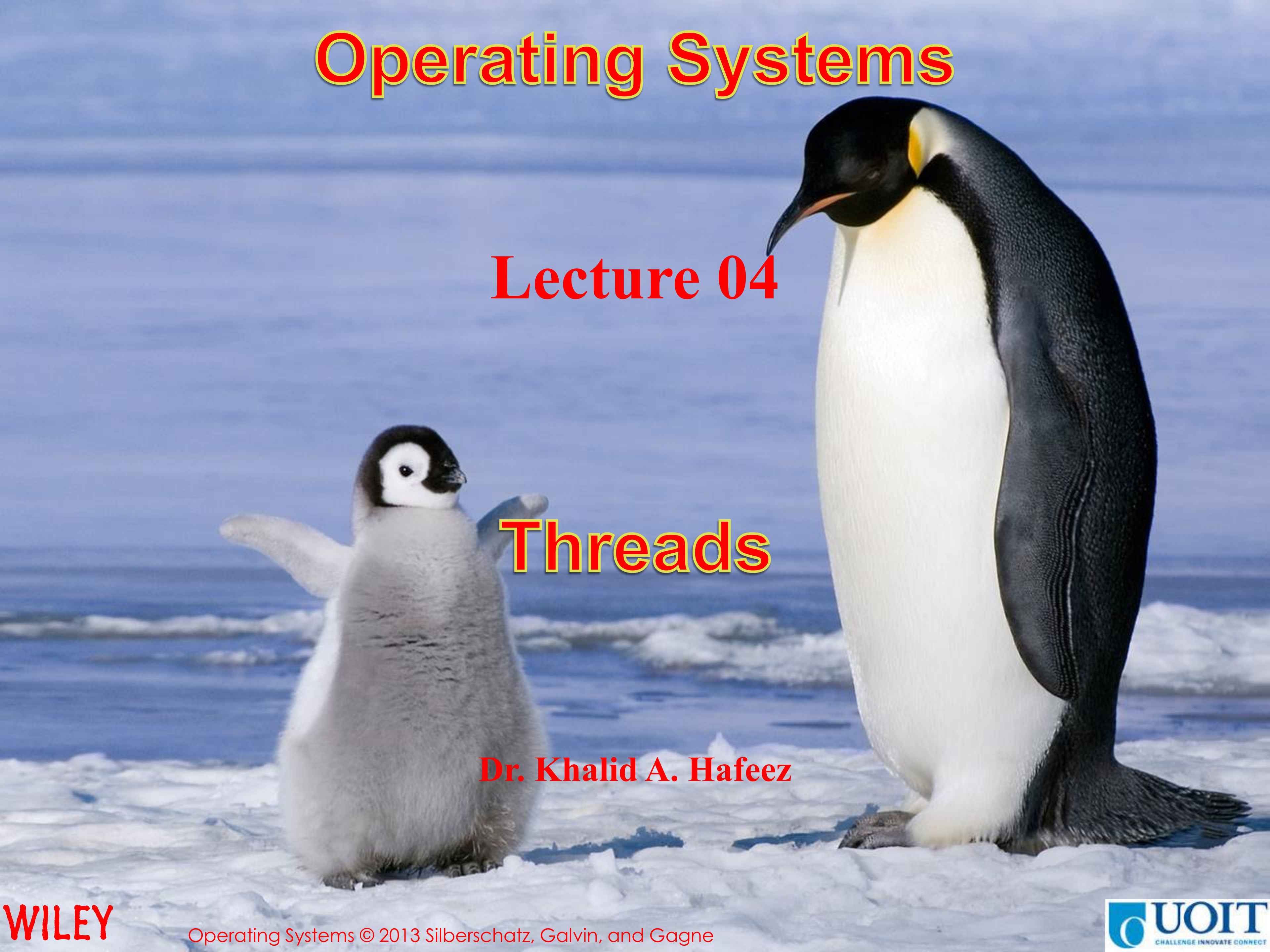
Operating Systems



Lecture 04

Threads

Dr. Khalid A. Hafeez

Threads

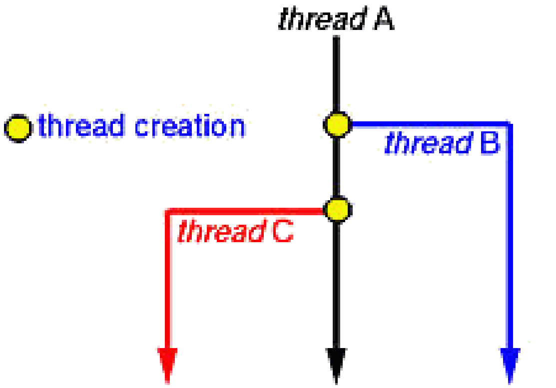
- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

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

Overview

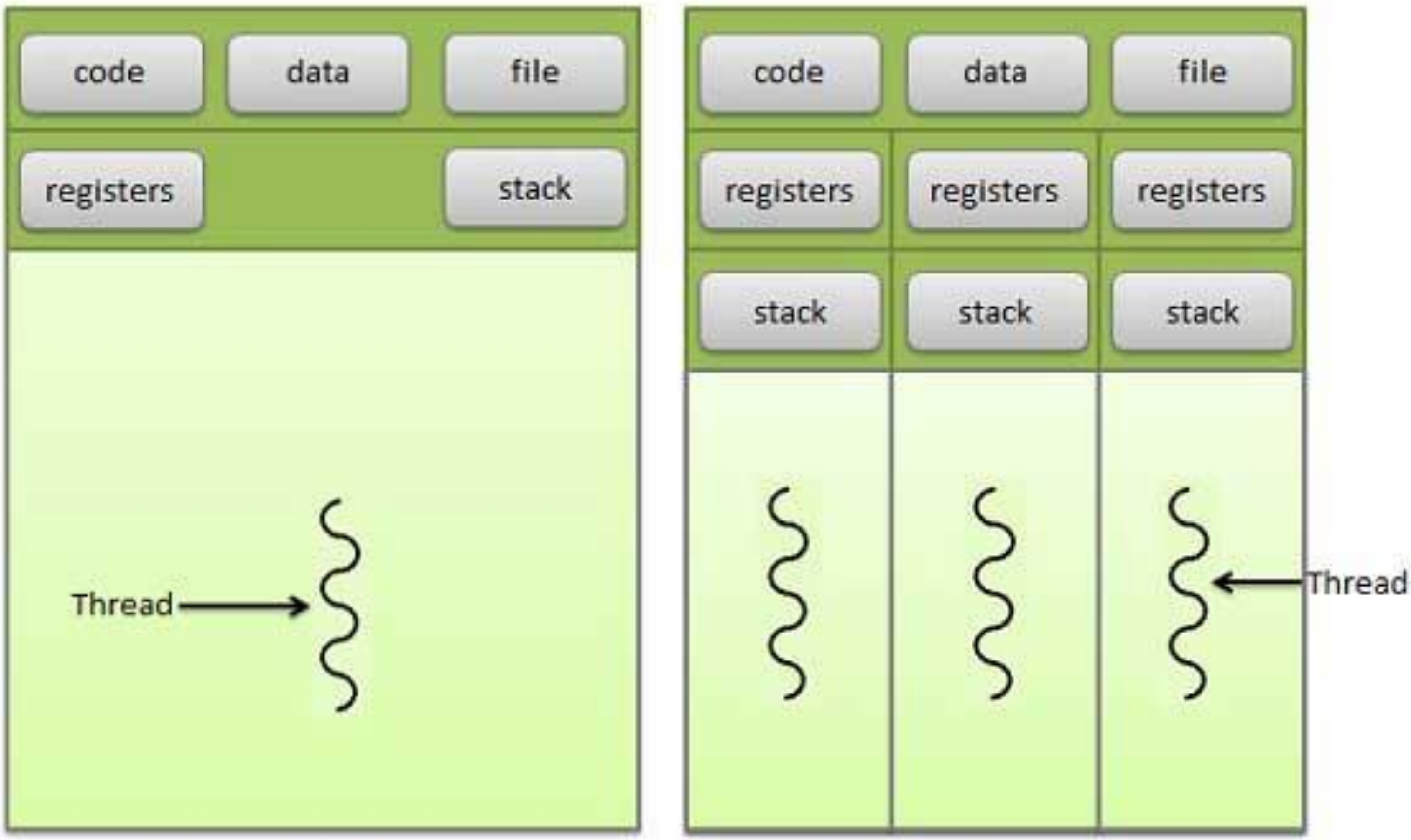
- A **thread** is a basic unit of CPU utilization;
  - **It consists of:**
    - Thread ID,
    - Program counter,
    - Register set,
    - Stack.



  - **It shares with other threads belonging to the same process**
    - Its code section,
    - Its data section,
    - Other operating-system resources, such as open files and signals

Single and Multithreaded Processes

- If a process has multiple threads of control, it can perform more than one task at a time.

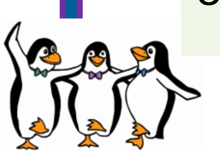


Single threaded Process

Multi-threaded Process

Difference between Process and Thread

Process	Thread
Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

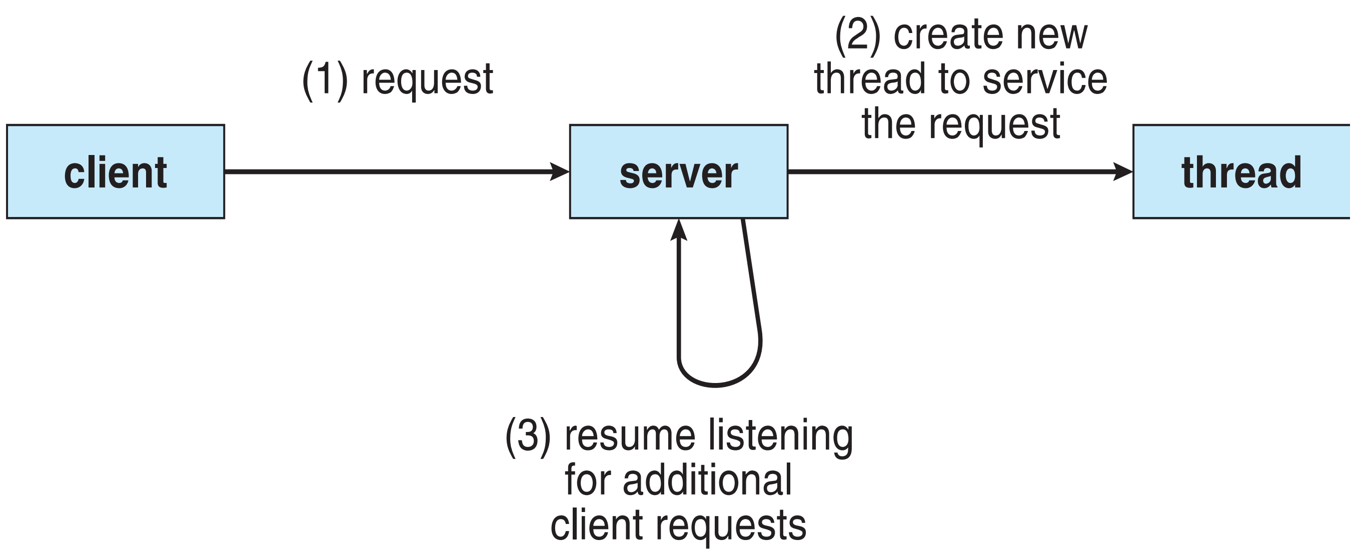


Motivation

- Most modern applications are typically multithreaded
  - An application is implemented as a separate process with several threads of control
  - Example: Multiple tasks within a word processor 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
  - Threads can simplify code, and increase efficiency
- Kernels are generally multithreaded
  - A thread for every task, such as managing devices, managing memory, or interrupt handling

Multithreaded Server Architecture

- A single application may be required to perform several similar tasks from different clients requests
- Example: Web Server
  - If the web-server process is multithreaded, the server will create a separate thread that listens for client requests.
  - When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.



Benefits

- The benefits of multithreaded programming can be broken down into four major categories:
  - Responsiveness: may allow continued execution if part of process is blocked, especially important for user interfaces
    - If a user clicks a button that results in the performance of a time-consuming operation: if it is single-threaded, then it will be unresponsive for long time
  - Resource Sharing: threads share resources of the process to which they belong, and it is much easier than shared memory or message passing between processes
  - Economy: cheaper than process creation, thread switching lower overhead than context switching
    - In Solaris, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.
  - Scalability: multithreading process can take advantage of multiprocessor architectures
    - Threads may run in parallel on different processing cores.

Multicore Programming

- Multithreaded programming provides a mechanism for more efficient use of the multiple computing cores and improved concurrency
- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
  - Dividing activities into separate, concurrent tasks.
  - Balance: ensure that the tasks perform equal work of equal value
  - Data splitting
  - Data dependency
  - Testing and debugging

- **Difference between parallelism and concurrency:**
  - **Parallelism** implies a system can perform more than one task simultaneously (needs multicore or multiprocessor systems)
  - **Concurrency** supports more than one task to make progress through time sharing
    - The system has a single processor or core, and the CPU schedulers are designed to provide the illusion of parallelism by rapidly switching between processes

Multicore Programming

- Concurrency vs. Parallelism
  - Concurrent execution on single-core system:
    - Concurrency means here: the execution of the threads will be interleaved over time

single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	...

time →

  - Parallelism on a multi-core system:
    - Concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core

core 1	T_1	T_3	T_1	T_3	T_1	...
core 2	T_2	T_4	T_2	T_4	T_2	...

time →

Multicore Programming

- Amdahl's Law
  - Identifies performance gains from adding additional cores to an application that has both serial and parallel components
  - S is the serial portion
  - N number of processing cores

$$speedup \le \frac{1}{S + \frac{(1-S)}{N}}$$

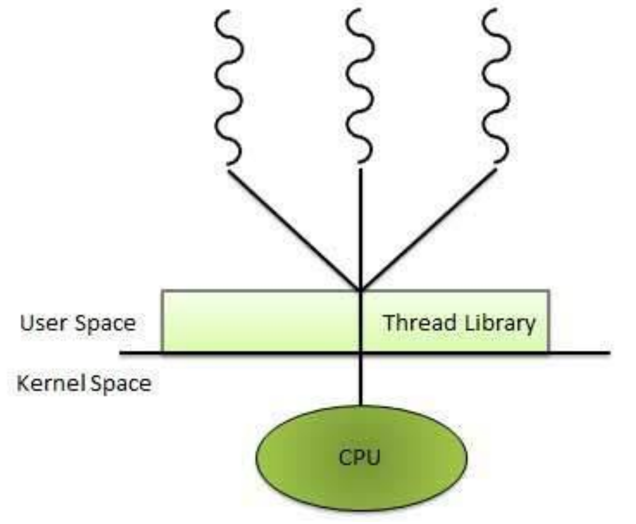
  - That is, if application is 75% parallel and 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
  - As N approaches infinity, speedup approaches 1 / S
  - Serial portion of an application has disproportionate effect on performance gained by adding additional cores
  - But does the law take into account contemporary multicore systems?

Multicore Programming

- Types of parallelism
  - Data parallelism: distributes subsets of the same data across multiple cores, and performing the same operation on each core
    - Example: summing the contents of an array of size N.
      - Thread A (on core 0): sum the elements [0] . . . [N/2 − 1]
      - Thread B (on core 1): sum the elements [N/2] . . . [N − 1]
  - Task parallelism: distributing threads across cores, each thread performing unique operation on the same data or on different data
    - Example: perform two different operation on the array of size N
      - Thread A (on core 0): sum the elements the array
      - Thread B (on core 1): find the max of the elements in the array
  - In practice, few applications strictly follow either data or task parallelism
    - They are hybrid
  - As the number of threads grows, so does architectural support for threading
    - CPUs have cores as well as hardware threads
      - Oracle SPARC T4 has 8 cores, and 8 hardware threads per core

Multithreading Models

- User Threads and Kernel Threads
  - The support for threads may be provided either at the user level, or at the kernel level.
    - **User threads** **(**User managed threads): management done by user-level threads library
      - Three primary thread libraries:
        - POSIX Pthreads
        - Windows threads
        - Java threads



    - **Kernel threads** **(**OS managed threads): Supported by the Kernel
      - Examples: virtually all general purpose operating systems, including:
        - Windows, Solaris, Linux, Tru64 UNIX, Mac OS X

Multithreading Models

- User Threads and Kernel Threads
  - User Level Threads
    - The application manages thread management and the kernel is not aware of the existence of threads.
    - The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts.
    - The application begins with a single thread and begins running in that thread.
    - **Advantages**
      - Thread switching does not require Kernel mode privileges.
      - User level thread can run on any operating system.
      - Scheduling can be application specific in the user level thread.
      - User level threads are fast to create and manage.
    - **Disadvantages**
      - In a typical OS, most system calls are blocking.
      - Multithreaded application cannot take advantage of multiprocessing.

Multithreading Models

- User Threads and Kernel Threads
  - Kernel Level Threads
    - Thread management done by the Kernel only.
    - Kernel threads are supported directly by the OS.
    - All threads of an application are supported within a single process.
    - The Kernel maintains context information for the process as a whole and for individuals threads within the process.
    - Scheduling by the Kernel is done on a thread basis.
    - Kernel do thread creation, scheduling and management in its space.
    - **Advantages**
      - Kernel can simultaneously schedule multiple threads from the same process on multiple processors.
      - If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
      - Kernel routines themselves can multithreaded.
    - **Disadvantages**
      - They are slower to create and manage than the user threads.
      - Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

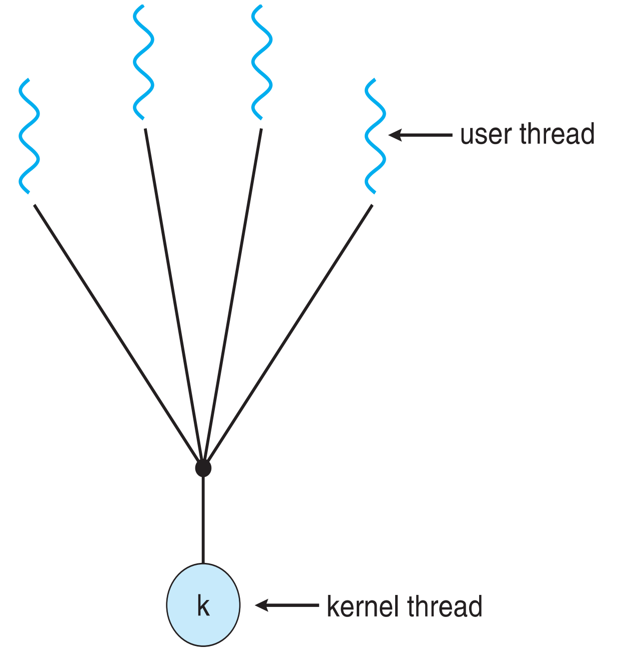
Multithreading Models

- Some OS provide a combined user level thread and Kernel level thread facility.
  - Example: Solaris
  - Multiple threads within the same application can run in parallel on multiple processors
  - A relationship must exist between user threads and kernel threads.
    - **Three common models for establishing this relationship:**
      - Many-to-One
      - One-to-One
      - Many-to-Many

Multithreading Models

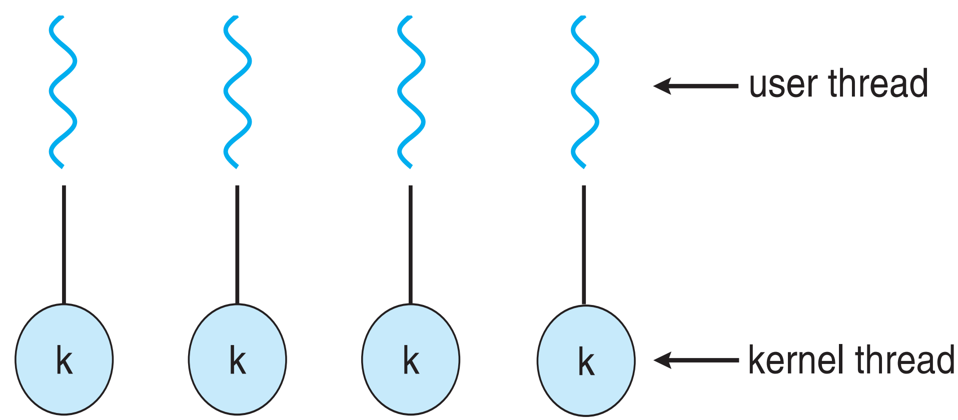
Many-to-One model

- Many user-level threads mapped to a single kernel thread
- One thread blocking causes all to block
- Multiple threads will not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



Multithreading Models

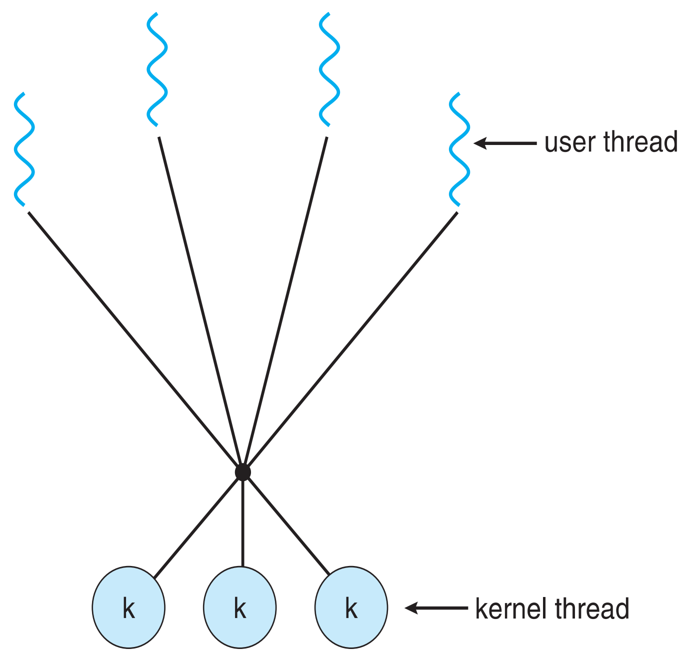
- One-to-One
  - Each user-level thread maps to one kernel thread
  - Creating a user-level thread needs creating a kernel thread
  - More concurrency than many-to-one
  - It support multiple thread to execute in parallel on microprocessors.
  - Number of threads per process sometimes restricted due to overhead of creating kernel threads
  - Examples
    - Windows
    - Linux
    - Solaris 9 and later



Multithreading Models

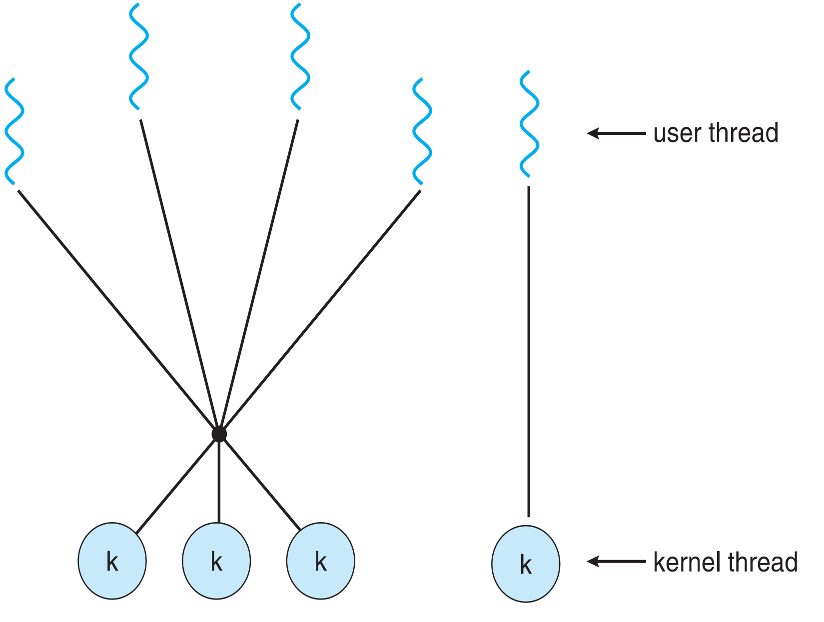
Many-to-Many Model

- Allows many user level threads to be multiplixed to a smaller or equal kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.
- Example:
  - Solaris prior to version 9
  - Windows with the ThreadFiber package



Multithreading Models

- Two-level Model
  - Similar to Many-to-Many, except that it allows a user thread to be bound to a kernel thread
  - Examples
    - IRIX
    - HP-UX
    - Tru64 UNIX
    - Solaris 8 and earlier



Thread Libraries

- Thread library provides programmer with an API for creating and managing threads
- **There are two primary ways of implementing a thread library:**
  - Provide a library entirely in user space with no kernel support
    - All code and data structures for the library exist in user space.
    - Therefore, invoking a function in the library results in a local function call in user space and not a system call
  - Implement a Kernel-level library supported directly by the OS
    - The code and data structures for the library exist in kernel space.
    - Therefore, Invoking a function in the API for the library typically results in a system call to the kernel.

Thread Libraries

- **There are three main thread libraries:**
  - POSIX Pthreads,
    - May be provided as either a user-level or a kernel-level library
  - Windows threads
    - It is a kernel-level library
  - Java threads
    - The Java thread API allows threads to be created and managed directly in Java programs
    - Because the JVM runs on top of a host operating system, the Java thread API is implemented using a thread library available on the host system.

  - For POSIX and Windows, any data declared globally (declared outside of any function) are shared among all threads belonging to the same process.
  - Because Java has no notion of global data, access to shared data must be explicitly arranged between threads.
  - Data declared local to a function are typically stored on the stack. Since each thread has its own stack, each thread has its own copy of local data.

Thread Libraries

- **There are two general strategies for creating multiple threads:**
  - Asynchronous threading,
    - Once the parent creates a child thread, the parent resumes its execution, and the parent thread need not know when its child terminates
  - Synchronous threading
    - Occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes
    - It is called fork-join strategy.

Thread Libraries

- Pthreads
  - May be provided either as user-level or kernel-level
  - A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
    - It is a specification for thread behavior not an implementation
  - Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Thread Libraries

- Pthreads example:
  - The C program shown demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a nonnegative integer in a separate thread.

When this program begins, a single thread of control begins in main()

```
#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;
   }
```

Thread Libraries

- Pthreads example: continue…

The main() creates a second thread that begins control in the runner() function

after creating the summation thread, the parent thread will wait for it to terminate by calling the pthread-join() function

```
   /* 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);
}
```

Thread Libraries

- **Pthreads example:**

Code for Joining 10 Threads

A simple method for waiting on several threads using the pthread_join() function is to enclose the operation within a simple for loop.

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Thread Libraries

- Windows Threads:

  Multithreaded C Program using the Windows API

- Threads are created in the Windows API using the CreateThread() function.
- Windows API use the WaitForSingleObject() function to block the main() until the summation thread has exited.
- If require waiting for multiple threads to complete, the WaitForMultipleObjects() function can be used.

```
#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;
   }
```

Thread Libraries

- Windows Threads:

  Multithreaded C Program using the Windows API

```
    /* 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);
    }
}
```

Thread Libraries

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
  - Extending the Thread class and to override its run() method.
  - Define a class that implements the Runnable interface
    - it must define a run() method
      - The code implementing the run() method is what runs as a separate thread

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

Thread Libraries

Java Threads

- Java program for the summation of a non-negative integer
- The Summation class implements the Runnable interface.
- Thread creation is performed by creating an object instance of the Thread class and passing the constructor a Runnable object.
- Calling the start() method for the new object does two things:
  - It allocates memory and initializes a new thread in the JVM.
  - It calls the run() method, making the thread eligible to be run by the JVM.

```java
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);
  }
}
```

Thread Libraries

Java Threads

- When the summation program runs, the JVM creates two threads.
  - Parent thread, which starts execution in the main() method.
  - Child thread is created when the start() method on the Thread object is invoked.
    - It begins execution in the run() method of the Summation class.
    - This thread terminates when it exits from its run() method.

```
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>"); }
}
```

Implicit Threading

- With the growth of multicore processing, applications containing hundreds or even thousands of threads are possible.
- Designing such applications is difficult
  - One solution is to transfer creation and management of threads to compilers and run-time libraries rather than programmers
  - It is called implicit threading
  - **Three methods explored**
    - Thread Pools
    - OpenMP
    - Grand Central Dispatch
  - Other methods include Microsoft Threading Building Blocks (TBB), java.util.concurrent package

Implicit Threading

- Thread Pools
  - Create a number of threads at process startup and place them into a pool.
  - When a server receives a request, it awakens a thread from this pool and passes it the request for service.
  - Once the thread completes its service, it returns to the pool and awaits more work.
  - If the pool is embty, the server waits until one becomes free.
  - Advantages:
    - It is faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound to the size of the pool
    - Separating task to be performed from mechanics of creating task allows different strategies for running task
      - i.e.Tasks could be scheduled to run periodically
  - Windows API supports thread pools:

Implicit Threading

- OpenMP
  - Set of compiler directives and an API for C, C++, FORTRAN
  - Provides support for parallel programming in shared-memory environments
  - Identifies parallel regions – blocks of code that can run in parallel

    #pragma omp parallel
  - Create as many threads as there are cores

    #pragma omp parallel for for(i=0;i<N;i++) {

        c[i] = a[i] + b[i];

    }
  - Run for loop in parallel

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

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

Implicit Threading

- Grand Central Dispatch
  - Apple technology for Mac OS X and iOS operating systems
  - Extensions to C, C++ languages, API, and run-time library
  - Allows identification of parallel sections
  - Manages most of the details of threading
  - Block is in “^{ }” -   ˆ{ printf("I am a block"); }
  - Blocks placed in dispatch queue
    - Assigned to available thread in thread pool when removed from queue

Implicit Threading

- Grand Central Dispatch
  - Two types of dispatch queues:
    - serial – blocks removed in FIFO order, queue is per process, called main queue
      - Programmers can create additional serial queues within program
    - concurrent – removed in FIFO order but several may be removed at a time
      - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Threading Issues

- Some of the issues to consider in designing multithreaded programs:
  - Semantics of fork() and exec() system calls
  - Signal handling
    - Synchronous and asynchronous
  - Thread cancellation of target thread
    - Asynchronous or deferred
  - Thread-local storage
  - Scheduler Activations

Threading Issues

- Semantics of fork() and exec()
  - If one thread in a program calls fork(), does fork() duplicate only the calling thread or all threads?
    - Some UNIXes have two versions of fork
  - exec() usually works as normal – replace the running process including all threads

Threading Issues

- Signal Handling
  - Signals are used in UNIX systems to notify a process that a particular event has occurred.
  - A signal handler is used to process signals
    - Signal is generated by particular event
    - Signal is delivered to a process
    - Signal is handled by one of two signal handlers:
      - default
      - user-defined
  - Every signal has default handler that kernel runs when handling signal
    - User-defined signal handler can override default
    - For single-threaded, signal delivered to process

Threading Issues

- Signal Handling
  - Where should a signal be delivered for multi-threaded?
    - 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

Threading Issues

- Thread Cancellation
  - Terminating a thread before it has finished
  - 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
  - Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Threading Issues

Thread Cancellation

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

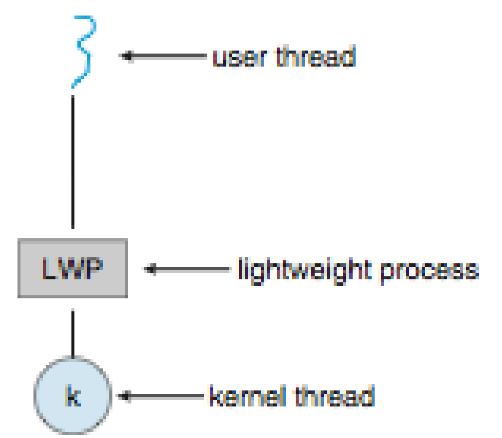
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    - I.e. pthread_testcancel()
    - Then cleanup handler is invoked
- On Linux systems, thread cancellation is handled through signals

Threading Issues

- Thread-Local Storage
  - Thread-local storage (TLS) allows each thread to have its own copy of data
  - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
  - Different from local variables
    - Local variables visible only during single function invocation
    - TLS visible across function invocations
  - Similar to static data
    - TLS is unique to each thread

Threading Issues

- Scheduler Activations
  - 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 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



Operating System Examples

- Windows Threads
- Linux Threads

Operating System Examples

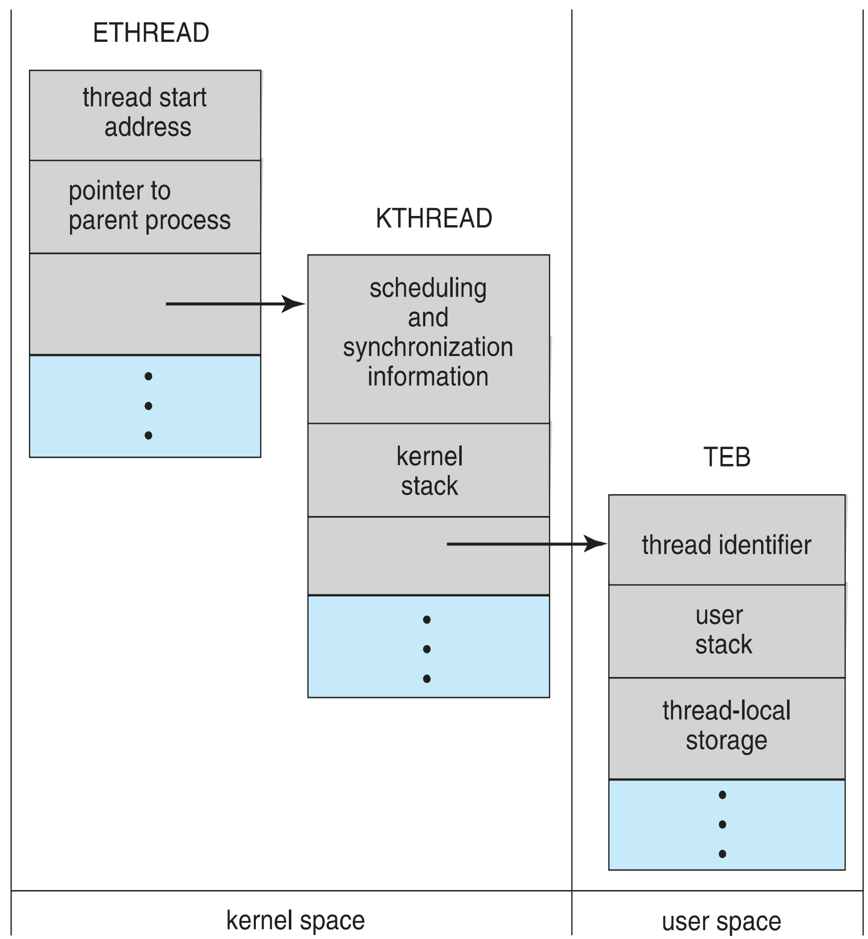
- Windows Threads
  - Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
  - Implements the one-to-one mapping, kernel-level
  - Each thread contains
    - A thread id
    - Register set representing state of processor
    - Separate user and kernel stacks for when thread runs in user mode or kernel mode
    - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
  - The register set, stacks, and private storage area are known as the context of the thread

Operating System Examples

- Windows Threads
  - The primary data structures of a thread include:
    - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
    - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
    - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Operating System Examples

- Windows Threads Data Structures



Operating System Examples

Linux Threads

- Linux refers to them as tasks rather than threads
- Thread creation is done through clone() system call
- clone() allows a child task to share the address space of the parent task (process)
  - Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- struct task_struct points to process data structures (shared or unique)