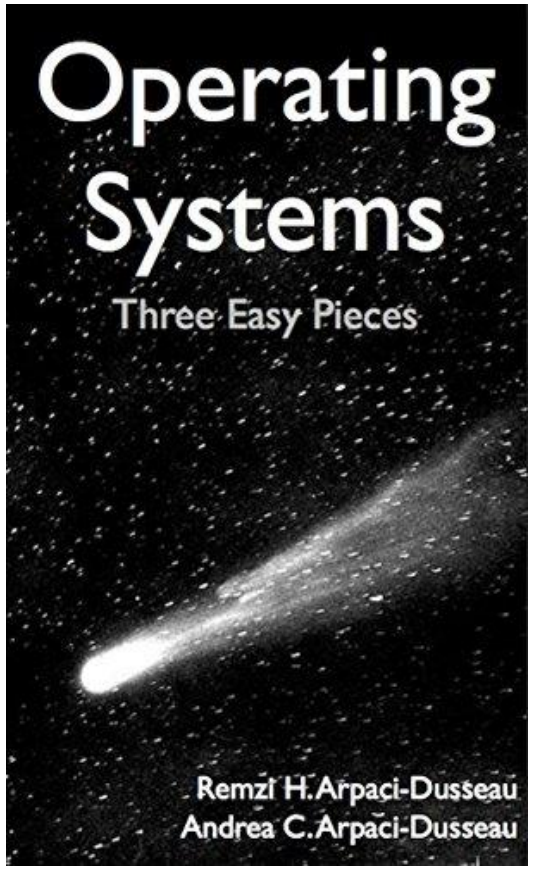
İşletim Sistemleri



7. Ders

Prof. Dr. Kemal Bıçakcı

26. Eşzamanlılık: Giriş

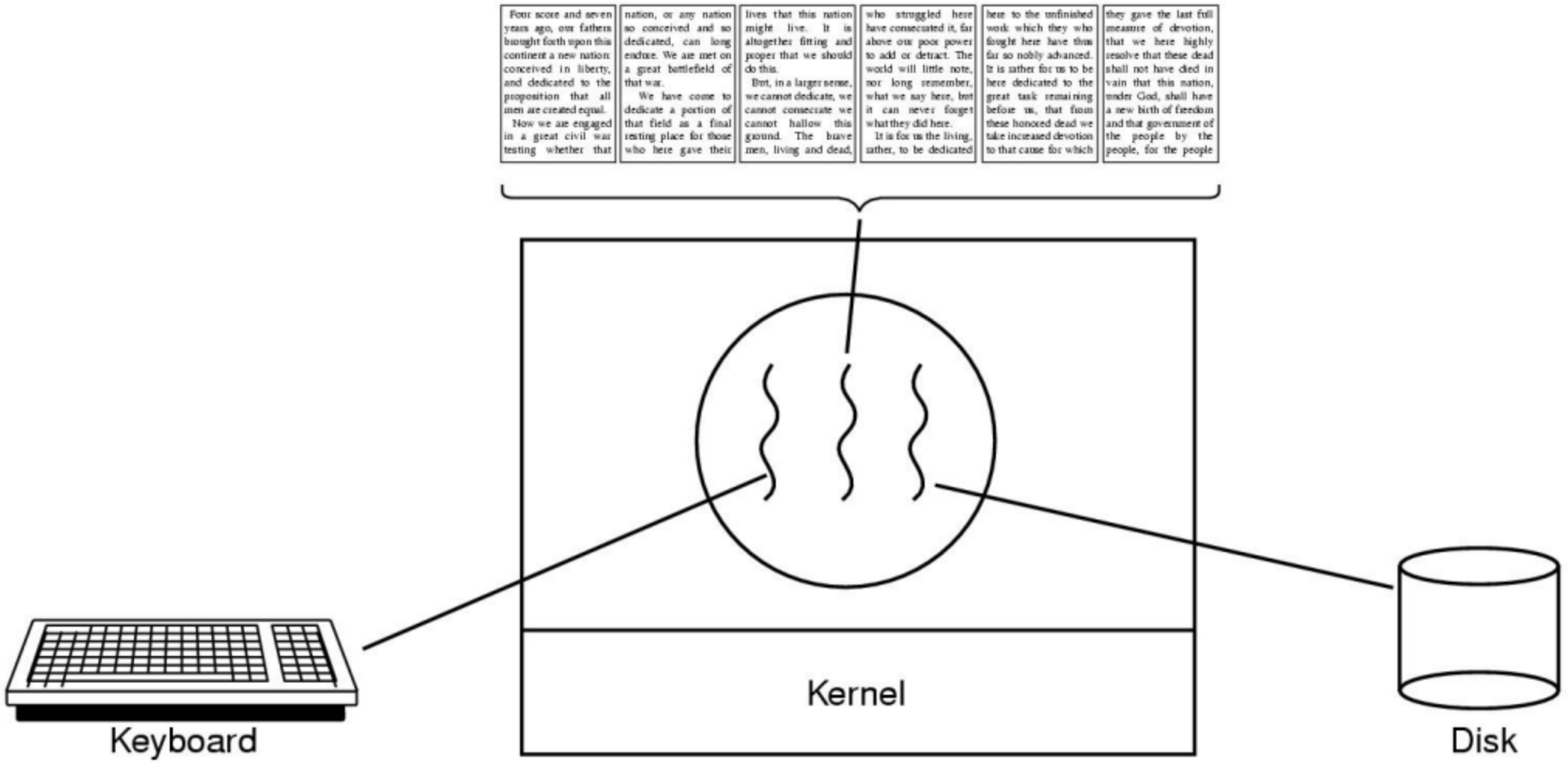
Operating System: Three Easy Pieces

İş Parçacığı (Thread)

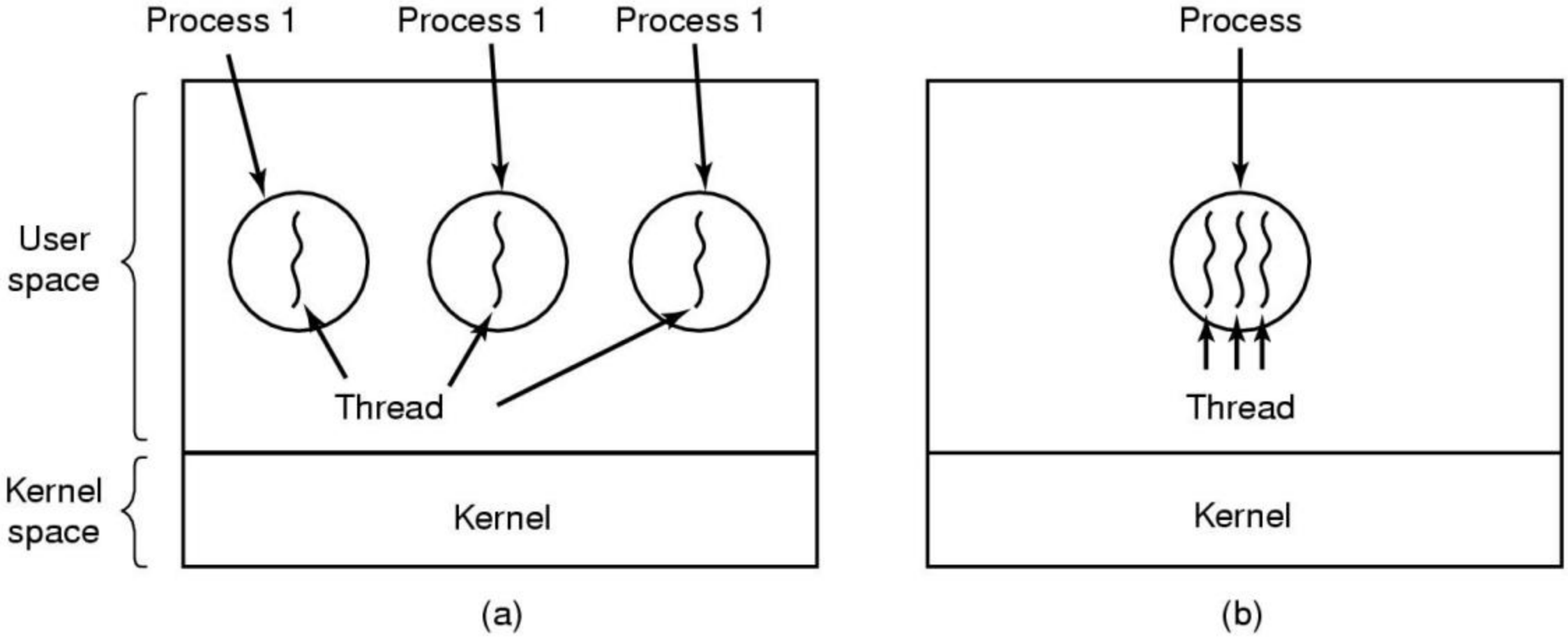
- Çalışan tek bir işlem için yeni bir soyutlama.
- Çok iş parçacıklı bir program (Multi-threaded program):
  - Çok iş parçacıklı bir programın birden fazla çalıştırma noktası vardır.
  - Birden çok Program Sayacı (PC)
  - Aynı adres alanını paylaşırlar.

Niye İş Parçacıkları Kullanılır?

- İki farklı neden var.
- (Gerçek) Paralellik (birden çok işlemcili bir sistemde)
  - Örnek: Çok büyük diziler veya matrisler ile çalışan bir program.
- Yavaş I/O nedeniyle program ilerlemesinin engellenmesini önlemek için.
  - Programınızdaki bir iş parçacığı beklerken, CPU zamanlayıcı diğer iş parçacıklarına geçebilir.
  - İş parçacıkları, programlardaki işlemler için çoklu programlamanın yaptığı gibi, G/Ç'nin tek bir program içindeki diğer etkinliklerle çakışmasını (overlap) sağlar.
- Neden işlemler yerine iş parçacıkları kullanıyoruz?



üç iş parçacıklı bir kelime işlemci

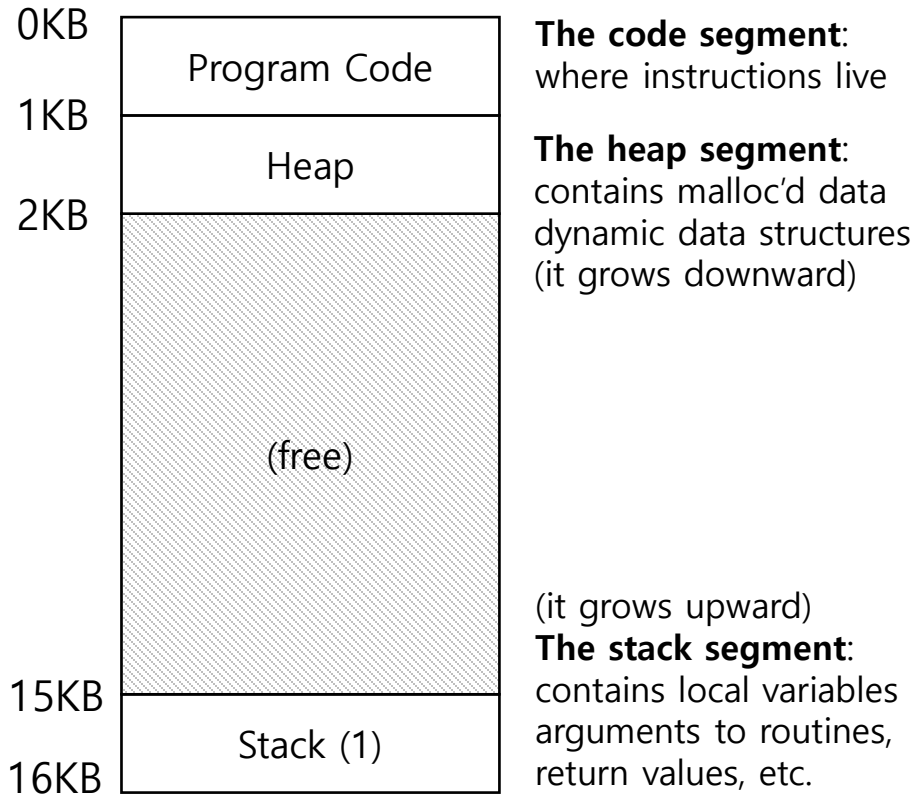


(a) Her biri tek iş parçacıklı üç işlem, (b) üç iş parçacıklı bir işlem

Context Switch
(İş parçacıkları arasında)

- Her iş parçacığının kendi program sayacı ve yazmaç kümesi vardır.
- Her iş parçacığının durumunu saklamak için bir veya daha fazla iş parçacığı kontrol bloğu (TCB) gerekir.
- Bir iş parçacığını (T1) çalıştırmaktan diğerine (T2) geçiş yaparken, T1'in yazmaç durumu kaydedilir ve T2'nin yazmaç durumu yüklenir.
- Adres uzayı ise aynı kalır.

İş parçacıklarının herbiri için ayrı bir yığın (stack) vardır.

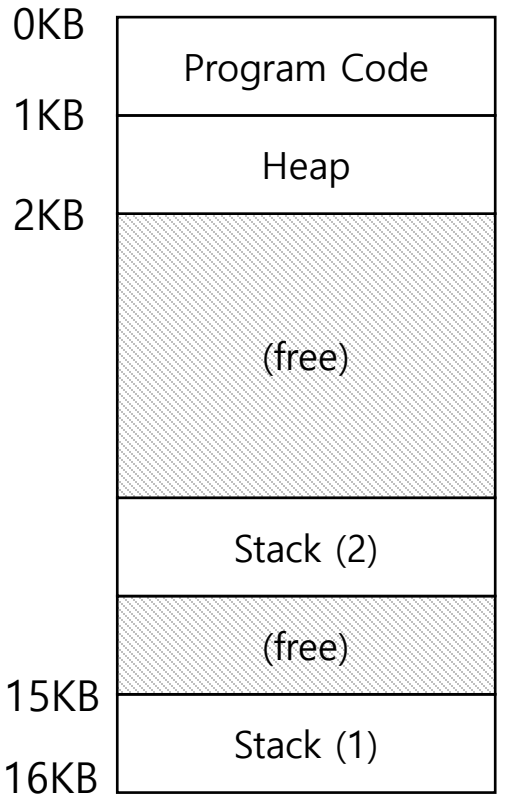


The code segment: where instructions live

The heap segment: contains malloc'd data dynamic data structures (it grows downward)

(it grows upward)
The stack segment: contains local variables arguments to routines, return values, etc.

A Single-Threaded Address Space



Two threaded Address Space

Yarış Durumu (Race Condition)

- İki iş parçacıklı bir örnek:
  - counter = counter + 1 (başlangıç değer: 50)
  - Sonucun 52 olmasını bekleriz. Acaba her zaman öyle midir?

OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	mov %eax, 0x8049a1c		113	51	**51**

Kritik Bölge (Critical Section)

TANIM: Paylaşılan bir değişkene erişen ve aynı anda birden fazla iş parçacığı tarafından çalıştırılmaması gereken kod parçası.

- Kritik bölgeyi aynı anda çalıştıran birden çok iş parçacığı, yarış durumuna neden olabilir.
- Kritik bölgeler için atomikliği (atomicity) destekleme ihtiyacı (karşılıklı dışlama – mutual exclusion) vardır.

Kilitler (Locks)

- Kritik bölgelerin tek bir atomik buyrukmuş gibi çalıştırıldığından emin olmak için kilit kullanılabilir.

```
1     lock_t mutex;
2     . . .
3     lock(&mutex);
4     balance = balance + 1;
5     unlock(&mutex);
```

Kritik bölge

Başka Tür bir Etkileşim

- Paylaşılan değişkenlere erişim ve kritik bölgeler için atomikliği destekleme ihtiyacı, iş parçacıkları arasındaki tek etkileşim türü değildir.
- Diğer bir etkileşim türü: Bir iş parçacığının, devam etmeden önce bazı eylemleri tamamlaması için diğerini beklemesi gerekebilir.
  - Daha sonra bu konu hakkında daha fazla bilgi verilecektir.

Niye bu konu İşletim Sistemleri dersinde ele alınıyor?

27. Ara Perde: İş Parçacığı Uygulama Programlama Arayüzü (Thread API)

Operating System: Three Easy Pieces

Thread API

- Bu bölümde, iş parçacığı API'sinin ana bölümlerini ele alacağız.
- Her kısım sonraki bölümlerde daha ayrıntılı olarak açıklanacaktır.
- Dolayısıyla bu bölümün bir referans kaynağı olarak kullanılması önerilmektedir.

İş Parçacığı Oluşturma

- POSIX'te iş parçacıkları nasıl oluşturulur ve kontrol edilir?

```
#include <pthread.h>

int
pthread_create(      pthread_t *       thread,
               const pthread_attr_t * attr,
                     void *           (*start_routine)(void*),
                     void *           arg);
```

  - `thread`: iş parçacığı ile etkileşim için kullanılır.
  - `attr`: Bu iş parçacığının özelliklerini belirlemek içindir.
    - Yığın boyutu, Zamanlama önceliği, …
  - `start_routine`: Bu iş parçacığının çalıştırmaya başlayacağı fonksiyon.
  - `arg`: fonksiyonun (`start routine`) argümanları
    - *boş işaretçi (void pointer) bize herhangi türde bir argüman geçirilmesine imkan verir.*

İş Parçacığı Oluşturma (Devam)

- Eğer `start_routine` başka türde bir argümana veya dönüşe gerek duyarsa, ifadeler şu şekilde değiştirilebilir:
  - Argüman bir tam sayı ise:

```
int
pthread_create(…, // first two args are the same
                  void*  (*start_routine)(int),
                  int    arg);
```

  - Fonksiyon bir tamsayı dönüyor ise:

```
int
pthread_create(…, // first two args are the same
                  int  (*start_routine)(void*),
                  void*    arg);
```

Örnek: İş Parçacığı Oluşturma

```
#include <pthread.h>

typedef struct __myarg_t {
        int a;
        int b;
} myarg_t;

void *mythread(void *arg) {
        myarg_t *m = (myarg_t *) arg;
        printf("%d %d\n", m->a, m->b);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        int rc;

        myarg_t args;
        args.a = 10;
        args.b = 20;
        rc = pthread_create(&p, NULL, mythread, &args);
        …
}
```

Bir iş parçacığının tamamlanmasını bekleme

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- `thread`: Hangi iş parçacığının beklendiğini belirtir
- `value_ptr`: Dönüş değerini gösteren bir işaretçi
  - `pthread_join()` rutini değu eri değiştirdiğinden, kendisi de işaretçi olan değerin işaretçisi geçirilir.

Örnek: Bir İş Parçacığının Tamamlanmasını Bekleme

```
1   #include <stdio.h>
2   #include <pthread.h>
3   #include <assert.h>
4   #include <stdlib.h>
5
6   typedef struct __myarg_t {
7       int a;
8       int b;
9   } myarg_t;
10
11  typedef struct __myret_t {
12      int x;
13      int y;
14  } myret_t;
15
16  void *mythread(void *arg) {
17      myarg_t *m = (myarg_t *) arg;
18      printf("%d %d\n", m->a, m->b);
19      myret_t *r = malloc(sizeof(myret_t));
20      r->x = 1;
21      r->y = 2;
22      return (void *) r;
23  }
24
```

Örnek: Bir İş Parçacığının Tamamlanmasını Bekleme (Devam)

```
25  int main(int argc, char *argv[]) {
26      int rc;
27      pthread_t p;
28      myret_t *m;
29
30      myarg_t args;
31      args.a = 10;
32      args.b = 20;
33      pthread_create(&p, NULL, mythread, &args);
34      pthread_join(p, (void **) &m);  // this thread has been
                                        // waiting inside of the
                                        // pthread_join() routine.
35      printf("returned %d %d\n", m->x, m->y);
36      return 0;
37  }
```

Örnek: Tehlikeli Kod

- Bir iş parçacığından değerlerin nasıl döndürüldüğüne dikkat etmek gerekir.

```
1   void *mythread(void *arg) {
2       myarg_t *m = (myarg_t *) arg;
3       printf("%d %d\n", m->a, m->b);
4       myret_t r; // ALLOCATED ON STACK: BAD!
5       r.x = 1;
6       r.y = 2;
7       return (void *) &r;
8   }
```

  - r değişkeni geri döndürüldüğünde, otomatik olarak bellek tahsisi kaldırılır.

İki Kod Parçasını Karşılaştıralım:

```
1   void *mythread(void *arg) {
2       myarg_t *m = (myarg_t *) arg;
3       printf("%d %d\n", m->a, m->b);
4       myret_t *r = malloc(sizeof(myret_t)); // ALLOCATED ON HEAP
5       r->x = 1;
6       r->y = 2;
7       return (void *) r;
8   }
```

```
1   void *mythread(void *arg) {
2       myarg_t *m = (myarg_t *) arg;
3       printf("%d %d\n", m->a, m->b);
4       myret_t r; // ALLOCATED ON STACK: BAD!
5       r.x = 1;
6       r.y = 2;
7       return (void *) &r;
8   }
```

Örnek: Bir İş Parçacığına daha basit şekilde Argüman Aktarma

- Sadece bir değer geçirilmek istenirse:

```
1   void *mythread(void *arg) {
2       int m = (int) arg;
3       printf("%d\n", m);
4       return (void *) (arg + 1);
5   }
6
7   int main(int argc, char *argv[]) {
8       pthread_t p;
9       int rc, m;
10      pthread_create(&p, NULL, mythread, (void *) 100);
11      pthread_join(p, (void **) &m);
12      printf("returned %d\n", m);
13      return 0;
14  }
```

Kilitler (Locks)

- Kritik bölgeye girişte karşılıklı dışlama (mutual exclusion) sağlar.
  - Arayüz:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - Kullanım (ilklendirme ve hata kontrolü göserilmemiştir):

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

  - Başka hiçbir iş parçacığı kilidi tutmamaktadır → iş parçacığı kilidi alır ve kritik bölgeye girer.
  - Başka bir iş parçacığı kilidi tutuyorsa → iş parçacığı kilidi elde edene kadar geri dönmeyecektir.

Kilitler (Devam)

- Tüm kilitler uygun şekilde ilklendirilmelidir.
  - `PTHREAD_MUTEX_INITIALIZER` kullanılabilir

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - Daha dinamik diğer bir yol: `pthread_mutex_init()` kullanımı

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Kilitler (Devam)

- Kilitlerken ve kilidi açarken hata kodunu kontrol ediniz
  - Örnek bir sarmalayıcı (wrapper):

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

- Kilitleme için bu iki çağrı da kullanılabilir:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

  - trylock: kilit alınmışsa başarısız sonuç döner
  - timelock: zaman aşımından sonra geri döner

Durum Değişkenleri (Condition Variables)

- **Durum değişkenleri,** iş parçacıkları arasında bir tür sinyalleşmenin gerçekleşmesi gerektiğinde kullanışlıdır.

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

  - `pthread_cond_wait`:
    - Çağrıyı yapan iş parçacığı uyur.
    - Başka bir iş parçacığının sinyali beklenir.
  - `pthread_cond_signal`:
    - Durum değişkeninde bloklanan (uyuyan) iş parçacıklarından en az birinin bloklanmasını kaldırır (uyandırır).

Durum Değişkenleri (Devam)

- Bekleme rutinini çağıran iş parçacığı:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (ready == 0)
        pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock);
```

  - Bekleme çağrısı, söz konusu çağrıyı yapanı uyku moduna geçirirken kilidi bırakır.
  - Uyandıktan sonra geri dönmeden önce, bekleme çağrısı kilidi yeniden alır.

- Sinyal rutinini çağıran iş parçacığı:

```
pthread_mutex_lock(&lock);
ready = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

Durum Değişkenleri (Devam)

- Bekleyen iş parçacığı, basit bir if ifadesi yerine bir while döngüsünde durumu kontrol eder.
- Bekleyen bir iş parçacığını sahte bir şekilde uyandırabilecek bazı pthread implementasyonları vardır.
- Böyle bir durumda tekrar kontrol etmez ise bekleyen iş parçacığı durum değişmediği halde değiştiğini düşünecektir.

Durum Değişkenleri (Devam)

- Asla yapılmaması gereken bir hata:
  - Bir iş parçacığı beklemek ister:

```
while(ready == 0)
        ; // spin
```

  - Diğeri de sinyal gönderir:

```
ready = 1;
```

  - Performansı kötüdür → gereksiz yere işlemci kullanılır
  - Daha da önemlisi, ciddi senkronizasyon hatalarına yol açabilir.

Derleme ve Çalıştırma

- Kod örneklerini derlemek için `pthread.h` başlığının eklenmesi gerekir.
  - -pthread bayrağını ekleyerek pthreads kütüphanesi ile açık bir bağlantı kurulur:

```
prompt> gcc -o main main.c -Wall -pthread
```

  - Daha fazla bilgi için:

```
man -k pthread
```

Thread API Kullanım Rehberi

- Basit tutun.
- İş parçacığı etkileşimlerini en aza indirin.
- Kilitleri ve durum değişkenlerini ilklendirin.
- Dönüş kodlarınızı kontrol edin.
- İş parçacıklarına argümanları nasıl ilettiğinize ve iş parçacıklarından nasıl değer döndürdüğünüze dikkat edin.
- Her iş parçacığının kendi yığını vardır.
- İş parçacıkları arasında sinyalleşme için her zaman durum değişkenlerini kullanın.
- manuel sayfalarını kullanın.

28. Kilitler

Operating System: Three Easy Pieces

Kilit: Temel Fikir

- Herhangi bir kritik bölgenin tek bir atomik buyrukmuş gibi çalıştırılmasını sağlar.
  - Bir örnek: paylaşılan bir değişkenin değerinin artırılması

```
balance = balance + 1;
```

  - Kritik bölge etrafına bazı kod satırları ekleriz:

```
1     lock_t mutex; // some globally-allocated lock 'mutex'
2     …
3     lock(&mutex);
4     balance = balance + 1;
5     unlock(&mutex);
```

Kilit: Temel Fikir

- Kilit değişkeni, kilidin durumunu tutar:
  - Kullanılabilir (veya kilidi açılmış veya serbest): Hiçbir iş parçacığı kilidi tutmamaktadır.
  - Elde edildi (veya kilitlendi veya tutuldu): Bir ve sadece bir iş parçacığı kilidi tutmaktadır ve muhtemelen kritik bölgededir.

lock() çağrısı

- `lock()`
  - Kilidi elde etmeye çalışır.
  - Kilidi başka bir iş parçacığı tutmuyorsa, kilidi elde edecektir.
  - Bu iş parçacığının kilidin sahibi olduğu söylenir.
  - Bu iş parçacığı kritik bölgeye girer.
  - Kilidi tutan iş parçacığı kritik bölgedeyken, diğer iş parçacıklarının kritik bölgeye girmesi engellenir.

Pthread kilidi: mutex

- `mutex`: POSIX kütüphanesinin kilit için kullandığı ad.
  - İş parçacıkları arasında karşılıklı dışlama sağlar.

```
1   pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3   Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
4   balance = balance + 1;
5   Pthread_mutex_unlock(&lock);
```

  - Farklı değişkenleri korumak için farklı kilitler kullanıyor olabiliriz. Bu sayede eşzamanlılık artar (daha ince taneli bir yaklaşım).

Bir Kilidin İnşası

- Bir kilit oluşturmak için donanım ve işletim sistemi beraber görev alır.

Kilitleri Değerlendirme – Temel kriterler

- **Karşılıklı Dışlama**
  - Kilit çalışıyor mu, birden fazla iş parçacığının kritik bir bölüme girmesini engelliyor mu?
- **Adalet**
  - Kilidi elde etmek için yarışan her iş parçacığı, serbest olduğunda onu edinme konusunda adil bir şans elde ediyor mu? (Açlık problemi var mı?)
- **Verim**
  - Kilit kullanıldığında eklenen zaman yükleri ne seviyede?

Kesmeleri Kontrol Etmek

- Kritik bölgeler için Kesmeleri devre dışı bırakarak kilit inşası.
  - Karşılıklı dışlama sağlamak için kullanılan en eski çözümlerden biridir.
  - Tek işlemcili sistemler için icat edilmiştir.

```
1   void lock() {
2       DisableInterrupts();
3   }
4   void unlock() {
5       EnableInterrupts();
6   }
```

  - Problem: Uygulamalara çok fazla güven gerektirir.
  - Açgözlü (veya kötü niyetli) program, işlemciyi tekeline alabilir.
  - Çoklu işlemcilerde çalışmaz.
  - Kesmeleri maskeleyen veya maske kaldıran kod, modern CPU'larda yavaşlığa sebep olur.

Niye donanım desteği gereklidir?

- İlk deneme: Kilidin elde edilip edilmediğini gösteren bir bayrak (flag) kullanmak.
  - Aşağıdaki kod sorunludur.

```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 → lock is available, 1 → held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)  // TEST the flag
10                  ;  // spin-wait (do nothing)
11      mutex->flag = 1;  // now SET it !
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

Niye donanım desteği gereklidir? (Devam)

- **Problem 1**: Karşılıklı dışlama sağlanmaz (varsayım: başlangıçta `flag=0`)

Thread1	Thread2
call `lock()` `while (flag == 1)` interrupt: switch to Thread 2	

	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1

`flag = 1;` // set flag to 1 (too!)	

- **Problem 2**: Spin-wait, başka bir iş parçacığını beklemek için zaman harcar.
- Bu nedenle, donanım tarafından sunulan bir atomik buyruğa ihtiyacımız vardır!
  - *test-and-set* buyruğu, atomik değişim buyruğu olarak da bilinir.

Test And Set Buyruğu (Atomik Değişim)

- Basit kilitlerin oluşturulmasını desteklemek için bir buyruk

```
1    int TestAndSet(int *ptr, int new) {
2        int old = *ptr;   // fetch old value at ptr
3        *ptr = new;       // store 'new' into ptr
4        return old;       // return the old value
5    }
```

  - *ptr* tarafından işaret edilen eski değeri döndürür (Test).
  - Aynı anda söz konusu değeri *new* değeri ile günceller (Set).
  - Bu işlem dizisi atomik olarak gerçekleştirilir.

Test and Set kullanan Basit «Spin» Kilidi

```
1    typedef struct __lock_t {
2        int flag;
3    } lock_t;
4
5    void init(lock_t *lock) {
6        // 0 indicates that lock is available,
7        // 1 that it is held
8        lock->flag = 0;
9    }
10
11   void lock(lock_t *lock) {
12       while (TestAndSet(&lock->flag, 1) == 1)
13                ;        // spin-wait
14   }
15
16   void unlock(lock_t *lock) {
17       lock->flag = 0;
18   }
```

Not: Tek işlemcili bir bilgisayarda düzgün çalışması için, boşaltmalı (preemptive) bir zamanlayıcı gerektirir.

«Spin» Kilitleri Değerlendirme

- **Doğruluk**: evet
  - Bu tür bir kilit, yalnızca tek bir iş parçacığının kritik bölgeye girmesine izin verir.
- **Adalet**: hayır
  - Bu tür kilitler herhangi bir adalet garantisi sağlamaz.
  - Gerçekten de, dönmeye başlayan bir iş parçacığı sonsuza kadar dönebilir.
- **Verim**: Tek CPU var ise, performans yükleri oldukça fazladır.
  - İş parçacığı sayısı kabaca CPU sayısına eşitse, bu tür kilitlerin iyi çalıştığı varsayılabilir (kritik bölgenin kısa olduğu varsayımı ile).

Diğer Benzer Buyruklar

- Compare-And-Swap
- Load-Linked and Store-Conditional
- Fetch-And-Add

Fetch-And-Add (Getir ve Ekle)

- Belirli bir adresteki eski değeri döndürürken bu değeri atomik olarak bir artırır.

```
1    int FetchAndAdd(int *ptr) {
2        int old = *ptr;
3        *ptr = old + 1;
4        return old;
5    }
```

Fetch-And-Add Hardware atomic instruction (C-style)

Bilet Kilidi (Ticket Lock)

- **Bilet kilidi Fetch-And-Add buyruğu ile oluşturulabilir.**
- **Tüm iş parçacıkları için ilerleme sağlanır → Adalet**

```
1   typedef struct __lock_t {
2       int ticket;
3       int turn;
4   } lock_t;
5
6   void lock_init(lock_t *lock) {
7       lock->ticket = 0;
8       lock->turn = 0;
9   }
10
11  void lock(lock_t *lock) {
12      int myturn = FetchAndAdd(&lock->ticket);
13      while (lock->turn != myturn)
14              ; // spin
15  }
16  void unlock(lock_t *lock) {
17      FetchAndAdd(&lock->turn);
18  }
```

Spin (Dönme): Nasıl Kurtuluruz?

- Donanım tabanlı «spin» kilitler basittir ve doğru çalışırlar.
- Fakat, bazı durumlarda, bu çözümler oldukça verimsiz olabilir.
- Bir iş parçacığı dönerken, bir değeri kontrol etmekten başka hiçbir şey yapmadan bütün bir zaman dilimini boşa harcar.

Dönmekten Nasıl Kaçınılır?
İşletim Sistemi desteğine de ihtiyacımız olacak!

Basit Bir Yaklaşım: Sadece Yol Ver

- Bir iş parçacığı döneceği zaman, CPU'yu başka bir iş parçacığına bırakır.
- Sistem çağrısı, bu iş parçacığını çalışır durumdan hazır duruma taşır.
- Context Switch maliyeti önemli olabilir ve açlık sorunu hala mevcuttur.

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```

Lock with Test-and-set and Yield

Kuyrukları Kullanmak: Dönmek Yerine Uyumak

- Hangi iş parçacıklarının kilidi elde etmek için beklediğini takip etmek için kuyruk kullanmak.
- `park()`
  - Çağrı yapan iş parçacığını uyku moduna geçir
- `unpark(threadID)`
  - `threadID` değerine sahip iş parçacığını uyandır

Kuyrukları Kullanmak: Dönmek Yerine Uyumak (Devam)

```
1   typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3   void lock_init(lock_t *m) {
4       m->flag = 0;
5       m->guard = 0;
6       queue_init(m->q);
7   }
8
9   void lock(lock_t *m) {
10      while (TestAndSet(&m->guard, 1) == 1)
11          ; // acquire guard lock by spinning
12      if (m->flag == 0) {
13          m->flag = 1; // lock is acquired
14          m->guard = 0;
15      } else {
16          queue_add(m->q, gettid());
17          m->guard = 0;
18          park();
19      }
20  }
21  …
```

Lock With Queues, Test-and-set, Yield, And Wakeup

Kuyrukları Kullanmak: Dönmek Yerine Uyumak (Devam)

```
22  void unlock(lock_t *m) {
23      while (TestAndSet(&m->guard, 1) == 1)
24          ; // acquire guard lock by spinning
25      if (queue_empty(m->q))
26          m->flag = 0; // let go of lock; no one wants it
27      else
28          unpark(queue_remove(m->q)); // hold lock (for next thread!)
29      m->guard = 0;
30  }
```

Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

Bu örnek kodda bir yarış durumu var mıdır?

Uyandırma/Bekleme yarış durumu

- İş parçacığı B, `park()` çağrısı yapmadan hemen önce iş parçacığı A kilidi serbest bırakırsa → İş parçacığı B sonsuza kadar uyuyabilir.
- Solaris bu sorunu üçüncü bir sistem çağrısı ekleyerek çözer: `setpark()`
- Bu çağrı ile bir iş parçacığı `park()` etmek üzere olduğunu gösterebilir. Eğer kesmeye uğrarsa ve `park` fiilen çağrılmadan önce başka bir iş parçacığı `unpark()` çağrısı yaparsa, sonraki `park()` uyumak yerine hemen geri döner.

```
1          queue_add(m->q, gettid());
2          setpark(); // new code
3          m->guard = 0;
4          park();
```

Code modification inside of `lock()`

İş parçacığı A

```
22  void unlock(lock_t *m) {
23      while (TestAndSet(&m->guard, 1) == 1)
24          ; // acquire guard lock by spinning
25      if (queue_empty(m->q))
26          m->flag = 0; // let go of lock; no one wants it
27      else
28          unpark(queue_remove(m->q)); // hold lock (for next thread!)
29      m->guard = 0;
30  }
```

Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

İş parçacığı B

```
1    typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3    void lock_init(lock_t *m) {
4        m->flag = 0;
5        m->guard = 0;
6        queue_init(m->q);
7    }
8
9    void lock(lock_t *m) {
10       while (TestAndSet(&m->guard, 1) == 1)
11           ; // acquire guard lock by spinning
12       if (m->flag == 0) {
13           m->flag = 1; // lock is acquired
14           m->guard = 0;
15       } else {
16           queue_add(m->q, gettid());
17           m->guard = 0;
18           park();
19       }
20   }
21   …
```

Lock With Queues, Test-and-set, Yield, And Wakeup

Futex

- Linux'de ise futex vardır (Solaris'deki `park` and `unpark` çağrılarına benzer).
- `futex_wait(address, expected)`
  - Çağıran diziyi uyku moduna geçir
  - Adresteki değer beklenen değere eşit değilse çağrı hemen geri döner.
- `futex_wake(address)`
  - Sırada bekleyen bir iş parçacığını uyandır.