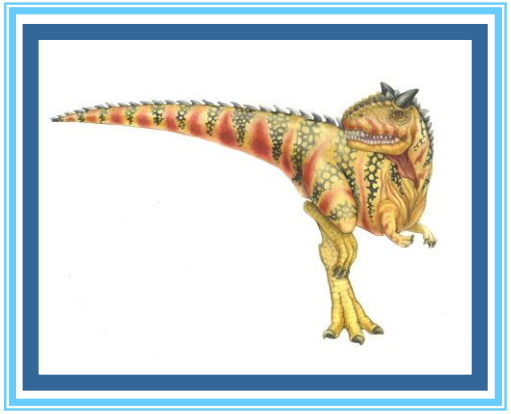
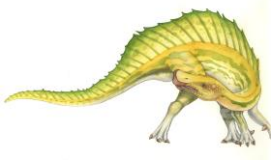
Hafta 4: İş Parçacıkları (Threads) & Eşzamanlılık (Concurrency)



Silberschatz, Galvin and Gagne ©2018

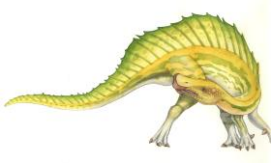
Bölüm 4: İş parçacıkları

- Genel bakış
- Çok Çekirdekli Programlama
- Çoklu İş Parçacığı Modelleri
- İş parçacığı kütüphaneleri
- Kapalı İşparçacığı
- İş Parçacığı Sorunları
- İşletim Sistemi Örnekleri



Hedefler

- İş parçacığının temel bileşenlerini ve iş parçacığı ve prosesler arasındaki farkı tanımlamak
- Çok iş parçacığı uygulamaları tasarlamanın avantajlarını ve zorluklarını açıklamak
- İş parçacığı havuzları, fork-join (oluştur-birleştir) ve Grand Central Dispatch dahil olmak üzere kapalı iş parçacığı farklı yaklaşımlarını göstermek
- Windows ve Linux işletim sistemlerinin iş parçacıklarını nasıl temsil ettiğini açıklamak
- Pthreads, Java ve Windows iş parçacığı API'lerini kullanarak çoklu iş parçacığı uygulamaları tasarlama

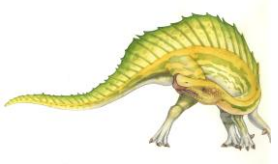


Motivasyon

- Çoğu modern uygulama çoklu iş parçacığı modeli ile tasarlanır
- iş parçacıkları uygulama altında çalışır
- Uygulama ile birden çok görev ayrı iş parçacıkları tarafından yerine getirilebilir
  - Ekranı güncelleştir
  - Veri getir
  - Yazım denetimi
  - Ağ isteğini yanıtlama
- İş parçacığı oluşturma hafif iken işlem oluşturma ağır
- Kodu basitleştirebilir, verimliliği artırabilir
- Çekirdekler genellikle çok iş parçacığı modelini kullanır

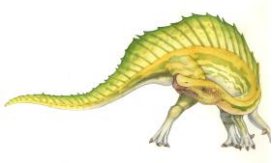
İş parçacığı ne demektir? (Bilişim Terimleri Sözlüğü-ODTU)

- *izlek - iş parçacığı*
  - Bir bilgi işleme sürecinde gerçekleştirilebilecek en küçük işlem birimi

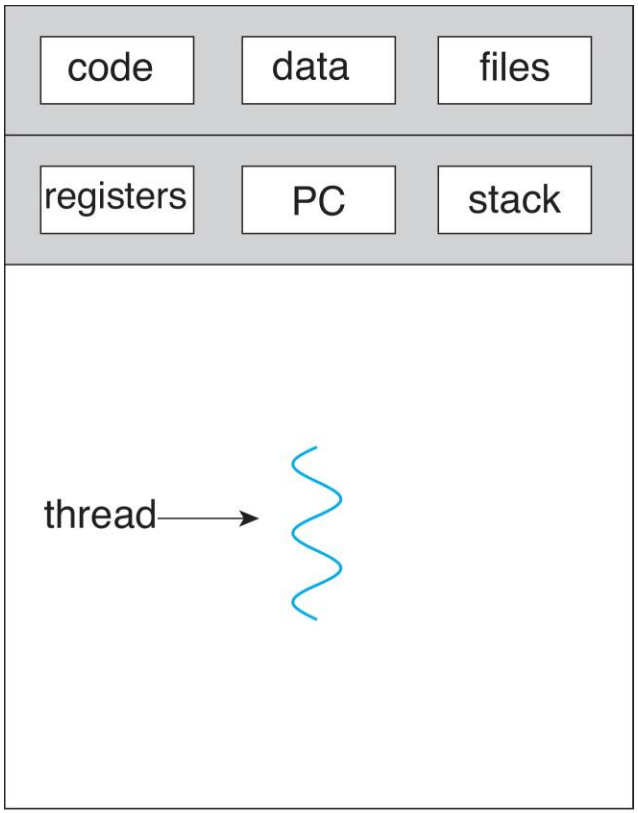


İş parçacığı ne demektir? (wikipedia)

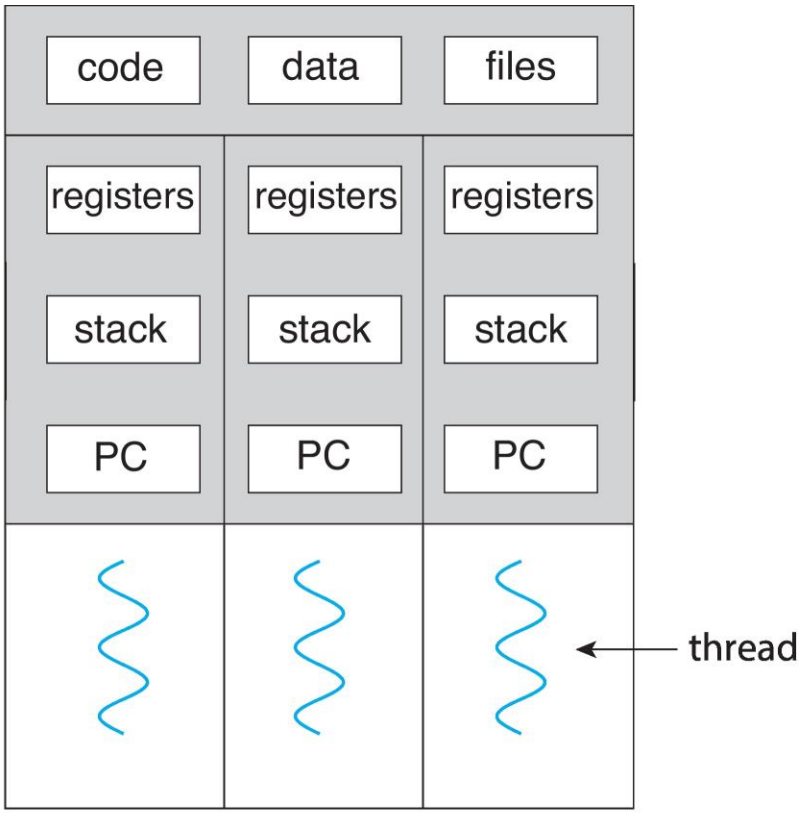
- **İş parçacığı (iş parçacığı)** bilgisayar biliminde, bir programın kendini eş zamanlı olarak çalışan birden fazla iş parçasına ayırabilmesinin bir yoludur.



Tek ve Çoklu İş Parçacıklı Prosesler

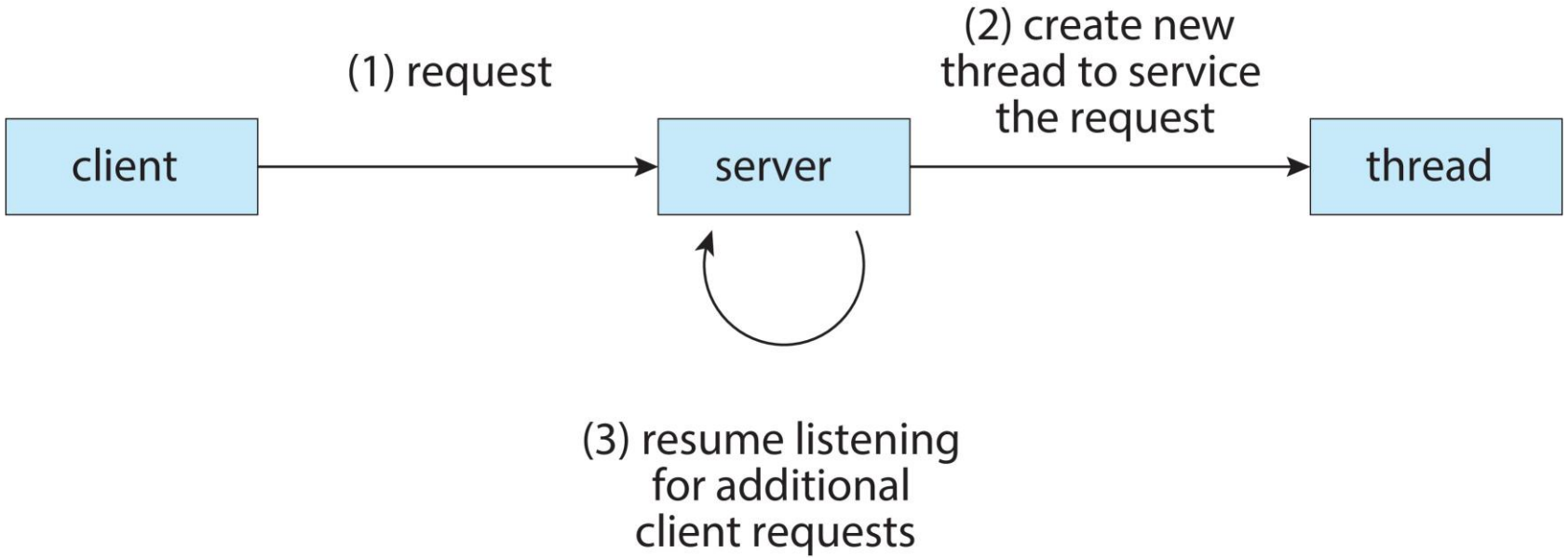


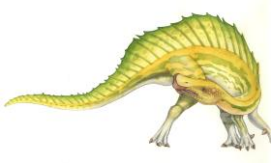
single-threaded process



multithreaded process

Çoklu İş Parçacığı Sunucu Mimarisi





Faydaları

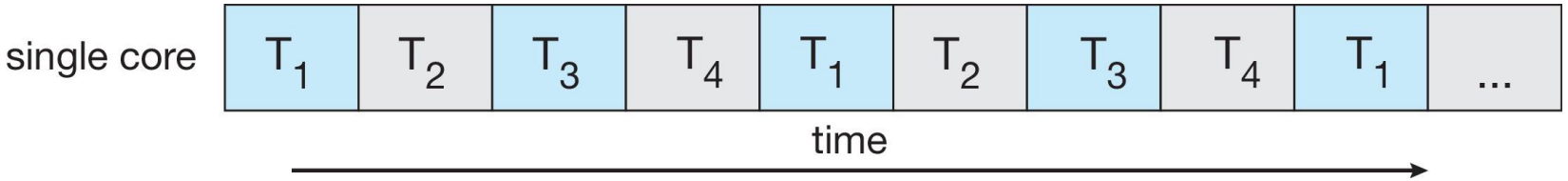
- **Duyarlılık –** özellikle kullanıcı arabirimleri için önemli olan prosesin bir kısmı bloke olursa, çalışma devam edebilir
- **Kaynak Paylaşımı –** iş parçacıkları proses kaynaklarını paylaşır, paylaşılan bellek veya mesaj iletiminden daha kolaydır.
- **Ekonomi –** proses oluşturmadan daha ucuz, iş parçacığı bağlam değişimi normal bağlam değişiminden daha düşük maliyeti vardır
- **Ölçeklenebilirlik –** proses çok çekirdekli mimarilerden yararlanabilir

Çok Çekirdekli Programlama

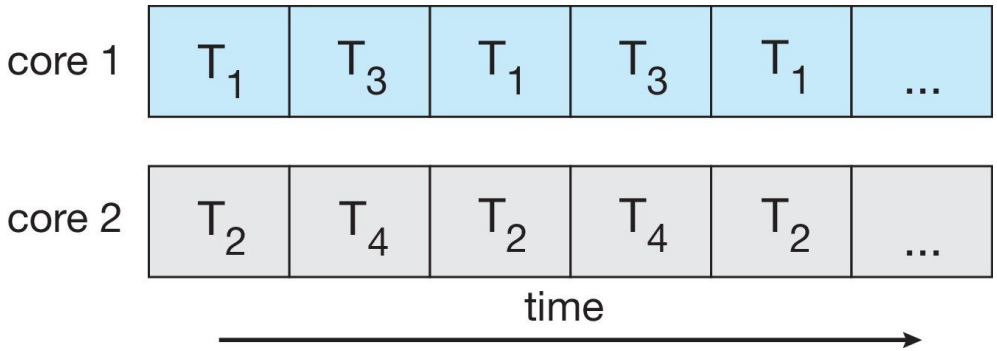
- **Çok çekirdekli** veya **çok işlemcili** sistemler programcılar üzerinde aşağıdaki baskı ve zorluklara neden olur: :
  - **Aktiviteleri bölme**
  - **Dengeleme**
  - **Veri bölme**
  - **Veri bağımlılığı**
  - **Test ve hata ayıklama**
- ***Parallelism*** bir sistemin eş zamanlı olarak birden fazla görev gerçekleştirebileceği anlamına gelir
- ***Eşzamanlılık*** birden fazla görevin ilerlemesini destekler
  - Tek işlemci / çekirdek, eşzamanlılık sağlayan sıralayıcı (scheduler)

Eşzamanlılık ve Paralellik

- **Tek çekirdekli sistemde eşzamanlı yürütme:**

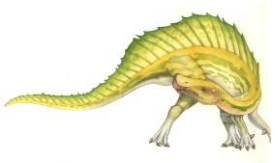


- **Çok çekirdekli sistemde paralellik:**

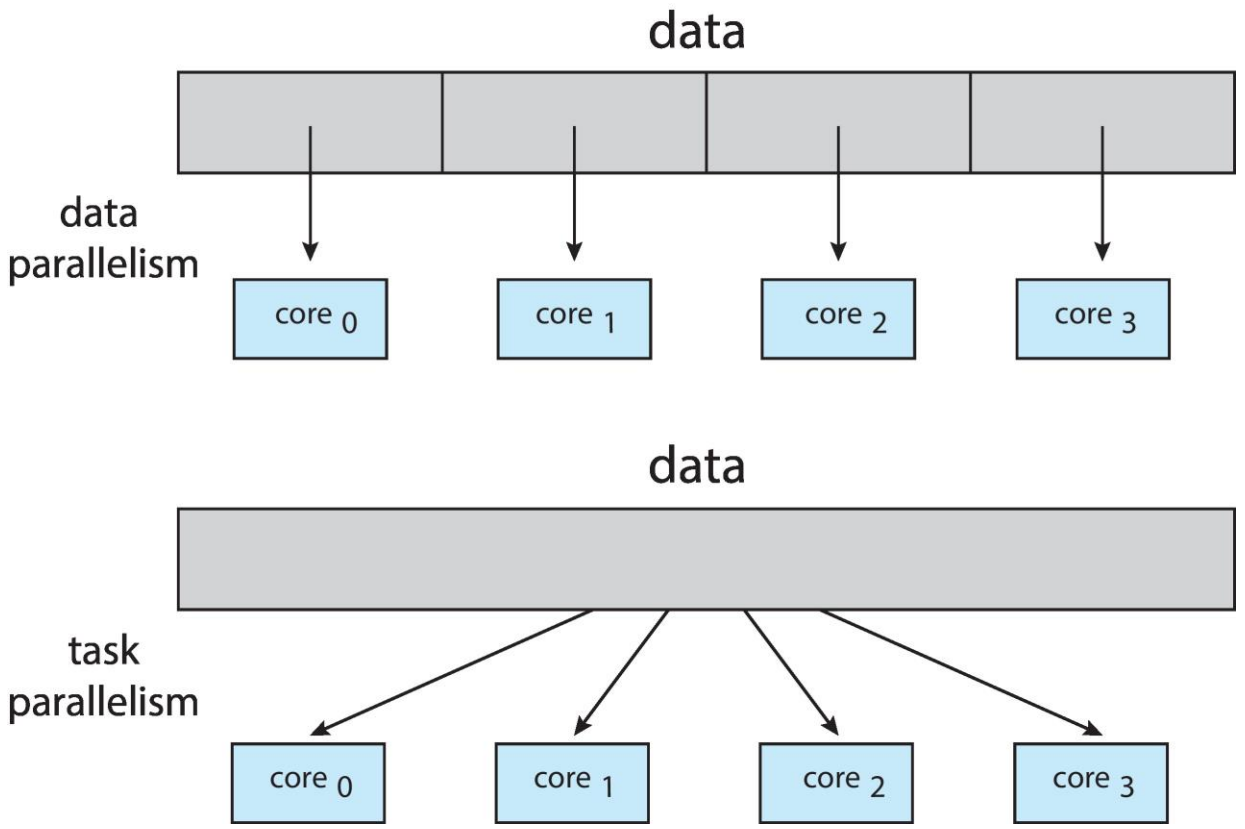


Çok Çekirdekli Programlama

- Paralellik türleri
  - **Veri paralelliği** – aynı verilerin alt kümelerini birden çok çekirdek arasında dağıtır, her birinde aynı işlem
  - **Görev paralelliği** – çekirdekler arasında iş parçacığı dağıtma, her iş parçacığı farklı işlem gerçekleştirir



Veri ve Görev Paralelliği



Amdahl Yasası

- Hem seri hem de paralel bileşenlere sahip bir uygulamaya ek çekirdek eklenmesinden kaynaklanan performans kazançlarını tanımlar
- *S* seri kısımdır
- *N* çekirdek sayısı

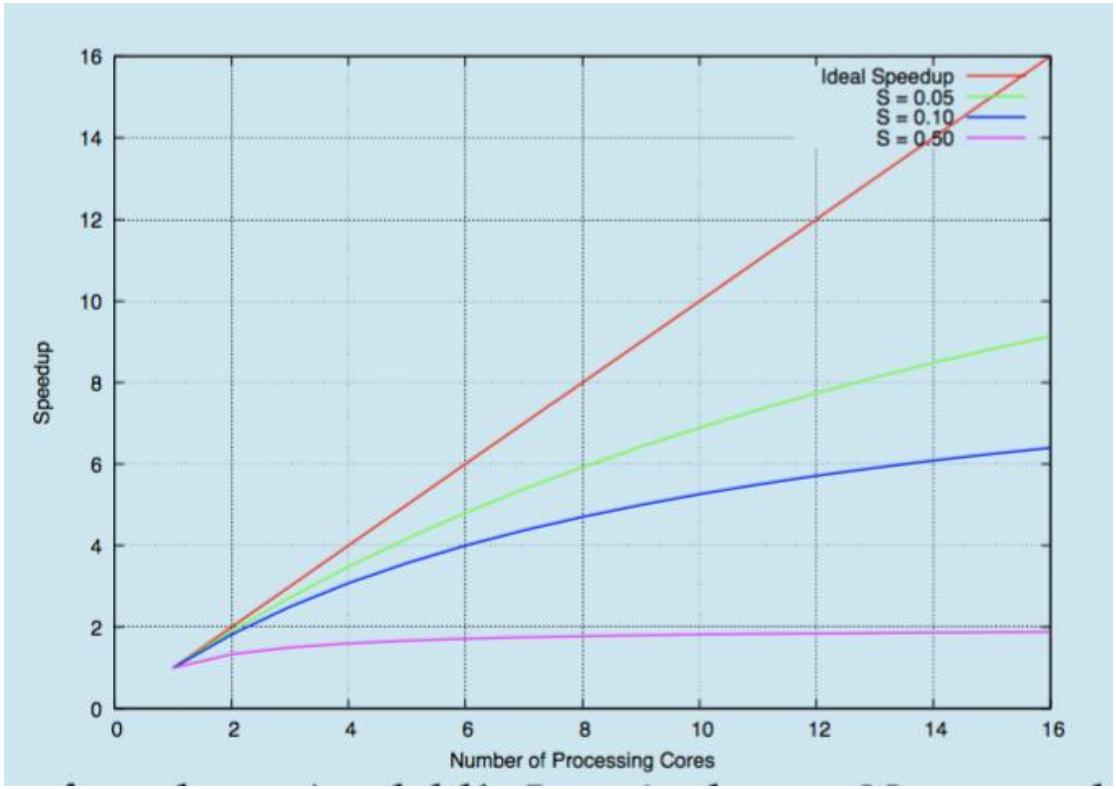
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

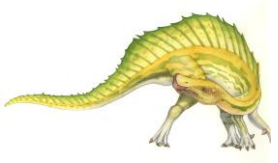
- Diğer bir deyişle, uygulama % 75 paralel / %25 seri ise, 1 çekirdekten 2'ye geçmek sistemi 1,6 kat hızlanır
- *N* sonsuza giderken , hız 1 / *S e gider*

Bir uygulamanın seri kısmı, ek çekirdekler eklenerek elde edilen performans üzerinde ters orantılı bir etkiye sahiptir

- Ama yasa çağdaş çok çekirdekli sistemleri dikkate alır mı?

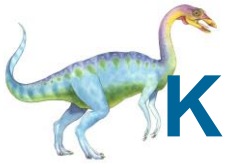
Amdahl Yasası



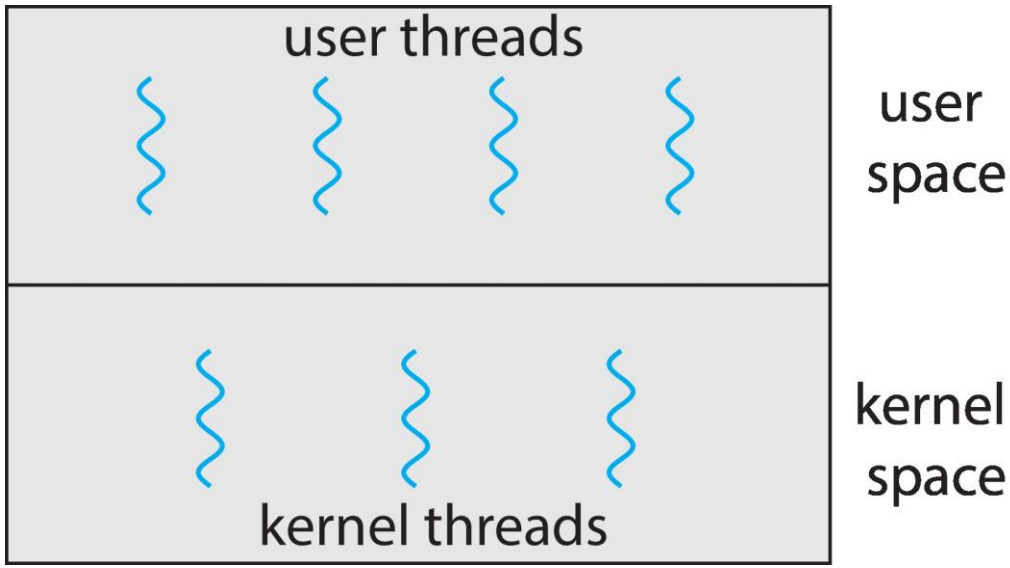


Kullanıcı ve Çekirdek İş Parçacıkları

- **Kullanıcı iş parçacıkları** – yönetim kullanıcı düzeyinde iş parçacıkları kütüphanesi tarafından yapılır
- Üç temel iş parçacığı kütüphanesi:
  - POSIX **Pthreads**
  - Windows iş parçacıkları
  - Java iş parçacıkları
- **Çekirdek iş parçacıkları**- Çekirdek tarafından desteklenir
- Örnekler – aşağıdakiler de dahil olmak üzere hemen hemen tüm genel amaçlı işletim sistemleri:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

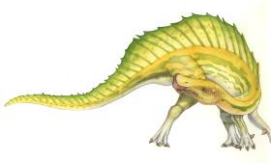


Kullanıcı ve Çekirdek İş Parçacıkları



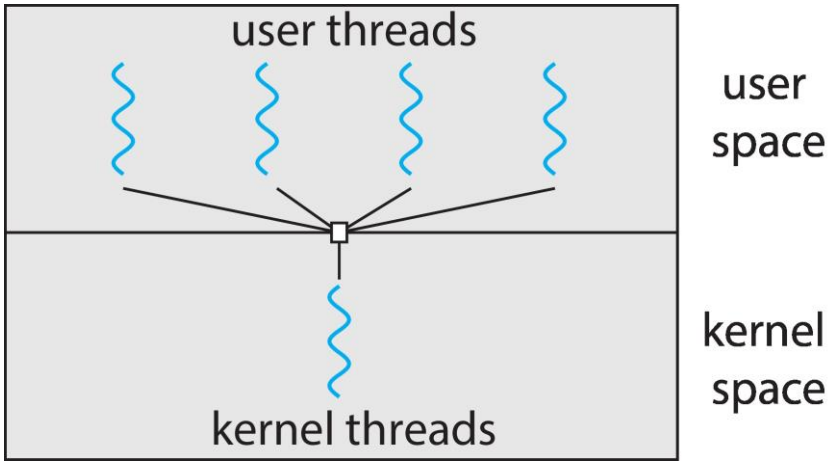
Çoklu İş Parçacığı Modelleri

- Çok a bir (Many-to-One)
- Bire bir (One-to-One)
- Çok a Çok (Many-to-many)



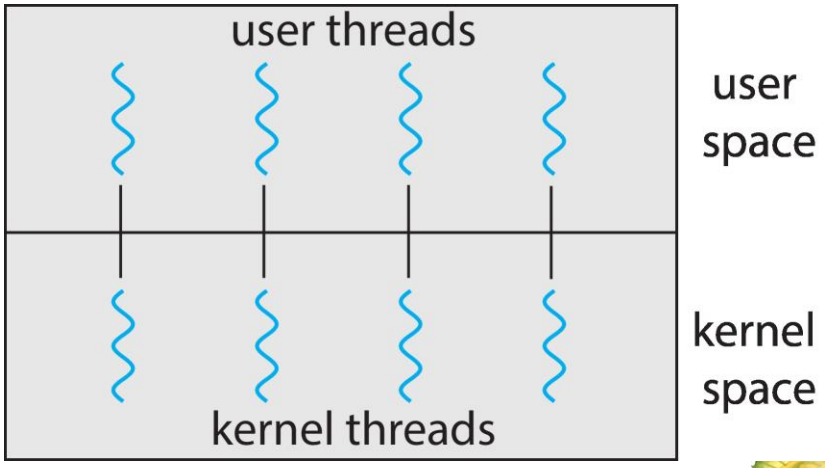
Çok a bir

- Birçok kullanıcı düzeyinde iş parçacığı tek bir çekirdek iş parçacığına den düşürülür
- Bir iş parçacığını engelleme tümünü engellemeye sebep olur
- Birden çok iş parçacığı aynı anda yalnızca bir iş parçacığı çekirdekte olabileceği için çok çekirdekli bir sistemde paralel olarak çalışmayabilir,
- Şu anda bu modeli kullanan çok az sistem bulunmakta
- Örnekler:
  - **Solaris Yeşil İş Parçacıkları**
  - **GNU Taşınabilir İş Parçacıkları**



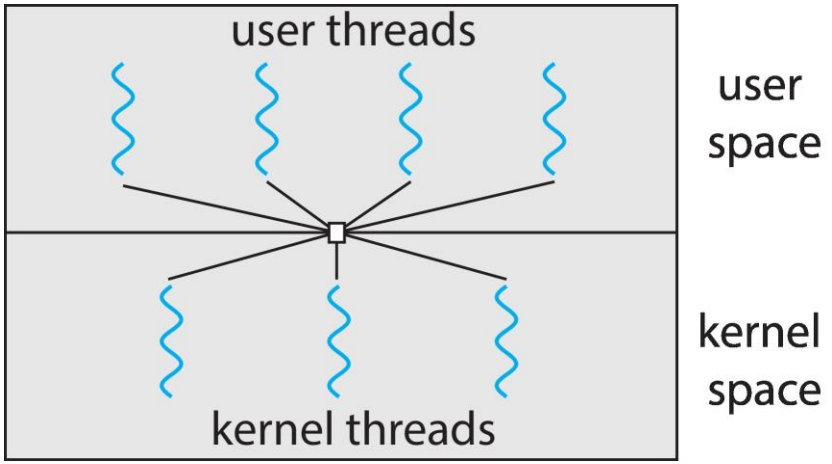
Bire bir

- Her kullanıcı düzeyinde iş parçacığı bir adet çekirdek iş parçacığı ile eşleşir
- Kullanıcı düzeyinde iş parçacığı oluşturma çekirdek iş parçacığı oluşturur
- Çoktan bire modeline göre daha fazla eşzamanlılık
- Proses başına iş parçacığı sayısı bazen maliyet nedeniyle kısıtlanır
- Örnekler
  - Windows
  - Linux



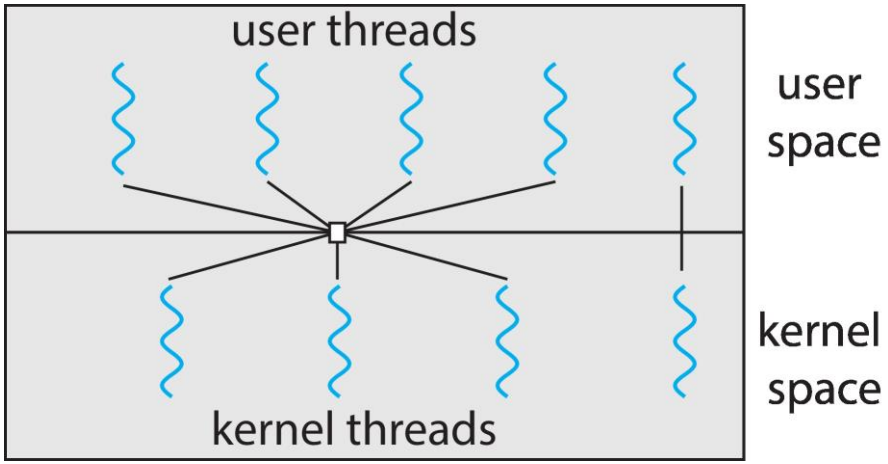
Çokdan-Çoka Modeli

- Birçok kullanıcı düzeyinde iş parçacığı birçok çekirdek iş parçacığı ile eşleşir
- İşletim sisteminin yeterli sayıda çekirdek iş parçacığı oluşturmasına izin verir
- Windows *ThreadFiber* paketi
- çok yaygın değil



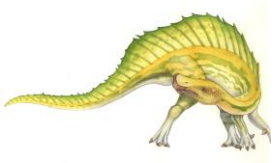
İki Seviyeli Model

- M:M'e benzer, ancak bir kullanıcı iş parçacığının çekirdek iş parçacığına **bağlanır**



İş Parçacığı Kütüphaneleri

- **İş Parçacığı kütüphaneleri** programcılara API yardımıyla iş parçacığı oluşturma ve yönetmeyi sağlar
- Uygulamanın iki temel yolu
  - Tamamen kullanıcı alanında kütüphane
  - OS tarafından desteklenen çekirdek düzeyli kütüphane

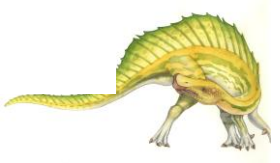


Pthreads

- Kullanıcı düzeyi veya çekirdek düzeyi olarak sağlanabilir
- İş parçacığı oluşturma ve senkronizasyon için bir POSIX standardı (IEEE 1003.1c) API
- ***Uygulama** değil **Şartname***
- API iş parçacığı kütüphane davranışını tanımlar, uygulama kütüphanenin geliştirilmesi ile ilgilidir
- UNIX işletim sistemlerinde yaygın (Linux & Mac OS X)

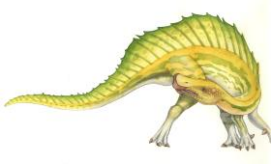
Pthreads Örneği

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```



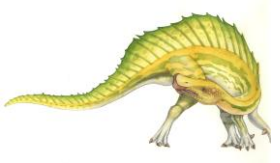
Pthreads Örneği (devam)

```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```



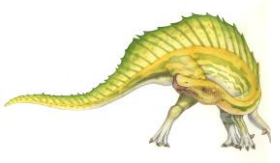
10 adet İş Parçacığını Birleştiren Pthreads Kodu

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```



Windows Çoklu İş parçacığı C Programı

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```



Windows Çoklu İş parçacığı C Programı (devam)

```
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

   printf("sum = %d\n",Sum);
}
```

Java İş Parçacıkları

- Java iş parçacıkları JVM tarafından yönetilir
- Genellikle altta yatan işletim sistemi tarafından sağlanan iş parçacıkları modeli kullanılarak uygulanır
- Java iş parçacıkları aşağıdaki şekilde oluşturulabilir:
  - Thread sınıfnın türetilmesi
  - Runnable arayüzünün uygulanması

```
public interface Runnable
{
    public abstract void run();
}
```

  - Standart uygulama Runnable arabirimi uygulamaktır (çoklu kalıtımdan dolayı)

Java İş Parçacıkları

Runnable arayüzü uygulama:

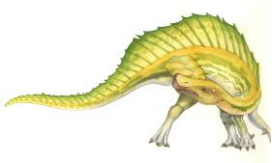
```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

İş parçacığı oluşturma:

```
Thread worker = new Thread(new Task());
worker.start();
```

Bir iş parçacığı bekleniyor:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```



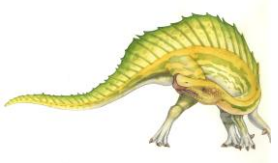
Java Executer Çerçevesi

- Java, iş parçacığı oluşturmak yerine, Executor arabirimi etrafında iş parçacığı oluşturulmasına da izin verir:

```
public interface Executor
{
   void execute(Runnable command);
}
```

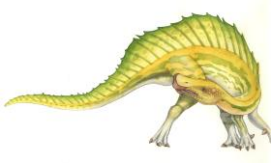
- Executer aşağıdaki gibi kullanılır:

```
Executor service = new Executor;
service.execute(new Task());
```



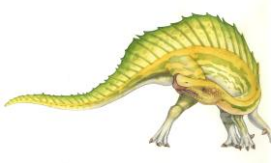
Java Executer Çerçevesi

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
   private int upper;
   public Summation(int upper) {
      this.upper = upper;
   }

   /* The thread will execute in this method */
   public Integer call() {
      int sum = 0;
      for (int i = 1; i <= upper; i++)
         sum += i;

      return new Integer(sum);
   }
}
```



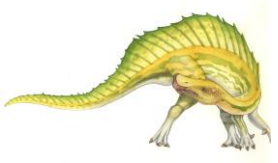
Java Executor Çerçevesi (devam)

```
public class Driver
{
 public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

    try {
        System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
  }
}
```



Kapalı İş Parçacığı

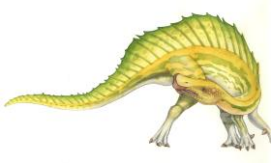
- İş parçacığı sayısı arttıkça popülaritesi artıyor, program doğruluğu açık iş parçacıkları ile daha zor
- İş parçacıklarının oluşturulması ve yönetimi programcılar yerine derleyiciler ve çalışma zamanı kütüphaneleri tarafından yapılıyor
- Beş yöntem;
  - İş parçacığı havuzları
  - Fork-join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading İnşa Blokları



İş Parçacığı Havuzları

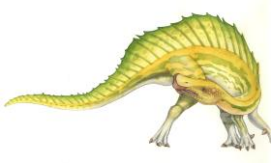
- Bir havuz içerisinde çalışmayı bekleyen iş parçacıkları
- Avantajları:
  - Bir isteği yeni bir iş parçacığı oluşturmaktan daha hızlı bir şekilde varolan bir iş parçacığıyla yerine getirmek daha hızlıdır
  - Uygulama(lar)daki iş parçacığı sayısının havuz boyutuna bağlanmasını sağlar
  - Yerine getirilecek görevlerin görev oluşturma mekaniğinden ayırılması, görevi çalıştırmak için farklı stratejiler sağlar
    - Yani, görevler periyodik olarak çalışacak şekilde zamanlanabilir
- Windows API iş parçacığı havuzlarını destekler:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```



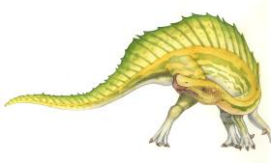
Java İş Parçacığı Havuzları

- Executors sınıfında iş parçacığı havuzları oluşturmak için üç yöntem:
  - `static ExecutorService newSingleThreadExecutor()`
  - `static ExecutorService newFixedThreadPool(int size)`
  - `static ExecutorService newCachedThreadPool()`



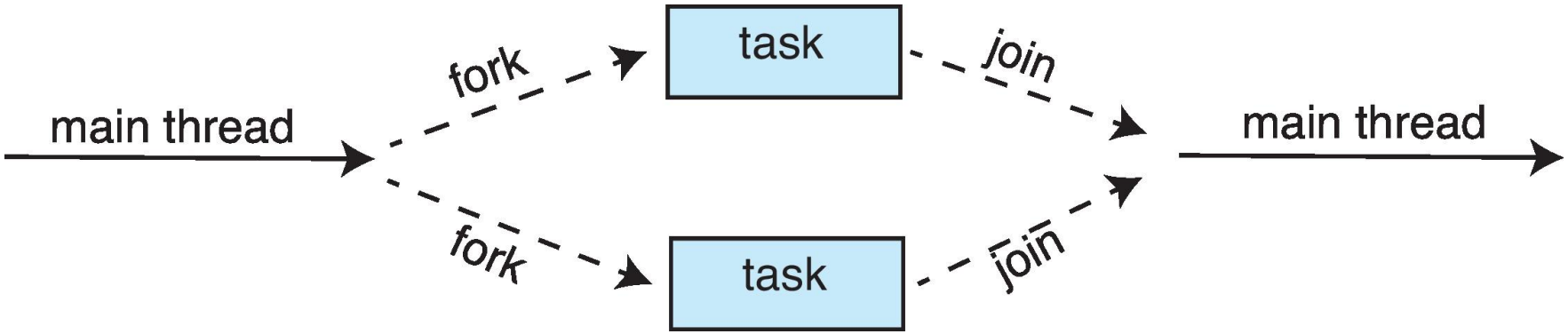
Java İş Parçacığı Havuzları (devam)

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
public static void main(String[] args) {
   int numTasks = Integer.parseInt(args[0].trim());

   /* Create the thread pool */
   ExecutorService pool = Executors.newCachedThreadPool();

   /* Run each task using a thread in the pool */
   for (int i = 0; i < numTasks; i++)
      pool.execute(new Task());

   /* Shut down the pool once all threads have completed */
   pool.shutdown();
}
```



Fork-Join Paralelliği

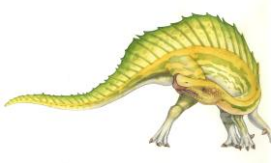
- Birden çok iş parçacığı (görev) **oluşturulur (fork)**, ve sonra **birbirine bağlanır(join)**.



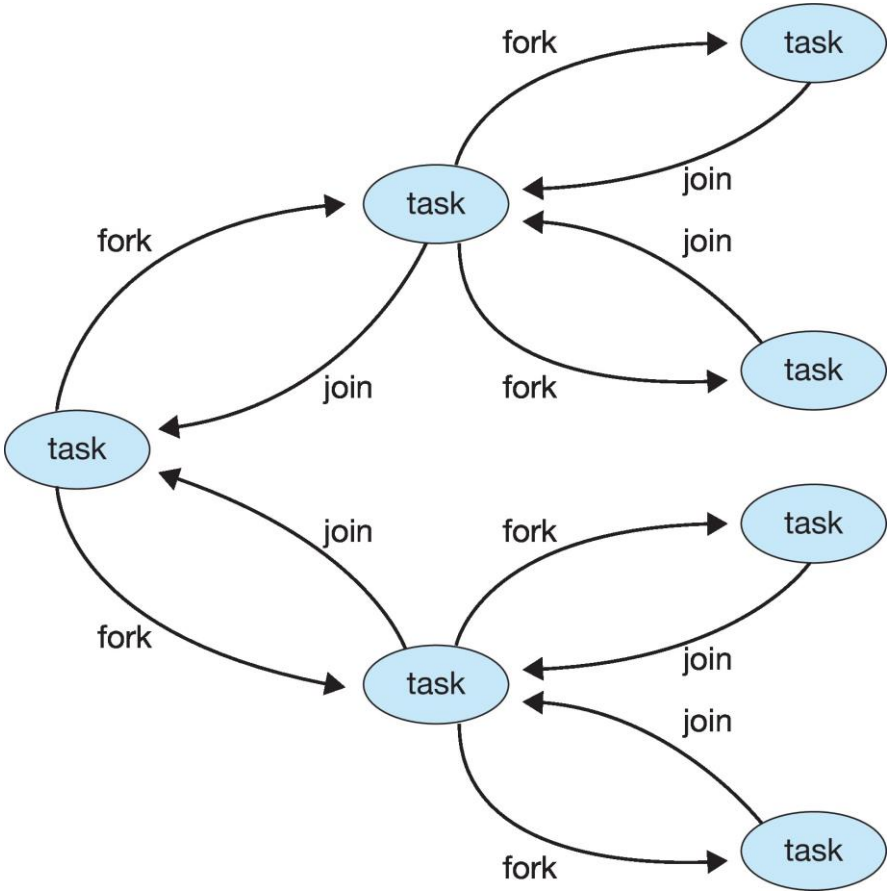
Fork-join Paralelliği

- Fork-join stratejisi için genel algoritma:

```
Task(problem)
   if problem is small enough
     solve the problem directly
   else
     subtask1 = fork(new Task(subset of problem)
     subtask2 = fork(new Task(subset of problem)

     result1 = join(subtask1)
     result2 = join(subtask2)

     return combined results
```

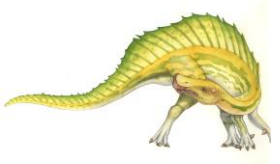


Fork-join Paralelliği



Java'da Fork-join Paralelliği

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

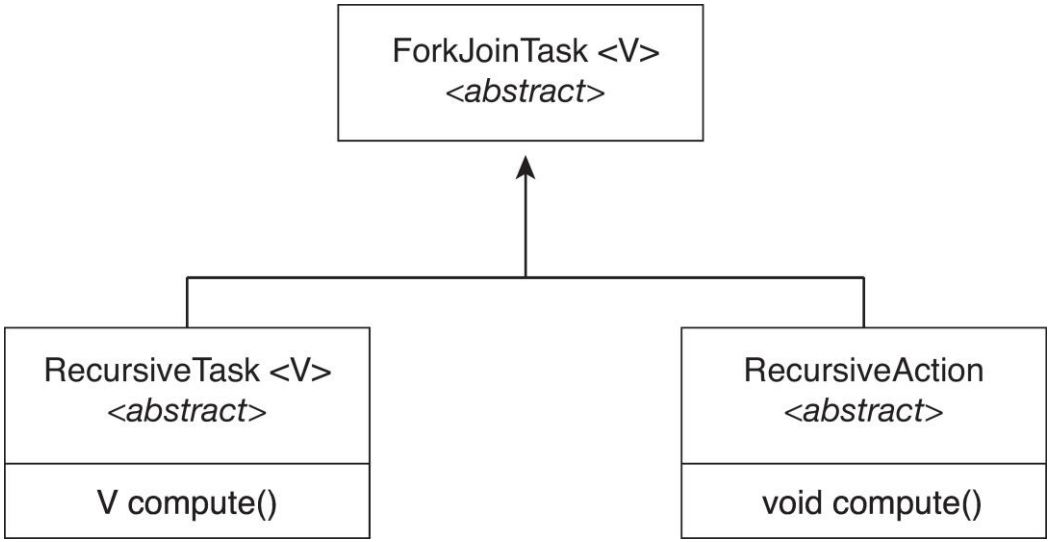


Java'da Çatal-Birleştirme Paralelliği

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
   static final int THRESHOLD = 1000;

   private int begin;
   private int end;
   private int[] array;

   public SumTask(int begin, int end, int[] array) {
      this.begin = begin;
      this.end = end;
      this.array = array;
   }

   protected Integer compute() {
      if (end - begin < THRESHOLD) {
         int sum = 0;
         for (int i = begin; i <= end; i++)
            sum += array[i];

         return sum;
      }
      else {
         int mid = (begin + end) / 2;

         SumTask leftTask = new SumTask(begin, mid, array);
         SumTask rightTask = new SumTask(mid + 1, end, array);

         leftTask.fork();
         rightTask.fork();

         return rightTask.join() + leftTask.join();
      }
   }
}
```

Java'da Fork-join Paralelliği

- Bu, **`ForkJoinTask`** soyut bir sınıftır
- **`RecursiveTask`** ve **`RecursiveAction`** sınıfları **`ForkJoinTask`** sınıfını genişletir
- **`RecursiveTask`** bir sonuç geri döndürür (**`compute()`** metodundan dönüş değeri üzerinden)
- **`RecursiveAction`** sonuç döndürmez



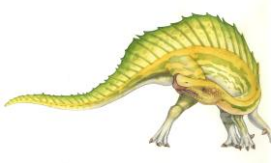
OpenMP

- Derleyici yönergeleri ve C, C++, FORTRAN için API
- Paylaşılan bellek ortamlarında paralel programlama desteği sağlar
- **paralel bölgeler** tanımlar - paralel olarak çalıştırılabilen kod blokları

`#pragma omp paralel`

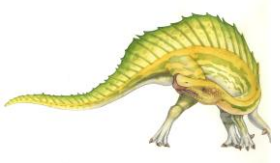
Çekirdekler kadar sayıda iş parçacığı oluşturur

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```



- For döngüsüne paralel olarak çalıştırma

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

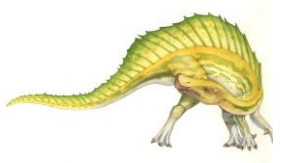


Büyük Merkezi Görevlendirme - Grand Central Dispatch

- macOS ve iOS işletim sistemleri için bir Apple teknolojisi
- C, C++ ve Objective-C dillerinin uzantıları, API ve çalışma zamanı kütüphanesi
- Paralel bölümlerin tanımlanmasını sağlar
- İş parçacığı ayrıntılarının çoğunu yönetir
- Blok içinde "^{ }" :

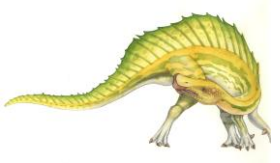
```
{ printf("Ben bir bloğum"); }
```

- Bloklar görevlendirme kuyruğuna yerleştirilir
  - Kuyruktan kaldırıldığında iş parçacığı havuzunda mevcut bir iş parçacığına atanır



Grand Central Dispatch

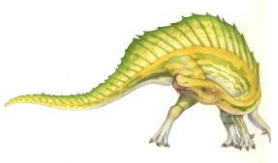
- İki tür görevlendirme sırası:
  - **Seri** – FIFO sırasına göre kaldırılan bloklar, proses başına kuyruk **ana kuyruk** olarak adlandırılır
    - Programcılar program içinde ek seri kuyrukları oluşturabilir
  - **Eşzamanlı** – FIFO sırasına göre kaldırılır, ancak bir seferde birkaç tane
    - Hizmet kalitesine göre bölünmüş dört sistem kuyruğu:
      - `QOS_CLASS_USER_INTERACTIVE`
      - `QOS_CLASS_USER_INITIATED`
      - `QOS_CLASS_USER_UTILITY`
      - `QOS_CLASS_USER_BACKGROUND`



Grand Central Dispatch

- Swift dili için bir görev kapatma (closure) olarak tanımlanır – blok benzeri,
- Kapatmalar `dispatch_async()` fonksiyonu kullanılarak kuyruğa gönderilir:

```
let queue = dispatch_get_global_queue
   (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue,{ print("I am a closure.") })
```



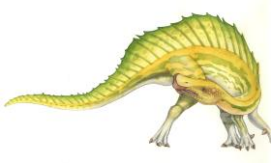
Intel İş Parçacığı İnşa Blokları (Threading Building Blocks - TBB)

- Paralel C++ programları tasarlamak için şablon kütüphanesi
- Basit bir döngü için bir seri versiyonu

```
for (int i = 0; i < n; i++) {
   apply(v[i]);
}
```

- Aynı döngünün TBB kullanılarak **`parallel_for`** deyimi ile yazılması:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```



İş Parçacığı Sorunları

- **fork()** ve **exec()** sistem çağrılarının semantiği
- Sinyal işleme
  - Senkron ve asenkron
- Hedef iş parçacığının iş parçacığı iptali
  - Asenkron veya ertelenmiş
- İş parçacığı yerel depolama
- Sıralayıcı Aktivasyonları

fork() ve exec() Semantiği

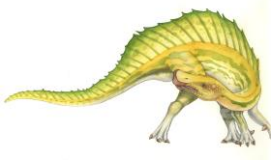
- `fork()` yalnızca çağıran iş parçacığını mı yoksa tüm iş parçacıklarını mı çoğaltır?
  - Bazı UNIXler fork iki farklı sürüme sahiptir
- `exec()` genellikle normal olarak çalışır – tüm iş parçacıklarını içeren çalışan prosesi değiştirir

Sinyal İşleme

- **Sinyal** UNIX sistemlerinde belirli bir olayın meydana geldiğinden bir prosesi haberdar etmek için kullanılır.
- **Sinyal işleyicisi** sinyalleri yönetmek için kullanılır
  1. Sinyal belirli bir olay tarafından oluşturulur
  2. Sinyal bir prosese teslim edilir
  3. Sinyal, iki sinyal işleyiciden biri tarafından yönetilir:
     1. Varsayılan
     2. kullanıcı tanımlı
- Her sinyal **varsayılan işleyiciye** sahiptir ve sinyali işlerken çekirdek tarafından çalıştırılır
  - **Kullanıcı tanımlı sinyal işleyicisi** varsayılanı geçersiz kılabilir
  - Tek iş parçacığı için, sinyal prosese teslim edilir

Sinyal İşleme (devam)

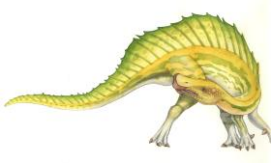
- Çoklu iş parçacığı modelinde sinyal nereye teslim edilmelidir?
  - Sinyali sinyalin uygulandığı iş parçacığına teslim et
  - Sinyali proses içindeki her iş parçacığına teslim et
  - Sinyali prosesteki belirli iş parçacıklarına teslim et
  - Belirli bir iş parçacığı prosesteki tüm sinyalleri almak için atama



İş Parçacığını İptal Etme

- İş parçacığının bitmeden sona erdirilmesi
- İptal edilecek iş parçacığı **hedef iş parçacığı**
- İki genel yaklaşım:
  - **Asenkron iptal** hedef iş parçacığı hemen sona erer
  - **Ertelenmiş iptal** hedef iş parçacığının iptal edilip edilmemesi gerektiğini periyodik olarak denetlemesini sağlar
- Bir iş parçacığı oluşturmak ve iptal etmek için Pthread kodu:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

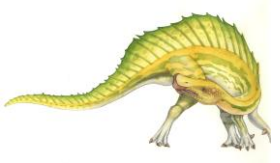


İş Parçacığını İptal Etme (devam)

- İş parçacığı iptali isteklerini iptal etmek, ancak gerçek iptal iİş parçacığı durumuna bağlıdır

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- İş parçacığı iptal devre dışı bırakılmışsa, iş parçacığı etkinleştirilene kadar iptal beklemede kalır
- Varsayılan tür ertelemeli iptaldir
  - İptal işlemi yalnızca iş parçacığı **iptal noktasına** ulaştığında gerçekleşir
    - Yani. **`pthread_testcancel()`**
    - Sonra **temizleme işleyicisi** çağrılır
- Linux sistemlerinde, iş parçacığı iptali sinyaller aracılığıyla gerçekleştirilir



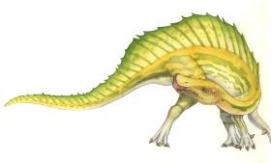
Java'da İş Parçacığı İptali

- Ertelenmiş iptal, **`interrupt() metodunu kullanır. Bu metod`** iş parçacığının durumunu iptal edilmiş olarak belirler.

```
Thread worker;

. . .

/* set the interruption status of the thread */
worker.interrupt()
```

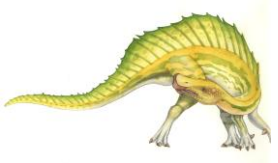
- Bir iş parçacığı daha sonra kesintiye uğrayıp uğramadığını denetleyebilir:

```
while (!Thread.currentThread().isInterrupted()) {
    . . .
}
```



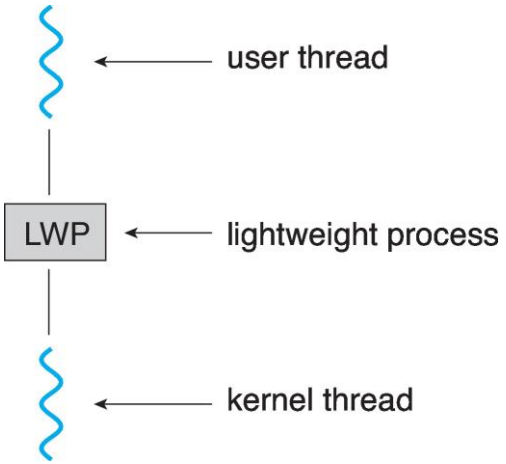
İş Parçacığı-Yerel Depolama

- **İş parçacığı yerel depolama** (**Thread-local storage - TLS**) her iş parçacığının kendi veri kopyasına sahip olmasını sağlar
- İş parçacığı oluşturma işlemi üzerinde denetim olmadığında kullanışlı (örneğin, iş parçacığı havuzu kullanırken)
- Yerel değişkenlerden farklı
  - Yalnızca tek işlev çağırma sırasında görünen yerel değişkenler
  - TLS işlev çağrıları arasında görünür
- **`statik`** veriye benzer
  - TLS her iş parçacığına özgüdür



Sıralayıcı Aktivasyonları

- Hem M:M hem de İki düzeyli modeller, uygun sayıda çekirdek iş parçacığının uygulamaya ayrılmasını korumak için iletişim gerektirir
- Genellikle kullanıcı ve çekirdek iş parçacıkları arasında bir ara veri yapısı kullanın – **hafif proses** (**lightweight process** - **LWP)**
  - Hangi prosesin kullanıcı iş parçacığı çalıştırıracağı sanal bir işlemci vazifesi görür
  - Her LWPÇekirdek iş parçacığına bağlanır
  - Kaç LWP oluşturulmalı?
- Zamanlayıcı aktivasyonları, **upcalls** – bir iş parçacığı kütüphanesinde çekirdekten **upcall işleyicisine** doğru bir iletişim mekanizması
- Bu iletişim, bir uygulamanın doğru sayıda çekirdeği iş parçacığına sahip olmasını sağlar

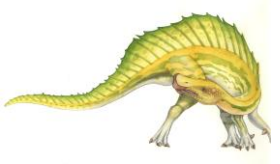


İşletim Sistemi Örnekleri

- Windows İş Parçacıkları
- Linux İş Parçacıkları

Windows İş Parçacıkları

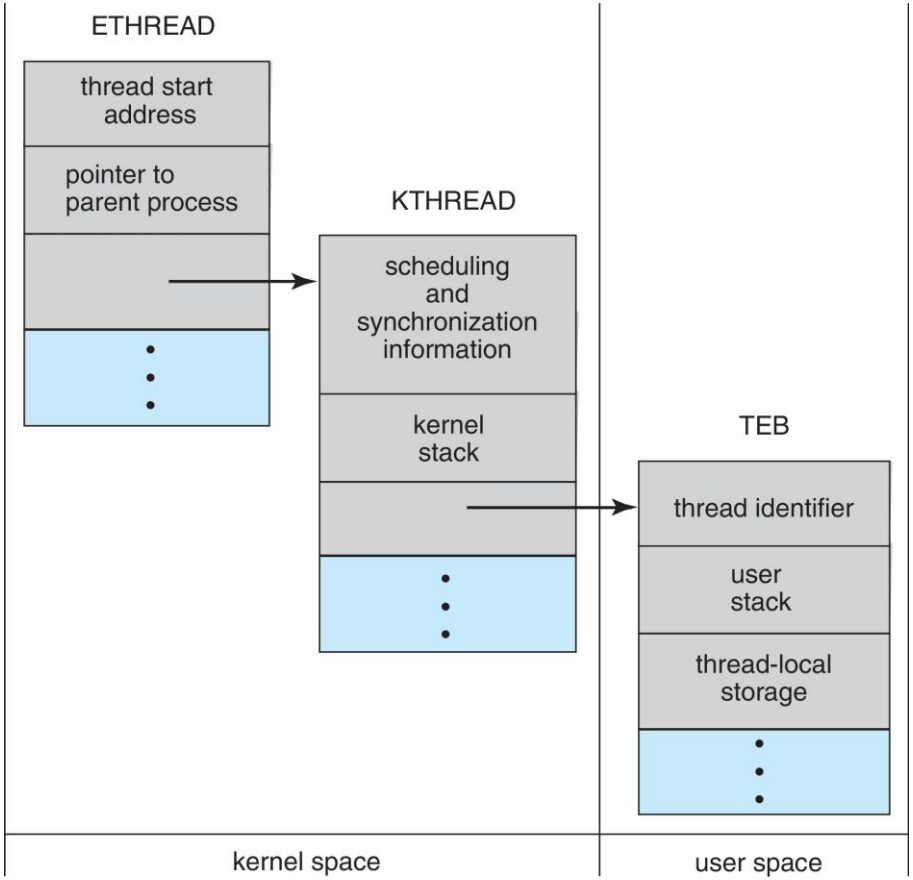
- Windows API – Windows uygulamaları için birincil API
- Bire bir eşleme, çekirdek düzeyinde uygular
- Her iş parçacığı aşağıdakileri içerir:
  - Bir iş parçacığı id si
  - İşlemci durumunu temsil eden kaydedici
  - İş parçacığı kullanıcı modunda veya çekirdek modunda çalıştığında kullanılan ayrı kullanıcı ve çekirdek yığınları
  - Çalışma zamanı kütüphaneleri ve dinamik bağlantı kütüphaneleri (DLLler) tarafından kullanılan özel veri depolama alanı
- Kayıt kümesi, yığınlar ve özel depolama alanı iş parçacığının **bağlamı(context)** olarak adlandırılır.

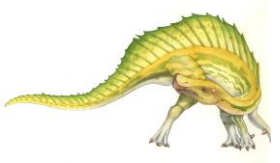


Windows İş Parçacıkları (devam)

- Bir iş parçacığının birincil veri yapıları şunlardır:
  - ETHREAD (executive thread block) – çekirdek alanında iş parçacığının ait olduğu prosese ve KTHREAD'e pointer içerir
  - KTHREAD (çekirdek iş parçacığı bloğu) – sıralama ve senkronizasyon bilgileri, çekirdek modu yığını, TEB işaretçisi, çekirdek alanında
  - TEB (iş parçacığı çevre bloğu) – iş parçacığı id, kullanıcı modu yığını, iş parçacığı yerel depolama, kullanıcı alanında

Windows İş Parçacıkları Veri Yapıları





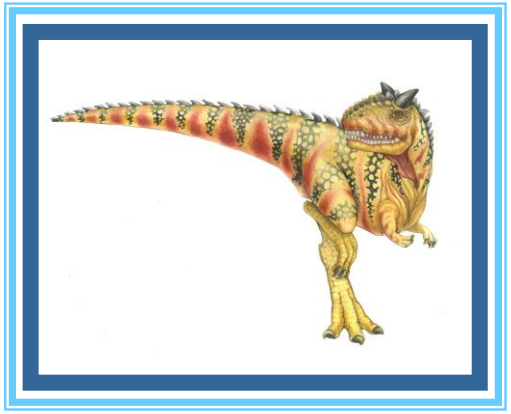
Linux İş Parçacıkları

- Linux iş parçacıkları: ***Görev*** (task)
- İş parçacığı oluşturma **`clone()`** sistem çağrısı yoluyla yapılır
- **`clone ()`** bir alt görevin üst görevin adres alanını (işlem) paylaşmasına izin verir
  - Davranışı kontrol eden bayraklar

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **`struct task_struct`** proses veri yapılarını gösterir (paylaşılan veya tek)

Bölüm 4'ün Sonu



Silberschatz, Galvin and Gagne ©2018