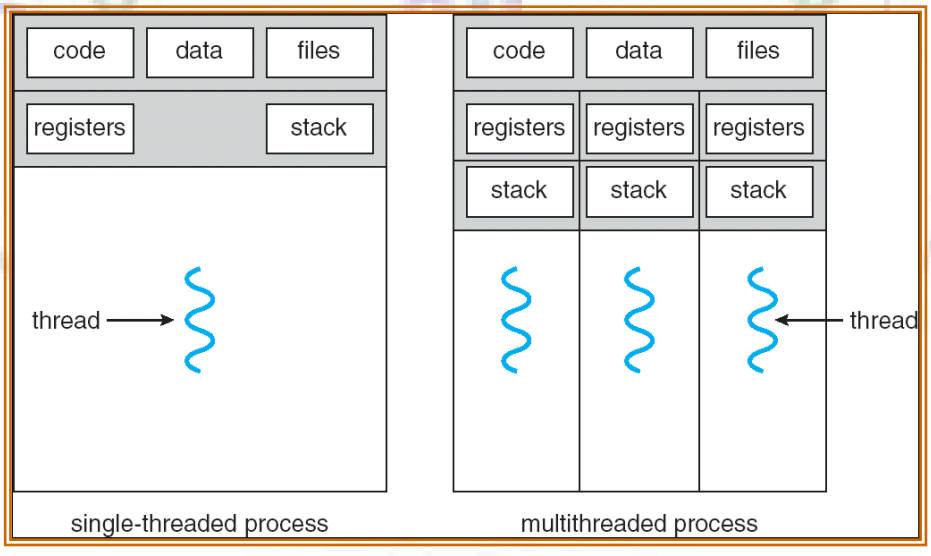
# Lesson 9, 10

## Objectives

- Threads
- Multithreading model
- Inter-process communication
- Implementation of threads in different programming languages

## THREADS

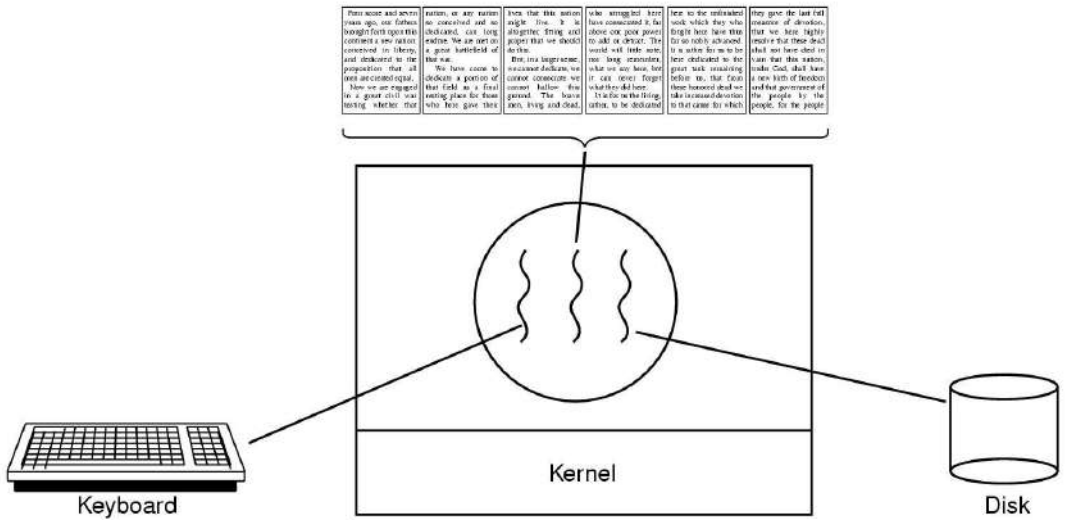
- A thread is a light weight process (LWP)
- It can be defined as a *basic unit of CPU utilization*
- It also has to maintain certain information, but it is far less than a process information requirements. The information required about:
  - Program Counter (PC)
  - Register Set (Limited)
  - Stack Space




- This information required is a fraction of *process information requirement*
- A heavy weight process (HWP) or *task* may contains more than one threads
- Single thread is always in a single task
- Threads of same task are synchronized and connected
- A thread shares with peer thread

- Its code section (Instructions)
- Data section (variables)
- OS resources
  - Files (open)
  - Signals (clocks)

- Threads are preferred since context switching in threads is faster than that of processes
- Two types of threads
  - User-level threads: where *kernel* is not involved, so run independently, no interrupts needed, hence execution is fast
  - Kernel-level thread: which involves *kernel* so interrupt occurrence is rapid. A single user call may make entire task wait until it returns (Win Xp, 2000, Solaris, Linux, Mac OS X)
- Benefits: Threads ensure cooperating, responsiveness, economy (In Solaris process creation is 30times slower than threads creation, and context switching is 5time)
- Three primary threads libraries are *i) POSIX PThreads ii) win32 threads iii) Java threads*
- Example of a word processor with three threads; for input, disk storage and display.




**Example**: CPU Vs User-threads

Let us have two processes namely P1 (single threaded) and P2 (100 threaded).

1- If these are user threads, and each process is given same CPU time, then thread 1 will get 100 times more CPU than process 2.
2- If these are *kernel* threads, then P2 will buzz the kernel 100 times more than P1 and hence will get 100 times CPU than P1.

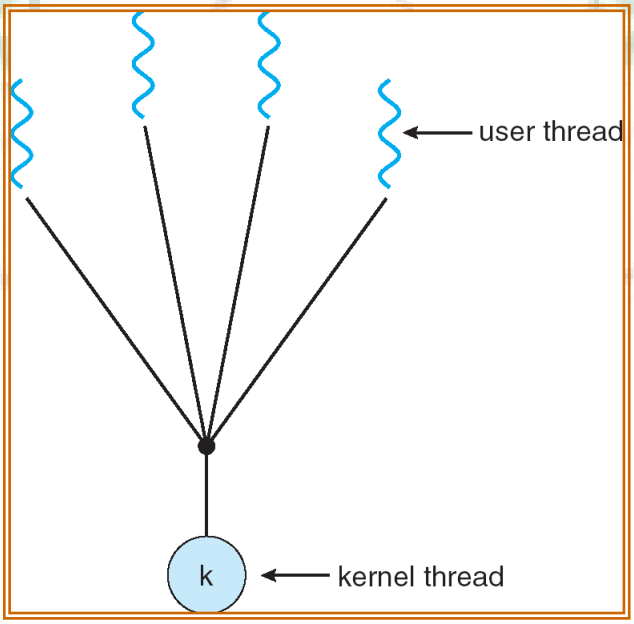
Some OS supports both types of threads, so called Hybrid threads systems.

**MULTITHREADING MODELS (Hybrid Systems)**

- Many-to-One
- One-to-One
- Many-to-Many

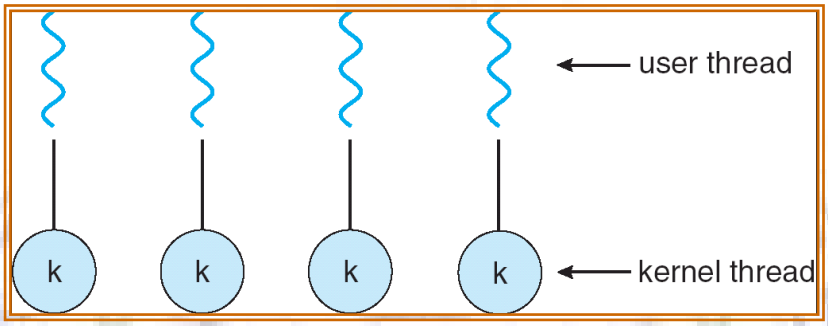
**Many-to-one**

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
- Only one thread is executing code in the kernel at a time so make wait remaining threads
- Provides synchronization of data access and locking of data is avoided.



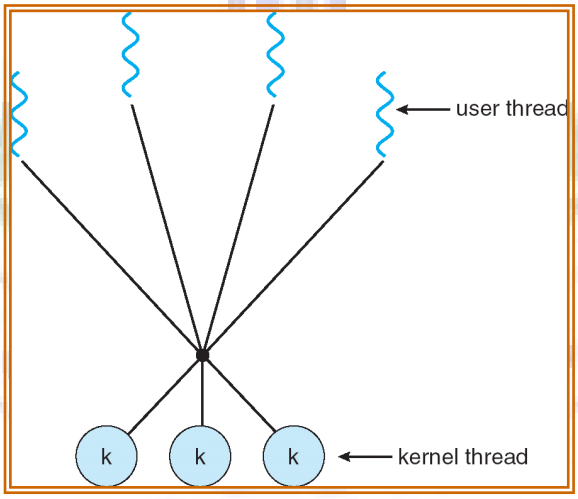
**One-to-one**

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later




**Many-to-many**

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9




**INTER-PROCESS COMMUNICATION**

There are two ways for inter-process communication

1- Through some common shared buffer (producer-consumer problem In computing, the **producer–consumer problem**[1][2] (also known as the **bounded-buffer problem**) is a classic example of a multi-process synchronizationproblem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from

the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.) explicitly managed by application program

2- Sending messages is an OS service used for synchronization and communication

Both schemes can work simultaneously

**Basic Structure**

- Two operations send / receive
- Message might be of fixed / variable length
- Existence of communication link (shared memory, H/W bus, network)

**Methods**

- Direct/ Indirect communication
- Symmetric/Asymmetric communication
- Automatic / Explicit buffering
- Send by copy or by reference
- Fixed /variable length messages

**Direct**

- Names of both ends should be known (Symmetric)
- Only single and one-to-one link between exactly two processes
- Send (P, message), *sends message to P*
- Receive (Q, message), *receives message from Q*
- Disadvantages: Requirement of names/ updates if names are changed, not suitable for separate compilation

**Indirect**

- Communication via a mailbox (ports)
- Link is subject to shared mailbox only
- Link may be among more than one process
- More than one link between two processes
- OS helps in creating, maintaining, destroying mailboxes

**Buffering**

There are three kinds of buffering:

1- **Zero capacity:** In this way the queue can't hold a message it simply make it thru. Sender has to wait until receiver receives the message. It works like a gateway. Synchronization (*rendezvous* means agreement, handshake) is basic requirements.
2- **Bounded capacity:** The queue has finite length *n*; thus at the most n messages can be absorbed. Sender will continue until buffer is filled then wait for vacancy, similarly, receiver will receive until it goes empty, then it have to wait for the next element.
3- **Unbounded capacity:** This is of considerably large space. Thus any number of messages can reside in it and sender never needs to wait for vacancy.

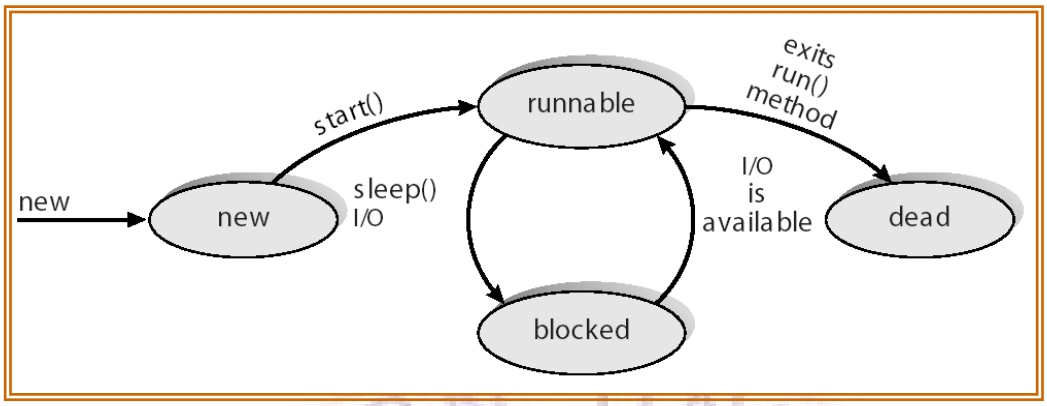
**THREADS IMPLEMENTATION IN DIFFERENT LANGUAGES**

**1- A C++ threading example using PThreads**

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

void *print_message_function( void *ptr );

main()
{
pthread_t thread1, thread2;
char *message1 = "Thread 1";
char *message2 = "Thread 2";
int  iret1, iret2;

iret1 = pthread_create( &thread1, NULL, print_message_function, (void*)
message1);
iret2 = pthread_create( &thread2, NULL, print_message_function, (void*)
message2);

     pthread_join( thread1, NULL);
pthread_join( thread2, NULL);

printf("Thread 1 returns: %d\n",iret1);
printf("Thread 2 returns: %d\n",iret2);
exit(0);
}

void *print_message_function( void *ptr )
{
char *message;
message = (char *) ptr;
printf("%s \n", message);
}
```

## 2- Java Threads (Optional)

- Java threads are managed by the JVM
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java has built in thread support for Multithreading
- Synchronization
- Thread Scheduling




- Inter-Thread Communication:
  - currentThread start setPriority
  - yield run getPriority
  - sleep stop suspend
  - resume
- Java Garbage Collector is a low-priority thread.

## 2.1 First Method

- Threads are implemented as objects that contains a method called run()

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

- Create a thread:

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

- Start Execution of threads:

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

- Create and Execute:

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

## 2.2 Second Method

```
class MyThread extends ABC implements Runnable
{
  .....
```

```
    public void run()
    {
       // thread body of execution
    }
}
```

- Creating Object:

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

- Creating Thread Object:

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

- Start Execution:

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

**Inter process communication Example in Java**

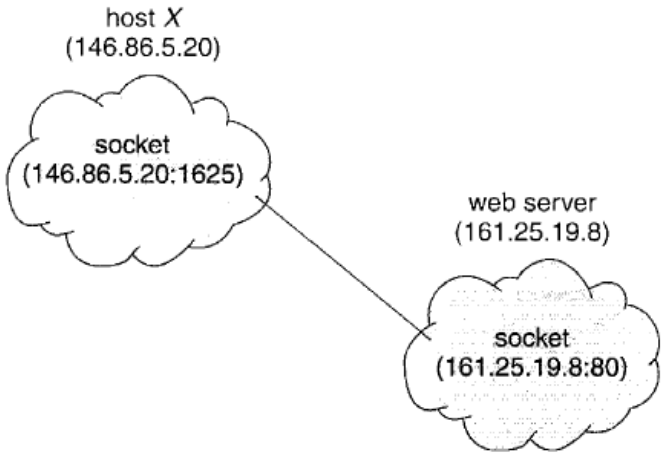
- Remote Procedure Call (RPC)
- Remote Method Invocation (RMI)

**Background**

**Socket and Client-Server Architecture**

A socket is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets-one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use client-server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.

Servers implementing specific services (such as telnet, ftp, and http) listen to well-known ports (a telnet server listens to port 23, an ftp server listens to port 21, and a web, or http, server listens to port 80). All ports below 1024 are considered well known; we can use them to implement standard services. When a client process initiates a request for a cOlu1ection, it is assigned a port by the host computer. This port is some arbitrary number greater than 1024. For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server.




Communication using sockets.

The packets traveling between the hosts are delivered to the appropriate process based on the destination port number. All connections must be unique. Therefore, if another process also on host X wished to establish another cOlli1ection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets. Although most program examples in this text use C, we will illustrate sockets using Java, as it provides a much easier interface to sockets and has a rich library for networking utilities. Those interested in socket programming in C or C++ should consult the bibliographical notes at the end of the chapter. Java provides three different types of sockets. **Connection-oriented** (TCP) sockets are implemented with the Socket class. **Connectionless** (UDP) sockets use the **DatagramSocket** class. Finally, the **MulticastSocket** class is a subclass of the **DatagramSocket** class. A multicast socket allows data to be sent to multiple recipients. Our example describes a date server that uses connection-oriented TCP sockets. The operation allows clients to request the current date and time from the server.

```
import java.net.*;
import java.io.*;
public class DateServer{
      public static void main(String[] args) {
      try {
            ServerSocket sock = new ServerSocket(6013);
// now listen for connections
            while (true) {
                  Socket client = sock.accept();
```

```
                  PrintWriter pout;
                  pout = new PrintWriter(client.getOutputStream(),
                  true);
// write the Date to the socket
                  pout.println(new java.util.Date() .toString());
// close the socket and resume
// listening for connections
                  client.close() ;
            }
      }
      catch (IOException ioe) {
            System.err.println(ioe) ;
      }
      }
}
```

Date server

```
import java.net.*;
import java.io.*;
public class DateClient{
      public static void main(String[] args) {
            try {
//make connection to server socket
                  Socket sock = new Socket("127.0.0.1",6013);
                  InputStream in = sock.getInputStream();
                  BufferedReader bin;
                  bin = new BufferedReader(new InputStreamReader(in));
// read the date from the socket
                  String line;
                  while ( (line = bin.readLine()) != null)
                        System.out.println(line) ;
// close the socket connection
                  sock.close() ;
            }
            catch (IOException ioe) {
                  System.err.println(ioe) ;
            }
      }
}
```

Date client