5.1 INTRODUCTION

SPREAD SPECTRUM

- Analog or digital data
- Analog signal
- Spread data over wide bandwidth
- Makes jamming and interception harder
- Frequency hoping
- Signal broadcast over seemingly random series of frequencies
- Direct Sequence
- Each bit is represented by multiple bits in transmitted signal
- Chipping code
- Spread Spectrum Concept
- Input fed into channel encoder
- Produces narrow bandwidth analog signal around central frequency
- Signal modulated using sequence of digits
- Spreading code/sequence
- Typically generated by pseudonoise/pseudorandom number generator
- Increases bandwidth significantly
- Spreads spectrum
- Receiver uses same sequence to demodulate signal
- Demodulated signal fed into channel decoder.

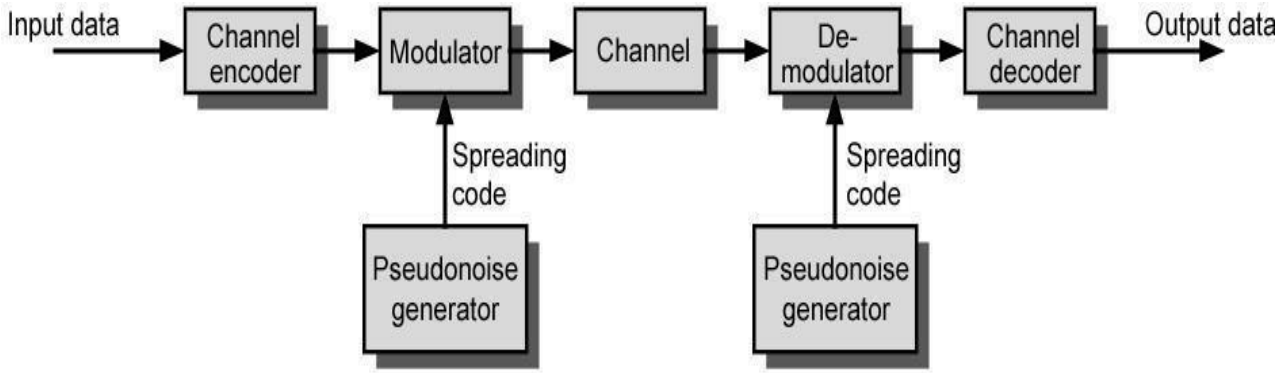

Figure 5.1.1 General Model of Spread Spectrum System

PSEUDORANDOM NUMBERS

- Generated by algorithm using initial seed
- Deterministic algorithm
- Not actually random
- If algorithm good, results pass reasonable tests of randomness
- Need to know algorithm and seed to predict sequence

5.2 FREQUENCY HOPPING SPREAD SPECTRUM (FHSS)

- Signal broadcast over seemingly random series of frequencies
- Receiver hops between frequencies in sync with transmitter
- Eavesdroppers hear unintelligible blips
- Jamming on one frequency affects only a few bits

Basic Operation

- Typically 2^k carriers frequencies forming 2^k channels
- Channel spacing corresponds with bandidth of input
- Each channel used for fixed interval
- 300 ms in IEEE 802.11
- Some number of bits transmitted using some encodingscheme
- May be fractions of bit (see later)
- Sequence dictated by spreading code

FREQUENCY HOPPING EXAMPLE

We now consider a spread-spectrum technique called frequency hopping (FH). The modulation most commonly used with this technique is M-ary frequency shift keying (MFSK), where k = log2M information bits are used to determine which one of M frequencies is to be transmitted. The position of the M-ary signal set is shifted pseudo randomly by the frequency synthesizer over a hopping bandwidth Wss. A typical FH/MFSK system block diagram is shown in Figure 3.1. In a conventional MFSK system, the data symbol modulates a fixed frequency carrier; in an FH/MFSK system, the data symbol modulates a carrier whose frequency is pseudo-randomly determined. In either case, a single tone is transmitted. The FH system in Figure 3.1 can be thought of as a two-step modulation process data modulation and frequency-hopping modulation even though it can be implemented as a single step whereby the frequency synthesizer produces a transmission tone based on the simultaneous dictates of the PN code and the data. At each frequency hop time, a PN generator feeds the frequency

synthesizer a frequency word (a sequence of t chips), which dictates one of 2f symbol-set positions. The frequency-hopping bandwidth Wss and the minimum frequency spacing between consecutive hop positions Af, dictate the minimum number of chips necessary in the frequency word.

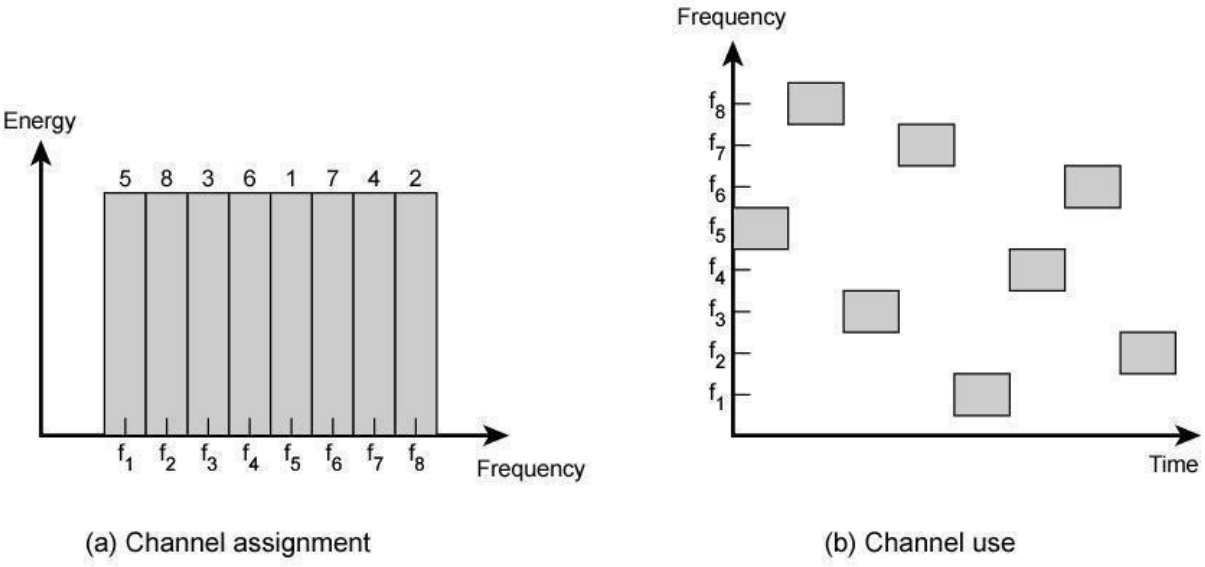

Figure 5.2. 1 (a) Channel Assignment (b) Channel Use

MODULATION FHSS

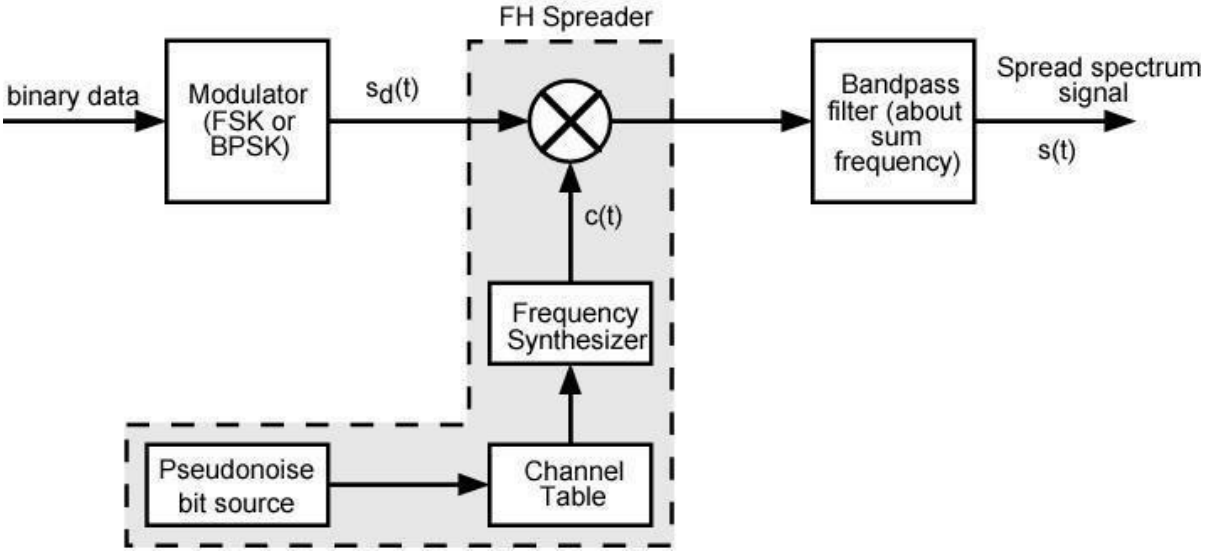

Figure 5.2.3 Modulation FHSS

For a given hop, the occupied transmission bandwidth is identical to the bandwidth of conventional MFSK, which is typically much smaller than Wss. However, averaged over many hops, the FH/MFSK spectrum occupies the entire

spread-spectrum bandwidth. Spread-spectrum technology permits FH bandwidths of the order of several giga hertz, which is an order of magnitude larger than implementable DS bandwidths, thus allowing for larger processing gains in FH compared to DS systems. Since frequency hopping techniques operate over such wide bandwidths, it is difficult to maintain phase coherence from hop to hop. Therefore, such schemes are usually configured using noncoherent demodulation. Nevertheless, consideration has been given to coherent FH in Reference. In Figure 3.1 we see that the receiver reverses the signal processing steps of the transmitter. The received signal is first FH demodulated (dehopped) by mixing it with the same sequence of pseudo randomly selected frequency tones that was used for hopping. Then the dehopped signal is applied to a conventional bank of M noncoherent energy detectors to select the most likely symbol.

DEMODULATION FHSS

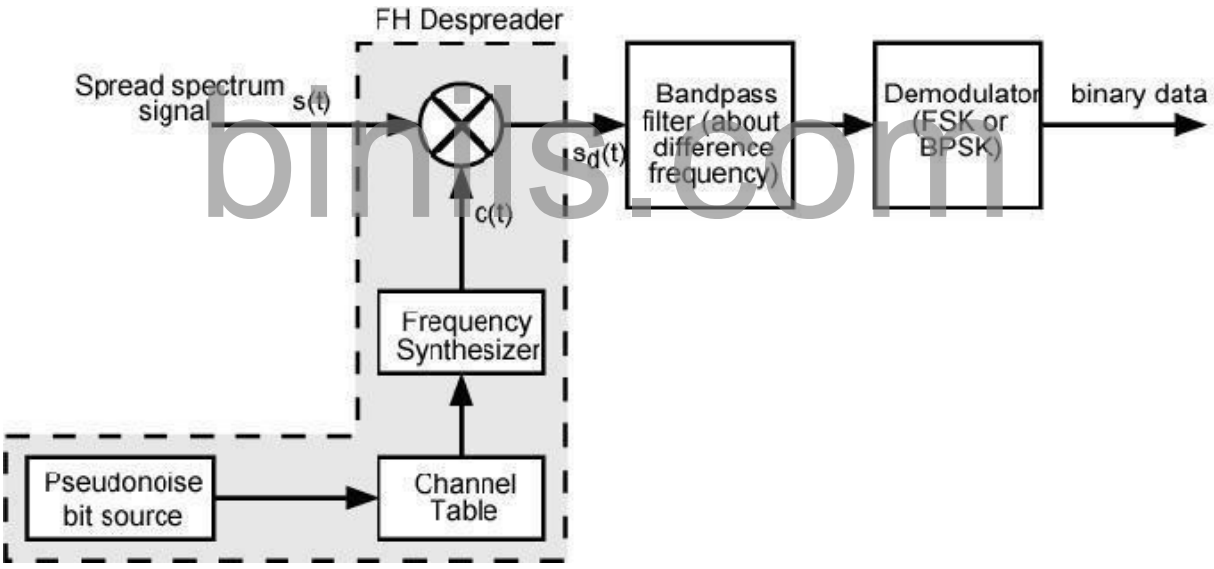

Figure 5.2.4 Demodulation FHSS

Consider the frequency hopping example illustrated in Figure 3.2. The input data consist of a binary sequence with a data rate of R = 150 bits/s. The modulation is 8-ary FSK. Therefore, the symbol rate is Rs = R/(log2 8) = 50 symbols/s (the symbol duration T = 1/50 = 20 ms). The frequency is hopped once per symbol,

and the hopping is time synchronous with the symbol boundaries. Thus, the hopping rate is 50 hops/s. Figure 3.2 depicts the timebandwidth plane of the communication resource; the abscissa represents time, and the ordinate represents the hopping bandwidth, Wss. The legend on the right side of the figure illustrates a set of 8-ary FSK symbol-to-tone assignments. Notice that the tone separation specified is 1/T = 50 Hz, which corresponds to the minimum required tone spacing for the orthogonal signaling of this noncoherent FSK example. A typical binary data sequence is shown at the top of Figure 3.2. Since the modulation is 8-ary FSK, the bits are grouped three at a time to form symbols. In a conventional 8-ary FSK scheme, a single-sideband tone (offset from fo, the fixed center frequency of the data band), would be transmitted according to an assignment like the one shown in the legend. The only difference in this FH/MFSK example is that the center frequency of the data band fo is not fixed. For each new symbol, fo hops to a new position in the hop bandwidth, and the entire data-band structure moves with it. In the example of Figure 3.2, the first symbol in the data sequence, 0 1 1, yields a tone 25 Hz above fo. The diagram depicts fo with a dashed line and the symbol tone with a solid line. During the second symbol interval, fo has hopped to a new spectral location, as indicated by the dashed line. The second symbol, 1 1 0, dictates that a tone indicated by the solid line, 125 Hz below f o, shall be transmitted. Similarly, the final symbol in this example, 0 0 1, calls for a tone 125 Hz above fo. Again, the center frequency has moved, but the relative positions of the symbol tones remain fixed.

SLOW AND FAST FHSS

- Frequency shifted every T_c seconds
- Duration of signal element is T_s seconds
- Slow FHSS has $T_c < T_s$
- Fast FHSS has $T_c < T_s$
- Generally fast FHSS gives improved performance in noise(or jamming)

In the case of direct-sequence spread-spectrum systems, the term "chip" refers to the PN code symbol (the symbol of shortest duration in a DS system). In a similar sense for frequency hopping systems, the term "chip" is used to characterize the shortest uninterrupted waveform in the system. Frequency hopping systems are classified as slow frequency hopping (SFH), which means there are several modulation symbols per hop, or as fastfrequency hopping (FFH), which means that there are several frequency hops per modulation symbol. For SFH, the shortest uninterrupted waveform in the system is that of the data symbol; however, for FFH, the shortest uninterrupted waveform is that of the hop. Figure 3.4a illustrates an example of FFH; the data symbol rate is 30 symbols/s and the frequency hopping rate is 60 hops/s. The figure illustrates the waveform s(t) over one symbol duration (1/30s). The waveform change in (the middle of) s(t) is due to a new frequency hop. In this example, a chip corresponds to a hop since the hop duration is shorter than the symbol duration. Each chip corresponds to half a symbol. Figure 3.4b illustrates an example of SFH; the data symbol rate is still 30 symbols/s, but the frequency hopping rate has been reduced to 10 hops/s. The waveform s(t) is shown over a duration of three symbols (1/10 S). In this example, the hopping boundaries appear only at the beginning and end of the three-symbol duration. Here, the changes in the waveform are due to the modulation state changes; therefore, in this

5.3 DIRECT SEQUENCE SPREAD SPECTRUM

DSSS, direct sequence spread spectrum is a form of spread spectrum transmission which uses spreading codes to spread the signal out over a wider bandwidth then would normally be required.

The technique behind direct sequence spread spectrum, DSSS is at first sight counter-intuitive, but DSSS is used in a number of areas where it enables considerable benefits to be gained.

Direct sequence spread spectrum is a form of transmission that looks very similar to white noise over the bandwidth of the transmission. However once received and processed with the correct descrambling codes, it is possible to extract the required data.

When transmitting a DSSS spread spectrum signal, the required data signal is multiplied with what is known as a spreading or chip code data stream. The resulting data stream has a higher data rate than the data itself. Often the data is multiplied using the XOR (exclusive OR) function.

- Each bit represented by multiple bits using spreading code
- Spreading code spreads signal across wider frequency band
- In proportion to number of bits used
- 10 bit spreading code spreads signal across 10 times bandwidth of 1 bit code

One method:

- Combine input with spreading code using XOR
- Input bit 1 inverts spreading code bit
- Input zero bit doesn’t alter spreading code bit
- Data rate equal to original spreading code
- Performance similar to FHSS

DIRECT SEQUENCE SPREAD SPECTRUM TRANSMITTER

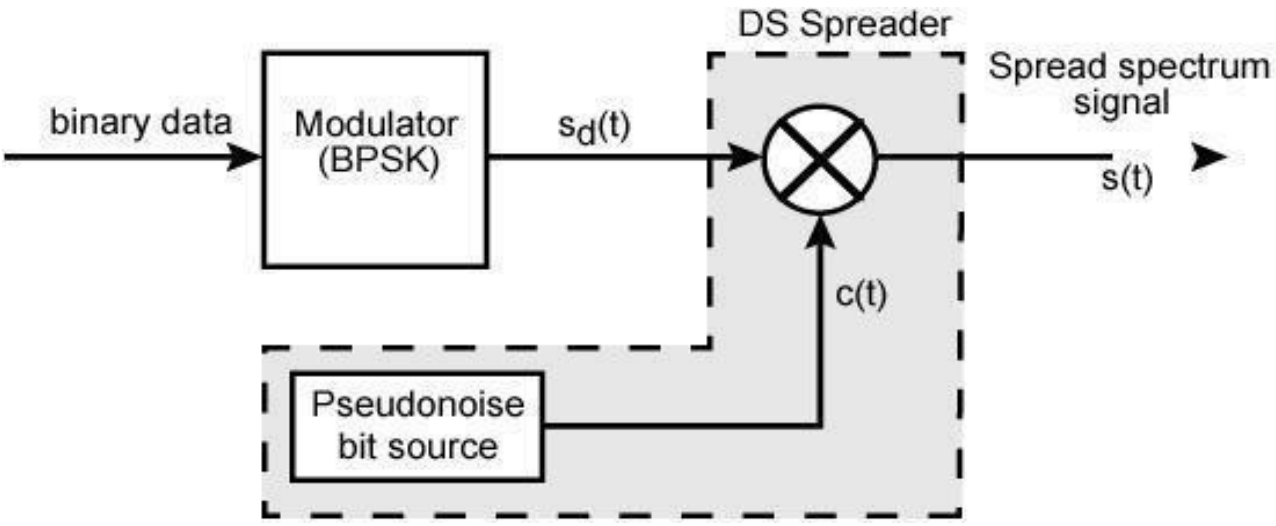

Figure 5.3.1 Direct Sequence Spread Spectrum Transmitter

Each bit in the spreading sequence is called a chip, and this is much shorter than each information bit. The spreading sequence or chip sequence has the same data rate as the final output from the spreading multiplier. The rate is called the chip rate, and this is often measured in terms of a number of M chips / sec.

The baseband data stream is then modulated onto a carrier and in this way the overall the overall signal is spread over a much wider bandwidth than if the data had been simply modulated onto the carrier. This is because, signals with high data rates occupy wider signal bandwidths than those with low data rates.

To decode the signal and receive the original data, the CDMA signal is first demodulated from the carrier to reconstitute the high speed data stream. This is multiplied with the spreading code to regenerate the original data. When this is done, then only the data with that was generated with the same spreading code is regenerated, all the other data that is generated from different spreading code streams is ignored.

DIRECT SEQUENCE SPREAD SPECTRUM RECEIVER

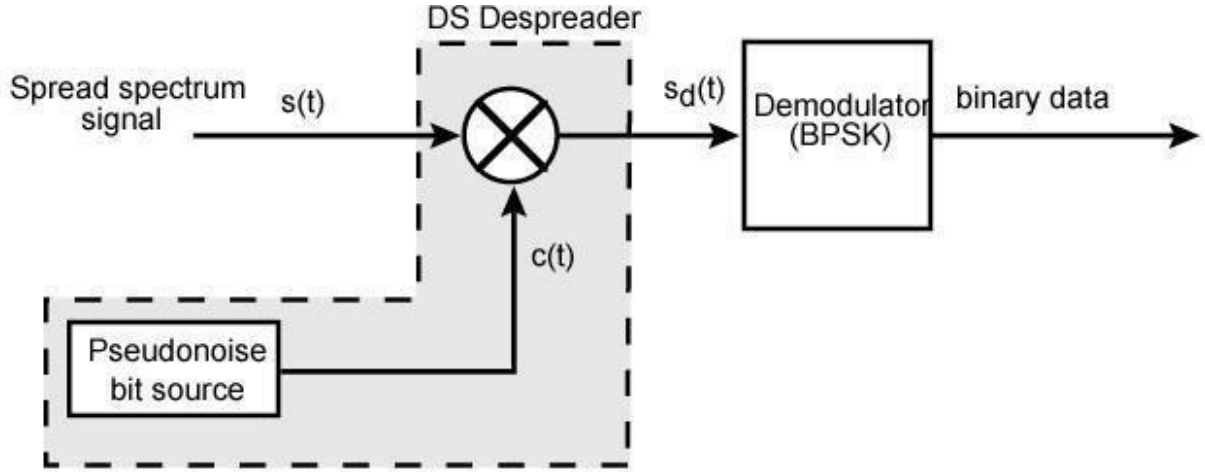

Figure 5.3.1 Direct Sequence Spread Sectrum Receiver

In order to visualize how the direct sequence spread spectrum process operates, the easiest method is to show an example of how the system actually operates in terms of data bits, and how the data is recovered from the DSSS, direct sequence spread spectrum signal.

The first part of the process is to generate the DSSS signal. Take as an example that the data to be transmitted is 1001, and the chip or spreading code is 0010. For each data bit, the complete spreading code is used to multiple the data, and in this way, for each data bits, the spread or expanded signal consists of four bits.

DSSS SPREADING GAIN

The bandwidth of the spread spectrum signal will be much wider than the original data stream. To quantify the increase in bandwidth, a term known as the spreading gain is used. If the bandwidth of the DSSS, direct sequence spread spectrum signal is **W** and the input data bit length or period **1/R** then the DSSS spreading gain can be defined:

$$\text{Spreading gain} = \frac{W}{R}$$

DIRECT SEQUENCE SPREAD SPECTRUM APPLICATIONS

DSSS is used in a number of areas where its properties have enabled it to provide some unique advantages over other techniques.

- ***Covert communications:*** DSSS was first used to provide secure and covert communications. The signals were initially difficult to detect as they sounded like broadband noise and often would have been mistaken for that. Also to access the data it is necessary to know the code used to generate the signal
- ***CDMA cellphone technology:*** The DSSS technique was used to provide a multiple access scheme that was used for 3G cellophane technology. Each mobile used a different access code or spreading code and this enabled multiple users to access the base station on the same frequency.

5.4 MULTIPLR ACCESS TECHNIQUES

The transmission from the BS in the downlink can be heard by each and every mobile user in the cell, and is referred as *broadcasting*. Transmission from the mobile users in the uplink to the BS is many-to-one, and is referred to as multiple access.

Multiple access schemes to allow many users to share simultaneously a finite amount of radio spectrum resources.

Should not result in severe degradation in the performance of the system as compared to a single user scenario.

Approaches can be broadly grouped into two categories: narrowband and wideband.

Multiple Accessing Techniques : with possible conflict and conflict- free

- Random access
- Frequency division multiple access (FDMA)
- Time division multiple access (TDMA)
- Spread spectrum multiple access (SSMA) : an example is Code division multiple access (CDMA)
- Space division multiple access (SDMA)
- For voice or data communications, must assure two way communication (duplexing, it is possible to talk and listen simultaneously). Duplexing may bedone using frequency or time domain techniques.
    - Forward (downlink) band provides traffic from the BS to the mobile
    - Reverse (uplink) band provides traffic from the mobile to the BS.
- Provides two distinct bands of frequencies for every user, one for downlink andone for uplink.
- A large interval between these frequency bands must be allowed so that interference is minimized.
- Frequency separation should be carefully decidedFrequency separation is constant
- In TDD communications, both directions of transmission use one contiguous frequency allocation, but two separate time slots to provide both a forward andreverse link.
- Because transmission from mobile to BS and from BS to mobile

alternates intime, this scheme is also known as “ping pong”.

- As a consequence of the use of the same frequency band, the communication quality in both directions is the same. This is different fromFDD.

5.5 FREQUENCY DIVISION MULTIPLE ACCESS

In FDMA, each user is allocated a unique frequency band or channel. During the period of the call, no other user can share the same frequency band.

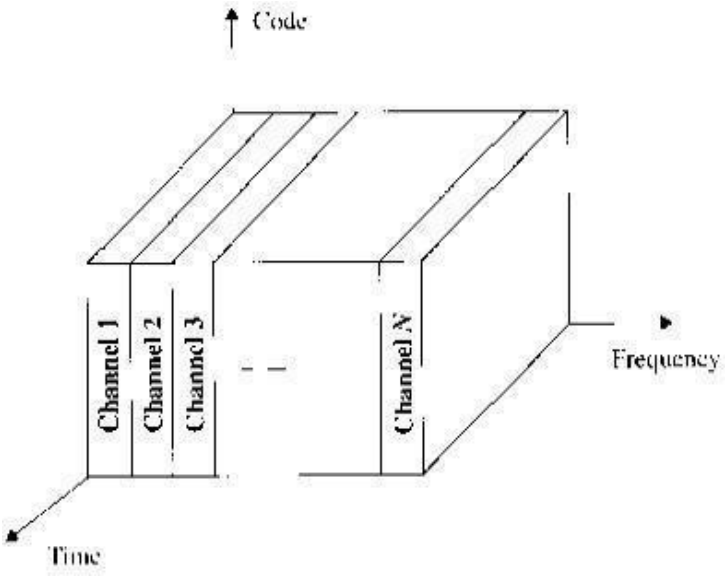

Figure 5.5.1 FDMA Channel

- All channels in a cell are available to all the mobiles. Channel assignment is carried out on a first-come first- served basis.
- The number of channels, given a frequency spectrum *BT* , depends on the modulation technique (hence *Bw* or *Bc*) and the guard bands between the channels 2*Bguard* . These guard bands allow for imperfect filters and oscillators and can be used to minimize adjacent channel interference.
- FDMA is usually implemented in narrowband systems.

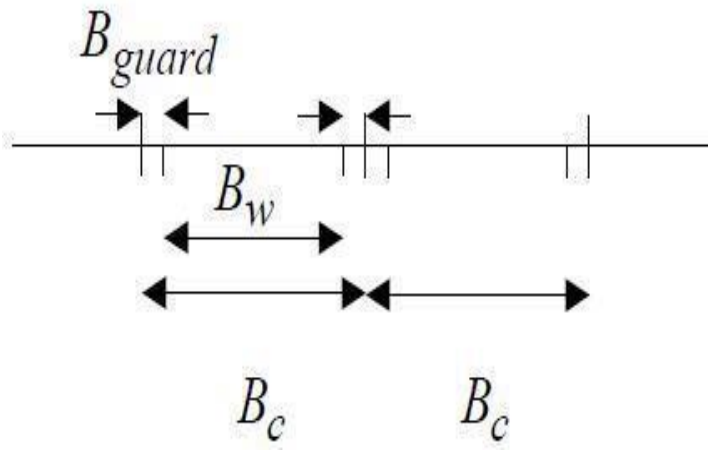

Figure 5.5.2 Frequency Spectrum

Continuous transmission: the channels, once assigned, are used on a non-time-sharing basis. This means that both subscriber and BS can use their corresponding allotted channels continuously and simultaneously.

Narrow bandwidth: Analog cellular systems use 25-30 kHz. Digital FDMA systems can make use of low bit rate speech coding techniques to reduce the channel band even more.

If FDMA channels are not in use, then they sit idle and cannot be used by other users to increase capacity.

Low ISI: Symbol time is large compared to delay spread. No equalizer is required(Delay spread is generally less than a few μs – flat fading).

Low overhead : Carry overhead messages for control, synchronization purposes. Asthe allotted channels can be used continuously, fewer bits need to be dedicated compared to TDMA channels.

Simple hardware at mobile unit and BS : (1) no digital processing needed tocombat ISI (2) ease of framing and synchronization.

Use of duplexer since both the transmitter and receiver operate at the same time. This results in an increase in the cost of mobile and BSs.

FDMA required tight RF filtering to minimize adjacent channelinterference.

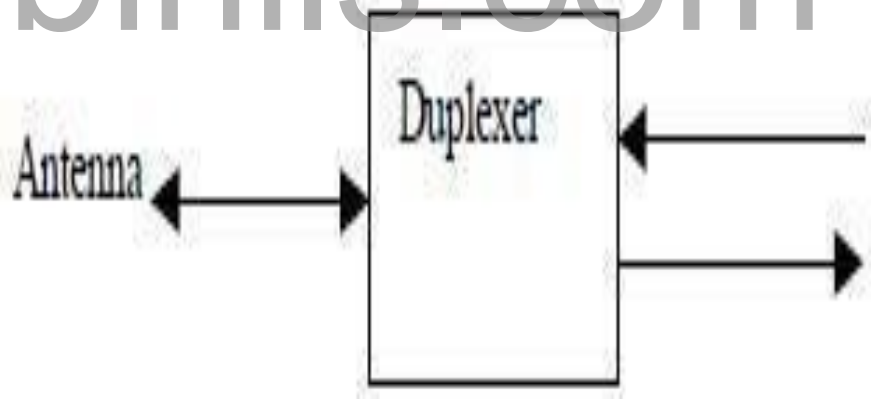

Figure 5.5.3 Duplexer

TIME DIVISION MULTIPLE ACCESS

- TDMA systems divide the channel time into frames. Each frame is further partitioned into time slots. In each slot only one user is allowed to either transmitor receive.
- Unlike FDMA, only digital data and digital modulation must be used.
- Each user occupies a cyclically repeating time slot, so a channel may be thought of as a particular time slot of every frame, where *N* time slots

comprisea frame.

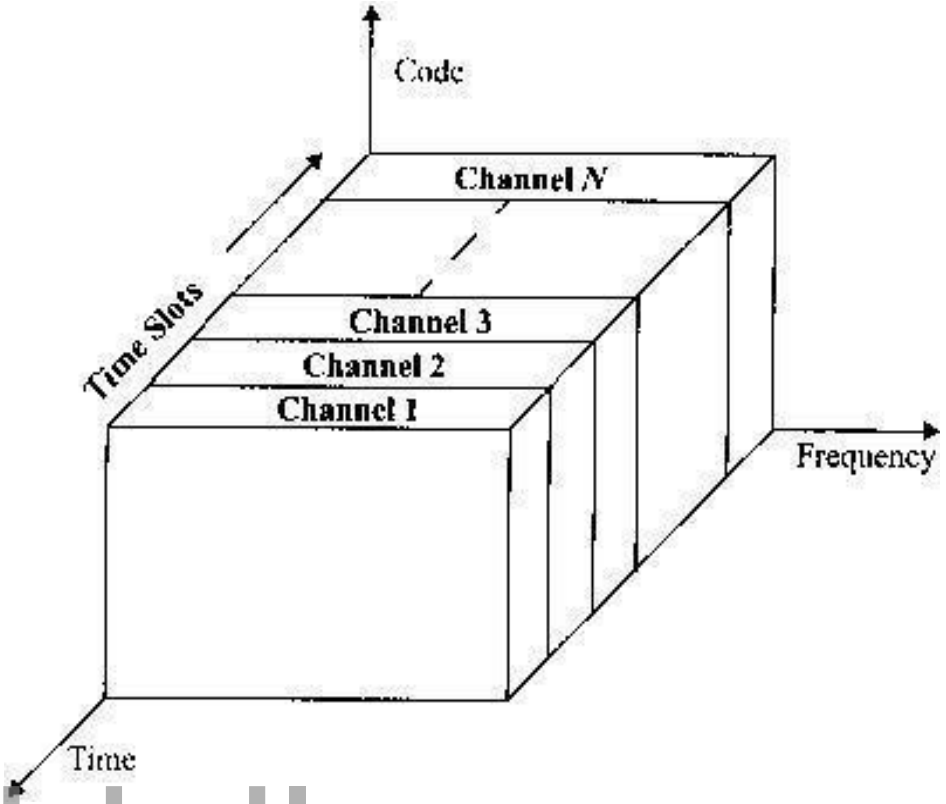

Figure 5.5.4 TDMA Channel

- Multiple channels per carrier or RF channels.
- Burst transmission since channels are used on a timesharing basis. Transmitter canbe turned off during idle periods.
- Narrow or wide bandwidth – depends on factors such as modulation scheme,number of voice channels per carrier channel.
- High ISI – Higher transmission symbol rate, hence resulting in high ISI. Adaptiveequalizer required.

FEATURES

High framing overhead – A reasonable amount of the total transmitted bits must be dedicated to synchronization purposes, channel identification. Alsoguard slots are necessary to separate users.

TDMA Frame

One TDMA Frame

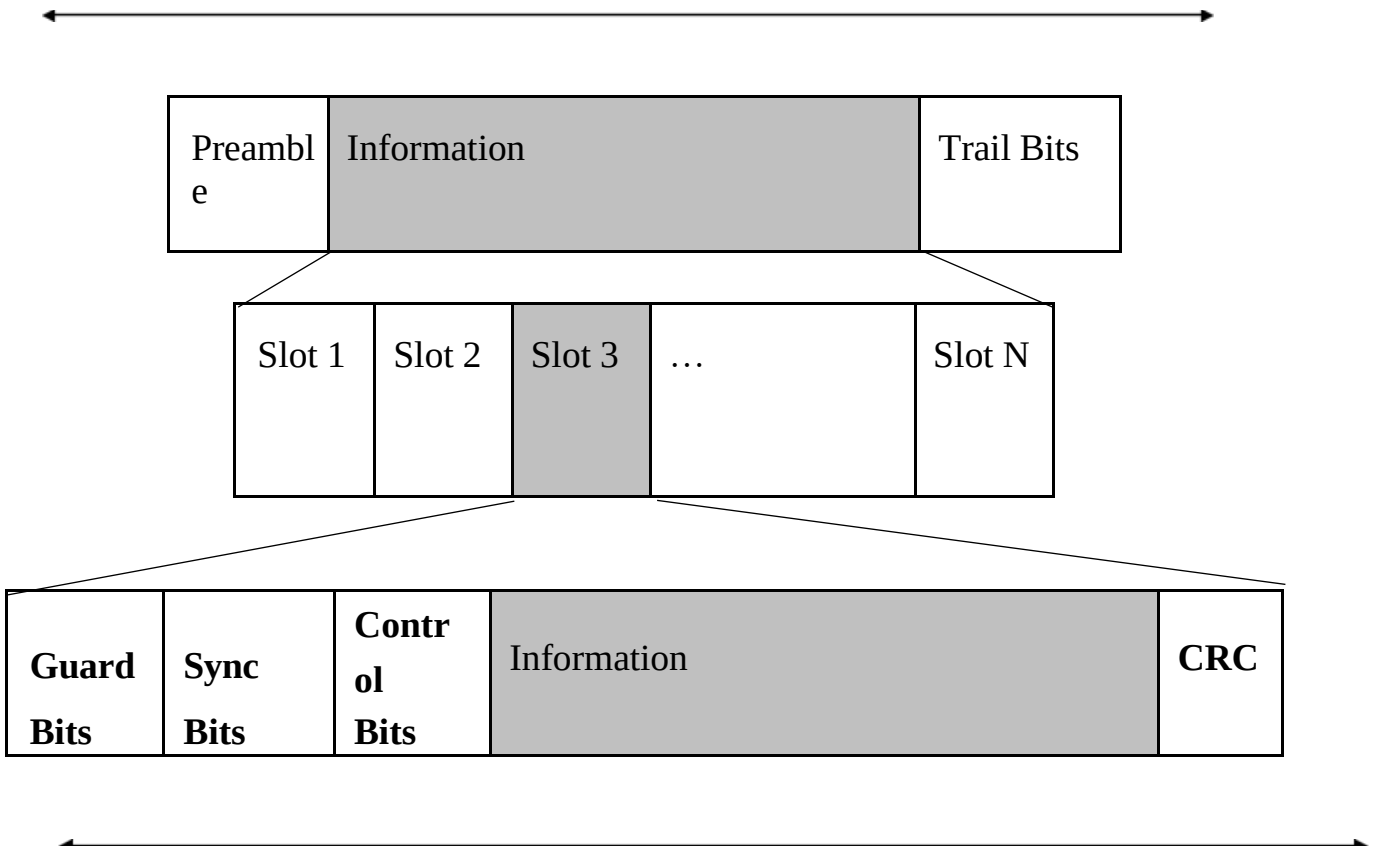

Figure 5.5.5 TDMA Frame

A Frame repeats in time

CODE DIVISION MULTIPLE ACCESS (CDMA)

- In CDMA, the narrowband message signal is *multiplied* by a very large bandwidthsignal called spreading signal (code) before modulation and transmission over the air. This is called spreading.

- CDMA is also called DSSS (Direct Sequence Spread Spectrum). DSSS is a moregeneral term.
- Message consists of symbols
- Has symbol period and hence, symbol rate
- Spreading signal (code) consists of chips
- Has Chip period and and hence, chip rate
- Spreading signal use a pseudo-noise (PN) sequence (a pseudo-randomsequence)
- PN sequence is called a codeword

- Each user has its own cordword
- Codewords are orthogonal. (low autocorrelation)
- Chip rate is order of magnitude larger than the symbol rate.
- The receiver correlator distinguishes the senders signal by examining the wideband signal with the same time-synchronized spreading code
- The sent signal is recovered by despreading process at the receiver.

CDMA ADVANTAGES

- Low power spectral density.
- Signal is spread over a larger frequency band
- Other systems suffer less from the transmitter
- Interference limited operation
- All frequency spectrum is used
- Privacy
- The codeword is known only between the sender and receiver. Hence other userscan not decode the messages that are in transit
- Reduction of multipath affects by using a larger spectrum

THREAD LIFE CYCLE

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1) New
2) Runnable
3) Blocked
4) Waiting
5) Timed Waiting
6) Terminated

The following figure represents various states of a thread at any instant of time:

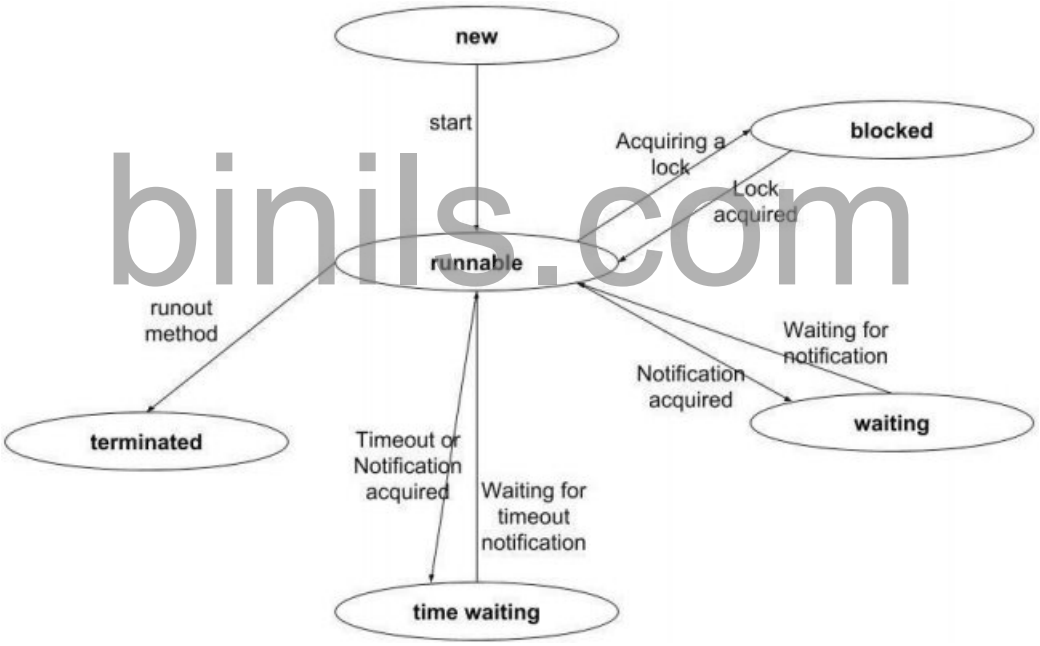

Figure: Life Cycle of a thread

1. New Thread:

- When a new thread is created, it is in the new state.
- The thread has not yet started to run when thread is in this state.
- When a thread lies in the new state, it's code is yet to be run and hasn't started to execute.

2. Runnable State:

- A thread that is ready to run is moved to runnable state.
- In this state, a thread might actually be running or it might be ready run at any instant of time.
- It is the responsibility of the thread scheduler to give the thread, time to run.
- A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

***3.* Blocked/Waiting state:**

- When a thread is temporarily inactive, then it's in one of the following states:
  - Blocked
  - Waiting
- For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/ waiting thread.
- A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states do not consume any CPU cycle.
- A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the threads which is blocked for that section and moves it to the runnable state. A thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.
- If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

***4.* Timed Waiting:**

- A thread lies in timed waiting state when it calls a method with a time out parameter.
- A thread lies in this state until the timeout is completed or until a notification is received.
- For example, when a thread calls sleep or a conditional wait, it is moved to timed waiting state.

***5.* Terminated State:**

- A thread terminates because of either of the following reasons:

- Because it exits normally. This happens when the code of thread has entirely executed by the program.
- Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

- A thread that lies in terminated state does no longer consumes any cycles of CPU.

Creating threads

- Threading is a facility to allow multiple tasks to run concurrently within a single process. Threads are independent, concurrent execution through a program, and each thread has its own stack.

In Java, There are two ways to create a thread:

1) By extending Thread class.
2) By implementing Runnable interface.

Java Thread Benefits

1. Java Threads are lightweight compared to processes as they take less time and resource to create a thread.
2. Threads share their parent process data and code
3. Context switching between threads is usually less expensive than between processes.
4. Thread intercommunication is relatively easy than process communication.

THREAD CLASS

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Commonly used methods of thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
1. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
2. **public int getPriority():** returns the priority of the thread.

3. **public int setPriority(int priority):** changes the priority of the thread.
4. **public string getname():** returns the name of the thread.
5. **public void setname(string name):** changes the name of the thread.
6. ***public thread currentthread():*** returns the reference of currently executing thread.

7. **public int getid():** returns the id of the thread.
8. **public thread.state getstate():** returns the state of the thread.
9. **public boolean isalive():** tests if the thread is alive.
10. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
11. **public void suspend():** is used to suspend the thread(depricated).
12. **public void resume():** is used to resume the suspended thread(depricated).
13. **public void stop():** is used to stop the thread(depricated).
14. **public boolean isdaemon():** tests if the thread is a daemon thread.
15. **public void setdaemon(boolean b):** marks the thread as daemon or user thread.
16. **public void interrupt():** interrupts the thread.
17. **public boolean isinterrupted():** tests if the thread has been interrupted.
18. **public static boolean interrupted():** tests if the current thread has been interrupted.

naming thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. But we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

public string getname(): is used to return the name of a thread.

public void setname(string name): is used to change the name of a thread.

extending thread

The first way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

Sample java program that creates a new thread by extending Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread
{
   NewThread()
   { // Create a new, second thread super("Demo
      Thread"); System.out.println("Child thread: "
      + this); start(); // Start the thread
   }
   // This is the entry point for the second thread.
   public void run()
```

```
    {
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Child interrupted.");
        }
        System.out.println("Child thread is exiting");
    }
}
public class ExtendThread
{
    public static void main(String args[])
    {
        new NewThread(); // create a new thread
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread is exiting.");
    }
```

```
}
```

Sample Output:

(output may vary based on processor speed and task load)

Child thread: Thread[Demo Thread,5,main]

Main Thread: 5

Child Thread: 5

Child Thread: 4

Main Thread: 4

Child Thread: 3

Child Thread: 2

Main Thread: 3

Child Thread: 1

Child thread is exiting.

Main Thread: 2

Main Thread: 1

Main thread is exiting.

The child thread is created by instantiating an object of NewThread, which is derived from Thread. The call to super( ) is inside NewThread. This invokes the following form of the Thread constructor:

public Thread(String threadName)

Here, threadName specifies the name of the thread.

implementing runnable

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- Runnable abstracts a unit of executable code. We can construct a thread on any object that implements Runnable.
- To implement Runnable, a class need only implement a single method called run( ), which is declared as:

  public void run( )
- Inside run( ), we will define the code that constitutes the new thread. The run( ) can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within the program. This thread will end when run( ) returns.
- After we create a class that implements Runnable, we will instantiate an object of type Thread from within that class.
- After the new thread is created, it will not start running until we call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ).

- The start( ) method is shown as:

  void start( )

Sample java program that creates a new thread by implementing Runnable:

```
// Create a second thread
class NewThread implements Runnable
{
   Thread t;
   NewThread()
   {
      // Create a new, second thread
      t = new Thread(this, “Demo Thread”);
      System.out.println(“Child thread: “ + t);
      t.start(); // Start the thread
    }
// This is the entry point for the second thread.
   public void run()
   {
      try
      {
         for(int i = 5; i > 0; i--)
         {
            System.out.println(“Child Thread: “ + i);
            Thread.sleep(500);
         }
      }
      catch (InterruptedException e)
      {
         System.out.println(“Child interrupted.”);
      }
      System.out.println(“Child thread is exiting.”);
   }
}
public class ThreadDemo
{
   public static void main(String args[])
```

```
    {
        new NewThread(); // create a new thread
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread is exiting.");
    }
}
```


Inside NewThread's constructor, a new Thread object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing this as the first argument indicates that we want the new thread to call the run( ) method on this object. Next, start( ) is called, which starts the thread of execution beginning at the run( ) method. This causes the child thread's for loop to begin. After calling start( ), NewThread's constructor returns to main(). When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish.

Sample Output:

```
(output may vary based on processor speed and task load)
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Child thread is exiting.
```

Main Thread: 2

Main Thread: 1

Main thread is exiting.

In a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may "hang." The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread.

Choosing an approach

The Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is run(). This is, of course, the same method required when we implement Runnable. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if we will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable.

Creating Multiple threads

The following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable
{
   String name; // name of thread
   Thread t;
   NewThread(String threadname)
   {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New thread: " + t);
      t.start(); // Start the thread
   }
   // This is the entry point for thread.
   public void run()
   {
      try
      {
         for(int i = 5; i > 0; i--)
         {
            System.out.println(name + ": " + i);
```

```
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}
public class MultiThreadDemo
{
    public static void main(String args[])
    {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");
        try
        {
            // wait for other threads to end
            Thread.sleep(10000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```


The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
 Two: 5
```

Three: 5

One: 4

Two: 4

Three: 4

One: 3

Three: 3

Two: 3

One: 2

Three: 2

Two: 2

One: 1

Three: 1

Two: 1

One exiting.

Two exiting.

Three exiting.

Main thread exiting.

As we can see, once started, all three child threads share the CPU. The call to sleep(10000) in main(). This causes the main thread to sleep for ten seconds and ensures that it will finish last.

using isalive( ) and join( )

We want the main thread to finish last. In the preceding examples, this is accomplished by calling sleep( ) within main( ), with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended?

Two ways exist to determine whether a thread has finished or not.

- First, we can call isAlive( ) on the thread. This method is defined by Thread.

Syntax:

final boolean isAlive( )

The isAlive( ) method returns true, if the thread upon which it is called is still running. It returns false, otherwise.

- Second, we can use join() to wait for a thread to finish.

Syntax:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it.

Sample Java program using join() to wait for threads to finish.

class NewThread implements Runnable

```
{
   String name; // name of thread
   Thread t;
   NewThread(String threadname)
   {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New thread: " + t);
      t.start(); // Start the thread
   }
   // This is the entry point for thread.
   public void run()
   {
      try
      {
         for(int i = 5; i > 0; i--)
         {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
         }
      }
      catch (InterruptedException e)
      {
         System.out.println(name + " interrupted.");
      }
      System.out.println(name + " is exiting.");
   }
}
public class DemoJoin
{
   public static void main(String args[])
   {
      NewThread ob1 = new NewThread("One");
      NewThread ob2 = new NewThread("Two");
      NewThread ob3 = new NewThread("Three");
```

```
        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());
        // wait for threads to finish
        try
        {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());
        System.out.println("Main thread is exiting.");
    }
}
```


sample output:

(output may vary based on processor speed and task load)

New thread: Thread[One,5,main]

New thread: Thread[Two,5,main]

One: 5

New thread: Thread[Three,5,main]

Two: 5

Thread One is alive: true Thread

Two is alive: true Thread Three

is alive: true Waiting for threads

to finish. Three: 5

One: 4

Two: 4

Three: 4

```
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
One    is    exiting.
Two    is    exiting.
Three is exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread is exiting.
```