31

Semaphores

Şimdi bildiğimiz gibi, çok çeşitli ilgili ve ilginç eş zamanlılık problemlerini çözmek için hem kilitlere hem de koşul değişkenlerine ihtiyaç vardır. Bunu yıllar önce fark eden ilk insanlardan biri, **Edsger Dijkstra (kesin tarihi bilmek zor olsa da** [GR92]), diğer şeylerin yanı sıra çizge teorisindeki ünlü "shortest paths" algoritmasıyla bilinir [D59], başlıklı yapılandırılmış programlama üzerine erken bir polemik "Zararlı Görülen Goto İfadeleri" [D68a] (ne harika bir başlık!), ve durumda burada çalışacağız, semafor adı verilen bir senkronizasyon ilkesinin tanıtılması [D68b, D72]. Aslında, Dijkstra ve meslektaşları, senkronizasyonla ilgili her şey için tek bir ilkel olarak semaforu icat ettiler; göreceğiniz gibi, semaforlar hem kilitler hem de koşul değişkenleri olarak kullanılabilir.

THE CRUX: Semafor Nasıl Kullanılır?

Kilitler ve koşul değişkenleri yerine semaforları nasıl kullanabiliriz? Bir semaforun tanımı nedir? İkili semafor nedir? Kilitlerden ve koşul değişkenlerinden bir semafor oluşturmak kolay mı? Semaforlardan kilitler ve koşul değişkenleri oluşturmak için mi?

31.1 Semafor: Tanım

Semafor, iki rutinle işleyebileceğimiz bir tamsayı değerine sahip bir nesnedir; POSIX standardında bu rutinler sem_wait() ve sem_post()1'dir. Semaforun başlangıç değeri davranışını belirlediğinden, semaforla etkileşim için başka bir rutini çağırmadan önce, Şekil 'deki kod gibi, onu bir değerle başlatmamız gerekir.

31.1 yapar.

1Historically, sem wait() was called P() by Dijkstra and sem post() called V(). These shortened forms come from Dutch words; interestingly, which Dutch words they supposedly derive from has changed over time. Originally, P() came from "passering" (to pass) and V() from "vrijgave" (release); later, Dijkstra wrote P() was from "prolaag", a contraction of "probeer" (Dutch for "try") and "verlaag" ("decrease"), and V() from "verhoog" which means "increase". Sometimes, people call them down and up. Use the Dutch versions to impress your friends, or confuse them, or both. See https://news.ycombinator.com/item?id=8761539) for details.

```
1 #include <semaphore.h>
2 sem_t s;
3 sem_init(&s, 0, 1);
```

Figure 31.1: **Bir Semaforu Başlatma**

Şekilde, bir s semaforu ilan ediyoruz ve üçüncü argüman olarak 1 ileterek onu 1 değerine başlatıyoruz. Sem_init() için ikinci argüman, göreceğimiz tüm örneklerde 0'a ayarlanacaktır bu, semaforun aynı işlemdeki iş parçacıkları arasında paylaşıldığını gösterir. İkinci argüman için farklı bir değer gerektiren semaforların diğer kullanımları (başka bir deyişle, farklı işlemler arasında erişimi senkronize etmek için nasıl kullanılabilecekleri hakkında ayrıntılar için kılavuz sayfasına bakın.

Bir semafor başlatıldıktan sonra, onunla etkileşim kurmak için iki işlevden birini çağırabiliriz, sem_wait () veya sem_post(). Bu iki fonksiyonun davranışı Şekil 31.2'de görülmektedir.

Şimdilik, açıkçası biraz özen gerektiren bu rutinlerin uygulanmasıyla ilgilenmiyoruz, sem_wait () ve sem_post()'u çağıran birden fazla iş parçacığı ile, bu kritik bölümlerin yönetilmesine açık bir ihtiyaç vardır. Şimdi bu ilkellerin nasıl kullanılacağına odaklanacağız, daha sonra nasıl inşa edildiğini tartışabiliriz.

Arayüzlerin birkaç göze çarpan yönünü burada tartışmalıyız. İlk olarak, sem_wait()'in ya hemen döneceğini görebiliriz (çünkü sem_wait()'i çağırdığımızda semaforun değeri bir ya da daha yüksekti) ya da arayanın bir sonraki gönderiyi beklerken yürütmeyi askıya almasına neden olur. Elbette, birden çok çağrı dizisi sem_wait()'i çağırabilir ve bu nedenle tümü uyandırılmayı beklerken sıraya alınabilir.

İkinci olarak, sem_post()'un sem_wait()'in yaptığı gibi belirli bir koşulun gerçekleşmesini beklemediğini görebiliriz. Bunun yerine, sadece semaforun değerini arttırır ve sonra, uyandırılmayı bekleyen bir iş parçacığı varsa, bunlardan birini uyandırır.

Üçüncüsü, semaforun değeri, negatif olduğunda, bekleyen iş parçacıklarının sayısına [D68b] eşittir. Değer genellikle semafor kullanıcıları tarafından görülmese de, bu değişmez. Bilmeye değer ve belki de bir semaforun nasıl çalıştığını hatırlamanıza yardımcı olabilir…

```
1 int sem_wait(sem_t *s) {
2     decrement the value of semaphore s by one
3     wait if value of semaphore s is negative
4 }
5
6 int sem_post(sem_t *s) {
7     increment the value of semaphore s by one
8     if there are one or more threads waiting, wake one
9 }
```

Figure 31.2: **Semafor: Bekleme ve Gönderme Tanımları**

```
1 sem_t m;
2 sem_init(&m, 0, X); // initialize to X; what should X be?
3
4 sem_wait(&m);
5 // critical section here
6 sem_post(&m);
```

Figure 31.3: **İkili Semafor (Yani, Bir Kilit)**

Semafor içinde olası görünen yarış koşulları hakkında (henüz) endişelenmeyin; yaptıkları eylemlerin atomik olarak gerçekleştirildiğini varsayın. Yakında bunu yapmak için kilitleri ve koşul değişkenlerini kullanacağız.

31.1 İkili Semaforlar (Kilitler)

Artık bir semafor kullanmaya hazırız. İlk kullanımımız zaten aşina olduğumuz bir kullanım olacak: bir semaforu kilit olarak kullanmak. Bkz. Şekil 31.3 kod parçacığı için orada, ilgilenilen kritik bölümü basitçe bir sem wait() /sem post() çifti ile çevrelediğimizi göreceksiniz. Bununla birlikte, bu işi yapmak için kritik olan, m semaforunun başlangıç değeridir (şekilde X ile başlatılmıştır). X ne olmalı?

... (Devam etmeden önce bir düşünün) ...

Yukarıdaki sem wait() ve sem post() rutinlerinin tanımlarına baktığımızda, başlangıç değerinin 1 olması gerektiğini görebiliriz.

Bunu açıklığa kavuşturmak için, iki iş parçacığına sahip bir senaryo hayal edelim. İlk iş parçacığı (İş parçacığı 0) sem wait()'i çağırır; önce semaforun değerini azaltarak 0'a değiştirir. Ardından, yalnızca değer 0'dan büyük veya 0'a eşit değilse bekler. Değer 0 olduğundan, sem wait() basitçe geri döner ve çağıran iş parçacığı devam edecek; 0 parçacığı artık kritik bölüme girmek için serbesttir. İş parçacığı 0 kritik bölümün içindeyken başka hiçbir iş parçacığı kilidi almaya çalışmazsa, sem_post()'u çağırdığında, semaforun değerini 1'e geri yükler (ve bekleyen bir iş parçacığını uyandırmaz, çünkü hiçbiri yoktur) . Şekil 31.4, bu senaryonun bir izini göstermektedir. Daha ilginç bir durum, Thread0 "kilidi tuttuğunda" (yani, sem wait ()'i çağırdı ama henüz sem post()'u çağırmadı ve başka bir thread (Thread1) sem_wait()'i çağırarak kritik bölüme girmeye çalışıyor. Bu durumda, Thread1 semaforun değerini -1'e düşürür ve

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	
0	`(crit sect)`	
0	call `sem_post()`	
1	`sem_post()` returns	

Figure 31.4: **İş Parçacığı İzleme: Semafor Kullanan Tek İş Parçacığı**

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` returns	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` returns	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` returns	Run

Figure 31.5: **İplik İzleme: Bir Semafor Kullanarak İki İş Parçacığı**

bu nedenle bekler (kendini uykuya sokar ve işlemciyi bırakır). Thread0 tekrar çalıştığında, sonunda sem post()'u çağırır, semaforun değerini tekrar sıfıra yükseltir ve ardından, kilidi kendisi için alabilecek olan bekleyen thread'i uyandırır (Thread1). Thread1 bittiğinde, semaforun değerini tekrar artıracak ve tekrar 1'e geri yükleyecektir.

Figure31.5, bu örneğin bir izini göstermektedir. İş parçacığı eylemlerine ek olarak, şekil her bir iş parçacığının zamanlayıcı durumunu gösterir: Çalıştır (iş parçacığı çalışıyor), Hazır (yani çalıştırılabilir ancak çalışmıyor) ve Uyku (iş parçacığı bloke). Halihazırda tutulan kilidi almaya çalıştığında, Thread1'in uyku durumuna geçtiğine dikkat edin; sadece Thread0 tekrar çalıştığında Thread1 uyandırılabilir ve potansiyel olarak tekrar çalışabilir.

Kendi örneğiniz üzerinde çalışmak istiyorsanız, birden çok iş parçacığının sıraya girip kilitlenmeyi beklediği bir senaryo deneyin. Böyle bir iz sırasında semaforun değeri ne olurdu?

Böylece semaforları kilit olarak kullanabiliriz. Kilitlerin yalnızca iki durumu olduğundan (tutulan ve tutulmayan), bazen kilit olarak kullanılan bir semaforu ikili semafor olarak adlandırırız. Bir semaforu yalnızca bu ikili biçimde kullanıyorsanız, burada sunduğumuz genelleştirilmiş semaforlardan daha basit bir şekilde uygulanabileceğini unutmayın.

31.3 Sıralama İçin Semaforlar

Semaforlar, eşzamanlı bir programdaki olayları sıralamak için de yararlıdır. Örneğin, bir iş parçacığı, bir listenin boş olmamasını beklemek isteyebilir.,

```
1  sem_t s;
2
3  void *child(void *arg) {
4      printf("child\n");
5      sem_post(&s); // signal here: child is done
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     sem_init(&s, 0, X); // what should X be?
11     printf("parent: begin\n");
12     pthread_t c;
13     Pthread_create(&c, NULL, child, NULL);
14     sem_wait(&s); // wait here for child
15     printf("parent: end\n");
16     return 0;
17 }
```

Figure 31.6: **Çocuğunu Bekleyen Bir Ebeveyn**

Böylece ondan bir öğe silebilir. Bu kullanım modelinde, genellikle bir şeyin olmasını bekleyen bir iş parçacığı ve başka bir iş parçacığının bir şeyin olmasını sağlayan ve ardından bunun olduğunu bildiren, böylece bekleyen iş parçacığını uyandıran bir iş parçacığı buluruz. Bu nedenle semaforu bir sıralama ilkeli olarak kullanıyoruz (daha önce koşul değişkenlerini kullanmamıza benzer şekilde).

Basit bir örnek aşağıdaki gibidir. Bir iş parçacığının başka bir iş parçacığı oluşturduğunu ve ardından yürütmesini tamamlamasını beklemek istediğini hayal edin (Şekil 31.6). Bu program çalıştığında, aşağıdakileri görmek isteriz.

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

O halde soru, bu etkiyi elde etmek için bir semaforun nasıl kullanılacağıdır; Görünüşe göre, cevabı anlamak nispeten kolaydır. Kodda görebileceğiniz gibi, Ebeveyn basitçe sem wait()'i ve Çocuk sem post()'u çağırarak yürütmesini bitirme koşulunun gerçekleşmesini bekler. Ancak bu şu soruyu gündeme getiriyor: Bu semaforun başlangıç değeri ne olmalıdır?

(Yine, ileriyi okumak yerine burada düşünün)

Cevap, elbette, semaforun değerinin 0 olarak ayarlanması gerektiğidir. Dikkate alınması gereken iki durum vardır. İlk olarak, çocuğu ebeveynin yarattığını, ancak çocuğun henüz koşmadığını (yani, hazır bir kuyrukta oturuyor ama koşmuyor) varsayalım. Bu durumda (Şekil 31.7, sayfa 6), çocuk sem_post()'u çağırmadan önce ebeveyn sem_wait()'i arayacaktır; ebeveynin çocuğunun koşmasını beklemesini istiyoruz. Bunun olmasının tek yolu, semaforun değerinin 0'dan büyük olmamasıdır; dolayısıyla 0 başlangıç değeridir. Ebeveyn çalışır, semaforu azaltır (-1'e), sonra bekler (uyur). Çocuk nihayet çalıştığında, sem_post()'u çağırır, değeri arttırır.

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists, can run)*	Ready
0	call `sem_wait()`	Run		Ready
-1	`decr sem`	Run		Ready
-1	`(sem<0)→sleep`	Sleep		Ready
-1	*Switch→Child*	Sleep	`child runs`	Run
-1		Sleep	call `sem_post()`	Run
0		Sleep	`inc sem`	Run
0		Ready	`wake(Parent)`	Run
0		Ready	`sem_post()` returns	Run
0		Ready	*Interrupt→Parent*	Ready
0	`sem_wait()` returns	Run		Ready

Figure 31.7: **İş Parçacığı İzleme: Çocuğunu Bekleyen Ebeveyn (Durum 1)**

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists; can run)*	Ready
0	*Interrupt→Child*	Ready	`child runs`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`inc sem`	Run
1		Ready	`wake(nobody)`	Run
1		Ready	`sem_post()` returns	Run
1	`parent runs`	Run	*Interrupt→Parent*	Ready
1	call `sem_wait()`	Run		Ready
0	`decrement sem`	Run		Ready
0	`(sem≥0)→awake`	Run		Ready
0	`sem_wait()` returns	Run		Ready

Figure 31.8: **İş Parçacığı İzleme: Çocuğunu Bekleyen Ebeveyn (Durum 2)**

Semaforu 0'a getirin ve programı bitirerek ebeveyni uyandırın.

İkinci durum (Şekil 31.8), ebeveyn sem_wait()'i çağırma şansı bulamadan çocuk tanımlamaya çalıştığında ortaya çıkar. Bu durumda, çocuk önce sem_post()'u çağırır, böylece semaforun değeri 0'dan 1'e çıkar. 1 olması ebeveynin değerini bu nedenle (0'a) düşürür ve beklemeden sem_wait() işlevinden döner, ayrıca istenen efekti elde eder.

31.4 Üretici/Tüketici (Sınırlı Tampon) Sorunu

Bu bölümde karşılaşacağımız bir sonraki problem, üretici/tüketici problemi veya bazen sınırlı tampon problemi [D72] olarak bilinir. Bu problem, koşul değişkenleri ile ilgili önceki bölümde ayrıntılı olarak açıklanmıştır; ayrıntılar için oraya bakın

ASIDE: BİR SEMAFORUN DEĞERİNİ BELİRLEME

Şimdi bir semafor başlatmanın iki örneğini gördük. İlk durumda, semaforu kilit olarak kullanmak için değeri 1 olarak ayarladık; ikincisinde, sipariş için semaforu kullanmak için 0'a. Peki semafor başlatma için genel kural nedir?

Perry Kivolowitz sayesinde bunu düşünmenin basit bir yolu, başlatmadan hemen sonra vermek istediğiniz kaynak sayısını düşünmektir. Kilit ile 1'di, çünkü başlatmadan hemen sonra kilidin kilitlenmesini (verilmesini) istiyorsunuz. Sipariş durumunda 0'dı, çünkü başlangıçta verilecek bir şey yok; yalnızca alt iş parçacığı tamamlandığında oluşturulan kaynaktır, bu noktada değer 1'e yükseltilir. Gelecekteki semafor sorunları üzerinde bu düşünceyi deneyin ve yardımcı olup olmadığına bakın...

İlk girişim

Sorunu çözmeye yönelik ilk girişimimiz, boş ve dolu olmak üzere iki semaforu tanıtıyor ve bu semaforlar, sırasıyla bir arabellek girişinin ne zaman boşaltıldığını veya dolduğunu belirtmek için iş parçacıklarını kullanıyor. Put ve get rutinlerinin kodu Şekil 31.9'da ve üretici ve tüketici problemini çözme girişimimiz Şekil 31.10'da (sayfa 8).

Bu örnekte, üretici önce veriyi içine koymak için bir arabelleğin boşalmasını bekler ve tüketici de benzer şekilde kullanmadan önce arabelleğin dolmasını bekler. Önce MAX=1 olduğunu (dizide yalnızca bir arabellek var) düşünelim ve bunun işe yarayıp yaramadığını görelim.

Bir üretici ve bir tüketici olmak üzere iki iş parçacığı olduğunu tekrar hayal edin. Tek bir CPU üzerinde belirli bir senaryoyu inceleyelim. Önce tüketicinin koşacağını varsayalım. Böylece, tüketici Şekil 31.10'da Satır C1'e basarak sem_wait(&full)'u çağıracaktır. full 0 değerine başlatıldığından,

```
1  int buffer[MAX];
2  int fill = 0;
3  int use  = 0;
4
5  void put(int value) {
6      buffer[fill] = value;    // Line F1
7      fill = (fill + 1) % MAX; // Line F2
8  }
9
10 int get() {
11     int tmp = buffer[use];   // Line G1
12     use = (use + 1) % MAX;   // Line G2
13     return tmp;
14 }
```

Figure 31.9: **Koy Ve Al Rutinleri**

```
1  sem_t empty;
2  sem_t full;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          sem_wait(&empty);       // Line P1
8          put(i);                 // Line P2
9          sem_post(&full);        // Line P3
10     }
11 }
12
13 void *consumer(void *arg) {
14     int tmp = 0;
15     while (tmp != -1) {
16         sem_wait(&full);        // Line C1
17         tmp = get();            // Line C2
18         sem_post(&empty);       // Line C3
19         printf("%d\n", tmp);
20     }
21 }
22
23 int main(int argc, char *argv[]) {
24     // ...
25     sem_init(&empty, 0, MAX); // MAX are empty
26     sem_init(&full, 0, 0);    // 0 are full
27     // ...
28 }
```

Figure 31.10: **Dolu ve Boş Koşulların Eklenmesi**

çağrı, dolu (-1'e) azaltır, tüketiciyi engeller ve istendiği gibi başka bir iş parçacığının sem post()'u tam olarak aramasını bekler.

Üreticinin daha sonra çalıştığını varsayalım. Satır P1'e çarpacak ve böylece sem_wait(&empty) rutinini çağıracaktır. Tüketiciden farklı olarak, üretici bu satırdan devam edecektir, çünkü boş MAX değerine başlatılmıştır (bu durumda, 1). Böylece boş, 0'a düşürülecek ve üretici, tamponun ilk girişine (Satır P2) bir veri değeri koyacaktır. Üretici daha sonra P3'e devam edecek ve tam semaforun değerini -1'den 0'a değiştirerek ve tüketiciyi uyandırarak sem post(&full)'u çağıracaktır (örneğin, engellenmiş durumundan hazır konumuna taşıyın).

Bu durumda, iki şeyden biri olabilir. Yapımcı çalışmaya devam ederse, kendi etrafında döner ve tekrar Line P1'e basar. Ancak bu sefer, boş semaforun değeri 0 olduğu için bloke eder. Bunun yerine üretici kesintiye uğrarsa ve tüketici çalışmaya başlarsa, sem_wait(&full)'den (Satır C1) döner, arabelleğin dolduğunu bulur ve tüketin. Her iki durumda da istenen davranışı elde ederiz.

Aynı örneği daha fazla iş parçacığıyla deneyebilirsiniz (örneğin, birden çok üretici ve birden çok tüketici). Hala çalışıyor olmalı.

```
1  void *producer(void *arg) {
2      int i;
3      for (i = 0; i < loops; i++) {
4          sem_wait(&mutex);       // Line P0 (NEW LINE)
5          sem_wait(&empty);       // Line P1
6          put(i);                 // Line P2
7          sem_post(&full);        // Line P3
8          sem_post(&mutex);       // Line P4 (NEW LINE)
9      }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&mutex);       // Line C0 (NEW LINE)
16         sem_wait(&full);        // Line C1
17         int tmp = get();        // Line C2
18         sem_post(&empty);       // Line C3
19         sem_post(&mutex);       // Line C4 (NEW LINE)
20         printf("%d\n", tmp);
21     }
22 }
```

Figure 31.11: **Karşılıklı Dışlama Ekleme (Yanlış)**

Şimdi MAX'ın 1'den büyük olduğunu düşünelim (örneğin MAX=10). Bu örnek için, birden çok üretici ve birden çok tüketici olduğunu varsayalım. Şimdi bir sorunumuz var: bir yarış durumu. Nerede oluştuğunu görüyor musunuz? (Biraz zaman ayırın ve arayın) Göremiyorsanız, işte bir ipucu put() ve get() koduna daha yakından bakın.

Tamam, sorunu anlayalım. İki üreticinin (Pa ve Pb) kabaca aynı anda put()'u çağırdığını hayal edin. İlk önce üretici Pa'nın çalıştığını ve ilk arabellek girişini doldurmaya başladığını varsayalım (Satır F1'de fill=0). Pa, doldurma sayacını 1'e yükseltme şansı bulamadan önce kesintiye uğrar. Üretici Pb çalışmaya başlar ve F1 Satırında verilerini de tamponun 0. elemanına koyar, bu da oradaki eski verilerin üzerine yazıldığı anlamına gelir! Bu eylem hayırdır; üreticiden herhangi bir verinin kaybolmasını istemiyoruz.

Bir Çözüm: Karşılıklı Dışlama Ekleme

Gördüğünüz gibi, burada unuttuğumuz şey *karşılıklı dışlamadır*. Bir tamponun doldurulması ve indeksin tampona arttırılması kritik bir bölümdür ve bu nedenle dikkatli bir şekilde korunmalıdır. Öyleyse arkadaşımız ikili semaforu kullanalım ve bazı kilitler ekleyelim. Şekil 31.11 girişimimizi göstermektedir.

Şimdi, YENİ SATIR yorumlarında belirtildiği gibi, kodun tüm put() / get() bölümlerinin etrafına bazı kilitler ekledik. Bu doğru fikir gibi görünüyor, ama aynı zamanda işe yaramıyor. Neden? Çıkmaz. Kilitlenme neden oluşur? Bunu düşünmek için bir dakikanızı ayırın; kilitlenmenin ortaya çıktığı bir durum bulmaya çalışın. Programın kilitlenmesi için hangi adım dizisinin gerçekleşmesi gerekir?

```
1  void *producer(void *arg) {
2      int i;
3      for (i = 0; i < loops; i++) {
4          sem_wait(&empty);       // Line P1
5          sem_wait(&mutex);       // Line P1.5 (MUTEX HERE)
6          put(i);                 // Line P2
7          sem_post(&mutex);       // Line P2.5 (AND HERE)
8          sem_post(&full);        // Line P3
9      }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&full);        // Line C1
16         sem_wait(&mutex);       // Line C1.5 (MUTEX HERE)
17         int tmp = get();        // Line C2
18         sem_post(&mutex);       // Line C2.5 (AND HERE)
19         sem_post(&empty);       // Line C3
20         printf("%d\n", tmp);
21     }
22 }
```

Figure 31.12: **Karşılıklı Dışlama Ekleme (Doğru)**

Kilitlenmeden Kaçınmak

Tamam, şimdi bunu anladığınıza göre, işte cevap. İki iplik düşünün, biri üretici diğeri tüketici. Tüketici önce koşar. Mutex'i (Satır C0) alır ve ardından tam semaforda (C1 Hattı) sem_wait() çağırır. Henüz veri olmadığından, bu çağrı tüketicinin CPU'yu engellemesine ve dolayısıyla vermesine neden olur. Daha da önemlisi, tüketici hala kilidi elinde tutuyor.

Daha sonra bir yapımcı çalışır. Üretecek verileri var ve eğer çalışabilseydi, tüketici ipliğini uyandırabilir ve her şey iyi olurdu. Neyse ki, yaptığı ilk şey, ikili muteks semaforunda (Satır P0) sem_wait() olarak adlandırmaktır. Kilit zaten tutuluyor. Bu nedenle, üretici de artık beklemeye sıkışmış durumda.

Burada basit bir döngü var. Tüketici muteksi *tutar* ve birinin dolu sinyal vermesini bekler. Yapımcı tam *sinyal* verebilir ama mutex'i bekliyor. Böylece, üretici ve tüketicinin her biri birbirini beklemek zorunda kalıyor. Klasik bir çıkmaz.

Sonunda, Çalışan Bir Çözüm

Bu sorunu çözmek için, kilidin kapsamını azaltmalıyız. Şekil 31.12 (sayfa 10) doğru çözümü göstermektedir. Gördüğünüz gibi, muteks edinme ve serbest bırakma işlemlerini kritik bölümün hemen etrafında olacak şekilde hareket ettiriyoruz;

dolu ve boş bekleme ve sinyal kodu dışında bırakılır. Sonuç, çok iş parçacıklı programlarda yaygın olarak kullanılan bir desen olan basit ve çalışan sınırlı bir arabellektir. Şimdi anlayın; daha sonra kullanın. Gelecek yıllar için bize teşekkür edeceksiniz. Ya da en azından, final sınavında veya bir iş görüşmesi sırasında aynı soru sorulduğunda bize teşekkür edeceksiniz.

31.5 Okuyucu-Yazar Kilitleri

Bir başka klasik sorun, farklı veri yapısı erişimlerinin farklı kilitleme türleri gerektirebileceğini kabul eden daha esnek bir kilitleme ilkesi arzusundan kaynaklanmaktadır. Örneğin, eklemeler ve basit aramalar dahil olmak üzere bir dizi eşzamanlı liste işlemi düşünün. In-sert'ler listenin durumunu değiştirirken (ve dolayısıyla geleneksel mantıklı bir kritik bölüm), aramalar sadece veri yapısını *okur*; Hiçbir eklemenin devam etmediğini garanti edebildiğimiz sürece, birçok arama aynı anda vazgeçti. Şimdi bu tür bir işlemi desteklemek için geliştireceğimiz özel kilit türü, **okuyucu-yazar kilidi** olarak bilinir [CHP71]. Böyle bir kilidin kodu Şekil 31.13'te (sayfa 12) mevcuttur.

Kod oldukça basittir. Bazı iş parçacığı söz konusu veri yapısını güncelleştirmek istiyorsa, yeni eşitleme operation çiftini çağırmalıdır: bir yazma kilidi elde etmek için rwlock_acquire_writelock(), ve rwlock_release_writelock(), bunu serbest bırakmak için dahili olarak, bunlar yalnızca tek bir yazarın kilidi sorgulayabilmesini sağlamak için yazma kilidi semaforunu kullanır ve böylece söz konusu veri yapısını güncellemek için kritik bölüme girer.

Daha ilginç olanı, okuma kilitlerini elde etmek ve serbest bırakmak için bir çift rutindir. Bir okuma kilidi alırken, okuyucu önce kilidi alır ve ardından veri yapısında şu anda kaç okuyucu olduğunu izlemek için okuyucular değişkenini artırır. Daha sonra rwlock_acquire_readlock() içinde atılan önemli adım, ilk okuyucu kilidi aldığında gerçekleşir; bu durumda, okuyucu ayrıca writelock semaforunda sem wait() öğesini çağırarak yazma kilidini edinir ve ardından sem post() öğesini çağırarak kilidi serbest bırakır.

Böylece, bir okuyucu bir okuma kilidi edindiğinde, daha fazla okuyucunun da okuma kilidini almasına izin verilecektir; ancak, yazma kilidini almak isteyen herhangi bir konu, *tüm* okuyucular bitene kadar beklemek zorunda kalacaktır kritik bölümden çıkan son bölüm "writelock" üzerine sem post() adını verir ve böylece bekleyen bir yazarın kilidi almasını sağlar.

Bu yaklaşım (istenildiği gibi) işe yarar, ancak özellikle adalet söz konusu olduğunda bazı olumsuzlukları vardır. Özellikle, okuyucuların yazarları aç bırakması nispeten kolay olacaktır. Bu soruna daha sofistike çözümler mevcuttur; belki daha iyi bir uygulama düşünebilirsiniz? İpucu: Bir yazar bekledikten sonra daha fazla okuyucunun kilide girmesini önlemek için ne yapmanız gerektiğini düşünün.

```
1  typedef struct _rwlock_t {
2    sem_t lock;      // binary semaphore (basic lock)
3    sem_t writelock; // allow ONE writer/MANY readers
4    int   readers;   // #readers in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8    rw->readers = 0;
9    sem_init(&rw->lock, 0, 1);
10   sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14   sem_wait(&rw->lock);
15   rw->readers++;
16   if (rw->readers == 1) // first reader gets writelock
17     sem_wait(&rw->writelock);
18   sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22   sem_wait(&rw->lock);
23   rw->readers--;
24   if (rw->readers == 0) // last reader lets it go
25     sem_post(&rw->writelock);
26   sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30   sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34   sem_post(&rw->writelock);
35 }
```

Figure 31.13: **Basit** Bir **Okuyucu-Yazar Kilidi**

Son olarak, okuyucu-yazar kilitlerinin biraz dikkatli kullanılması gerektiğine dikkat edilmelidir. Genellikle daha fazla ek yük eklerler (özellikle daha karmaşık uygulamalarla) ve bu nedenle sadece basit ve hızlı kilitleme ilkellerini kullanmaya kıyasla performansı hızlandırmazlar [CB08]. Her iki durumda da semaforları ilginç ve kullanışlı bir şekilde nasıl kullanabileceğimizi bir kez daha gösteriyorlar.

TIP: BASİT HER ZAMAN DAHA İYİDİR (H ILL'S YASASI)

Basit ve saçma yaklaşımın en iyisi olabileceği fikrini asla küçümsememelisiniz. Kilitleme ile, bazen basit bir döndürme kilidi en iyi sonucu verir, çünkü uygulanması kolay ve hızlıdır. Okuyucu / yazar kilitleri gibi bir şey kulağa hoş gelse de karmaşıktır ve karmaşık yavaş anlamına gelebilir. Bu nedenle, her zaman önce basit ve saçma yaklaşımı deneyin.

Sadeliğe hitap eden bu fikir birçok yerde bulunur. Erken kaynaklardan biri, Mark Hill'in CPU'lar için önbelleklerin nasıl tasarlanacağını inceleyen tezidir [H87]. Hill, basit doğrudan eşlenmiş önbelleklerin süslü küme ilişkisel tasarımlardan daha iyi çalıştığını buldu (bunun bir nedeni, önbelleğe almada daha basit tasarımların daha hızlı aramalara olanak sağlamasıdır). Hill'in çalışmalarını kısaca özetlediği gibi: "Büyük ve saçma daha iyidir." Ve bu nedenle bu benzer tavsiyeye **Hill's yasası** diyoruz.

31.6 Yemek Filozofları

Dijkstra tarafından ortaya atılan ve çözülen en ünlü eşzamanlılık sorunlarından biri, **yemek filozofunun problemi** olarak bilinir [D71]. Prob-lem ünlüdür çünkü eğlenceli ve entelektüel olarak biraz ilginçtir; ancak, pratik faydası düşüktür. Bununla birlikte, şöhreti buradaki ilgisini zorlar; Gerçekten de bazı röportajlarda size sorulabilir ve bu soruyu kaçırırsanız ve işi almazsanız, işletim sistemi profesörünüzden gerçekten nefret edersiniz. Tersine, işi alırsanız, lütfen işletim sistemi profesörünüze güzel bir not veya bazı hisse senedi seçenekleri göndermekten çekinmeyin.

Sorunun temel düzeni şudur (Şekil 31.14'te gösterildiği gibi): Bir masanın etrafında oturan beş "filozof" olduğu gibi her filozof çifti arasında tek bir çatal (ve dolayısıyla toplam beş) vardır. Felsefecilerin her birinin düşündükleri zamanları vardır ve çatallara ihtiyaç duymazlar ve yemek yedikleri zamanlar. Yemek yemek için, bir filozofun iki çatala ihtiyacı vardır hem solundaki çatal hem de sağındaki çatal. Bu çatallar için çekişme ve ortaya çıkan senkronizasyon problemleri, bunu eşzamanlı programlamada çalıştığımız bir problem yapan şeydir.

İşte her filozofun temel döngüsü, her birinin 0'dan 4'e (dahil) kadar benzersiz bir iş parçacığı tanımlayıcısı p'ye sahip olduğunu varsayarsak:

```
while (1) {
    think ();
    get_forks(p);
    eat();
    put_forks(p);
}
```

O halde temel zorluk, get_forks() ve çatalları () koyun, böylece kilitlenme olmaz, hiçbir filozof aç kalmaz ve açlıktan ölmez.

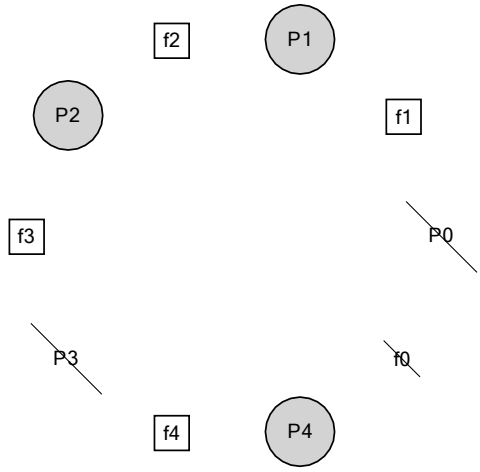

Figure 31.14: **Yemek Filozofları**

asla yemek yiyemez ve eşzamanlılık yüksektir (yani, mümkün olduğunca çok filozof aynı anda yiyebilir).

Downey'nin çözümlerini [D08] takiben, bizi bir çözüme ulaştırmak için birkaç yardımcı işlev kullanacağız. Bunlar:

```
int left(int p)  { return p; }
int right(int p) { return (p + 1) % 5; }
```

Filozof p, soldaki çatala atıfta bulunmak istediğinde, basitçe sol(p) olarak adlandırırlar. Benzer şekilde, bir filozofun sağındaki çatala sağ(p) denilerek atıfta bulunulur; buradaki modulo operatörü, son filozofun (p=4) sağındaki çatalı, yani çatal 0'ı tutmaya çalıştığı bir durumu ele alır.

Bu sorunu çözmek için bazı semaforlara da ihtiyacımız olacak. Her çatal için bir tane olmak üzere beş tane olduğunu varsayalım: sem t forks[5].

Kırık Çözüm

Soruna ilk çözümümüzü deniyoruz. Her semaforu (çatallar dizisinde) 1 değerine başlattığımızı varsayalım. Ayrıca her filozofun kendi sayısını bildiğini varsayalım (p). Böylece get_forks() ve put_forks() rutinini koyabiliriz (Şekil 31.15, sayfa 15).

Bu (kırılmış) çözümün ardındaki sezgi aşağıdaki gibidir. Çatalları elde etmek için, her birinde bir "kilit" alırız: önce soldaki,

```
1 void get_forks(int p) {
2     sem_wait(&forks[left(p)]);
3     sem_wait(&forks[right(p)]);
4 }
5
6 void put_forks(int p) {
7     sem_post(&forks[left(p)]);
8     sem_post(&forks[right(p)]);
9 }
```

Figure 31.15: **The getforks()ve putforks()rutin**

```
1 void get_forks(int p) {
2     if (p == 4) {
3         sem_wait(&forks[right(p)]);
4         sem_wait(&forks[left(p)]);
5     } else {
6         sem_wait(&forks[left(p)]);
7         sem_wait(&forks[right(p)]);
8     }
9 }
```

Figure 31.16: **Bağımlılığı Kırmak get_forks()**

ve sonra sağdaki. Yemek yemeyi bitirdiğimizde, onları serbest bırakırız. Basit, değil mi? Ne yazık ki, bu durumda, basit araçlar kırıldı. Ortaya çıkan sorunu görebiliyor musunuz? Bir düşünün.

Sorun **ölümcül kitlenme(deadlock)**. Eğer her filozof, herhangi bir filozof sağındaki çatalı tutamadan önce solundaki çatalı tutarsa, her biri bir çatalı tutup sonsuza dek diğerini beklerken sıkışıp kalacaktır. Belirleyici, filozof 0 çatal 0'ı yakalar, filozof 1 çatal 1'i yakalar, filozof 2 Çatal 2'yi yakalar, filozof 3 çatalı 3'ü yakalar ve filozof 4 çatal 4'ü yakalar; tüm çatallar elde edilir ve tüm filozoflar başka bir filozofun sahip olduğu çatalı beklerken sıkışıp kalırlar. Çıkmazı yakında daha ayrıntılı olarak inceleyeceğiz; şimdilik, bunun çalışan bir çözüm olmadığını söylemek güvenlidir.

Bir Çözüm: Bağımlılığı Kırmak

Bu soruna saldırmanın en basit yolu, filozoflardan en az biri tarafından çatalların nasıl sorgulandığını değiştirmektir; aslında, Dijkstra'nın kendisi sorunu böyle çözdü. Özellikle, filozof 4'ün (en yüksek numaralandırılan) çatalları diğerlerinden *farklı* bir sırada aldığını varsayalım (Şekil 31.16); put_forks() kodu aynı kalır.

Son filozof soldan önce kavramaya çalıştığı için, her filozofun bir çatalı tutup diğerini beklerken sıkışıp kaldığı bir durum yoktur; bekleme döngüsü kırılır. Bu çözümün sonuçlarını düşünün ve kendinizi işe yaradığına ikna edin.

Bunun gibi başka "ünlü" problemler de var, örneğin **sigara içen kişinin (cigarette smoker's problem) sorunu** veya **uyuyan berber problemi (sleeping barber problem).** Birçoğu eşzamanlılık hakkında düşünmek için sadece bahaneler; bazılarının büyüleyici isimleri var. Daha fazla şey öğrenmek veya sadece eşzamanlı bir şekilde düşünme konusunda daha fazla pratik yapmakla ilgileniyorsanız onlara bakın [D08].

1. İş Parçacığı Azaltma

Semaforlar için bir başka basit kullanım durumu zaman zaman ortaya çıkar ve bu nedenle burada sunarız. Özel sorun şudur: Bir programcı "çok fazla" iş parçacığının aynı anda bir şey yapmasını ve sistemi tıkamasını nasıl önleyebilir? Cevap: "Çok fazla" için bir eşiğe karar verin ve ardından söz konusu kod parçasını eşzamanlı olarak yürüten iş parçacığı sayısını sınırlamak için bir semafor kullanın. Bu **yaklaşıma kısıtlama (throttling)** [T99] diyoruz ve bunu bir **kabul kontrolü (admission control)** biçimi olarak görüyoruz.

Daha spesifik bir örnek düşünelim. Bazı problemler üzerinde paralel olarak çalışmak için yüzlerce iş parçacığı oluşturduğunuzu hayal edin. Bununla birlikte, kodun belirli bir bölümünde, her iş parçacığı hesaplamanın bir bölümünü gerçekleştirmek için büyük miktarda özellik kazanır; kodun bu bölümüne *bellek yoğun bölge* diyelim. *Tüm* iş parçacıkları yoğun bellek kullanan bölgeye aynı anda girerse, tüm bellek ayırma isteklerinin toplamı makinedeki fiziksel bellek miktarını aşar. Sonuç olarak, makine parçalanmaya başlayacak (yani, diske ve diskten sayfa değiştirme) ve tüm hesaplama bir taramaya yavaşlayacaktır.

Basit bir semafor bu sorunu çözebilir. Semaforun değerini, bellek yoğun bölgeye bir kerede girmek istediğiniz maksimum iş parçacığı sayısına başlatarak ve ardından bölgenin etrafına bir sem_wait() ve sem_post() koyarak, bir semafor doğal olarak tehlikeli bölgede eşzamanlı olarak bulunan iş parçacığı sayısını azaltabilir.

1. Semaforlar Nasıl Uygulanır

Son olarak, düşük seviyeli senkronizasyon ilkellerimizi, kilitlerimizi ve koşul değişkenlerimizi kullanarak, kendi semafor versiyonumuzu oluşturmak için ... *(Davul rulosu burada)* ... **Zemaforlar**. Bu görev, Şekil 31.17'de (sayfa 17) görebileceğiniz gibi oldukça basittir.

Yukarıdaki kodda, yalnızca bir kilit ve bir koşul değişkeninin yanı sıra semaforun değerini izlemek için bir durum değişkeni kullanıyoruz. Gerçekten anlayana kadar kodu kendiniz inceleyin.

Semaforumuz ile Dijkstra tarafından tanımlanan saf semaforlar arasındaki ince bir fark, semaforun değerinin, negatif olduğunda, bekleyen ipliklerin sayısını yansıttığı değişmezliğini korumamamızdır; aslında, değer asla sıfırdan düşük olmayacaktır. Bu davranışın uygulanması daha kolaydır ve geçerli Linux uygulamasıyla eşleşir.

```
1  typedef struct __Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }
```

Figure 31.17: **Zemaforların Kilitler ve CV'lerle Uygulanması**

İlginçtir ki, semaforlardan durum değişkenleri oluşturmak çok daha zor bir önermedir. Bazı son derece deneyimli eşzamanlı programcılar bunu Windows ortamında yapmaya çalıştı ve birçok farklı hata ortaya çıktı [B04]. Kendiniz deneyin ve semaforlardan durum değişkenleri oluşturmanın neden bir problemin göründüğünden daha zor olduğunu anlayıp anlayamadığınızı görün.

31.9.1 Özet

Semaforlar, eşzamanlı programlar yazmak için güçlü ve esnek bir ilkeldir. Bazı programcılar basitlikleri ve kullanışlılıkları nedeniyle bunları yalnızca kilitlerden ve koşul değişkenlerinden kaçınarak kullanırlar.

Bu bölümde, sadece birkaç klasik problem ve çözüm sunduk. Daha fazlasını öğrenmekle ilgileniyorsanız, başvurabileceğiniz başka birçok malzeme var. Harika bir (ve ücretsiz referans) Allen Downey'nin eşzamanlılık ve semaforlarla programlama hakkındaki kitabıdır [D08]. Bu kitapta, anlayışınızı geliştirmek için üzerinde çalışabileceğiniz birçok bulmaca var.

GENELLEŞTİRMEYE DİKKAT EDİN

Bu nedenle, soyut genelleme tekniği, iyi bir fikrin biraz daha geniş hale getirilebileceği ve böylece daha büyük bir sınıf kuralının çözülebileceği sistem tasarımında oldukça yararlı olabilir. Sorun. Ancak, genelleme yaparken dikkatli olun; Lampson'un bizi uyardığı gibi "Genelleme yapmayın; genellemeler genellikle yanlıştır" [L83].

Semaforları kilitlerin ve koşul değişkenlerinin genellemesi olarak görebiliriz; ancak, böyle bir genellemeye ihtiyaç var mı? Ve bir semaforun üstünde bir koşul değişkeni gerçekleştirmenin zorluğu göz önüne alındığında, belki de bu genelleme düşündüğünüz kadar genel değildir. .

Her iki semaforun özel olarak ve genel olarak eşzamanlılık. Gerçek bir eşzamanlılık uzmanı olmak yıllarca çaba gerektirir. Bu derste öğrendiklerinizin ötesine geçmek, şüphesiz böyle bir konuda uzmanlaşmanın anahtarıdır.

Kaynaklar

[B04] "Implementing Condition Variables with Semaphores" by Andrew Birrell. December 2004. *An interesting read on how difficult implementing CVs on top of semaphores really is, and the mistakes the author and co-workers made along the way. Particularly relevant because the group had done a ton of concurrent programming; Birrell, for example, is known for (among other things) writing various thread-programming guides.*

[CB08] "Real-world Concurrency" by Bryan Cantrill, Jeff Bonwick. ACM Queue. Volume 6, No. 5. September 2008. *A nice article by some kernel hackers from a company formerly known as Sun on the real problems faced in concurrent code.*

[CHP71] "Concurrent Control with Readers and Writers" by P.J. Courtois, F. Heymans, D.L. Parnas. Communications of the ACM, 14:10, October 1971. *The introduction of the reader-writer problem, and a simple solution. Later work introduced more complex solutions, skipped here because, well, they are pretty complex.*

[D59] "A Note on Two Problems in Connexion with Graphs" by E. W. Dijkstra. Numerische Mathematik 1, 269271, 1959. Available: `http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf`. *Can you believe people worked on algorithms in 1959? We can't. Even before computers were any fun to use, these people had a sense that they would transform the world...*

[D68a] "Go-to Statement Considered Harmful" by E.W. Dijkstra. CACM, volume 11(3), March 1968. `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF`. *Sometimes thought of as the beginning of the field of software engineering.*

[D68b] "The Structure of the THE Multiprogramming System" by E.W. Dijkstra. CACM, volume 11(5), 1968. *One of the earliest papers to point out that systems work in computer science is an engaging intellectual endeavor. Also argues strongly for modularity in the form of layered systems.*

[D72] "Information Streams Sharing a Finite Buffer" by E.W. Dijkstra. Information Processing Letters 1, 1972. `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF`. *Did Dijkstra invent everything? No, but maybe close. He certainly was the first to clearly write down what the problems were in concurrent code. However, practitioners in OS design knew of many of the problems described by Dijkstra, so perhaps giving him too much credit would be a misrepresentation.*

[D08] "The Little Book of Semaphores" by A.B. Downey. Available at the following site: `http://greenteapress.com/semaphores/`. *A nice (and free!) book about semaphores. Lots of fun problems to solve, if you like that sort of thing.*

[D71] "Hierarchical ordering of sequential processes" by E.W. Dijkstra. Available online here: `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF`. *Presents numerous concurrency problems, including Dining Philosophers. The wikipedia page about this problem is also useful.*

[GR92] "Transaction Processing: Concepts and Techniques" by Jim Gray, Andreas Reuter. Morgan Kaufmann, September 1992. *The exact quote that we find particularly humorous is found on page 485, at the top of Section 8.8: "The first multiprocessors, circa 1960, had test and set instructions ... presumably the OS implementors worked out the appropriate algorithms, although Dijkstra is generally credited with inventing semaphores many years later." Oh, snap!*

[H87] "Aspects of Cache Memory and Instruction Buffer Performance" by Mark D. Hill. Ph.D. Dissertation, U.C. Berkeley, 1987. *Hill's dissertation work, for those obsessed with caching in early systems. A great example of a quantitative dissertation.*

[L83] "Hints for Computer Systems Design" by Butler Lampson. ACM Operating Systems Review, 15:5, October 1983. *Lampson, a famous systems researcher, loved using hints in the design of computer systems. A hint is something that is often correct but can be wrong; in this use, a signal() is telling a waiting thread that it changed the condition that the waiter was waiting on, but not to trust that the condition will be in the desired state when the waiting thread wakes up. In this paper about hints for designing systems, one of Lampson's general hints is that you should use hints. It is not as confusing as it sounds.*

[T99] "Re: NT kernel guy playing with Linux" by Linus Torvalds. June 27, 1999. Available: `https://yarchive.net/comp/linux/semaphores.html`. *A response from Linus himself about the utility of semaphores, including the throttling case we mention in the text. As always, Linus is slightly insulting but quite informative.*

Ödev (Kod)

Bu ödevde, bazı iyi bilinen eşzamanlılık problemlerini çözmek için semaforlar kullanacağız. Bunların çoğu, Downey'nin bir dizi klasik problemi bir araya getirmenin yanı sıra birkaç yeni varyant getirme konusunda iyi bir iş çıkaran mükemmel "Küçük Semafor Kitabı"ndan3 alınmıştır; İlgilenen okuyucular daha fazla eğlence için Küçük Kitap'a göz atmalıdır. Aşağıdaki soruların her biri bir kod iskeleti sağlar; işiniz semaforlar verilerek çalışmasını sağlamak için kodu doldurmaktır. Linux'ta yerel semaforlar kullanacaksınız. Mac'te (semafor desteğinin olmadığı yerlerde), önce bir uygulama oluşturmanız gerekir (bölümde açıklandığı gibi kilitleri ve koşul değişkenlerini kullanarak). İyi şanslar!

Sorular

1- İlk problem, metinde açıklandığı gibi sadece fork/join problemine bir çözüm uygulamak ve test etmektir. Bu çözüm metinde anlatılmış olsa da kendi başınıza yazmanız faydalı olacaktır; Bach bile Vivaldi'yi yeniden yazacak ve yakında usta olacak birinin mevcut bir ustadan öğrenmesine izin verecektir. Ayrıntılar için fork-join.c'ye bakın. Çalıştığından emin olmak için uyku (1) (sleep (1)) çağrısını ekleyin.

```
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ g
cc -o fork-join fork-join.c -Wall -pthread
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ .
/fork-join
parent: begin
child
parent: end
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$
```

Bu problem, bir işletim sisteminde birden fazla işlemin aynı anda yürütülmesini sağlamak için kullanılan bir yöntemdir.
Bir fork-join çözümü, bir işletim sisteminin bir işlemin ayrı bir iş parçacığı olarak çalıştırılmasını sağlar. Bu iş parçacığı, işlemin tamamlanmasını bekleyen bir "join" işlemiyle sonuçlandırılır. Bu sayede, birden fazla işlemin aynı anda yürütülmesi sağlanmış olur.

2- Rendezvouas problemini inceleyerek bunu biraz genelleştirelim. Sorun şudur: Her biri kodda buluşma noktasına girmek üzere olan iki iş parçacığınız var. İkisi de kodun bu bölümünden diğeri girmeden çıkmamalıdır. Bu görev için iki semafor kullanmayı düşünün ve ayrıntılar için rendezvous.c adresine bakın.

```
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ g
cc -o rendezvous rendezvous.c -Wall -pthread
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ .
/rendezvous
parent: begin
child 2: before
child 1: before
child 1: after
child 2: after
parent: end
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$
```

Rendezvous problemi, birden fazla iş parçacığının bir araya gelip birbirleriyle etkileşim kurabilecekleri bir yapıya ihtiyaç duyulduğu durumlarda ortaya çıkan bir problemdir. Bu problemi çözmek için, iki iş parçacığının aynı anda bir noktaya (kodda buluşma noktasına) ulaşmalarını sağlamak için iki semafor kullanılabilir. Bu sayede, iki iş parçacığı da birbirlerini bekleyerek veya birbirlerine bildirim göndererek koddaki buluşma noktasına erişebilir. Detaylı bilgi için, rendezvous.c adresinde bulunan kodda yer alan açıklamalara bakılabilir.

3- Şimdi bariyer senkronizasyonuna genel bir çözüm uygulayarak bir adım daha ileri gidin. Sıralı bir kod parçasında P1 ve P2 olarak adlandırılan iki nokta olduğunu varsayalım. P1 ve P2 arasına bir bariyer koymak, herhangi bir iş parçacığı P2'yi yürütmeden önce tüm iş parçacıklarının P1'i yürütmesini garanti eder. Senin görevin: yaz bu şekilde kullanılabilecek bir bariyer() işlevini uygulayan kod. N (çalışan programdaki toplam iş parçacığı sayısı) bildiğinizi ve tüm N iş parçacığının engele girmeye çalışacağını varsaymak güvenlidir. Yine, çözümü elde etmek için muhtemelen iki semafor ve şeyleri saymak için başka tamsayılar kullanmalısınız. Ayrıntılar için barrier.c'ye bakın.

```
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ g
cc -o barrier barrier.c -Wall -pthread
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ .
/barrier 4
parent: begin
child 0: before
child 1: before
child 2: before
child 3: before
child 2: after
child 3: after
child 0: after
child 1: after
parent: end
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ .
```

Bir bariyer fonksiyonunu yazmak için, ilk olarak bir semafor nesnesi oluşturmak ve bu nesneyi başlangıçta kilitlemek gerekir. Daha sonra, her iş parçacığı P1 noktasına gelirken semaforu kilitlemelidir ve P2 noktasına gelirken de kilidini açmalıdır. Böylece, P2 noktasına gelen her iş parçacığı P1 noktasında bekleyen diğer iş parçacıklarının tamamlanmasını bekleyecektir. Ayrıca, bir sayaç da kullanarak tüm iş parçacıklarının P1 noktasına geldiğini ve P2 noktasına geçtiğini takip etmeliyiz.

4- Şimdi yine metinde anlatıldığı gibi okuyucu-yazar problemini çözelim. İçinde Bu ilk çekim, açlıktan endişe etmeyin. Ayrıntılar için reader-writer.c'deki koda bakın. Beklediğiniz gibi çalıştığını göstermek için kodunuza sleep() uyku() çağrıları ekleyin. Açlık sorununun varlığını gösterebilir misiniz?

```
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ g
cc -o reader-writer reader-writer.c -Wall -pthread
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ .
/reader-writer 1 2 3
begin
read 0
read 0
read 0
write 1
write 2
write 3
write 4
write 5
write 6
end: value 6
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$
```

ES

Okuyucu-yazar problemi, eşzamanlı süreçlerin senkronizasyonu ile ilgilenen klasik bir bilgisayar bilimi problemidir. Sorun genellikle şu şekilde ifade edilir: birden çok iş parçacığı (veya işlem), bir dosya veya veritabanı gibi paylaşılan bir kaynağa erişmek ister, ancak bunu çakışmaları önleyecek ve veri bütünlüğünü koruyacak şekilde yapmalıdır.

Verdiğiniz kodda okuyucu-yazar probleminin çözümü semaforlar kullanılarak gerçekleştirilmiştir. Semaforlar, iş parçacıklarının paylaşılan kaynaklara erişimlerini koordine etmesine izin veren bir eşitleme mekanizmasıdır. Bu durumda semaforlar, paylaşılan kaynağa herhangi bir zamanda yalnızca bir iş parçacığının (bir okuyucu veya bir yazar) erişebilmesini sağlamak için kullanılır.

sleep()Kodun beklendiği gibi çalıştığını göstermek için okuyucu ve yazar dizilerine çağrılar ekleyebilirsiniz . Bu, iş parçacıklarının belirli bir süre boyunca duraklamasına neden olarak, paylaşılan kaynağa erişimi koordine etmek için semaforların nasıl kullanıldığını görmenizi sağlar.

Açlık sorununa gelince, bu, bir veya daha fazla iş parçacığının, kaynak kullanılabilir olmasına rağmen, paylaşılan bir kaynağa erişiminin defalarca reddedildiği bir durumu ifade eder. Okuyucu-yazar probleminde, yazarlardan çok okuyucu varsa bu olabilir veya tam tersi de olabilir. Örneğin, yazarlardan çok okuyucu varsa, okuyucular sürekli olarak eriştiği için yazarlar paylaşılan kaynağa erişimden mahrum kalabilir.

Aç kalma sorununun varlığını göstermek için, çok sayıda okuyucu veya yazar iş parçacığı oluşturmak için kodu değiştirebilir ve ardından paylaşılan kaynağa erişimi koordine etmek için semaforların nasıl kullanıldığını gözlemleyebilirsiniz. Kaynak mevcut olmasına rağmen bir tür ileti dizisinin (okuyucuların veya yazarların) kaynağa erişiminin sürekli olarak reddedildiğini görürseniz, o zaman bir açlık sorunu olduğu sonucuna varabilirsiniz.

5- Okur-yazar sorununa tekrar bakalım ama bu sefer açlıktan endişelenelim. Tüm okuyucuların ve yazarların sonunda ilerleme kaydetmesini nasıl sağlayabilirsiniz? Ayrıntılar için okuyucu-yazar-nostarve.c'ye bakın.

```
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ g
cc -o reader-writer-nostarve reader-writer-nostarve.c -Wall -pthread
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ .
/reader-writer-nostarve 1 2 3
begin
read 0
write 1
write 2
write 3
write 4
write 5
write 6
read 6
read 6
end: value 6
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$
```

Okuyucu-yazar sorunu, birden çok işlem veya iş parçacığı dosya veya veritabanı gibi paylaşılan bir kaynağı okumaya ve bu kaynağa yazmaya çalıştığında ortaya çıkan klasik bir eşitleme sorunudur. Orijinal problemde sadece okuyucuların ve yazarların paylaşılan kaynağa birbirlerini etkilemeden erişebilmelerini sağlamaya önem verdik. Bununla birlikte, sorunun açlığı dikkate alan versiyonunda, tüm okuyucuların ve yazarların sonunda ilerleme kaydetmelerini ve paylaşılan kaynağa erişimden "aç kalmamalarını" sağlamamız gerekir.

Tüm okuyucuların ve yazarların ilerleme kaydetmesini sağlamanın bir yolu, yazarlara okuyuculardan daha yüksek önceliğin verildiği, önceliğe dayalı bir yaklaşım kullanmaktır. Bu, bir yazar paylaşılan kaynağa erişmeyi beklerken, bekleyen herhangi bir okuyucudan önce buna izin verileceği anlamına gelir. Bu, eninde sonunda paylaşılan kaynağa her zaman erişebileceklerinden yazarların aç kalmamasını sağlar.

Açlığı önlemenin bir başka yolu da, paylaşılan kaynağa erişmeleri için her okuyucuya ve yazara belirli sayıda "dönüş" verildiği bir "adalet" politikası kullanmaktır. Örneğin, bir adalet ilkesi, bir okuyucunun kaynaktan okumasına izin vermeden önce her yazarın paylaşılan kaynağa iki kez yazmasına izin verebilir. Bu, hem okuyucuların hem de yazarların paylaşılan kaynağa erişmek için eşit fırsatlara sahip olmasını sağlar ve herhangi bir işlemin aç kalmasını önler.

Her iki durumda da, paylaşılan kaynağa erişimi kontrol etmek için kullanılan senkronizasyon mekanizmalarını dikkatli bir şekilde tasarlamak, tüm okuyucuların ve yazarların ilerleme kaydedebilmesini ve hiçbir işlemin aç kalmamasını sağlamak önemlidir. Bu, semaforlar, muteksler veya diğer senkronizasyon ilkelleri gibi çeşitli teknikler kullanılarak yapılabilir.

6- Aç kalmayan bir mutex oluşturmak için semaforları kullanın; mutex elde etmek sonunda onu elde edecektir. mutex-nostarve.c içindeki koda bakın daha fazla bilgi için.

```
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ g
cc -o mutex-nostarve mutex-nostarve.c -Wall -pthread
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$ .
/mutex-nostarve 4
parent: begin
counter: 4
parent: end
mert@mert-virtual-machine:~/Desktop/ostep-homework-master/ostep-hw-master/31$
```

Semaforları kullanarak aç kalmayan bir muteks uygulamak için, daha yüksek önceliğe sahip iş parçacıklarına daha düşük önceliğe sahip olanlara göre öncelik verildiği, önceliğe dayalı bir yaklaşım kullanabiliriz. Bu, tüm iş parçacıkları eninde sonunda muteksi elde edebileceğinden, hiçbir iş parçacığının aç kalmamasını sağlar.

Bunu yapmak için iki semafor kullanabiliriz: biri mutekse erişimi kontrol etmek için, diğeri ise muteksi elde etmeyi bekleyen iş parçacığı sayısını izlemek için. Bir iş

parçacığı muteksi her edinmeye çalıştığında, önce ikinci semaforun değerini kontrol eder. Değer sıfırdan farklıysa, muteksi elde etmek için bekleyen başka evreler olduğu anlamına gelir, bu nedenle evre ikinci semaforun değerini arttırır ve bekler.

Bir iş parçacığı muteksi serbest bıraktığında, ikinci semaforun değerini tekrar kontrol eder. Değer sıfır değilse, muteksi elde etmek için bekleyen iş parçacıkları olduğu anlamına gelir, bu nedenle iş parçacığı ikinci semaforun değerini azaltır ve en yüksek önceliğe sahip bekleyen iş parçacığına devam etmesi için sinyal verir. Bu, en yüksek önceliğe sahip iş parçacığının muteksi alabilmesini ve hiçbir iş parçacığının aç kalmamasını sağlar.

Bu yaklaşım, muteksi elde etmeye çalışan tüm iş parçacıkları sonunda bunu yapabileceğinden, hiçbir iş parçacığının aç kalmamasını sağlar. Bununla birlikte, herhangi bir potansiyel adaletsizliği veya verimsizliği önlemek için hangi iş parçacığının muteksi alması gerektiğini belirlemek için kullanılan öncelik sistemini dikkatli bir şekilde tasarlamak önemlidir.

7- Bu sorunları beğendiniz mi? Onlar gibi daha fazlası için Downey'nin ücretsiz metnine bakın. Ve unutmayın, iyi eğlenceler! Ama kod yazarken her zaman yaparsın, değil mi?

[3]: `http://greenteapress.com/semaphores/downey08semaphores.pdf`.