THREADS
Process Management

BADRUS ZAMAN

Agendas
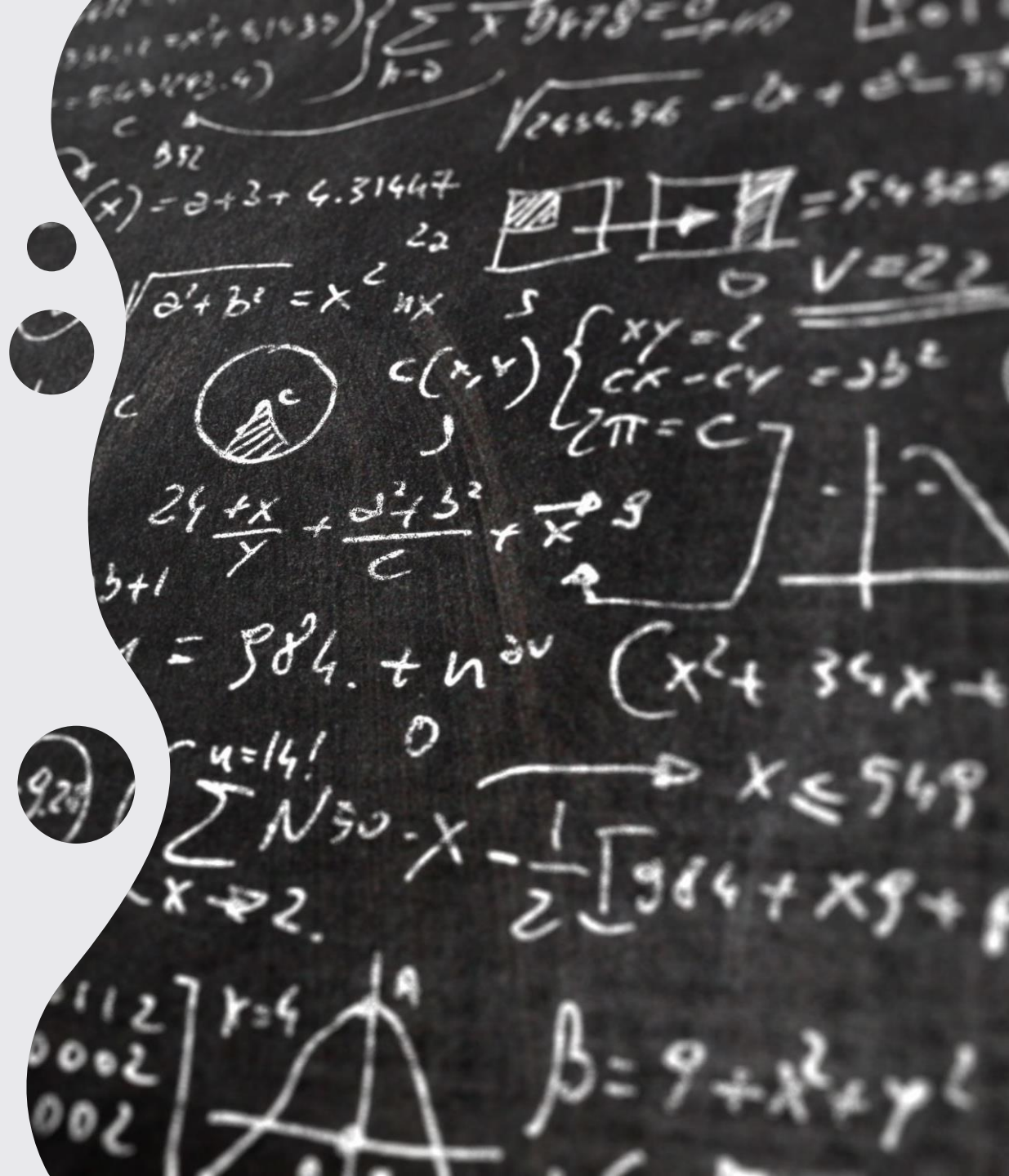
Motivation

- 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
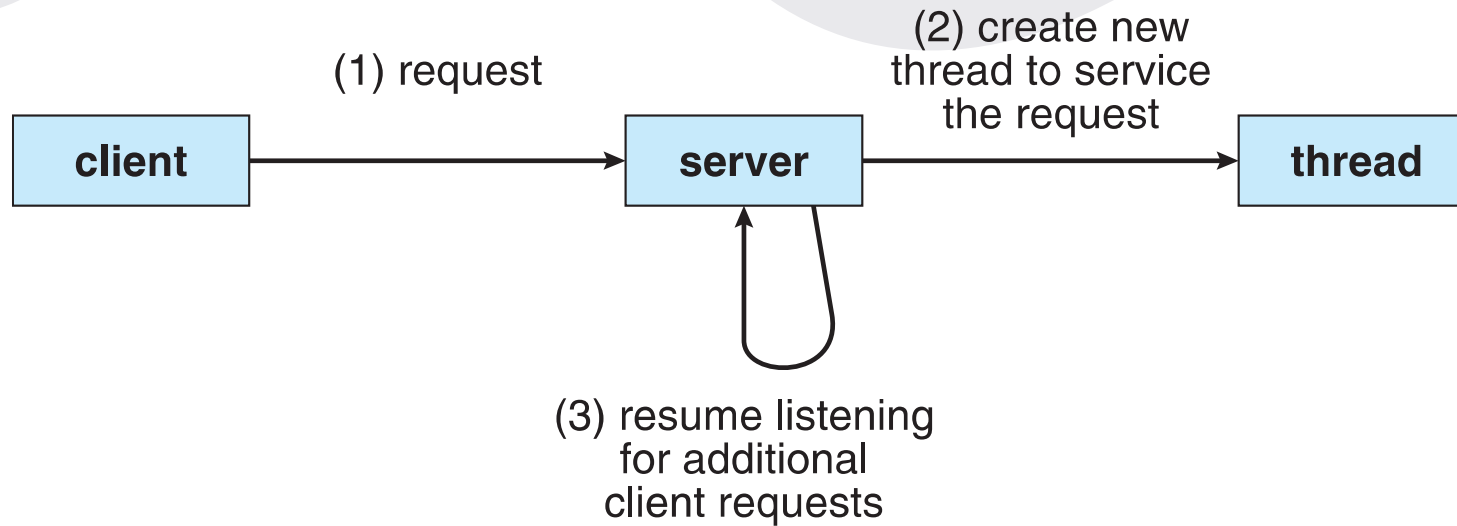
Benefits

- **Responsiveness** – 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

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 making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

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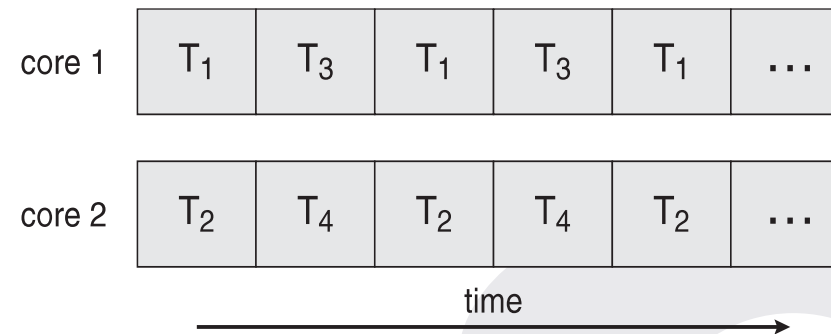 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as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

Concurrency vs. Parallelism

Concurrent execution on single-core system:

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

time →
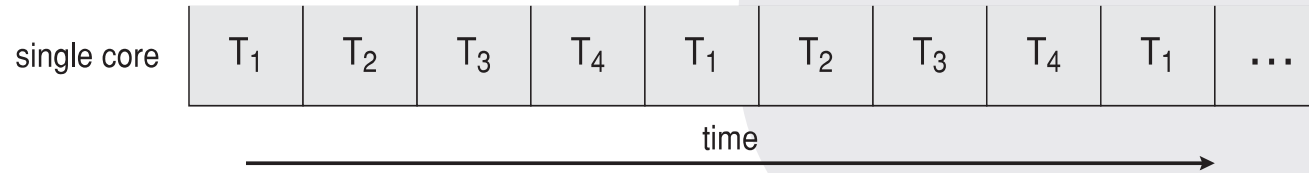
Parallelism on a multi-core system:

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 →

Single and Multithreaded Processes

single-threaded process   multithreaded process
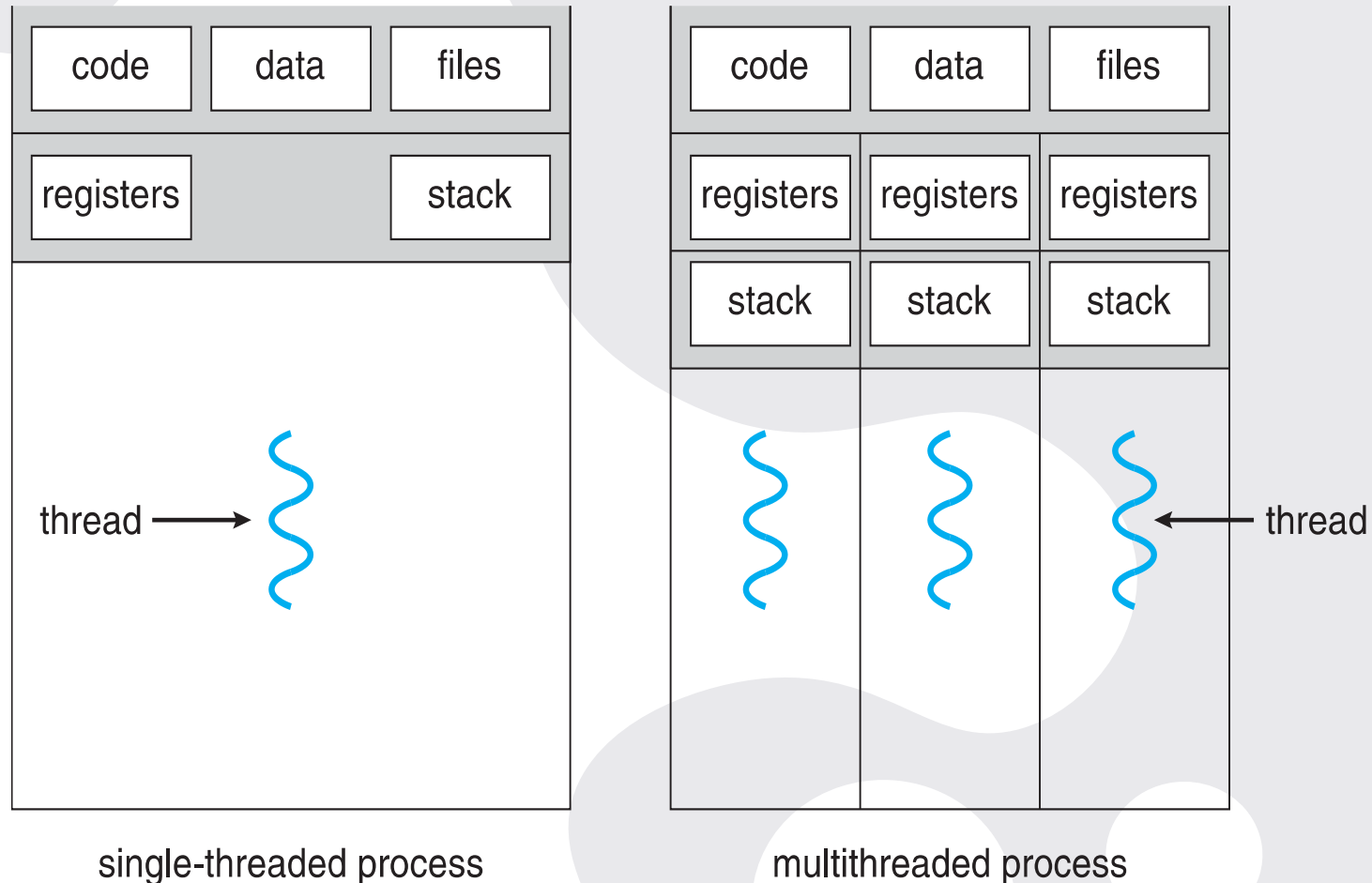
Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components

  - S is serial portion
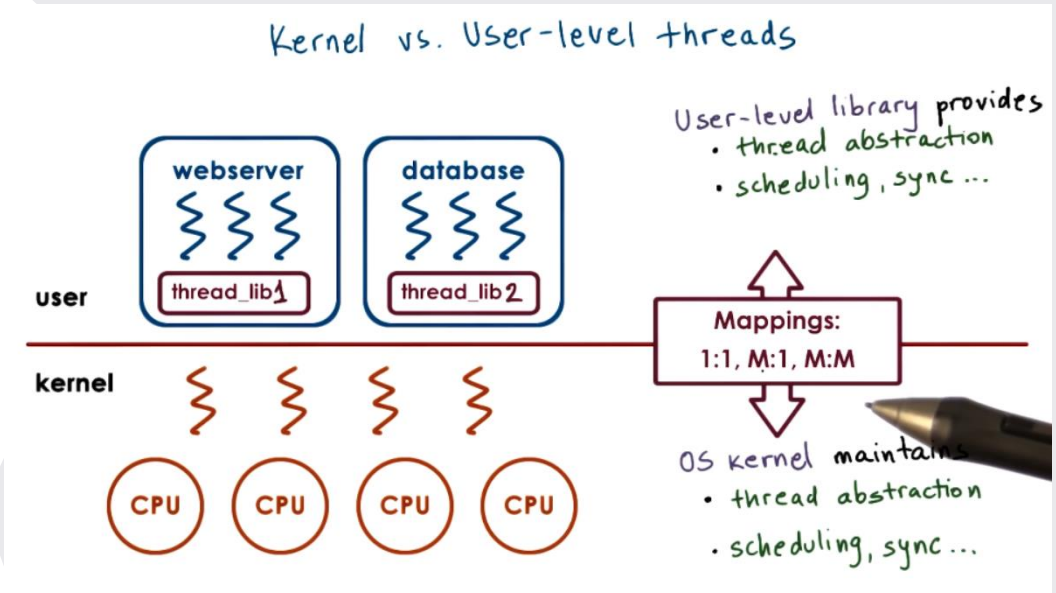 - N processing cores
  - That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
  - As N approaches infinity, speedup approaches 1 / S

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

- **"if 50% of the execution time is sequential, the maximum speedup is 2, no matter how many cores you use"**

- Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Amdahl's Law
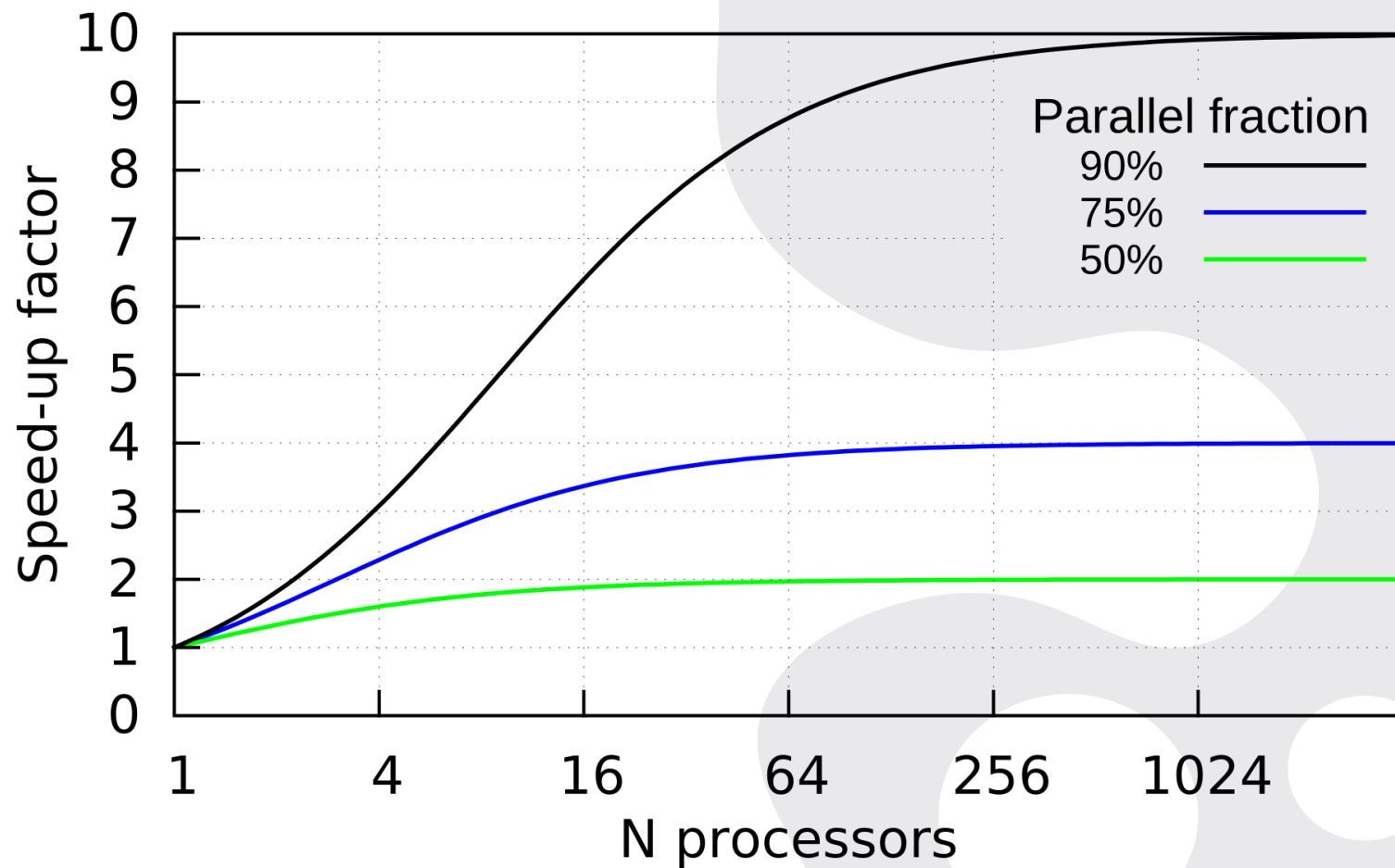
User Threads and Kernel Threads

- **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

Kernel vs. User-level threads

User-level library provides
- thread abstraction
- scheduling, sync ...

Mappings:
1:1, M:1, M:M

OS kernel maintains
- thread abstraction
- scheduling, sync ...

User vs Kernel Threads

| 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.
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.

User vs Kernel Threads

No	Parameters	User Level Thread	Kernel Level Thread
9	Operating System	Any operating system can support user-level threads.	Kernel level threads are operating system-specific.
10	Thread Management	The thread library contains the code for thread creation, message passing, thread scheduling, data transfer and thread destroying	The application code does not contain thread management code. It is merely an API to the kernel mode. The Windows operating system makes use of this feature.
11	Advantages	User Level Threads are simple and quick to create.	Scheduling of multiple threads that belong to same process on different processors is possible in kernel level threads.
		Can run on any operating system	Multithreading can be there for kernel routines.
		They perform better than kernel threads since they don't need to make system calls to create threads.	When a thread at the kernel level is halted, the kernel can schedule another thread for the same process.
		In user level threads, switching between threads does not need kernel mode privileges.	

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
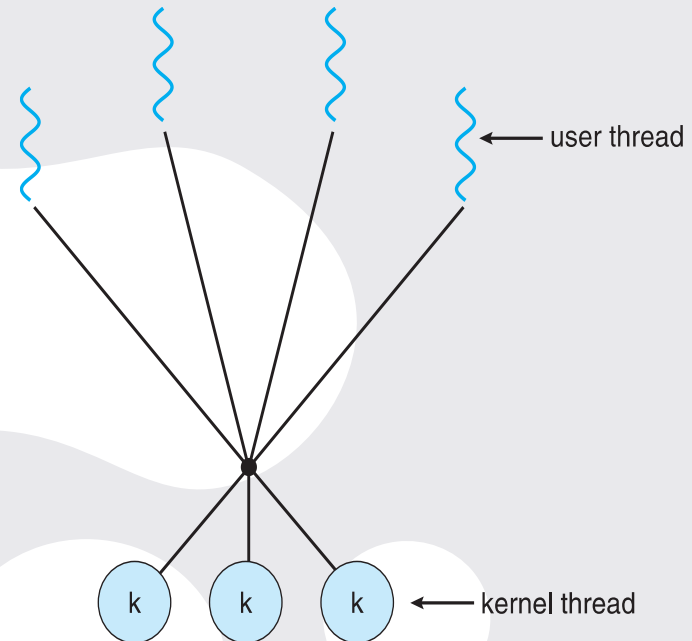
← user thread
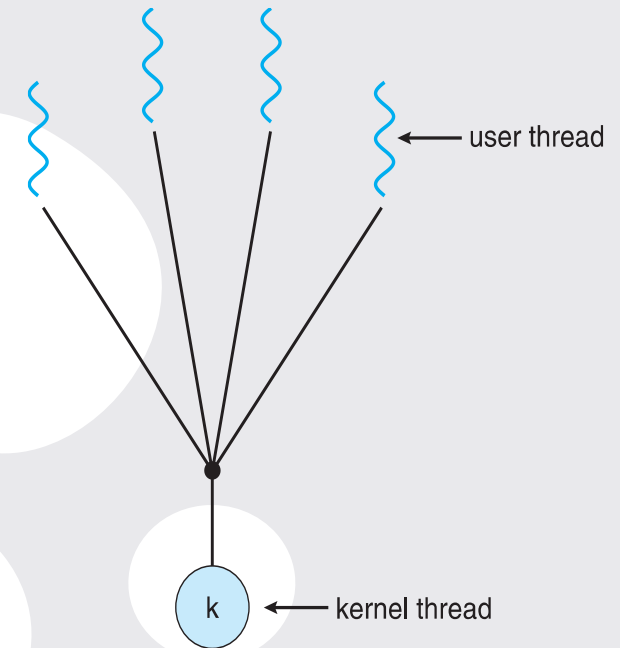
k ← kernel thread

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

←— user thread
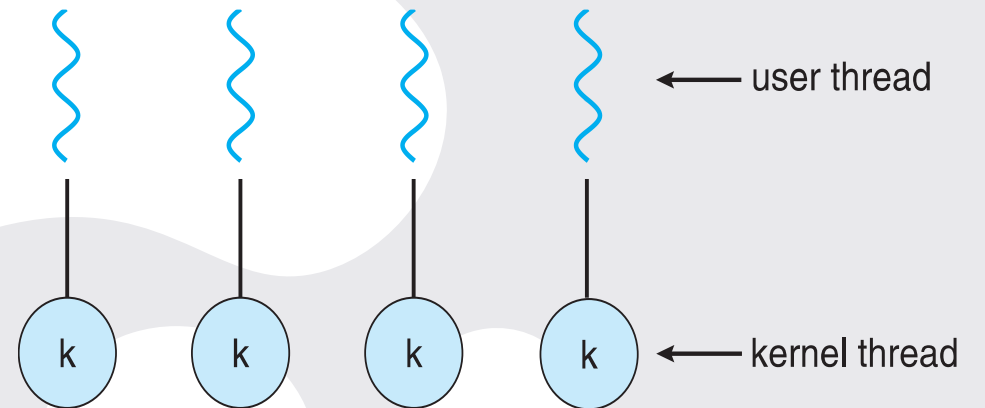
k   k   k   k   ←— kernel thread

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

user thread

kernel thread

k k k

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
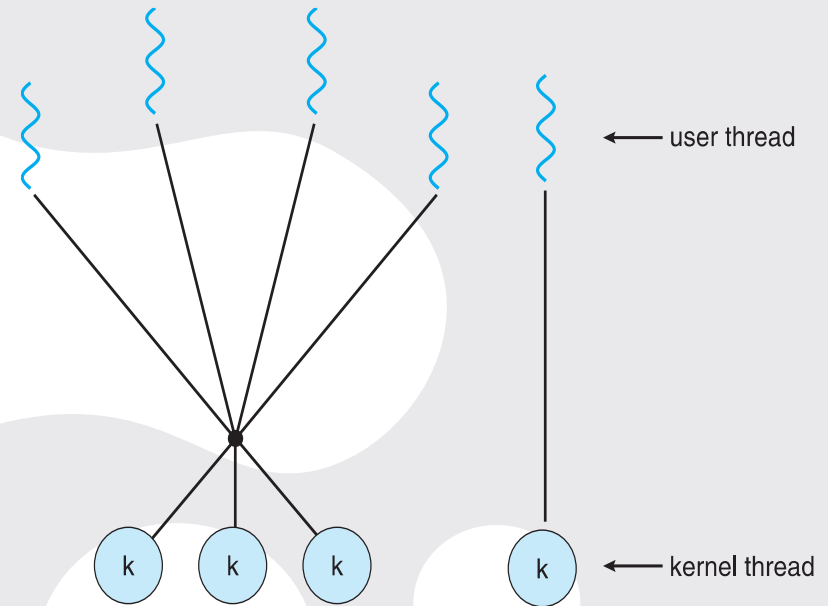
Thread Libraries

- 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

Pthreads

- 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)

Pthreads Example

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

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

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

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

Pthreads Example (Cont.)

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

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

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

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

    pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```c
#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

```c
#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.)

```c
/* 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

```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);
  }
}
```

Java Multithreaded Program (Cont.)

```java
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

- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch

- Other methods include Microsoft Threading Building Blocks (TBB), **java.util.concurrent** package

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

```c
#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

- Grand Central Dispatch (GCD or libdispatch), is a technology developed by Apple Inc. to optimize application support for systems with multi-core processors and other symmetric multiprocessing systems.

- It is an implementation of task parallelism based on the thread pool pattern.

- The fundamental idea is to move the management of the thread pool out of the hands of the developer, and closer to the operating system.

- The developer injects "work packages" into the pool oblivious of the pool's architecture. This model improves simplicity, portability and performance.

- GCD was first released with Mac OS X 10.6, and is also available with iOS 4 and above.

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; one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.

    - But which version of fork() and when?

- **exec()** usually works as normal – replace the running process including all threads

    - If exec() is called immediately after forking

        - Then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate

    - If the separates process does not call exec() after forking

        - Then separates process should duplicate all thread.

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: 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

- 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**

- On Linux systems, thread cancellation is handled through signals

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

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

- 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
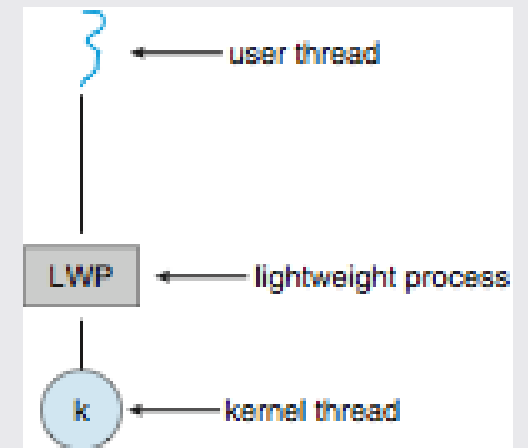
Windows Threads

- 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
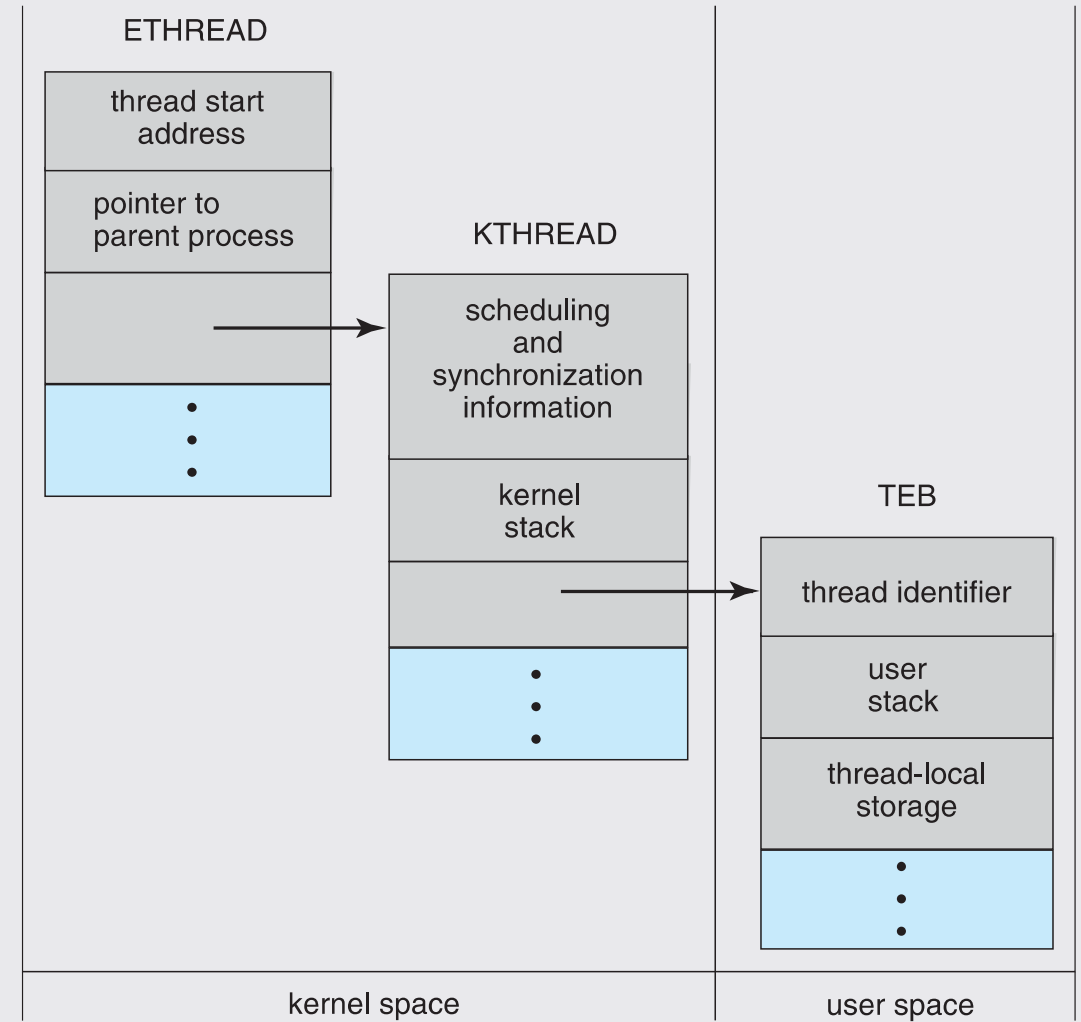
Linux Threads

- 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)

Reff.

Operating System Concepts Ninth Edition (Avi Silberschatz, Peter Baer Galvin, Greg Gagne)

Any
Question?