30

Koşul Değişkenleri (Condition Variables)

Şimdiye kadar bir kilit kavramını geliştirdik ve doğru donanım ve işletim sistemi desteği kombinasyonu ile nasıl düzgün bir şekilde oluşturulabileceğini gördük. Ne yazık ki, eşzamanlı programlar oluşturmak için gereken tek ilkeller kilitler değildir.Özellikle, bir iş parçacığının yürütülmesine devam etmeden önce bir koşul (**condition)** 'un doğru olup olmadığını kontrol etmek istediği birçok durum vardır. Örneğin, bir üst iş parçacığı, devam etmeden önce bir alt iş parçacığının tamamlanıp tamamlanmadığını kontrol etmek isteyebilir (buna genellikle birleştirme () denir); 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: Çocuğunu Bekleyen Bir Ebeveyn (**A Parent Waiting For Its Child).**

Burada görmek istediğimiz şey şu çıktı:

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

Şekil 30.2'de gördüğünüz gibi paylaşılan bir değişken kullanmayı deneyebiliriz. Bu çözüm genellikle işe yarayacaktır, ancak ebeveyn döndükçe oldukça verimsizdir.

```
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); // create
       child
13     while (done == 0)
14         ; // spin
15     printf("parent: end\n");
16     return 0;
17 }
```

Şekil 30.2: Çocuğu Bekleyen Ebeveyn: Spin Tabanlı Yaklaşım

ve CPU zamanını boşa harcar. Bunun yerine burada istediğimiz şey, beklediğimiz durum gerçekleşene kadar (örneğin, çocuğun yürütülmesi bitene kadar) ebeveyni uyutmanın bir yoludur.

İŞİN ÖZÜ: BİR KOŞUL NASIL BEKLENİR

Çok iş parçacıklı programlarda, bir iş parçacığının devam etmeden önce bazı koşulların gerçekleşmesini beklemesi genellikle yararlıdır. Koşul gerçekleşene kadar sadece dönmenin basit yaklaşımı büyük ölçüde verimsizdir ve CPU döngülerini boşa harcar ve bazı durumlarda yanlış olabilir. Bu nedenle, bir iş parçacığı bir koşulu nasıl beklemelidir?

30.1 (Tanım ve Rutinler) Definition and Routines.

Bir koşulun gerçekleşmesini beklemek için, bir iş parçacığı koşul değişkeni (**condition variable)** olarak bilinen şeyi kullanabilir.Bir koşul değişkeni(**condition variable)**, bazı yürütme durumu (yani, bazı koşullar) istenildiği gibi olmadığında (koşulu bekleyerek) iş parçacıklarının kendilerini açabileceği açık bir kuyruktur; Başka bir iş parçacığı, söz konusu durumu değiştirdiğinde, bu bekleyen iş parçacıklarından birini (veya daha fazlasını) uyandırabilir ve böylece devam etmelerine izin verin (duruma göre **sig-naling** yaparak).Fikir Dijkstra'nın "özel semaforları" kullanmasına kadar uzanıyor [D68]; Benzer bir fikir daha sonra Hoare tarafından monitörler üzerindeki çalışmasında "koşul değişkeni" olarak adlandırıldı [H74].

Böyle bir koşul değişkenini bildirmek için şöyle bir şey yazılır: pthread cond t c;, c'yi bir koşul değişkeni olarak bildirir (not: uygun başlatma da gereklidir). Bir koşul değişkeninin kendisiyle ilişkili iki işlemi vardır: wait() ve signal(). Wait() çağrısı, bir iş parçacığı kendisini uyku moduna geçirmek 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: Çocuğu Bekleyen Ebeveyn: Bir Koşul Değişkeni Kullanın.

bir iş parçacığı programdaki bir şeyi değiştirdiğinde ve bu nedenle bu durumda bekleyen uyuyan 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() çağrısı hakkında fark edebileceğiniz bir şey, parametre olarak bir muteks almasıdır; wait () çağrıldığında bu muteksin kilitlendiğini varsayar. Wait() öğesinin sorumluluğu kilidi serbest bırakmak ve çağıran iş parçacığını uyku moduna geçirmektir (atomik olarak); İş parçacığı uyandığında (başka bir iş parçacığı bunu işaret ettikten sonra), arayana dönmeden önce kilidi yeniden alması gerekir. Bu karmaşıklık, kesinliği önleme arzusundan kaynaklanmaktadır.

bir iplik kendini uyutmaya çalışırken meydana gelen yarış koşulları. Bunu daha iyi anlamak için birleştirme sorununun çözümüne bir göz atalım (Şekil 30.3).

Dikkate alınması gereken iki durum var. İlkinde, ebeveyn alt iş parçacığını oluşturur, ancak kendi kendine çalışmaya devam eder (yalnızca tek bir işlemcimiz olduğunu varsayalım) ve böylece alt iş parçacığının tamamlanmasını beklemek için hemen thr join() öğesini çağırır. Bu durumda kilidi alır, çocuğun işi bitip bitmediğini kontrol eder (bitmez) ve wait() (dolayısıyla kilidi serbest bırakır) diyerek kendini uyutur. Çocuk sonunda çalışacak, "çocuk" mesajını yazdıracak ve üst iş parçacığını uyandırmak için thr exit () öğesini arayacaktır; Bu kod sadece kilidi alır, durum değişkenini ayarlar bitti ve ebeveyne sinyal vererek onu uyandırır. Son olarak, ebeveyn çalışır (kilit tutularak wait() öğesinden döner), kilidin kilidini açar ve son "parent: end" mesajını yazdırır.

İkinci durumda, çocuk yaratılıştan hemen sonra çalışır, done değerini 1 olarak ayarlar, uyuyan bir ipliği uyandırmak için signal çağırır (ancak hiçbiri yoktur, bu yüzden geri döner) ve tamamdır. Ebeveyn daha sonra çalışır, çağırır thr join (), yapıldığını görür 1ve böylece beklemez ve geri döner.

Son bir not: Koşulu bekleyip beklememeye karar verirken ebeveynin yalnızca bir if ifadesi yerine bir while döngüsü kullandığını gözlemleyebilirsiniz. Bu, programın mantığına göre 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 uygulamayı deneyelim. İlk olarak, durum değişkeninin yapılması gerekip gerekmediğini merak ediyor olabilirsiniz. Kod aşağıdaki örneğe benziyorsa ne olur? (Şekil 30.4)

Ne yazık ki bu yaklaşım bozuldu. Çocuğun hemen koştuğu ve hemen thr exit() çağırdığı durumu hayal edin; Bu durumda çocuk sinyal verir, ancak koşulda uyuyan bir iplik yoktur. Ebeveyn çalıştığında, sadece beklemeye çağırır ve sıkışır; hiçbir iş parçacığı onu uyandırmaz. Bu örnekten, yapılan durum değişkeninin önemini takdir etmelisiniz; İş parçacıklarının bilmekle ilgilendiği değeri kaydeder. Uyku, uyanma ve kilitlenmenin hepsi 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: Ebeveyn Bekleme: Durum Değişkeni Yok

```
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: Ebeveyn Bekleme: Kilit Yok

Burada (Şekil 30.5) başka bir zayıf uygulama var. Bu örnekte, sinyal vermek ve beklemek için birinin kilit tutmasına gerek olmadığını hayal ediyoruz. Burada ne gibi bir sorun olabilir? Bir düşünün 1! Buradaki sorun ince bir yarış koşuludur. Özellikle, ebeveyn join() öğesini çağırır ve ardından done değerini kontrol ederse, bunun 0 olduğunu görür ve böylece uyumaya çalışır. Ancak aramadan hemen önce uyumayı bekleyin, ebeveyn kesintiye uğrar ve çocuk koşar. Çocuk, yapılan durum değişkenini 1 olarak değiştirir ve sinyal verir, ancak hiçbir iş parçacığı beklemez ve bu nedenle hiçbir iş parçacığı uyanmaz. Ebeveyn tekrar koştuğunda sonsuza kadar uyur ki bu üzücüdür.
Umarım, bu basit birleştirme örneğinden, koşul değişkenlerini doğru kullanmanın bazı temel gereksinimlerini görebilirsiniz. Anladığınızdan emin olmak için şimdi daha karmaşık bir örnekten geçiyoruz: üretici / tüketici(**producer/consumer)** veya sınırlı arabellek sorunu (**bounded buffer).**

İPUCU: SİNYAL VERİRKEN KİLİDİ DAİMA BASILI TUTUN
Her durumda kesinlikle gerekli olmasa da, koşul değişkenlerini kullanırken sinyal verirken kilidi tutmak muhtemelen en basit ve en iyisidir. Yukarıdaki örnek, doğru olması için kilidi tutmanız gereken bir durumu göstermektedir; Bununla birlikte, muhtemelen yapmamanın iyi olduğu, ancak muhtemelen kaçınmanız gereken başka durumlar da vardır. Bu nedenle, basitlik için, sinyali çağırırken kilidi tutun(hold the lock when calling signal**).**Bu ipucunun tersi, yani beklerken kilidi tutmak sadece bir ipucu değil, bekle semantiği tarafından zorunlu kılınmıştır, çünkü her zaman bekle
(a) aradığınızda kilidin tutulduğunu varsayar, (b) arayanı uyuturken söz konusu kilidi serbest bırakır ve (c) geri dönmeden hemen önce kilidi yeniden alır. Bu nedenle, bu ipucunun genelleştirilmesi doğrudur: sinyali ararken kilidi tutun veya bekleyin(hold the lock when calling signal or wait) ve her zaman iyi durumda olacaksınız.

1BU örneğin "gerçek" kod olmadığını unutmayın, çünkü pthread cond wait() çağrısı her zaman bir koşul değişkeninin yanı sıra bir muteks gerektirir; Burada, yalnızca arabirimin negatif örnek uğruna bunu yapmadığını iddia ediyoruz.

```
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: Koy Ve Al Yordamları (v1)

30.2 Üretici / Tüketici (Sınırlı Arabellek) Sorunu

Bu bölümde karşılaşacağımız bir sonraki senkronizasyon sorunu, ilk olarak Dijkstra [D72] tarafından gündeme getirilen üretici / tüketici sorunu(**producer/consumer**) veya bazen sınırlı arabellek sorunu (**buffer problem**), olarak bilinir. Genelleştirilmiş Dijkstra ve meslektaşları, bu üretici / tüketici sorununu icat etmeye yol açan gerçekten semafor (değişken veya koşul olarak kullanılabilecek bir kilit) [D01]; Daha sonra semaforlar hakkında daha fazla bilgi edineceğizBir veya daha fazla üretici iş parçacığı ve bir veya daha fazla tüketici iş parçacığı düşünün. Üreticiler veri öğeleri oluşturur ve bunları bir arabelleğe yerleştirir; Tüketiciler söz konusu öğeleri arabellekten alır ve bir şekilde tüketir.

Bu düzenleme birçok gerçek sistemde gerçekleşir. Örneğin, çok iş parçacıklı bir web sunucusunda, bir üretici HTTP isteklerini bir çalışma kuyruğuna (yani sınırlı arabelleğe) koyar; tüketici iş parçacıkları bu kuyruktan istekleri alır ve işler.Bir programın çıktısını diğerine aktardığınızda, örneğin grep foo dosyası gibi sınırlı bir arabellek de kullanılır.txt / wc -l. Bu örnek aynı anda iki işlemi çalıştırır; grep dosyadan satır yazar.içinde foo dizesi bulunan txt, standart çıktı olduğunu düşündüğü şeye; UNİX kabuğu, çıktıyı UNİX kanalı (kanal**(Pipe**) sistem çağrısı tarafından oluşturulan) olarak adlandırılana yönlendirir. Bu borunun diğer ucu, sadece giriş akışındaki satır sayısını sayan ve sonucu yazdıran işlem wc'nin standart girişine bağlanır. Böylece grep süreci üreticidir; wc süreci tüketicidir; aralarında çekirdek içi sınırlı bir arabellek vardır; Bu örnekte siz sadece mutlu kullanıcısınız. Sınırlı arabellek paylaşılan bir kaynak olduğundan, elbette ona senkronize erişim gerektirmeliyiz, lst2 bir yarış koşulu ortaya çıkar. Bu sorunu daha iyi anlamaya başlamak için bazı gerçek kodları inceleyelim.

İhtiyacımız olan ilk şey, bir üreticinin veri koyduğu ve bir tüketicinin veri aldığı paylaşılan bir arabellektir. Sadece bir tane kullanalım

2Burası sana ciddi eski ingilizceyi bıraktığımız yer, ve dilek kipi formu

```
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 Konuları (v1) (**Producer/Consumer Threads (v1)**

basitlik için tamsayı (bunun yerine bir veri yapısına bir işaretçi yerleştirmeyi kesinlikle hayal edebilirsiniz) ve paylaşılan arabelleğe bir değer koymak ve arabellekten bir değer almak için iki iç yordam. Ayrıntılar için Şekil 30.6'ya (sayfa 6) bakın.

Oldukça basit, değil mi? Put() yordamı arabelleğin boş olduğunu varsayar (ve bunu bir onaylama işlemi ile kontrol eder) ve ardından paylaşılan arabelleğe bir değer koyar ve count değerini 1 olarak ayarlayarak onu dolu olarak işaretler. Get() yordamı tam tersini yapar, arabelleği boş olarak ayarlar (yani, count değerini 0 olarak ayarlar) ve değeri döndürür. Bu paylaşılan arabelleğin yalnızca tek bir girişi olduğundan endişelenmeyin; Daha sonra, birden fazla girişi tutabilen ve göründüğünden daha eğlenceli olacak bir sıraya genelleyeceğiz.Şimdi, içine veri koymak veya ondan veri almak için arabelleğe erişmenin ne zaman uygun olduğunu bilen bazı rutinler yazmamız gerekiyor. Bunun koşulları açık olmalıdır: yalnızca sayım sıfır olduğunda (yani arabellek boş olduğunda) arabelleğe veri koyun ve yalnızca sayım bir olduğunda (yani arabellek dolduğunda) arabellekten veri alın. Senkronizasyon kodunu, bir üreticinin verileri tam bir arabelleğe koyacağı veya bir tüketicinin boş bir arabellekten veri alacağı şekilde yazarsak, yanlış bir şey yaptık (ve bu kodda bir onaylama işlemi tetiklenir).

Bu çalışma, biri üretici(**producer**) iş parçacığı, diğeri tüketici(**consumer**) iş parçacığı olarak adlandıracağımız iki tür iş parçacığı tarafından yapılacaktır. Şekil 30.7, paylaşılan arabellek döngülerine bir tamsayı koyan bir üreticinin kodunu ve paylaşılan arabellekten çektiği veri öğesini her yazdırdığında verileri bu paylaşılan arabellekten (sonsuza kadar) alan bir tüketiciyi göstermektedir. paylaşılan arabellek.

Bozuk Bir Çözüm.(A Broken Solution)

Şimdi tek bir üreticimiz ve tek bir tüketicimiz olduğunu hayal edin. Açıkçası, put () ve get () yordamları, put () arabelleği güncellediği ve get () ondan okuduğu için içlerinde kritik bölümlere sahiptir. Ancak, kodun etrafına bir kilit koymak 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: Üretici/Tüketici: Tek CV ve If Beyanı

(Producer/Consumer: Single CV And If Statement)

Şaşırtıcı olmayan bir şekilde, daha fazlası bazı koşul değişkenleridir. Bu (bozuk) ilk denemede (Şekil 30.8), tek bir koşul değişkeni cond ve ilişkili kilit muteksimiz var.Üreticiler ve tüketiciler arasındaki sinyal mantığını inceleyelim. Bir üretici arabelleği doldurmak istediğinde, arabelleğin boş olmasını bekler (p1– p3). Tüketici aynı mantığa sahiptir, ancak farklı bir koşul bekler: dolgunluk (c1-c3).

Sadece tek bir üretici ve tek bir tüketici ile Şekildeki kod

30.8 çalışır. Bununla birlikte, bu iş parçacıklarından birden fazlasına (örneğin iki tüketiciye) sahipsek, çözümün iki kritik sorunu vardır. Onlar ne?

... (düşünmek için burada durun) ...Beklemeden önce if durumu ile ilgili olan ilk sorunu anlayalım. İki tüketici (Tc1 ve Tc2) ve bir üretici (Tp) olduğunu varsayalım. İlk olarak, bir tüketici (Tc1) çalışır; kilidi (c1) alır, arabelleklerin tüketime hazır olup olmadığını kontrol eder (c2) ve hiçbirinin olmadığını bulur, bekler (c3) (kilidi serbest bırakır).

Ardından üretici (Tp) çalışır. Kilidi (p1) alır, hepsinin olup olmadığını kontrol eder.

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Koşmak		Hazir		Ready	0	
c2	Koşmak		Hazir		Ready	0	
c3	Uyku		Hazir		Ready	0	Nothing to get Alacak bir şey yok
	Uyku		Hazir	p1	Koşmak	0	
	Uyku		Hazir	p2	Koşmak	0	
	Uyku		Hazir	p4	Koşmak	1	Buffer now full
	Hazir		Hazir	p5	Koşmak	1	T_{c1} awoken Tampon şimdi dolu Tc1 uyandı
	Hazir		Hazir	p6	Koşmak	1	
	Hazir		Hazir	p1	Koşmak	1	
	Hazir		Hazir	p2	Koşmak	1	
	Hazir		Hazir	p3	Uyku	1	Buffer full; sleep
	Hazir	c1	Run		Uyku	1	T_{c2} sneaks in … Tampon dolu; uyku Tc2 içeri giriyor...
	Hazir	c2	Run		Uyku	1	
	Hazir	c4	Run		Uyku	0	... and grabs data
	Hazir	c5	Run		Hazir	0	T_p awoken The tampon is full; sleep Tc2 is coming in...
	Hazir	c6	Run		Hazir	0	
c4	Koşmak		Ready		Hazir	0	Oh oh! No data Oh oh! Veri yok

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

arabellekler dolu (p2) ve durumun böyle olmadığını tespit ederek devam eder ve arabelleği doldurur (p4). Üretici daha sonra bir tamponun doldurulduğunu bildirir (p5). Kritik olarak, bu, ilk tüketiciyi (Tc1) bir koşul değişkeni üzerinde uyumaktan hazır kuyruğa taşır; Tc1 artık çalışabilir (ancak henüz çalışmıyor). Üretici daha sonra tamponun dolduğunu fark edene kadar devam eder ve bu noktada uyur (p6, p1–p3).

Sorunun oluştuğu yer burasıdır: başka bir tüketici (Tc2) gizlice girer ve arabellekteki mevcut bir değeri tüketir (c1, c2, c4, c5, c6, arabellek dolu olduğu için c3'te beklemeye atlama- ping). Şimdi Tc1'in çalıştığını varsayalım; beklemeden dönmeden hemen önce kilidi yeniden alır ve sonra geri döner. Daha sonra get () (c4) çağırır, ancak tüketilecek arabellek yoktur! Bir onaylama işlemi tetiklenir ve kod istendiği gibi çalışmadı. Açıkçası, Tc1'in tüketmeye çalışmasını bir şekilde engellemeliydik çünkü Tc2 içeri girdi ve üretilen arabellekteki bir değeri tüketti. Şekil 30.9, her bir iş parçacığının gerçekleştirdiği eylemi ve zaman içindeki zamanlayıcı durumunu (Hazır, Çalışıyor veya Uyuyor) gösterir.Sorun basit bir nedenden dolayı ortaya çıkıyor: üretici Tc1'i uyandırdıktan sonra, ancak Tc1 çalıştırılmadan önce, sınırlı arabelleğin durumu değişti (Tc2

sayesinde). Bir iş parçacığına sinyal vermek yalnızca onları uyandırır; Bu nedenle, dünyanın durumunun değiştiğine dair bir ipucudur (bu durumda, arabelleğe bir değer yerleştirilmiştir), ancak uyanan iş parçacığı çalıştığında durumun yine de istenildiği gibi olacağına dair bir garanti yoktur. Bir sinyalin ne anlama geldiğinin bu yorumu, bir koşul değişkenini bu şekilde oluşturan ilk araştırmadan sonra genellikle Mesa semantiği(**Mesa semantics**) olarak anılır [LR80]; kontrast, olarak anılır

```
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 Süre (Producer/Consumer: Single CV And While)

Hoare semantiği(**Hoare semantics)**oluşturmak daha zordur, ancak uyanan ipliğin uyandığında hemen çalışacağına dair daha güçlü bir garanti sağlar [H74]. Şimdiye kadar yapılmış hemen hemen her sistem Mesa semantiğini kullanır.Daha iyi, Ama Yine de Bozuk: Neyse ki Olmasa da, bu düzeltme kolaydır (Şekil 30.10): ıf'yi bir süreliğine değiştirin. Bunun neden işe yaradığını düşünün; Şimdi tüketici Tc1 uyanır ve (kilit tutulduğunda) paylaşılan değişkenin (c2) durumunu hemen yeniden kontrol eder. Bu noktada arabellek boşsa, tüketici basitçe uyku moduna geçer (c3). Sonuç ıf, yapımcıda da bir süre olarak değiştirilir (p2). Mesa semantiği sayesinde koşul değişkenleri ile hatırlanması gereken basit bir kural her zaman while döngülerini kullanmaktır. Bazen durumu tekrar 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. Görebiliyor musun? Bunun tek bir koşul değişkeni olduğu gerçeğiyle bir ilgisi var. İleriyi okumadan önce sorunun ne olduğunu bulmaya çalışın. Yap şunu! (düşünmeniz için duraklatın veya gözlerinizi kapatın...)

Figure 30.11: **Thread Trace: Broken Solution (v2)**

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Koşmak		Hazır		Hazır	0	
c2	Koşmak		Hazır		Hazır	0	
c3	Uyku		Hazır		Hazır	0	Nothing to get Alacak bir şey yok
	Uyku	c1	Run		Hazır	0	
	Uyku	c2	Run		Hazır	0	Alacak bir şey yok
	Uyku	c3	Sleep		Hazır	0	Nothing to get
	Uyku		Uyku	p1	Koşmak	0	
	Uyku		Uyku	p2	Koşmak	0	
	Uyku		Uyku	p4	Koşmak	1	Buffer now full
	Uyku		Uyku	p5	Koşmak	1	T_{c1} awoken
	Uyku		Uyku	p6	Koşmak	1	((Tampon şimdi dolu Tc1 uyandı))
	Uyku		Uyku	p1	Koşmak	1	
	Hazır		Uyku	p2	Koşmak	1	
	Hazır		Uyku	p3	Uyku	1	Must sleep (full)
c2	Koşmak		Uyku		Uyku	1	Recheck condition
c4	Koşmak		Uyku		Uyku	0	T_{c1} grabs data
c5	Koşmak		Hazır		Uyku	0	Oops! Woke T_{c2}
c6	Koşmak		Hazır		Uyku	0	Uyumalı (dolu) Durumu tekrar kontrol et Tc1 verileri yakalar Eyvah! Tc2'yi Uyandırdı
c1	Koşmak		Hazır		Uyku	0	
c2	Koşmak		Hazır		Uyku	0	
c3	Uyku		Hazır		Uyku	0	Nothing to get Alacak bir şey yok
	Uyku	c2	Koşmak		Uyku	0	
	Uyku	c3	Uyku		Uyku	0	Everyone asleep… Herkes uyuyor

İş parçacığı izleme: Bozuk çözüm (v2)

Let's confirm your understanding

ya da belki şimdi uyanık olduğunuzu ve kitabın bu bölümünü okuduğunuzu doğrulayalım. Sorun, iki tüketici önce çalıştırdığında (Tc1 ve Tc2) ve her ikisi de uyku moduna geçtiğinde (c3) oluşur. Ardından üretici çalışır, arabelleğe bir değer koyar ve tüketicilerden birini uyandırır (Ts1 deyin). Üretici daha sonra geri döngü yapar (yol boyunca kilidi serbest bırakır ve yeniden sorgular) ve arabelleğe daha fazla veri koymaya çalışır; Arabellek dolu olduğundan, üretici bunun yerine koşulu bekler (böylece uyur). Şimdi, bir tüketici çalışmaya hazır (Tc 1) ve iki iş parçacığı bir koşulda uyuyor (Tc2 ve Tp). Bir soruna neden olmak üzereyiz: işler heyecanlanıyor!

Tüketici Tc1 daha sonra wait() (c3) öğesinden dönerek uyanır, koşulu (c2) yeniden kontrol eder ve arabelleği dolu bularak değeri (c4) tüketir. Bu tüketici daha sonra kritik olarak durumu (c5) işaret ederek uyuyan tek bir ipliği uyandırır. Ancak, hangi ipliği uyandırmalı?

Tüketici tamponu boşalttığı için üreticiyi açıkça uyandırması gerekir. Ancak, Tc2 tüketicisini uyandırırsa (ki bu, bekleme sırasının nasıl yönetildiğine bağlı olarak kesinlikle mümkündür), bir sorunumuz var. Özellikle, tüketici Tc2 uyanır ve arabelleği boş bulur (c2) ve uykuya geri döner (c3). Değeri olan üretici Tp

```
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 Özgeçmiş Ve SüreFigure
Producer/Consumer: Two CVs And While

tampon içine koymak için, uyku bırakılır. Diğer tüketici ipliği Tc1 de uykuya geri döner. Her üç iplik de uykuda kaldı, açık bir hata; Bu korkunç felaketin acımasız adım adım ilerlemesi için Şekil 30.11'e bakın. Sinyalizasyon açıkça gereklidir, ancak daha yönlendirilmelidir. Bir tüketici diğer tüketicileri, yalnızca üreticileri uyandırmamalı ve bunun tersi de geçerlidir.

Tek Tampon Üretici / Tüketici Çözümü
(The Single Buffer Producer/Consumer Solution)

Buradaki çözüm bir kez daha küçük: sistemin durumu değiştiğinde hangi tür iş parçacığının uyanması gerektiğini doğru bir şekilde bildirmek için bir yerine iki koşul değişkeni kullanın. Şekil 30.12'de ortaya çıkan kod gösterilmektedir.

Kodda, üretici iş parçacıkları boş durumda bekler ve sinyaller doldurulur(**fill**). Tersine, tüketici iş parçacıkları dolmayı bekler ve boş sinyal verir. Bunu yaparak, yukarıdaki ikinci sorundan tasarım gereği kaçınılır: bir tüketici bir tüketiciyi asla tesadüfen uyandıramaz ve bir üretici bir üreticiyi asla tesadüfen 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 30.13: Doğru Yerleştirme ve Alma 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 WHİLE (EĞER DEĞİLSE) KULLANIN

Çok iş parçacıklı bir programda bir koşul olup olmadığını kontrol ederken, bir while döngüsü kullanmak her zaman doğrudur; Yalnızca bir if ifadesi kullanmak, sinyalin anlambiliminde beklemede olabilir. Bu nedenle, her zaman while kullanın ve kodunuz beklendiği gibi davranacaktır.

Koşullu denetimlerin etrafındaki while döngülerini kullanmak, sahte uyanmaların meydana geldiği durumu da ele alır. Bazı iş parçacığı paketlerinde, uygulamanın ayrıntıları nedeniyle, yalnızca tek bir sinyal gerçekleşmiş olsa da iki iş parçacığının uyanması mümkündür [L11]. Sahte uyanmalar, bir iş parçacığının beklediği durumu yeniden kontrol etmek için başka bir nedendir.

Doğru Üretici/Tüketici Çözümü (The Correct Producer/Consumer Solution)

Artık tamamen genel olmasa da çalışan bir üretici / tüketici çözümümüz var. Yaptığımız son değişiklik, daha fazla eşzamanlılık ve verimlilik sağlamaktır; Özellikle, daha fazla arabellek yuvası ekliyoruz, böylece uyumadan önce birden fazla değer üretilebiliyor ve benzer şekilde uyumadan önce birden fazla değer tüketilebiliyor. Sadece tek bir üretici ve tüketici ile bu yaklaşım, bağlam anahtarlarını azalttığı için daha verimlidir; Birden fazla üretici veya tüketici (veya her ikisi) ile eşzamanlı üretim veya tüketimin gerçekleşmesine bile izin vererek eşzamanlılığı artırır. Neyse ki, mevcut çözümümüzden küçük bir değişiklik.

Bu doğru çözüm için ilk değişiklik, arabellek yapısının kendisinde ve karşılık gelen put() ve get() (Şekil 30.13). Üreticilerin ve tüketicilerin uyuyup uyumayacaklarını belirlemek için kontrol ettikleri koşulları da biraz değiştiriyoruz. Ayrıca doğru bekleme ve sinyal mantığını da gösteriyoruz (Şekil 30.14). Üretici, yalnızca tüm arabellekler şu anda doluysa uyur (p2); Benzer şekilde, tüketici yalnızca tüm arabellekler şu anda boşsa uyur (c2). Ve böylece üretici / tüketici sorununu çözüyoruz; arkanıza yaslanıp soğuk bir tane içmenin zamanı geldi.

30.2 Kaplama Koşulları (Covering Conditions).

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

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 serbest 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  // 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: Kaplama Koşulları: Bir ÖrnekFigure (Covering Conditions: An Example)

Aşağıdaki senaryoyu göz önünde bulundurun. Boş sıfır bayt olduğunu varsayalım; iş parçacığı Ta çağrıları allocate(100), ardından iş parçacığı Tb arayarak daha az bellek isteyen allocate(10). Böylece hem Ta hem de Tb koşulu bekler ve uyur; Bu isteklerden herhangi birini karşılayacak kadar boş bayt yoktur.Bu noktada, üçüncü bir iş parçacığı olan Tc'nin ücretsiz aradığını varsayalım (50). Ne yazık ki, bekleyen bir iş parçacığını uyandırmak için sinyal çağırdığında, yalnızca 10 baytın serbest bırakılmasını bekleyen doğru bekleyen iş parçacığı Tb'yi uyandırmayabilir; Yeterli bellek henüz boş olmadığından Ta beklemeye devam etmelidir. Bu nedenle, şekildeki kod çalışmaz, çünkü diğer iş parçacıklarını uyandıran iş parçacığı hangi iş parçacığının (veya iş parçacıklarının) uyanacağını bilmez.

Lampson ve Redell tarafından önerilen çözüm basittir: pthread cond signal() çağrısını, bekleyen tüm iş parçacıklarını uyandıran pthread cond broadcast () çağrısıyla yukarıdaki koda yeniden yerleştirin. Bunu yaparak, uyandırılması gereken tüm ipliklerin olduğunu garanti ederiz. Dezavantajı, elbette, (henüz) uyanık olmaması gereken diğer birçok bekleme ipliğini gereksiz yere uyandırabileceğimiz için olumsuz bir performans etkisi olabilir. Bu iplikler basitçe uyanacak, durumu tekrar kontrol edecek ve ardından hemen uykuya geri dönecektir.Lampson ve Redell, bir ipliğin uyanması gereken tüm durumları kapsadığı için böyle bir koşulu örtme koşulu (**covering condition**) olarak adlandırır (muhafazakar olarak); Tartıştığımız gibi maliyet, çok fazla ipliğin uyandırılabileceğidir

Anlayışlı okuyucu, bu yaklaşımı daha önce kullanabileceğimizi de fark etmiş olabilir (yalnızca tek bir koşul değişkenine sahip üretici / tüketici sorununa bakın). Ancak bu durumda bizim için daha iyi bir çözüm mevcuttu ve bu yüzden kullandık. Genel olarak, programınızın yalnızca sinyallerinizi yayın olarak değiştirdiğinizde çalıştığını fark ederseniz (ancak bunu yapmanız gerekmediğini düşünüyorsanız), muhtemelen bir hatanız vardır; düzelt! Ancak yukarıdaki bellek ayırıcısı gibi durumlarda, yayın mevcut en basit çözüm olabilir.

30.3 Özet

Kilitlerin ötesinde bir başka önemli senkronizasyon ilkelinin tanıtımını gördük: koşul değişkenleri. Cv'ler, bazı program durumları istenmediği zaman iş parçacıklarının uyumasına izin vererek, ünlü (ve yine de önemli) üretici / tüketici sorunu ve kapsama koşulları dahil olmak üzere bir dizi önemli senkronizasyon sorununu düzgün bir şekilde çözmemizi sağlar. “Ağabeyi sevdi” gibi daha dramatik bir sonuç cümlesi buraya gidecekti [O49].

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?

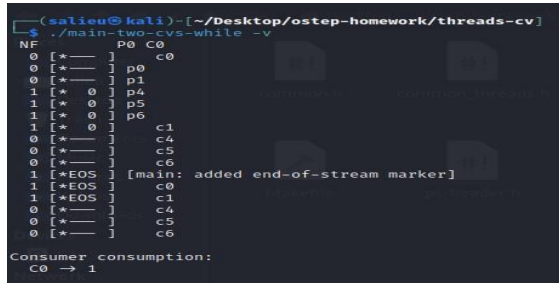
Ödev (Kod)---Homework (Code)

Bu ödev, bölümde tartışılan üretici / tüketici kuyruğunun çeşitli biçimlerini uygulamak için kilitler ve koşul değişkenleri kullanan bazı gerçek kodları keşfetmenizi sağlar. Gerçek koda bakacak, çeşitli konfigürasyonlarda çalıştıracak ve neyin işe yarayıp neyin yaramadığını ve diğer incelikleri öğrenmek için kullanacaksınız. Ayrıntılar için README sayfasını okuyun.

Sorular

1. İlk sorumuz main-two-cvs-while.c (çalışma çözümü). İlk önce kodu inceleyin. Programı çalıştırdığınızda ne olması gerektiğine dair bir fikriniz olduğunu düşünüyor musunuz?

Evet ,Bu bölümdeki şekillerin daha ayrıntılı, işleyen bir örneğidir.

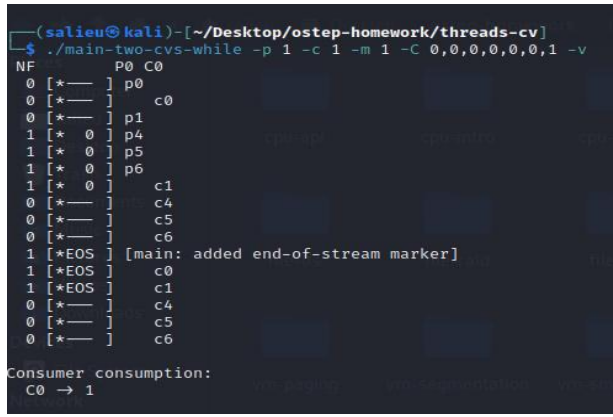


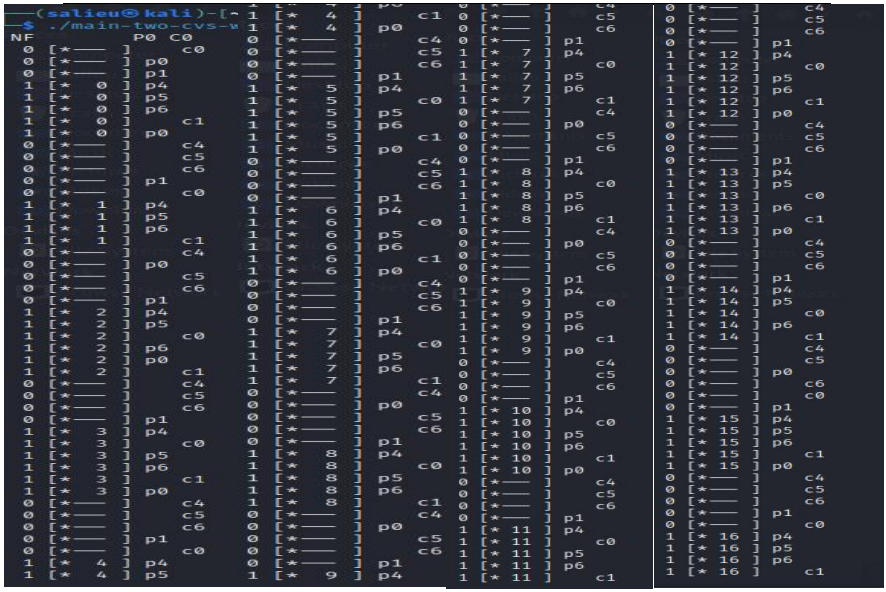
2. Bir üretici ve bir tüketici ile çalışın ve üreticinin birkaç değer üretmesini sağlayın. Bir arabellekle (boyut 1) başlayın ve ardından artırın. Kodun davranışı daha büyük arabelleklerle nasıl değişir? (yoksa öyle mi?) Tüketici uyku dizesini varsayılandan (uyku yok) -C 0,0,0,0,0,0,1 olarak değiştirdiğinizde, num full'un farklı arabellek boyutlarıyla (ör. -m 10) ve farklı sayıda üretilen öğeyle (ör. -l 100) ne olacağını tahmin edersiniz?

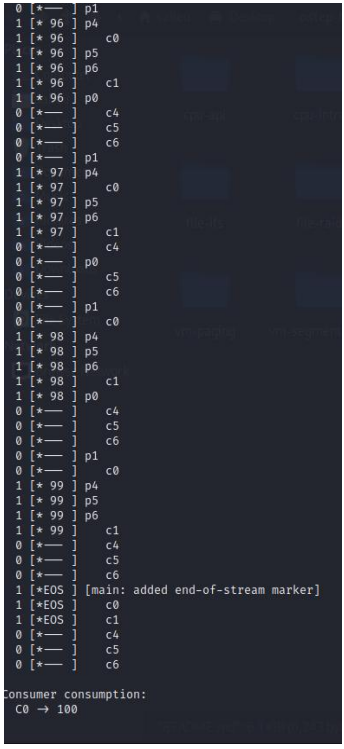
Kullanıcının arabelleğe birden çok değer ekleyebilmesi için geste değişir beklemeden önce ortam anahtarlarına bağlı olarak arabellek-1'in uzunluğuna kadar ve tüketici için tam tersi.

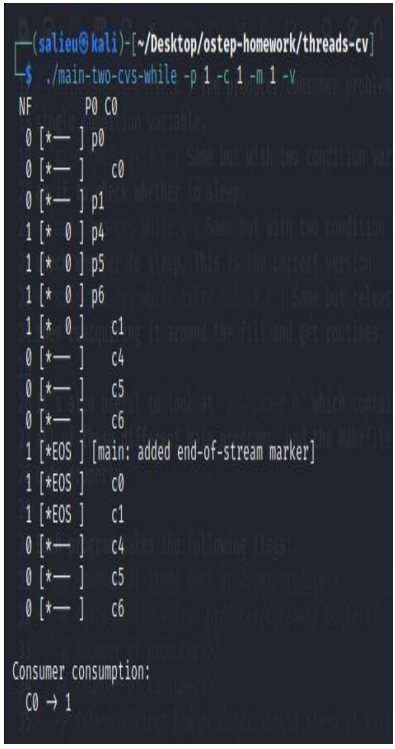
Num_full için beklentim, en büyük ve daha küçük arabellek boyutları arasında dalgalanmasıdır. Daha büyük boyutlarda ekstremiteye çarpmayabilir veya hiç çarpmayabilir. -P 1 -c 1 -m 10 -l 100 -v ile olan budur.

Arabellek boyutu> 1 ve -C 0,0,0,0,0,0,1 üretici doldurma Arabelleği beklenirken, bu noktada müşteri bir kez çalışır ve 1 dakika dinlenir. Tek bir tasfiye listesi olduğundan, döngünün bitimine kadar herkes sırayla birer birer ekleyerek / kaldırarak bir kez çalışacaktır




(salieu@kali)-[~
$ ./main-two-cvs-w
NF P0 C0


Consumer consumption:
C0 → 100

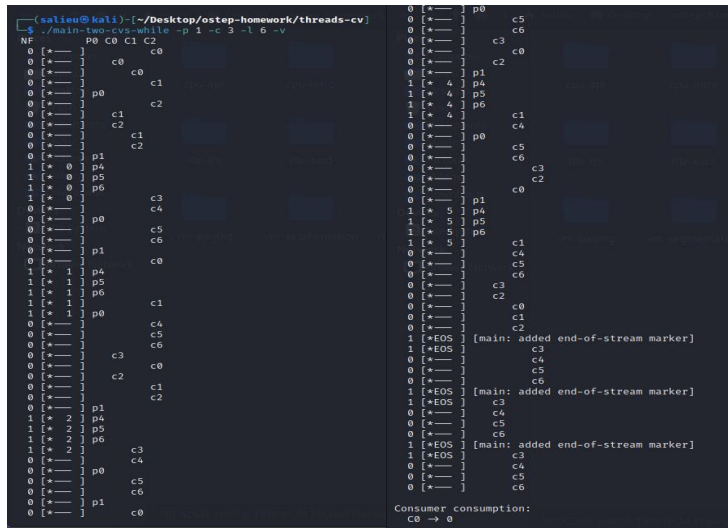

(salieu@kali)-[~/Desktop/ostep-homework/threads-cv]
$ ./main-two-cvs-while -p 1 -c 1 -m 1 -v
NF P0 C0
Consumer consumption:
C0 → 1

```
┌──(saliou㉿kali)-[~/Desktop/ostep-homework/threads-cv]
└─$ ./main-two-cvs-while -p 1 -c 1 -m 10 -v
 NF                                                P0 C0
  0 [*—  —  —  —  —  —  —  —  —  — ]   c0
  0 [*—  —  —  —  —  —  —  —  —  — ] p0
  0 [*—  —  —  —  —  —  —  —  —  — ]   c1
  0 [*—  —  —  —  —  —  —  —  —  — ]   c2
  0 [*—  —  —  —  —  —  —  —  —  — ] p1
  1 [u  0 f—  —  —  —  —  —  —  —  — ] p4
  1 [u  0 f—  —  —  —  —  —  —  —  — ] p5
  1 [u  0 f—  —  —  —  —  —  —  —  — ] p6
  1 [u  0 f—  —  —  —  —  —  —  —  — ]   c3
  0 [ —  *—  —  —  —  —  —  —  —  — ]   c4
  0 [ —  *—  —  —  —  —  —  —  —  — ]   c5
  0 [ —  *—  —  —  —  —  —  —  —  — ]   c6
  1 [ —  uEOS f—  —  —  —  —  —  —  — ] [main: added end-of-stream marker]
  1 [ —  uEOS f—  —  —  —  —  —  —  — ]   c0
  1 [ —  uEOS f—  —  —  —  —  —  —  — ]   c1
  0 [ —  —  *—  —  —  —  —  —  —  — ]   c4
  0 [ —  —  *—  —  —  —  —  —  —  — ]   c5
  0 [ —  —  *—  —  —  —  —  —  —  — ]   c6

Consumer consumption:
  C0 → 1
```

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

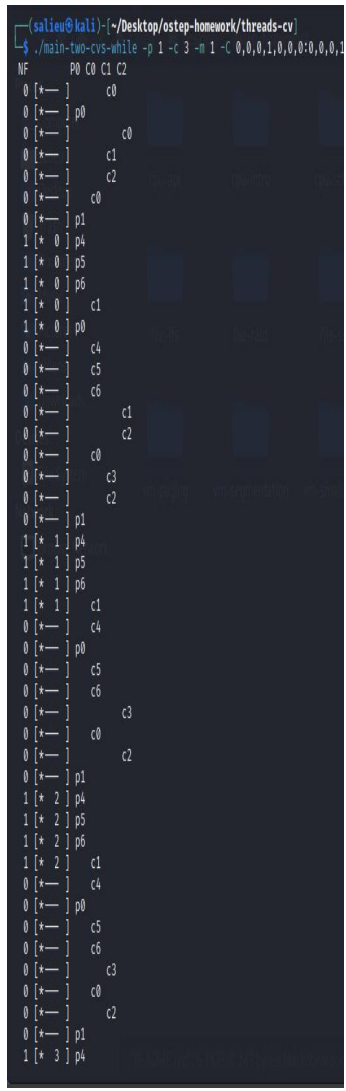
Evet. Iki İşlemcili bir Virtualbox'ta çalışan Linux'ta sonuç neredeyse şu şekildedir sadece bir işlemci.

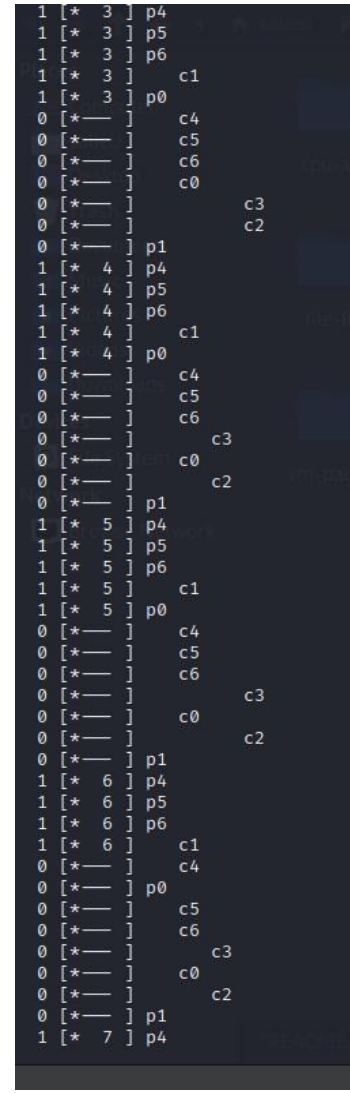


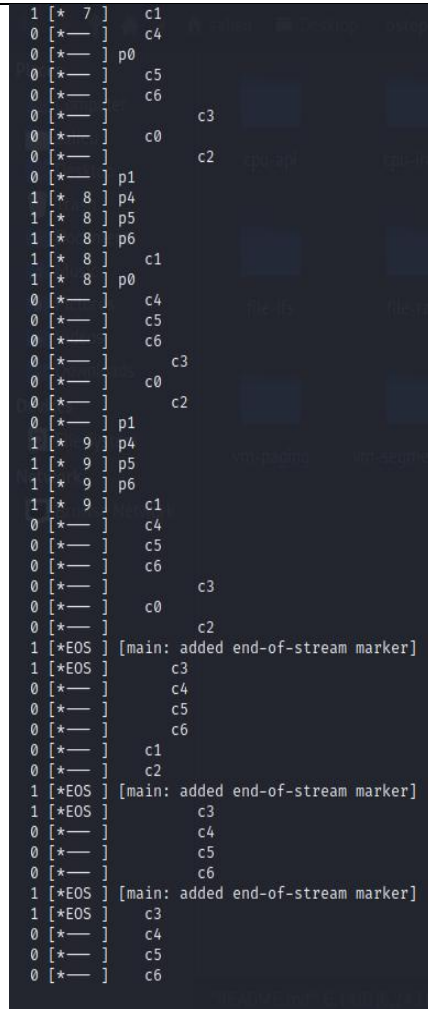
4. Bazı zamanlamalara bakalım. Bir üretici, üç tüketici, tek bir giriş paylaşılan arabelleği ve her tüketicinin c3 noktasında bir saniye duraklatılmasıyla aşağıdaki yürütmenin ne kadar süreceğini

düşünüyorsunuz? ./ ana-iki özgeçmiş-süre -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

Yanıtın yürütme süresinin ~ 12 saniye olduğu belirtilen son yanıt olduğu ortaya çıktı.

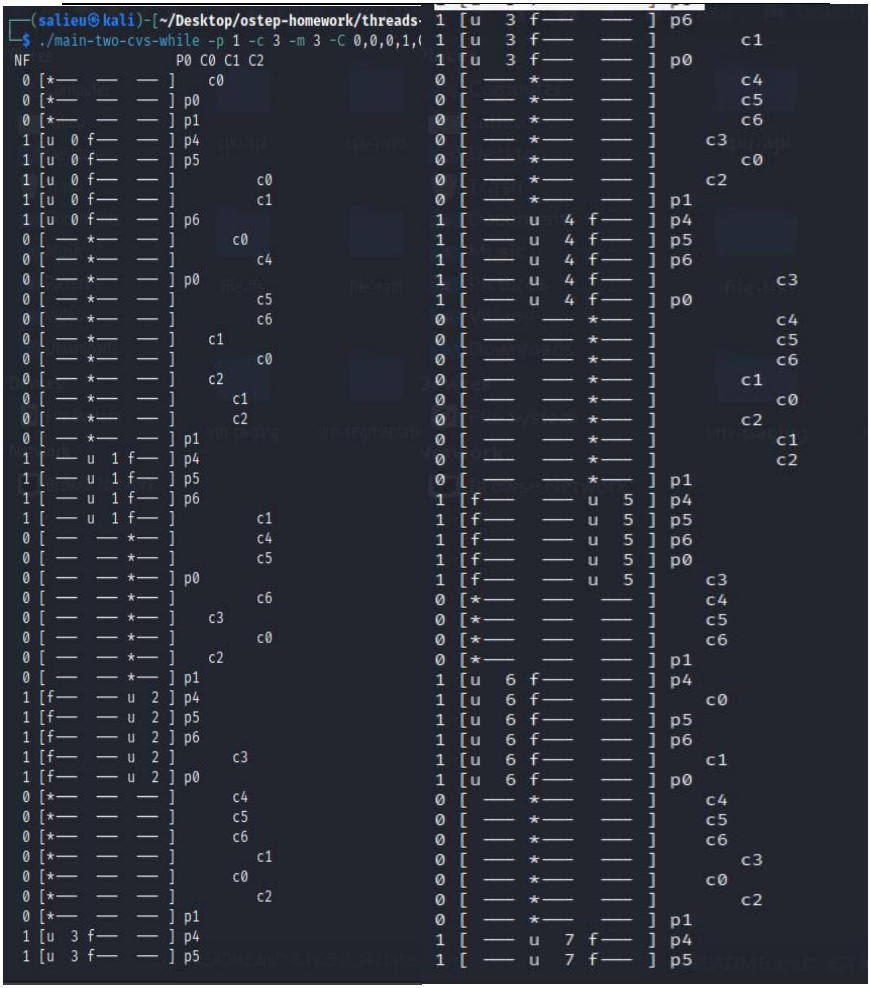


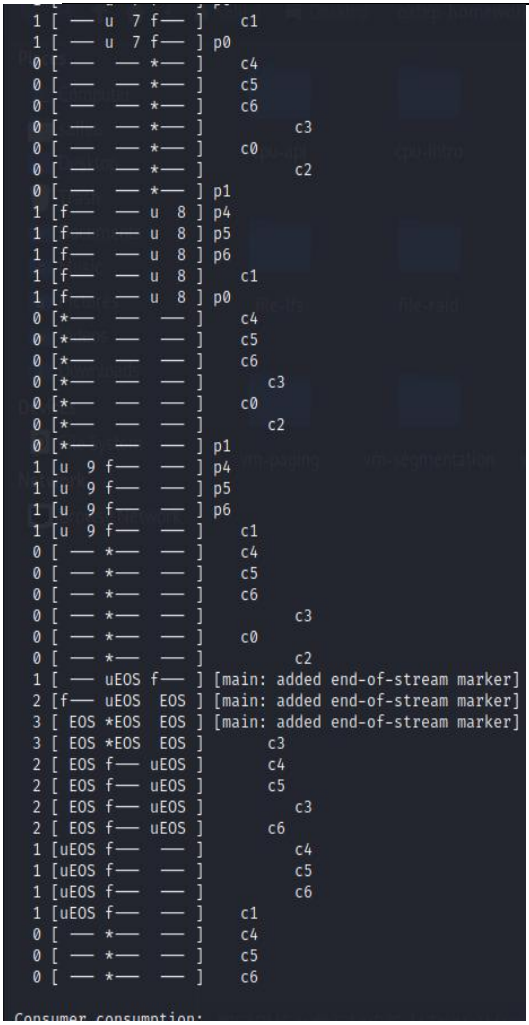




5. Şimdi paylaşılan arabelleğin boyutunu 3 (-m 3) olarak değiştirin. Bu toplam süre içinde herhangi bir fark yaratacak mı?

Tahminim neredeyse doğruydu. İddianame süresi yaklaşık olarak aynı, biraz daha az yanılıyor.

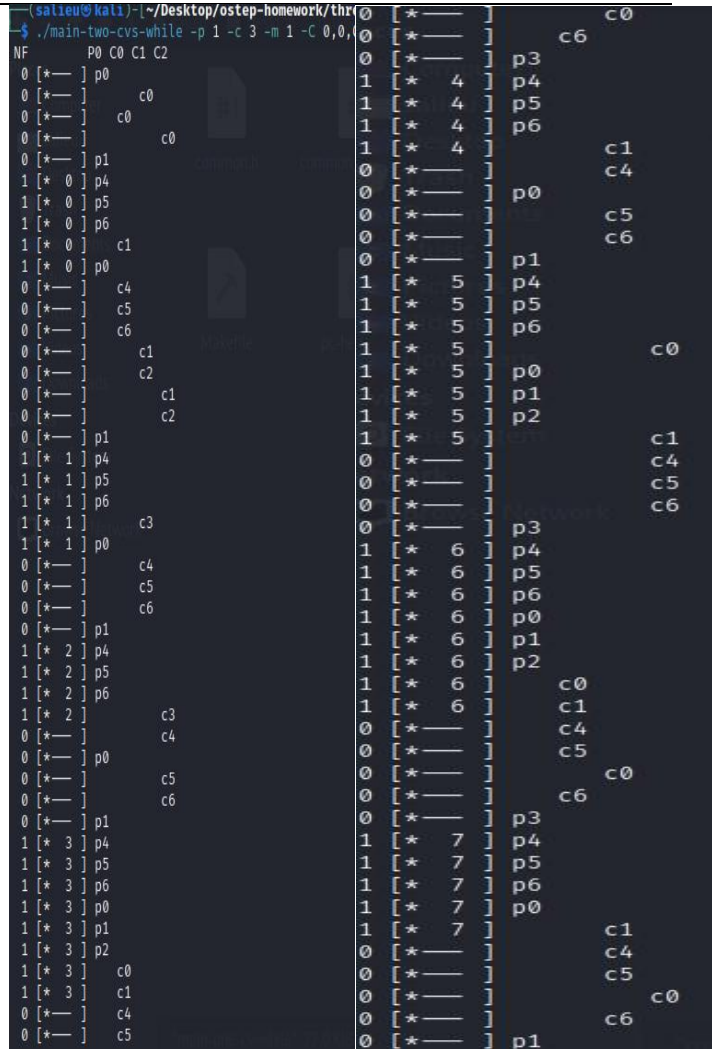

(salieu㉿kali)-[~/Desktop/ostep-homework/threads
$ ./main-two-cvs-while -p 1 -c 3 -m 3 -C 0,0,0,1,
NF P0 C0 C1 C2

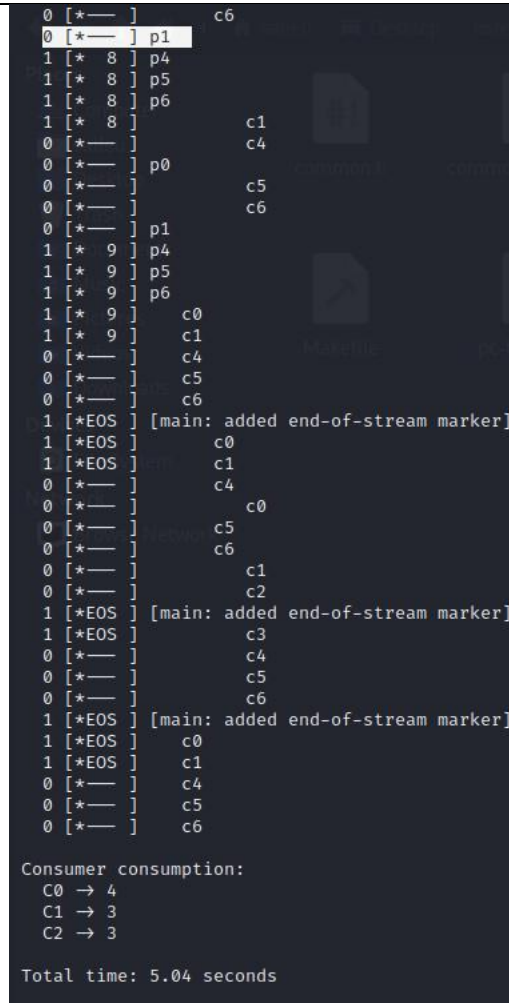

1 [── u 7 f──] c1
1 [── u 7 f──] p0
0 [── ── *──] c4
0 [── ── *──] c5
0 [── ── *──] c6
0 [── ── *──] c3
0 [── ── *──] c0
0 [── ── *──] c2
0 [── ── *──] p1
1 [f── ── u 8] p4
1 [f── ── u 8] p5
1 [f── ── u 8] p6
1 [f── ── u 8] c1
1 [f── ── u 8] p0
0 [*── ── ──] c4
0 [*── ── ──] c5
0 [*── ── ──] c6
0 [*── ── ──] c3
0 [*── ── ──] c0
0 [*── ── ──] c2
0 [*── ── ──] p1
1 [u 9 f── ──] p4
1 [u 9 f── ──] p5
1 [u 9 f── ──] p6
1 [u 9 f── ──] c1
0 [── *── ──] c4
0 [── *── ──] c5
0 [── *── ──] c6
0 [── *── ──] c3
0 [── *── ──] c0
0 [── *── ──] c2
1 [── uEOS f──] [main: added end-of-stream marker]
2 [f── uEOS EOS] [main: added end-of-stream marker]
3 [EOS *EOS EOS] [main: added end-of-stream marker]
3 [EOS *EOS EOS] c3
2 [EOS f── uEOS] c4
2 [EOS f── uEOS] c5
2 [EOS f── uEOS] c3
2 [EOS f── uEOS] c6
1 [uEOS f── ──] c4
1 [uEOS f── ──] c5
1 [uEOS f── ──] c6
1 [uEOS f── ──] c1
0 [── *── ──] c4
0 [── *── ──] c5
0 [── *── ──] c6
Consumer consumption:

6.Şimdi uykunun konumunu c6 olarak değiştirin (bu, tüketicinin kuyruktan bir şey çıkarmasını ve ardından onunla bir şeyler yapmasını modeller), yine tek girişli bir arabellek kullanarak. Bu durumda ne zaman tahmin edersiniz? ./ana-iki-cv-iken -p1 -c3 -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

Bir alıcı iş parçacığının, temel bölüm için çocuk oyuncağını boşalttıktan sonra iddianameyi 1 saniye askıya alacağı düşüncesi, bu nedenle tampondan alınan saygınlığın birkaç değerli işi taklit etmesi.

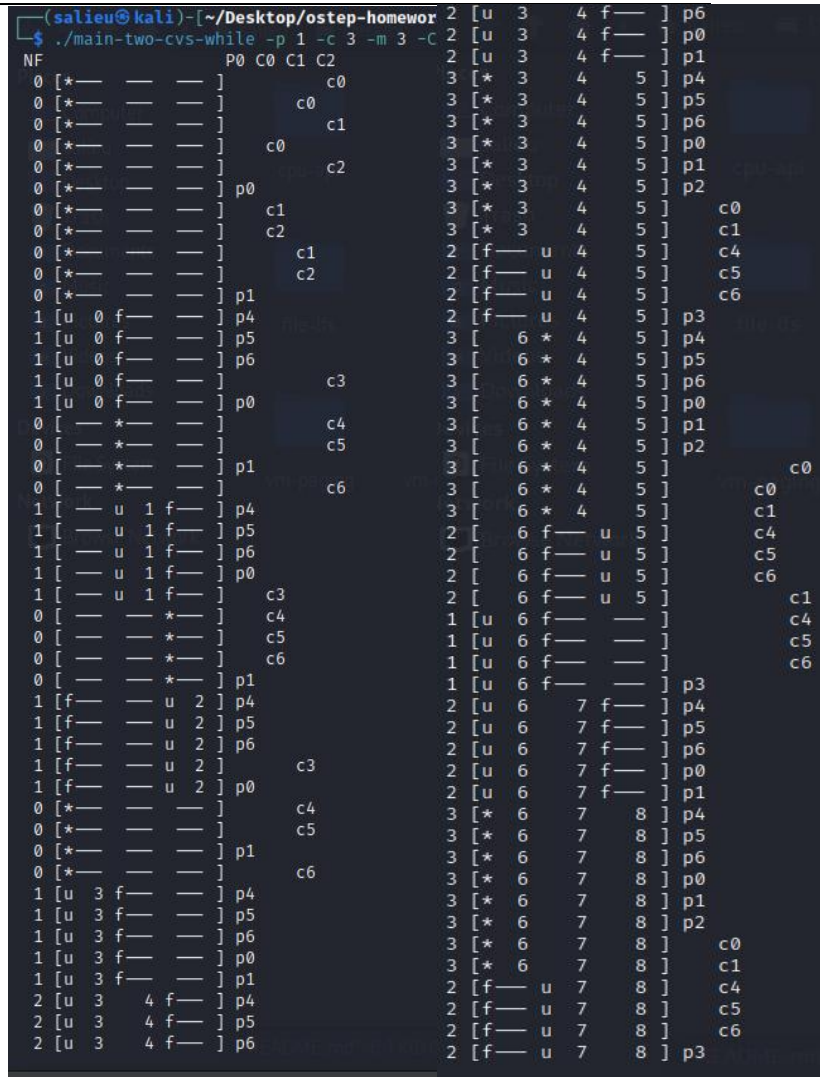
Kali linux'ta ~ 5 saniye. 10% 3'ten beri!= 0 kalan tüketiciler, bir EOS'a ulaşmak için tüketici işlevini tamamlayacak ,2 saniye daha ekleniyor.

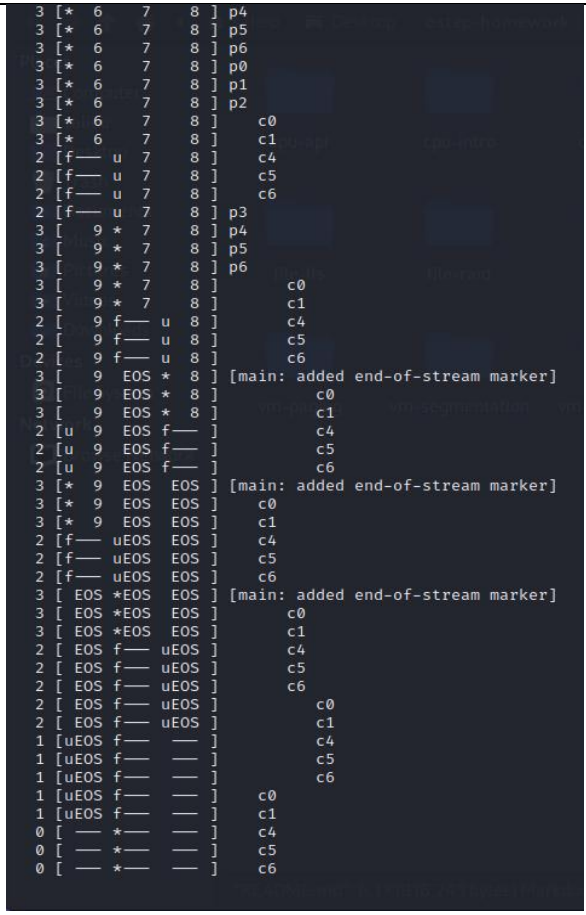

(saliou㉿kali)-[~/Desktop/ostep-homework/thr
$ ./main-two-cvs-while -p 1 -c 3 -m 1 -C 0,0,
NF P0 C0 C1 C2
0 [*—] p0
0 [*—] c0
0 [*—] c0
0 [*—] c0
0 [*—] p1
1 [* 0] p4
1 [* 0] p5
1 [* 0] p6
1 [* 0] c1
1 [* 0] p0
0 [*—] c4
0 [*—] c5
0 [*—] c6
0 [*—] c1
0 [*—] c2
0 [*—] c1
0 [*—] c2
0 [*—] p1
1 [* 1] p4
1 [* 1] p5
1 [* 1] p6
1 [* 1] c3
1 [* 1] p0
0 [*—] c4
0 [*—] c5
0 [*—] c6
0 [*—] p1
1 [* 2] p4
1 [* 2] p5
1 [* 2] p6
1 [* 2] c3
0 [*—] c4
0 [*—] p0
0 [*—] c5
0 [*—] c6
0 [*—] p1
1 [* 3] p4
1 [* 3] p5
1 [* 3] p6
1 [* 3] p0
1 [* 3] p1
1 [* 3] p2
1 [* 3] c0
1 [* 3] c1
0 [*—] c4
0 [*—] c5
0 [*—] c0
0 [*—] c6
0 [*—] p3
1 [* 4] p4
1 [* 4] p5
1 [* 4] p6
1 [* 4] c1
0 [*—] c4
0 [*—] p0
0 [*—] c5
0 [*—] c6
0 [*—] p1
1 [* 5] p4
1 [* 5] p5
1 [* 5] p6
1 [* 5] c0
1 [* 5] p0
1 [* 5] p1
1 [* 5] p2
1 [* 5] c1
0 [*—] c4
0 [*—] c5
0 [*—] c6
0 [*—] p3
1 [* 6] p4
1 [* 6] p5
1 [* 6] p6
1 [* 6] p0
1 [* 6] p1
1 [* 6] p2
1 [* 6] c0
1 [* 6] c1
0 [*—] c4
0 [*—] c5
0 [*—] c0
0 [*—] c6
0 [*—] p3
1 [* 7] p4
1 [* 7] p5
1 [* 7] p6
1 [* 7] p0
1 [* 7] c1
0 [*—] c4
0 [*—] c5
0 [*—] c0
0 [*—] c6
0 [*—] p1



7.Son olarak, arabellek boyutunu tekrar 3 olarak değiştirin (-m 3). Nasıl olacağını saat Kaçta tahmin ediyorsun?

Üretici tüketicinin önünde olacak, ancak buradaki darboğaz hala devam ediyor tüketici. Dolayısıyla beklenen yürütme süresi yukarıdakilere benzer.


(salieu㉿kali)-[~/Desktop/ostep-homewor
$ ./main-two-cvs-while -p 1 -c 3 -m 3 -C
NF P0 C0 C1 C2



8.Şimdi main-one-cv-while.c. Bu kodla ilgili bir soruna neden olması için tek bir üretici, bir tüketici ve 1 boyutunda bir arabellek varsayarak bir uyku dizesi yapılandırabilir misiniz?

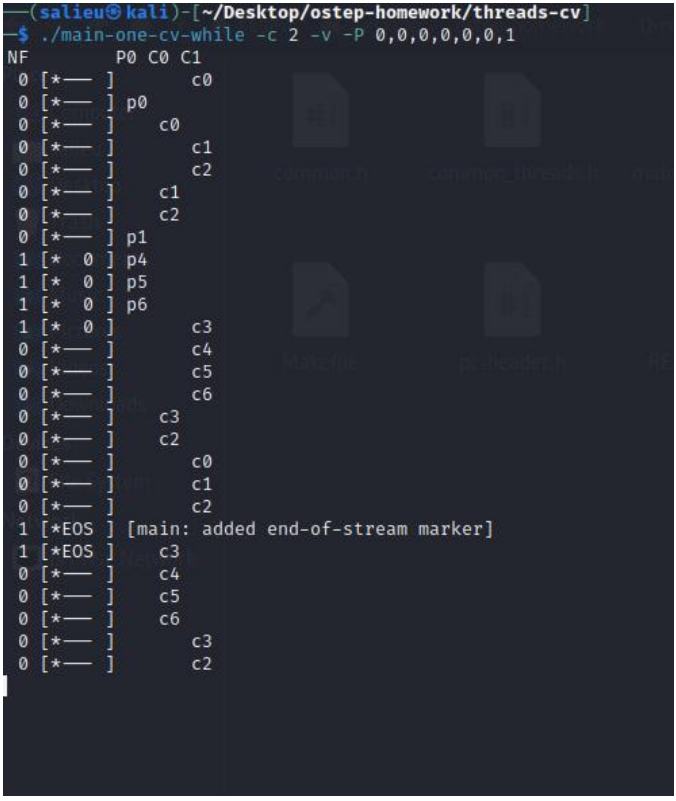
Bu durumda, felakete yol açacak bir uyku zinciri çalıştırabileceğimizi sanmıyorum. Tüketiciler uyanırsa felaketi heceleyebilecek tek bir özgeçmiş tüketici (veya üreticiyi uyandıran üretici), ancak yalnızca iki iş parçacığımız olduğu için Yanlış bir uyanış olamaz. yani bu durumda yapımcı sadece birini uyandırabilir tüketiciler ve tam tersi.

.

9. Şimdi tüketici sayısını ikiye değiştirin. Kodda bir soruna neden olacak şekilde üretici ve tüketiciler için uyku dizeleri oluşturabilir misiniz?

Evet ./main-one-cv-while -c 2 -v -P 0,0,0,0,0,0,1.

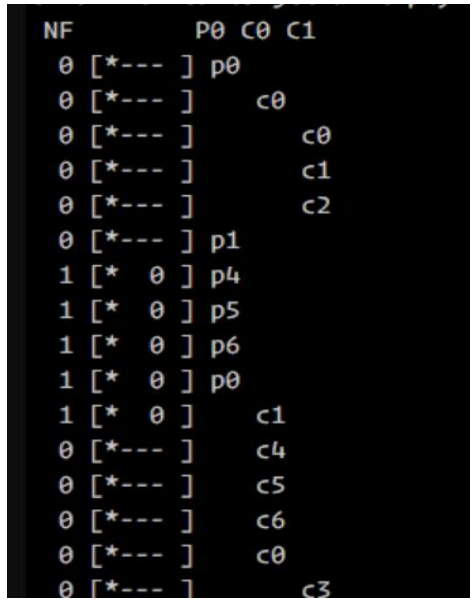
Bu bunu yapmanın en kolay yolu, üreticinin cv'yi beklemediğinden emin olmaktır bir tüketici diğerini arar.



10.Şimdi main-two-cvs-ıf'yi inceleyin.c. Bu kodda bir sorunun oluşmasına neden olabilir misiniz? Yine, yalnızca bir tüketicinin olduğu durumu ve ardından birden fazla tüketicinin olduğu durumu düşünün.

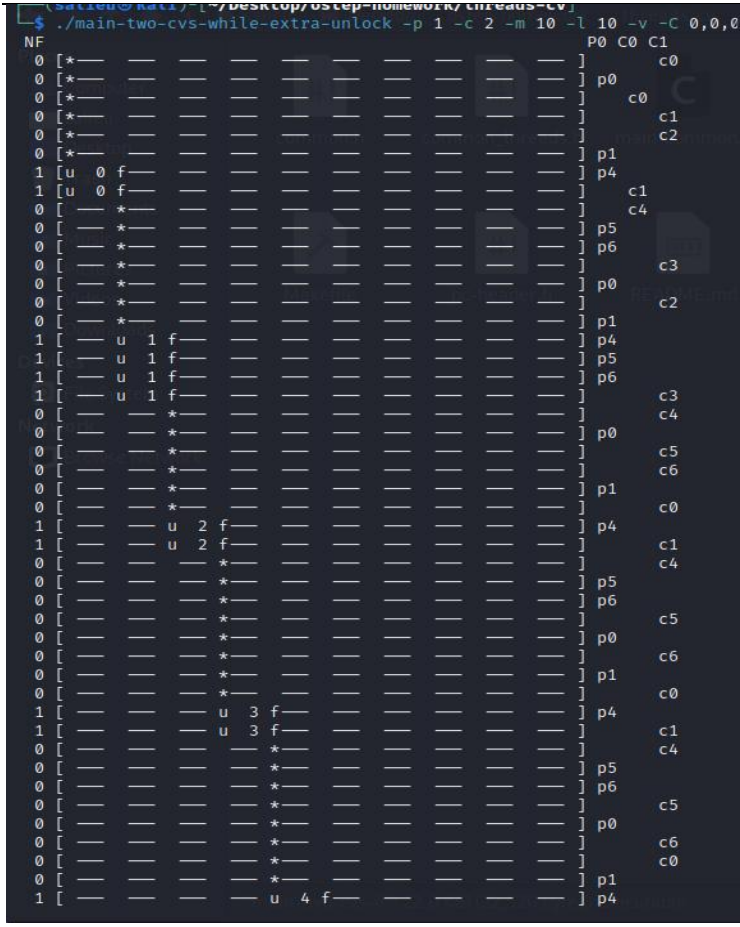
Her şeyden önce, bir üreticiyi ve bir tüketiciyi adil hale getirdiğimizden, bu kod işe yarıyor.

İki tüketici ile: ./ ana-iki-özgeçmiş-eğer -p 1 -c 2 -m 1 -l 2 -P 1 -C 1:0,0,0,1 -v / less, c1'in 1. yürütmesine izin vererek boş bir arabellek hatası üretir bir sinyal bekler.



11.Son olarak, ana-iki-cvs-while-extra-kilidini inceleyin.c. Put veya get yapmadan önce kilidi serbest bıraktığınızda hangi sorun ortaya çıkar? Uyku ipleri göz önüne alındığında, böyle bir sorunun güvenilir bir şekilde ortaya çıkmasına neden olabilir misiniz? Ne kötü şey olabilir ki?

İç mekanın alıp koyduğu temel alanlar için ortak kaçınmayı ortadan kaldırıyoruz. özellikle, istemciler veya oluşturucular yürütülürken paylaşılan bellek arabellek, fill_ptr, use_ptr ve num_full için bir yarış koşulu vardır.


./main-two-cvs-while-extra-unlock -p 1 -c 2 -m 10 -l 10 -v -C 0,0,0
NF
P0 C0 C1