# Multithreading

The objectives of this chapter are:

- To understand the purpose of multithreading
- To describe Java's multithreading mechanism
- To explain concurrency issues caused by multithreading
- To outline synchronized access to shared resources

## What is Multithreading?

- Multithreading is similar to multi-processing.
- A multi-processing Operating System can run several processes at the same time
  - Each process has its own address/memory space
  - The OS's scheduler decides when each process is executed
    - Only one process is actually executing at any given time.  However, the system appears to be running several programs simultaneously
- Separate processes to not have access to each other's memory space
  - Many OSes have a shared memory system so that processes can share memory space
- In a multithreaded application, there are several points of execution within the same memory space.
  - Each point of execution is called a thread
  - Threads share access to memory

## What is Multithreading?

| Multithreading | Multitasking |
|---|---|
| Programming concept | Operating system concept |
| Supports execution of multiple parts of a single program simultaneously | Supports execution of multiple programs simultaneously |
| Processor has to switch between different parts or threads of a program. | Processor has to switch between different programs. |
| Highly efficient. | Less efficient |
| A thread is the smallest unit in multithreading | A program/process is the smallest unit in multitasking |
| Helps in developing efficient programs. | Helps in developing efficient OS. |
| Cost effective in case of context switching | Costly in case of context switching |

## Why use Multithreading?

- In a single threaded application, one thread of execution must do everything
  - If an application has several tasks to perform, those tasks will be performed when the thread can get to them.
  - A single task which requires a lot of processing can make the entire application appear to be "sluggish" or unresponsive.
- In a multithreaded application, each task can be performed by a separate thread
  - If one thread is executing a long process, it does not make the entire application wait for it to finish.
- If a multithreaded application is being executed on a system that has multiple processors, the OS may execute separate threads simultaneously on separate processors.

## What Kind of Applications Use Multithreading?

- Any kind of application which has distinct tasks which can be performed independently
  - Any application with a GUI.
    - Threads dedicated to the GUI can delegate the processing of user requests to other threads.
    - The GUI remains responsive to the user even when the user's requests are being processed
  - Any application which requires asynchronous response
    - Network based applications are ideally suited to multithreading.
      - Data can arrive from the network at any time.
      - In a single threaded system, data is queued until the thread can read the data
      - In a multithreaded system, a thread can be dedicated to listening for data on the network port
      - When data arrives, the thread reads it immediately and processes it or delegates its processing to another thread

## How does it all work?

- Each thread is given its own "context"
  - A thread's context includes virtual registers and its own calling stack
- The "scheduler" decides which thread executes at any given time
  - The VM may use its own scheduler
  - Since many OSes now directly support multithreading, the VM may use the system's scheduler for scheduling threads
- The scheduler maintains a list of ready threads (the run queue) and a list of threads waiting for input (the wait queue)
- Each thread has a priority. The scheduler typically schedules between the highest priority threads in the run queue
  - Note: the programmer cannot make assumptions about how threads are going to be scheduled. Typically, threads will be executed differently on different platforms.

## Thread Support in Java

- Few programming languages directly support threading
  - Although many have add-on thread support
  - Add on thread support is often quite cumbersome to use
- The Java Virtual machine has its own runtime threads
  - Used for garbage collection
- Threads are represented by a Thread class
  - A thread object maintains the state of the thread
  - It provides control methods such as interrupt, start, sleep, yield, wait
- When an application executes, the main method is executed by a single thread.
  - If the application requires more threads, the application must create them.

B612

A unique property of Java is its
pport for multithreading. That is,
a enables us to use multiple flows
ontrol in developing programs.
flow of control may be thought
s a separate tiny program (or
e) known as a *thread* that runs in
to others as shown in
2. A program that contains
flows of control is known as
*ded program*. Figure 12.2
a Java program with four
e main and three others.
read is actually the **main**
le, which is designed to
t the other three threads,
d C.

ed by the main thread,
and C run concurrently
esources jointly. It is
in joint families and
esources among all
ity of a language to
ds is referred to as
e threads

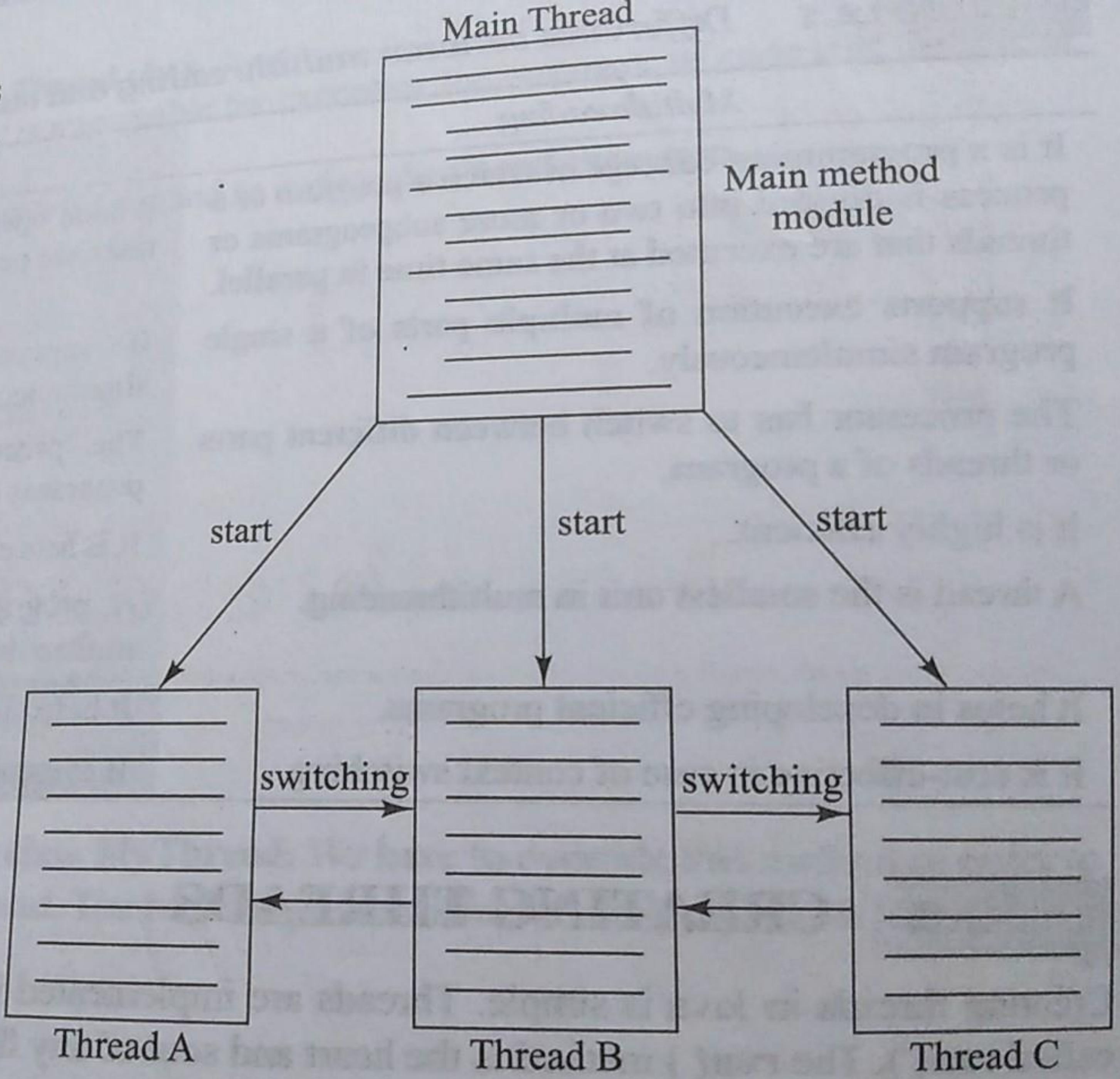
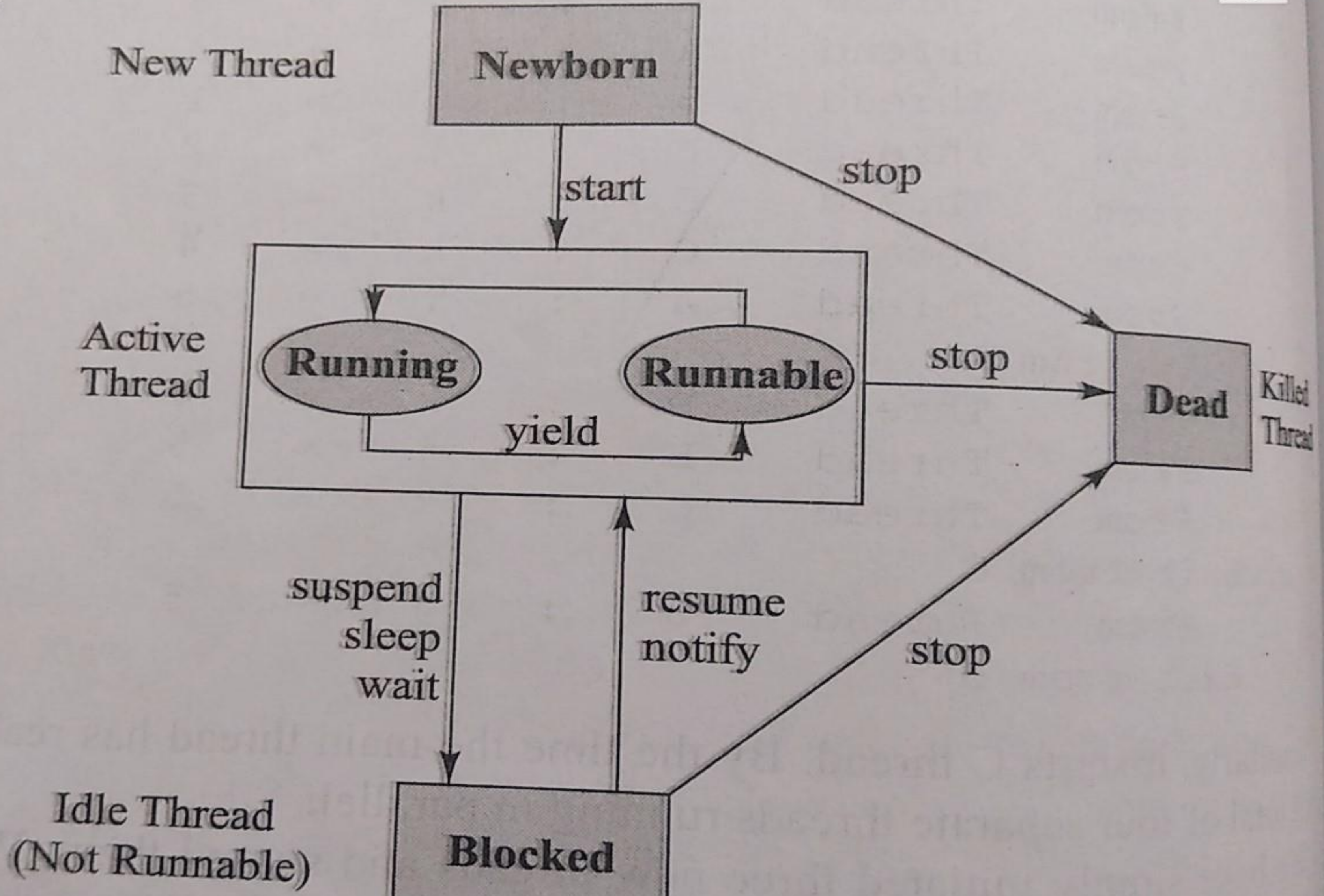


**Fig. 12.2** *A Multithreaded program*

# Thread life cycle

## E CYCLE OF A THREAD

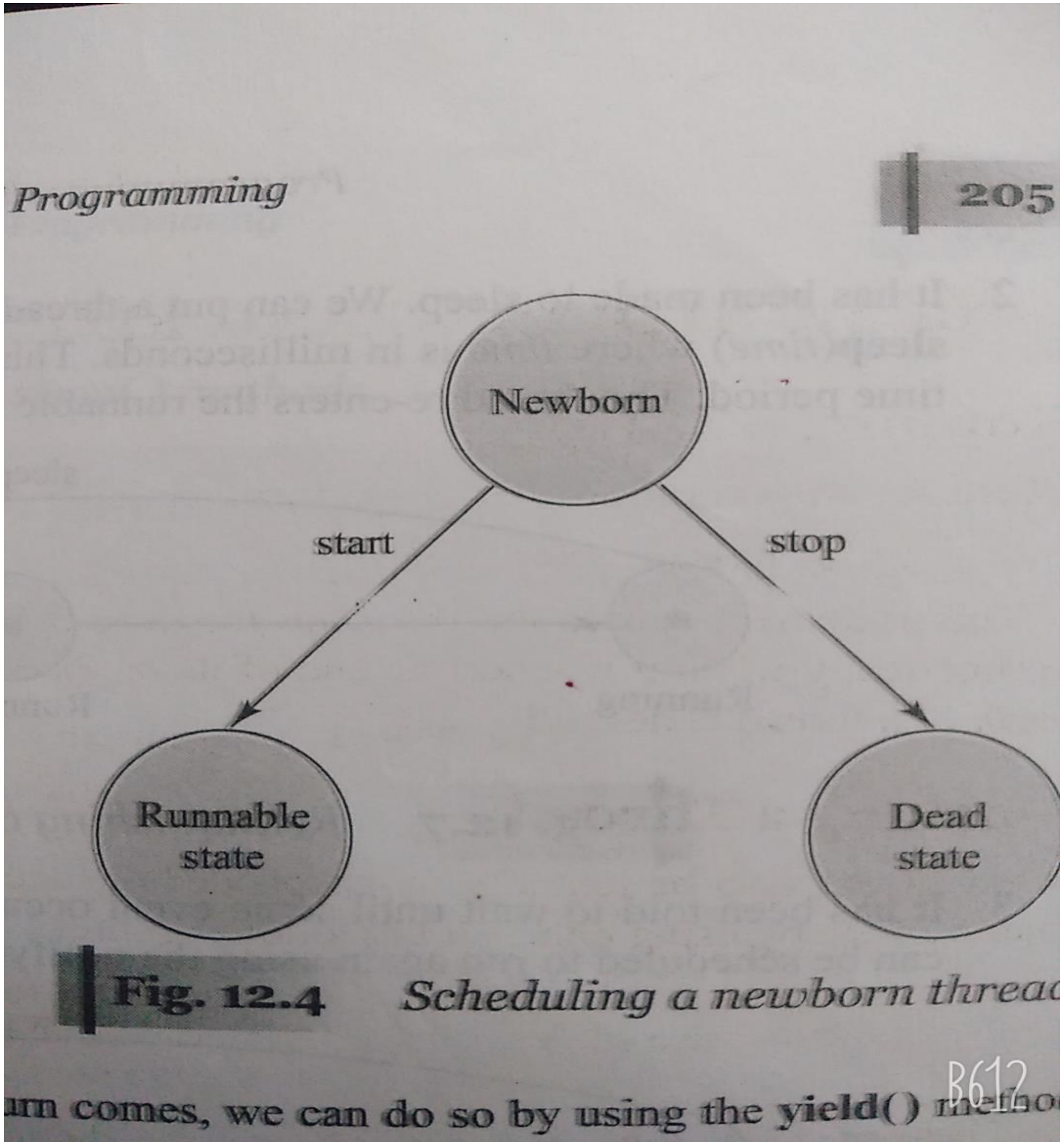
f a thread, there
an enter. They



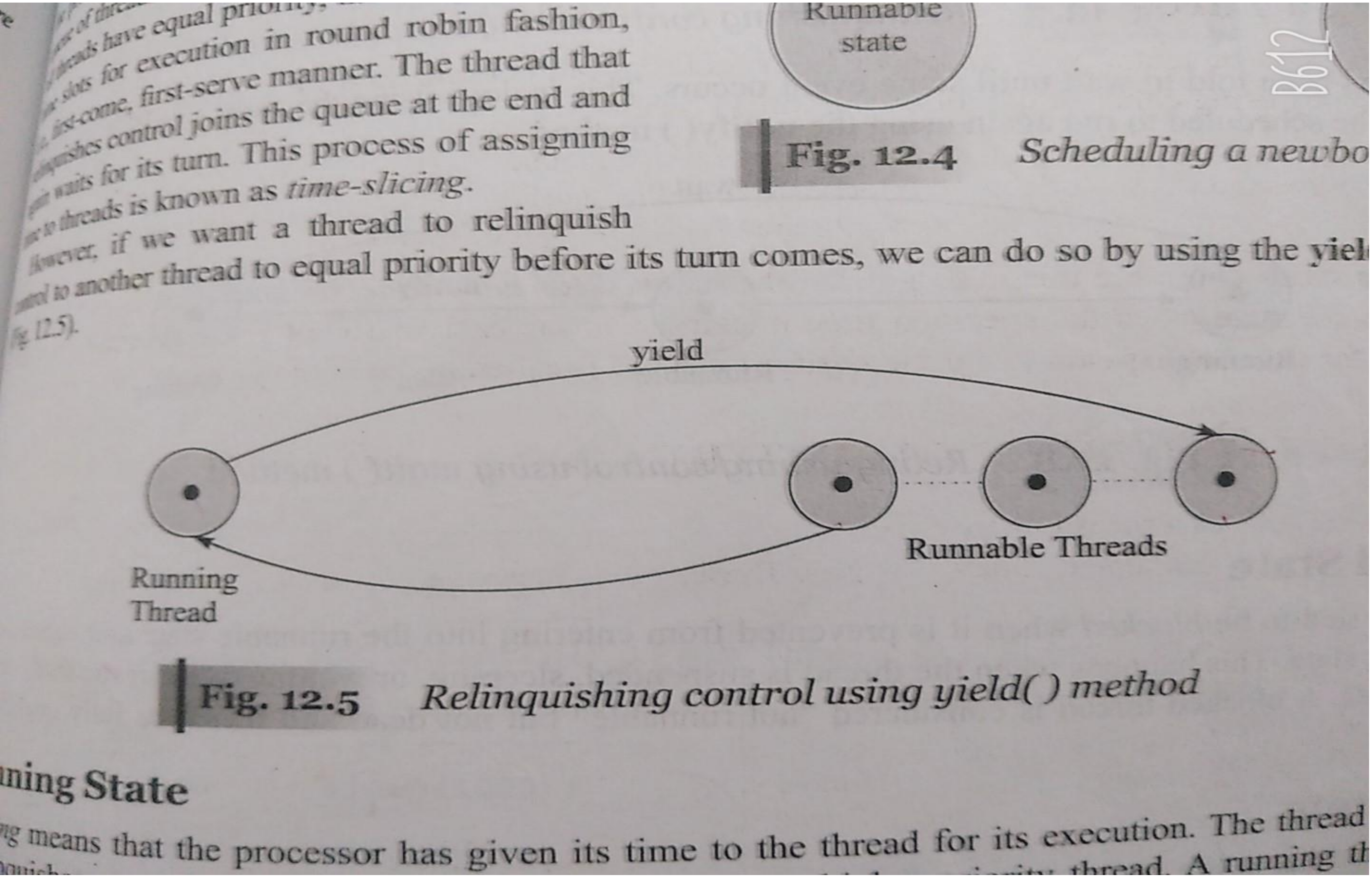
n one of these
ve from one
riety of ways

d object, the
aid to be in
ad is not yet
At this state,

Thread life cycle


Programming
205
Newborn
start
stop
Runnable state
Dead state
Fig. 12.4 Scheduling a newborn threa
urn comes, we can do so by using the yield() metho
B612

## Thread life cycle



...ads have equal priority, ...
...slots for execution in round robin fashion,
first-come, first-serve manner. The thread that
...shes control joins the queue at the end and
...waits for its turn. This process of assigning
...to threads is known as *time-slicing*.
However, if we want a thread to relinquish
control to another thread to equal priority before its turn comes, we can do so by using the yiel
Fig. 12.5).

Runnable state

**Fig. 12.4** *Scheduling a newbo*

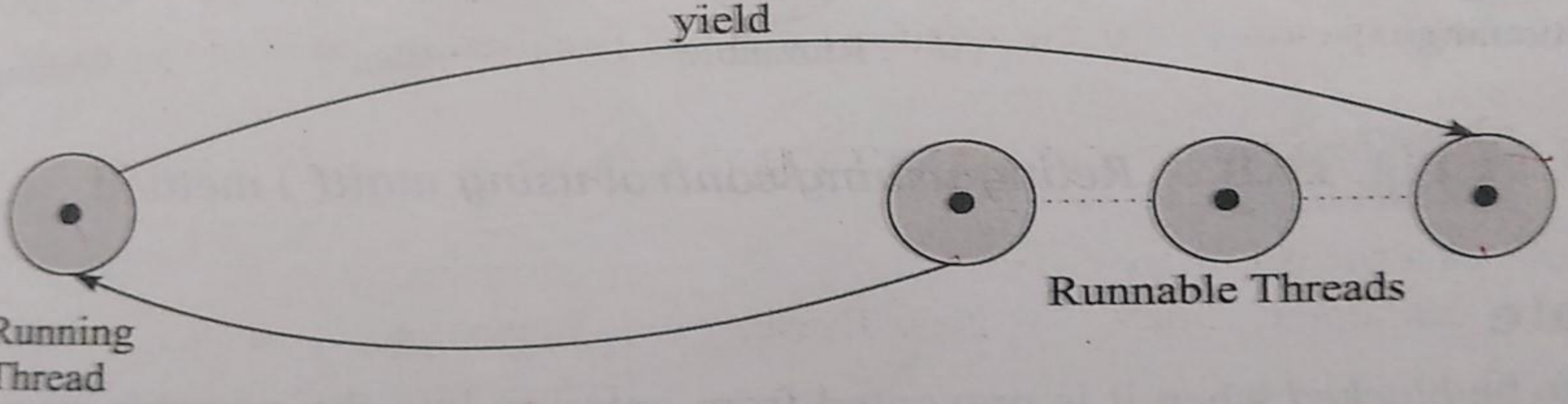
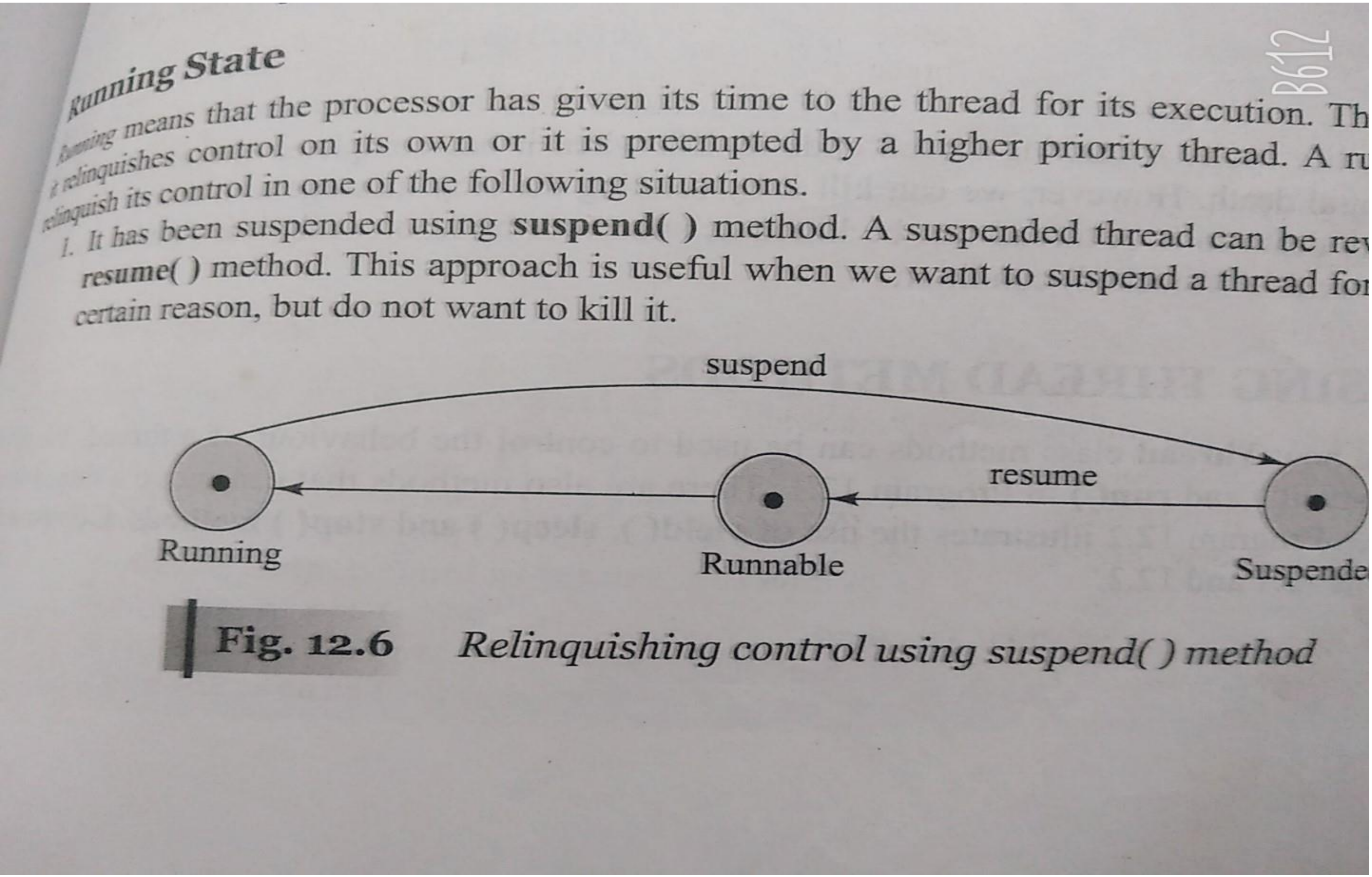


**Fig. 12.5** *Relinquishing control using yield( ) method*

**ning State**

...g means that the processor has given its time to the thread for its execution. The thread
...quish... ...ority thread. A running th

# Thread life cycle



**Running State**

Running means that the processor has given its time to the thread for its execution. Th
it relinquishes control on its own or it is preempted by a higher priority thread. A ru
relinquish its control in one of the following situations.

1. It has been suspended using **suspend( )** method. A suspended thread can be re
**resume( )** method. This approach is useful when we want to suspend a thread fo
certain reason, but do not want to kill it.

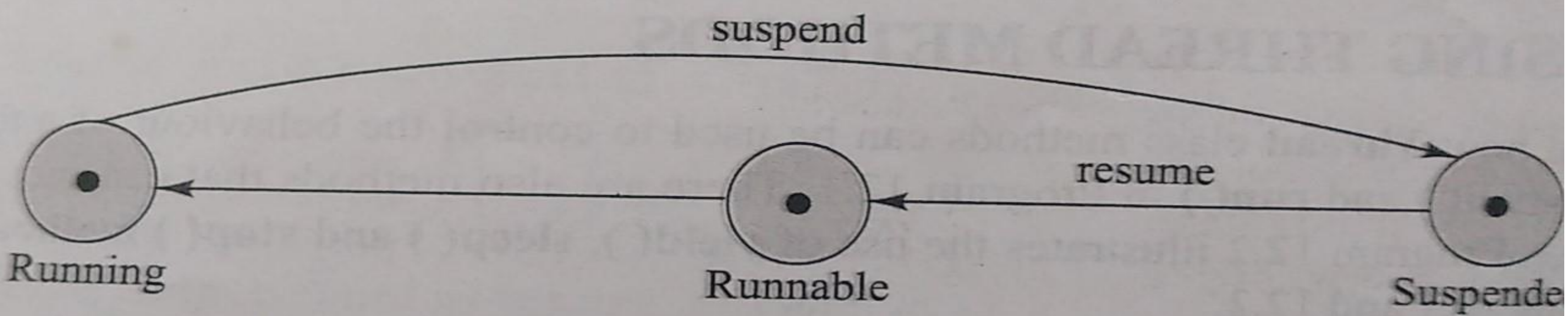


**Fig. 12.6** *Relinquishing control using suspend( ) method*

# Thread life cycle

2. It has been made to sleep. We can put a thread to sleep for a specified time period using the **sleep**(*time*) where *time* is in milliseconds. This means that the thread is out of the queue dur time period. The thread re-enters the runnable state as soon as this time period is elapsed

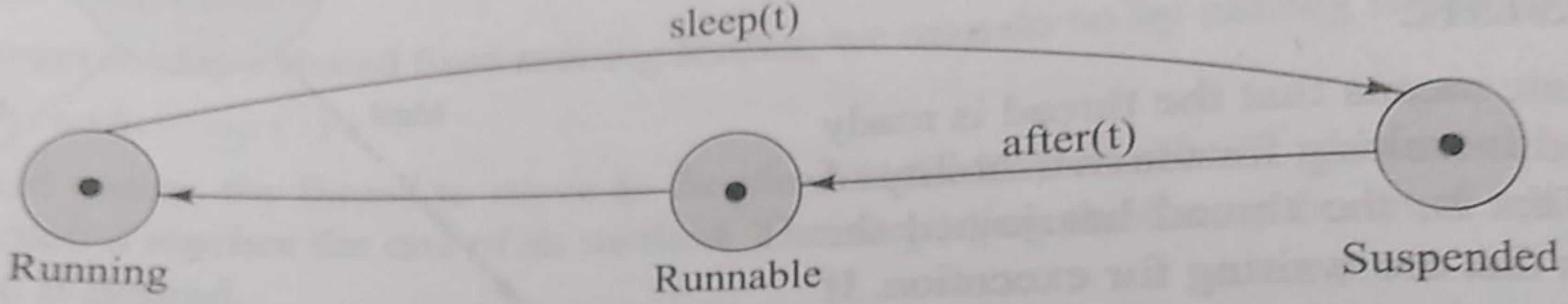


**Fig. 12.7** *Relinquishing control using sleep( ) method*

It has been told to wait until some event occurs. This is done using the **wait( )** method. Th can be scheduled to run again using the **notify( )** method.

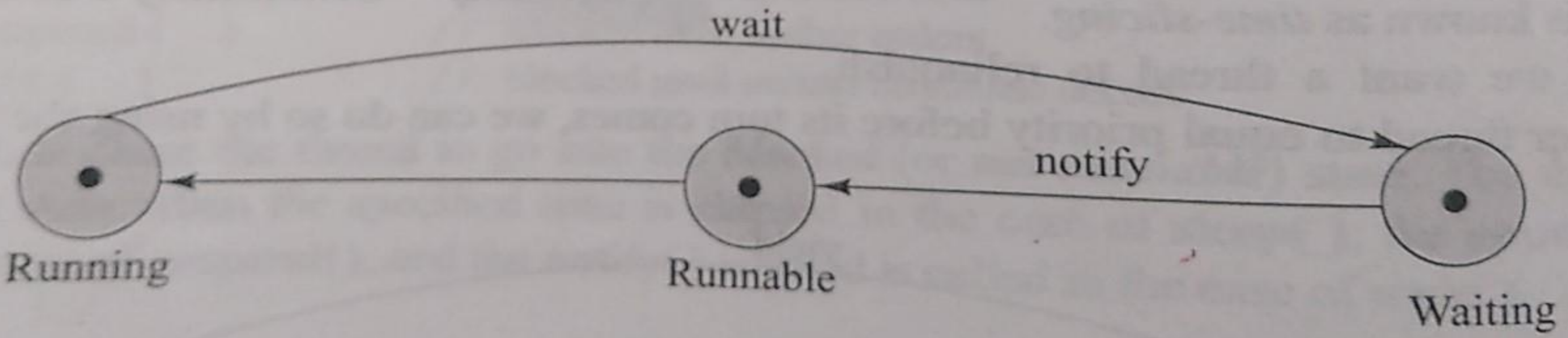


**Fig. 12.8** *Relinquishing control using wait( ) method*

## Creating your own Threads

- There are two ways

1. The obvious way to create your own threads is to subclass the Thread class and then override the run() method
   - This is the easiest way to do it
   - It is not the recommended way to do it.

- Because threads are usually associated with a task, the object which provides the run method is usually a subclass of some other class
  - If it inherits from another class, it cannot inherit from Thread.

2. The solution is provided by an interface called Runnable.
   - Runnable defines one method - public void run()

- One of the Thread classes constructor takes a reference to a Runnable object
  - When the thread is started, it invokes the run method in the runnable object instead of its own run method.

## Creating your own Threads

**By extending the Thread class: 3 steps**

1. Declare the class as extending **Thread** class:

```
 class MyThread extends Thread
{
 …..
 }
```

2.Implementing the run() method:

```
 public void run()
{
   ………..//Thread code here
   ………..
}
```

3.Starting New Thread:

```
  MyThread  a= new MyThread();
   a.start();            //invokes run() method
```

## Example Code: By extending Thread class

```
class Multi extends Thread
{
public void run()
{
System.out.println("thread is running...");
}
}
Class abc
{
public static void main(String args[])
{
Multi t1=new Multi();
t1.start();
 }
}
```

## By extending Thread class

Write a program to create two threads Thread A and Thread B. Thread A displays the numbers from 1 to 7 and thread B displays days in a week. Create main method to call these two threads and display the output.

## Creating your own Threads

**By implementing the “Runnable”interface: 4 steps**

1. Declare the class by implementing **Runnable** interface:

```
 class MyThread implements Runnable
{
 …..
 }
```

2.Implementing the run() method:

```
 public void run()
{
   ………..//Thread code here
   ………..
}
```

3.Create a thread by defining an object that is instantiated from this “runnable” class as the target of the thread.

```
MyThread  a= new MyThread();
Thread x=new Thread(a);
```

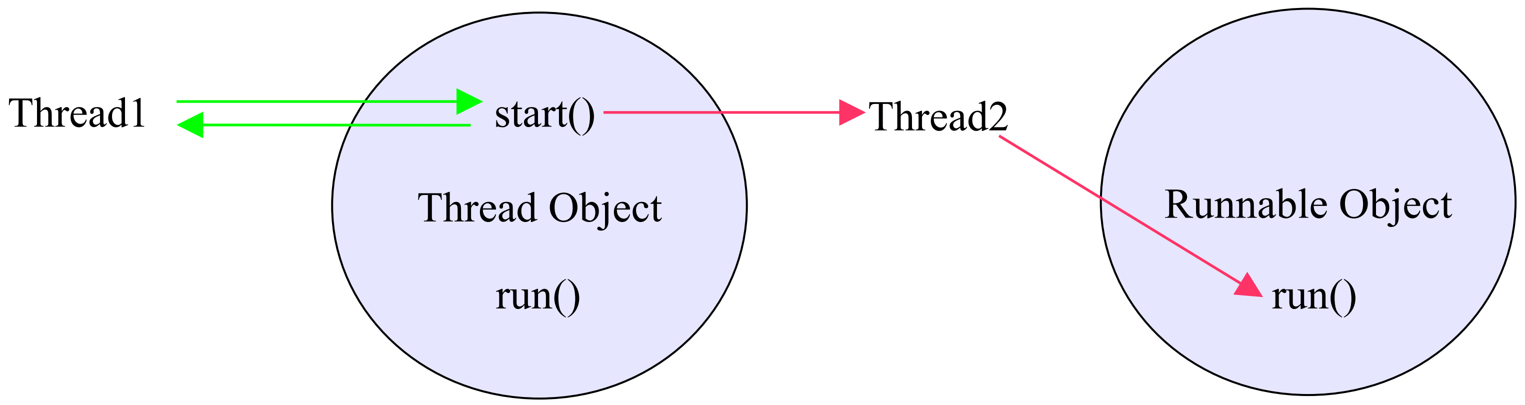
4. Call the thread’s start() method to run the thread.

```
 x.start();            //invokes run() method
```

## Using Runnable

- In the example below, when the Thread object is instantiated, it is passed a reference to a "Runnable" object
  - The Runnable object must implement a method called "run"

- When the thread object receives a start message, it checks to see if it has a reference to a Runnable object:
  - If it does, it runs the "run" method of that object
  - If not, it runs its own "run" method



## Example using Runnable interface

```
class Multi3 implements Runnable
{
public void run()
{
System.out.println("thread is running...");
}

public static void main(String args[])
{
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

# Creating Multiple Threads

- The previous example illustrates a Runnable class which creates its own thread when the start method is invoked.
- If one wished to create multiple threads, one could simple create multiple instances of the Runnable class and send each object a start message
  - Each instance would create its own thread object
- Is the a maximum number of threads which can be created?
  - There is no defined maximum in Java.
  - If the VM is delegating threads to the OS, then this is platform dependent.
  - A good rule of thumb for maximum thread count is to allow 2Mb of ram for each thread
    - Although threads share the same memory space, this can be a reasonable estimate of how many threads your machine can handle.

**public void run():** is used to perform action for a thread.
**public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
**public void sleep(long milliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
**public void join():** waits for a thread to die.
**public void join(long milliseconds):** waits for a thread to die for the specified miliseconds.
**public int getPriority():** returns the priority of the thread.
**public int setPriority(int priority):** changes the priority of the thread.
**public String getName():** returns the name of the thread.
**public void setName(String name):** changes the name of the thread.
**public Thread currentThread():** returns the reference of currently executing thread.

Thread(Runnable threadobj,String threadName)

**public int getId():** returns the id of the thread.
**public Thread.State getState():** returns the state of the thread.
**public boolean isAlive():** tests if the thread is alive.
**public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
**public void suspend():** is used to suspend the thread(depricated).
**public void resume():** is used to resume the suspended thread(depricated).
**public void stop():** is used to stop the thread(depricated).
**public void interrupt():** interrupts the thread.
**public boolean isInterrupted():** tests if the thread has been interrupted.
**public static boolean interrupted():** tests if the current thread has been interrupted.

## Thread Priorities

- Every thread is assigned a priority (between 1 and 10)
  - The default is 5
  - The higher the number, the higher the priority
  - Can be set with setPriority(int aPriority)
- The standard mode of operation is that the scheduler executes threads with higher priorities first.
  - This simple scheduling algorithm can cause problems.  Specifically, one high priority thread can become a "CPU hog".
  - A thread using vast amounts of CPU can share CPU time with other threads by invoking the yield() method on itself.
- Most OSes do not employ a scheduling algorithm as simple as this one
  - Most modern OSes have thread aging
    - The more CPU a thread receives, the lower its priority becomes
    - The more a thread waits for the CPU, the higher its priority becomes
  - Because of thread aging, the effect of setting a thread's priority is dependent on the platform

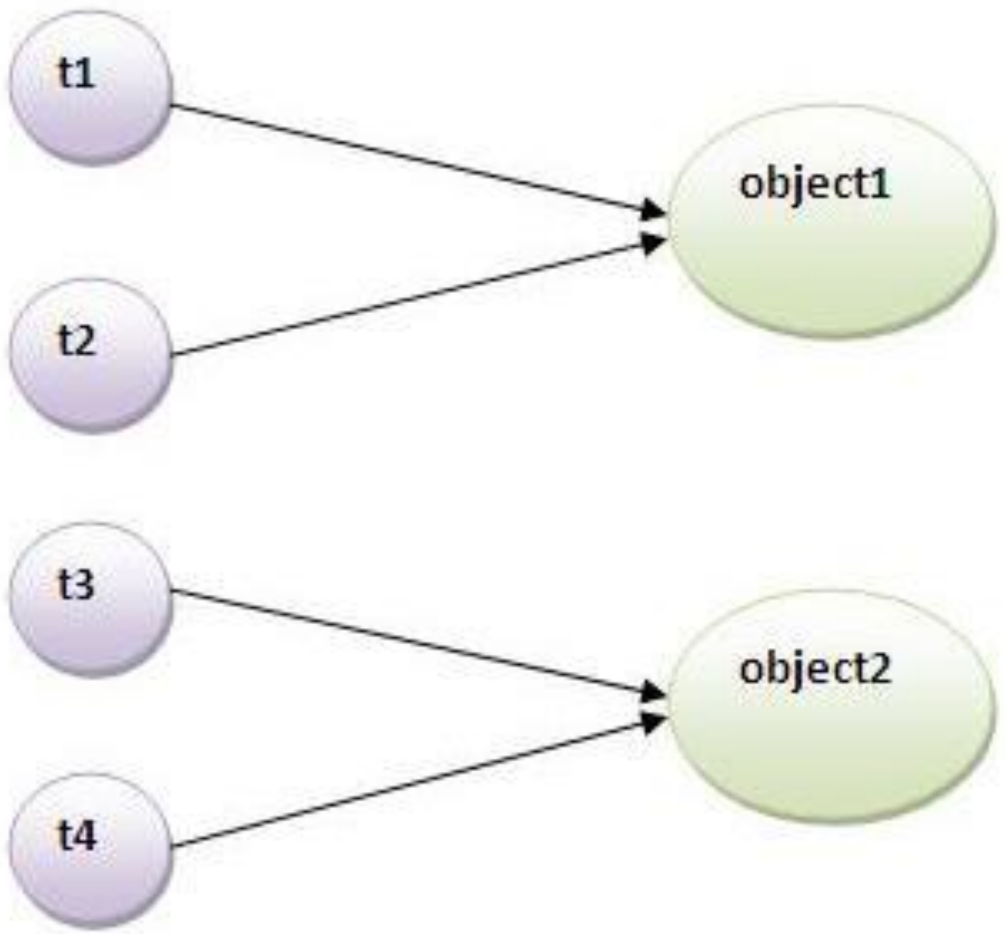
## Thread Program

- Write a program in Java to create two threads thread A and thread B. Thread A stores an array of characters whereas Thread B stores an array of numbers. Make them run concurrently and display  characters and numbers.

## Synchronized block in java

- Synchronized block can be used to perform synchronization on any specific resource of the method.

```
synchronized (object reference expression) {
  //code block
}
```


t1
t2
object1
t3
t4
object2

## Thread Synchronization:

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1.Mutual Exclusive

   Synchronized method.

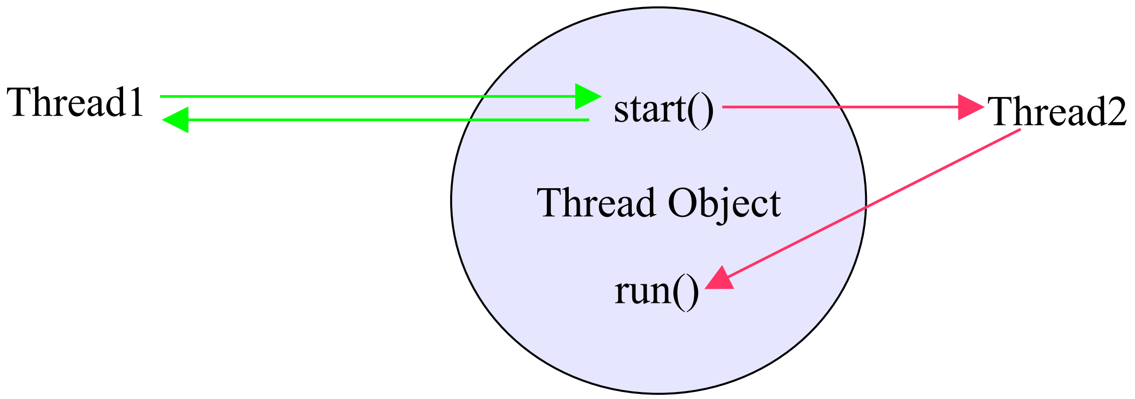
   Synchronized block.

   static synchronization.

2.Cooperation (Inter-thread communication in java)

## How does a Thread run?

- The thread class has a run() method
  - run() is executed when the thread's start() method is invoked
- The thread terminates if the run method terminates
  - To prevent a thread from terminating, the run method must not end
  - run methods often have an endless loop to prevent thread termination
- One thread starts another by calling its start method
  - The sequence of events can be confusing to those more familiar with a single threaded model.



## Terminating Thread Example

```
public class Test implements Runnable
{
  private Thread theThread;
  private boolean stopThread = false;

  public void start()
  {
       if (theThread == null)
       {
              theThread = new Thread(this);
              theThread.start();
       }
  }

  public void setStopThread(boolean aValue)
  {
       stopThread = aValue;
  }

  public void run()
  {
       while(true)
       {
              if (stopThread)
                     break;
              //
```

## Yield() and Sleep()

- Sometimes a thread can determine that it has nothing to do
  - Sometimes the system can determine this. ie. waiting for I/O
- When a thread has nothing to do, it should not use CPU
  - This is called a busy-wait.
  - Threads in busy-wait are busy using up the CPU doing nothing.
    - Often, threads in busy-wait are continually checking a flag to see if there is anything to do.
- It is worthwhile to run a CPU monitor program on your desktop
  - You can see that a thread is in busy-wait when the CPU monitor goes up (usually to 100%), but the application doesn't seem to be doing anything.
- Threads in busy-wait should be moved from the Run queue to the Wait queue so that they do not hog the CPU
  - Use yield() or sleep(time)
  - Yield simply tells the scheduler to schedule another thread
  - Sleep guarantees that this thread will remain in the wait queue for the specified number of milliseconds.

## Concurrent Access to Data

- Those familiar with databases will understand that concurrent access to data can lead to data integrity problems
  - Specifically, if two sources attempt to update the same data at the same time, the result of the data can be undefined.
  - The outcome is determined by how the scheduler schedules the two sources.
    - Since the schedulers activities cannot be predicted, the outcome cannot be predicted
- Databases deal with this mechanism through "locking"
  - If a source is going to update a table or record, it can lock the table or record until such time that the data has been successfully updated.
  - While locked, all access is blocked except to the source which holds the lock.
- Java has the equivalent mechanism.  It is called synchronization
  - Java has a keyword called synchronized

## Synchronization

- In Java, every object has a lock
  - To obtain the lock, you must synchronize with the object
- The simplest way to use synchronization is by declaring one or more methods to be synchronized
  - When a synchronized method is invoked, the calling thread attempts to obtain the lock on the object.
    - if it cannot obtain the lock, the thread goes to sleep until the lock becomes available
  - Once the lock is obtained, no other thread can obtain the lock until it is released. ie, the synchronized method terminates
  - When a thread is within a synchronized method, it knows that no other synchronized method can be invoked by any other thread
    - Therefore, it is within synchronized methods that critical data is updated

## Providing Thread Safe Access to Data

- If an object contains data which may be updated from multiple thread sources, the object should be implemented in a thread-safe manner
  - All access to critical data should only be provided through synchronized methods (or synchronized blocks).
  - In this way, we are guaranteed that the data will be updated by only one thread at a time.

```
public class SavingsAccount
{
  private float balance;

  public synchronized void withdraw(float anAmount)
  {
       if ((anAmount>0.0) && (anAmount<=balance))
              balance = balance - anAmount;
  }

  public synchronized void deposit(float anAmount)
  {
       if (anAmount>0.0)
              balance = balance + anAmount;
  }
```

## Thread Safety Performance Issues

- However, there is an overhead associated with synchronization
  - Many threads may be waiting to gain access to one of the object's synchronized methods
  - The object remains locked as long as a thread is within a synchronized method.
  - Ideally, the method should be kept as short as possible.
- Another solution is to provide synchronization on a block of code instead of the entire method
  - In this case, the object's lock is only held for the time that the thread is within the block.
  - The intent is that we only lock the region of code which requires access to the critical data.  Any other code within the method can occur without the lock.
  - In high load situations where multiple threads are attempting to access critical data, this is by far a much better implementation.

# Block Synchronization

```
public class SavingsAccount
{
  private float balance;

  public void withdraw(float anAmount)
  {
       if (anAmount<0.0)
              throw new IllegalArgumentException("Withdraw amount negative");
       synchronized(this)
       {
              if (anAmount<=balance)
                     balance = balance - anAmount;
       }
  }

  public void deposit(float anAmount)
  {
       if (anAmount<0.0)
              throw new IllegalArgumentException("Deposit amount negative");
       synchronized(this)
       {
              balance = balance + anAmount;
       }
  }
```

## Inter Thread Communication

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

It is implemented by following method of **Object class**:

wait()

notify()

notifyAll()

- wait()-Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
- public final void wait()throws InterruptedException
- public final void wait(long timeout)throws InterruptedException

- **notify()** method Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation
- **public final void notify()**
- **notifyAll()** method Wakes up all threads that are waiting on this object's monitor.
- **public final void notifyAll()**

- Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException.

If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.

- **public void interrupt()**
- **public static boolean interrupted()**
- **public boolean isInterrupted()**

## Join method:

- Thread class provides the **join() method** which allows one thread to wait until another thread completes its execution. If t is a Thread object whose thread is currently executing, then t. **join()** will make sure that t is terminated before the next instruction is executed by the program.

- **Why we use join() method?**
- In normal circumstances we generally have more than one thread, thread scheduler schedules the threads, which does not guarantee the order of execution of threads.

## Join method:

- **Without using join()**
- Here we have three threads th1, th2 and th3. Even though we have started the threads in a sequential manner the thread scheduler does not start and end them in the specified order. Everytime you run this code, you may get a different result each time.
- **The same example with join()**
- Lets say our requirement is to execute them in the order of first, second and third. We can do so by using join() method appropriately.