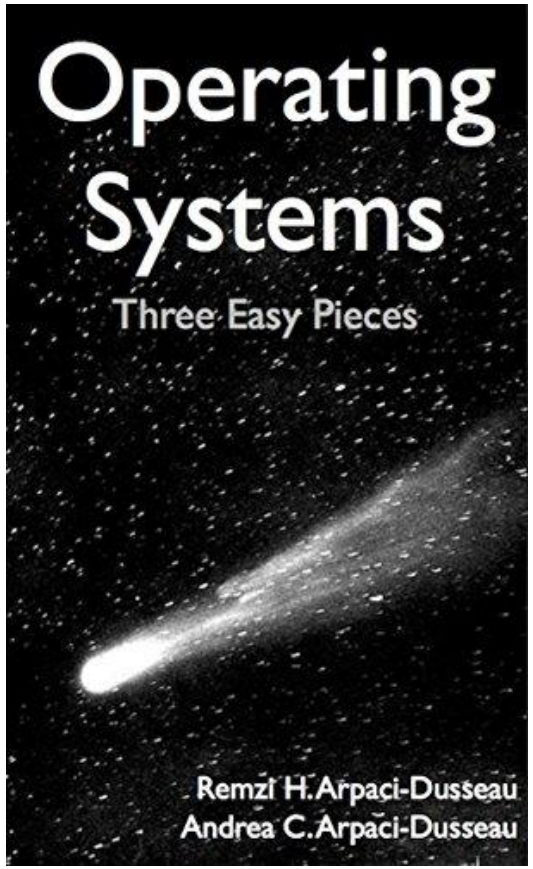
İşletim Sistemleri



8. Ders

Prof. Dr. Kemal Bıçakcı

Alıştırma Sorusu

- Hangisi thread API kullanılırken yapılan programlama hatalarından biri değildir?

a) Kilitleri ve durum değişkenlerini ilklendirmemek

b) Dönüş kodlarını (Return codes) kontrol etmemek

c) İş parçacıkları tarafından döndürülen yerel değişkenler için bellek ayırmayı Yığın (Stack) üzerinde yapmak.

d) İş parçacıkları arasında sinyal vermek için durum değişkenleri kullanmak

29. Kilit-tabanlı Eş Zamanlı Veri Yapıları

Operating System: Three Easy Pieces

Kilit-tabanlı Eş Zamanlı Veri Yapıları

- Bir veri yapısına kilit eklemek, bu veri yapısını iş parçacığı güvenli (thread safe) hale getirir.
- Kilitlerin nasıl eklendiği ise veri yapısının hem doğruluğunu hem de performansını belirler.

Örnek:
Kilit Bulunmayan Sayaç (Counter)

- Basit eş zamanlı olmayan sayaç:

```
1       typedef struct __counter_t {
2               int value;
3       } counter_t;
4
5       void init(counter_t *c) {
6               c->value = 0;
7       }
8
9       void increment(counter_t *c) {
10              c->value++;
11      }
12
13      void decrement(counter_t *c) {
14              c->value--;
15      }
16
17      int get(counter_t *c) {
18              return c->value;
19      }
```

Örnek:
Kilit Kullanan Sayaç (Counter)

- Nasıl «thread safe» hale getirebiliriz?: Bir kilit ekleyerek.
- Kilit, veri yapısını değiştiren herhangi bir rutin çağrılırken elde edilmektedir.

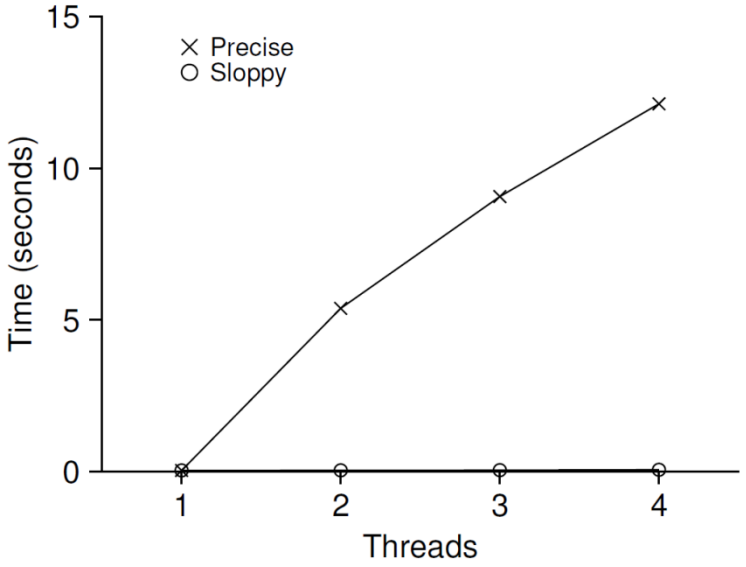
```
1       typedef struct __counter_t {
2               int value;
3               pthread_lock_t lock;
4       } counter_t;
5
6       void init(counter_t *c) {
7               c->value = 0;
8               Pthread_mutex_init(&c->lock, NULL);
9       }
10
11      void increment(counter_t *c) {
12              Pthread_mutex_lock(&c->lock);
13              c->value++;
14              Pthread_mutex_unlock(&c->lock);
15      }
16
```

Örnek:
Kilit Kullanan Sayaç (Counter)(Devam)

```
(Cont.)
17      void decrement(counter_t *c) {
18              Pthread_mutex_lock(&c->lock);
19              c->value--;
20              Pthread_mutex_unlock(&c->lock);
21      }
22
23      int get(counter_t *c) {
24              Pthread_mutex_lock(&c->lock);
25              int rc = c->value;
26              Pthread_mutex_unlock(&c->lock);
27              return rc;
28      }
```

Basit yaklaşımın performans maliyetleri

- Her iş parçacığı, paylaşılan bir sayacı günceller.
  - Güncelleme: bir milyon kez.
  - Dört Intel 2.7GHz i5 CPU'lu iMac



Performance of Traditional vs. Sloppy Counters
(Threshold of Sloppy, S, is set to 1024)

Senkronize (Eşzamanlı) sayacın ölçeklenmesi iyi değildir.

Sayacın Ölçeklenebilirliği: Gerçekten önemli midir?

Mükemmel Ölçeklenebilirlik

- Daha fazla iş yapılsa da paralel olarak yapıldığı için görevi tamamlamak için harcanan süre artmaz.

Özensiz sayaç (Sloppy counter)

- Özensiz sayaç, CPU çekirdeği başına bir tane olmak üzere çok sayıda yerel fiziksel sayaç ve tek bir genel sayaç aracılığıyla tek bir mantıksal sayacı temsil ederek çalışır.
- Kilitlerin durumu:
  - Her yerel sayaç için bir tane ve genel sayaç için bir tane.
  - Örnek: dört CPU'lu bir makinede dört yerel sayaç ve bir genel sayaç

Özensiz sayacın temel fikri

- Bir çekirdek (CPU) üzerinde çalışan bir iş parçacığı sayacı artırmak istediğinde:
  - Yerel sayacını artırır.
  - Her CPU'nun kendi yerel sayacı vardır:
  - CPU'lardaki iş parçacıkları, yerel sayaçları birbiriyle çakışma olmadan güncelleyebilir.
  - Böylece sayaç güncellemeleri ölçeklenebilir hale gelir.
- Lokal değerler periyodik olarak global sayaca aktarılır.
  - Genel kilit edinilir.
  - Global sayacın değeri yerel sayacın değeri kadar artırılır.
  - Yerel sayaç sıfırlanır.
  - Genel kilit bırakılır.

Özensiz sayacın temel fikri (Devam)

- Yerelden genele aktarımın ne sıklıkta gerçekleşeceği bir eşik değer ile belirlenir: S (sloppiness – özensizlik parametresi).
- Daha Küçük S:
  - Sayaç, ölçeklenemeyen sayaç gibi davranır.
- Daha Büyük S:
  - Sayaç daha ölçeklenebilir olur.
  - Öte yandan, genel sayaç değeri gerçek değerden daha fazla uzaklaşabilir.

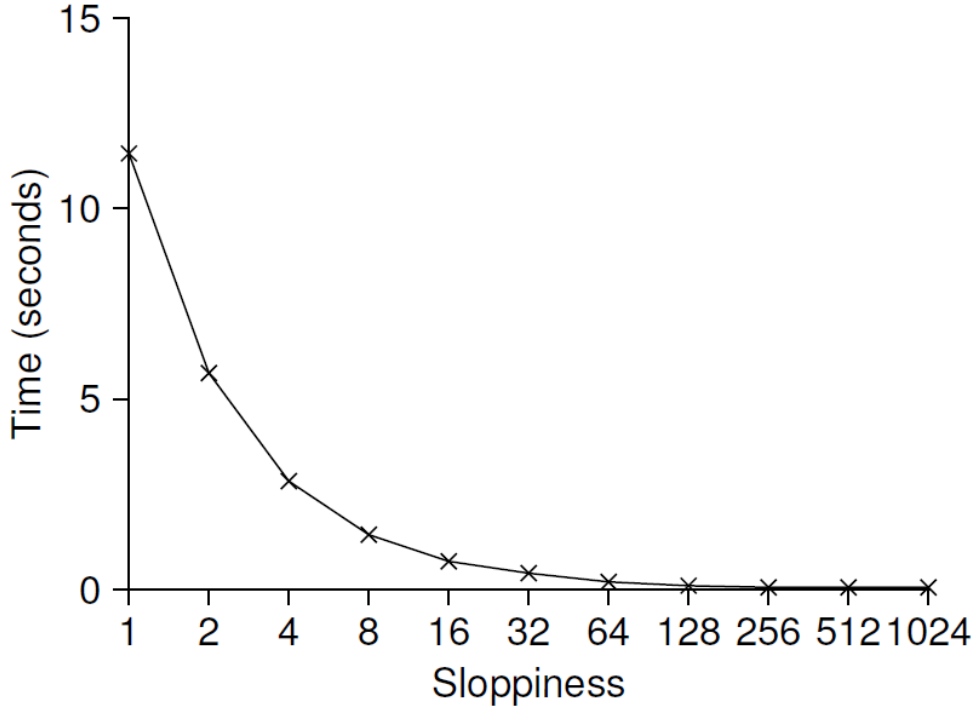
Örnek

- Özensiz Sayacın çalışmasını izlersek:
  - S = 5
  - İş parçacığı sayısı: 4
  - CPU (çekirdek) sayısı: 4

Time	$\mathbf{L_1}$	$\mathbf{L_2}$	$\mathbf{L_3}$	$\mathbf{L_4}$	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

S Eşik Değerin Önemi

- Her dört iş parçacığı, dört CPU'da da sayacı 1 milyon kez artırır.
  - Düşük S → Performans kötü, genel sayaç doğru değere çok yakın.
  - Yüksek S → Performance mükemmel, genel sayaç doğru değerden çok geride olabilir.



Scaling Sloppy Counters

Implementasyon

```
1       typedef struct __counter_t {
2           int global;             // global count
3           pthread_mutex_t glock;  // global lock
4           int local[NUMCPUS];     // local count (per cpu)
5           pthread_mutex_t llock[NUMCPUS]; // ... and locks
6           int threshold;          // update frequency
7       } counter_t;
8
9       // init: record threshold, init locks, init values
10      //       of all local counts and global count
11      void init(counter_t *c, int threshold) {
12          c->threshold = threshold;
13
14          c->global = 0;
15          pthread_mutex_init(&c->glock, NULL);
16
17          int i;
18          for (i = 0; i < NUMCPUS; i++) {
19              c->local[i] = 0;
20              pthread_mutex_init(&c->llock[i], NULL);
21          }
22      }
23
```

Implementasyon (Devam)

(Cont.)

```
24      // update: usually, just grab local lock and update local amount
25      //         once local count has risen by 'threshold', grab global
26      //         lock and transfer local values to it
27      void update(counter_t *c, int threadID, int amt) {
28          pthread_mutex_lock(&c->llock[threadID]);
29          c->local[threadID] += amt;      // assumes amt > 0
30          if (c->local[threadID] >= c->threshold) { // transfer to global
31              pthread_mutex_lock(&c->glock);
32              c->global += c->local[threadID];
33              pthread_mutex_unlock(&c->glock);
34              c->local[threadID] = 0;
35          }
36          pthread_mutex_unlock(&c->llock[threadID]);
37      }
38
39      // get: just return global amount (which may not be perfect)
40      int get(counter_t *c) {
41          pthread_mutex_lock(&c->glock);
42          int val = c->global;
43          pthread_mutex_unlock(&c->glock);
44          return val;    // only approximate!
45      }
```

Eşzamanlı Bağlı Listeler (Concurrent Linked Lists)

```
1       // basic node structure
2       typedef struct __node_t {
3               int key;
4               struct __node_t *next;
5       } node_t;
6
7       // basic list structure (one used per list)
8       typedef struct __list_t {
9               node_t *head;
10              pthread_mutex_t lock;
11      } list_t;
12
13      void List_Init(list_t *L) {
14              L->head = NULL;
15              pthread_mutex_init(&L->lock, NULL);
16      }
17
(Cont.)
```

Eşzamanlı Bağlı Listeler (Devam)

```
(Cont.)
18      int List_Insert(list_t *L, int key) {
19              pthread_mutex_lock(&L->lock);
20              node_t *new = malloc(sizeof(node_t));
21              if (new == NULL) {
22                      perror("malloc");
23                      pthread_mutex_unlock(&L->lock);
24                      return -1; // fail
26              new->key = key;
27              new->next = L->head;
28              L->head = new;
29              pthread_mutex_unlock(&L->lock);
30              return 0; // success
31      }
(Cont.)
```

Eşzamanlı Bağlı Listeler (Devam)

```
(Cont.)
32
32      int List_Lookup(list_t *L, int key) {
33              pthread_mutex_lock(&L->lock);
34              node_t *curr = L->head;
35              while (curr) {
36                      if (curr->key == key) {
37                              pthread_mutex_unlock(&L->lock);
38                              return 0; // success
39                      }
40                      curr = curr->next;
41              }
42              pthread_mutex_unlock(&L->lock);
43              return -1; // failure
44      }
```

Eşzamanlı Bağlı Listeler (Devam)

- Ekleme (Insert) rutinine giriş sonrasında kilit elde edilir.
- Kod, çıkışta kilidi serbest bırakır.
- *malloc()* başarısız olursa, ekleme başarısız sonucu döndürülmeden önce kilidin de serbest bırakılması gerekir.
- Bu tür istisnai kontrol akışının (exceptional control flow) oldukça hata eğilimli olduğu bilinmektedir.
- Çözüm:
  - Kilitleme ve serbest bırakma, ekleme kodunda yalnızca gerçek kritik bölgeyi çevrelesin.
  - İstisnai kontrol akışı kullanılmasın.

Eşzamanlı Bağlı Listeler: Yeniden Yazalım

```
1       void List_Init(list_t *L) {
2               L->head = NULL;
3               pthread_mutex_init(&L->lock, NULL);
4       }
5
6       void List_Insert(list_t *L, int key) {
7               // synchronization not needed
8               node_t *new = malloc(sizeof(node_t));
9               if (new == NULL) {
10                      perror("malloc");
11                      return;
12              }
13              new->key = key;
14
15              // just lock critical section
16              pthread_mutex_lock(&L->lock);
17              new->next = L->head;
18              L->head = new;
19              pthread_mutex_unlock(&L->lock);
20      }
21
```

Eşzamanlı Bağlı Listeler: Yeniden Yazalım (Devam)

```
(Cont.)
22      int List_Lookup(list_t *L, int key) {
23              int rv = -1;
24              pthread_mutex_lock(&L->lock);
25              node_t *curr = L->head;
26              while (curr) {
27                      if (curr->key == key) {
28                              rv = 0;
29                              break;
30                      }
31                      curr = curr->next;
32              }
33              pthread_mutex_unlock(&L->lock);
34              return rv; // now both success and failure
35      }
```

Bağlı Listenin Ölçeklenebilirliği

- Listenin tamamı için tek bir kilide sahip olmak yerine, listenin her bir düğümü başına bir kilit eklenebilir mi?
  - Listede gezinirken, önce bir sonraki düğümün kilidi alınır.
  - Ardından geçerli düğümün kilidi serbest bırakılır.
  - Bu sayede, liste işlemlerinde yüksek derecede eşzamanlılık sağlanır.
- Bu çözümün pratikte uygulanmasının önünde listde gezinirken her düğüm için kilit alma ve serbest bırakmanın getirdiği ek yükler engelleyicidir.

Eşzamanlı Kuyruk (Michael ve Scott Çözümü)

- İki kilit var
  - Sıranın başı için bir tane
  - Kuyruk için bir tane
- Bu iki kilidin amacı, kuyruğa alma (enqueue) ve kuyruktan çıkarma (dequeue) işlemlerinin eşzamanlılığını sağlamaktır.
- Sahte düğüm ekleme:
  - İlklendirme kodunda eklenir.
  - Baş ve kuyruk işlemlerinin birbirinden ayrılmasını sağlar.

Eşzamanlı Kuyruk (Devam)

```
1       typedef struct __node_t {
2               int value;
3               struct __node_t *next;
4       } node_t;
5
6       typedef struct __queue_t {
7               node_t *head;
8               node_t *tail;
9               pthread_mutex_t headLock;
10              pthread_mutex_t tailLock;
11      } queue_t;
12
13      void Queue_Init(queue_t *q) {
14              node_t *tmp = malloc(sizeof(node_t));
15              tmp->next = NULL;
16              q->head = q->tail = tmp;
17              pthread_mutex_init(&q->headLock, NULL);
18              pthread_mutex_init(&q->tailLock, NULL);
19      }
20
(Cont.)
```

Eşzamanlı Kuyruk (Devam)

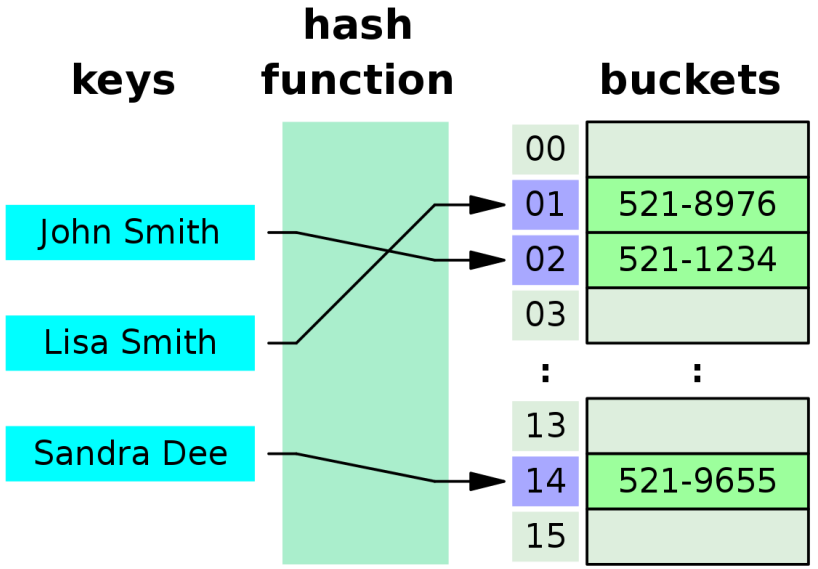
```
(Cont.)
21      void Queue_Enqueue(queue_t *q, int value) {
22              node_t *tmp = malloc(sizeof(node_t));
23              assert(tmp != NULL);
24
25              tmp->value = value;
26              tmp->next = NULL;
27
28              pthread_mutex_lock(&q->tailLock);
29              q->tail->next = tmp;
30              q->tail = tmp;
31              pthread_mutex_unlock(&q->tailLock);
32      }
(Cont.)
```

Eşzamanlı Kuyruk (Devam)

```
(Cont.)
33      int Queue_Dequeue(queue_t *q, int *value) {
34              pthread_mutex_lock(&q->headLock);
35              node_t *tmp = q->head;
36              node_t *newHead = tmp->next;
37              if (newHead == NULL) {
38                      pthread_mutex_unlock(&q->headLock);
39                      return -1; // queue was empty
40              }
41              *value = newHead->value;
42              q->head = newHead;
43              pthread_mutex_unlock(&q->headLock);
44              free(tmp);
45              return 0;
46      }
```

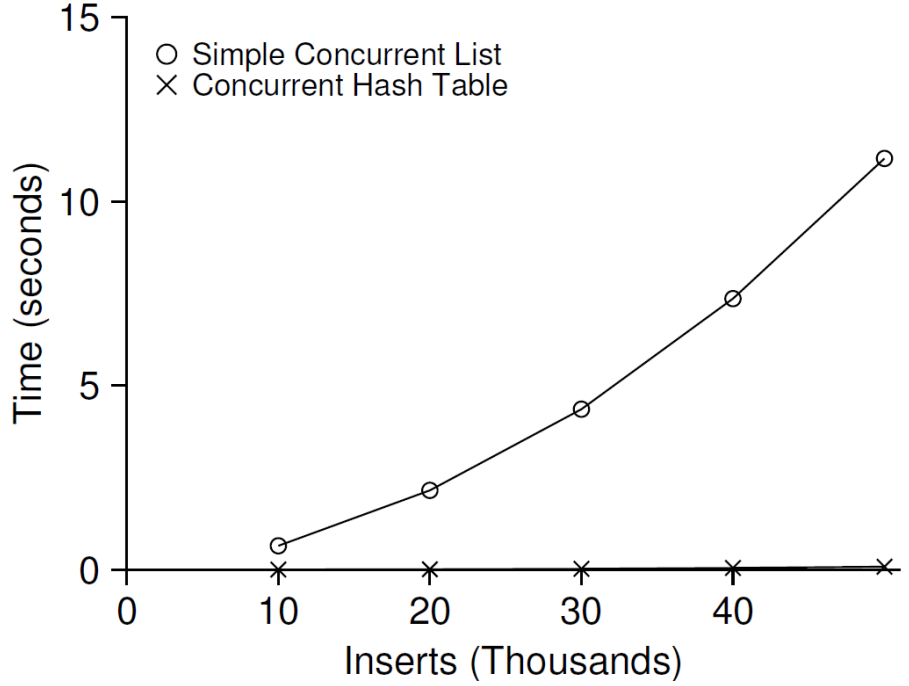
Eşzamanlı Anahtarlı Tablo (Concurrent Hash Table)

- Basit bir anahtarlı tabloya odaklanıyoruz.
  - Tablo yeniden boyutlandırılmaz.
  - Eşzamanlı listeler kullanılarak oluşturulur.
  - Her biri bir liste ile temsil edilen kova (bucket) başına bir kilit kullanılır.



Eşzamanlı Anahtarlı Tablonun Performansı

- Dört iş parçacığının her birinden 10.000 - 50.000 eşzamanlı güncelleme.
  - Dört adet Intel 2.7GHz i5 CPU'lu iMac.



Basit eşzamanlı anahtarlı tablonun ölçeklenebilirliği oldukça yüksektir.

Eşzamanlı Anahtarlı Tablo

```
1       #define BUCKETS (101)
2
3       typedef struct __hash_t {
4               list_t lists[BUCKETS];
5       } hash_t;
6
7       void Hash_Init(hash_t *H) {
8               int i;
9               for (i = 0; i < BUCKETS; i++) {
10                      List_Init(&H->lists[i]);
11              }
12      }
13
14      int Hash_Insert(hash_t *H, int key) {
15              int bucket = key % BUCKETS;
16              return List_Insert(&H->lists[bucket], key);
17      }
18
19      int Hash_Lookup(hash_t *H, int key) {
20              int bucket = key % BUCKETS;
21              return List_Lookup(&H->lists[bucket], key);
22      }
```

30. Koşul Değişkenleri (Condition Variables)

Operating System: Three Easy Pieces

Koşul (Durum) Değişkenleri (Condition Variables)

- Bir iş parçacığının, çalışmaya devam etmeden önce bir koşulun doğru olup olmadığını kontrol etmek istediği birçok durum vardır.
- Örnek:Bir ana iş parçacığı, bir çocuk iş parçacığının tamamlanıp tamamlanmadığını kontrol etmek isteyebilir.
- Bunu genellikle *join()* ile yapar.

Koşul Değişkenleri (Devam)

Çocuğun bitmesini bekleyen Ana

```
1       void *child(void *arg) {
2           printf("child\n");
3           // XXX how to indicate we are done?
4           return NULL;
5       }
6
7       int main(int argc, char *argv[]) {
8           printf("parent: begin\n");
9           pthread_t c;
10          Pthread_create(&c, NULL, child, NULL); // create child
11          // XXX how to wait for child?
12          printf("parent: end\n");
13          return 0;
14      }
```

Ne görmeyi bekliyoruz?:

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

Çocuğu Bekleyen Ana: Dönmeye Dayalı Yaklaşım

```
1       volatile int done = 0;
2
3       void *child(void *arg) {
4           printf("child\n");
5           done = 1;
6           return NULL;
7       }
8
9       int main(int argc, char *argv[]) {
10          printf("parent: begin\n");
11          pthread_t c;
12          Pthread_create(&c, NULL, child, NULL); // create child
13          while (done == 0)
14              ; // spin
15          printf("parent: end\n");
16          return 0;
17      }
```

- Ana iş parçacığı dönerek CPU zamanını boşa harcadığı için oldukça verimsizdir.

Bir Koşul için nasıl beklenir?

- Koşul değişkeni (Condition variable)

- (Bir koşulu) Bekleme (Wait):
  - Belirli bir çalışma koşulu istenildiği gibi olmadığında iş parçacıklarının kendilerini koyabilecekleri ve uyuyabilecekleri bir kuyruk var.
- (Bir koşula ait) Sinyal verme (Signal):
  - Başka bir iş parçacığı, söz konusu koşulu değiştirdiğinde, bekleyen iş parçacıklarından birini uyandırabilir ve devam etmelerine izin verebilir.

Tanımlar ve Rutinler

- Koşul değişkeni tanımlama

```
pthread_cond_t c;
```

  - Aynı zamanda uygun şekilde ilklendirilir.

- (POSIX) çağrıları:

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);    // wait()
pthread_cond_signal(pthread_cond_t *c);                      // signal()
```

  - wait() çağrısı argüman olarak bir mutex değişkeni de alır.
    - wait() çağrısı kilidi serbest bırakır ve çağıran iş parçacığını uyku moduna geçirir.
    - İş parçacığı uyandığında kilidi yeniden alması gerekir.

Çocuğu bekleyen Ana
(Bir koşul değişkeni ile)

```
1       int done = 0;
2       pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3       pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5       void thr_exit() {
6               Pthread_mutex_lock(&m);
7               done = 1;
8               Pthread_cond_signal(&c);
9               Pthread_mutex_unlock(&m);
10      }
11
12      void *child(void *arg) {
13              printf("child\n");
14              thr_exit();
15              return NULL;
16      }
17
18      void thr_join() {
19              Pthread_mutex_lock(&m);
20              while (done == 0)
21                      Pthread_cond_wait(&c, &m);
22              Pthread_mutex_unlock(&m);
23      }
24
```

Çocuğu bekleyen Ana
(Bir koşul değişkeni ile) (Devam)

```
(cont.)
25      int main(int argc, char *argv[]) {
26              printf("parent: begin\n");
27              pthread_t p;
28              Pthread_create(&p, NULL, child, NULL);
29              thr_join();
30              printf("parent: end\n");
31              return 0;
32      }
```

Çocuğu bekleyen Ana
(Bir koşul değişkeni ile) (Devam)

- **Ana:**
  - Çocuk iş parçacığını oluştur ve çalışmaya devam et.
  - Çocuk iş parçacığının tamamlanmasını beklemek için thr_join() çağrısı yap.
  - Kilidi al.
  - Çocuğun bitip bitmediğini kontrol et.
  - *wait()*'i çağırarak kendini uyku moduna geçir.
  - Kilidi serbest bırak.
- **Çocuk:**
  - *"Child"* mesajını yaz.
  - Ana iş parçacığını uyandırmak için thr_exit() çağrısı yap.
  - Kilidi elde et.
  - *"done"* değişkenini ayarla.
  - Ana iş parçacığına sinyal göndererek onu uyandır.

Done durum değişkeninin önemi

```
1       void thr_exit() {
2               Pthread_mutex_lock(&m);
3               Pthread_cond_signal(&c);
4               Pthread_mutex_unlock(&m);
5       }
6
7       void thr_join() {
8               Pthread_mutex_lock(&m);
9               Pthread_cond_wait(&c, &m);
10              Pthread_mutex_unlock(&m);
11      }
```

`thr_exit()` and `thr_join()` without variable `done`

- Çocuğun hemen çalıştığı durumu düşünelim.
- Çocuk sinyal verecek, ancak koşulda uyuyan iş parçacığı henüz yok.
- Ana iş parçacığı çalıştığında, beklemeye ve uyumaya başlayacak. Uyandıracak iş parçacığı da mevcut değil.

Diğer bir kötü implementasyon

```
1       void thr_exit() {
2               done = 1;
3               Pthread_cond_signal(&c);
4       }
5
6       void thr_join() {
7               if (done == 0)
8                       Pthread_cond_wait(&c);
9       }
```

- Buradaki sorun: ince bir yarış durumu.
- Ana, *thr_join()* çağrısı yapar, *«done»* değerini kontrol eder ve 0 olduğunu görür. Uyumaya çalışır.
- Bekle ve uyumaya gitme çağrısından hemen önce, kesme olur ve çocuk çalışmaya başlar.
- Çocuk *«done»* durum değişkenini 1 olarak değiştirir ve sinyal verir .Ancak hiçbir iş parçacığı beklememektedir ve dolayısıyla hiçbir iş parçacığı uyandırılmaz.
- Ana tekrar çalıştığında, sonsuza kadar uyuyacaktır.

Üretici/ Tüketici
(Sınırlı Arabellek) (Bounded Buffer) Problemi

- **Üretici:**
  - Veri öğeleri üretir.
  - Veri öğelerini bir arabelleğe yerleştirmek ister.
- **Tüketici:**
  - Veri öğelerini arabellekten alır.
  - Onları bir şekilde tüketir.
- **Örnek:**
  - Çok iş parçacıklı web sunucusu: Bir üretici, HTTP isteklerini bir iş kuyruğuna koyar. Tüketici ise istekleri bu kuyruktan alır ve işler.

Sınırlı Arabellek (Tampon Bellek)

- Bir programın çıktısını diğerine **«pipe»** ile aktardığınızda, sınırlı bir arabellek kullanılır.
- Örnek: `grep foo file.txt | wc -l`
  - `grep` işlemi üreticidir.
  - `wc` işlemi ise tüketicidir.
  - Aralarında kernel'da yönetilen bir sınırlı arabellek vardır.
- Sınırlı arabellek paylaşılan kaynaktır → Erişimin senkronize edilmesi gerekir.

Put ve Get Rutinleri (Versiyon 1)

```
1       int buffer;
2       int count = 0;    // initially, empty
3
4       void put(int value) {
5               assert(count == 0);
6               count = 1;
7               buffer = value;
8       }
9
10      int get() {
11              assert(count == 1);
12              count = 0;
13              return buffer;
14      }
```

- Arabellek büyüklüğü 1’dir.
- Yalnızca count 0 olduğunda arabelleğe veri koyulur (arabellek boş olduğunda).
- Yalnızca count 1 olduğunda arabellekten veri alınır (arabellek dolduğunda).

Üretici/Tüketici İş parçacıkları (Versiyon 1)

```
1       void *producer(void *arg) {
2               int i;
3               int loops = (int) arg;
4               for (i = 0; i < loops; i++) {
5                       put(i);
6               }
7       }
8
9       void *consumer(void *arg) {
10              int i;
11              while (1) {
12                      int tmp = get();
13                      printf("%d\n", tmp);
14              }
15      }
```

- Üretici, paylaşılan arabelleğe bir döngü içinde bir tamsayı değerini pek çok kez koyar.
- Tüketici, verileri bu paylaşılan arabellekten alır.

Üretici/Tüketici:
Bir tane Koşul Değişkeni ve If İfadesi

- **Sadece bir tane `cond` durum değişkeni ve ilgili `mutex` kilidi**

```
1       cond_t cond;
2       mutex_t mutex;
3
4       void *producer(void *arg) {
5           int i;
6           for (i = 0; i < loops; i++) {
7               Pthread_mutex_lock(&mutex);             // p1
8               if (count == 1)                         // p2
9                   Pthread_cond_wait(&cond, &mutex);   // p3
10              put(i);                                 // p4
11              Pthread_cond_signal(&cond);             // p5
12              Pthread_mutex_unlock(&mutex);           // p6
13          }
14      }
15
16      void *consumer(void *arg) {
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);             // c1
```

Üretici/Tüketici:
Bir tane Koşul Değişkeni ve If İfadesi (Devam)

```
20              if (count == 0)                          // c2
21                  Pthread_cond_wait(&cond, &mutex);    // c3
22              int tmp = get();                         // c4
23              Pthread_cond_signal(&cond);              // c5
24              Pthread_mutex_unlock(&mutex);            // c6
25              printf("%d\n", tmp);
26          }
27      }
```

- p1-p3: Üretici arabelleğin boşalmasını bekler.
- c1-c3: Tüketici arabelleğin dolmasını bekler.
- Tek bir üretici ve tek bir tüketici ile kod doğru çalışır.

Ya birden fazla üreticimiz ve/veya tüketicimiz varsa?

Hatalı Çözümün İzlenmesi (Versiyon 1)

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

Hatalı Çözümün İzlenmesi (Versiyon 1)

- Sorun basit bir nedenden dolayı ortaya çıkmaktadır.
- Üretici T_c1'i uyandırır, ancak T_c1 çalıştırılmadan önce, sınırlı arabelleğin durumu T_c2 tarafından değiştirilir.
- Uyandırılan iş parçacığı çalıştığında, durumun hala istenildiği gibi olacağının garantisi yoktur → Mesa semantiği.
- Hemen hemen şimdiye kadarki her sistem Mesa semantiğini kullanır.
- Alternatif: Hoare semantiğinde ise uyandırılan iş parçacığının uyandırıldıktan hemen sonra çalışacağına dair daha güçlü bir garanti sağlanır.

Üretici/Tüketici:
Bir tane Koşul Değişkeni ve While İfadesi

- If ifadesini While ile değiştirelim.
- Tüketici T_c1 uyanır ve paylaşılan değişkenin durumunu yeniden kontrol eder. Arabellek boşsa, tüketici tekrar uykuya geri döner.

```
1       cond_t cond;
2       mutex_t mutex;
3
4       void *producer(void *arg) {
5           int i;
6           for (i = 0; i < loops; i++) {
7               Pthread_mutex_lock(&mutex);             // p1
8               while (count == 1)                      // p2
9                   Pthread_cond_wait(&cond, &mutex);   // p3
10              put(i);                                 // p4
11              Pthread_cond_signal(&cond);             // p5
12              Pthread_mutex_unlock(&mutex);           // p6
13          }
14      }
15
```

Üretici/Tüketici:
Bir tane Koşul Değişkeni ve While İfadesi

```
(Cont.)
16      void *consumer(void *arg) {
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);             // c1
20              while (count == 0)                      // c2
21                  Pthread_cond_wait(&cond, &mutex);   // c3
22              int tmp = get();                        // c4
23              Pthread_cond_signal(&cond);             // c5
24              Pthread_mutex_unlock(&mutex);           // c6
25              printf("%d\n", tmp);
26          }
27      }
```

- Durum değişkenleriyle ilgili hatırlanması gereken basit bir kural: her zaman while kullan.
- Ancak, bu kodda hala bir hata var (sonraki slayt).

Hatalı Çözümün İzlenmesi (Versiyon 2)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	**Oops! Woke T_{c2}**

Hatalı Çözümün İzlenmesi (Versiyon 2)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
…	…	…	…	…	…	…	(*cont.*)
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	**Everyone asleep …**

- Bir tüketici diğer tüketicileri değil, sadece üreticileri uyandırmalıdır ve bunun tersi de geçerlidir.

Üretici/Tüketici Problemi Çözümü (Arabellek Büyüklüğü = 1)

- while ile birlikte iki koşul değişkeni kullanılır
  - **Üretici** `empty` koşulunda bekler and `fill` değişkenine sinyal gönderir
  - **Tüketici** `fill` koşulunda bekler and `empty` değişkenine sinyal gönderir

```
1       cond_t empty, fill;
2       mutex_t mutex;
3
4       void *producer(void *arg) {
5           int i;
6           for (i = 0; i < loops; i++) {
7               Pthread_mutex_lock(&mutex);
8               while (count == 1)
9                   Pthread_cond_wait(&empty, &mutex);
10              put(i);
11              Pthread_cond_signal(&fill);
12              Pthread_mutex_unlock(&mutex);
13          }
14      }
15
```

Üretici/Tüketici Problemi Çözümü (Arabellek Büyüklüğü = 1) (Devam)

```
(Cont.)
16      void *consumer(void *arg) {
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);
20              while (count == 0)
21                  Pthread_cond_wait(&fill, &mutex);
22              int tmp = get();
23              Pthread_cond_signal(&empty);
24              Pthread_mutex_unlock(&mutex);
25              printf("%d\n", tmp);
26          }
27      }
```

Üretici/Tüketici Problemi Çözümü (Arabellek Büyüklüğü = MAX)

- Arabellek büyüklüğü artarsa eşzamanlılık ve verimlilik de artar.
- Eşzamanlı üretmek ve tüketmek mümkün olur.
- «context switch» sayısı azalır.

```
1       int buffer[MAX];
2       int fill = 0;
3       int use = 0;
4       int count = 0;
5
6       void put(int value) {
7           buffer[fill] = value;
8           fill = (fill + 1) % MAX;
9           count++;
10      }
11
12      int get() {
13          int tmp = buffer[use];
14          use = (use + 1) % MAX;
15          count--;
16          return tmp;
17      }
```

The Final Put and Get Routines

Üretici/Tüketici Problemi Çözümü (Arabellek Büyüklüğü = MAX) (Devam)

```
1       cond_t empty, fill;
2       mutex_t mutex;
3
4       void *producer(void *arg) {
5           int i;
6           for (i = 0; i < loops; i++) {
7               Pthread_mutex_lock(&mutex);             // p1
8               while (count == MAX)                    // p2
9                   Pthread_cond_wait(&empty, &mutex);  // p3
10              put(i);                                 // p4
11              Pthread_cond_signal(&fill);             // p5
12              Pthread_mutex_unlock(&mutex);           // p6
13          }
14      }
15
16      void *consumer(void *arg) {
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);             // c1
20              while (count == 0)                      // c2
21                  Pthread_cond_wait(&fill, &mutex);   // c3
22              int tmp = get();                        // c4
```

Üretici/Tüketici Problemi Çözümü (Arabellek Büyüklüğü = MAX) (Devam)

```
(Cont.)
23              Pthread_cond_signal(&empty);                    // c5
24              Pthread_mutex_unlock(&mutex);                   // c6
25              printf("%d\n", tmp);
26          }
27      }
```

The Final Working Solution (Cont.)

- p2: **Üretici, yalnızca tüm arabellek doluysa uyur.**
- c2: **Tüketici, yalnızca tüm arabellek boşsa uyur.**

Tüm Durumları Kapsamak

- Belleğin tamamen dolu olduğunu varsayalım.
  - T_a çağrısı: `allocate(100)`.
  - T_b çağrısı: `allocate(10)`.
  - Hem T_a hem de T_b koşulda bekler ve uyurlar.
  - T_c çağrısı: `free(50)`.

Hangi bekleyen iş parçacığı uyandırılmalıdır?

Tüm Durumları Kapsamak (Devam)

```
1       // how many bytes of the heap are free?
2       int bytesLeft = MAX_HEAP_SIZE;
3
4       // need lock and condition too
5       cond_t c;
6       mutex_t m;
7
8       void *
9       allocate(int size) {
10          Pthread_mutex_lock(&m);
11          while (bytesLeft < size)
12              Pthread_cond_wait(&c, &m);
13          void *ptr = ...;                // get mem from heap
14          bytesLeft -= size;
15          Pthread_mutex_unlock(&m);
16          return ptr;
17      }
18
19      void free(void *ptr, int size) {
20          Pthread_mutex_lock(&m);
21          bytesLeft += size;
22          Pthread_cond_signal(&c);        // whom to signal??
23          Pthread_mutex_unlock(&m);
24      }
```

Tüm Durumları Kapsamak (Devam)

- Çözüm (Lampson ve Redell tarafından önerilmiştir)
  - `pthread_cond_signal()` çağrısını `pthread_cond_broadcast()` ile değiştir.

- `pthread_cond_broadcast()`:
  - Tüm bekleyen iş parçacıklarını uyandır.
  - Maliyet: çok fazla iş parçacığı uyandırılabilir.
  - Uyanık olmaması gerekenler uyanır, durumu yeniden kontrol eder ve ardından uyku moduna geri döner.

31. Semafor

Operating System: Three Easy Pieces

Semafor: Tanım

- Tamsayı değer alan bir nesne.
- İki rutini var: *sem_wait()* ve *sem_post().*
- İlklendirme örneği: Bir semafor *s* tanımla ve 1 değerini ata.
  - İkinci bağımsız değişken olan 0, semaforun aynı işlemdeki iş parçacıkları arasında paylaşıldığını göstermektedir.

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

Semafor: Etkileşim

- `sem_wait()`:
  - Çağrıldığında semaforun değeri bir azaltılır.
  - Değer negatif ise çağrıyı yapan iş parçacığı bekler ve uyur.
  - Semaforun mutlak değeri bekleyen iş parçacığı sayısına eşittir.

```
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  }
```

Semafor: Etkileşim (Devam)

- `sem_post()`:
  - Basitçe semaforun değerini bir artırır.
  - Uyandırılmayı bekleyen iş parçacıkları varsa bunlardan birini uyandırır.

```
1  int sem_post(sem_t *s) {
2        increment the value of semaphore s by one
3        if there are one or more threads waiting, wake one
4  }
```

İkili Semafor (Kilit)

- **X** ’in başlangıç değeri ne olmalıdır?

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

İkili Semafor (Kilit)

- **X** ’in başlangıç değeri ne olmalıdır?
- Doğru Cevap: 1

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

Semafor Kullanan Bir İş parçacığının İzlenmesi

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

Semafor Kullanan İki İş parçacığının İzlenmesi

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	`call sem_wait()`	Running		Ready
0	`sem_wait() returns`	Running		Ready
0	`(crit set: begin)`	Running		Ready
0	*`Interrupt; Switch → T1`*	Ready		Running
0		Ready	`call sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem < 0)→sleep`	sleeping
-1		Running	*`Switch → T0`*	sleeping
-1	`(crit sect: end)`	Running		sleeping
-1	`call sem_post()`	Running		sleeping
0	`increment sem`	Running		sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post() returns`	Running		Ready
0	*`Interrupt; Switch → T1`*	Ready		Running
0		Ready	`sem_wait() retruns`	Running
0		Ready	`(crit sect)`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`sem_post() returns`	Running

Semaforların Koşul Değişkeni olarak kullanılması

- **X** 'in başlangıç değeri ne olmalıdır?

```
1   sem_t s;
2
3   void *
4   child(void *arg) {
5       printf("child\n");
6       sem_post(&s); // signal here: child is done
7       return NULL;
8   }
9
10   int
11   main(int argc, char *argv[]) {
12      sem_init(&s, 0, X); // what should X be?
13      printf("parent: begin\n");
14      pthread_t c;
15      pthread_create(c, NULL, child, NULL);
16      sem_wait(&s); // wait here for child
17      printf("parent: end\n");
18      return 0;
19   }
```

A Parent Waiting For Its Child

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

The execution result

Semaforların Koşul Değişkeni olarak kullanılması

- **X** 'in başlangıç değeri ne olmalıdır?
- Doğru Cevap: 0

```
1   sem_t s;
2
3   void *
4   child(void *arg) {
5       printf("child\n");
6       sem_post(&s); // signal here: child is done
7       return NULL;
8   }
9
10   int
11   main(int argc, char *argv[]) {
12      sem_init(&s, 0, X); // what should X be?
13      printf("parent: begin\n");
14      pthread_t c;
15      pthread_create(c, NULL, child, NULL);
16      sem_wait(&s); // wait here for child
17      printf("parent: end\n");
18      return 0;
19   }
```

A Parent Waiting For Its Child

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

The execution result

Ana İş parçacığı Çocuğu Bekliyor (Senaryo 1)

- Çocuk iş parçacığı `sem_post()` çağrısı yapmadan Ana iş parçacığı `sem_wait()` çağrısı yapar.

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	`call sem_wait()`	Running		Ready
-1	`decrement sem`	Running		Ready
-1	`(sem < 0)→sleep`	sleeping		Ready
-1	*Switch→Child*	sleeping	`child runs`	Running
-1		sleeping	`call sem_post()`	Running
0		sleeping	`increment sem`	Running
0		Ready	`wake(Parent)`	Running
0		Ready	`sem_post() returns`	Running
0		Ready	*Interrupt; Switch→Parent*	Ready
0	`sem_wait() retruns`	Running		Ready

Ana İş parçacığı Çocuğu Bekliyor (Senaryo 2)

- Çocuk, ana iş parçacığı `sem_wait()` çağrısı yapmadan sonlanır.

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*`Interrupt; switch→Child`*	Ready	`child runs`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post() returns`	Running
1	`parent runs`	Running	*`Interrupt; Switch→Parent`*	Ready
1	`call sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem<0)→awake`	Running		Ready
0	`sem_wait() retruns`	Running		Ready

Üretici/ Tüketici (Sınırlı Arabellek) (Bounded Buffer) Problemi

- **Üretici**: `put()` arayüzü
  - İçine veri koymak için arabelleğin boşalmasını bekler.
- **Tüketici**: `get()` arayüzü
  - Kullanmadan önce arabelleğin doldurulmasını bekler.

```
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  }
```

Üretici/ Tüketici
(Sınırlı Arabellek) (Bounded Buffer) Problemi

```
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 i, 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  …
```

İlk Deneme: Dolu ve Boş Koşulları Ekleme

Üretici/ Tüketici
(Sınırlı Arabellek) (Bounded Buffer) Problemi

```
21   int main(int argc, char *argv[]) {
22       // …
23       sem_init(&empty, 0, MAX);       // MAX buffers are empty to begin with…
24       sem_init(&full, 0, 0);          // … and 0 are full
25       // …
26   }
```

İlk Deneme: Dolu ve Boş Koşulları Ekleme (Devam)

- MAX'ın 1'den büyük olduğunu düşünelim.
- Birden fazla üretici varsa, f1 satırında yarış durumu gerçekleşebilir.
- Bu durum, oradaki eski verilerin üzerine yazıldığı anlamına gelir.
- Burada karşılıklı dışlama gerektiğini unuttuk.
- Bir arabelleğin doldurulması ve indeksin bir arttırılması kritik bir bölgedir.

Bir Çözüm: Karşılıklı Dışlama Eklenmesi

```
1    sem_t empty;
2    sem_t full;
3    sem_t mutex;
4
5    void *producer(void *arg) {
6        int i;
7        for (i = 0; i < loops; i++) {
8                sem_wait(&mutex);          // line p0 (NEW LINE)
9                sem_wait(&empty);          // line p1
10               put(i);                    // line p2
11               sem_post(&full);           // line p3
12               sem_post(&mutex);          // line p4 (NEW LINE)
13       }
14   }
15
(Cont.)
```

Karşılıklı Dışlama Ekleme (Yanlış Çözüm)

Bir Çözüm: Karşılıklı Dışlama Eklenmesi (Devam)

```
(Cont.)
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&mutex);          // line c0 (NEW LINE)
20          sem_wait(&full);           // line c1
21          int tmp = get();           // line c2
22          sem_post(&empty);          // line c3
23          sem_post(&mutex);          // line c4 (NEW LINE)
24          printf("%d\n", tmp);
25      }
26  }
```

Karşılıklı Dışlama Ekleme (Yanlış Çözüm) (Devamı)

Yanlışlık Nerede?

- İki iş parçacığı düşünelim: bir üretici ve bir tüketici.
- Tüketici mutex kilidini elde eder (satır c0).
- Tüketici tam semaforun (satır c1) sem_wait()'ini çağırdığında uyumaya başlar. Fakat tüketici hala mutex'i elinde tutuyor!
- Üretici, mutex kilidin (satır p0) sem_wait()'ini çağırır.
- Üretici artık beklemeye mahkûmdur → klasik bir kilitlenme (deadlock) durumu.

Sonunda Doğru Çalışan Tam bir Çözüm

```
1    sem_t empty;
2    sem_t full;
3    sem_t mutex;
4
5    void *producer(void *arg) {
6        int i;
7        for (i = 0; i < loops; i++) {
8                sem_wait(&empty);          // line p1
9                sem_wait(&mutex);          // line p1.5 (MOVED MUTEX HERE…)
10               put(i);                    // line p2
11               sem_post(&mutex);          // line p2.5 (… AND HERE)
12               sem_post(&full);           // line p3
13       }
14   }
15
(Cont.)
```

Karşılıklı Dışlama Ekleme (Doğru Çözüm)

Sonunda Doğru Çalışan Tam bir Çözüm (Devam)

```
(Cont.)
16   void *consumer(void *arg) {
17       int i;
18       for (i = 0; i < loops; i++) {
19               sem_wait(&full);          // line c1
20               sem_wait(&mutex);         // line c1.5 (MOVED MUTEX HERE…)
21               int tmp = get();          // line c2
22               sem_post(&mutex);         // line c2.5 (… AND HERE)
23               sem_post(&empty);         // line c3
24               printf("%d\n", tmp);
25       }
26   }
27
28   int main(int argc, char *argv[]) {
29       // …
30       sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with …
31       sem_init(&full, 0, 0);    // ... and 0 are full
32       sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33       // …
34   }
```

Karşılıklı Dışlama Ekleme (Doğru Çözüm)

Reader-Writer Locks

- Imagine a number of concurrent list operations, including **inserts** and simple **lookups**.
  - **insert:**
    - Change the state of the list
    - A traditional critical section makes sense.
  - **lookup:**
    - Simply *read* the data structure.
    - As long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently.

This special type of lock is known as a reader-write lock.

Okuyucu-Yazıcı Kilitleri

- Kilidi yalnızca tek bir yazıcı alabilir.
- Bir okuyucu bir okuma kilidi edindiğinde, daha fazla okuyucunun da okuma kilidini almasına izin verilir.
- Bir yazıcı, tüm okuyucuların işi bitene kadar beklemek zorundadır.

```
1    typedef struct _rwlock_t {
2        sem_t lock;       // binary semaphore (basic lock)
3        sem_t writelock; // used to allow ONE writer or MANY readers
4        int readers;      // count of readers reading 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       …
```

Okuyucu-Yazıcı Kilitleri (Devam)

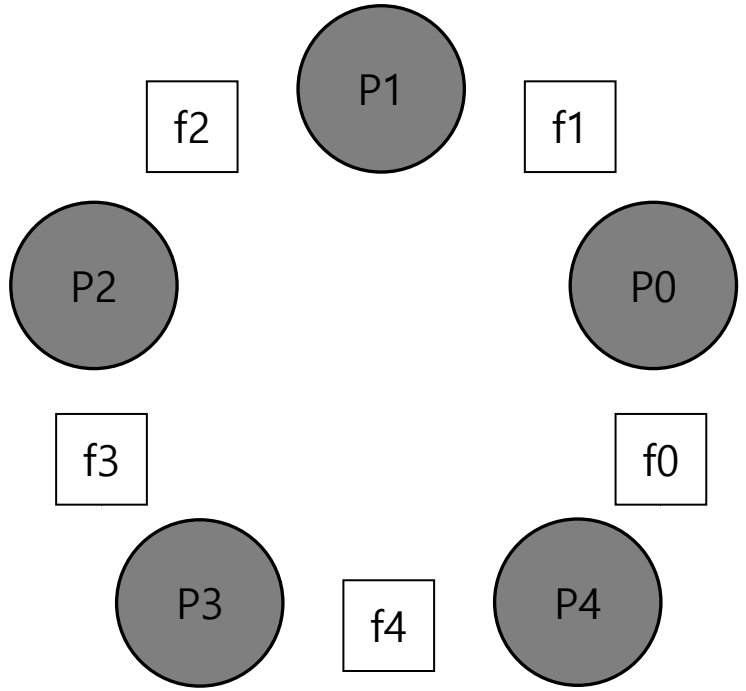
```
15        rw->readers++;
16        if (rw->readers == 1)
17                sem_wait(&rw->writelock); // first reader acquires writelock
18        sem_post(&rw->lock);
19    }
20
21    void rwlock_release_readlock(rwlock_t *rw) {
22        sem_wait(&rw->lock);
23        rw->readers--;
24        if (rw->readers == 0)
25                sem_post(&rw->writelock); // last reader releases 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    }
```

Okuyucu-Yazıcı Kilitleri (Devam)

- Okuyucu-yazıcı kilitlerinde bir adalet sorunu vardır.
- Okuyucunun yazıcıları aç bırakması görece olarak kolaydır.
- Bir yazıcı beklerken daha fazla okuyucunun kilide girmesi nasıl önlenir?

Yemek Yiyen Filozoflar Problemi (Dining Philosophers Problem)

- Bir masanın etrafında oturan beş "filozof" olduğunu varsayalım.
- Her filozof çifti arasında tek bir çatal vardır (toplam beş çatal).
- Filozofların her birinin düşündüğü (çatala ihtiyaç duymadığı) ve yemek yediği zaman dilimleri vardır.
- Bir filozofun yemek yiyebilmesi için sağında ve solundaki iki çatala ihtiyacı vardır.
- Bu durum çatallar için çekişme (contention) durumu oluşturur.



Yemek Yiyen Filozoflar Problemi (Dining Philosophers Problem) (Devam)

- Temel zorluklar:
  - Kilitlenme olmaması.
  - Açlık çeken ve asla yemek yiyemeyen filozof olmaması.
  - Yüksek düzeyde eşzamanlılık sağlanması.

```
while (1) {
        think();
        getforks();
        eat();
        putforks();
}
```

Basic loop of each philosopher

```
// helper functions
int left(int p) { return p; }

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

Helper functions (Downey's solutions)

- Filozof p, solundaki çatalı istiyor → `left(p)` çağrısı yapar.
- Filozof p, sağındaki çatalı istiyor → `right(p)` çağrısı yapar.

Yemek Yiyen Filozoflar Problemi (Dining Philosophers Problem) (Devam)

- Her çatal için bir semafora ihtiyacımız var: `sem_t forks[5]`.

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

getforks() ve putforks() Rutinleri (Yanlış Çözüm)

- Kilitlenme (Deadlock) meydana gelir!
  - Her filozof solundaki çatalı kaparsa, herhangi bir filozof sağındaki çatalı elde edemez.
  - Her biri sonsuza dek bir çatalı tutup diğerini bekleyecek.

Bir Çözüm: Bağımlılığı Kaldırmak

- Çatalların elde edilme sırasını değiştirelim.
- Filozof 4 çatalları farklı bir sırayla alsın.

```
1    void getforks() {
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    }
```

- Artık her filozofun bir çatalı elde edip diğerini beklemesi diye bir durum yoktur.
- Bekleme döngüsü bozulmuş oldu.

Semaforları Nasıl Implement Ederiz?

- Zemafor ismini verdiğimiz kendi semafor versiyonumuzu oluşturalım:

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

Semaforları Nasıl Implement Ederiz? (Devam)

```
22  void Zem_post(Zem_t *s) {
23      Mutex_lock(&s->lock);
24      s->value++;
25      Cond_signal(&s->cond);
26      Mutex_unlock(&s->lock);
27  }
```

- Zemafor, semaforun değerini bire bir korumaz.
  - Değer asla sıfırdan küçük olamaz.
- Öte yandan uygulanması daha kolaydır ve mevcut Linux uygulamaları için yeterlidir.