31

Semaforlar

Şimdi bildiğimiz gibi, çok çeşitli alakalı ve ilginç **eşzamanlılık (concurrency)** problemlerini çözmek için hem **kilitlere(locks)** hem de **koşul değişkenlerine(condition variables)** ihtiyaç duyulur.Bunu yıllar önce ilk fark edenlerden biri Edsger Dijkstra'ydı (geçmişini tam olarak bilmek zor olsa da [GR92]), diğer şeylerin yanı sıra grafik teorisindeki ünlü "en kısa yollar" algoritmasıyla tanınır [D59], "Zararlı Kabul Edilen İfadelere Git" [D68a] başlıklı yapılandırılmış **programlama (programming)** üzerine erken bir polemik (ne harika bir başlık!), ve, burada inceleyeceğimiz durumda, **semafor (semaphores)** [D68b, D72] adı verilen bir senkronizasyon ilkelinin tanıtılmasıdır. Aslında, Dijkstra ve meslektaşları, semaforu senkronizasyonla ilgili her şey için tek bir **ilkel(primitive)** olarak icat ettiler; göreceğiniz gibi **semaforlar(semaphores)** hem **kilitler(locks)** hem de **koşul değişkenleri(condition variables)** olarak kullanılabilir.

TEMEL ÖZELLİKLER: SEMAFOR NASIL KULLANILIR

Kilitler(locks) ve **durum değişkenleri(state veriables)** yerine semaforları nasıl kullanabiliriz? Bir semaforun tanımı nedir? İkili semafor nedir? **Kilitlerden(locks)** ve **koşul değişkenlerinden (contion veriables)** bir semafor oluşturmak kolay mı? Semaforlardan **kilitler(locks**) ve koşul değişkenleri oluşturmak için mi?

31.1Semaforlar: Tanım

Bir semafor, iki rutin ile işleyebileceğimiz bir **tamsayı (integer)** değerine sahip bir nesnedir; **taşınabilir işletim sistemi arabirimi (POSIX)** standardında, bu rutinler sem_wait() ve sem_post()[1] Semaforun başlangıç **değeri (value)** davranışını belirlediği için, semaforla **etkileşime (interaction)** geçmek için başka bir rutini çağırmadan önce, önce onu bir **değere(value)** başlatmalıyız,

Şekil 31.1'deki **kodun (code)** yaptığı gibi.

.

[1]Tarihsel olarak, sem wait(), Dijkstra tarafından P() olarak, sem post() ise V() olarak adlandırıldı. Bu kısaltılmış biçimler Hollandaca sözcüklerden gelir; ilginç bir şekilde, türediği varsayılan Hollandaca kelimeler zamanla değişti. Aslında, P() "geçmekten" (geçmek) ve V() "vrijgave"den (bırakmak) geldi; sonra,Dijkstra, P()'nin "prolaag"dan olduğunu yazdı, "probeer" ("denemek" için Hollandaca) ve "verlaag" ("azaltma") kısaltması, ve "artırma" anlamına gelen "verhoog"dan V().Ara sıra, insanlar onları aşağı ve yukarı çağırır. Arkadaşlarınızı etkilemek için Hollandaca versiyonları kullanın,ya da karıştır,ya da her ikisi de. Ayrıntılar için https://news.ycombinator.com/ item?id=8761539) bakın.

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

Şekil 31.1: Semafor Başlatma

Şekilde, bir semafor s bildiririz ve üçüncü argüman olarak 1'i ileterek onu 1 değerine başlatırız.Sem_init()'in ikinci argümanı, göreceğimiz tüm örneklerde 0 olarak ayarlanacaktır; bu, init() semaforunun aynı **süreçte(process) evreler (threads**) arasında paylaşıldığını gösterir. Semaforların diğer kullanımlarıyla ilgili ayrıntılar için kılavuz sayfasına bakın (yani, farklı **süreçler (process)** arasında **erişimi (access)** senkronize etmek için nasıl kullanılabilecekleri), ki bu ikinci **bağımsız değişken (independent veriable)** için farklı bir **değer(value)** gerektirir. Bir semafor başlatıldıktan sonra, onunla **etkileşime(interaction)** geçmek için iki **işlevden(function)** birini **çağırabiliriz(call),** sem_wait() veya sem_post işlevleri. Şekil 31.2'de görülmektedir.

Şimdilik, bu rutinlerin uygulanmasıyla ilgilenmiyoruz, ki bu açıkça biraz dikkat gerektiriyor; sem_wait() ve sem_post()'u **çağıran(calling**) birden çok **iş parçacığıyla(thread),** bu **kritik bölümleri(critical section]** yönetmek için bariz bir ihtiyaç var. Şimdi bu **ilkelleri(primitive)** nasıl kullanacağımıza odaklanacağız; daha sonra nasıl inşa edildiklerini tartışabiliriz. Burada **arayüzlerin(interface)** birkaç göze çarpan yönünü tartışmalıyız. Öncelikle, sem_wait()'in ya hemen **döneceğini(return)** görebiliriz (çünkü sem wait()'i çağırdığımızda semaforun değeri bir veya daha yüksekti), veya arayan kişinin bir sonraki gönderiyi **beklerken(wait) yürütmeyi(executive)** askıya almasına neden olur. Tabii ki, birden çok **çağrı dizisi(call sequence)** sem_wait() **işlevini(thread)** çağırabilir, ve böylece hepsi uyandırılmayı **beklerken(wait) kuyruğa(queue)** alınır. İkinci, sem_post()'un sem_wait()'in yaptığı gibi belirli bir koşulun gerçekleşmesini beklemediğini görebiliriz. Yerine, sadece semaforun değerini arttırır ve sonra, uyandırılmayı **bekleyen(waiting)** bir konu varsa, birini uyandırır. Üçüncüsü, negatif olduğunda semaforun değeri, **bekleyen(waiting**) **iş parçacığı(thread)** 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 }
```

Şekil 31.2: Semafor: Bekleme(wait) ve Gönderme(post) 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);
```

Şekil 31.3: **İkili (Binary) Semafor (Bir Kilit (lock))**

Semafor içinde mümkün görünen **yarış koşulları(race condition)**hakkında (henüz) endişelenmeyin;
yaptıkları eylemlerin atomik olarak gerçekleştirildiğini varsayın.
Yakında tam da bunu yapmak için **kilitleri(locks)** ve **koşul değişkenlerini (condition variables)** kullanacağız.

İkili Semaforlar (Kilitler (Locks))

Artık bir semafor kullanmaya hazırız. İlk kullanımımız zaten aşina olduğumuz bir kullanım olacak: semaforu **kilit(lock**) olarak kullanmak.Şekle Bakın bir **kod(code)** parçacığı için; orada,ilgilenilen **kritik bölümü(critical section)** bir sem_wait()/sem_post() çifti ile çevrelediğimizi göreceksiniz. Bu işi yapmak için kritik Yine de, m semaforunun başlangıç değeridir (şekilde X olarak başlatılmıştır). X ne olmalı?

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

Yukarıdaki sem_wait() ve sem_post() rutinlerinin tanımına bakıldığında, ilk **değerin(value)** 1 olması gerektiğini görebiliriz

Bunu açıklığa kavuşturmak için, iki **iş parçacıklı (thread)** bir senaryo düşünelim. İlk **iş parçacığı (thread) (İplik(thread)** 0) sem_wait();'i **çağırır(calling)**; önce semaforun değerini azaltacak, 0 olarak değiştiriyor. O zamanlar, yalnızca değer 0'dan büyük veya 0'a eşit değilse **bekler(wait)**. Değer 0 olduğu için, sem_wait() basitçe geri döner ve **çağrı dizisi(call sequence)** devam eder; Konu 0 artık **kritik bölüme(critical section)** girmekte serbesttir. **İş parçacığı(thread)** 0 **kritik bölümün(critical section)** içindeyken başka bir **iş parçacığı (thread) kilidi(locks)** almaya çalışmıyorsa, Sem_post() **çağrıldığında(calling)**, basitçe semaforun değerini 1'e geri yükler (ve **bekleyen(wait)** bir **iş parçacığını(thread)** uyandırmaz, çünkü hiçbiri yok). Şekil 31.4 bu senaryonun izini göstermektedir.

Daha ilginç bir durum, **İplik(thread)** 0 "kilidi tuttuğunda" ortaya çıkar (yani, sem_wait()'i **çağırdı(call)** ama henüz sem_post()'u çağırmadı), ve başka bir **iş parçacığı(thread)** (İplik 1) sem wait()'i çağırarak **kritik bölüme(critical section)** girmeye çalışır. Bu durumda, Konu 1, 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	

Şekil 31.4: **Konu İzleme: Semafor Kullanan Tek İş Parçacığı (Thread)**

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

Şekil 31.5: **Konu İzleme: Semafor Kullanan İki İş Parçacığı (Thread)**

böylece bekleyin (kendini **uyku moduna(sleep)** geçirerek ve **işlemciden(processor)** vazgeçerek). Thread 0 tekrar **çalıştığında(running),** sonunda sem_post()'u **çağıracak(call**), semaforun değerini tekrar sıfıra çıkaracak, ve ardından **bekleyen(wait) ipliği(thread)** uyandırır (Konu 1), bu daha sonra **kilidi(lock)** kendisi için alabilecektir. Konu 1 bittiğinde, semaforun değerini tekrar artıracaktır, tekrar 1'e geri yükleniyor.

Şekil 31.5 bu örneğin izini göstermektedir**. İş parçacığı(thread)** işlemlerine ek olarak, şekil, her **iş parçacığının(thread) programlayıcı durumunu(scheduler state)** gösterir: **Çalıştır(running) (iş parçacığı(thread) çalışıyor(running)**), **Hazır(ready)** (yani çalıştırılabilir ancak çalışmıyor), ve **Uyku(sleepy)** (**iş parçacığı(thread)** bloke edilmiştir). **İş Parçacığı(thread)** 1'in zaten tutulan **kilidi(lock)** almaya çalıştığında **uyku durumuna(sleep)** geçtiğini unutmayın; yalnızca **İş Parçacığı(thread)** 0 tekrar çalıştığında **İş Parçacığı(thread)** 1 uyandırılabilir ve potansiyel olarak tekrar **çalıştırılabilir(running**).

Kendi örneğinizle çalışmak isterseniz, birden çok **iş parçacığının(thread)** bir **kilit(lock) beklerken(wait) kuyruğa (queque)** girdiği bir senaryo deneyin. Böyle bir iz sırasında semaforun değeri ne olur? Böylece semaforları **kilit(lock)** olarak kullanabiliyoruz.Kilitlerin yalnızca iki durumu olduğundan (tutulur ve tutulmaz), bazen **kilit(lock)** olarak kullanılan bir semaforu ikili semafor olarak adlandırırız.Yalnızca bu ikili modda bir semafor kullanıyorsanız, burada sunduğumuz genelleştirilmiş semaforlardan daha basit bir şekilde uygulanabilir.

Sıralamak İçin Semaforlar

Semaforlar**, eşzamanlı(concurrency)** bir programdaki olayları sıralamak için de kullanışlıdır. Örneğin, bir **ileti dizisi(thread),** 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 }
```

Şekil 31.6: **Çocuğunu Bekleyen Bir Ebeveyn**

böylece ondan bir öğeyi silebilir.Bu kullanım biçiminde, genellikle bir şeyin olmasını bekleyen bir **iş parçacığı(thread)** buluruz, ve bir şeyin gerçekleşmesini sağlayan ve sonra bunun gerçekleştiğinin sinyalini veren başka bir **ileti dizisi(thread)**, böylece **bekleyen(waiting) ipliği(thread)** uyandırır. Bu nedenle semaforu bir sıralama **ilkel(primitive)** olarak kullanıyoruz (daha önce koşul değişkenlerini kullanmamıza benzer).

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

Soru, sonra, bu etkiyi elde etmek için bir semaforun nasıl kullanılacağıdır; anlaşıldığı üzere, cevabın anlaşılması nispeten kolaydır. Kodda da görebileceğiniz gibi, **ebeveyn(parent)**, yürütmesini bitiren **çocuğun(child)** koşulunun gerçekleşmesini beklemek için basitçe sem_wait() ve çocuk sem_post()'u **çağırır(call).** Yine de, bu şu soruyu gündeme getiriyor: Bu semaforun başlangıç değeri ne olmalıdır?

(Tekrar, burada düşün, önceden okumak yerine) Cevap, elbette, semaforun değerinin 0 olarak ayarlanması gerektiğidir. Dikkate alınması gereken iki durum var. Öncelikle, Ebeveynin **çocuğu(child)** yarattığını ancak **çocuğun(child)** henüz çalışmadığını varsayalım (yani, **hazır(ready) kuyruğunda(queue)** oturuyor ama çalışmıyor). Bu durumda (Şekil 31.7, sayfa 6),çocuk sem_post()'u çağırmadan önce **ebeveyn(parent)** sem_wait()'i **çağıracak(call)**; ebeveynin çocuğun **çalışmasını[running) beklemesini(wait)** isteriz. Bunun olmasının tek yolu, semaforun değerinin 0'dan büyük olmamasıdır; buradan, 0 başlangıç değeridir. **Ebeveyn(parent) çalışır(running**), semaforu azaltır (-1'e), sonra (**uyuyarak(sleeping)) bekler(wait). Çocuk(child)** nihayet **çalıştığında(running)** ,sem_post()'u **çağıracak(call)** ,değeri artır

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

Şekil 31.7: **Konu İzleme: Çocuğu 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

Şekil 31.8 **Konu İzleme: Çocuğu Bekleyen Ebeveyn (Durum 2)**

Semaforun 0'a, ve daha sonra sem_wait() işlevinden dönecek ve programı bitirecek olan **ebeveyni(parent)** uyandırın. İkinci durum (Şekil 31.8), ebeveyn sem_wait()'i çağırma şansı bulamadan **çocuk(child)** tamamlamaya **çalıştığında(run)** meydana gelir. Bu durumda,**çocuk(child)** önce sem_post()'u **çağıracak(call)** , böylece semaforun değeri 0'dan 1'e çıkar. **Ebeveyn(parent)** daha sonra **çalışma(run**) şansı yakaladığında, sem_wait()'i **çağıracak(call)** ve semaforun değerini 1 olarak bulacaktır; böylece **ebeveyn(parent)** değeri (0'a) düşürür ve beklemeden sem_wait()'den geri döner ve ayrıca istenen etkiyi elde eder.

Üretici/Tüketici (Sınırlı Tampon) Problemi

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

AYRICA: BİR SEMAFORUN DEĞERİNİ BELİRLEMEK

Şimdi bir semafor başlatmanın iki örneğini gördük. İlk durumda, semaforu **kilit(lock)** olarak kullanmak için değeri 1 olarak ayarladık; saniyede, 0'a, sıralama için semaforu kullanmak için. Öyleyse semafor başlatmanın genel kuralı nedir? Bunu düşünmenin basit bir yolu, Perry Kivolowitz sayesinde, başlatmanın hemen ardından vermek istediğiniz kaynak sayısını dikkate almaktır. Kilitle, 1'di, çünkü başlatmanın hemen ardından kilidin kilitlenmesini (verilmesini) istiyorsunuz. Sıralama durumu ile, 0'dı, çünkü başlangıçta verecek bir şey yoktur; yalnızca alt iş parçacığı tamamlandığında kaynak oluşturulur, hangi noktada, değer 1'e yükseltilir. Gelecekteki semafor problemlerinde bu düşünce tarzını deneyin, ve yardımcı olup olmadığına bakın.

İlk Girişim

Problemi çözmeye yönelik ilk girişimimiz**, iş parçacıklarının(threads)** sırasıyla bir **arabellek girişinin(buffer entry)** ne zaman boşaltıldığını veya doldurulduğunu belirtmek için kullanacağı boş ve dolu olmak üzere iki semafor sunar. **Koy ve al(put and get)** rutinlerinin kodu Şekil 31.9'dadır ve **üretici(producer)** ve **tüketici(consumer)** problemini çözme girişimimiz Şekil 31.10'dadır (sayfa 8).

Bu örnekte, **üretici(producer)**, içine **veri(data**) koymak için önce bir **arabelleğin(buffer memory)** boşalmasını bekler ve benzer şekilde tüketici, kullanmadan önce bir **arabelleğin(buffer memory)** dolmasını bekler. Önce MAX=1 (dizide yalnızca bir arabellek var) olduğunu hayal edelim ve bunun işe yarayıp yaramadığını görelim.

Yine iki **iş parçacığı(thread)** olduğunu düşünün, bir **üretici(producer)** ve bir **tüketici(consumer).** Tek bir CPU üzerinde belirli bir senaryoyu inceleyelim. Tüketicinin önce koşacağını varsayalım. Böylece, **tüketici(consumer)** Şekil 31.10'daki C1 Satırına basacak ve sem_wait(&full) **çağıracaktır(call).** 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 }
```

Şekil 31.9: **Koy ve Al(Put and Get) 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 }
```

Şekil 31.10: **Dolu ve Boş Koşulları Ekleme**

çağrı doluyu (-1'e) azaltır, tüketiciyi engeller ve istendiği gibi başka bir **iş parçacığının(thread)** sem_post()'u tam olarak **çağırmasını(call) bekler(wait)**.

Üreticinin(producer) daha sonra **çalıştığını(run)** varsayalım. Hat P1'e ulaşacak ve böylece sem_wait(&empty) yordamını çağıracaktır. **Tüketiciden(consumer)** farklı olarak, **üretici(producer)** bu satırdan devam edecektir çünkü boş, MAX değerine (bu durumda, 1) başlatılmıştır. Böylece boş 0'a düşürülecek ve **üretici tamponun(manufacturer buffer)** ilk girişine (Satır P2) bir **veri(data)** değeri koyacaktır. Üretici daha sonra P3'e devam edecek ve sem_post(&full) çağıracak, tam semaforun değerini -1'den 0'a değiştirecek ve **tüketiciyi(consumer)** uyandıracaktır (örn.

Bu durumda, iki şeyden biri olabilir. Yapımcı **çalışmaya(running**) devam ederse, **döngüye(loop)** girecek ve tekrar Line P1'e ulaşacaktır. Ancak bu sefer boş semaforun değeri 0 olduğu için bloke eder. Bunun yerine **üretici(producer) kesintiye(interrupt)** uğrarsa ve **tüketici (consumer)** çalışmaya başlarsa, sem_wait(&full)'den (C1 Satırı) döner, **tamponun (buffer)** dolu olduğunu bulun. , ve tüketin. Her iki durumda da, istenen davranışı elde ederiz.

Aynı örneği daha fazla **iş parçacığı(thread**) ile deneyebilirsiniz (örneğin, birden çok üretici ve birden çok tüketici). Hala çalışması gerekir.

```
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 }
```

Şekil 31.11: **Karşılıklı Ekleme Dışlama (Hatalı olarak)**

Şimdi MAX'ın 1'den büyük olduğunu hayal edelim (MAX=10 diyelim). Bu örnek için, birden çok **üretici(producer)** ve birden çok **tüketici(consumer)** olduğunu varsayalım. Şimdi bir sorunumuz var: bir **yarış durumu(race condition)**. Nerede meydana geldiğini görüyor musunuz? (biraz zaman ayırın ve arayın) Göremiyorsanız, işte size bir ipucu: put() ve get() kodlarına daha yakından bakın..

Tamam, sorunu anlayalım. İki **üreticinin(producer)** (Pa ve Pb) aşağı yukarı aynı anda put() işlevini **çağırdığını(call)** hayal edin. **Üretici** Pa'nın ilk önce **çalıştığını(run)** ve ilk **arabellek** girişini doldurmaya başladığını varsayalım (F1 Satırında fill=0). Pa, doldurma sayacını 1'e yükseltme şansı elde etmeden önce kesintiye uğrar. Üretici Pb çalışmaya başlar ve F1 Satırında ayrıca verilerini tamponun 0. elemanına koyar, bu da oradaki eski verilerin üzerine yazıldığı anlamına gelir! Bu eylem bir hayır-hayırdır; üreticiden hiçbir verinin kaybolmasını istemiyoruz.

Bir Çözüm: Karşılıklı Çıkarma Ekleme

Gördüğünüz gibi, burada unuttuğumuz karşılıklı dışlamadır. Bir **tamponun(buffer)** doldurulması ve indeksin tampona eklenmesi kritik bir bölümdür ve bu nedenle dikkatli bir şekilde korunmalıdır. Öyleyse arkadaşımız ikili semaforu kullanalım ve bazı **kilitler(locks)** ekleyelim. Şekil 31.11 girişimimizi göstermektedir. Şimdi, YENİ SATIR yorumlarında gösterildiği gibi, kodun tüm put()/get() bölümlerinin çevresine bazı kilitler ekledik. Bu doğru fikir gibi görünüyor, ama aynı zamanda işe yaramıyor. Neden? Niye? **Kilitlenme(locking).** Kilitlenme neden oluşur? Düşünmek için bir dakikanızı ayırın; kilitlenmenin ortaya çıktığı bir durum bulmaya çalışın. Programın kilitlenmesi için hangi adımlar dizisi gerçekleşmelidir?

```
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 }
```

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

Kilitlenmeden(Locking) Kaçınma

Tamam, şimdi anladığınıza göre, işte cevap. İki **iş parçacığı(thread),** bir üretici ve bir tüketici düşünün. **Tüketici(consumer)** önce koşar. Muteksi alır (C0 Satırı) ve ardından tam semaforda (C1 Satırı) sem_wait()'i **çağırır(call)**; henüz **veri(data)** olmadığı için bu çağrı, tüketicinin bloke etmesine ve dolayısıyla CPU'yu vermesine neden olur; daha da önemlisi, yine de **tüketici(consumer) kilidi(lock)** elinde tutuyor.

Ardından bir **üretici(producer)** çalışır. Üretecek verileri var ve çalışabilseydi, **tüketici(consumer**) **iş parçacığını(thread)** uyandırabilirdi ve her şey iyi olurdu. Ne yazık ki, yaptığı ilk şey, ikili muteks semaforunda (P0 Satırı) sem wait()'i **çağırmak(call)** olur. **Kilit(lock**) zaten tutuldu. Bu nedenle, **üretici(producer)** da artık **beklemek(wait)** zorunda kaldı.

Burada basit bir **döngü(loop)** var. **Tüketici(consumer)** muteksi tutar ve birinin dolu sinyali vermesini **bekler(waiting)**. Yapımcı dolu sinyali verebilir ancak muteksi **bekliyor(waiting)**. Böylece **üretici(producer)** ve **tüketici(consumer)** birbirini **beklemeye(wait)** mahkûmdur: klasik bir **kilitlenme(locking)**

Sonunda Çalışan Bir Çözüm

Bu sorunu çözmek için, basitçe **kilidin(lock)** kapsamını azaltmalıyız. Şekil 31.12 (sayfa 10) doğru çözümü göstermektedir. Gördüğünüz gibi, muteks edinmeyi ve **salıvermeyi kritik bölümün (critical section**) hemen yakınında olacak şekilde hareket ettiriyoruz;

dolu ve boş **bekleme(wait)** ve sinyal kodu dışarıda[2] bırakılır. Sonuç, çok **iş parçacıklı (thread)**programlarda yaygın olarak kullanılan bir model olan basit ve çalışan bir sınırlanmış **arabellektir(buffer memory)**. Şimdi anlayın; daha sonra kullanın. Gelecek yıllar için bize teşekkür edeceksiniz. Ya da en azından final sınavında veya bir iş görüşmesinde aynı soru sorulduğunda bize teşekkür edeceksiniz.

Okuyucu- Yazıcı Kilitleri (Reader-Writer Locks)

Başka bir klasik problem, farklı **veri yapısı(data structure)** erişimlerinin farklı **kilitleme(locking)** türleri gerektirebileceğini kabul eden daha esnek bir kilitleme **ilkel(primitive)** arzusundan kaynaklanmaktadır. Örneğin, eklemeler ve basit **çağrılar(call**) dahil olmak üzere bir **dizi(array) eşzamanlı(synchronous)** liste işlemi hayal edin. Ekler listenin durumunu değiştirirken (ve dolayısıyla geleneksel bir **kritik bölüm(critical section)** anlamlıdır), **çağrılar(call)** basitçe **veri(data)** yapısını okur; devam eden bir ekleme olmadığını garanti edebildiğimiz sürece, birçok **çağrının (call)** aynı anda ilerlemesine izin verebiliriz. Bu tür işlemleri desteklemek için şimdi geliştireceğimiz özel **kilit(lock)** türü, okuyucu-yazıcı kilidi [CHP71] olarak bilinir. Böyle bir **kilidin(lock)** kodu Şekil 31.13'te (sayfa 12) mevcuttur.

Kod oldukça basit. Eğer bir **iş parçacığı(thread)** söz konusu **veri yapısını(data structure)** güncellemek istiyorsa, yeni senkronizasyon işlemleri çiftini çağırmalıdır: bir yazma **kilidi(lock)** elde etmek için rwlock_accept_writelock() ve onu serbest bırakmak için rwlock_release_writelock(). Dahili olarak, bunlar yalnızca tek bir yazarın kilidi alabilmesini ve böylece söz konusu **veri yapısını(data structure) güncellemek(update) i**çin **kritik bölüme(critical section)** girebilmesini sağlamak için ”writelock” semaforunu kullanır.

Daha ilginç olanı, **okuma kilitlerini(read locks) almak(get)** ve serbest bırakmak için kullanılan rutin çiftidir. Bir **okuma kilidi(read locks**) edinirken, **okuyucu(reader)** önce **kilidi(lock**) alır ve ardından o anda **veri yapısının(data structure)** içinde kaç okuyucunun olduğunu izlemek için okuyucular değişkenini artırır. Daha sonra rwlock_accept_readlock() içinde atılan önemli adım, ilk **okuyucu(reader) kilidi(lock)** aldığında gerçekleşir; bu durumda okuyucu yazma kilidi semaforunda sem_wait()'i **çağırarak(call)** ve ardından sem_post()'u **çağırarak(call**) kilidi serbest bırakarak da **yazma kilidini(write locks)** alır.

Böylece, bir **okuyucu(reader)** bir kez **okuma kilidi(read locks)** edindiğinde, daha fazla okuyucunun da **okuma kilidini(read locks)** almasına izin verilecektir; ancak, **yazma kilidini(write locks)** almak isteyen herhangi bir **iş parçacığı(thread),** tüm okuyucular bitene kadar **beklemek(wait)** zorunda kalacaktır; **kritik bölümden(critical section)** son çıkan, "writelock" üzerinde sem_post()'u **çağırır(call)** ve böylece **bekleyen(wait**) bir **yazarın(writer**) **kilidi(lock)** almasını sağlar.

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

[2] Gerçekten de, muteks alma/bırakma işlevinin içine yerleştirmek daha doğal olabilirdi. modülerlik amacıyla put() ve get() işlevleri.

```
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 }
```

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

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

İPUCU: BASİT VE SAÇMA DAHA İYİ OLABİLİR (HILL YASASI)

Basit ve aptalca yaklaşımın en iyisi olabileceği fikrini asla hafife almamalısınız. **Kilitlemede(locking)**, uygulanması kolay ve hızlı olduğu için bazen basit bir döner kilit en iyi sonucu verir. **Okuyucu/yazıcı(reader\writer)** kilitleri gibi bir şey kulağa hoş gelse de karmaşıktır ve karmaşık, yavaş anlamına gelebilir. Bu nedenle, her zaman önce basit ve aptalca yaklaşımı deneyin.

Bu sadeliğe hitap etme fikri birçok yerde bulunur. Erken bir kaynak, CPU'lar için **önbelleklerin(cache)** nasıl tasarlanacağını inceleyen Mark Hill'in tezidir [H87]. Hill, basit doğrudan eşlemeli önbelleklerin, gösterişli küme ilişkilendirmeli tasarımlardan daha iyi çalıştığını buldu (bunun bir nedeni, önbelleğe almada daha basit tasarımların daha hızlı aramalara olanak sağlamasıdır). Hill'in çalışmasını kısa ve öz bir şekilde özetlediği gibi: "Büyük ve aptal daha iyidir." Ve bu yüzden bu benzer tavsiyeye Hill Yasası diyoruz.

Yemek Filozofları

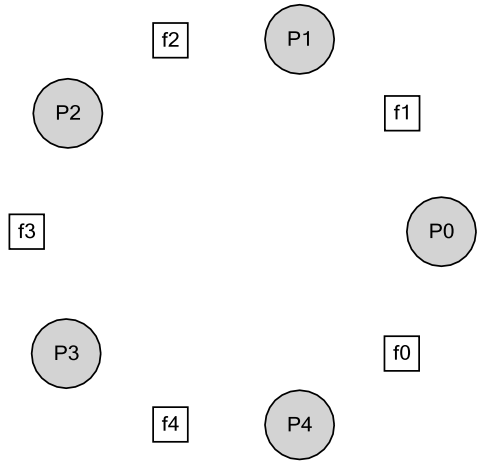
Dijkstra tarafından ortaya atılan ve çözülen en ünlü **eşzamanlılık(concurrency)** problemlerinden biri yemek filozofunun problemi [D71] olarak bilinir. Sorun, eğlenceli ve bir şekilde entelektüel açıdan ilginç olduğu için ünlüdür; ancak pratik faydası düşüktür. Ancak şöhreti onu buraya dahil etmeye zorluyor; gerçekten de, bir röportajda size bu soru sorulabilir ve bu soruyu kaçırırsanız ve işi alamazsanız OS profesörünüzden gerçekten nefret edersiniz. Tersine, eğer işi alırsanız, lütfen OS profesörünüze güzel bir not veya bazı hisse senedi seçenekleri göndermekten çekinmeyin.

Problemin temel kurulumu şudur (Şekil 31.14'te gösterildiği gibi): bir masanın etrafında oturan beş "filozof" olduğunu varsayalım. Her filozof çifti arasında tek bir çatal vardır (ve dolayısıyla toplam beş). Filozofların her birinin düşündüğü, çatala ihtiyaç duymadığı ve yemek yediği zamanlar vardır. Bir filozofun yemek yiyebilmesi için sağında ve solunda olmak üzere iki çatala ihtiyacı vardır. Bu çatallar için çekişme ve bunu takip eden senkronizasyon problemleri, bunu **eşzamanlı(synchronous)** programlamada incelediğimiz bir problem haline getiren şeydir.

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

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

O halde temel zorluk, get_forks() ve
Put_forks() koyun, böylece kilitlenme olmaz, hiçbir filozof aç kalmaz ve



Şekil 31.14: **Yemek Filozofları**

asla yemek yemez ve **eşzamanlılık(concurrency)** yüksektir (yani, mümkün olduğu kadar çok filozofun aynı anda yiyebileceği kadar).

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

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

Filozof p solundaki çatala atıfta bulunmak istediğinde, sadece left(p) adını verir. Benzer şekilde, bir filozof p'nin sağındaki çatala right(p) denilerek anılır; 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 (forks dizisindeki) 1 değerine atadığımızı varsayalım. Ayrıca her filozofun kendi sayısını (p) bildiğini varsayalım. Böylece get_forks() ve put_forks() yordamını yazabiliriz (Şekil 31.15, sayfa 15).

Bu (kırık) çözümün arkasındaki sezgi aşağıdaki gibidir. Çatalları almak için her birine bir "**kilit(lock)"** tutturuyoruz: 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 }
```

Şekil 31.15: **Get forks() Ve put forks() Rutinleri**

```
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 }
```

Şekil 31.16: **Get forks()'ta Bağımlılığı Kırmak**

ve sonra sağdaki. Yemek yemeyi bitirdiğimizde onları serbest bırakıyoruz. Basit, değil mi? Ne yazık ki, bu durumda basit, bozuk demektir. Ortaya çıkan sorunu görebiliyor musunuz? Bunu düşün.

Sorun **kilitlenme(locking).** Herhangi bir filozof sağındaki çatalı alamadan önce her filozof solundaki çatalı kaparsa, her biri sonsuza kadar bir çatalı tutarken diğerini beklemek zorunda kalır. Spesifik olarak, filozof 0 çatalı alır 0, filozof 1 çatalı alır 1, filozof
2 çatalı 2 alır, filozof 3 çatalı 3 alır ve filozof 4 çatalı 4 alır; tüm çatallar elde edildi ve tüm filozoflar, başka bir filozofun sahip olduğu bir çatalı **beklemek(wait)** zorunda kaldı. **Kilitlenmeyi(locking)** yakında daha ayrıntılı olarak inceleyeceğiz; şimdilik,
Bunun çalışan bir çözüm olmadığını söylemek güvenlidir.

Bir Çözüm: Bağımlılığı Kırmak

Bu sorunun çözümünün en basit yolu, filozoflardan en az birinin çatalları nasıl elde ettiğini değiştirmektir; gerçekten de Dijkstra'nın kendisi sorunu böyle çözdü. Spesifik olarak, filozof 4'ün (en yüksek numaralı olan) çatalları diğerlerinden farklı bir sırayla aldığını varsayalım (Şekil 31.16); put forks() kodu aynı kalır.

Son filozof soldan önce sağı kapmaya çalıştığı için, her filozofun bir çatalı kapıp diğerini **beklemesi(wait)** gibi bir durum yoktur; **bekleme(wait)** döngüsü bozuldu. Bu çözümün sonuçlarını düşünün ve işe yaradığına kendinizi ikna edin.

Bunun gibi başka "ünlü" sorunlar da var, örneğin sigara içenlerin sorunu veya uyuyan berber sorunu. Çoğu, **eşzamanlılık(concurrency)** hakkında düşünmek için bahanelerdir; bazılarının büyüleyici isimleri var. Daha fazla öğrenmekle ilgileniyorsanız veya aynı anda daha fazla düşünme pratiği yapmak istiyorsanız bunlara bakın [D08].

İplik Kısma

Semaforlar için başka bir basit kullanım durumu zaman zaman ortaya çıkar ve bu nedenle onu burada sunuyoruz. Spesifik sorun şudur: Bir programcı "çok fazla" **iş parçacığının(thread)** aynı anda bir şeyler yapmasını ve sistemi tıkamasını nasıl önleyebilir? Yanıt: "çok fazla" için bir eşik belirleyin ve ardından söz konusu **kod parçasını(piece of code)**aynı anda yürüten **iş parçacığının(thread)** sayısını sınırlamak için bir semafor kullanın. Bu yaklaşımı **kısma(throttling**) [T99] olarak adlandırıyoruz ve bunu bir kabul denetimi biçimi olarak görüyoruz.

Daha spesifik bir örnek ele alalım. Bazı problemler üzerinde paralel olarak çalışmak için yüzlerce **iş parçacığı(thread)** oluşturduğunuzu hayal edin. Bununla birlikte, kodun belirli bir bölümünde, her **iş parçacığı(thread)**, hesaplamanın bir bölümünü gerçekleştirmek için büyük miktarda **bellek(memory)** edinir; kodun bu kısmına hafıza yoğun bölge diyelim. Tüm **iş parçacıkları(threads)** aynı anda yoğun bellek bölgesine girerse, tüm **bellek(memory)** ayırma isteklerinin toplamı makinedeki fiziksel **bellek(memory)** miktarını aşacaktır. Sonuç olarak, makine çalışmaya başlayacak (yani, sayfaları diske ve diskten değiştirecek) ve tüm hesaplama bir taramaya yavaşlayacaktır.

Basit bir semafor bu sorunu çözebilir. Semaforun değerini, **belleği(memory**) yoğun bölgeye bir kerede girmek istediğiniz maksimum **iş parçacığı(thread)** sayısına ayarlayarak ve ardından bölgenin etrafına bir sem_wait() ve sem_post() koyarak, bir semafor, kodun tehlikeli bölgesinde **eşzamanlı(synchronous)** olarak bulunan **iş parçacığınınthread)**sayısını doğal olarak azaltabilir.

Semaforlar Nasıl Uygulanır?

Son olarak, düşük seviyeli senkronizasyon **ilkellerimizi(primitive)**, **kilitleri(lock)** ve **koşul değişkenlerimizi(condition veriables)** kullanarak ... (burada davul sesi) ... Zemafor adı verilen kendi semafor sürümümüzü oluşturalım. Şekil 31.17'de (sayfa 17) görebileceğiniz gibi, bu görev oldukça basittir.

Yukarıdaki kodda, semaforun değerini izlemek için yalnızca bir **kilit(lock)** ve bir **koşul değişkeni (condition veriables)** ile bir **durum değişkeni(state veriables)** kullanıyoruz. Gerçekten anlayana kadar kodu kendiniz çalışın. Yap!

Dijkstra tarafından tanımlandığı şekliyle Zemaforumuz ve saf semaforlarımız arasındaki ince bir fark, semaforun değerinin negatif olduğunda, **bekleyen(waiting) iş parçacığının(thread)** sayısını yansıttığı değişmezini korumamamızdır; aslında, değer asla sıfırdan düşük olmayacaktır. Bu davranışın uygulanması daha kolaydır ve mevcut Linux uygulamasıyla eşleşir.

```
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

Şekil 31.17: **Kilitler ve Özgeçmişlerle Semaforları Uygulamak**

Merakla, semaforlardan **koşul değişkenleri(condition veriables)** oluşturmak çok daha hileli bir önermedir. Bazı son derece deneyimli **eşzamanlı(synchronous)** programcılar bunu Windows ortamında yapmaya çalıştı ve birçok farklı hata ortaya çıktı [B04]. Kendiniz deneyin ve semaforlardan **durum değişkenleri(state veriables)** oluşturmanın bir problem için neden göründüğünden daha zorlayıcı olduğunu anlayıp anlayamayacağınıza bakın.

Özet

Semaforlar, **eşzamanlı(synchronous)** programlar yazmak için güçlü ve esnek bir **ilkeldir(primitive**). Bazı programcılar, basitlikleri ve kullanışlılıkları nedeniyle **kilitleri(locks**) ve **koşul değişkenlerini(condition veriables)** göz ardı ederek bunları özel olarak kullanırlar.
Bu bölümde, sadece birkaç klasik problem ve çözüm sunduk. Daha fazlasını öğrenmekle ilgileniyorsanız, başvurabileceğiniz birçok başka malzeme vardır. Harika (ve ücretsiz referanslardan) biri, Allen Downey'nin **eşzamanlılık(concurrency)** ve semaforlarla programlama üzerine kitabıdır [D08]. Bu kitap, anlayışınızı geliştirmek için üzerinde çalışabileceğiniz birçok bulmaca içeriyor

İPUCU: GENELLEME KONUSUNDA DİKKATLİ OLUN

Soyut genelleme tekniği, iyi bir fikrin biraz daha genişletilebildiği ve böylece daha geniş bir problem sınıfını çözebildiği sistem tasarımında oldukça yararlı olabilir. Ancak genelleme yaparken dikkatli olun; Lampson'ın bizi uyardığı gibi "Genelleme yapmayın; genellemeler genellikle yanlıştır" [L83].

Semaforlar, kilitlerin ve durum değişkenlerinin bir genellemesi olarak görülebilir; ancak böyle bir genellemeye gerek var mı? Ve bir semaforun üzerinde bir koşul değişkeni gerçekleştirmenin zorluğu göz önüne alındığında, belki de bu genelleme sandığınız kadar genel değildir.

özel olarak her iki semaforun ve genel olarak eş zamanlı olarak kullanılması. Gerçek bir **eşzamanlılık(concurrency)** uzmanı olmak yıllarca çaba gerektirir; Bu derste öğrendiklerinizin ötesine geçmek, hiç şüphesiz böyle bir konuya hakim olmanın anahtarıdır.

References

[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 Sunon 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-writerproblem, 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 instruc- tions ... presumably the OS implementors worked out the appropriate algorithms, although Dijkstra isgenerally 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(concurrency)** problemlerini çözmek için semaforları kullanacağız. Bunların çoğu, Downey'nin mükemmel "Little Book of Semaphores"3 kitabından alınmıştır; ilgilenen okuyucular daha fazla eğlence için Küçük Kitap'a göz atmalıdır.

Aşağıdaki soruların her biri bir kod iskeleti sağlar; işiniz verilen semaforların 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(locks)** ve koşul değişkenlerini kullanarak). İyi şanlar!

Sorular

1. 1. İlk problem, metinde açıklandığı gibi çatal/birleştirme problemine bir çözüm uygulamak ve test etmektir. Bu çözüm metinde açıklanmış olsa da, kendi başınıza yazmanız faydalı olacaktır; Bach bile Vivaldi'yi yeniden yazar, müstakbel ustalardan birinin var olandan bir şeyler öğrenmesini sağlardı. Ayrıntılar için fork-join.c'ye bakın. Çalıştığından emin olmak için çocuğa uyku(1) çağrısını ekleyin.

2. Şimdi buluşma problemini inceleyerek bunu biraz genelleştirelim.Sorun şu: her biri yaklaşık olan iki ileti diziniz var;kodda buluşma noktasını girmek için. İkisi de bu kısımdan çıkmamalıdiğeri girmeden önceki kod. Bunun için iki semafor kullanmayı düşününgöreve bakın ve ayrıntılar için rendezvous.c'ye bakın.

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. Göreviniz: yazmak
bu şekilde kullanılabilecek bir bariyer() işlevini uygulayan kod. N'yi (çalışan programdaki toplam iş parçacığı sayısı) bildiğinizi ve tüm N iş parçacığının bariyere girmeye çalışacağını varsaymak güvenlidir. Yine, çözüme ulaşmak için muhtemelen iki semafor ve şeyleri saymak için başka tamsayılar kullanmalısınız. Ayrıntılar için bariyer.c'ye bakın.

4. Şimdi yine metinde anlatıldığı gibi okuyucu-yazar problemini çözelim. Bu ilk çekimde, açlık konusunda endişelenmeyin. Ayrıntılar için reader-writer.c'deki koda bakın. Beklediğiniz gibi çalıştığını göstermek için kodunuza sleep() çağrıları ekleyin. Açlık sorununun varlığını gösterebilir misiniz?

5. Okur-yazar sorununa tekrar bakalım ama bu kez açlıktan endişe edin. Tüm okuyucuların ve yazarların sonunda ilerleme kaydetmesini nasıl sağlayabilirsiniz? Ayrıntılar için reader-writer-nostarve.c'ye bakın

6. Muteksi elde etmeye çalışan herhangi bir iş parçacığının sonunda onu elde edeceği aç kalmayan bir muteks oluşturmak için semaforları kullanın. Daha fazla bilgi için mutex-nostarve.c'deki koda bakın.

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]Available: `http://greenteapress.com/semaphores/downey08semaphores.pdf`.