CHAPTER 5: THREADS

OPERATING SYSTEMS (CS-220)
FALL 2020, FAST NUCES

COURSE SUPERVISOR: ANAUM HAMID

anaum.hamid@nu.edu.pk

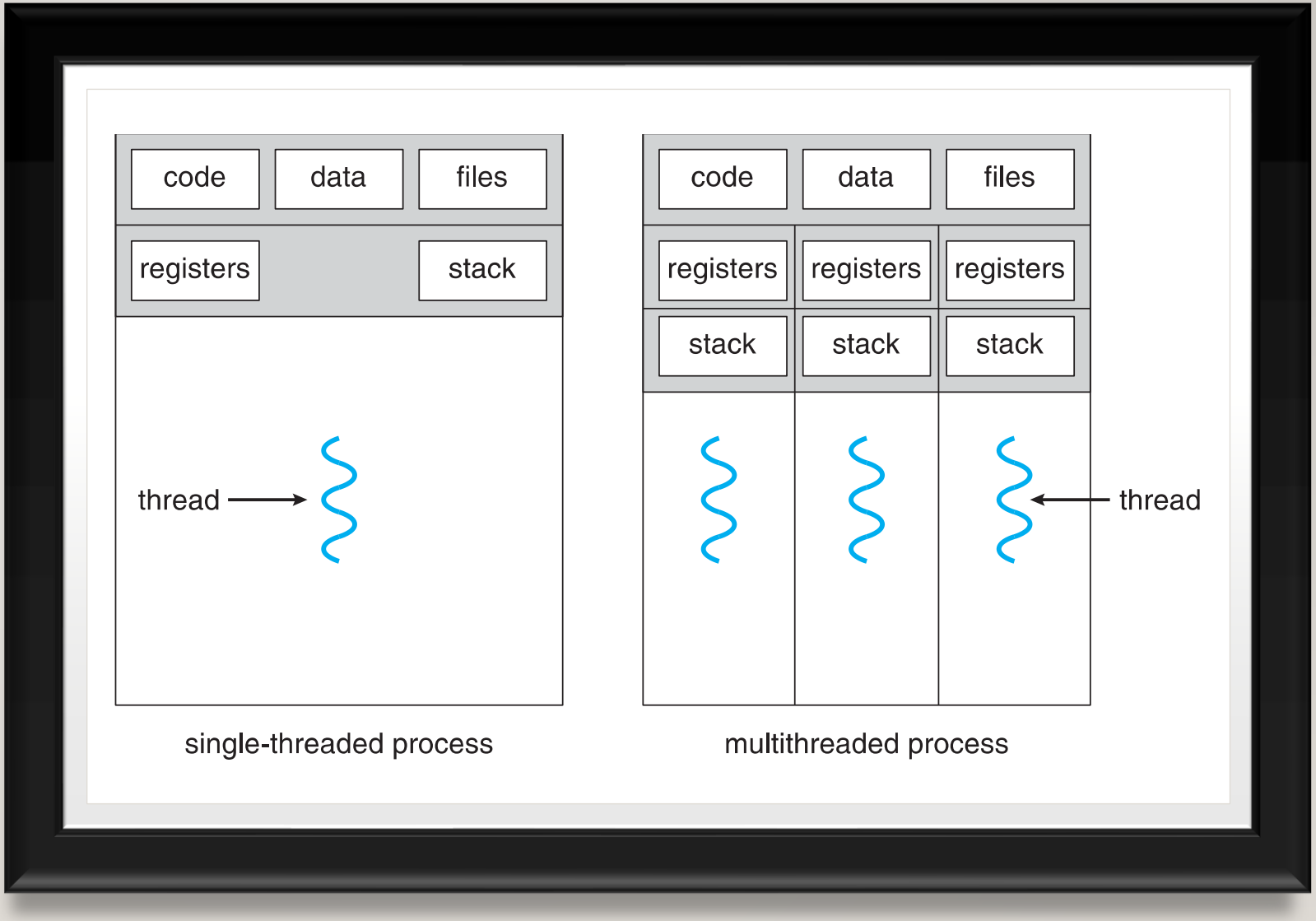
2 ROADMAP

1. Introduction
2. Multicore Programming
3. Multithreading Models
4. Threads Libraries
5. Implicit Threading
6. Threading Issues
7. Threads Scheduling

3 INTRODUCTION

- 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.)
- Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- Multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.

SINGLE AND MULTITHREADED PROCESSES



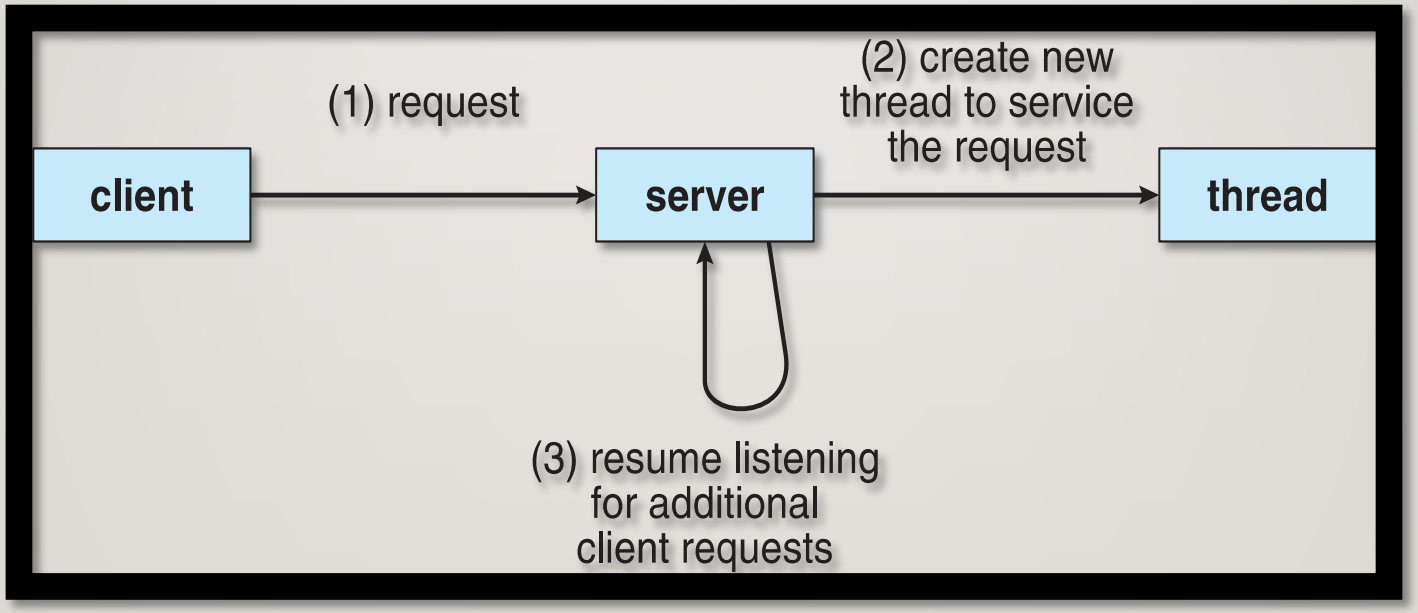
PROCESS VS. THREADS

S.N.	Process	Thread
1.	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2.	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3.	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.
4.	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.
5.	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6.	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 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

MULTITHREADED SERVER ARCHITECTURE



BENEFITS

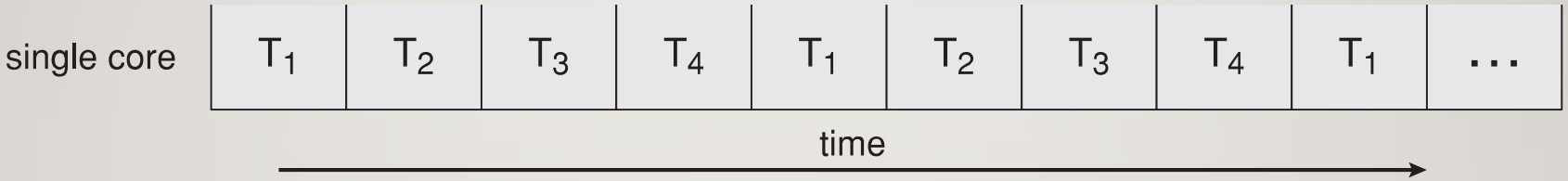
- **Responsiveness –** One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
- **Resource Sharing –** By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
- **Economy –** Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
- **Scalability –** Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors.

9 MULTICORE PROGRAMMING

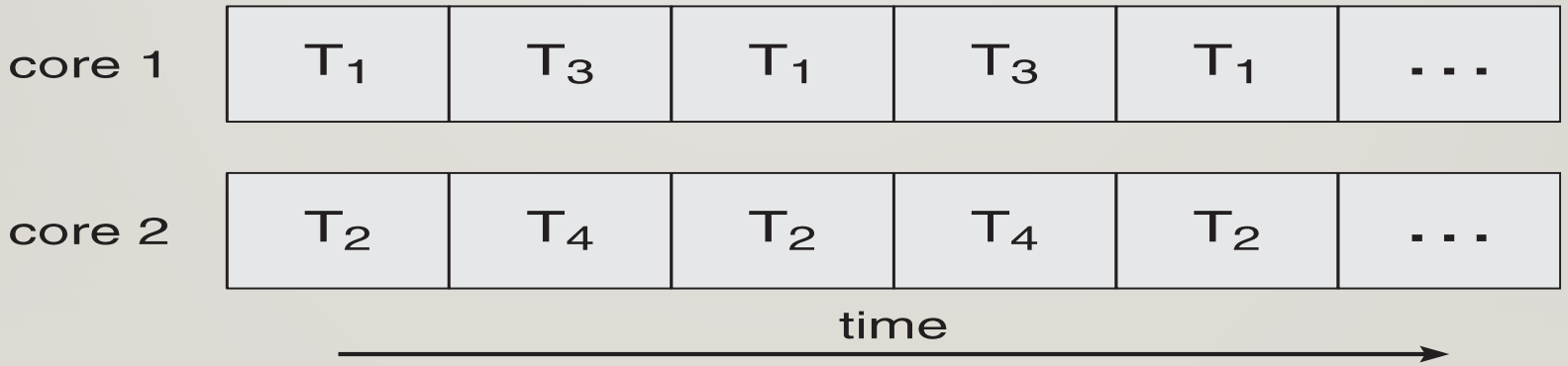
- A recent trend in computer architecture is to produce chips with multiple ***cores***, or CPUs on a single chip.
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

CONCURRENCY VS. PARALLELISM

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



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



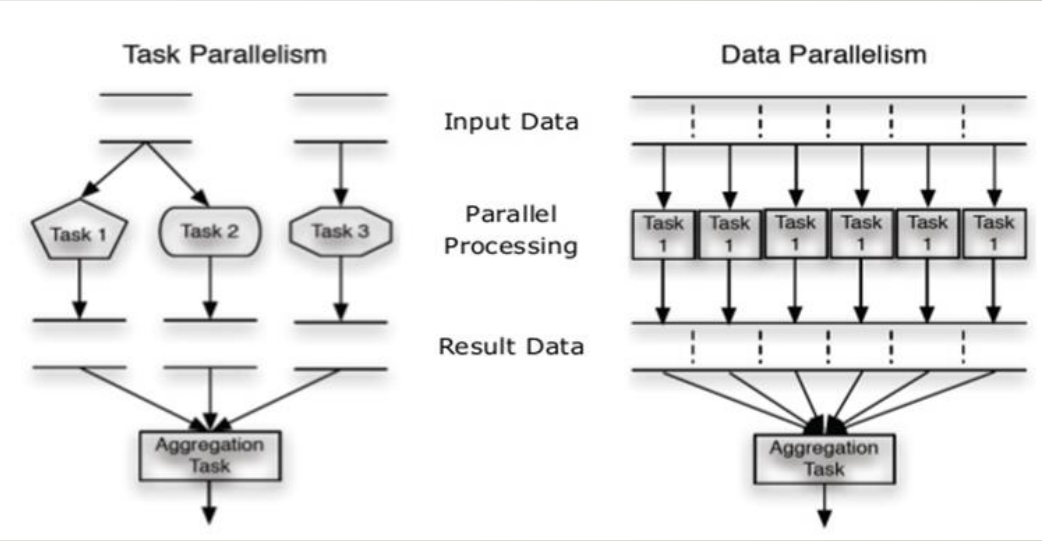
11 TYPES OF PARALLELISM

In theory there are two different ways to parallelize the workload:

1. **Data parallelism** divides the data up amongst multiple cores (threads) and performs the same task on each subset of the data. For example dividing a large image up into pieces and performing the same digital image processing on each piece on different cores.
2. **Task parallelism** divides the different tasks to be performed among the different cores and performs them simultaneously.

In practice no program is ever divided up solely by one or the other of these, but instead by some sort of hybrid combination.

TYPES OF PARALLELISM



DATA VS. TASK PARALLELISM

Data Parallelism	Task Parallelism
Same operations are performed on different subsets of same data.	Different operations are performed on the same or different data.
Synchronous computation	Asynchronous computation
Speedup is more as there is only one execution thread operating on all sets of data.	Speedup is less as each processor will execute a different thread or process on the same or different set of data.
Amount of parallelization is proportional to the input data size.	Amount of parallelization is proportional to the number of independent tasks to be performed
Designed for optimum load balance on multi processor system.	Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling.

14 MULTICORE PROGRAMMING

- A multi-threaded application running on a traditional single-core chip would have to interleave the threads. On a multi-core chip, however, the threads could be spread across the available cores, allowing true parallel processing.
- Multi-threading becomes more pervasive and more important (thousands instead of tens of threads), CPUs have been developed to support more simultaneous threads per core in hardware. (Hyperthreading)

15 PROGRAMMING CHALLENGES

For Multicore application programmers, there are five areas where multi-core chips present new challenges:

1. **Identifying tasks** - Examining applications to find activities that can be performed concurrently.
2. **Balance** - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.
3. **Data splitting** - To prevent the threads from interfering with one another.
4. **Data dependency** - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
5. **Testing and debugging** - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

AMDAHL'S LAW

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

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

- That is, if application is 75% parallel / 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

17 TYPES OF THREADS

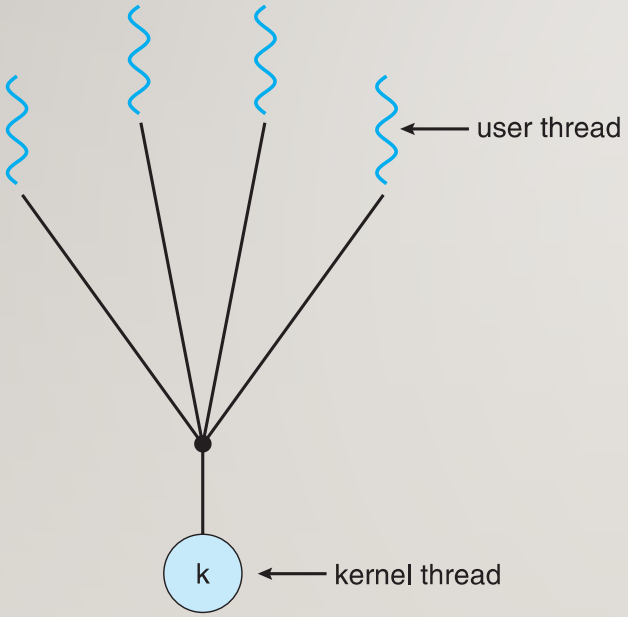
- There are two types of threads to be managed in a modern system:
- USER THREADS are supported by user level thread library (Pthreads, Windows Threads, Java Threads), without kernel support. These are the threads that application programmers would put into their programs.
- KERNEL THREADS are supported within the kernel of the OS itself. All modern OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

18 MULTITHREADING MODELS

In a specific implementation, the user threads must be mapped to kernel threads, using these listed below Multithreading Models:

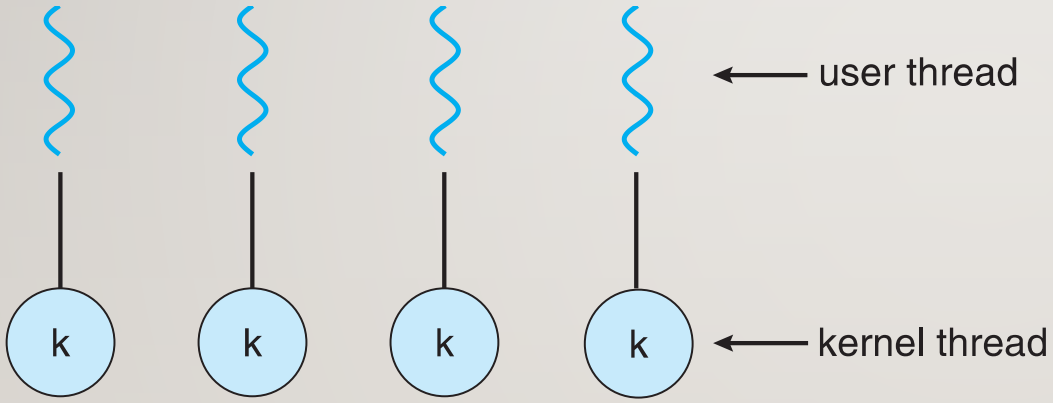
1. Many-to-One
2. One-to-One
3. Many-to-Many

19 MANY-TO-ONE



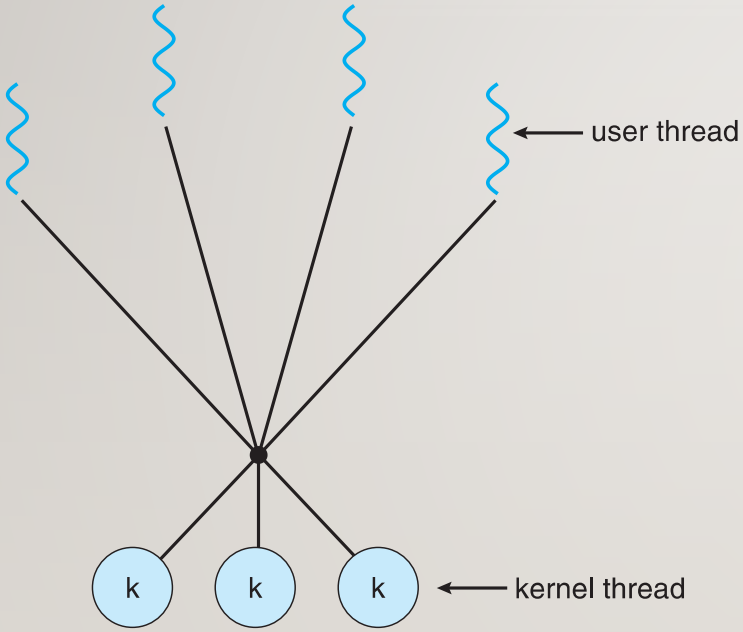
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable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

20 ONE-TO-ONE



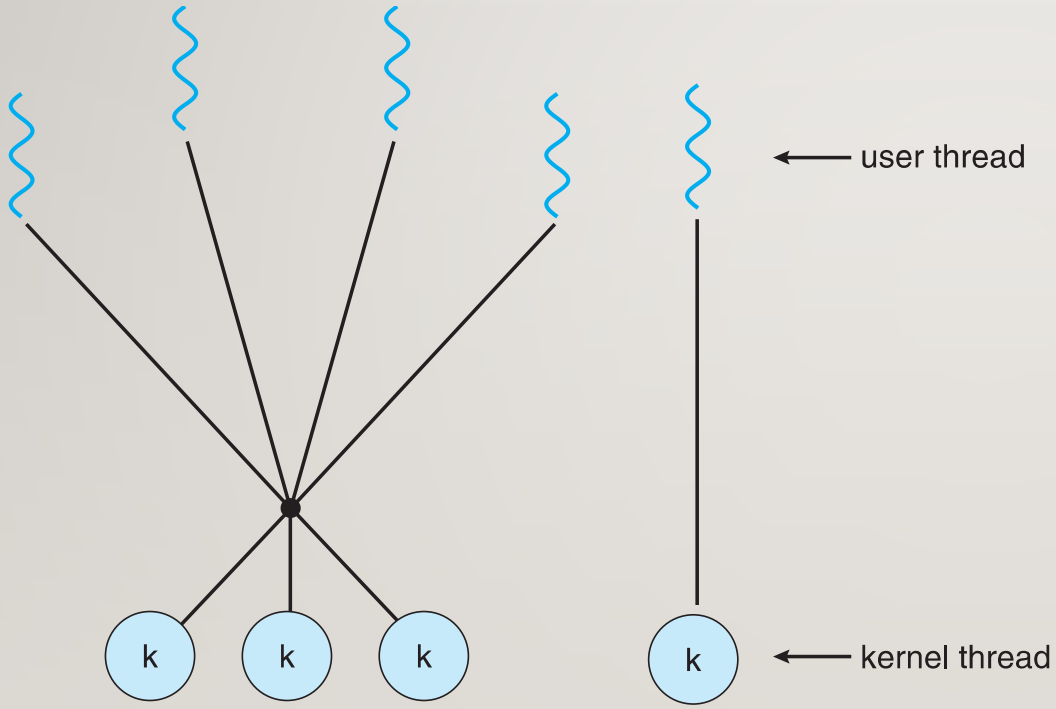
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

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

22 TWO-TIER/LEVEL MODEL



- Like 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

23 THREAD LIBRARIES

- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls and requires a kernel with thread library support.
- There are three main thread libraries in use today:
  1. POSIX PTHREADS - may be provided as either a user or kernel library, as an extension to the POSIX standard.
  2. WIN32 THREADS - provided as a kernel-level library on Windows systems.
  3. JAVA THREADS - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

24 PROBLEM:

Programming Code will demonstrate the use of threads in all three systems for calculating the sum of integers from 0 to N in a separate thread and storing the result in a variable "sum".

25 PTHREADS

- The POSIX standard (IEEE 1003.1c) defines the creation and synchronization for pThreads, this is *specification* not the *implementation*.
- PThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
- Windows doesn't support Pthreads natively, third party implementation is available for it.
- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function, in this example the runner( ) function:

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.)

```
   /* 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 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>"); }
}
```

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
- These listed below methods explored
  1. Thread Pools
  2. Fork Join
  3. OpenMP
  4. Grand Central Dispatch
  5. Intel Thread Building Block (TBB)
  6. `java.util.concurrent.*` package

THREAD POOLS

- Creating new threads every time one is needed and then deleting it when it is done can be inefficient and can also lead to a very large (unlimited) number of threads being created.
- An alternative solution is to create several threads when the process first starts and put those threads into a ***thread pool***.
  - Threads are allocated from the pool as needed and returned to the pool when no longer needed.
  - When no threads are available in the pool, the process may have to wait until one becomes available.
- The (maximum) number of threads available in a thread pool may be determined by adjustable parameters, possibly dynamically in response to changing system loads.

THREAD POOLS

- Win32 provides thread pools through the "PoolFunction" function. Java also provides support for thread pools through the java.util.concurrent package, and Apple supports thread pools under the Grand Central Dispatch architecture..
- Windows API supports thread pools:

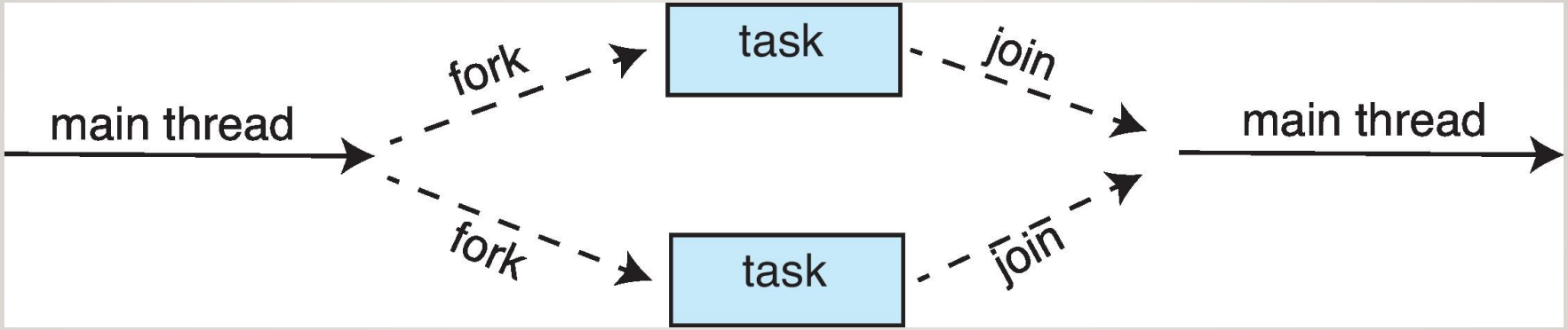
```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```

THREAD POOLS

- 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.

38 FORK-JOIN PARALLELISM

- Multiple threads (tasks) are **forked**, and then **joined**.

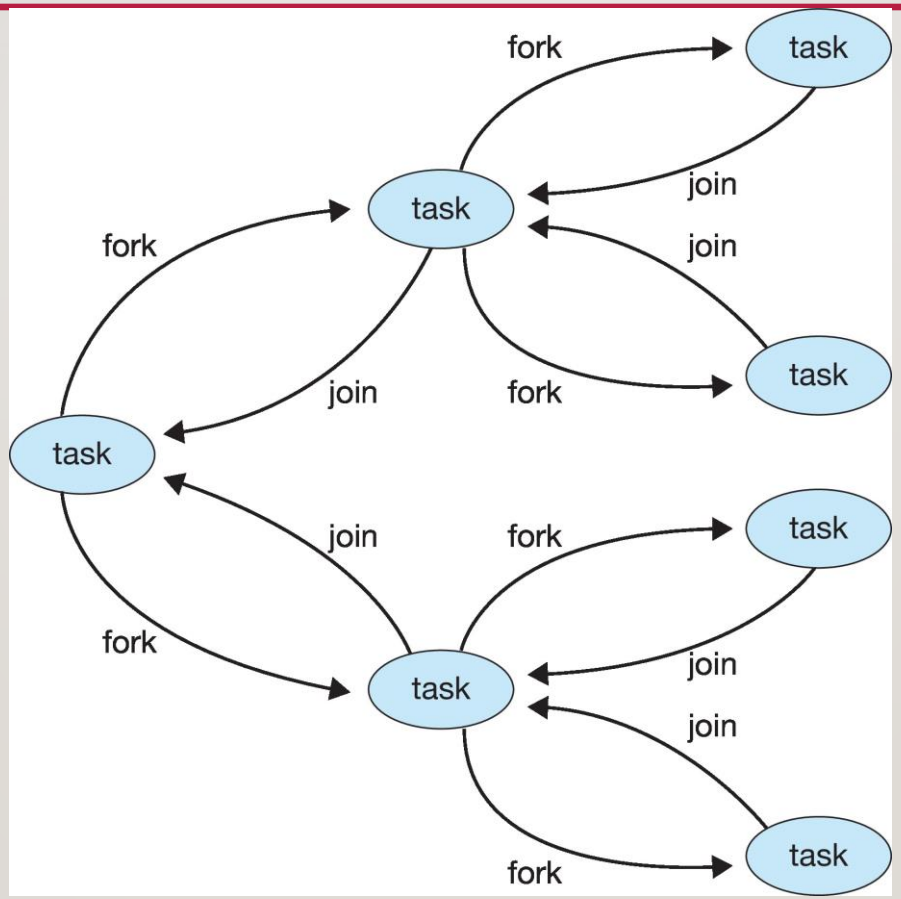


39 FORK-JOIN PARALLELISM

- General algorithm for fork-join strategy:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

40 FORK-JOIN PARALLELISM



41 FORK-JOIN PARALLELISM IN JAVA

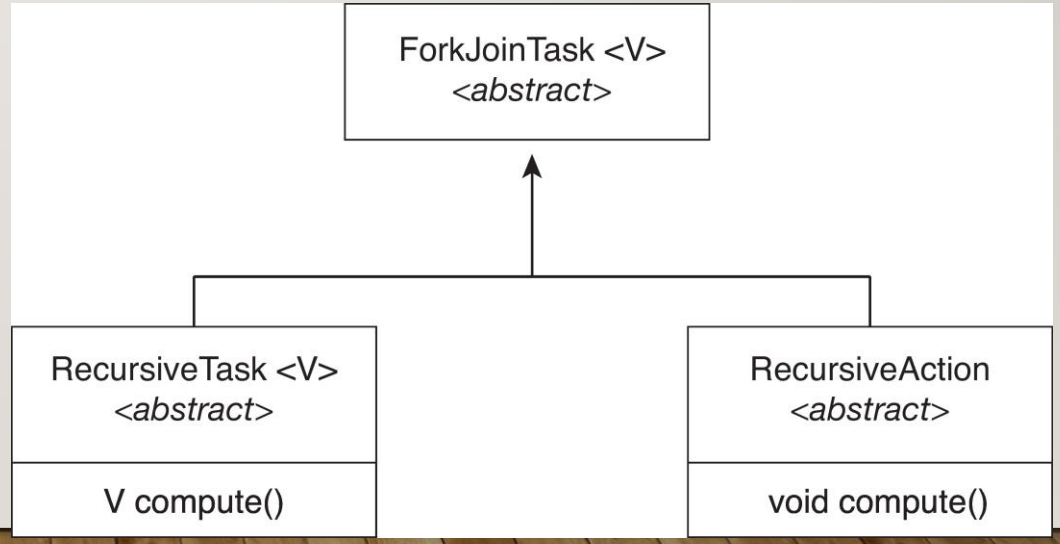
```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

FORK-JOIN PARALLELISM IN JAVA

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
   static final int THRESHOLD = 1000;

   private int begin;
   private int end;
   private int[] array;

   public SumTask(int begin, int end, int[] array) {
      this.begin = begin;
      this.end = end;
      this.array = array;
   }

   protected Integer compute() {
      if (end - begin < THRESHOLD) {
         int sum = 0;
         for (int i = begin; i <= end; i++)
            sum += array[i];

         return sum;
      }
      else {
         int mid = (begin + end) / 2;

         SumTask leftTask = new SumTask(begin, mid, array);
         SumTask rightTask = new SumTask(mid + 1, end, array);

         leftTask.fork();
         rightTask.fork();

         return rightTask.join() + leftTask.join();
      }
   }
}
```

FORK-JOIN PARALLELISM IN JAVA

- The **`ForkJoinTask`** is an abstract base class
- **`RecursiveTask`** and **`RecursiveAction`** classes extend **`ForkJoinTask`**
- **`RecursiveTask`** returns a result (via the return value from the **`compute()`** method)
- **`RecursiveAction`** does not return a result



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 (GCD)

- 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 (GCD)

- 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 processes 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."); });
```

GRAND CENTRAL DISPATCH

- For the Swift language, a task is defined as a closure – like a block, minus the caret
- Closures are submitted to the queue using the `dispatch_async()` function:

```
let queue = dispatch_get_global_queue
   (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue,{ print("I am a closure.") })
```

- Four system wide queues divided by quality of service:
  - `QOS_CLASS_USER_INTERACTIVE`
  - `QOS_CLASS_USER_INITIATED`
  - `QOS_CLASS_USER_UTILITY`
  - `QOS_CLASS_USER_BACKGROUND`

THREADING ISSUES

1. Semantics of **fork()** and **exec()** system calls
2. Signal handling
    1. Synchronous and asynchronous
3. Thread cancellation of target thread
    1. Asynchronous or deferred
4. Thread-local storage
5. Scheduler Activations

SEMANTICS OF FORK() AND EXEC()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork (System dependent)
- **`exec()`** usually works as normal – replace the running process including all threads.
  - Many versions of UNIX provide multiple versions of the fork call for this purpose.

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?
  1. Deliver the signal to the thread to which the signal applies
  2. Deliver the signal to every thread in the process
  3. Deliver the signal to certain threads in the process
  4. 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.)

- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked.

THREAD SCHEDULING

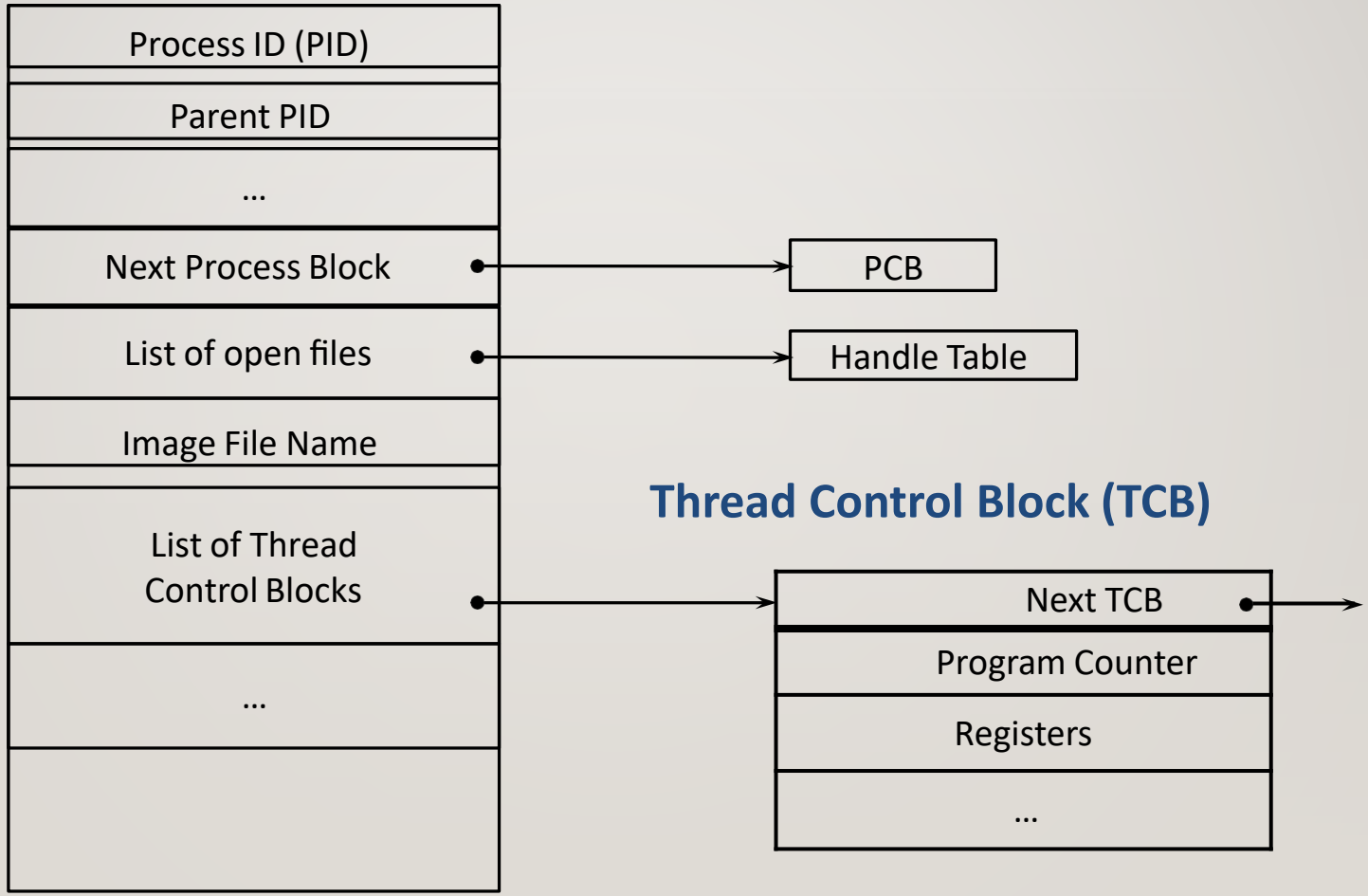
In systems that support user and kernel-level threads, kernel-level threads are scheduled by the OS.

- Kernel-level threads instead of processes are scheduled
- User-level threads are managed by a thread library.
- To run on the CPU, the user-level thread must be mapped on an associated kernel-level thread

CONTROL BLOCKS

- Information associated with each process: **Process Control Block**
  - Memory management information
  - Accounting information
- Information associated with each thread: **Thread Control Block**
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Pending I/O information

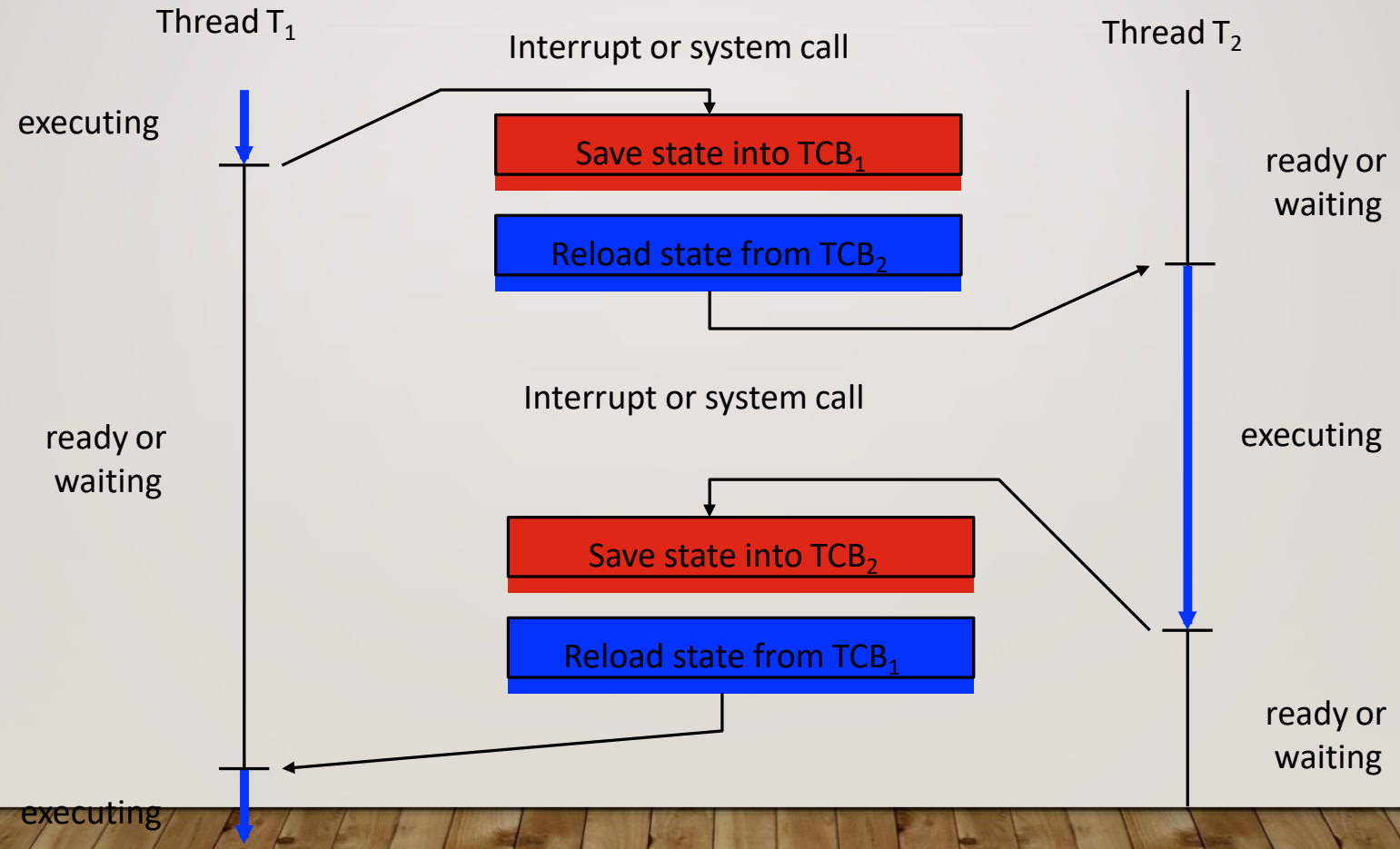
56 CONTROL BLOCKS



THREAD STATES

- The typical states for a thread are **running**, **ready**, **blocked**
- Typical **thread operations** associated with a change in thread state are:
  - **Spawn**: a thread within a process may spawn another thread
    - Provides instruction pointer and arguments for the new thread
    - New thread gets its own register context and stack space
  - **Block**: a thread needs to wait for an event
    - Saving its user registers, program counter, and stack pointers
  - **Unblock**: When the event for which a thread is blocked occurs
  - **Finish**: When a thread completes, its register context and stacks are deallocated.

THREAD DISPATCHING



THREAD-LOCAL STORAGE

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data.
- major thread libraries (pThreads, Win32, Java) provide support for thread-specific data, known as **thread-local storage** or **TLS.**
- 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
- Like `static` data
  - TLS is unique to each thread

THREADS

Threads share….

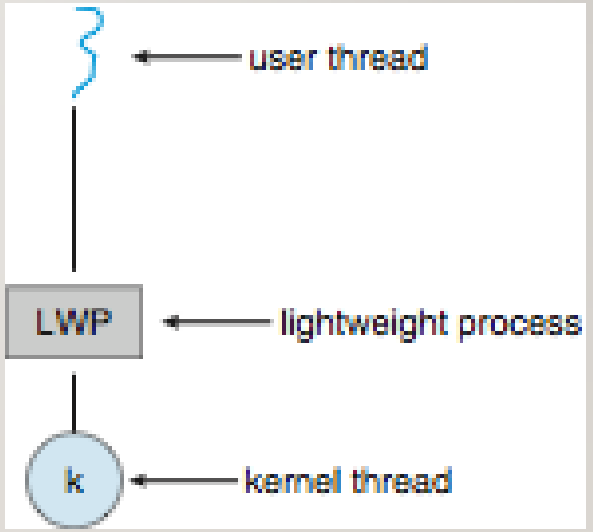
- Global memory
- Process ID and parent process ID
- Controlling terminal
- Process credentials (user)
- Open file information
- Timers
- ……

Threads specific Attributes….

- Thread ID
- Thread specific data
- CPU affinity
- Stack (local variables and function call linkage information)
- ……

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

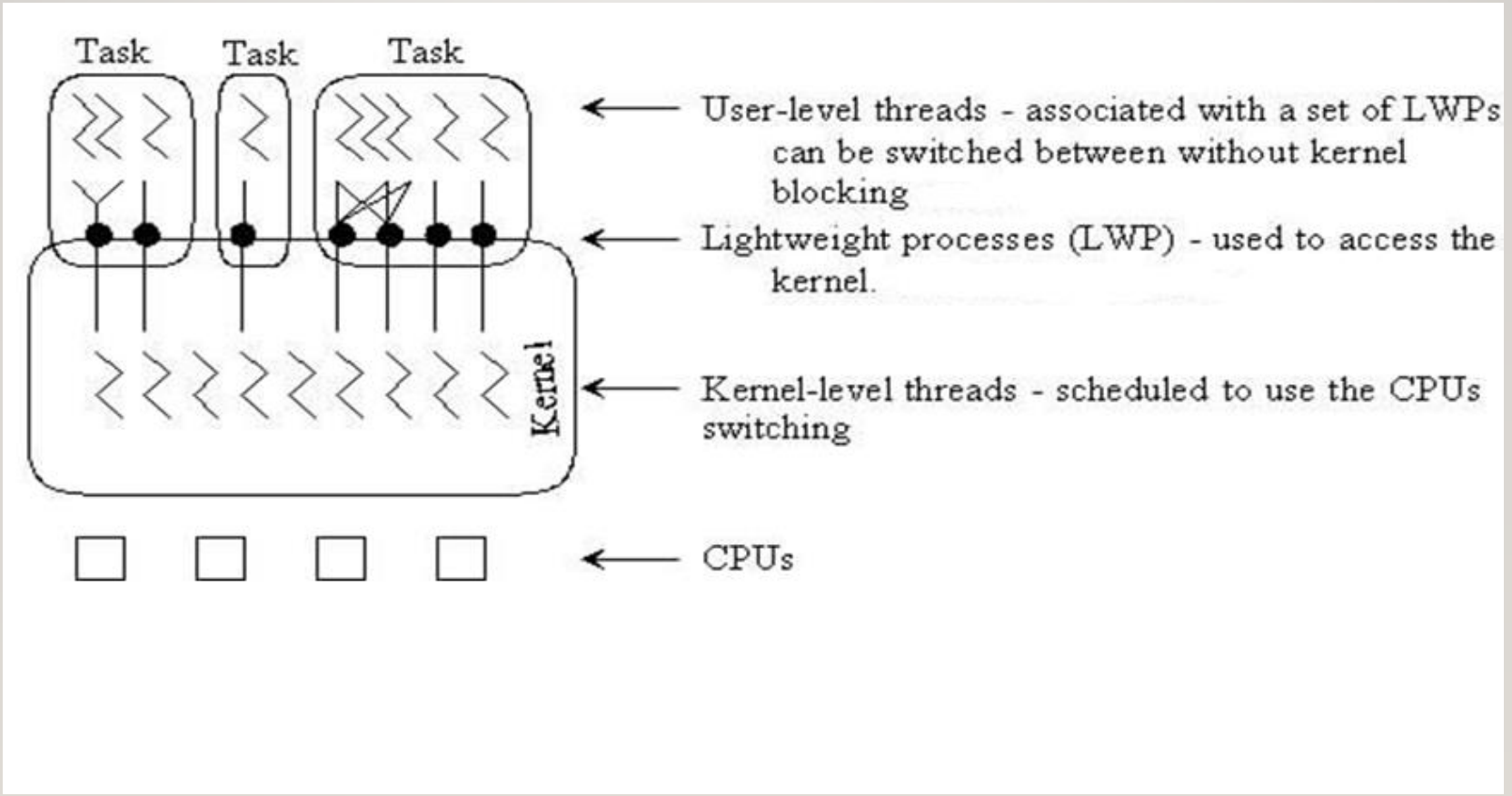


CONTENTION SCOPE

- On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as **process contention scope (PCS)**,

- (When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the threads are running on a CPU. That would require the operating system to schedule the kernel thread onto a physical CPU.) To decide which kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**.

USER VS. KERNEL THREAD



THANK YOU