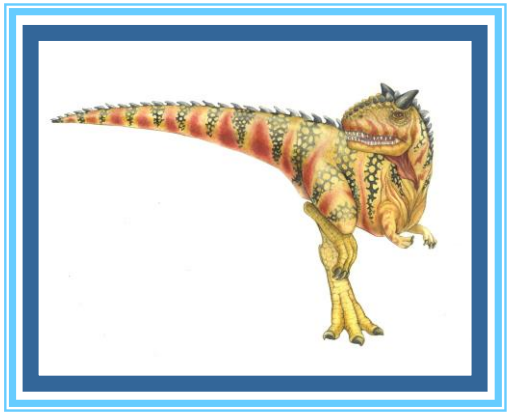
Module 2.2: Threads



Chapter 4: Threads

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

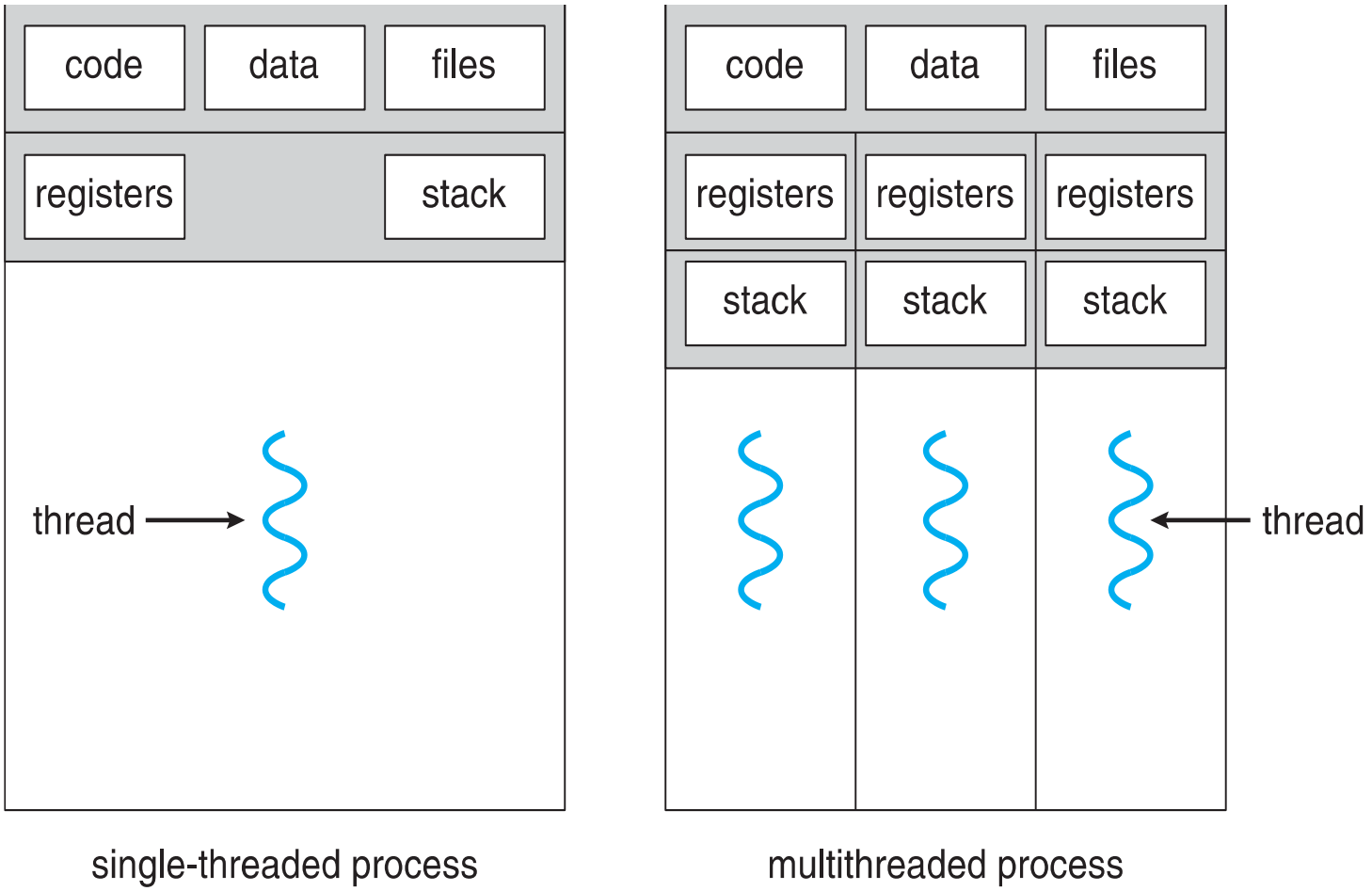
Objectives

- 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, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)
- 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

Single and Multithreaded Processes



Problem with single threaded process?
Advantages of Multi threaded process?

Process vs Threads

Process	Thread
A process is an instance of a program that is being executed or processed.	Thread is a segment of a process or a lightweight process that is managed by the scheduler independently.
Processes are independent of each other and hence don't share a memory or other resources.	Threads are interdependent and share memory.
Each process is treated as a new process by the operating system.	The operating system takes all the user-level threads as a single process.
If one process gets blocked by the operating system, then the other process can continue the execution.	If any user-level thread gets blocked, all of its peer threads also get blocked because OS takes all of them as a single process.

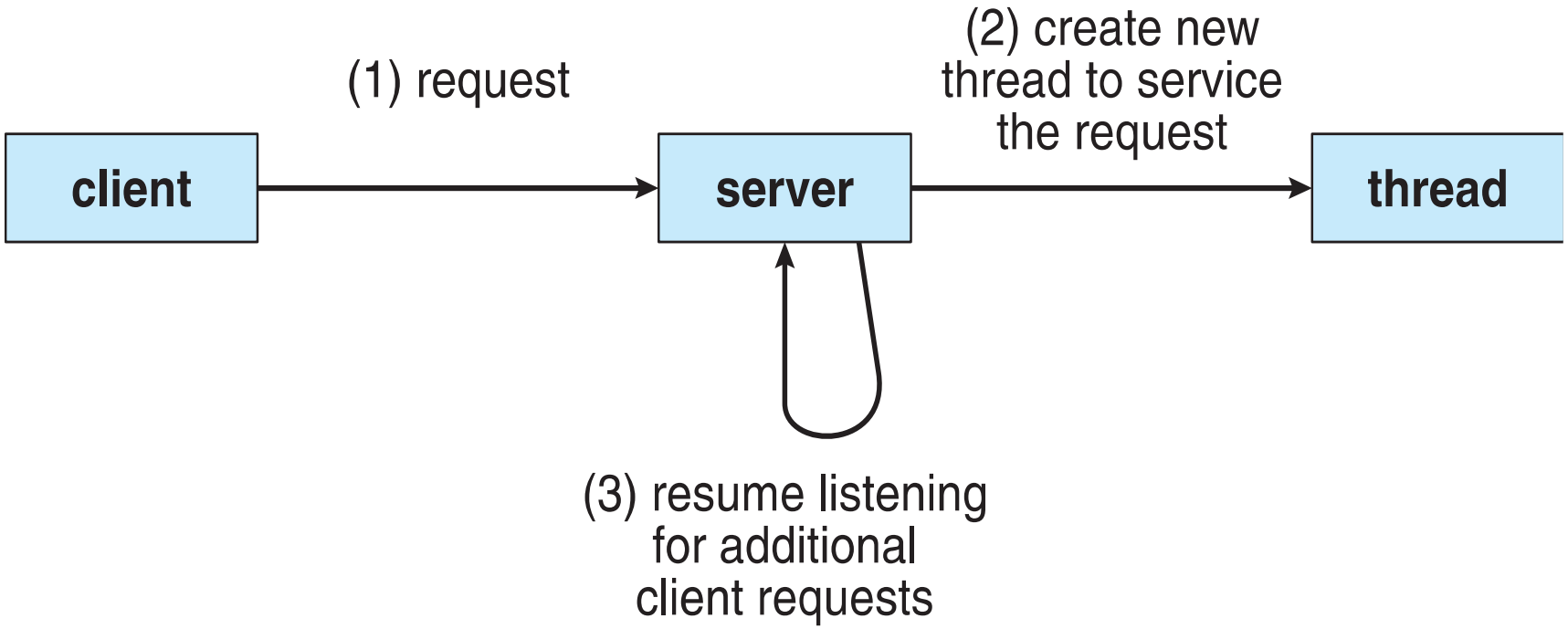
Process vs Threads

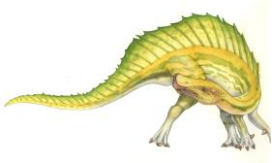
Process	Threads
Context switching between two processes takes much time as they are heavy compared to thread.	Context switching between the threads is fast because they are very lightweight.
The data segment and code segment of each process are independent of the other.	Threads share data segment and code segment with their peer threads; hence are the same for other threads also.
The operating system takes more time to terminate a process.	Threads can be terminated in very little time.
New process creation is more time taking as each new process takes all the resources.	A thread needs less time for creation.

Benefits

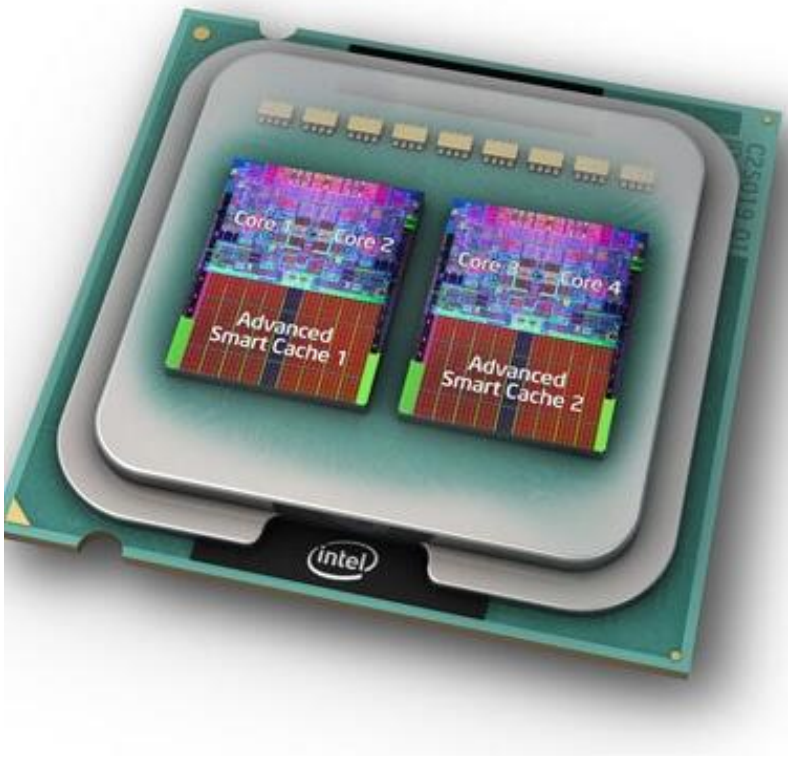
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower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

Multithreaded Server Architecture





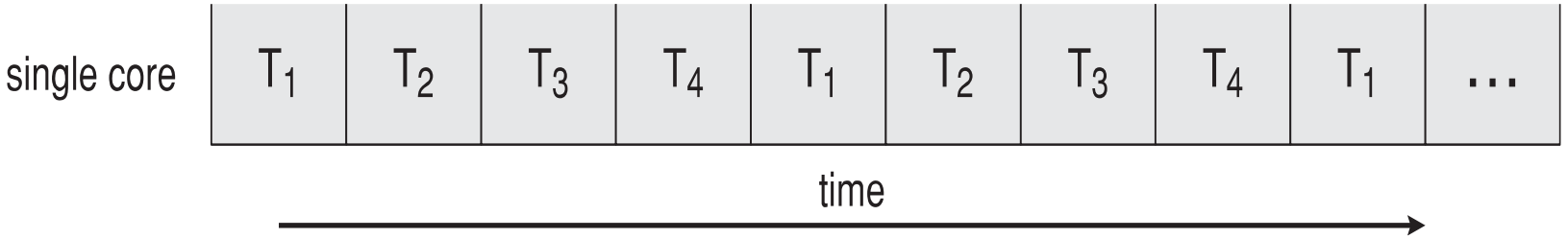
Multicore Programming



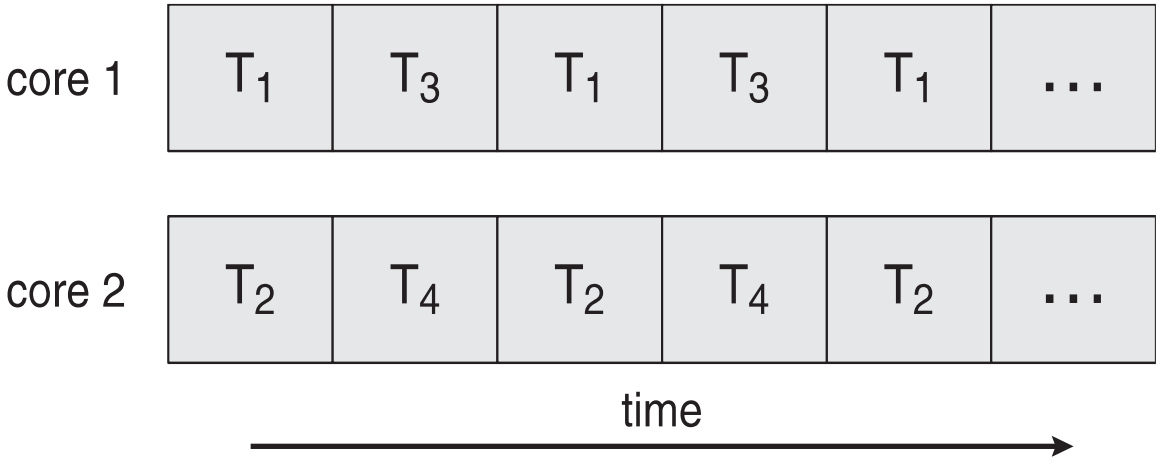
Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency

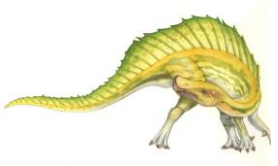
Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

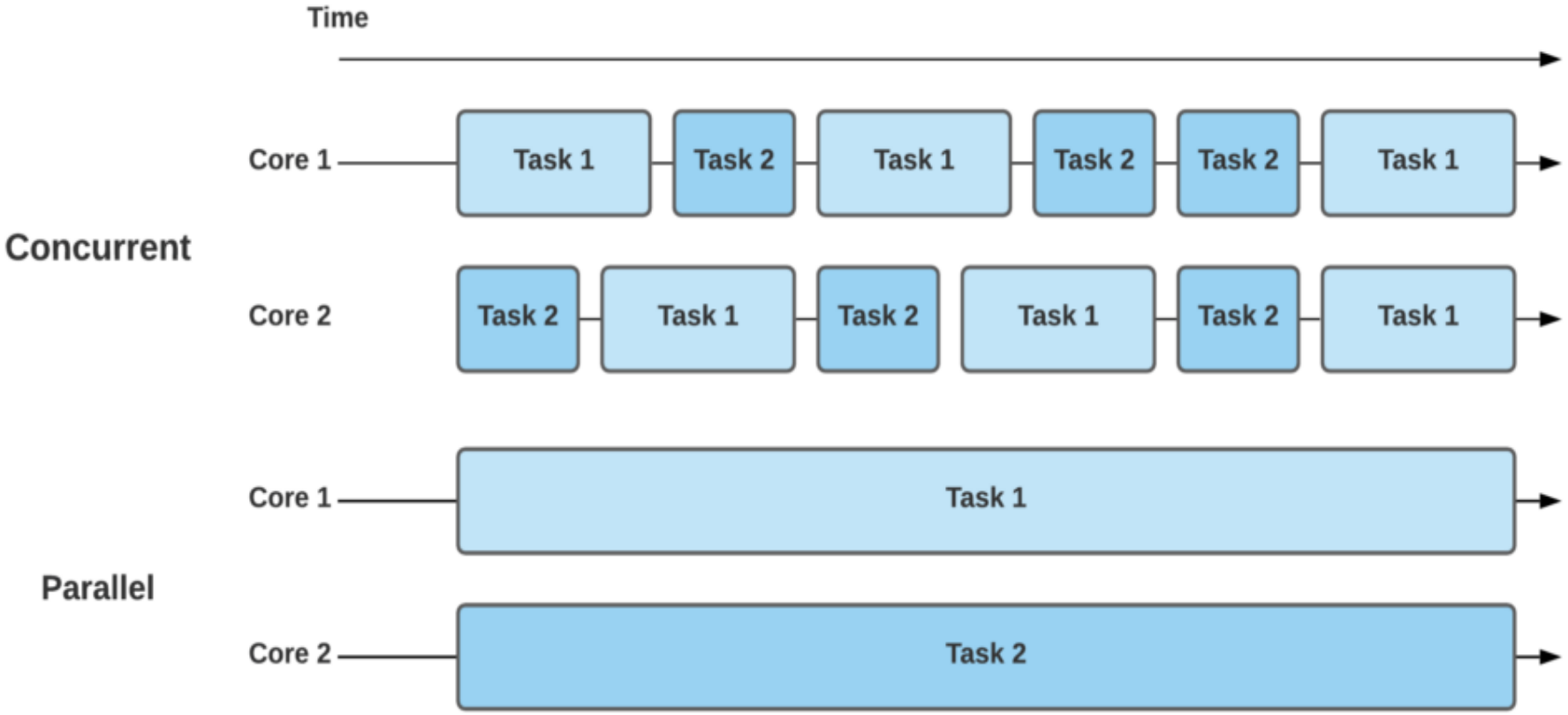


- **Parallelism on a multi-core system:**



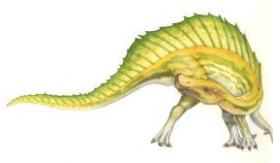


Concurrency vs. Parallelism



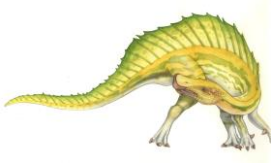
Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task by allowing all task to make progress.
  - Single processor / core, scheduler providing concurrency

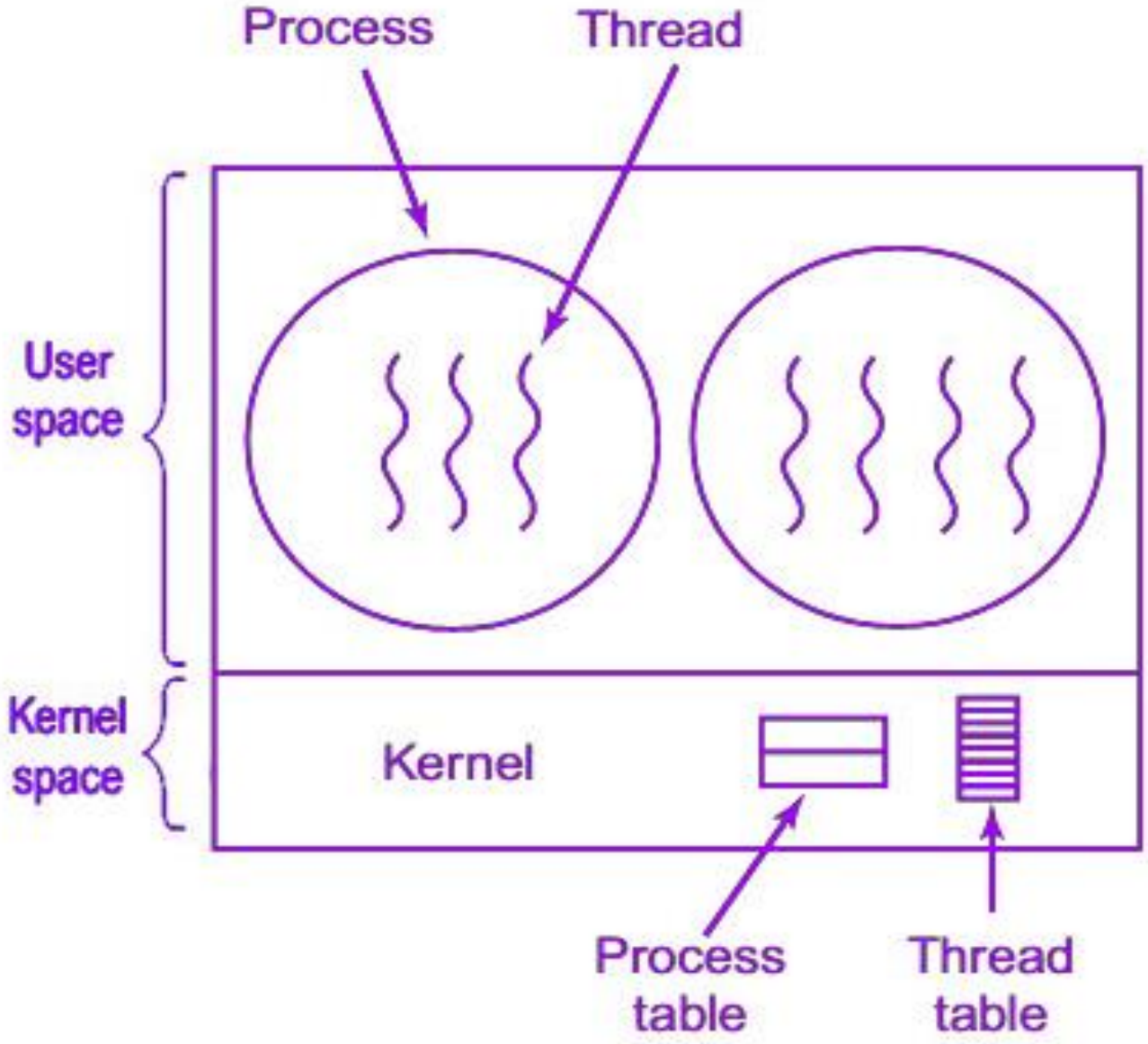


Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each core
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core



User Threads and Kernel Threads



ULT:
Managed without kernel support

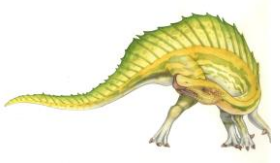
KLT:
Managed by OS

User Threads and Kernel Threads

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

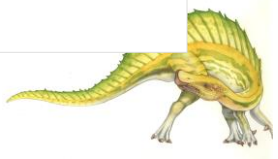
Thread Libraries

- **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*



User Level thread vs Kernel Level thread

S. No.	Parameters	User Level Thread	Kernel Level Thread
1.	**Implemented by**	User threads are implemented by users.	Kernel threads are implemented by Operating System (OS).
2.	**Recognize**	Operating System doesn't recognize user level threads.	Kernel threads are recognized by Operating System.
3.	**Implementation**	Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
4.	**Context switch time**	Context switch time is less.	Context switch time is more.



User Level thread vs Kernel Level thread

5.	**Hardware support**	Context switch requires no hardware support.	Hardware support is needed.
6.	**Blocking operation**	If one user level thread performs blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
7.	**Multithreading**	Multithread applications cannot take advantage of multiprocessing.	Kernels can be multithreaded.
8.	**Creation and Management**	User level threads can be created and managed more quickly.	Kernel level threads take more time to create and manage.

GATE | GATE-CS-2014

Which one of the following is **FALSE?**

(A) User level threads are not scheduled by the kernel.

(B) When a user level thread is blocked, all other threads of its process are blocked.

(C) Context switching between user level threads is faster than context switching between kernel level threads.

(D) Kernel level threads cannot share the code segment

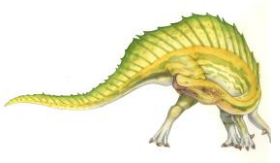
Pthreads

- Parallel execution model.
- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- 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)

pthread library and the thread functions

- pthread_create()
- Creation of a thread can be done using the pthread_create() function in the pthread library.

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

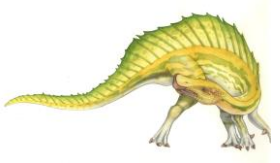


thread functions

- Number of arguments: 4
- 1st argument is a pointer to pthread_t and it represents the TID(thread ID). This is a unique ID assigned to the threads in a certain process.
- 2nd argument speaks about attributes and using this we can specify the features (or) properties of the current thread.
- When the pthread_create function is called it will create a context for the thread. A context is nothing but all the data that a certain thread needs to start its execution. The context will start its execution from the function specified in the 3rd argument.
- 4th arg is a pointer to the arguments of the function that we pass as the 3rd argument.

Thread functions

- pthread_join()
- The pthread_join() is used in order to wait for a thread to complete its execution.
- int pthread_join(pthread_t th, void **thread_return);
- Number of arguments : 2
- 1st argument is the TID(thread id) of the thread that the current thread is waiting for.
- 2nd argument is the pointer that points to the location that stores the return status of the thread ID that is referred to in the 1st argument.



Pthreads example (1)

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

void* samplejob(){
        printf("Hello World!");
}
 int main (int argc,char* argv[]) {
pthread_t first_thread;
pthread_create( &first_thread , NULL , &samplejob , NULL );
pthread_join(first_thread,NULL);
return 0;
}
```

Pthreads example (2)

```
#include <pthread.h>
// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
        sleep(1);
        printf("Printing from Thread \n");
        return NULL;
}
```

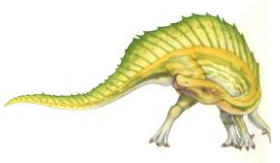
```
int main()
{
        pthread_t thread_id;
        printf("Before Thread\n");
        pthread_create(&thread_id, NULL, myThreadFun, NULL);
        pthread_join(thread_id, NULL);
        printf("After Thread\n");
        exit(0);
}
```

Output:

```
Before Thread
Printing from Thread
After Thread
```

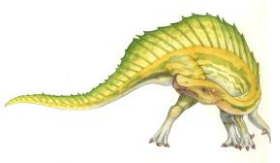
Pthreads Example (3)

```
#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.)

```
   /* 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);
}
```



Pthreads Code for Joining 10 Threads

```
#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);
```

Windows Threads

- CreateThread() function
- The creating thread must specify the **starting address** of the code that the new thread is to execute.
- The starting address is the name of a function defined in the program code
- This function takes a single parameter and returns a **DWORD** value. A process can have multiple threads simultaneously executing the same function.

Windows thread

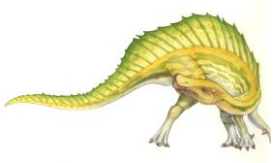
```
#include <windows.h>

DWORD WINAPI ThreadFunc(void* data) {
  // Do stuff.  This will be the first function called on the new thread.
  // When this function returns, the thread goes away.  See MSDN for more details.
  return 0;
}


int main() {
  HANDLE thread = CreateThread(_____);
  if (thread) {
    // Optionally do stuff, such as wait on the thread.
  }
}
```

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);
   }
}
```



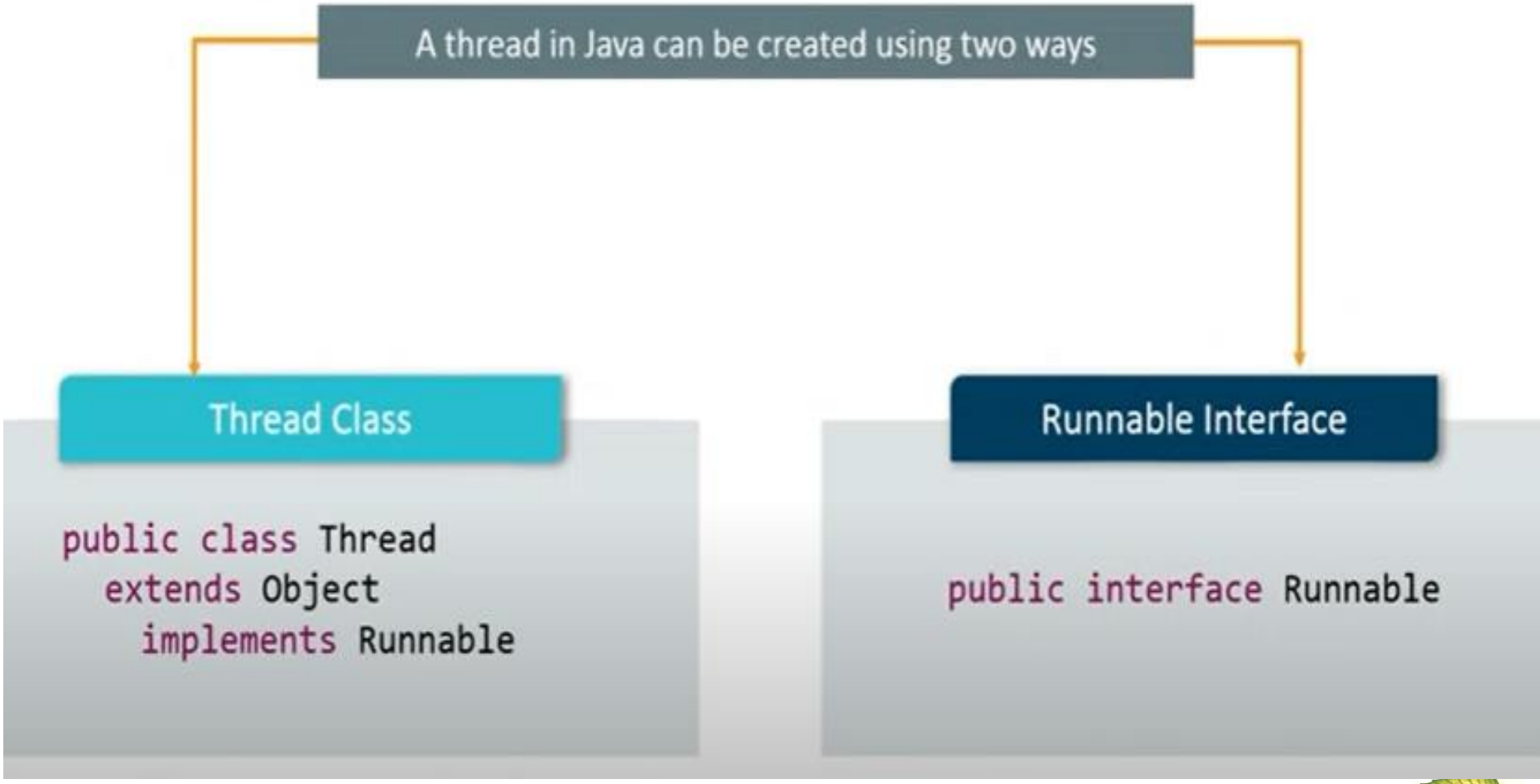
Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- **Created & controlled by java.lang.Thread class**
- Java threads may be created by:

1. Extending Thread class and to override its run () method
2. Implementing the Runnable interface

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

Thread creation in Java

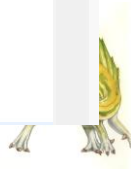


Java Threads

- **Thread creation by extending the Thread class**
  We create a class that extends **java.lang.Thread** class.
- This class overrides the run() method available in the Thread class. A thread begins its life inside run() method.
- We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

Java Threads: Extending thread class

```
class SampleThread extends Thread{

    public void run() {
        System.out.println("Thread is under Running...");
        for(int i= 1; i<=10; i++) {
            System.out.println("i = " + i);
        }
    }
}

public class My_Thread_Test {

    public static void main(String[] args) {
        SampleThread t1 = new SampleThread();
        System.out.println("Thread about to start...");
        t1.start();
    }
```



Java threads

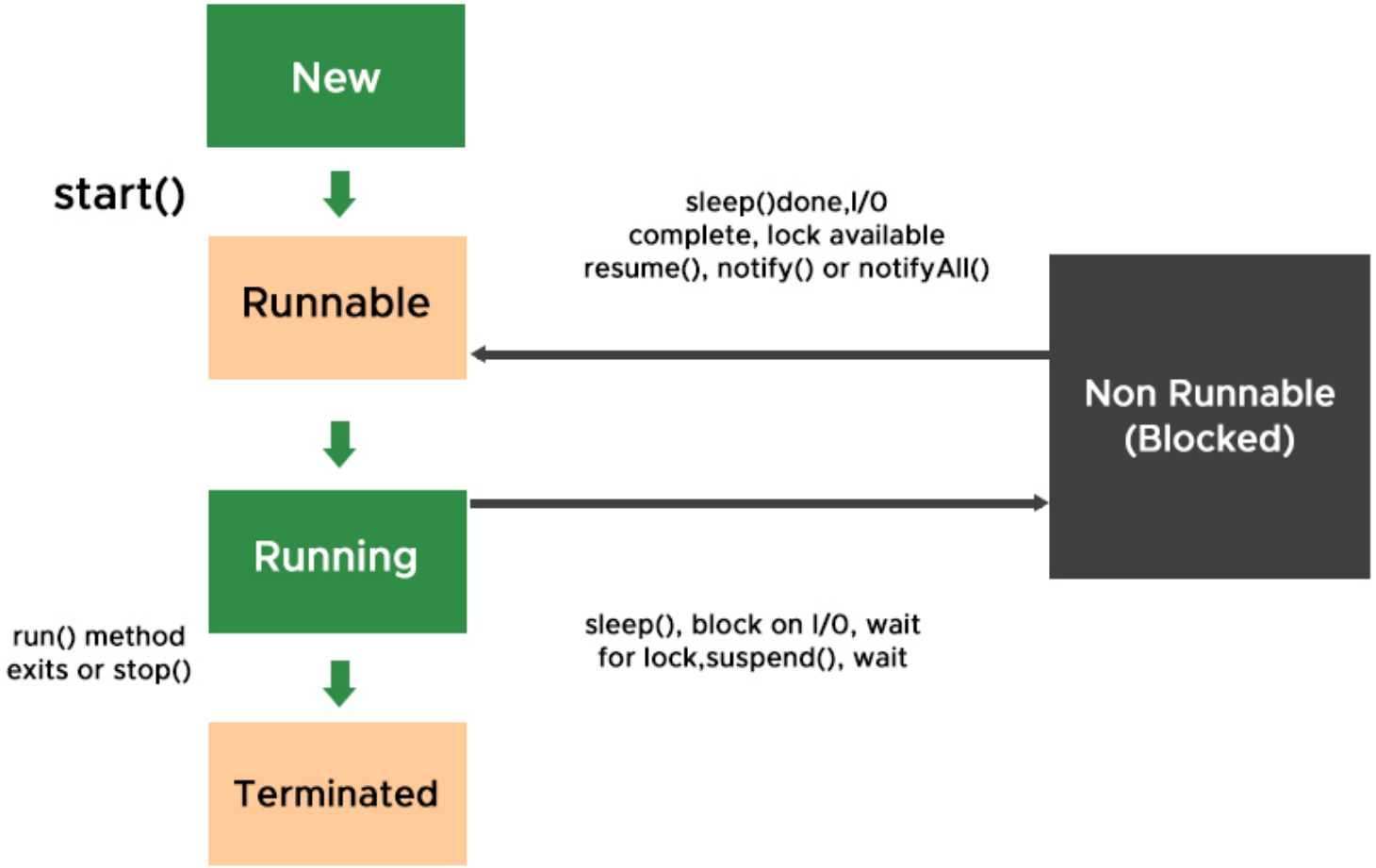
- Starting a thread:
- The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:
  - A new thread starts(with new callstack).
  - The thread moves from New state to the Runnable state.
  - When the thread gets a chance to execute, its target run() method will run.

Runnable interface

- The java contains a built-in interface Runnable inside the java.lang package.
- The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.
- public void run(): is used to perform action for a thread.
- *If you are not extending the Thread class, your class object would not be treated as a thread object.You need to explicitly create the Thread class object.*

Thread creation by implementing the Runnable Interface

```
class Multi3 implements Runnable{
public void run(){
        System.out.println("thread is running...");
}
  public static void main(String args[]){
        Multi3 m1=new Multi3();
        Thread t1 =new Thread(m1);   // Using the constructor
                                          Thread(Runnable r)
        t1.start();
        }
}
```

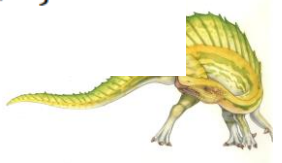

New
start()
Runnable
sleep()done,I/O
complete, lock available
resume(), notify() or notifyAll()
Non Runnable
(Blocked)
Running
run() method
exits or stop()
sleep(), block on I/O, wait
for lock,suspend(), wait
Terminated

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>"); }
}
```

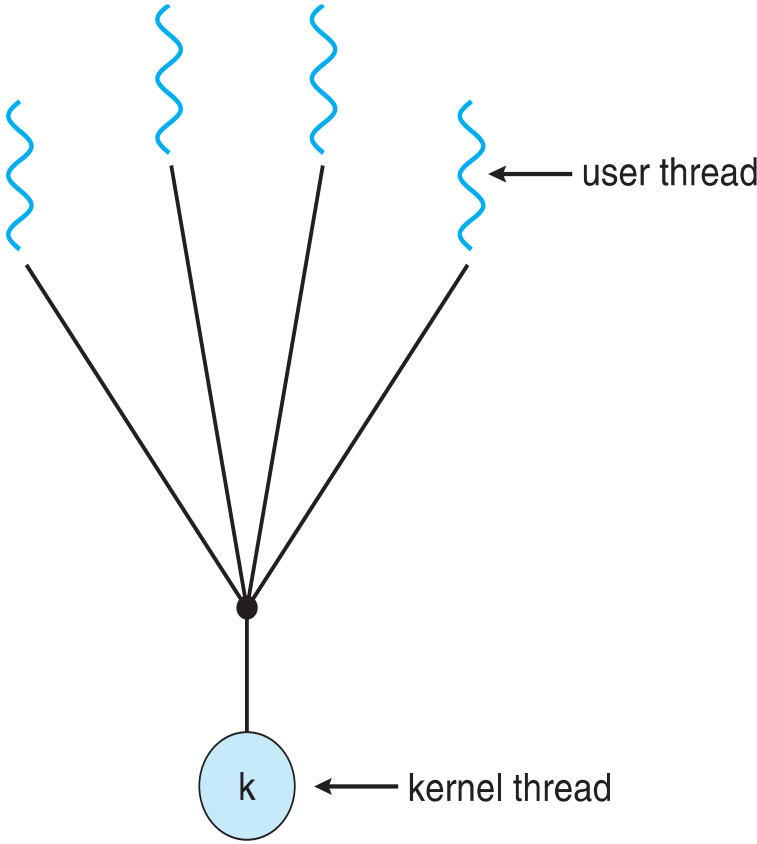


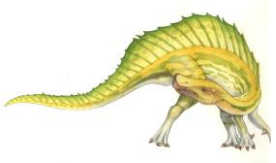
Multithreading Models

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

Many-to-One

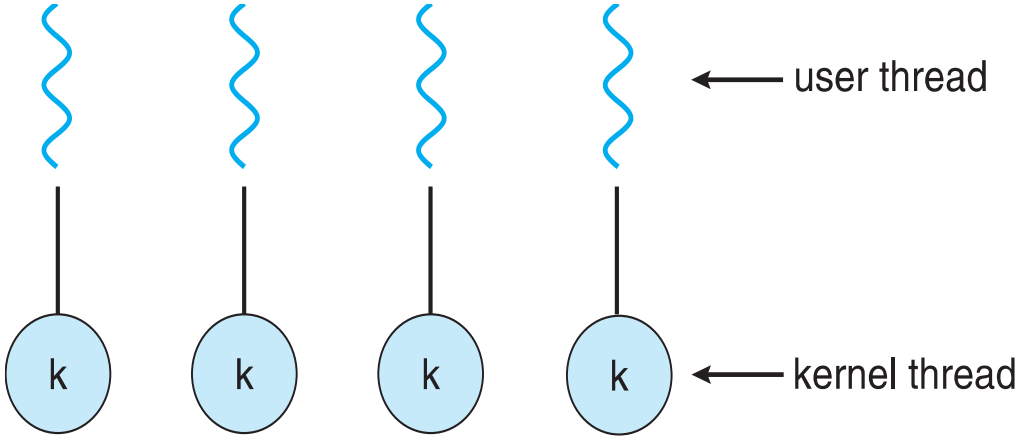
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable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

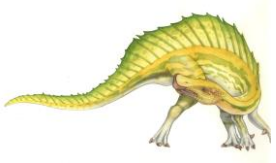




One-to-One

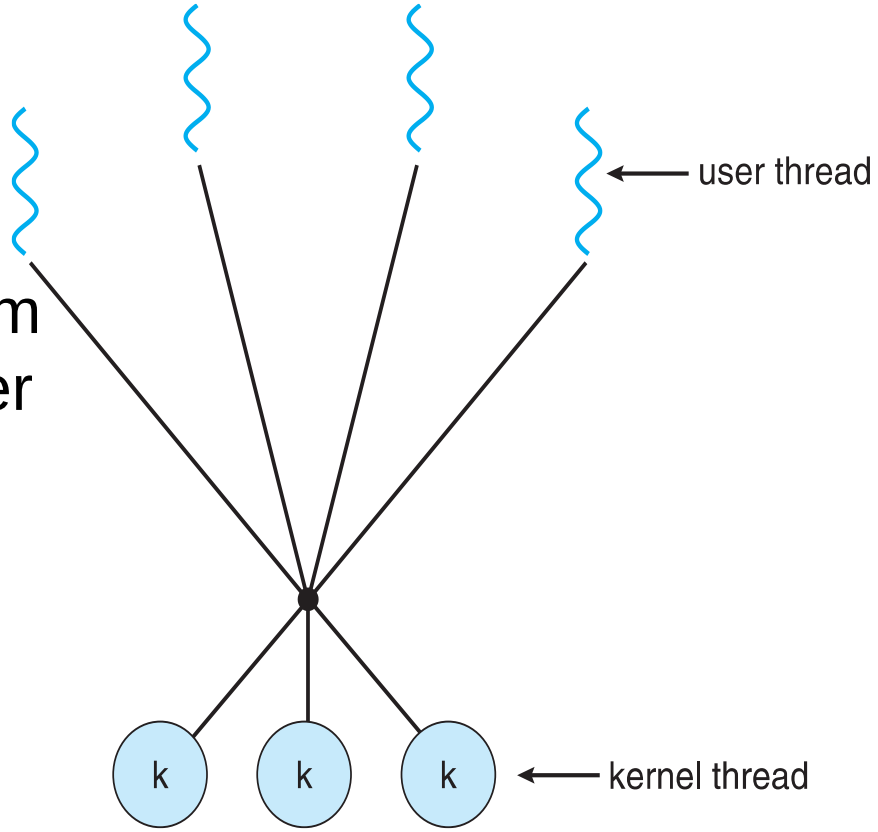
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

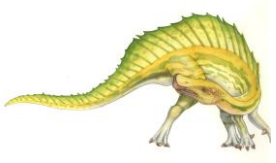




Many-to-Many Model

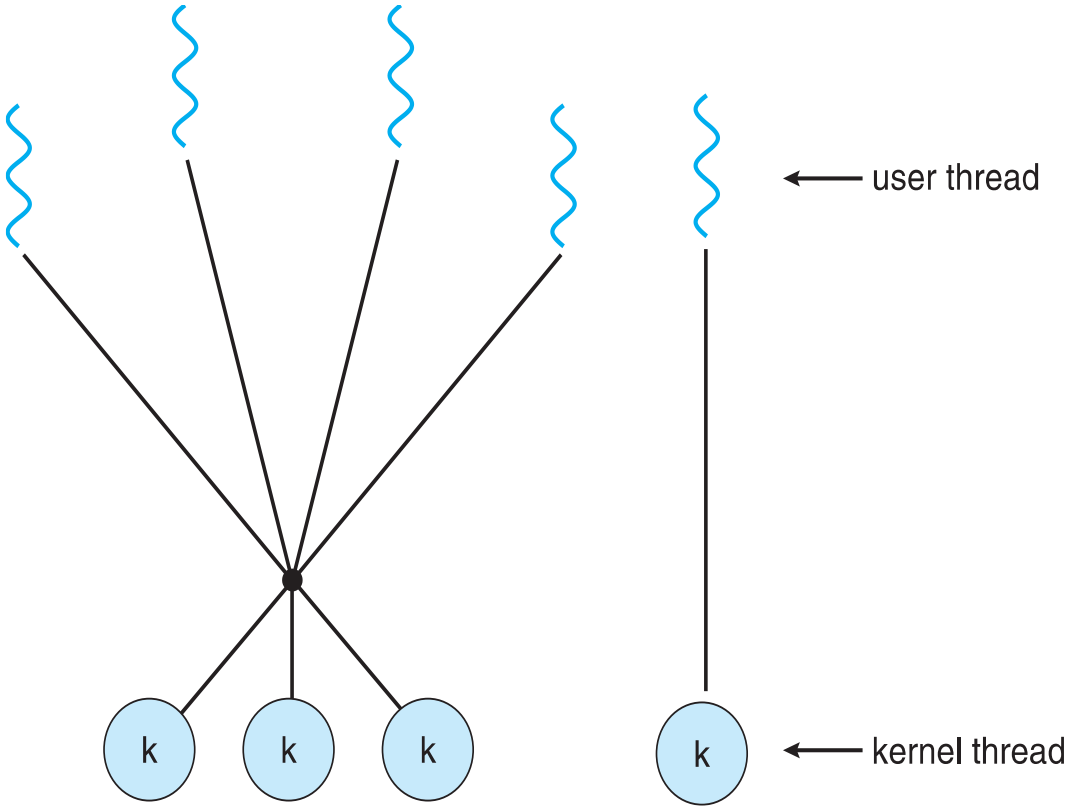
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package





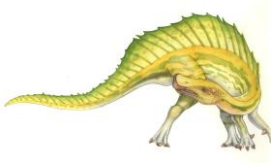
Two-level Model

- 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

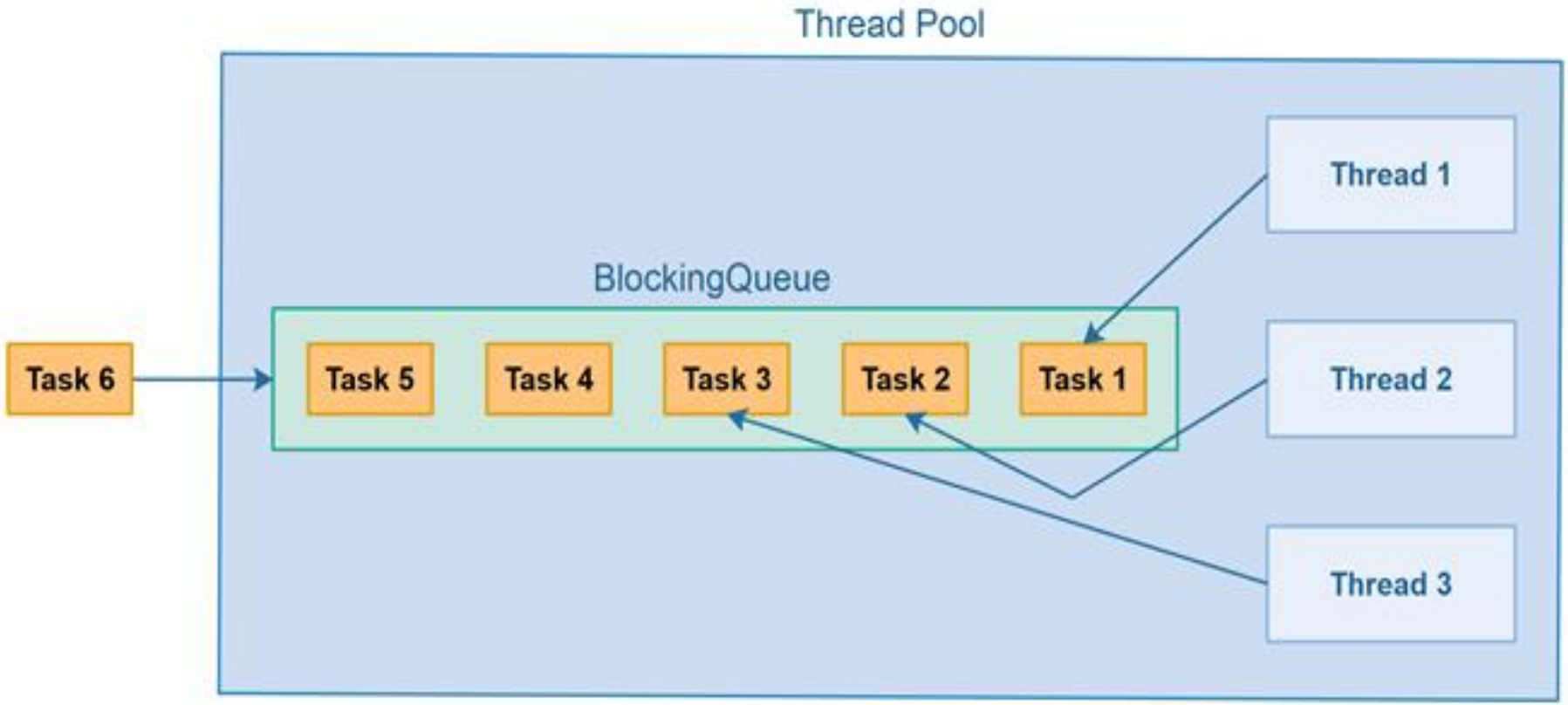


Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), **`java.util.concurrent`** package



Thread Pool


Thread Pool
BlockingQueue
Task 6
Task 5
Task 4
Task 3
Task 2
Task 1
Thread 1
Thread 2
Thread 3

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly 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:

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

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

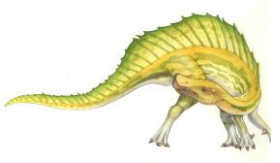
```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for
   for(i=0;i<N;i++) {
     c[i] = a[i] + b[i];
}
```

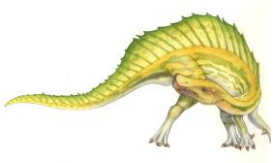
Run for loop in parallel

```
#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;
}
```



Grand Central Dispatch

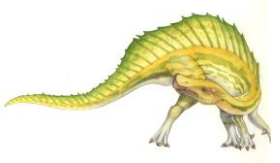
- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in "^{ }" - `^{ printf("I am a block"); }`
- Blocks placed in dispatch queue
  - Assigned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



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."); });
```

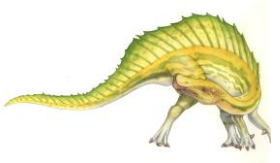


Threading Issues

- 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

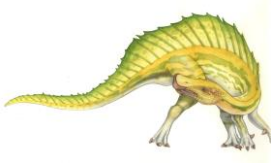
Semantics of fork() and exec()

- 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



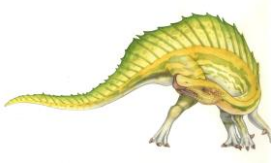
Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. 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



Signal Handling (Cont.)

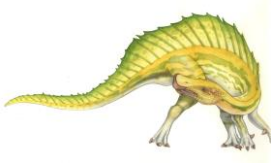
- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process



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);
```



Thread Cancellation (Cont.)

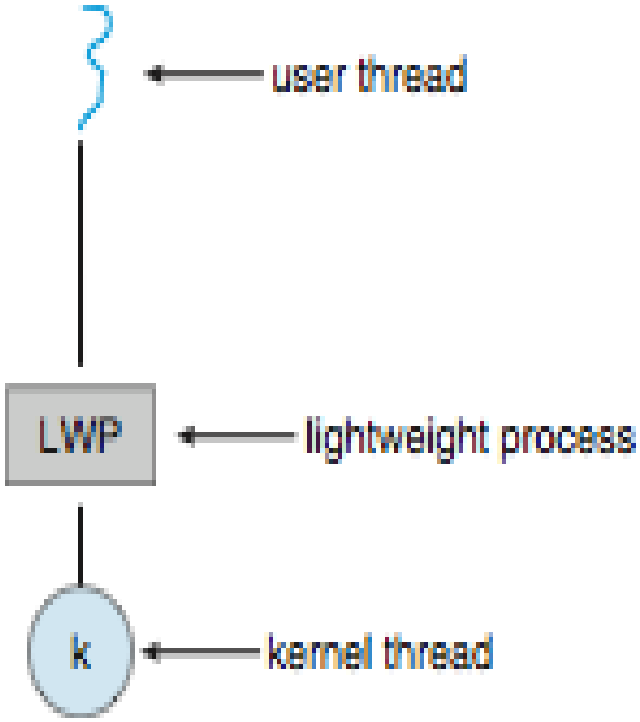
- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

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

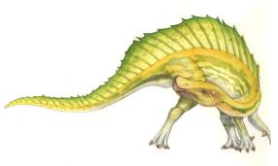
Scheduler Activations

- 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

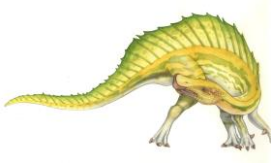


Windows Threads

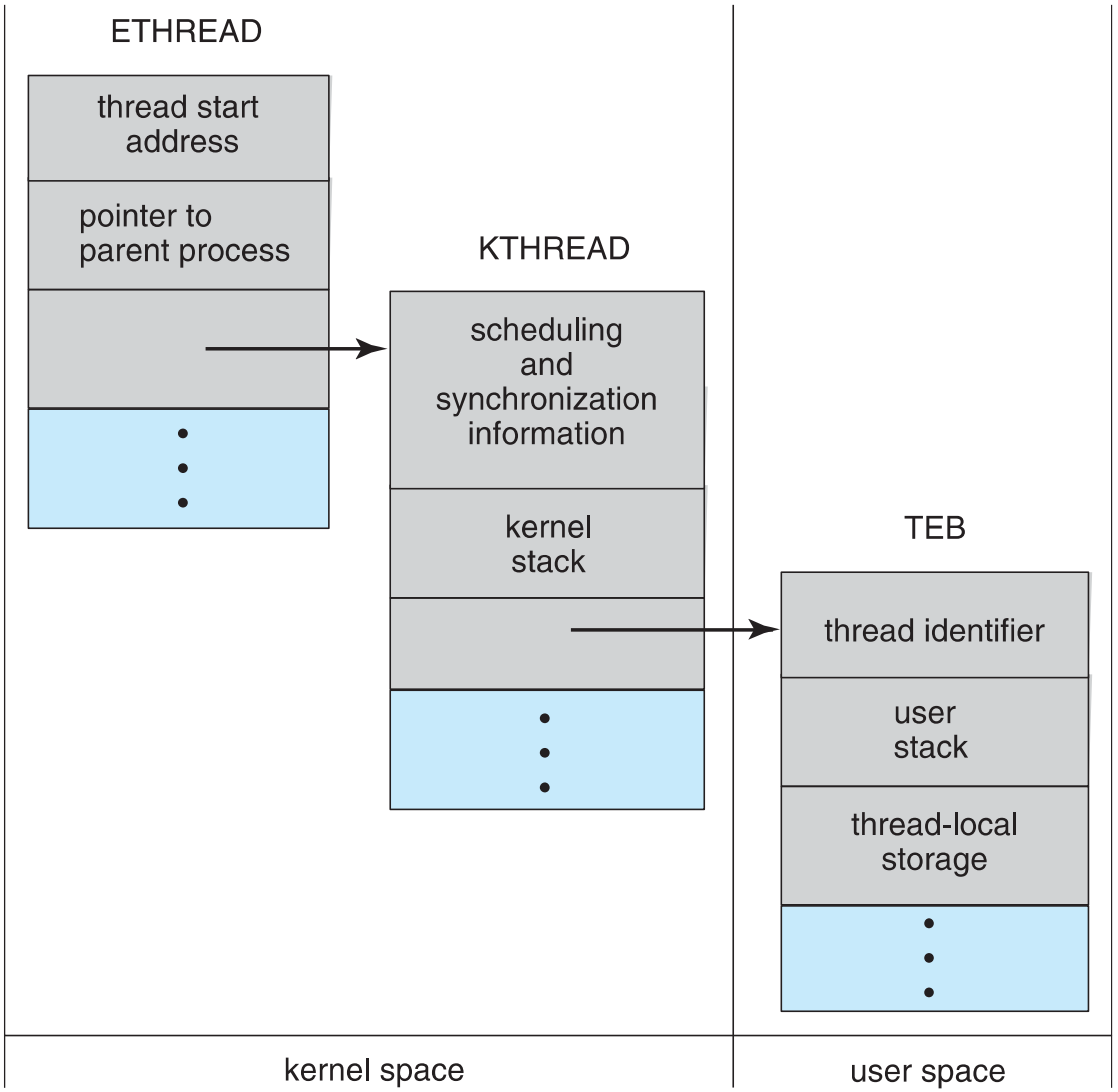
- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user 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

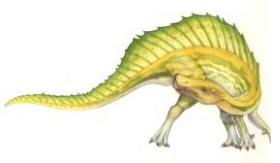
Windows Threads (Cont.)

- 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



Windows Threads Data Structures



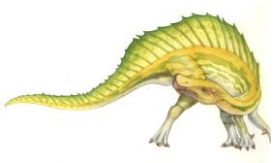


Linux Threads

- 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)



End of Chapter