
SİVAS CUMHURİYET ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

- İşletim Sistemleri -

2019141068 – Sedat DİKBAŞ

## İÇİNDEKİLER TABLOSU

# SEMAFORE

Şimdi bildiğimiz gibi, çok çeşitli ilgili ve ilginç eşzamanlılık problemlerini çözmek için hem kilitlere hem de koşul değişkenlerine ihtiyaç vardır. Bunu yıllar önce farkeden ilk kişilerden biri Edsger Dijkstra'ydı (tam tarihi bilmek zor olsa da [GR92]), diğer şeylerin yanı sıra, grafik teorisindeki ünlü "en kısa yollar" algoritmasıyla [D59], erken bir polemik olarak biliniyordu. "Zararlı Olarak Kabul Edilen İfadelere Git" [D68a] (ne harika bir başlık!) başlıklı yapılandırılmış programlama ve burada inceleyeceğimiz durumda, semafor [D68b, D72] olarak adlandırılan bir senkronizasyon ilkesinin tanıtımı. Gerçekten de, Dijkstra ve meslektaşları semaforu senkronizasyonla ilgili her şey için tek bir ilkel olarak icat ettiler; göreceğiniz gibi, semaforlar hem kilit hem de koşul değişkenleri olarak kullanılabilir.

CRUX: SEMAFOR NASIL KULLANILIR 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?

Semafor, iki rutinle işleyebileceğimiz bir tamsayı değerine sahip bir nesnedir; POSIX standardında bu rutinler sem wait() ve sem post()1 şeklindedir. . Semaforun başlangıç değeri davranışını belirlediğinden, semaforla etkileşim kurmak için başka bir rutini çağırmadan önce, Şekil'deki kod gibi, onu bir değere başlatmamız gerekir. 31.1 yapar.

1Tarihsel olarak, sem wait()'e Dijkstra tarafından P() ve sem post()'a V() adı verildi. Bu kısaltılmış formlar Hollandaca kelimelerden gelir; ilginç bir şekilde, sözde türettikleri Hollandaca kelimelerin zaman içinde değişmesi. Orijinal olarak, P() "pas geçmek"ten (geçmek) ve V() "vrijgave"den (serbest bırakmak) geliyordu; Daha sonra Dijkstra, P()'nin "prolaag"dan, "probeer" (Hollandaca "denemek" ve "verlaag" ("azalmak") sözcüklerinin bir kısaltmasından ve V()'nin "artış" anlamına gelen "verhoog"dan geldiğini yazdı. Bazen, insanlar onları aşağı ve yukarı çağırır. Arkadaşlarınızı etkilemek, onların kafasını karıştırmak veya her ikisini birden yapmak için Hollandaca versiyonları kullanın. Ayrıntılar için bkz. https://news.ycombinator.com/item?id=8761539).

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

**Şekil 1Bir 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 süreçteki 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, 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 veya daha yüksekti) ya da arayanın sonraki bir 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.

```c
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

**Şekil 2 Semafor: Bekle ve Gönderin Tanımları**

```c
sem_t m;
sem_init(&m, 0, X); // initialize to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

**Şekil 3 İkili Semafor (Yani, Bir Kilit)**

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

## BİNARY SEMAPHORES (LOCKS)( İKİLİ SEMAFORLAR (KİLİTLER))

Artık bir semafor kullanmaya hazırız. İlk kullanımımız zaten aşina olduğumuz bir kullanım olacak: bir semaforu kilit olarak kullanmak. Bir kod parçacığı için Şekil 31.3'e bakın; 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 tanımına 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 olarak değiştirecektir. Ardından, yalnızca değer 0'dan büyük veya 0'a eşit değilse bekleyecektir. Değer 0 olduğundan, sem wait() basitçe geri dönecek ve çağıran iş parçacığı devam edecek; 0 parçacığı artık kritik bölüme girmek için serbesttir. 0 iş parçacığı 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, Thread 0 "kilidi tuttuğunda" (yani, sem wait()'i çağırmış ancak henüz sem post()'u çağırmamış) ve başka bir thread (Thread 1) sem wait'i çağırarak kritik bölüme girmeye çalıştığında ortaya çıkar. (). Bu durumda, Thread 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 4: Thread Trace: 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 |

**Şekil 5: Bir Semafor Kullanan İki Thread**

bu nedenle bekleyin (kendini uykuya sokar ve işlemciyi bırakır). İş parçacığı 0 tekrar çalıştığında, sonunda sem post()'u çağırır, semaforun değerini tekrar sıfıra yükseltir ve ardından kilidi kendisi için alabilecek olan bekleyen iş parçacığını uyandırır (İş parçacığı 1). Thread 1 bittiğinde, semaforun değerini tekrar artıracak ve tekrar 1'e geri yükleyecektir.

Şekil 31.5, bu örneğin 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 engellendi). Halihazırda tutulan kilidi almaya çalıştığında Thread 1'in uyku durumuna geçtiğini unutmayın; sadece Thread 0 tekrar çalıştığında Thread 1 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.

## SİPARİŞ İÇİN SEMAFORLAR

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

```
sem_t s;

void *child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

**Şekil 6: Parent'ini bekleyen child process**

böylece ondan bir öğe silebilir. Bu kullanım modelinde, genellikle bir iş parçacığının bir şeyin olmasını beklediğini ve başka bir iş parçacığının bir şeyin olmasını sağladığını ve ardından bunun gerçekleştiğini bildirdiğini ve böylece bekleyen iş parçacığını uyandırdığını görürüz. 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, parent basitçe sem wait()'i ve alt sem post()'u çağırarak çocuğun 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 parentin yarattığını, ancak çocuğun henüz koşmadığını (yani, hazır bir kuyrukta oturuyor ama koşmuyor) varsayalım. Bu durumda (Şekil 31.7, sayfa 6), child sem post()'u çağırmadan önce parent sem wait()'i arayacaktır; parentin çocuğun koşmasını beklemesini isteriz. Bunun olmasının tek yolu, semaforun değerinin 0'dan büyük olmamasıdır; dolayısıyla 0 başlangıç değeridir. Parent çalışır, semaforu azaltır (-1'e), sonra bekler (uyku). Child 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: **Thread Trace: Parent Waiting For Child (Case 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 7: Thread Trace: Child'ini bekleyen Parent**

semaforu 0'a getirin ve parenti uyandırın, bu daha sonra sem wait() işlevinden dönecek ve programı bitirecektir. İkinci durum (Şekil 31.8), parent sem wait()'i çağırma şansı bulamadan child tamamlamaya çalıştığında ortaya çıkar. Bu durumda, child önce sem post()'u çağırır, böylece semaforun değeri 0'dan 1'e çıkar. 1 olmak; parent bu nedenle değeri (0'a) düşürür ve beklemeden sem wait() işlevinden döner ve ayrıca istenen efekti elde eder.

## ÜRETİCİ/TÜKETİCİ (BOUNDED BUFFER)SORUNU

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

### ASIDE: BİR SEMAFORUN DEĞERİNİ BELİRLEMEK

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 kaynakların 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 çıkarılır. 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 semafor sunar; bunlar, iş parçacıklarının sırasıyla bir arabellek girişinin ne zaman boşaltıldığını veya doldurulduğunu belirtmek için kullanır. 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) 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 }
```

**Şekil 8 The 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 9: 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 (örneğin, bloke durumundan hazır'a hareket ettirerek) sem post(&full) öğesini çağıracaktır.

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 engellenir. Bunun yerine üretici kesintiye uğrar ve tüketici çalışmaya başlarsa, sem wait(&full) (Satır C1)'den döner, arabelleğin dolduğunu bulur ve tüketir. 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 }
```

**Şekil 10: Karşılıklı Dışlama Ekleme (Incorrectly)**

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, konuyu anlayalım. İki üreticinin (Pa ve Pb) put()'u aşağı yukarı aynı anda 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 bir 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, NEW LINE yorumlarında belirtildiği gibi, kodun tüm put()/get() bölümlerinin etrafına bazı kilitler ekledik. Bu doğru fikir gibi görünüyor, ama aynı zamanda işe yaramıyor. Neden? Niye? Kilitlenme. 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 11:Karşılıklı Dışlama Ekleme (Correctly)**

## DEADLOCK'TAN KAÇINMAK

Kilitlenmeden Kaçınmak Tamam, şimdi anladın, işte cevap. İki iş parçacığı düşünün, bir üretici ve bir tüketici. Tüketici önce koşar. Muteksi (Satır C0) alır ve ardından tam semaforda (Satır C1) sem wait()'i çağırır; henüz veri olmadığı için, bu çağrı tüketicinin bloke etmesine ve böylece CPU'yu vermesine neden olur; daha da önemlisi, tüketici hala kilidi elinde tutuyor.

Bir yapımcı daha sonra çalışır. Üretecek verileri var ve çalışabilseydi, tüketici zincirini uyandırabilirdi ve her şey iyi olurdu. Ne yazık ki, yaptığı ilk şey ikili muteks semaforunda (Line P0) sem wait()'i çağırmaktır. Kilit zaten tutuluyor. Dolayısıyla yapımcı da artık beklemekte.

Burada basit bir döngü var. Tüketici muteksi tutar ve birinin dolu sinyali vermesini bekler. Üretici dolu sinyali verebilir ancak muteks için bekliyor. Böylece üretici ve tüketici birbirini beklemek zorunda kalır: klasik bir çıkmaz.

**Sonunda, Çalışan Bir Çözüm**

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

Full ve boş bekleme(empty) ve sinyal(signal) kodu dışarıda bırakılır2 . Sonuç, çok iş parçacıklı programlarda yaygın olarak kullanılan bir kalıp olan basit ve çalışan sınırlı bir arabellektir.

Anlayın artık; 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.

## READER-WRİTER LOCKS(OKUYUCU VE YAZICI KLİTLENMESİ)

Başka bir klasik sorun, farklı veri yapısı erişimlerinin farklı türde kilitlemeler gerektirebileceğini kabul eden daha esnek bir kilitleme ilkesine duyulan istekten kaynaklanmaktadır. Örneğin, eklemeler ve basit aramalar dahil olmak üzere bir dizi eşzamanlı liste işlemi hayal edin. Ekler listenin durumunu değiştirirken (ve dolayısıyla geleneksel bir kritik bölüm mantıklıdır), aramalar sadece veri yapısını okur; hiçbir eklemenin devam etmediğini garanti edebildiğimiz sürece, birçok aramanın aynı anda devam etmesine izin verebiliriz. Bu tür bir işlemi desteklemek için şimdi geliştireceğimiz özel kilit türü, okuyucu-yazıcı kilidi [CHP71] olarak bilinir. Böyle bir kilidin kodu Şekil 31.13'te (sayfa 12) mevcuttur.

Kod oldukça basit. Eğer bir iş parçacığı söz konusu veri yapısını güncellemek isterse, yeni senkronizasyon işlemi çiftini çağırmalıdır: bir yazma kilidi elde etmek için rwlock writelock(), ve onu serbest bırakmak için rwlock release writelock(). Dahili olarak, bunlar sadece tek bir yazarın kilidi alabilmesini sağlamak için writelock semaforunu kullanır ve böylece söz konusu veri yapısını güncellemek için kritik bölüme girer.

Daha ilginç olanı, okuma kilitlerini almak ve serbest bırakmak için bir çift rutindir. Bir okuma kilidi alırken, okuyucu önce kilidi alır ve ardından veri yapısı içinde şu anda kaç okuyucu olduğunu izlemek için okuyucu değişkenini artırır. Daha sonra rwlock get readlock() içinde atılan önemli adım, ilk okuyucu kilidi aldığında gerçekleşir; bu durumda, okuyucu ayrıca writelock semaforunda sem wait()'i çağırarak ve ardından sem post()'u çağırarak kilidi serbest bırakarak yazma kilidini elde eder.

Böylece, bir okuyucu bir kez bir okuma kilidi edindiğinde, daha fazla okuyucunun da okuma kilidini almasına izin verilecektir; ancak, yazma kilidini almak isteyen herhangi bir iş parçacığı, tüm okuyucular bitene kadar beklemek zorunda kalacaktır; kritik bölümden en son çıkan, "writelock" üzerinde sem post()'u çağırır ve böylece bekleyen bir yazarın kilidi almasını sağlar. Bu yaklaşım (istendiği gibi) işe yarar, ancak özellikle adalet söz konusu olduğunda bazı olumsuzlukları vardır. Özellikle, okuyucuların yazarları aç bırakması nispeten kolay olurdu. Bu soruna daha sofistike çözümler mevcuttur; belki daha iyi bir uygulama düşünebilirsiniz? İpucu: Bir yazar

beklerken daha fazla okuyucunun kilide girmesini önlemek için ne yapmanız gerektiğini düşünün.

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // allow ONE writer/MANY readers
  int   readers;   // #readers in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

**Şekil 12: Reader-Writer Kilitlenmesi**

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

### İPUCU: BASİT VE DUMB DAHA İYİ OLABİLİR (HILL'S LAW)

Basit ve aptal yaklaşımın en iyisi olabileceği fikrini asla küçümsememelisiniz. Kilitleme ile bazen basit bir döndürme kilidi en iyi sonucu verir, çünkü uygulaması kolay ve hızlıdır. Okuyucu/yazar kilitleri gibi bir şey kulağa hoş gelse de, bunlar karmaşıktır ve karmaşık, yavaş anlamına gelebilir. Bu nedenle, her zaman önce basit ve aptal yaklaşımı deneyin. Sadeliğe hitap eden bu fikir birçok yerde bulunur. İlk 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 çalışmasını kısaca özetlediği gibi: "Büyük ve aptal daha iyidir." Ve böylece buna benzer tavsiye diyoruz Hill Yasası.

Dijkstra tarafından ortaya atılan ve çözülen en ünlü eşzamanlılık problemlerinden biri, yemek filozofunun problemi olarak bilinir [D71]. Sorun ünlüdür çünkü eğlencelidir ve entelektüel açıdan biraz ilginçtir; bununla birlikte, pratik faydası düşüktür. Ancak ünü buraya dahil edilmesini zorunlu kılıyor; gerçekten de, bir röportajda size sorulabilir ve bu soruyu kaçırırsanız ve işi alamazsanız, işletim sistemi profesörünüzden gerçekten nefret edersiniz. Tersine, işi alırsanız, lütfen işletim sistemi profesörünüze güzel bir not veya bazı hisse senedi seçenekleri göndermekten çekinmeyin.

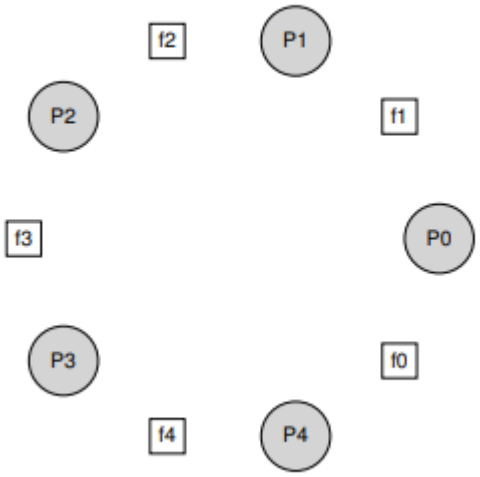
Problemin temel kurulumu şudur (Şekil 31.14'te gösterildiği gibi): Bir masanın etrafında oturan beş "filozof" olduğunu varsayalım. Her bir filozof çifti arasında tek bir çatal (dolayısıyla toplam beş) vardır.

Filozofların her birinin düşündükleri ve çatala ihtiyaç duymadıkları zamanları ve yemek yedikleri zamanları vardır. Bir filozofun yemek yiyebilmesi için hem solunda hem de sağında olmak üzere iki çatala ihtiyacı vardır. Bu çatallar için çekişme ve ortaya çıkan senkronizasyon sorunları, bunu eşzamanlı programlamada incelediğimiz bir sorun haline getiriyor.

Her birinin 0 ila 4 (dahil) arasında benzersiz bir p parçacığı tanımlayıcısı olduğunu varsayarak, her filozofun temel döngüsü:

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

O halde kilit zorluk, get forks() rutinlerini yazmak ve forks() koymak, böylece hiçbir kilitlenme, hiçbir filozof açlığı ve




**Şekil 13: The Dining Philosophers (Yemek Yiyen Filozoflar)**

asla yemek yiyemez ve eşzamanlılık yüksektir (yani, mümkün olduğunca çok filozof aynı anda yiyebilir). Downey'nin çözümlerini [D08] izleyerek, bizi bir çözüme götürmek için birkaç yardımcı fonksiyon kullanacağız. Bunlar:

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

Filozof p, sollarındaki çatala atıfta bulunmak istediğinde, sadece sol(p) derler. Benzer şekilde, bir filozofun sağındaki çatala p, sağ(p); buradaki modulo operatörü, son filozofun (p=4) sağındaki çatalı, yani çatal 0'ı yakalamaya 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 çatal[5].

**Broken Solution(Kırık Çözüm)**

Soruna ilk çözümümüzü denedik. başlattığımızı varsayalım her semaforu (forks dizisinde) 1 değerine getirin. Ayrıca her filozofun kendi numarasını (p) bildiğini varsayın. Böylece get forks() ve put forks() rutinini yazabiliriz (Şekil 31.15, sayfa 15). Bu (kırık) çözümün arkasındaki sezgi aşağıdaki gibidir. Çatalları almak için her birine bir "kilit" 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  }
```

Figure 31.15: **The get_forks() And put_forks() Routines**

```
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 14: Get forks()'ta Bağımlılığı Kırmak**

ve sonra sağdaki. Yemek yemeyi bitirdiğimizde onları serbest bırakırız. Basit, hayır? Ne yazık ki, bu durumda, basit kırık anlamına gelir. Ortaya çıkan sorunu görebiliyor musunuz? Bunu düşün. Sorun kilitlenme. Herhangi bir filozof sağındaki çatalı tutmadan önce her filozof sol tarafındaki çatalı tutarsa, her biri sonsuza kadar bir çatalı tutup diğerini beklemek zorunda kalacaktır. Spesifik olarak, filozof 0 çatal 0'ı, filozof 1 çatal 1'i, filozof 2 çatal 2'yi, filozof 3 çatal 3'ü ve filozof 4 çatal 4'ü tutuyor; tüm çatallar elde edilir ve tüm filozoflar, başka bir filozofun sahip olduğu çatalı beklerken sıkışıp kalır. Kilitlenmeyi yakında daha ayrıntılı olarak inceleyeceğiz; şimdilik bunun işe yarayan bir çözüm olmadığını söyleyebiliriz.

**Bir Çözüm: Bağımlılığı Kırmak**

Bu soruna saldırmanın en basit yolu, en az bir filozof tarafından çatalların nasıl elde edildiğini değiştirmektir; gerçekten, Dijkstra'nın kendisi sorunu bu şekilde çözmüştür. Spesifik olarak, 4. filozofun (en yüksek numaralı) çatalları diğerlerinden farklı bir sırada aldığını varsayalım (Şekil 31.16); put forks() kodu aynı kalır. Son filozof soldan hemen önce kapmaya çalıştığı için, her filozofun bir çatal alıp diğerini beklerken takıldığı bir durum yoktur; bekleme döngüsü bozuldu. Bu çözümün sonuçlarını düşünün ve işe yaradığına kendinizi ikna edin.

Bunun gibi başka "ünlü" sorunlar da var, örneğin sigara sigara içen kişinin sorunu ya da uyuyan berber sorunu. Çoğu sadece eşzamanlılık hakkında düşünmek için bahaneler; bazılarının ilginç isimleri var. Daha fazlasını öğrenmek ya da sadece eşzamanlı bir şekilde düşünme pratiği yapmakla ilgileniyorsanız onlara bakın [D08].

## THREAD THROTTLİNG(İPLİK KISMA)

Ara sıra semaforlar için başka bir basit kullanım durumu ortaya çıkar ve bu nedenle onu burada sunuyoruz. Spesifik sorun şudur: Bir programcı "çok fazla" iş parçacığının aynı anda bir şeyler yapmasını ve sistemi tıkamasını nasıl önleyebilir? Cevap: "çok fazla" için bir eşiğe karar verin ve ardından söz konusu kod parçasını aynı anda yürüten iş parçacığı sayısını sınırlamak için bir semafor kullanın. Bu yaklaşıma kısma [T99] diyoruz ve bunu bir kabul denetimi biçimi olarak görüyoruz.

Daha spesifik bir örnek düşünelim. Paralel olarak bir problem üzerinde çalışmak için yüzlerce iş parçacığı oluşturduğunuzu hayal edin. Ancak, kodun belirli bir bölümünde, her bir iş parçacığı, hesaplamanın bir bölümünü gerçekleştirmek için büyük miktarda bellek alır; Kodun bu kısmına hafıza yoğun bölge diyelim. Tüm iş parçacıkları aynı anda yoğun bellek kullanan bölgeye girerse, tüm bellek ayırma isteklerinin toplamı makinedeki fiziksel bellek miktarını aşacaktır. Sonuç olarak, makine çökmeye başlayacak (yani, sayfaları diske ve diskten takas ederek) ve tüm hesaplama yavaşlamaya başlayacaktır.

Basit bir semafor bu sorunu çözebilir. Semaforun değerini, bellek yoğun bölgeye bir kerede girmek istediğiniz maksimum iş parçacığı sayısına başlatarak ve ardından bölgeye bir sem wait() ve sem post() koyarak, bir semafor doğal olarak sayısını azaltabilir. aynı anda kodun tehlikeli bölgesinde bulunan iş parçacıkları.

## SEMAFORLAR NASIL UYGULANIR

Son olarak, kendi semafor sürümümüzü oluşturmak için düşük seviyeli senkronizasyon temel öğelerimizi, kilitlerimizi ve koşul değişkenlerimizi kullanalım. (davul burada) ... Zemaforlar.

Bu görev, Şekil 31.17'de (sayfa 17) görebileceğiniz gibi oldukça basittir. Yukarıdaki kodda, semaforun değerini izlemek için sadece bir kilit ve bir koşul değişkeni artı bir durum değişkeni kullanıyoruz. Kodu gerçekten anlayana kadar kendiniz inceleyin. Yap!

Zemaforumuz ile Dijkstra tarafından tanımlanan saf semaforlar arasındaki ince bir fark, semaforun değerinin negatif olduğunda bekleyen iş parçacığı sayısını yansıttığı değişmezini korumamamızdır; gerçekten, 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 15: Kilitli ve CV'li Semaforları Gerçekleştirme**

Tuhaf bir şekilde, semaforlardan koşul değişkenleri oluşturmak çok daha zor bir önermedir. Bazı çok deneyimli eşzamanlı programcılar bunu Windows ortamında yapmaya çalıştı ve birçok farklı hata ortaya çıktı [B04]. Kendiniz deneyin ve semaforlardan koşul değişkenleri oluşturmanın neden göründüğünden daha zor bir problem olduğunu çözüp çözemeyeceğinize bakın.

## ÖZET

Semaforlar, eşzamanlı programlar yazmak için güçlü ve esnek bir ilkeldir. Bazı programcılar, basitlikleri ve kullanışlılıkları nedeniyle bunları yalnızca kilitleri ve koşul değişkenlerini dışlayarak kullanırlar. Bu bölümde, sadece birkaç klasik problem ve çözüm sunduk. Daha fazlasını öğrenmekle ilgileniyorsanız, başvurabileceğiniz başka birçok materyal 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.

İPUCU: GENELLEME YAPARKEN DİKKATLİ OLUN

Soyut genelleme tekniği, iyi bir fikrin biraz daha genişletilebildiği ve böylece daha büyük bir problem sınıfını çözebildiği sistem tasarımında oldukça faydalı olabilir. Ancak genelleme

yaparken dikkatli olun; Lampson'ın bizi uyardığı gibi "Genelleme yapmayın; genellemeler genellikle yanlıştır" [L83]. Semaforları, kilitlerin ve koşul değişkenlerinin bir genellemesi olarak görebiliriz; ama böyle bir genellemeye gerek var mı? Ve bir semaforun üstünde bir koşul değişkenini 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.

## REFERANSLAR


**[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, iyi bilinen bazı eşzamanlılık problemlerini çözmek için semaforları kullanacağız. Bunların çoğu, Downey'nin bir dizi klasik problemi bir araya getirmenin yanı sıra birkaç yeni varyant sunma konusunda iyi bir iş çıkaran mükemmel "Little Book of Semaphores"3'ten alınmıştır; ilgilenen okuyucular daha fazla eğlence için Küçük Kitap'a göz atmalıdır.

Aşağıdaki soruların her biri bir kod iskeleti sağlar; senin işin verilen semaforların çalışmasını sağlamak için kodu doldurmak. Linux'ta yerel semaforlar kullanacaksınız; Mac'te (semafor desteğinin olmadığı durumlarda), önce bir uygulama oluşturmanız gerekir (bölümde açıklandığı gibi kilitleri ve koşul değişkenlerini kullanarak). İyi şanlar!

## 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 birinden öğrenmesine izin verecekti. Ayrıntılar için fork-join.c'ye bakın. Çalıştığından emin olmak için çocuğa uyku (1) çağrısını ekleyin.

```
s = sem_open("/sem", O_CREAT, S_IRWXU, 0);
```

2. Randevu problemini inceleyerek bunu biraz genelleştirelim. Sorun şu: 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'ye bakın.

```
$ gcc -o rendezvous rendezvous.c -Wall -pthread

$ ./rendezvous
parent: begin
child 2: before
child 1: before
child 1: after
child 2: after
parent: end
```

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: bu şekilde kullanılabilecek bir bariyer() işlevi uygulamak için kodu yazın. N'yi (çalışan programdaki toplam iş parçacığı sayısı) bildiğinizi ve ardından tüm N iş parçacığının bariyere girmeye çalışacağını varsaymak güvenlidir. Yine, çözümü elde etmek için muhtemelen iki semafor ve şeyleri saymak için başka tamsayılar kullanmalısınız. Ayrıntılar için bariyer.c'ye bakın.

```
$ gcc -o barrier barrier.c -Wall -pthread -g

$ ./barrier 5
parent: begin
child 0: before
child 2: before
child 1: before
child 4: before
child 3: before
child 0: after
child 2: after
child 1: after
child 4: after
child 3: after
parent: end
```

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

```
$ gcc -o reader-writer reader-writer.c -Wall -pthread -g
$ ./reader-writer 4 2 3
begin
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
read 0
write 1
write 2
write 3
write 4
write 5
write 6
end: value 6
```

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

Bir veritabanının birkaç eşzamanlı işlem arasında paylaşılacağını varsayalım. Bu süreçlerin bir kısmı sadece veritabanını okumak isteyebilirken, diğerleri veritabanını güncellemek (yani okuyup yazmak) isteyebilir. Birincisine okuyucular, ikincisine de yazarlar olarak atıfta bulunarak bu iki tür süreci birbirinden ayırıyoruz. Açıkçası, iki okuyucu paylaşılan verilere aynı anda erişirse, hiçbir olumsuz etki ortaya çıkmayacaktır. Bununla birlikte, bir yazar ve başka bir işlem (okuyucu veya yazar) veritabanına aynı anda erişirse, kaos ortaya çıkabilir.

Bu zorlukların ortaya çıkmamasını sağlamak için, yazarların veri tabanına yazarken paylaşılan veri tabanına özel erişime sahip olmalarını şart koşuyoruz. Bu eşitleme sorununa okuyucu-yazar sorunu denir.

Okur-yazar sorununun, hepsi de öncelikleri içeren çeşitli varyasyonları vardır. Birinci okuyucu-yazar problemi olarak anılan en basit problem, bir yazar paylaşılan nesneyi kullanmak için izin almadıkça hiçbir okuyucunun bekletilmemesini gerektirir. Başka bir deyişle, hiçbir okuyucu sırf bir yazar bekliyor diye diğer okuyucuların bitirmesini beklememelidir.

6. Muteksi elde etmeye çalışan herhangi bir iş parçacığının sonunda onu elde edeceği, açlıktan ölmeyen bir muteks oluşturmak için semaforları kullanın. Daha fazla bilgi için mutex-nostarve.c içindeki koda bakın.

```
while True:
  mutex.wait()
  # critical section
  mutex.signal()
```

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?

https://greenteapress.com/wp/semaphores/