S J P N Trust's

Hirasugar Institute of Technology, Nidasoshi.

*Inculcating Values, Promoting Prosperity*

Accredited at 'A' Grade by NAAC

Programmes Accredited by NBA: CSE, ECE, EEE & ME

**Subject:** Operating System (18CS43)

# Multi-threaded Programming

Module 2- Chapter 4

By: Prof. Prasanna Patil
Asst. Prof. , Dept. of Computer Science & Engg.,
Hirasugar Institute of Technology, Nidasoshi

# Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues

## Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

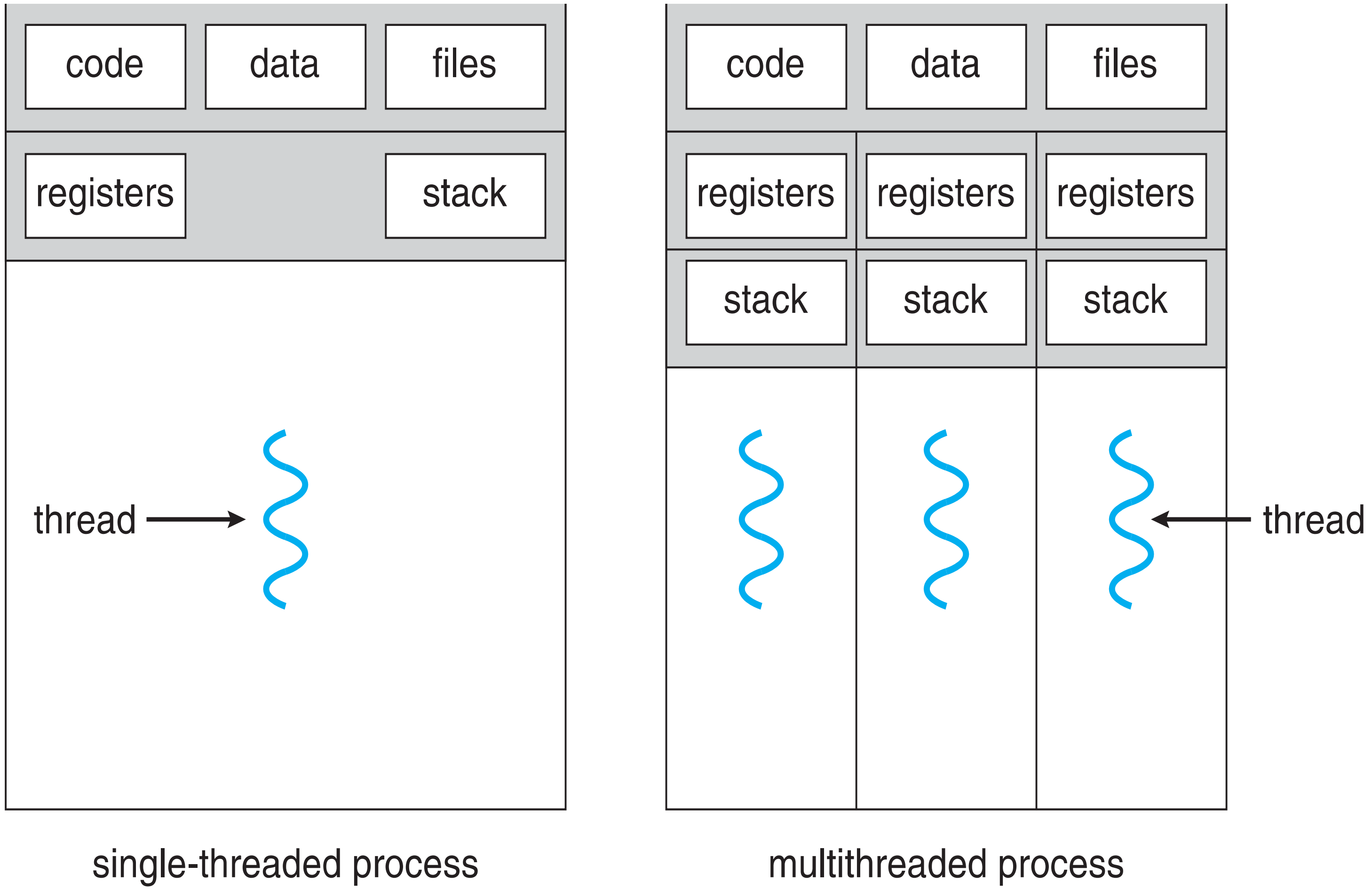
# Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

## Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

# Single and Multithreaded Processes



# User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

## Multithreading Models

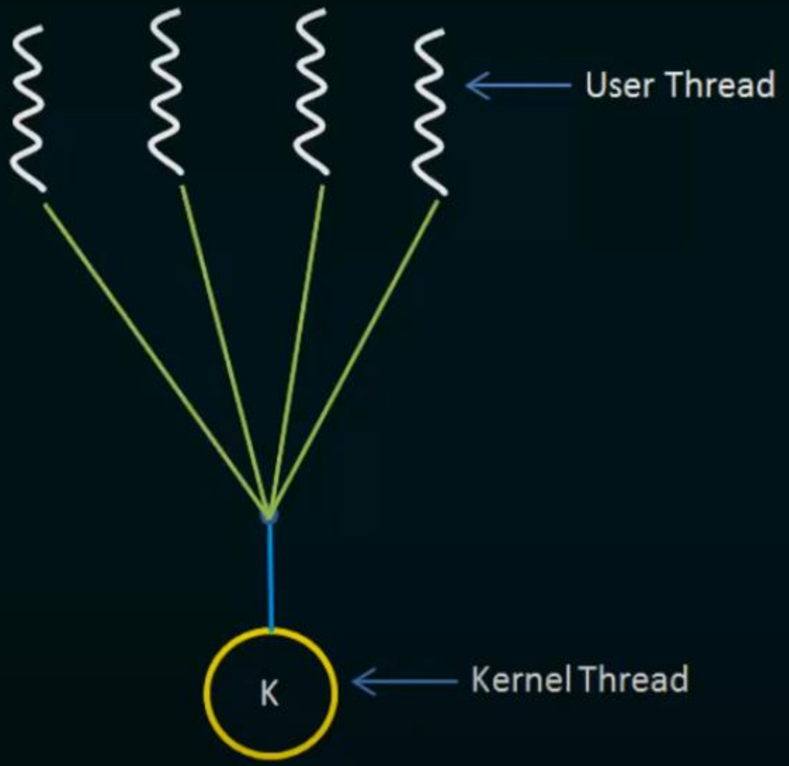
- Many-to-One
- One-to-One
- Many-to-Many

## Many-to-One Model

- Many user-level threads mapped to single kernel thread

- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

# Many-to-One Model

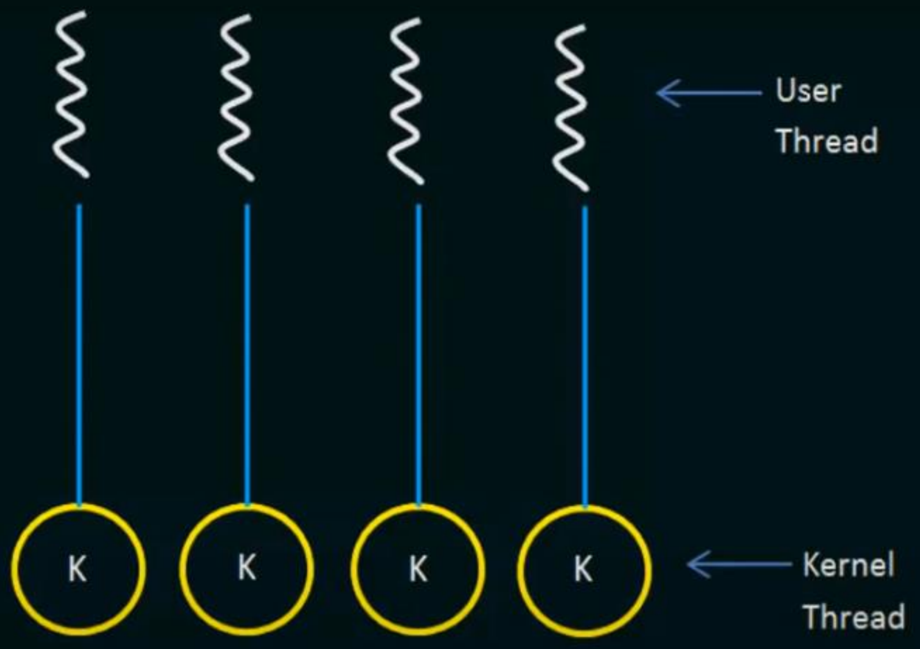


- Maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- The entire process will block if a thread makes a blocking system call.
  - Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

## One-to-one Model

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

# One-to-one Model

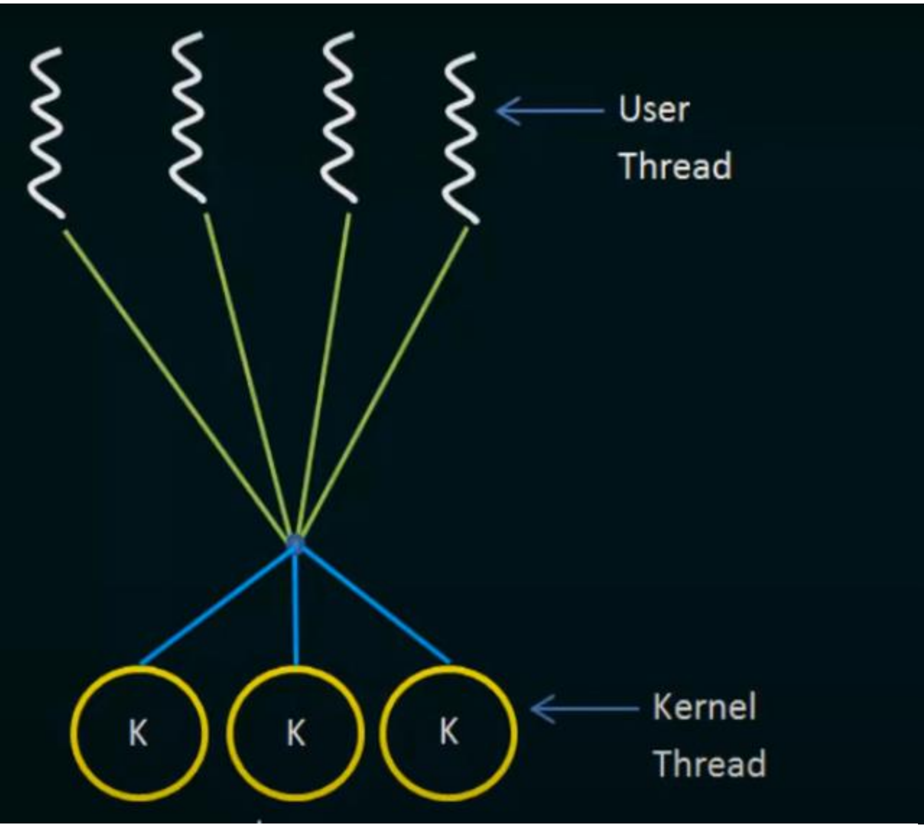


- Maps each user thread to a kernel thread.
- Provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call;
- Also allows multiple threads to run in parallel on multiprocessors.
- Creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.

# Many-to-Many Model

- 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
- Windows NT/2000 with the *ThreadFiber* package

# Many-to-Many Model

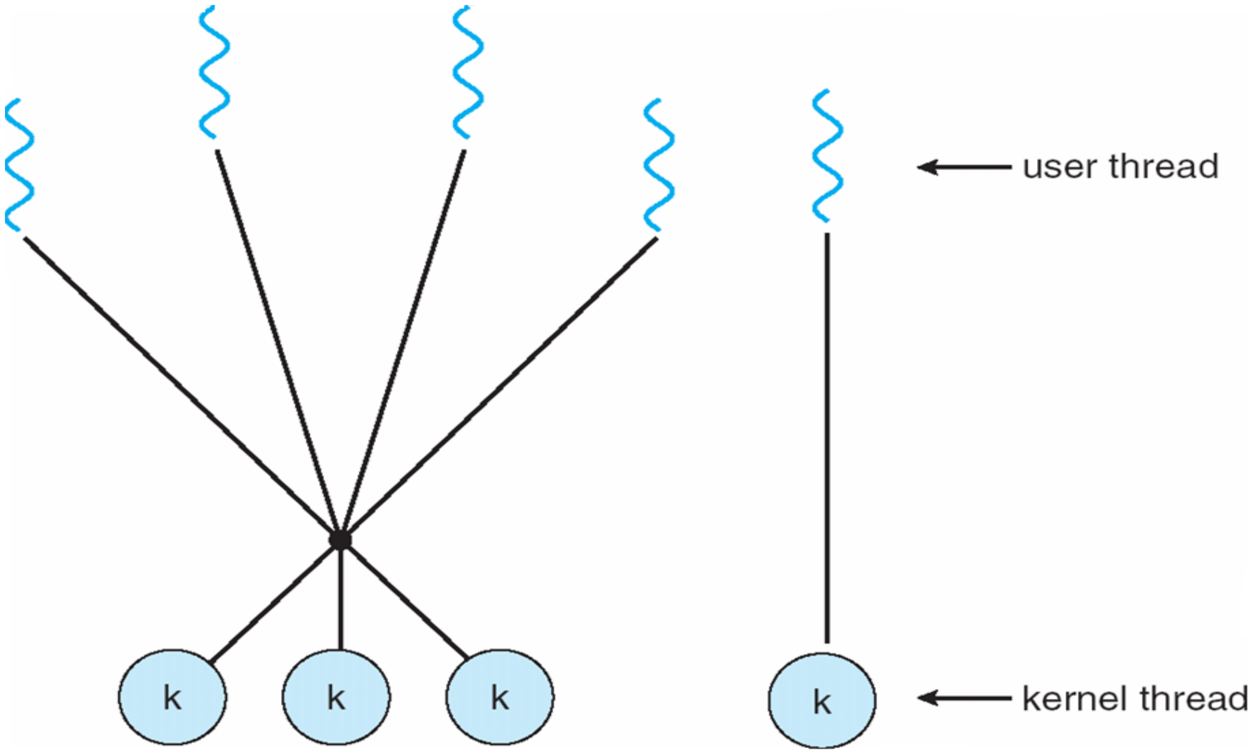


- Multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine.
- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

## Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

# Two-level Model



## Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

- Three primary thread libraries:
  - POSIX **Pthreads** (user or kernel Level)
  - Windows threads (kernel Level)
  - Java threads (implemented using a thread library available on the host system)

# Pthreads

- May be provided either as **user-level** or **kernel-level**
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X & Tru64 UNIX)

# Pthreads Example

- Here is a program that creates a new thread.
  - Hence a process will have two threads :
    - 1 - the initial/main thread that is created to execute the main() function (that thread is always created even there is no support for multithreading);
    - 2 - the **new thread**.

    (both threads have equal power)

- The program will just create a **new thread** to do a simple computation. The **new thread** will get a parameter, an integer value (as a string), and will sum all integers from 1 up to that value.
  - sum = 1+2+…+ parameter value

$$sum = \sum_{i=1}^{N} i$$

- The **main** thread will wait until sum is computed into a global variable.

- Then the **main** thread will print the result.

# Pthreads Example

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

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

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

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }
```

## Pthreads Example (Cont.)

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

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

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

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

   pthread_exit(0);
}
```

- **pthtread_attr_t attr** represents thread attr. declaration including stack size and scheduling info.
- set attr using **pthtread_attr_init(& attr** )
- **argv[1]** gives the integer val as cmd line arg (i.e. n) for summation function

# Win32 Threads

- The technique for creating threads using the Win32 thread library is similar to the Pthreads technique in several ways.
- We must include the windows.h when using the Win32 API.
- Just as in the Pthreads version data shared by the separate threads—in this case, Sum—are declared globally (the DWORD data type is an unsigned 32-bit integer.
- We also define the Summation function to be performed in a separate thread. This function is passed a pointer to a void, which Win32 defines as LPVOID.
- The thread performing this function sets the global data Sum to the value of the summation from 0 to the parameter
- passed to Summation().

# Win32 Threads

- Threads are created in the Win32 API using the CreateThread() function and—just as in Pthreads—a set of attributes for the thread is passed to this function.
- These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state.
- In this program, we use the default values for these attributes (which do not initially set the thread to a suspended state and instead make it eligible to be run by the CPU scheduler).
- Once the summation thread is created, the parent must wait for it to complete before outputting the value of Sum, asthe value is set by the summation thread. Recall that the Pthread program had the parent thread wait for the summation thread using the pthread_j oin () statement. We perform the equivalent of this in the Win32 APIusing the WaitForSingleObj ect () function, which causes the creating threadto block until the summation thread has exited.

## Windows Multithreaded C Program

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

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

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

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

# Windows Multithreaded C Program (Cont.)

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

    if (ThreadHandle != NULL) {
       /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

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

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

## Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

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

  - Extending Thread class
  - Implementing the Runnable interface

# Java Multithreaded Program

```
class Sum
{
  private int sum;

  public int getSum() {
   return sum;
  }

  public void setSum(int sum) {
   this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }

  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setSum(sum);
  }
}
```

# Java Multithreaded Program (Cont.)

```
public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
     if (Integer.parseInt(args[0]) < 0)
      System.err.println(args[0] + " must be >= 0.");
     else {
      Sum sumObject = new Sum();
      int upper = Integer.parseInt(args[0]);
      Thread thrd = new Thread(new Summation(upper, sumObject));
      thrd.start();
      try {
         thrd.join();
         System.out.println
                 ("The sum of "+upper+" is "+sumObject.getSum());
     } catch (InterruptedException ie) { }
     }
    }
    else
     System.err.println("Usage: Summation <integer value>"); }
}
```

## Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  – Synchronous and asynchronous
- Thread cancellation of target thread
  – Asynchronous or deferred
- Thread pools
- Thread- specific data
- Scheduler Activations

# fork() and exec() System Calls

- fork()- used to create separate, duplicate process.

- exec()- when a exec() system call is invoked, the program specified in the parameter to exec() will replace the entire process – including all threads.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
Int main() {
        printf(“Good Morning \n PID= %d \n”, getpid() );
        return 0;
}
```

```
Output:
Good Morning
PID= 7890
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
Int main() {
        fork();
        printf("Good Morning \n PID= %d \n", getpid() );
        return 0;
}
```

```
Output:
Good Morning
PID= 7890
Good Morning
PID= 7891
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
Int main() {
        fork();
        fork();
        fork();
        printf("Good Morning \n PID= %d \n", getpid() );
        return 0;
}
```

Output:

```
Good Morning          Good Morning
PID= 7890             PID= 7898
Good Morning          Good Morning
PID= 7893             PID= 7892
Good Morning          Good Morning
PID= 7896             PID= 7897
Good Morning          Good Morning
PID= 7891             PID= 7899
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
Int main(int argc, char *argv[ ]) {
        printf(“PID of prg1.c= %d \n”, getpid() );
        char *args[ ]={‘good’, ‘ morning’, ‘all’, NULL);
        exec (“ ./prg2.c”, args);
        printf( “Back to prg1.c”);
        return 0;
}
```

prg1.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
Int main(int argc, char *argv[ ]) {
        printf(“ We are in prg2.c \n”);
        printf(“PID of prg2.c= %d \n”, getpid() );
        return 0;
}
```

prg2.c

## Commands to execute the files

```
gcc prg1.c –o prg1
gcc prg2.c –o prg2
prg1
```

Output:

```
PID of prg1.c= 4567
We are in prg2.c
PID of prg2.c= 4567
```

## Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`**
  - The program specified in the parameter to exec() will replace the entire process including all threads

## But which version of fork() to use and when?

- It depends on the application
  - If exec() is called immediately after forking
    - Then duplicating all threads is unnecessary. As the program specified in the parameters to exec() will replace the process.
    - In this case, duplicating only the calling thread is appropriate.
  - If the separate process does not call exec() after forking.
    - Then the separate process should duplicate all threads.

# Thread Cancellation

- **Thread Cancellation** is the task of **terminating** a thread before it has finished.
- For ex, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
- Another ex, when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page is loaded using several threads—each image is loaded in a separate thread. When a user presses the *stop button on the browser, all threads loading the page are* canceled.
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

# Difficulty in Thread Cancellation

- The difficulty with cancellation occurs in situations where **resources have been allocated to a canceled thread** or where a thread is canceled while in **the midst of updating data it is sharing with other threads**.
- This becomes especially troublesome with asynchronous cancellation.
- Often, the OS will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread synchronously may not free a necessary system-wide resource.

- **With deferred cancellation**, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine if it should be canceled or not.
- This allows a thread to check whether it should be canceled at a point when it can be canceled safely.
- **Pthreads** refers to such points as **cancellation points.**

## Signal Handling

- A signal is used in UNIX systems to notify a process that a particular event has occurred.
- All signals follow the same pattern:
  - A signal is generated by the occurrence of a particular event.
  - A generated signal is delivered to a process.
  - Once delivered, the signal must be handled.

- **Synchronous** signals are delivered to the same process that performed the operation that caused the signal.
  - Examples
    - illegal memory access
    - division by zero
- When a signal is generated by an event external to a running process, that process receives the signal **asynchronously.**
  - Examples
    - terminating a process with specific keystrokes (ctrl +C)

- Every signal may be handled by one of two possible handlers:
  - 1. A default signal handler
  - 2. A user-defined signal handler
- Every signal has a **default signal handler** that is run by the **kernel** when handling that signal.
- This default action can be overridden by a **user-defined signal handler** that is called to handle the signal.

# Handling signals in multi-threaded programs

- Delivering signals is more complicated in multithreaded programs, where a process may have several threads. **Where should a signal be delivered?**
- In general, the following options exist:
  - Deliver the signal to the thread to which the signal applies.
  - Deliver the signal to every thread in the process.
  - Deliver the signal to certain threads in the process.
  - Assign a specific thread to receive all signals for the process

## Thread Pool

Why?

## Why Thread Pools

- Whenever **multithreading web server** receives a request, it creates a separate thread to service the request. Whereas creating a separate thread is certainly superior to creating a separate process.
- But multithreaded server has potential problems.
- The **first** concerns the **amount of time required** to **create** the thread prior to servicing the request and also **discarding** of thread once it has completed its work.
- The **second** issue is: If we allow all concurrent requests to be serviced in a new thread, we have not placed a **bound on the number of threads** concurrently **active** in the system.
- Unlimited threads could exhaust system resources, such as CPU time or memory.
- One solution to this issue is to use a **thread pool.**

## Thread pool concept

- The general concept of a thread pool is to create a number of threads at **process startup** and place them into a *pool, where they sit and wait for work.*
- When a server receives a request, it awakens a thread from this pool—if one is available—and passes it the request to service. Once the thread completes its service, it returns to the pool and awaits more work.
- If the pool contains no available thread, the server waits until one becomes free.

## Advantages of Thread Pools

- Usually slightly **faster** to service a request with an existing thread than create a new thread
- A thread pool limits the **number of threads** in the application(s) to be bound to the size of the pool. More suitable for the system that can not support large number of concurrent threads.

• Windows API supports thread pools:

- A function that is to run as a separate thread is defied as below

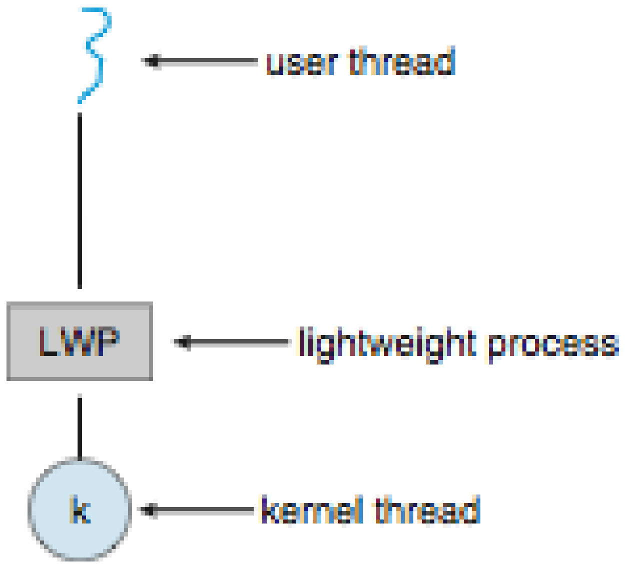
```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
     * this function runs as a separate thread.
    */
}
```

- A pointer to PoolFunction() is passed to one of the functions in the thread pool API, and a thread from the pool executes this function.
- One such member in thread pool API is : QueueUserWorkItem() , which is passed with 3 parameters:
  - **LPTHREAD_START-ROUTINE** Function—a pointer to the function that is to run as a separate thread
  - **PVOID Param**—the parameter passed to Function
  - **ULONG Flags**—flags indicating how the thread pool is to create and manage execution of the thread
- An example of an invocation is:

  **QueueUserWorkItem(&PoolFunction, NULL, 0);**
- This causes a thread from the thread pool to invoke **PoolFunction()** on behalf of the programmer.
- In this instance, we pass no parameters to PoolFunction (). Because we specify 0 as a flag, we provide the thread pool with no special instructions for thread creation.

# Scheduler Activations

- Final issue with multi-threaded programs concerns **communication between the kernel and the thread library**.
- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.
- Typically use an **intermediate data structure** between user and kernel threads: **lightweight process** (**LWP**)
  - Appears to be a **virtual processor** on which a process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads



End of Chapter 4