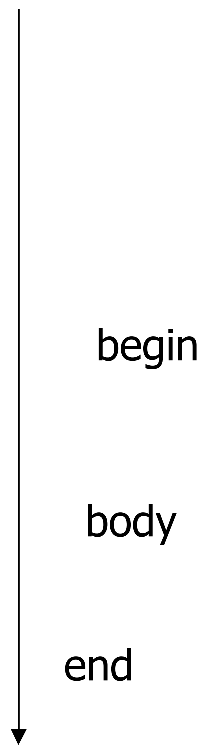
Dağıtık Sistemler

3. Ders

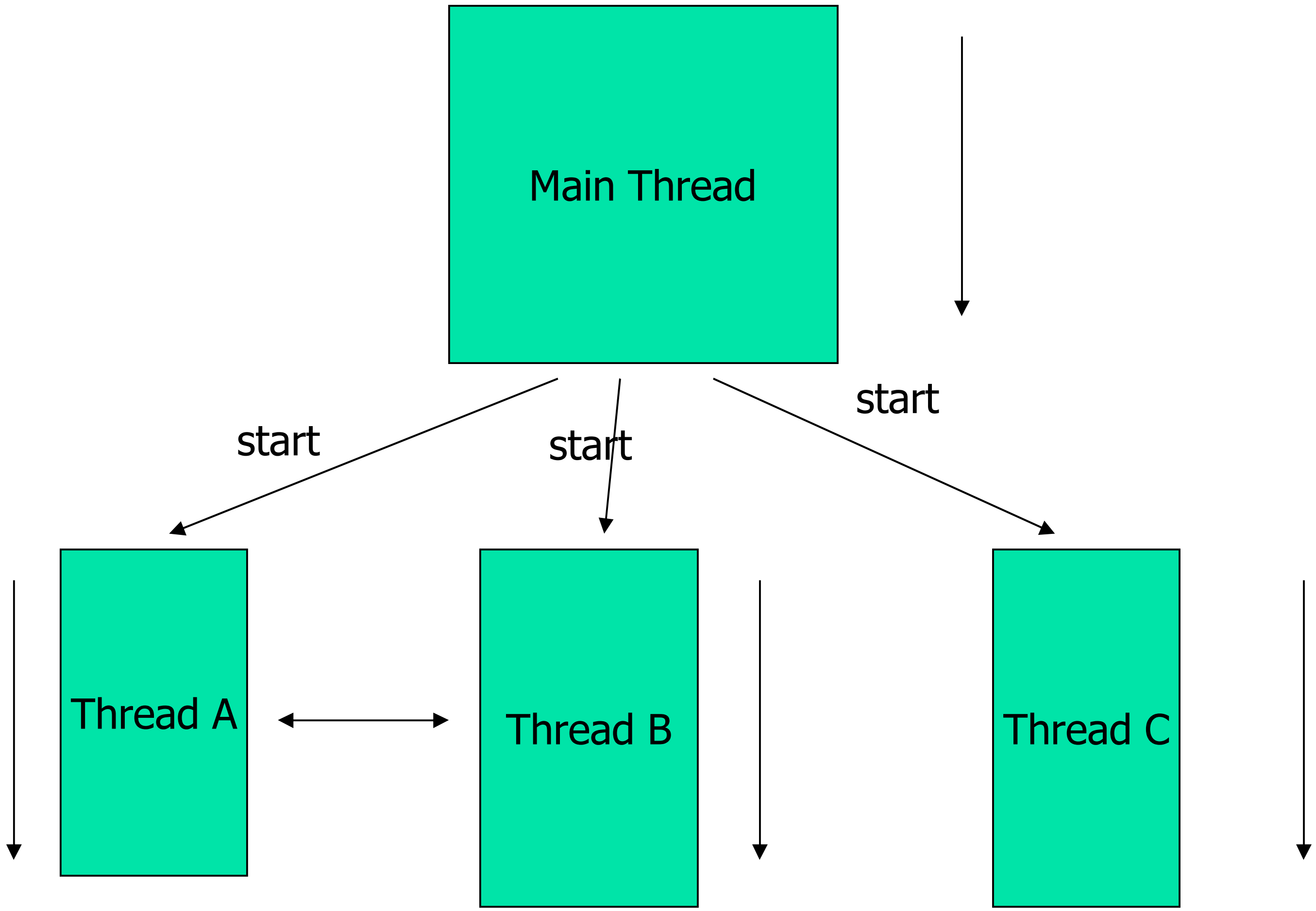
DS için OS Desteği: Görevler (Process) ve Threadler

Tek threadli bir program

```
class ABC
{
....
    public void main(..)
    {
    ...
    ..
    }
}
```



Çok-threadli bir Program



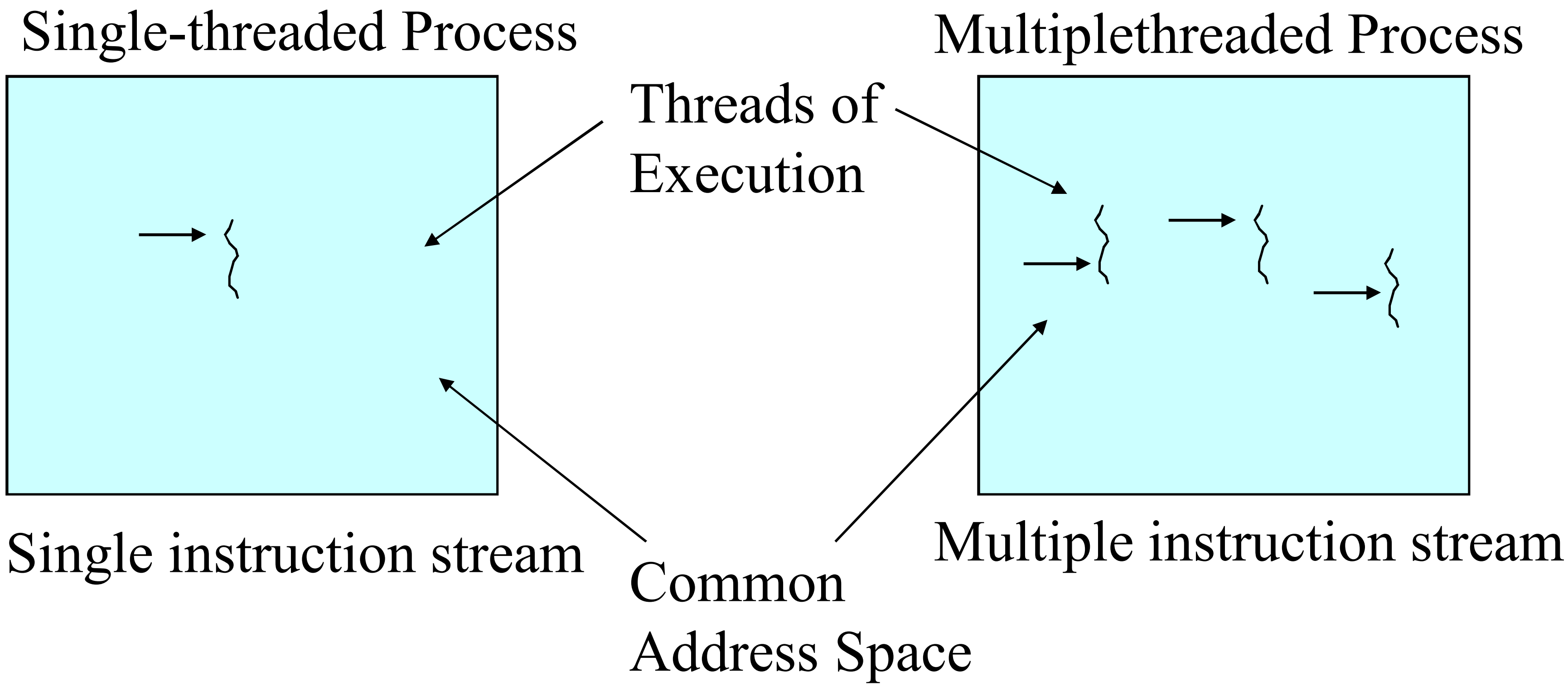
Threads may switch or exchange data/results. JVM allows an application to have multiple threads of execution running concurrently.

Introduction to Threads

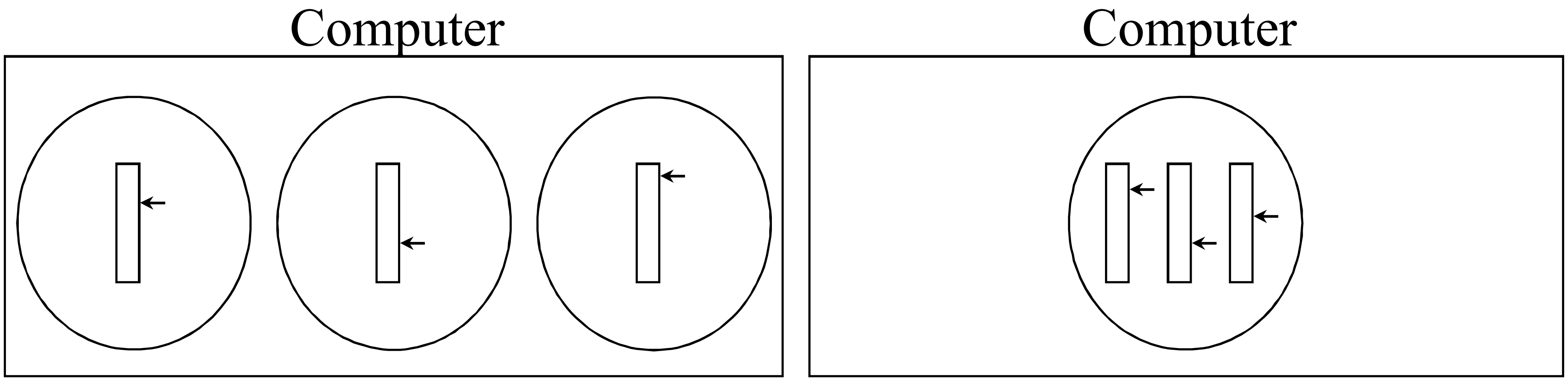
- In traditional system, each process has an address space and a single thread of control.
- There are situations in which it is desirable to have multiple threads of control sharing a single address space but running in quasi parallel.
- A file server has to block waiting for the disk; If the server had multiple threads of control, a second thread could run while the first one was blocked. The net result would be a higher throughput (requests/sec) and better performance.
- It is not possible to achieve this goal by creating two independent server processes because they must share a common buffer cache, which requires them to be in the same address space.
- Therefore, we have threads, and sometimes it is called lightweight processes.
- Each thread runs strictly sequentially and has its own **program counter** and **stack** to keep track of where it is.
- Threads share the CPU just as processes do. Only on multiprocessor do they actually run in parallel.
- Threads can create child threads and can block waiting for system calls to complete, just like regular processes.

Single and Multithreaded Processes
Tek ve Çok threadli Processler

threads are light-weight processes within a process
threadler, bir process içindeki hafif processlerdir



Threads Vs. Processes



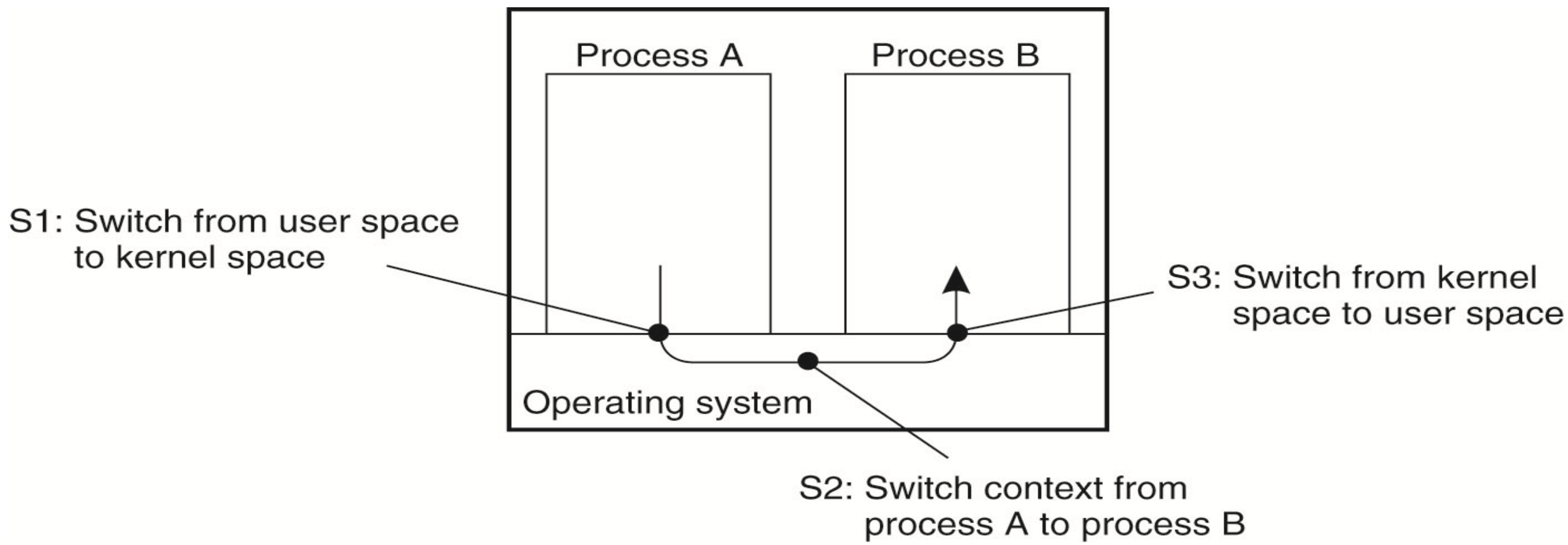
Per thread items
Program counter Stack Register set Child threads State

Per process items
Address space Global variables Open files Child processes Timers Signals Semaphores Accounting information

Threads vs. Processes

- Blocking system call: process as a whole is blocked; if one thread is blocked, other threads (of same process) can keep running
  - Ex: spreadsheet program:input, update, save
- Parallelism: program exec on (uniprocessor) / multiprocessor system
- Large apps (cooperation by IPC: pipes, msg queue, shared mem-shm): comms requires extensive context switching (next slide)
- Large apps (diff parts are executed by separate threads). Comm b/w parts is dealt with shared data. Thread switch sometimes in user space. Dramatic improvement in performance
- Pure s/w engineering reason to use thread: many apps easier to structure as a collection of threads
  - Ex: word processor: user input, spell/grammer check, doc layout, index generation, save

Context switching as the result of IPC



1. IPC requires kernel intervention. Change mem map, flush TLB
2. Within the kernel, a process switch
3. switch from kernel mode to user mode to activate B. Change mem map, flush TLB

Thread Implementation

- Thread package: operations to create/destroy threads, operations on synch vars (like mutex)
- Two ways to implement a thread package:

1. Thread library that is executed in user mode (**User-Level Threads, ULT**)

   Cheap to create/destroy threads (cost of allocating memory for stack / freeing memory for stack: all in user' address space)

   Switching thread context in a few instructions (CPU Regs save/load, **no need** to change memory maps, flush TLB, CPU accounting.

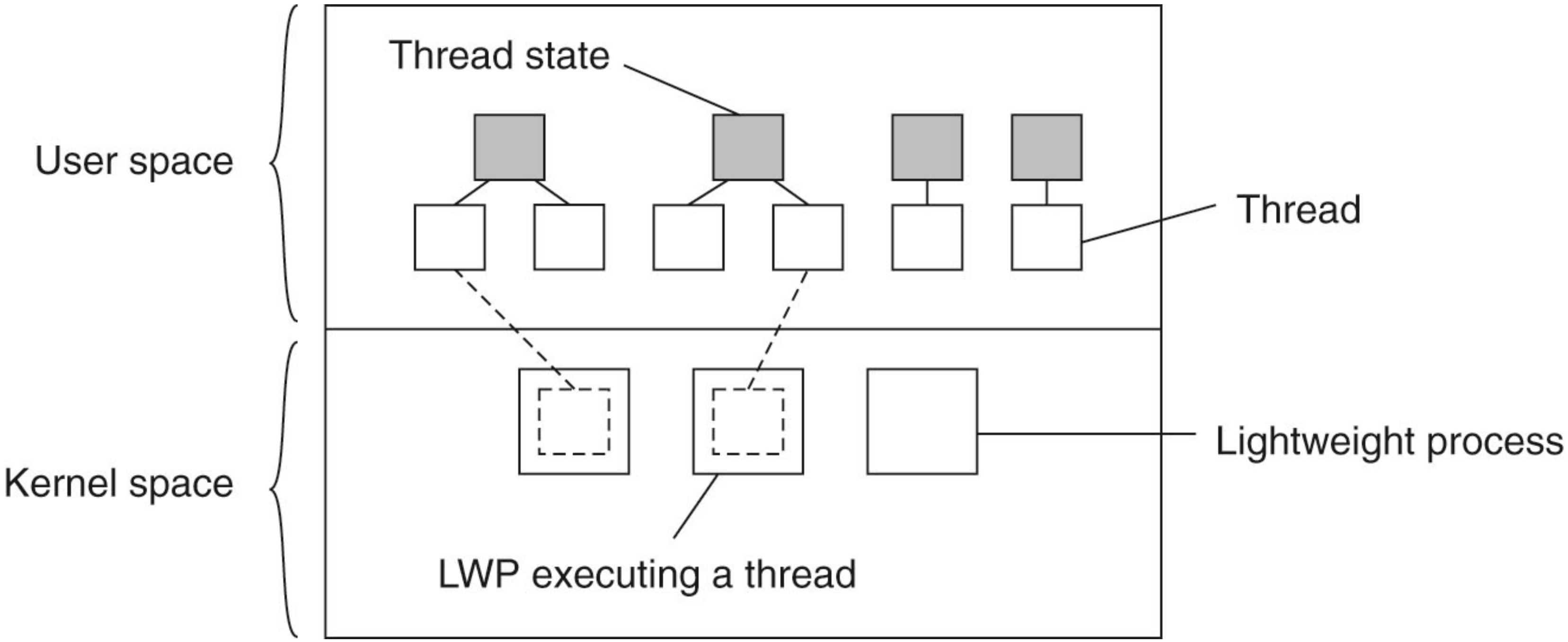
   But, invocation of blocking system call blocks the entire process (all threads in that process)

2. Kernel aware of threads and schedule them (**Kernel-Level Threads, KLT**)

   Thread ops carried out in by kernel (requires system call) (high price)

   Switching thread contexts as epensive as switching process contexts

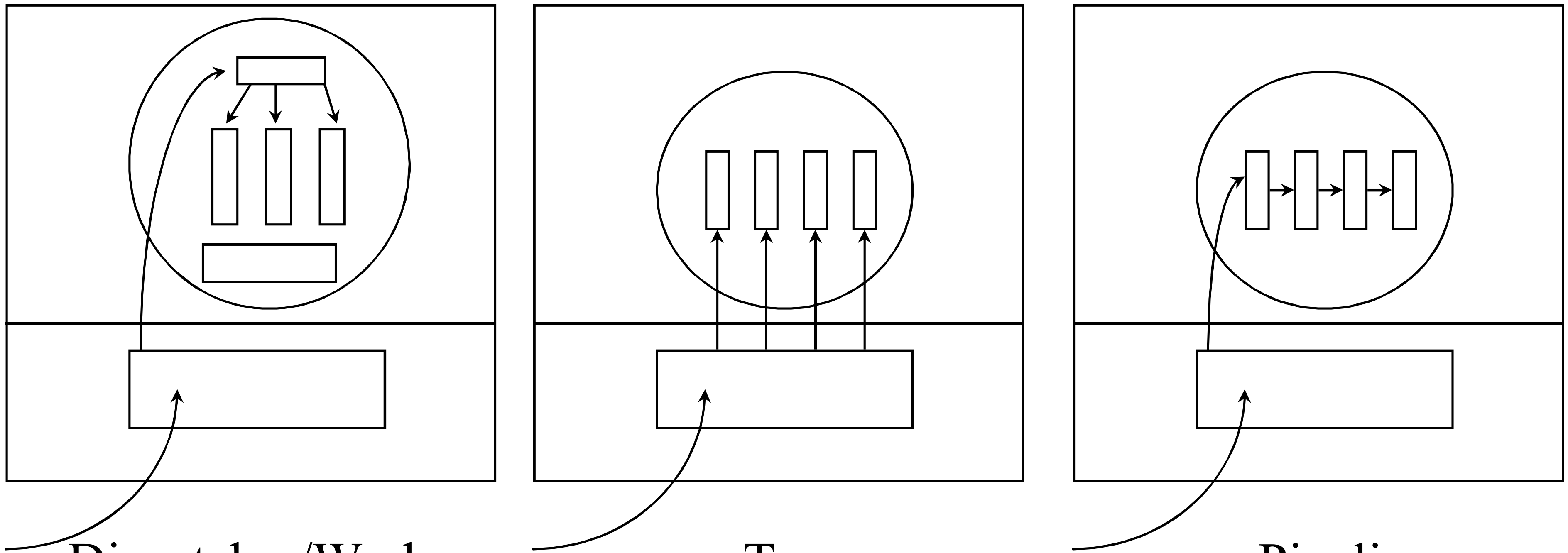
Combining kernel-level lightweight processes and user-level threads



Hybrid form of ULT and KLT

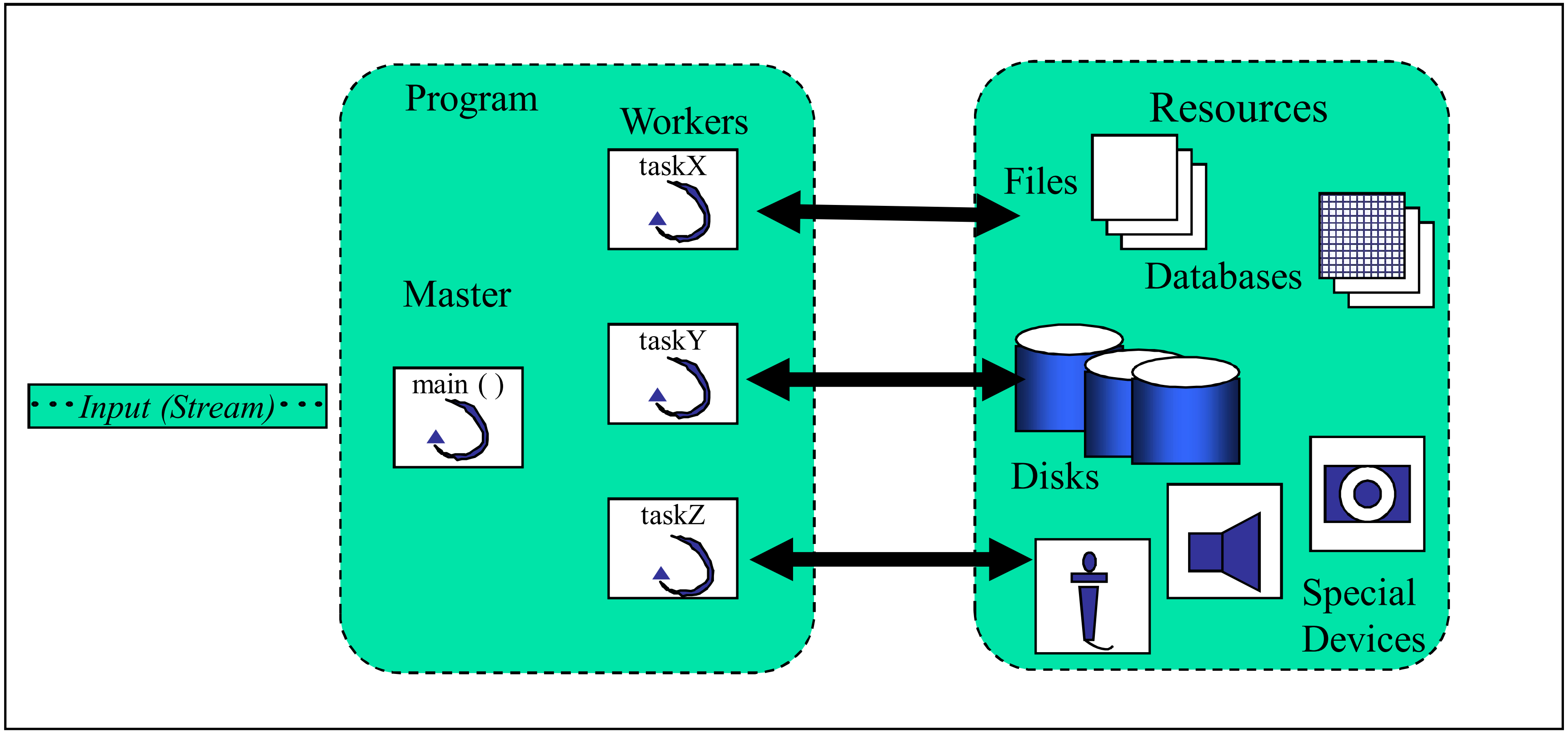
3 Organizations of threads in a process

- Dispatcher/Worker model: One dispatcher and several workers (file server)
- Team model: Some threads are dedicated for special services (interrupt handler thread)
- Pipeline model: The first thread generates some data and passes them on to the next thread for processing and so on (producer-consumer program)



Dispatcher/Worker   Team   Pipeline

dispatcher/worker modeli

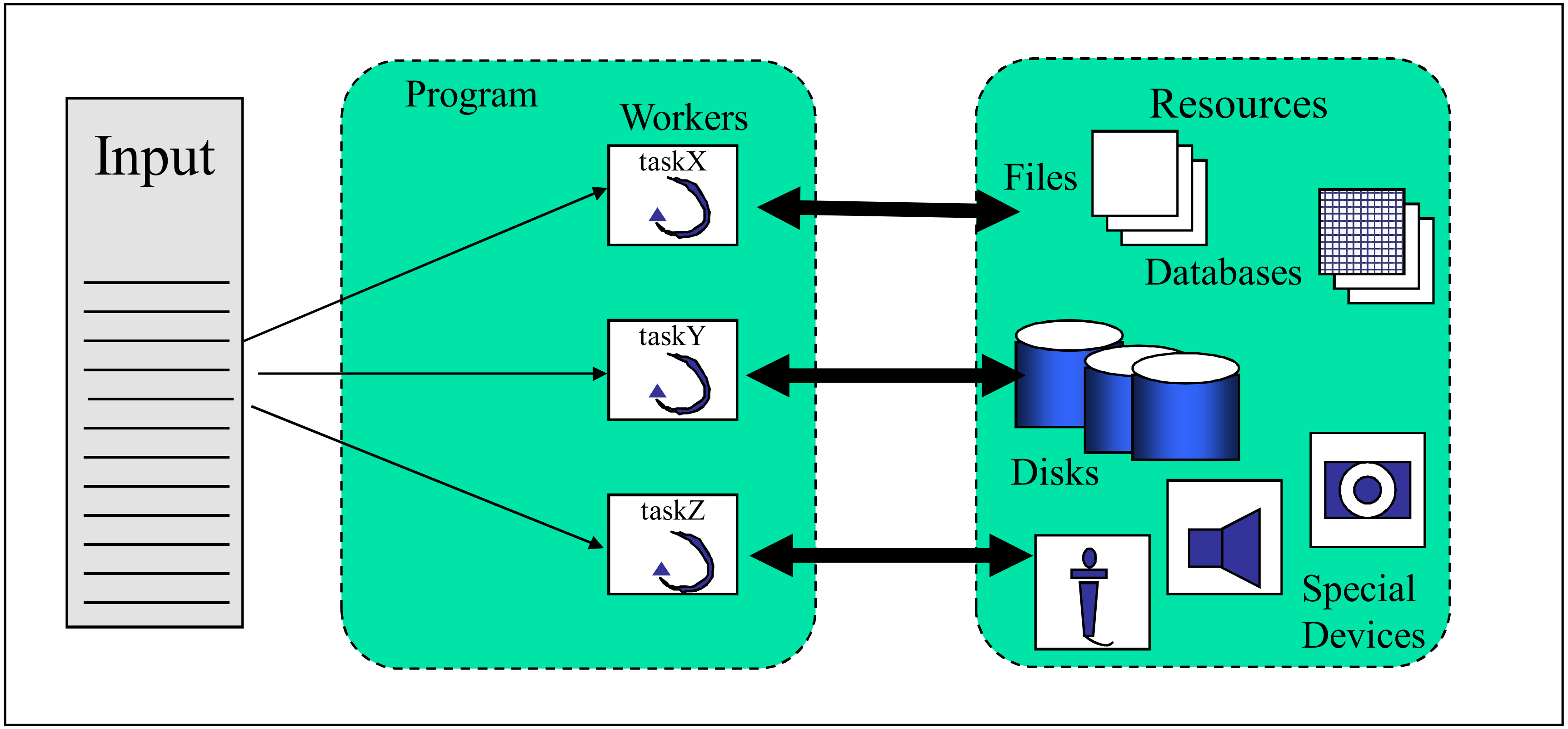


Örnek

```
main() /* the master */
{
 forever {
                    get a request;
                    switch( request )
                    case X: pthread_create(....,taskX);
                    case Y: pthread_create(....,taskY);
                    ....
        }
}
taskX() /* worker */
{
  perform the task, sync if accessing shared resources
  görevi yap, paylaşılan kaynaklara erişiliyorsa senkronize ol
}
taskY() /* worker */
{
  perform the task, sync if accessing shared resources
  görevi yap, paylaşılan kaynaklara erişiliyorsa senkronize ol
}
....
```

- -- Çalışma zamanındaki thread oluşturma ek yükü (overhead), thread havuzu ile hafifletilebilir
- * master thread program başlatılırken tüm worker threadleri oluşturur
- sonra her bir worker thread hemen beklemeye geçip
- master tarafından uyandırılana kadar uykuda bekler

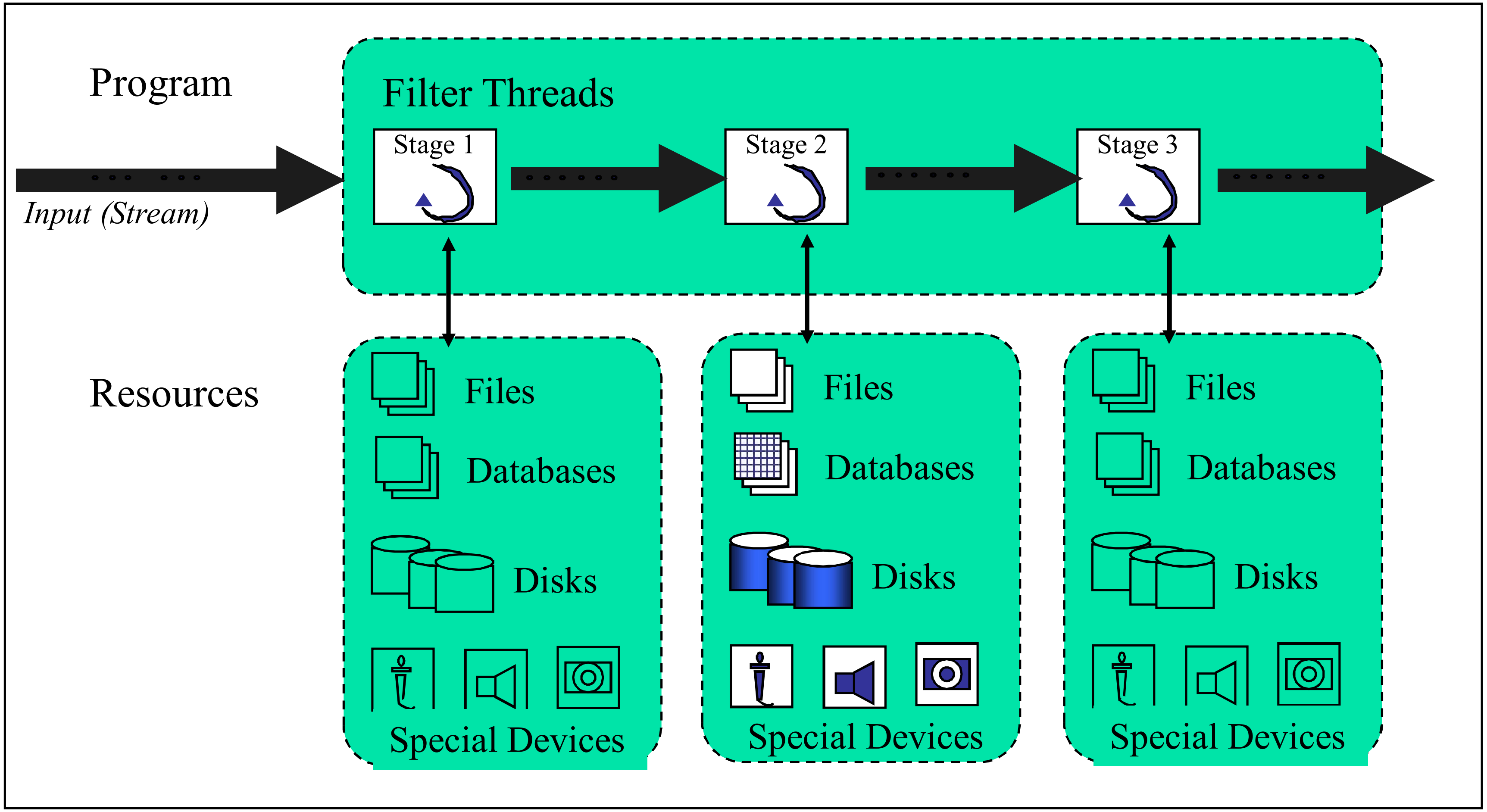
Takım (team, peer) modeli



Örnek

```
main()
{
        pthread_create(....,thread1...taskX);
        pthread_create(....,thread2...taskY);
                ....
        signal all workers to start            Tüm çalışanlara başlama sinyali gönder
        wait for all workers to finish         Tüm çalışanların bitirmelerini bekle
        do any cleanup                         Gereken temizlikleri yap
        }
}
taskX() /* worker */
{
   wait for start
   perform the task, sync if accessing shared resources
}
taskY() /* worker */
{
  wait for start
   perform the task, sync if accessing shared resources
}
```

thread pipeline



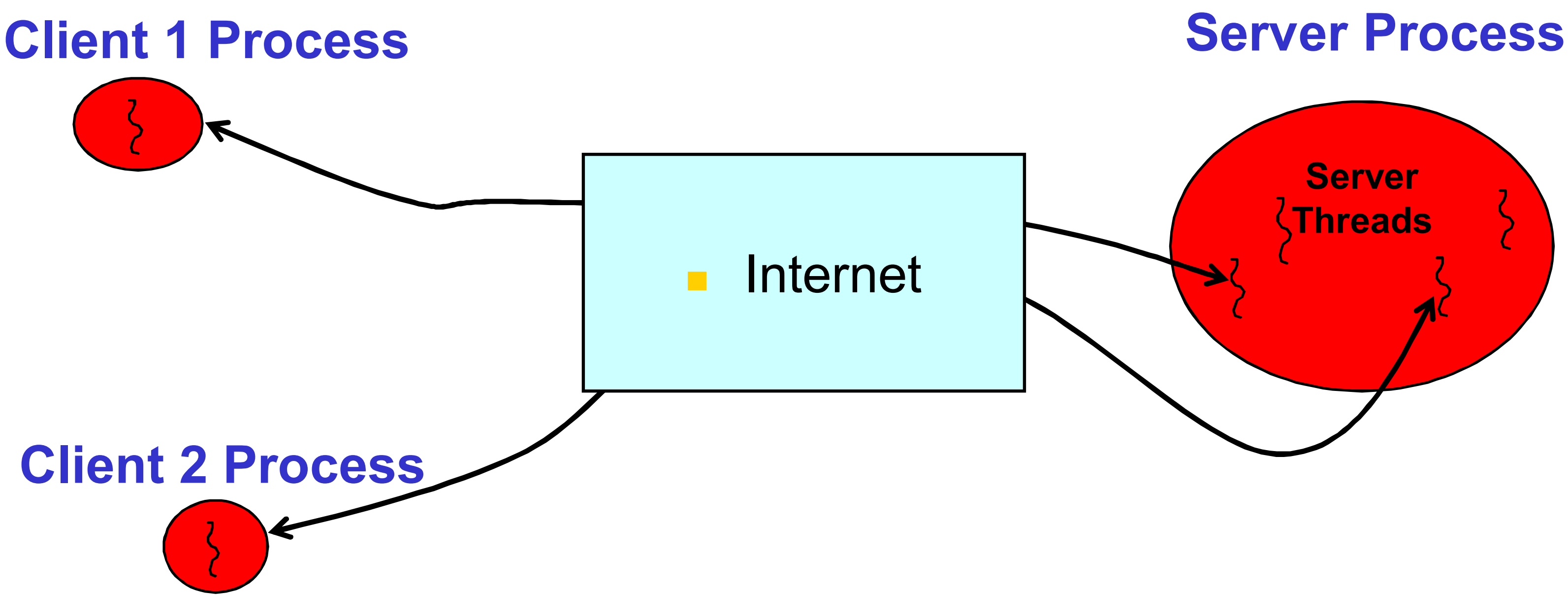
Örnek

```
main()
{
   pthread_create(....,stage1);
   pthread_create(....,stage2);
   ....
   wait for all pipeline threads to finish
          Tüm pipeline threadlerinin bitmesini bekle
   do any cleanup              Gereken temizlikleri yap
}
stage1() {
   get next input for the program      Sıradaki inputu al
   do stage 1 processing of the input   Inputun işlenmesinin 1. kısmını yap
   pass result to next thread in pipeline        Sonucu sıradaki threade yönlendir
}
stage2(){
   get input from previous thread in pipeline     Önceki threadden inputu al
   do stage 2 processing of the input    Inputun işlenmesinin 2. kısmını yap
   pass result to next thread in pipeline         Sonucu sıradaki threade yönlendir
}
stageN()
{
   get input from previous thread in pipeline      Önceki threadden inputu al
   do stage N processing of the input   Inputun işlenmesinin N. kısmını yap
   pass result to program output.       Sonucu  outputa gönder
}
```

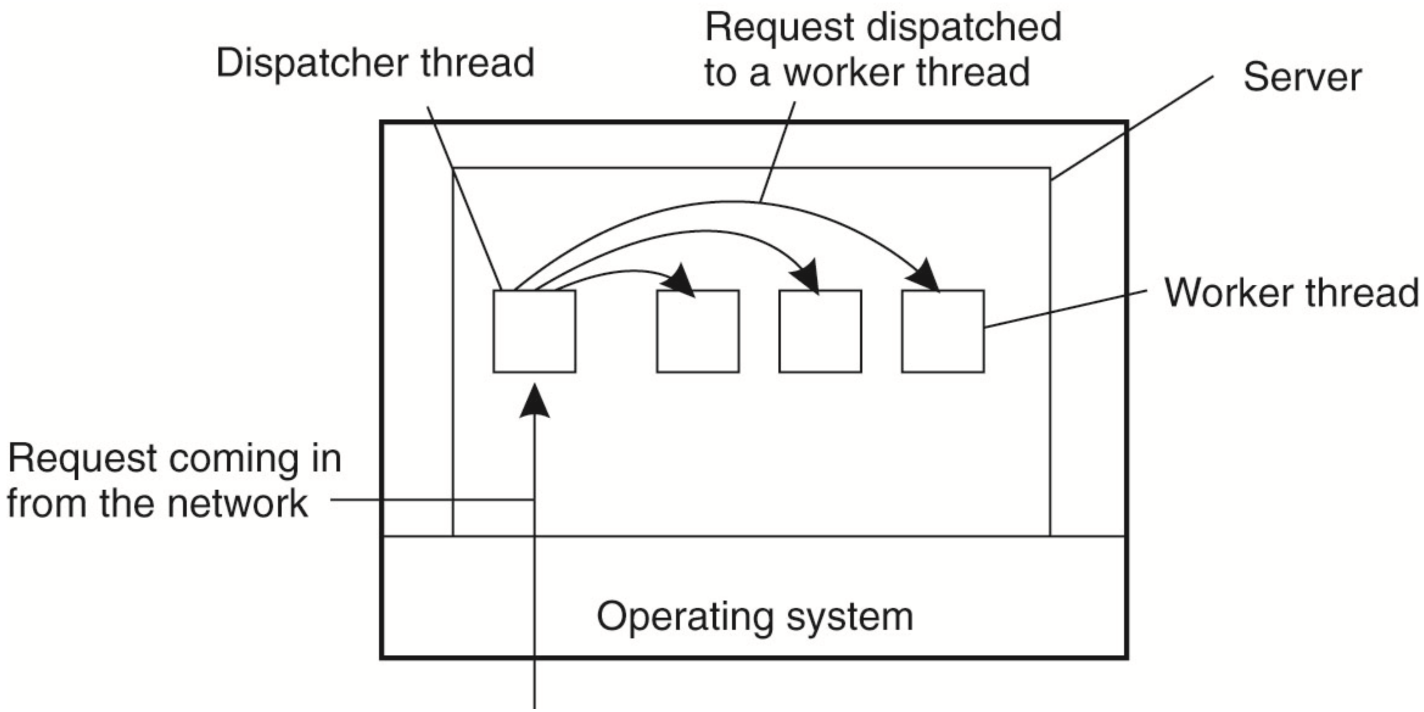
Multithreaded Client

- Web Browser: requesting a page with text, icons, image etc… multiple files
- Distribution transperency: hide comm delays
  - Set up connection
  - Read incoming data
  - Pass it to display component
- Some browser start displaying data as they come in while user dealing with the main (partial) page.

Multithreaded Server: Birçok clienta birden hizmet sunabilme



Multithreaded Servers



A multithreaded server organized in a dispatcher/worker model.

-- Requestleri işlemek için bir çalışan thread havuzu

-- Bir **dağıtıcı thread**: requestleri clientlardan alır ve çalışanlar tarafından alınmak üzere paylaşılan request kuyruğuna ekler

Thread Usage: File server

A single thread file server:

– the only thread gets a request, examines it and carries it out to completion before getting the next request. The CPU is simply idle while the file server is waiting for the disk.

Multiple-thread file server:

– Here one thread, the dispatcher, reads incoming work requests from the system mailbox. After examining the request, it choose an idle worker thread and hands it the request. When the worker wakes up, it checks to see if the request can be satisfied from the shared block cache, to which all threads have access. If not, it sends a message to the disk to get the needed block and goes to sleep awaiting completion of the disk operation. The dispatcher in the meantime can serve other requests.

Finite state machine:

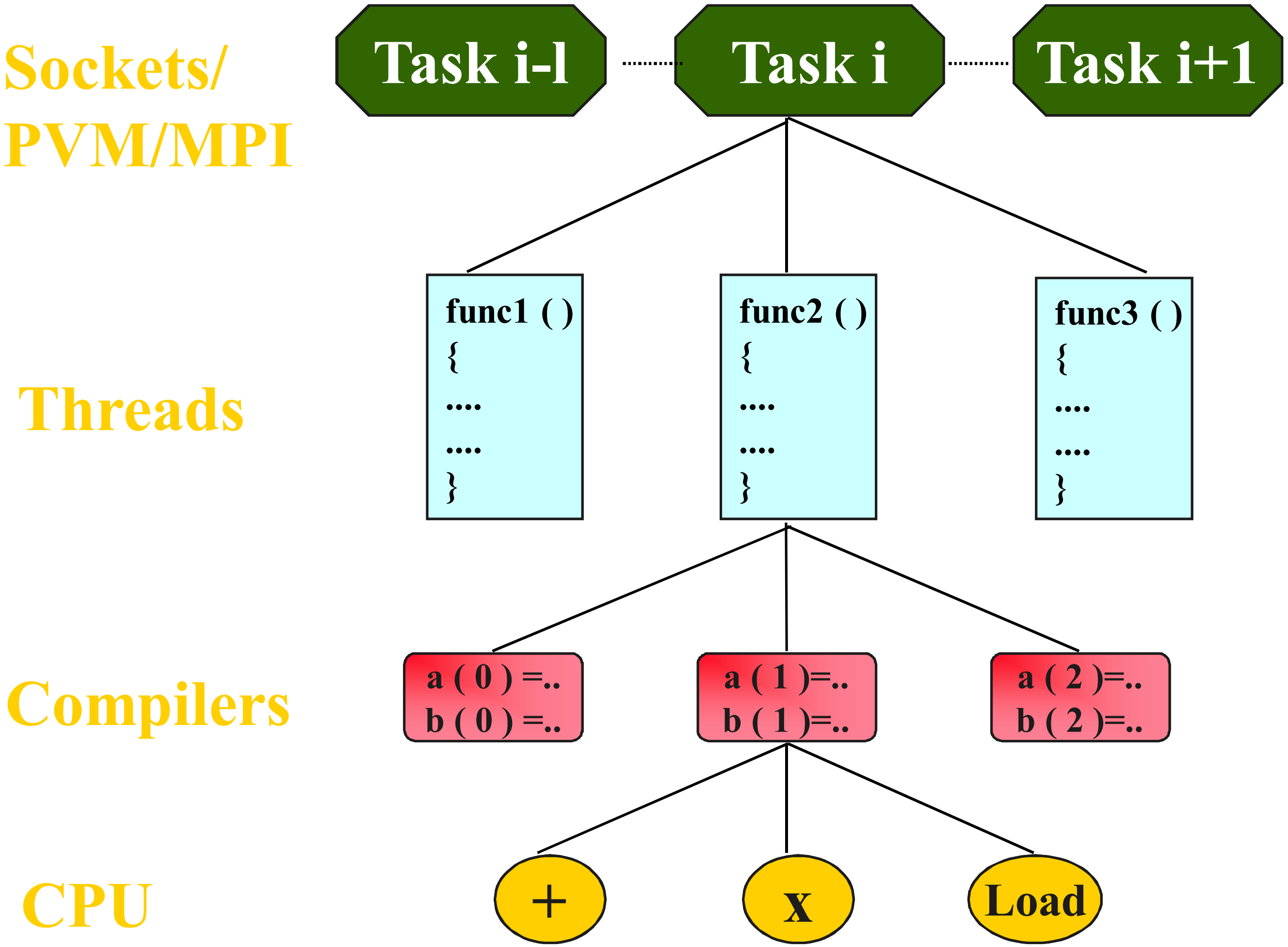
– When request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a message must be sent to the disk. However, instead of blocking, it records the state of the current request in a table and then goes and gets the next message. The next message may either be a request for new work or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed.

Thread Usage (continue)

- Threads make it possible to retain the idea of sequential processes that make blocking systems calls and still achieve parallelism.
- Blocking system calls make programming easier and parallelism improve performance.
- The single threaded server retains the ease of blocking system calls, but give up performance.
- The finite state machine approach achieves high performance through parallelism, but use non-blocking calls and thus is hard to program.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-thread process	No parallelism, blocking system calls
Finite-state-machine	Parallelism, non-blocking system calls

Levels of Parallelism
Paralellik Düzeyleri



Code-Granularity

Code Item

Large grain (task level)

Program

Medium grain (control level)

Function (thread)

Fine grain (data level)

Loop (Compiler)

Very fine grain (multiple issue)

With hardware

Java Threadleri

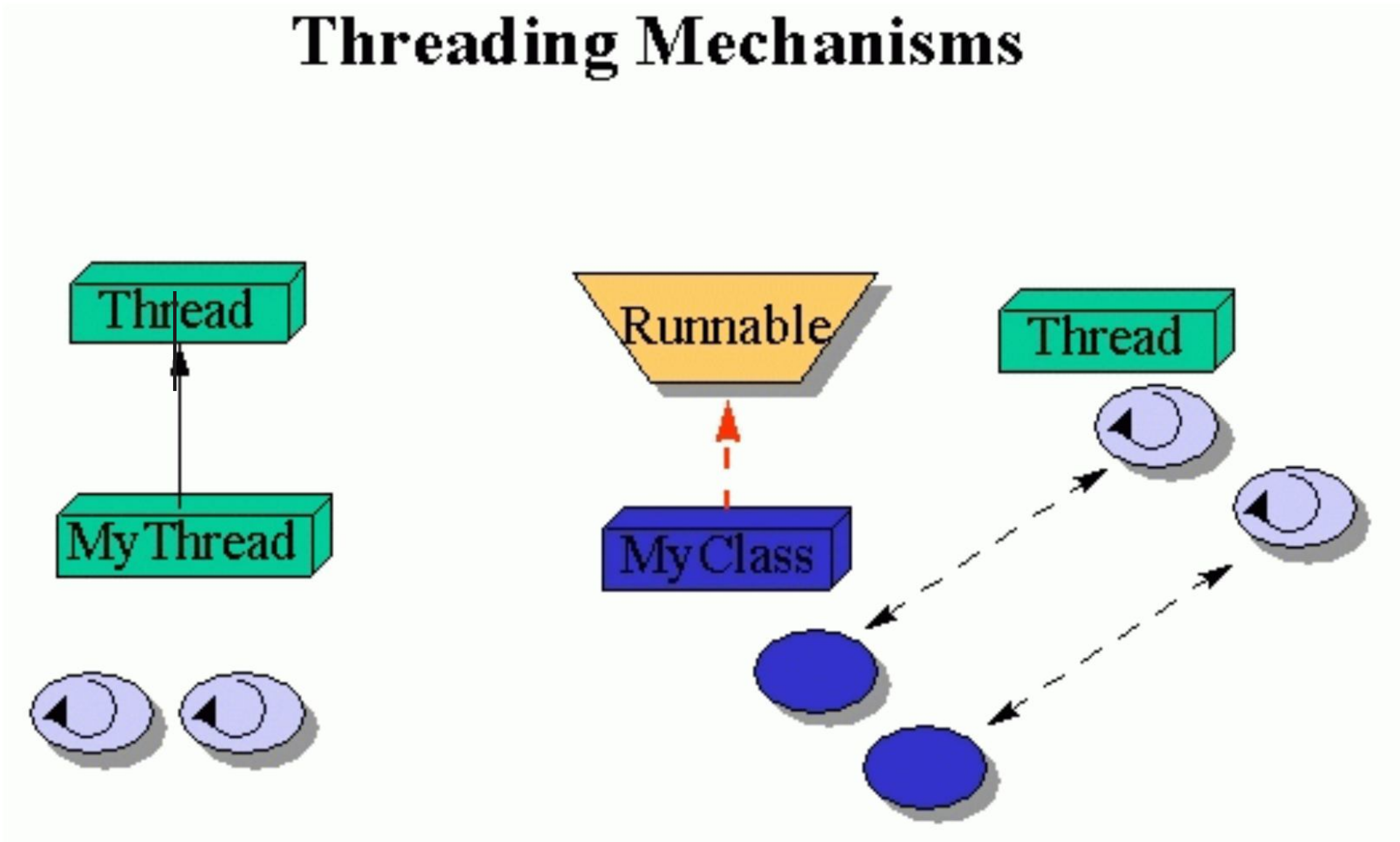
- Birçok threadi bulunan bir görevi/uygulamayı programlama – Multithreading veya Multithreaded Programming (concurrent programming, synchronization)
- Thread Scheduling - Thread Zamanlama
- Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority

  Threadler arası işlemler: thread mgmt
  - currentThread start setPriority
  - yield run getPriority
  - sleep(ms) stop suspend
  - resume Thread(grp, target, name)
- Java Garbage Collector is a low-priority thread.

  Java Çöp Toplayıcısı, düşük öncelikli bir threaddir.

Threading Mechanisms...
Thread Oluşturma Mekanizmaları

1. Thread class'ını extend eden bir class oluştur
2. Runnable interface'ini implement eden bir class oluştur



1st method: Extending Thread class
1. metot: Thread classının extend edilmesi

- Threadler run() isimli bir metot bulunduran alt nesneler olarak implement edilirler

```
class MyThread extends Thread
{
    public void run()
    {
        // thread body of execution
    }
}
```

- Bir thread oluştur:

```
MyThread thr1 = new MyThread();
```

- Threadlerin çalışmasını başlat:

```
thr1.start();
```

- Oluştur ve Çalıştır:

```
new MyThread().start();
```

Bir örnek

```
class MyThread extends Thread {          // the thread
       public void run() {
              System.out.println(" this thread is running ... ");
       }
} // end class MyThread

class ThreadEx1 {                 // a program that utilizes the thread
       public static void main(String [] args  ) {
             MyThread t = new MyThread();
             MyThread t2 = new MyThread();
             // due to extending the Thread class (above)
             // I can call start(), and this will call
             // run(). start() is a method in class Thread.
             t2.start();
            t.start();
      } // end main()
}       // end class ThreadEx1
```

2nd method: Threads by implementing Runnable interface
2. metot: Runnable interface'i implement edilerek

```
class MyThread extends ABC implements Runnable
{
  .....
  public void run()
  {
     // thread body of execution
  }
}
```

- Nesnenin Oluşturulması:

```
MyThread myObject = new MyThread();
```

- Thread Nesnesinin Oluşturulması:

```
Thread thr1 = new Thread( myObject );
```

- Çalışmanın Başlatılması:

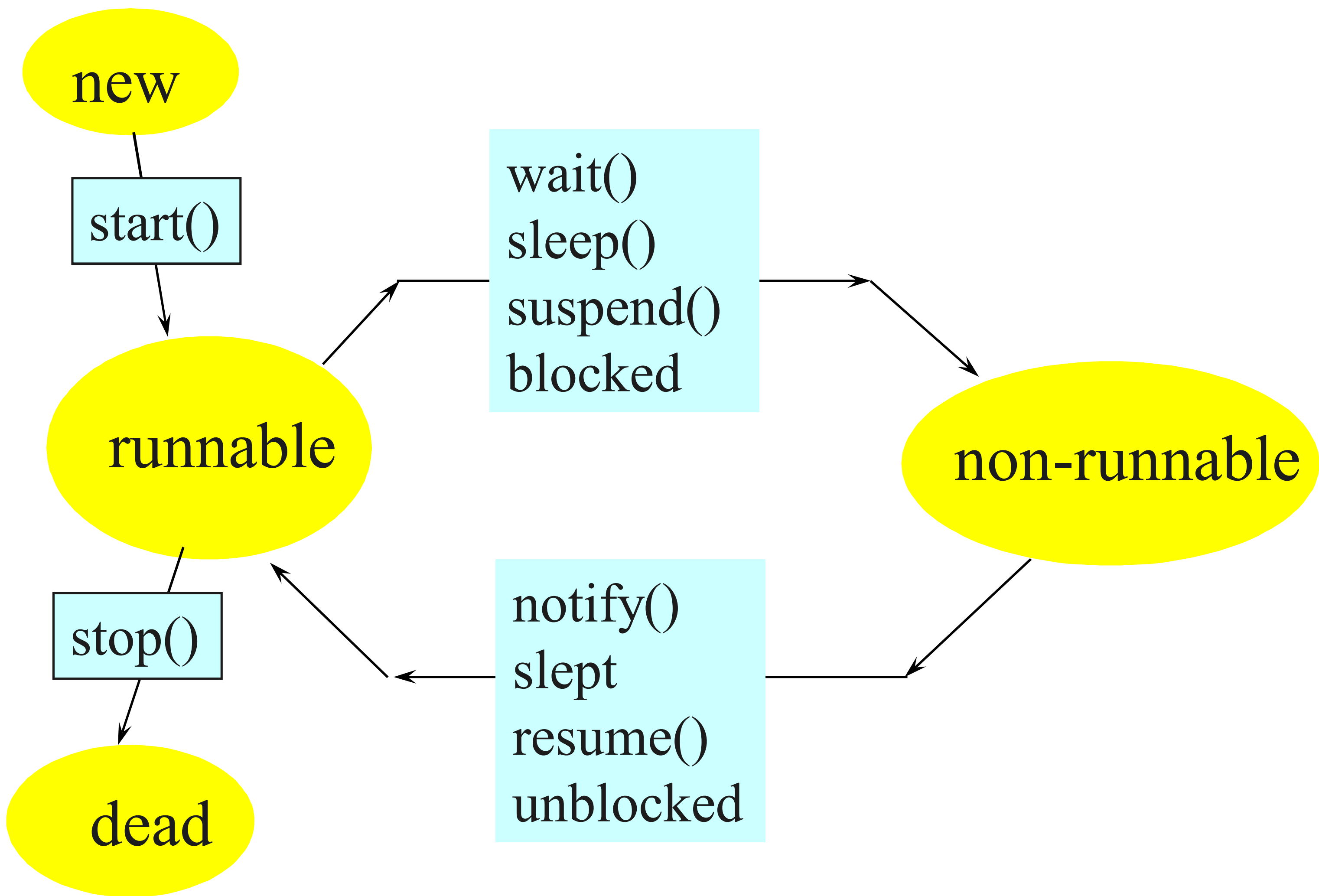
```
thr1.start();
```

Bir örnek

```
class MyThread implements Runnable  {
        public void run() {
                System.out.println(" this thread is running ... ");
        }
} // end class MyThread

class ThreadEx2 {
        public static void main(String [] args  ) {
                Thread t = new Thread(new MyThread());
                            // due to implementing the Runnable interface
                           // I can call start(), and this will call run().
                t.start();
        } // end main()
}       // end class ThreadEx2
```

Thread'in Hayat Döngüsü



Üç Java Thread'li bir Program

- 3 thread oluşturan bir program yazın

Üç thread örneği

```
class A extends Thread
{
     public void run()
      {
          for(int i=1;i<=5;i++)
           {
               System.out.println("\t From ThreadA: i= "+i);
           }
            System.out.println("Exit from A");
      }
}

class B extends Thread
{
     public void run()
      {
          for(int j=1;j<=5;j++)
           {
               System.out.println("\t From ThreadB: j= "+j);
           }
            System.out.println("Exit from B");
      }
}
```

Üç thread örneği

```
class C extends Thread
{
      public void run()
       {
         for(int k=1;k<=5;k++)
          {
            System.out.println("\t From ThreadC: k= "+k);
          }

          System.out.println("Exit from C");
       }
}

class ThreadTest
{
        public static void main(String args[])
         {
            new A().start();
            new B().start();
            new C().start();
         }
}
```

Run 1 (Çalışma 1)

- [user@speedy] threads [1:76] java ThreadTest

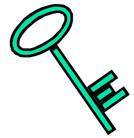
```
    From ThreadA: i= 1
    From ThreadA: i= 2
    From ThreadA: i= 3
    From ThreadA: i= 4
    From ThreadA: i= 5
Exit from A
    From ThreadC: k= 1
    From ThreadC: k= 2
    From ThreadC: k= 3
    From ThreadC: k= 4
    From ThreadC: k= 5
Exit from C
    From ThreadB: j= 1
    From ThreadB: j= 2
    From ThreadB: j= 3
    From ThreadB: j= 4
    From ThreadB: j= 5
Exit from B
```

Run2 (Çalışma2)

- [user@speedy] threads [1:77] java ThreadTest

```
    From ThreadA: i= 1
    From ThreadA: i= 2
    From ThreadA: i= 3
    From ThreadA: i= 4
    From ThreadA: i= 5
    From ThreadC: k= 1
    From ThreadC: k= 2
    From ThreadC: k= 3
    From ThreadC: k= 4
    From ThreadC: k= 5
Exit from C
    From ThreadB: j= 1
    From ThreadB: j= 2
    From ThreadB: j= 3
    From ThreadB: j= 4
    From ThreadB: j= 5
Exit from B
Exit from A
```

Shared Resources
Paylaşılan Kaynaklar



- Bir thread veriyi okurken başka bir threadin aynı veriyi değiştirmeye çalışması, tutarsızlığa yol açar.
- Bu durum veri erişiminin senkronize edilmesiyle engellenebilir.
- “Synchronized” metot:
  - public synchronized void update()
  - {
    - …
  - }

the driver:
3 Threadin aynı nesneyi paylaşmaları

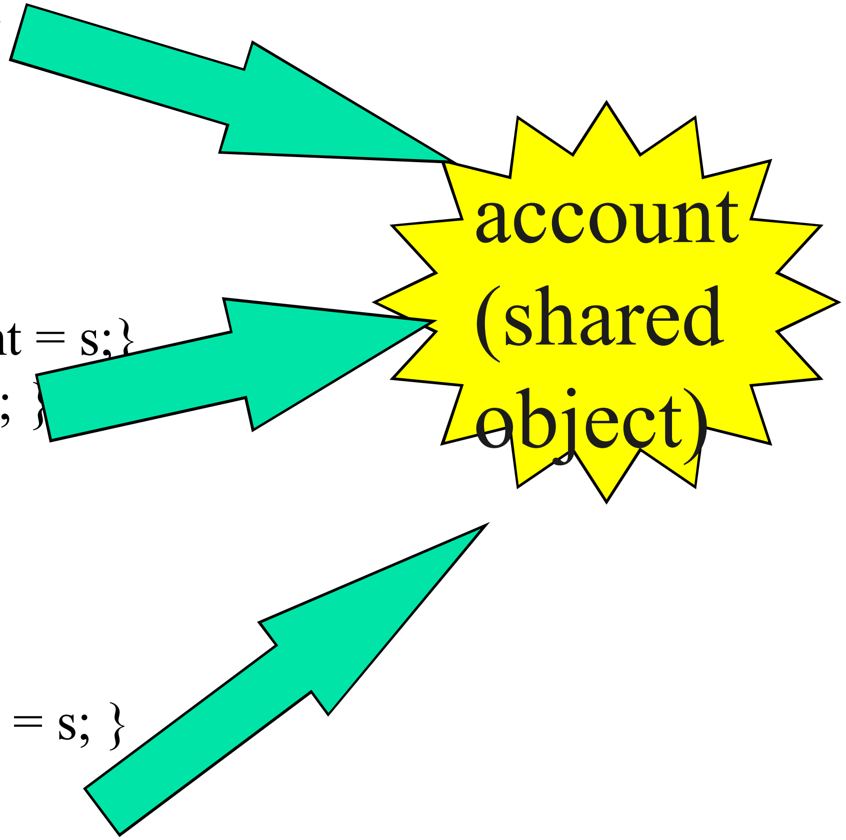
```
class InternetBankingSystem {
    public static void main(String [] args  ) {
        Account accountObject = new Account ();
        Thread t1 = new Thread(new MyThread(accountObject));
        Thread t2 = new Thread(new YourThread(accountObject));
        Thread t3 = new Thread(new HerThread(accountObject));
        t1.start();
        t2.start();
        t3.start();
       // DO some other operation
    } // end main()
}
```

3 thread arasında paylaşılan account nesnesi

```
class MyThread implements Runnable  {
 Account account;
        public MyThread (Account s) {  account = s;}
        public void run() { account.deposit(); }
} // end class MyThread

class YourThread implements Runnable  {
 Account account;
        public YourThread (Account s) { account = s;}
        public void run() { account.withdraw(); }
} // end class YourThread

class HerThread implements Runnable  {
 Account account;
        public HerThread (Account s) { account = s; }
        public void run() {account.enquire(); }
} // end class HerThread
```



Monitor (paylaşılan nesne erişimi): paylaşılan nesne işlemlerini sıralar

```
class Account {   // the 'monitor'
  int balance;

      // if 'synchronized' is removed, the outcome is unpredictable
       public synchronized void deposit( ) {
         // METHOD BODY : balance += deposit_amount;
        }

         public synchronized void withdraw( ) {
          // METHOD BODY: balance -= deposit_amount;
         }
         public synchronized void enquire( ) {
          // METHOD BODY: display balance.
         }
}
```