30

Condition Variables (Durum 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 kontrol etmek istediği birçok durum vardır, yürütmeye devam etmeden önce bir **koşulun (condition)** doğru olup olmadığı gibi. 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 how to indicate we are done?
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); // create child
11     // XXX how to wait for child?
12     printf("parent: end\n");
13     return 0;
14 }
```

Şekil 30.1: A Parent Waiting For Its Child (Çocuğunu Bekleyen Bir Ebeveyn)

Burada görmek istediğimiz çıktı aşağıdaki gibidir:

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

Şekil 30.2'de gördüğünüz gibi, paylaşılan bir değişken kullanmayı deneyebiliriz. Bu çözüm genellikle işe yarayacaktır, ancak 'parent' dönüp CPU zamanını boşa harcadığı için oldukça verimsizdir.

```c
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Şekil 30.2: Parent Waiting For Child: Spin-based Approach

Bunun yerine burada istediğimiz şey, parent beklediğimiz koşul (örneğin, çocuğun yürütmesi bitene kadar) gerçekleşene kadar uyutmanın bir yolunu bulmaktır.

ÖNEMLİ NOKTA: BİR ŞART İÇİN NASIL BEKLENİR
Çok iş parçacıklı programlarda, bir iş parçacığının beklemesi devam etmeden önce gerçekleşmesi gereken bazı koşullar için 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 variable)** olarak bilinen şeyi kullanabilir. Bir **koşul değişkeni (condition variable)**, 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 parçacıklarından birini (veya daha fazlasını) uyandırabilir ve böylece (koşul üzerinde **sinyal vererek (signaling)**) 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).

```
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: Parent Waiting For Child: Use A Condition Variable

Bir koşul değişkeninin kendisiyle ilişkilendirilmiş iki işlemi vardır: `wait()` ve `signal()`. `wait()` çağrısı, bir iş parçacığı uyku moduna geçmek istediğinde yürütülür; `signal()` çağrısı, 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, ebeveyn alt iş parçacığını oluşturur, ancak kendisini çalıştırmaya devam eder (yalnızca tek bir işlemcimiz olduğunu varsayalım) ve böylece alt iş parçacığının tamamlanmasını beklemek için hemen thr `join()` işlevini çağırır. Bu durumda, kilidi alacak, çocuğun yapılıp yapılmadığını kontrol edecek (değil) ve `wait()`'i çağırarak (dolayısıyla kilidi serbest bırakarak) kendini uykuya sokacaktır. Çocuk en sonunda çalışacak, "child" mesajını yazdıracak ve üst diziyi uyandırmak için `thr_exit()` öğesini çağıracaktır; bu kod sadece kilidi alır, yapılan durum değişkenini ayarlar ve ebeveyne sinyal vererek onu uyandırır. Son olarak, ebeveyn çalışacak (kilidi basılı tutarak `wait()` dönerek), kilidi açacak ve "parent: end" son mesajını yazdıracaktır.

İkinci durumda, çocuk yaratılıştan hemen sonra çalışır, kümeler 1'e yapılır, uyuyan bir iş parçacığını uyandırmak için sinyal çağırır (ancak hiçbiri yoktur, bu yüzden sadece geri döner) ve tamamlanır. Ebeveyn daha sonra çalışır,`thr_join()`'i çağırır, done'in 1 olduğunu görür ve bu nedenle beklemez ve geri döner.

Son bir not: Ebeveynin, 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. Çocuğun hemen koştuğu ve hemen `thr_exit()` çağırdığı bir durumu hayal edin; bu durumda, çocuk sinyal verir, ancak koşulda uyuyan iş parçacığı yoktur. Ebeveyn çalıştığında, sadece beklemeyi arayacak ve takılıp kalacak; hiçbir iş parçacığı onu uyandıramaz. Bu örnekten, yapılan 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 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 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!

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

Umarız, 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-buffer)** sorunu.

İPUCU: SİNYAL VERİRKEN DAİMA KİLİDİ TUTUN

Her durumda kesinlikle gerekli olmasa da, koşul değişkenlerini kullanırken sinyal gönderirken kilidi tutmak muhtemelen en basit 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 söz konusu kilit, arayanı uyuturken ve (c) geri dönmeden hemen önce kilidi yeniden alır. Bu nedenle, bu ipucunun genellemesi doğrudur:**sinyal çağırırken veya beklerken kilidi tutun (hold the lock when calling signal or wait)**.

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

```
1  int buffer;
2  int count = 0; // initially, empty
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: 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/tüketici (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 üretici/tüketici sorunuydu; semaforlar hakkında daha sonra daha fazla şey öğreneceğiz. Bir veya daha fazla üretici iş parçacığı ve bir veya daha fazla tüketici iş parçacığı düşünün. Üreticiler veri öğeleri oluşturur ve bunları bir ara belleğe yerleştirir; tüketiciler, söz konusu öğeleri tampondan alır ve bir şekilde tüketir. Bu düzenleme birçok gerçek sistemde gerçekleşir. Örneğin, çok iş parçacıklı bir web sunucusunda, bir üretici HTTP isteklerini bir iş kuyruğuna (yani, sınırlı arabelleğe) koyar; tüketici iş parçacıkları, istekleri bu kuyruktan alır ve işler. Sınırlı arabellek, bir programın çıktısını diğerine aktardığınızda da kullanılır, örneğin, `grep foo file.txt | wc -l`. 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 kanalı olarak adlandırılan şeye yönlendirir (**boru (pipe)** sistem çağrısı tarafından oluşturulur). Bu borunun diğer ucu, giriş akışındaki satır sayısını basitçe sayan ve sonucu yazdıran `wc` işleminin standart girişine bağlanır. Böylece `grep` işlemi üreticidir; `wc` işlemi tüketicidir; aralarında çekirdekle sınırlı bir arabellek bulunur; bu örnekte siz sadece mutlu kullanıcısınız.

Sınırlı arabellek paylaşılan bir kaynak olduğu için, bir yarış durumu ortaya çıkmasın (2) diye, elbette ona senkronize erişim gerektirmeliyiz. Bu sorunu daha iyi anlamaya başlamak için bazı gerçek kodları inceleyelim. İhtiyacımız olan ilk şey, bir üreticinin içine veri koyduğu ve bir tüketicinin veri aldığı paylaşılan bir arabellek.

```
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: Producer/Consumer Threads (v1)

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

Şimdi arabelleğe veri koymak veya ondan veri almak için erişmenin ne zaman uygun olduğunu bilen bazı rutinler yazmamız gerekiyor. Bunun için koşullar açık olmalıdır: yalnızca `count` sıfır olduğunda (yani, arabellek boş olduğunda) arabelleğe veri koyun ve yalnızca sayı bir olduğunda (yani, arabellek dolduğunda) arabellekten veri alın. Senkronizasyon kodunu, bir üreticinin verileri dolu bir arabelleğe koyacağı veya bir tüketicinin boş bir arabellekten veri alacağı şekilde yazarsak, yanlış bir şey yapmış oluruz (ve bu kodda bir iddia ortaya çıkar).

Bu iş iki tip thread ile yapılacak, bunlardan birine **üretici (producer)** ipleri, diğer grubuna da **tüketici (consumer)** ipleri diyeceğiz. Şekil 30.7, paylaşılan arabellek döngülerine birkaç kez bir tamsayı koyan bir üreticinin ve paylaşılan arabellekten çektiği veri öğesini her yazdırdığında verileri bu paylaşılan arabellekten (sonsuza kadar) alan bir tüketicinin kodunu gösterir.

Kırık Bir Çözüm

Şimdi tek bir üreticimiz ve tek bir tüketicimiz olduğunu hayal edin. Açıkçası, `put()` ve `get()` yordamlarının içlerinde kritik bölümleri vardır, çünkü `put()` arabelleği günceller ve `get()` arabelleği okur. Ancak, kodu kilitlemek işe yaramaz; daha fazlasına ihtiyacımız var.

```
1  int loops; // must initialize somewhere...
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: Producer/Consumer: Single CV And If Statement

Daha fazlasının bazı koşul değişkenleri olması şaşırtıcı değildir. Bu (bozuk) ilk denemede (Şekil 30.8), tek koşul değişkenimiz `cond` ve ilişkili kilit `mutex` var.

Üreticiler ve tüketiciler arasındaki sinyalleşme mantığını inceleyelim. Üretici tamponu doldurmak istediğinde, tamponun boşalmasını bekler (p1–p3). Tüketici tamamen aynı mantığa sahiptir, ancak farklı bir koşul bekler: dolgunluk (c1–c3). Tek bir üretici ve tek bir tüketici ile Şekil 30.8'deki kod çalışır. Ancak, bu iş parçacığından birden fazlasına sahipsek (örneğin, iki tüketici), çözümün iki kritik sorunu vardır. Onlar neler?

...(düşünmek için burada duraklayın)...

Beklemeden önceki if ifadesiyle ilgili olan ilk sorunu anlayalım. İki tüketici (Tc1 ve Tc2) ve bir üretici (Tp) olduğunu varsayalım. Önce bir tüketici (Tc1) çalışır; kilidi alır (c1), herhangi bir tamponun tüketime hazır olup olmadığını kontrol eder (c2) ve hiçbirinin hazır olmadığını görünce bekler (c3) (kilidi serbest bırakır).

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Run	0	
	Sleep		Ready	p2	Run	0	
	Sleep		Ready	p4	Run	1	Buffer now full
	Ready		Ready	p5	Run	1	T_{c1} awoken
	Ready		Ready	p6	Run	1	
	Ready		Ready	p1	Run	1	
	Ready		Ready	p2	Run	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Run		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Run		Sleep	1	
	Ready	c4	Run		Sleep	0	... and grabs data
	Ready	c5	Run		Ready	0	T_p awoken
	Ready	c6	Run		Ready	0	
c4	Run		Ready		Ready	0	Oh oh! No data

Şekil 30.9: Thread Trace: Broken Solution (v1)

Ardından üretici (Tp) çalışır. Kilidi alır (p1), tüm tamponların dolu olup olmadığını kontrol eder (p2) ve durumun böyle olmadığını anlayarak devam eder ve tamponu doldurur (p4). Üretici daha sonra bir tamponun doldurulduğunu bildirir (p5). Kritik olarak, bu, birinci tüketiciyi (Tc1) bir koşul değişkeninde uyumaktan hazır kuyruğuna taşır; Tc1 artık çalışabilir (ancak henüz çalışmıyor). Üretici daha sonra tamponun dolduğunu anlayana kadar devam eder ve bu noktada uyku moduna geçer (p6, p1–p3).

Sorunun ortaya çıktığı yer burasıdır: başka bir tüketici (Tc2) gizlice içeri girer ve arabellekte var olan bir değeri (c1, c2, c4, c5, c6, arabellek dolu olduğu için c3'teki beklemeyi atlayarak) tüketir. Şimdi Tc1'in çalıştığını varsayalım; beklemeden dönmeden hemen önce kilidi yeniden alır ve ardından geri döner. Daha sonra get() (c4) öğesini çağırır, ancak tüketilecek arabellek yoktur! Bir iddia tetiklenir ve kod istendiği gibi çalışmaz. Açıkçası, Tc1'in tüketmeye çalışmasını bir şekilde engellemeliydik çünkü Tc2 gizlice girdi ve üretilen arabellekteki 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: Üretici Tc1'i uyandırdıktan sonra, ancak Tc1 hiç çalışmadan önce, sınırlandırılmış tamponun durumu değişti (Tc2 sayesinde). Bir iş parçacığının sinyalini vermek onları yalnızca uyandırır; bu nedenle, dünyanın durumunun değiştiğine dair bir ipucudur (bu durumda, tampona bir değer yerleştirilmiştir), ancak uyandırılan iş parçacığı çalıştığında, durumun hala istendiği gibi olacağına dair bir garanti yoktur.

```
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: Producer/Consumer: Single CV And While

Bir sinyalin ne anlama geldiğine ilişkin bu yoruma, bu şekilde bir koşul değişkeni oluşturan ilk araştırmadan sonra genellikle **Mesa semantiği (Mesa semantics)** denir [LR80]; **Hoare semantiği (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 (Mesa semantics)** kullanır.

Daha İyi, Ama Yine de Bozuk: While, Not If

Neyse ki, bu düzeltme kolaydır (Şekil 30.10): `if` 'i `while` olarak değiştirin. Bunun neden işe yaradığını düşünün; şimdi tüketici Tc1 uyanır ve (kilit tutularak) paylaşılan değişkenin (c2) durumunu hemen yeniden kontrol eder. Bu noktada arabellek boşsa, tüketici basitçe uykuya geri döner (c3). Sonuç `if` de üreticide bir `while` olarak değiştirilir (p2). Mesa semantiği sayesinde, koşul değişkenleri ile hatırlanması gereken basit bir kural, **her zaman while döngüsü 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}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Run		Ready	0	
	Sleep	c2	Run		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Run	0	
	Sleep		Sleep	p2	Run	0	
	Sleep		Sleep	p4	Run	1	Buffer now full
	Ready		Sleep	p5	Run	1	T_{c1} awoken
	Ready		Sleep	p6	Run	1	
	Ready		Sleep	p1	Run	1	
	Ready		Sleep	p2	Run	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Run		Sleep		Sleep	1	Recheck condition
c4	Run		Sleep		Sleep	0	T_{c1} grabs data
c5	Run		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Run		Ready		Sleep	0	
c1	Run		Ready		Sleep	0	
c2	Run		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Run		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Şekil 30.11: Thread Trace: Broken Solution (v2)

Doğru anladığınızı onaylayalım veya belki de şu anda uyanık olduğunuzu ve kitabın bu bölümünü okuduğunuzu onaylayalım. Sorun, önce iki tüketici çalıştığında (Tc1 ve Tc2) ve her ikisi de uykuya daldığında (c3) ortaya çıkar. Ardından, üretici çalışır, tampona bir değer koyar ve tüketicilerden birini uyandırır (Tc1 diyelim). Üretici daha sonra geri döner (yol boyunca kilidi serbest bırakır ve yeniden alır) ve ara belleğe daha fazla veri koymaya çalışır; arabellek dolu olduğundan, üretici bunun yerine koşulu bekler (dolayısıyla uyur). Şimdi, bir tüketici çalışmaya hazır (Tc1) ve iki iş parçacığı bir koşulda uyuyor (Tc2 ve Tp). Bir soruna neden olmak üzereyiz: işler heyecan verici bir hal alıyor!

Tc1 tüketicisi daha sonra `wait()` (c3) işlevinden dönerek uyanır, koşulu yeniden kontrol eder (c2) ve arabelleği dolu bularak değeri (c4) tüketir. Bu tüketici daha sonra, kritik olarak, yalnızca uykuda olan bir iş parçacığını uyandıran koşul (c5) hakkında sinyal verir. Ancak, hangi iş parçacığını uyandırmalı?

Tüketici tamponu boşalttığı için üreticiyi açıkça uyandırmalıdır. Ancak, tüketici Tc2'yi uyandırırsa (bu, bekleme kuyruğunun nasıl yönetildiğine bağlı olarak kesinlikle mümkündür), bir sorunumuz var demektir. Spesifik olarak, tüketici Tc2 uyanır ve arabelleği boş bulur (c2) ve uykuya geri döner (c3).

```
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 30.13: Producer/Consumer: Two CVs And While

Tampona girecek değeri olan üretici Tp uykuda bırakılır. Diğer tüketici iş parçacığı Tc1 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 tüketici diğer tüketicileri değil, sadece üreticileri uyandırmalıdır ve bunun tersi de geçerlidir.

Tek Arabellekli Üretici/Tüketici 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, üretici iş parçacıkları **boş (empty)** koşulu bekler ve sinyaller **dolar (fill)**. Tersine, tüketici iş parçacıkları **dolduğunda (fill)** bekler ve **boş (empty)** sinyali verir. Bunu yaparak, tasarım yoluyla yukarıdaki ikinci sorundan kaçınılır: bir tüketici asla bir tüketiciyi yanlışlıkla uyandıramaz ve bir üretici asla bir üreticiyi yanlışlıkla uyandıramaz.

```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

Figure 30.13: **The Correct Put And Get Routines**

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        while (count == MAX)                   // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        put(i);                                // p4
        Pthread_cond_signal(&fill);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            Pthread_cond_wait(&fill, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&empty);           // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

Şekil 30.14: The Correct Producer/Consumer Synchronization

İPUCU: ŞARTLAR İÇİN WHILE (NOT IF) KULLANIN

Çok iş parçacıklı bir programda bir koşulu kontrol ederken, bir `while` döngüsü kullanmak her zaman doğrudur; Bir `if` ifadesi kullanmak, 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 Üretici/Tüketici Çözümü

Artık tamamen genel olmasa da çalışan bir üretici/tüketici çözümümüz var. Yaptığımız son değişiklik, daha fazla eş zamanlılık ve verimlilik sağlamak; özellikle, daha fazla arabellek yuvası ekliyoruz, böylece uyumadan önce birden çok değer üretilebilir ve benzer şekilde uyumadan önce birden çok değer tüketilebilir. Yalnızca tek bir üretici ve tüketici ile bu yaklaşım, içerik geçişlerini azalttığı için daha verimlidir; birden fazla üretici veya tüketici (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, tampon yapısının kendisinde ve karşılık gelen `put()` ve `get()` içindedir (Şekil 30.13). Üreticilerin ve tüketicilerin uyuyup uyumamak için kontrol ettiği koşulları da biraz değiştiriyoruz. Ayrıca doğru bekleme ve sinyal verme mantığını da gösteriyoruz (Şekil 30.14). Bir üretici yalnızca tüm arabellekler o anda doluysa uyur (p2); benzer şekilde, bir tüketici yalnızca tüm arabellekler şu anda boşsa uyur (c2). Üretici/tüketici sorununu da böylece çözmüş oluyoruz; arkanıza yaslanıp soğuk bir şeyler içme zamanı

30.3 Kaplama Koşulları

Şimdi koşul değişkenlerinin nasıl kullanılabileceğine dair bir örneğe daha bakacağız. Bu kod çalışması, yukarıda açıklanan **Mesa semantiğini (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 bir çok 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 olması 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  // how many bytes of the heap are free?
2  int bytesLeft = MAX_HEAP_SIZE;
3
4  // need lock and condition too
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 = ...; // get mem from heap
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); // whom to signal??
23     Pthread_mutex_unlock(&m);
24 }
```

Şekil 30.15: Covering Conditions: An Example

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

Bu noktada, üçüncü bir iş parçacığının, Tc'nin `free(50)` çağırdığını varsayalım. Ne yazık ki, bekleyen bir iş parçacığını uyandırmak için sinyal çağırdığında, yalnızca 10 baytın serbest bırakılmasını bekleyen doğru bekleyen iş parçacığını, Tb uyandırmayabilir; Henüz yeterli bellek olmadığı için Ta 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.

Zeki okuyucu, bu yaklaşımı daha önce kullanabileceğimizi de fark etmiş olabilir (yalnızca tek koşul değişkenli üretici/tüketici sorununa bakın). Ancak o durumda elimizde 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; düzelt! 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. Özgeçmişler, bazı program durumları istenildiği gibi olmadığında iş parçacıklarının uyumasına izin vererek, ünlü (ve hala önemli olan) üretici/tüketici sorunu ve kapsama koşulları dahil olmak üzere bir dizi önemli senkronizasyon sorununu düzgün bir şekilde çözmemizi sağlar. Buraya "Big Brother'ı severdi" [K49] gibi daha dramatik bir sonuç cümlesi gelirdi.

Referanslar

[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 üretici/tüketici 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" 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 fikriniz var mı?

   Cevap:

2. Bir üretici ve bir tüketici ile çalıştırın ve üreticinin birkaç değer üretmesini sağlayın. Bir arabellekle (boyut 1) başlayın ve ardından artırın. Kodun davranışı daha büyük arabelleklerle nasıl değişir? (ya da öyle mi?) Tüketici uyku dizisini varsayılandan (uyku yok) değiştirdiğinizde, farklı arabellek boyutları (ör. -m 10) ve farklı sayıda üretilen öğe (ör. -l 100) ile num_full'un ne olacağını tahmin edersiniz?) -C 0,0,0,0,0,0,1?

   Cevap:

   ```
   $ ./main-two-cvs-while -p 1 -c 1 -m 1 -v
   $ ./main-two-cvs-while -p 1 -c 1 -m 10 -v
   $ ./main-two-cvs-while -p 1 -c 1 -m 1 -l 100 -v
   $ ./main-two-cvs-while -p 1 -c 1 -m 1 -C 0,0,0,0,0,0,1 -v
   $ ./main-two-cvs-while -p 1 -c 1 -m 10 -l 10 -C 0,0,0,0,0,0,1 -v
   ```

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

   Cevap: Linux `$ ./main-two-cvs-while -p 1 -c 3 -l 6 -v`

4. Bazı zamanlamalara bakalım. Bir üretici, üç tüketici, tek girişli paylaşılan arabellek ve her tüketicinin c3 noktasında bir saniye duraklaması ile aşağıdaki yürütmenin ne kadar süreceğini düşünüyorsunuz? `./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,0,1,0,0,0:0,0,0,1,0,0,0:0,0,0,1,0,0,0 -l 10 -v -t`

   Cevap: 11 saniye. Thread kilitlendikten sonra uygulama uykuya geçer.

5. Şimdi paylaşılan arabelleğin boyutunu 3 (`-m 3`) olarak değiştirin. Bu toplam süre içinde herhangi bir fark yaratacak mı?

   Cevap: 11 c3 var, yani 11 saniye.

6. Şimdi yine tek girişli bir arabellek kullanarak uyku konumunu c6 olarak değiştirin (bu, bir tüketiciyi sıradan bir şey alıp sonra onunla bir şeyler yaparken modeller). Bu durumda ne zaman tahmin edersiniz?
   `./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,0,0,0,0,1:0,0,0,0,0,0,1:0,0,0,0,0,0,1 -l 10 -v -t`

   Cevap: 12 kez 'sleep' çağırır, ancak 5 saniye kullanır. Çünkü, diğer iş parçacıklarının hareket edebilmesi için kilidi açtıktan sonra 'sleep'tedir.

7. Son olarak arabellek boyutunu tekrar 3 olarak değiştirin (`-m 3`). Şimdi ne zaman tahmin ediyorsunuz?

   Cevap: 13 c6; 5 saniye.

8. Şimdi `main-one-cv-while.c`'ye bakalım. Bu kodla ilgili bir soruna yol açacak şekilde, tek bir üretici, bir tüketici ve 1 boyutunda bir arabellek varsayarak bir uyku stringi yapılandırabilir misiniz?

   Cevap: Yapılandırılamaz.

9. Şimdi tüketici sayısını ikiye değiştirin. Kodda sorun yaratacak şekilde üretici ve tüketiciler için uyku dizileri oluşturabilir misiniz?

   Cevap: `$ ./main-one-cv-while -c 2 -v -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 tüketicinin olduğu durumu ve ardından birden fazla tüketicinin olduğu durumu düşünün.

    Cevap: Biri iyi, ikisi Şekil 30.7'deki gibi: c3'te hazır, ancak çalışırken veri akışı olmaz. 'while' döngüsü kullanılması gerekir.

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?

    Cevap:

    ```
    $ ./main-two-cvs-while-extra-unlock -p 1 -c 2 -m 10 -l 10
    -v -C 0,0,0,0,1,0,0:0,0,0,0,0,0,0
    ```