anda boşsa uyur (c2). Böylece producer/consumer sorununu çözmüş oluyoruz; arkanıza yaslanıp soğuk bir şeyler içme zamanı.

30.3 Kapsam Koşulları

Şimdi koşul değişkenlerinin nasıl kullanılabileceğine dair bir örneğe daha bakacağız. Bu kod çalışması, yukarıda açıklanan **Mesa anlambilimini (Mesa semantics)** ilk kez uygulayan aynı grup olan Lampson ve Redell'in Pilot [LR80] hakkındaki makalesinden alınmıştır (kullandıkları dil Mesa'dır, dolayısıyla adı da buradan gelmektedir).

Karşılaştıkları sorun en iyi şekilde basit bir örnekle, bu durumda basit birçok iş parçacıklı bellek ayırma kitaplığında gösterilir. Şekil 30.15, sorunu gösteren bir kod parçacığını göstermektedir.

Kodda görebileceğiniz gibi, bir iş parçacığı bellek ayırma kodunu çağırdığında, daha fazla belleğin boş kalması için beklemesi gerekebilir. Tersine, bir iş parçacığı belleği boşalttığında, daha fazla belleğin boş olduğunu gösterir. Ancak, yukarıdaki kodumuzun bir sorunu var: hangi bekleyen iş parçacığı (birden fazla olabilir) uyandırılmalıdır?

```
1  // yığının kaç baytı boş?
2  int bytesLeft = MAX_HEAP_SIZE;
3
4  // kilit ve koşula da ihtiyaç var
5  cond_t  c;
6  mutex_t m;
7
8  void *
9  allocate(int size) {
10     Pthread_mutex_lock(&m);
11     while (bytesLeft < size)
12         Pthread_cond_wait(&c, &m);
13     void *ptr = ...; // yığından bellek al
14     bytesLeft -= size;
15     Pthread_mutex_unlock(&m);
16     return ptr;
17 }
18
19 void free(void *ptr, int size) {
20     Pthread_mutex_lock(&m);
21     bytesLeft += size;
22     Pthread_cond_signal(&c); // kime işaret?
23     Pthread_mutex_unlock(&m);
24 }
```

Şekil 30.15: **Kapsama Koşulları: Bir Örnek (Covering Conditions: An Example)**

Aşağıdaki senaryoyu düşünün. Boş sıfır bayt olduğunu varsayalım; iş parçacığı T_a `allocate(100)` çağırır, ardından `allocate(10)` çağırarak daha az bellek isteyen T_b iş parçacığı gelir. Hem T_a hem de T_b böylece durumu bekler ve uykuya dalar; bu isteklerden herhangi birini karşılamak için yeterli boş bayt yok.

Bu noktada, üçüncü bir iş parçacığının, T_c'nin `free(50)` çağırdığını varsayalım. Ne yazık ki, bekleyen bir iş parçacığını uyandırmak için signali çağırdığında, yalnızca 10 baytın serbest bırakılmasını bekleyen doğru bekleyen iş parçacığını, T_b uyandırmayabilir; Henüz yeterli bellek olmadığı için T_a beklemeye devam etmelidir. Böylece, diğer iş parçacıklarını uyandıran iş parçacığı hangi iş parçacığını (veya iş parçacıklarını) uyandıracağını bilmediğinden, şekildeki kod çalışmaz.

Lampson ve Redell tarafından önerilen çözüm basittir: yukarıdaki koddaki `pthread_cond_signal()` çağrısını, bekleyen tüm evreleri uyandıran bir `pthread_cond_broadcast()` çağrısıyla değiştirin. Bunu yaparak, uyandırılması gereken tüm iş parçacıklarının uyandığını garanti ediyoruz. Dezavantajı, elbette, olumsuz bir performans etkisi olabilir, çünkü (henüz) uyanmaması gereken diğer birçok bekleyen ileti dizisini gereksiz yere uyandırabiliriz. Bu ileti dizileri basitçe uyanacak, durumu yeniden kontrol edecek ve ardından hemen uyku moduna geri dönecektir.

Lampson ve Redell, bir iş parçacığının uyanması gereken tüm durumları kapsadığı için (konservatif olarak) böyle bir koşulu **kapsayan koşul (covering condition)** olarak adlandırır; Tartıştığımız gibi maliyet, çok fazla ileti dizisinin uyandırılmış olabileceğidir.

Dikkatli okuyucu, bu yaklaşımı daha önce kullanmış olabileceğimizi de fark etmiş olabilir (yalnızca tek koşul değişkenli producer/consumer problemine bakın). Ancak o durumda daha iyi bir çözüm vardı ve biz de onu kullandık. Genel olarak, programınızın yalnızca sinyallerinizi yayın olarak değiştirdiğinizde çalıştığını fark ederseniz (ancak buna gerek olmadığını düşünüyorsanız), muhtemelen bir hatanız vardır; düzeltin! Ancak yukarıdaki bellek ayırıcı gibi durumlarda, yayın mevcut en basit çözüm olabilir.

30.4 Özet

Kilitlerin ötesinde başka bir önemli senkronizasyon ilkelinin tanıtıldığını gördük: durum değişkenleri. CV'ler, bazı program durumları istenildiği gibi olmadığında iş parçacıklarının uyumasına izin vererek, ünlü (ve hala önemli olan) producer/consumer sorunu ve kapsama koşulları dahil olmak üzere bir dizi önemli senkronizasyon sorununu düzgün bir şekilde çözmemizi sağlar. Buraya "Ağabey'i severdi" [K49] gibi daha dramatik bir sonuç cümlesi gelirdi

References

[D68] "Cooperating sequential processes" by Edsger W. Dijkstra. 1968. Available online here: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF. *Another classic from Dijkstra; reading his early works on concurrency will teach you much of what you need to know.*

[D72] "Information Streams Sharing a Finite Buffer" by E.W. Dijkstra. Information Processing Letters 1: 179–180, 1972. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF *The famous paper that introduced the producer/consumer problem.*

[D01] "My recollections of operating system design" by E.W. Dijkstra. April, 2001. Available: http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF. *A fascinating read for those of you interested in how the pioneers of our field came up with some very basic and fundamental concepts, including ideas like "interrupts" and even "a stack"!*

[H74] "Monitors: An Operating System Structuring Concept" by C.A.R. Hoare. Communications of the ACM, 17:10, pages 549–557, October 1974. *Hoare did a fair amount of theoretical work in concurrency. However, he is still probably most known for his work on Quicksort, the coolest sorting algorithm in the world, at least according to these authors.*

[L11] "Pthread_cond_signal Man Page" by Mysterious author. March, 2011. Available online: http://linux.die.net/man/3/pthread_cond_signal. *The Linux man page shows a nice simple example of why a thread might get a spurious wakeup, due to race conditions within the signal/wakeup code.*

[LR80] "Experience with Processes and Monitors in Mesa" by B.W. Lampson, D.R. Redell. Communications of the ACM. 23:2, pages 105-117, February 1980. *A classic paper about how to actually implement signaling and condition variables in a real system, leading to the term "Mesa" semantics for what it means to be woken up; the older semantics, developed by Tony Hoare [H74], then became known as "Hoare" semantics, which is a bit unfortunate of a name.*

[O49] "1984" by George Orwell. Secker and Warburg, 1949. *A little heavy-handed, but of course a must read. That said, we kind of gave away the ending by quoting the last sentence. Sorry! And if the government is reading this, let us just say that we think that the government is "double plus good". Hear that, our pals at the NSA?*

Ödev (Kod)

Bu ev ödevi, bölümde tartışılan çeşitli producer/consumer kuyruğu biçimlerini uygulamak için kilitleri ve koşul değişkenlerini kullanan bazı gerçek kodları keşfetmenizi sağlar. Gerçek koda bakacak, onu çeşitli yapılandırmalarda çalıştıracak ve onu neyin işe yarayıp neyin yaramadığını ve diğer incelikleri öğrenmek için kullanacaksınız. Detaylar için "README" yi okuyun.

Sorular

1. İlk sorumuz `main-two-cvs-while.c` (çalışan çözüm) üzerine odaklanıyor. İlk olarak, kodu inceleyin. Programı çalıştırdığınızda ne olması gerektiğine dair bir anlayışa sahip olduğunuzu düşünüyor musunuz?

2.Bir producer ve bir consumer ile çalıştırın ve producer'ın birkaç değer üretmesini sağlayın. Bir buffer'la (boyut 1) başlayın ve ardından artırın. Kodun davranışı daha büyük buffer'larla nasıl değişir? (ya da öyle mi?). Consumer uyku dizesini varsayılandan `-C 0,0,0,0,0,0,1` olarak değiştirdiğinizde farklı buffer boyutları (ör. `-m 10`) ve farklı sayıda üretilen öğe (ör. `-l 100`) ile "`num_full`" un ne olacağını tahmin edersiniz?
2. Davranış, daha büyük arabellekle değişmez. NF her zaman 0 veya 1'dir. `-C 0,0,0,0,0,0,1` Kullanıldığında consumer her tüketimden sonra 1 saniye uyur. Producer bu süre içinde tüm buffer'ı doldurur.

3. Mümkünse, kodu farklı sistemlerde çalıştırın (ör. Mac ve Linux). Bu sistemlerde farklı davranışlar görüyor musunuz?

4.Bazı zamanlamalara bakalım. Bir producer, üç consumer, tek girişli paylaşılan buffer ve her consumer'ın c3 noktasında bir saniye duraklaması ile aşağıdaki yürütmenin ne kadar süreceğini düşünüyorsunuz? `./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,1,0,0,0:0,0,0,1,0,0,0:0,0,0,1,0,0,0 -l 10 -v t –`
4. İlk consumer beklemeden ve dolayısıyla uyumadan bir değer alır. Sonra ikinci consumer beklemeye geçiyor ama producer ürettikten sonra kilidi açtığında birinci consumer tekrar kilidi alıyor. Yani ikinci consumer uyandığında bir değer almıyor ve kilidi tutarak bir saniye uyuyor. Bu sayede ikinci ve üçüncü consumer toplamda 9 saniye uyur. Ve akışın sonunda, üç consumer'ın her biri 1 saniye uyur. Bu nedenle, program toplam 12 saniye ve gerçek yürütme için biraz zaman alır.

5. Şimdi paylaşılan buffer'ın boyutunu 3 (`-m 3`) olarak değiştirin. Bu, toplam süre içinde herhangi bir fark yaratacak mı?
5.Şimdi 11 saniye sürüyor. Sadece producer'lar olduğu için pek bir değişiklik olmuyor. Ancak şu anda yeterli buffer yuvası olduğu için, iş parçacıklarından birinin (bu EOS'ları ana üretmeden önce c2'ye gelmeyen) EOS'u beklemesine gerek yok.

6. Şimdi yine tek girişli bir buffer kullanarak uyku konumunu `c6` olarak değiştirin (bu, bir consumer'ın kuyruktan bir şey alıp sonra onunla bir şeyler yapmasını modeller). Bu durumda ne zaman öngörüyorsunuz?
```
./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,0,0,0,1
:0,0,0,0,0,1:0,0,0,0,0,1 -l 10 -v -t
```
6. 5 saniye. Producer'lar uyurken kilidi tutmadıklarından, her consumer bir partiden 1 saniye sonra uyurken üç partide 9 tüketim gerçekleşir. Bundan sonra 1 consumer'ın son değeri tüketmesi ve bir saniye uyuması gerekirken, diğer ikisi EOS'u tüketir ve bundan sonra 1 saniye uyur. Ardından son değeri tüketen consumer uyanır, EOS tüketir ve 1 saniye uyur.

7. Son olarak buffer boyutunu tekrar 3 (`-m 3`) olarak değiştirin. Şimdi ne zaman tahmin ediyorsunuz?
7. Hala 11 saniyedir.

8. Şimdi `main-one-cv-while.c'ye` bakalım. Bu kodla ilgili bir soruna neden olmak için, tek bir producer, bir consumer ve 1 boyutunda bir buffer varsayarak bir uyku dizesi yapılandırabilir misiniz?
8. Bu mümkün değil. Çünkü burada tek producer ve tek consumer var. Yani bir koşul değişkeni yeterlidir.

9. Şimdi consumer sayısını ikiye değiştirin. Kodda sorun yaratacak şekilde producer'lar ve consumer'lar için uyku dizileri oluşturabilir misiniz?
9. Belki zamanlama önceliği veya farklı CPU'ların yürütme hızları nedeniyle producer'ları ve consumer'ları aynı anda uyutamadık, ancak ana iş parçacığını ve bir consumer'ı aynı anda uyutmayı başardık.
```
./main-one-cv-while -l 3 -m 1 -p 1 -c 2 -v -t -P
0,0,0,0,0,0,1
```

10. Şimdi `main-two-cvs-if.c'yi` inceleyin. Bu kodda bir sorun oluşmasına neden olabilir misiniz? Yine sadece bir consumer'ın olduğu durumu ve ardından birden fazla consumer'ın olduğu durumu düşünün.
10. Tek consumer ile mümkün değil. Pek çok tüketicide sorun uyumadan oluyor.

11. Son olarak `main-two-cvs-while-extra-unlock.c'yi` inceleyin. Bir koyma veya alma işlemi yapmadan önce kilidi açtığınızda hangi sorun ortaya çıkar? Uyku dizeleri göz önüne alındığında, güvenilir bir şekilde böyle bir sorunun olmasına neden olabilir misiniz? Ne gibi kötü bir şey olabilir?
11. İlk consumer yalnızca bir değer üretir.