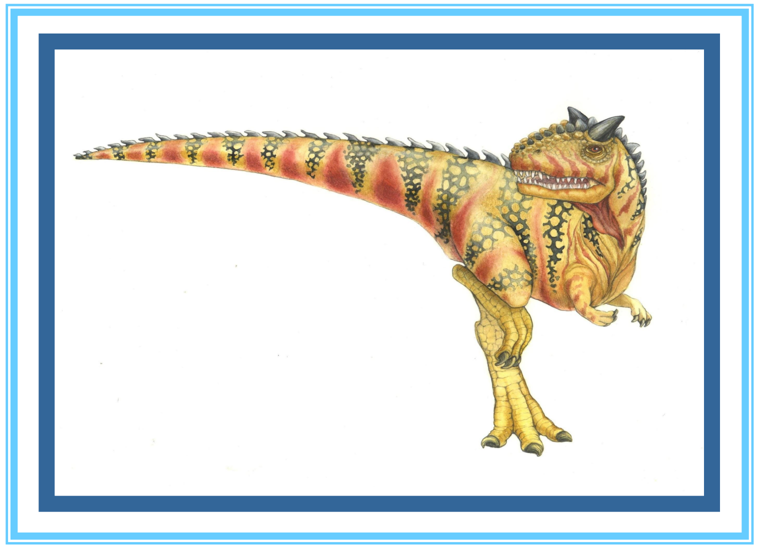
Chapter 4: Threads

Presented By

NARZU TARANNUM(NTR)



Chapter 4: Threads

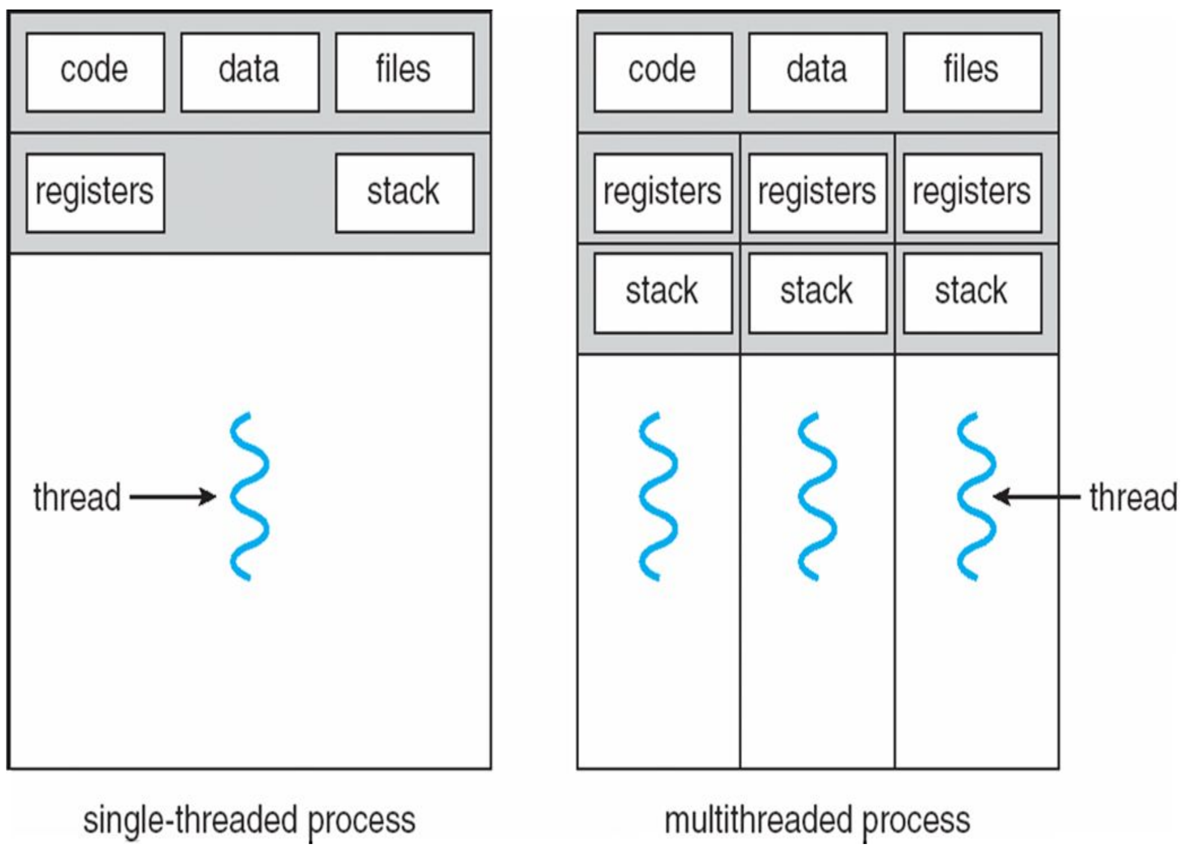
- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples

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
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Threads

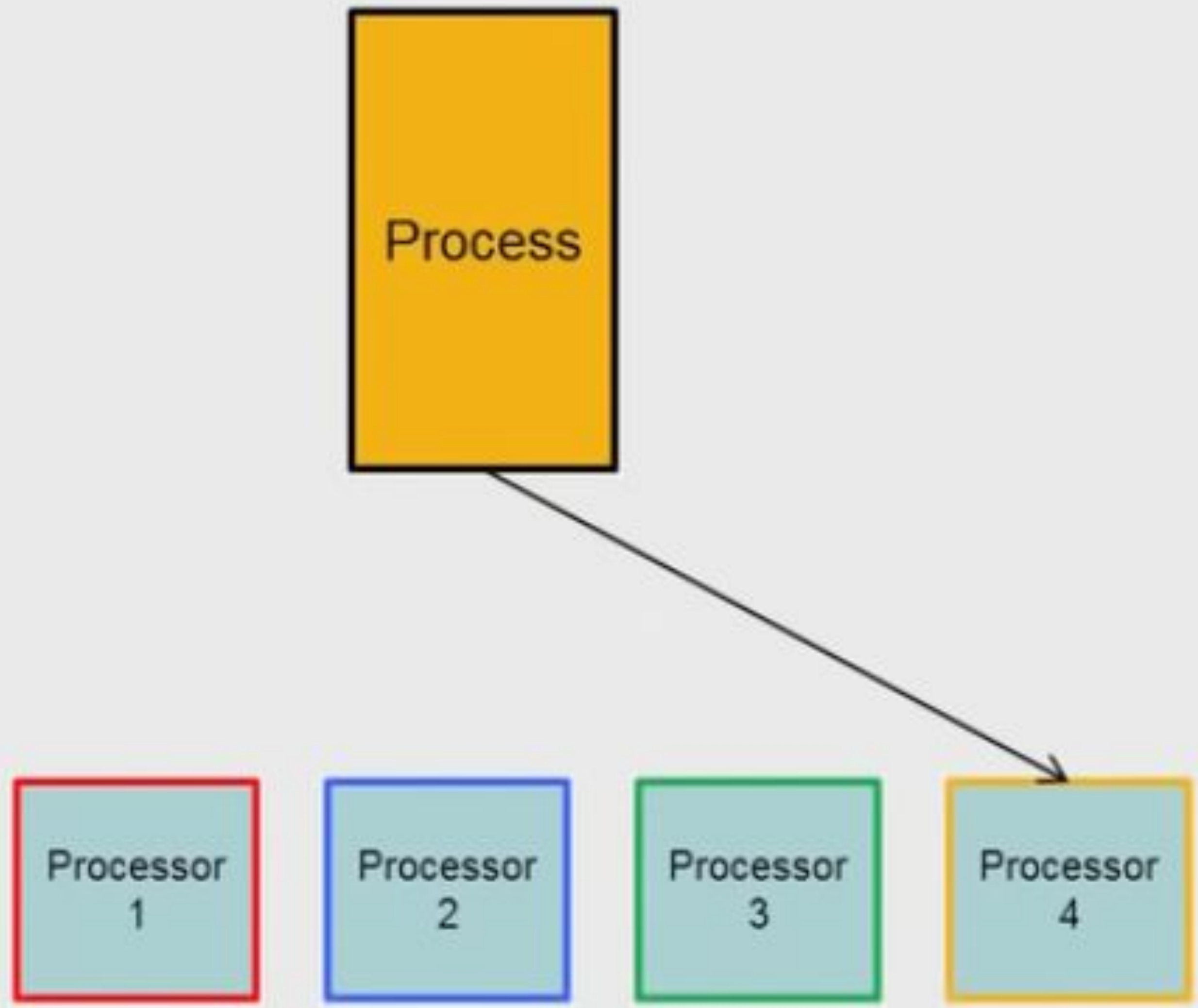
- A tread is a basic unit of CPU utilization.
- A traditional / heavyweight has a single thread of control.
- **If a process has multiple threads of control, it can perform more than one task at a time.**



- A thread contains -
  - Thread ID
  - Program Counter
  - Register Set
  - Stack
- Shares with other threads belonging to the same process -
  - Code Section
  - Data Section
  - OS resources

Consider this Example

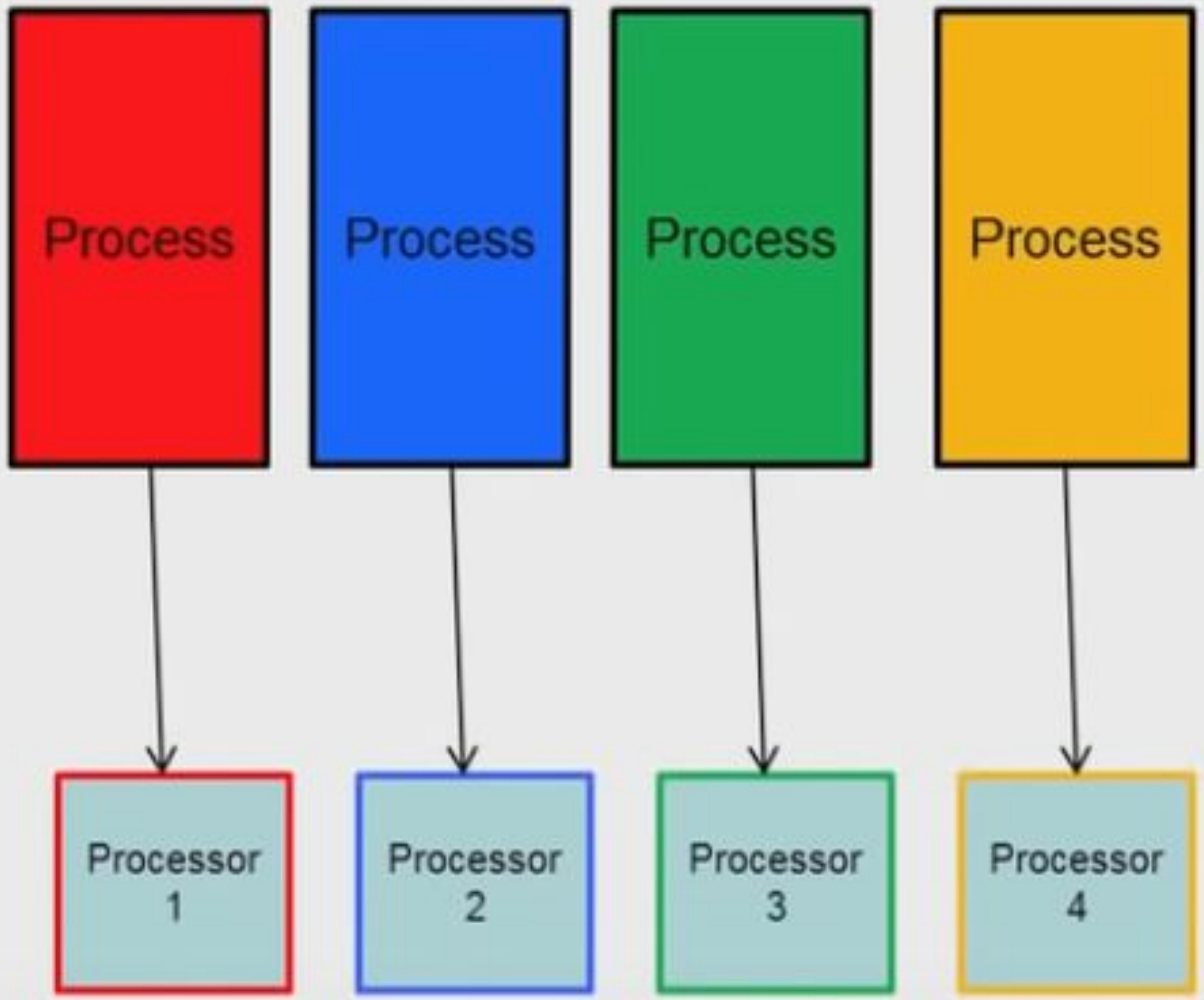
```c
#include <stdio.h>


unsigned long addall(){
  int i=0;
  unsigned long sum=0;

  while (i< 10000000){
     sum += i;
     i++;
  }
  return sum;
}


int main()
{
   unsigned long sum;
   srandom(time(NULL));
   sum = addall();
   printf("%lu\n", sum);
}
```



Speeding up with multiple processes

10000000 / 4 = 2500000

Create 4 processes, each loop does 1/4th of the work



Properties:

4 fork system calls needed; one for creating each process

Each process is isolated from each other

IPC mechanisms to communicate – more system calls

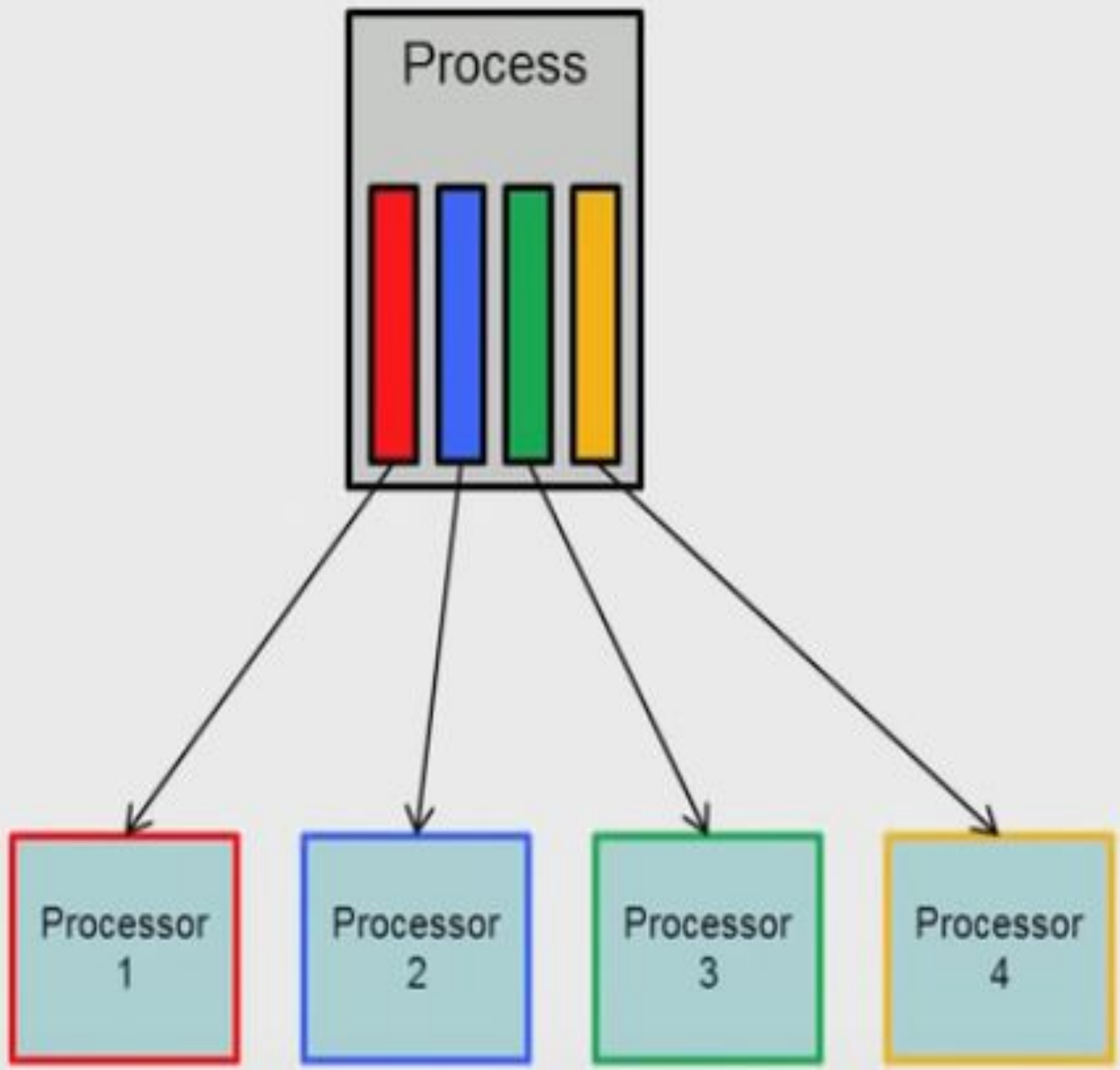
Process management with system calls

Each process has its own memory map – its own

Thread model

10000000 / 4 = 2500000

Create 1 process with 4 threads; each loop does 1/4th of the work



Properties:

1 fork system call needed; 4 threads need to be created --- much more lighter.

Each thread is not isolated from others

Management of threads with fewer or no system calls.

Threads vs Processes

- A thread has no data segment or heap
- A thread cannot live on its own. It needs to be attached to a process
- There can be more than one thread in a process. Each thread has its own stack
- If a thread dies, its stack is reclaimed

- A process has code, heap, stack, other segments
- A process has at-least one thread.
- Threads within a process share the same code, files.
- If a process dies, all threads die.

Benefits

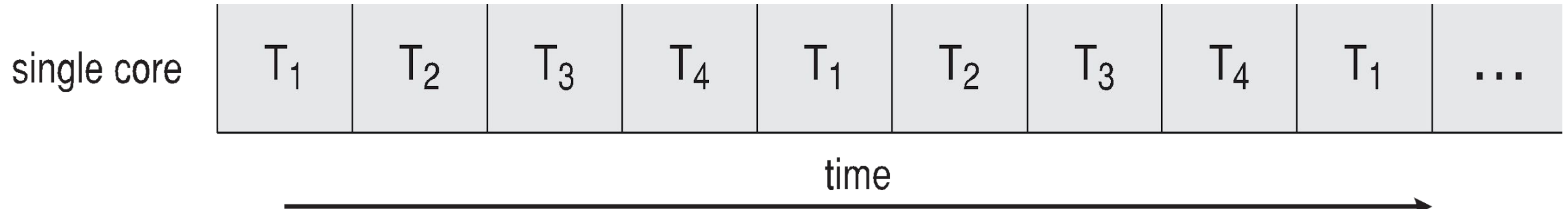
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower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

Multicore Programming

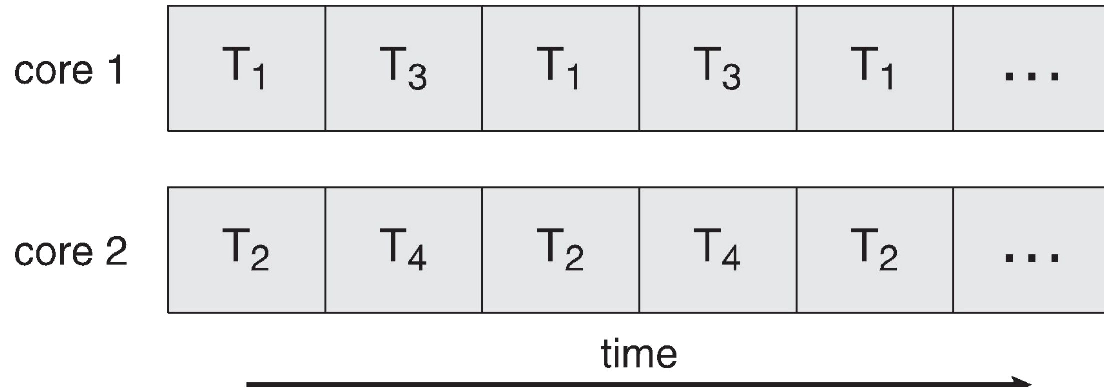
- In most recent design whether the cores appear across CPU chips or within CPU chips, we call these systems multicore or multiprocessor systems.
- Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improve concurrency.
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

Concurrency vs. Parallelism

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



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

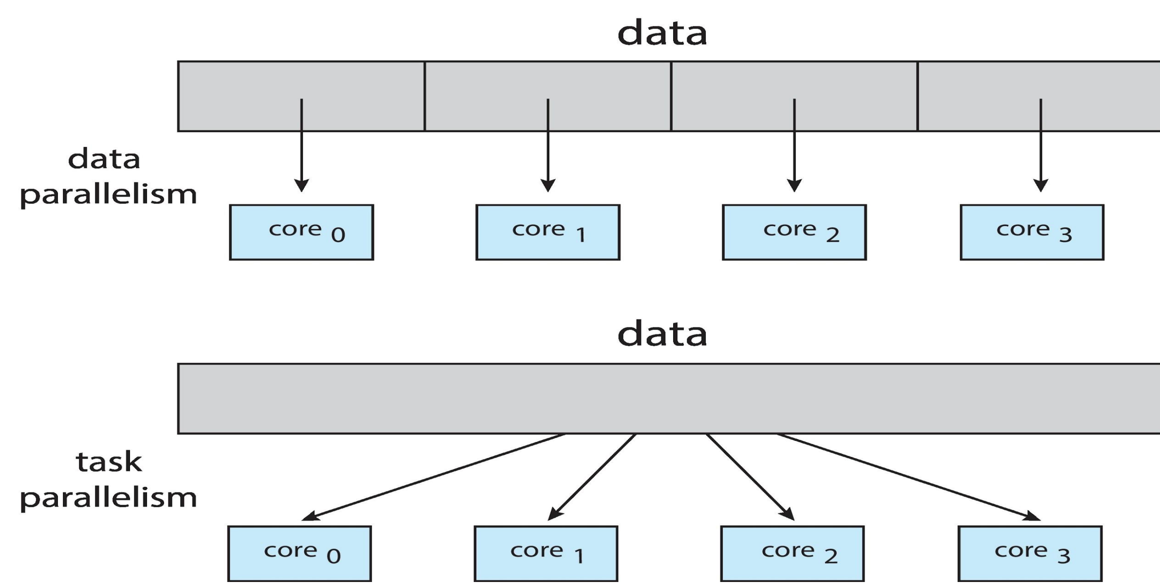


- ***Parallelism*** implies a system can perform more than one task simultaneously
  - Running multiple threads/processes in parallel over different CPU cores
- ***Concurrency*** supports more than one task making progress

  Single processor / core, scheduler providing concurrency
  - Running multiple threads/processes at the same time, even on single CPU core, by interleaving their execution.

Data and Task Parallelism

- Two Types of parallelism
  - **Data parallelism** – data parallelism is achieved when each processor performs the same task on different pieces of distributed data.
  - **Data parallelism** is a form of parallel computing for multiple processors using a technique for distributing the data across different parallel processor nodes.
  - Synchronous computation is performed.
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
  - Task Parallelism means concurrent execution of the different task on multiple computing cores on the same or different data.
  - Asynchronous computation is performed.



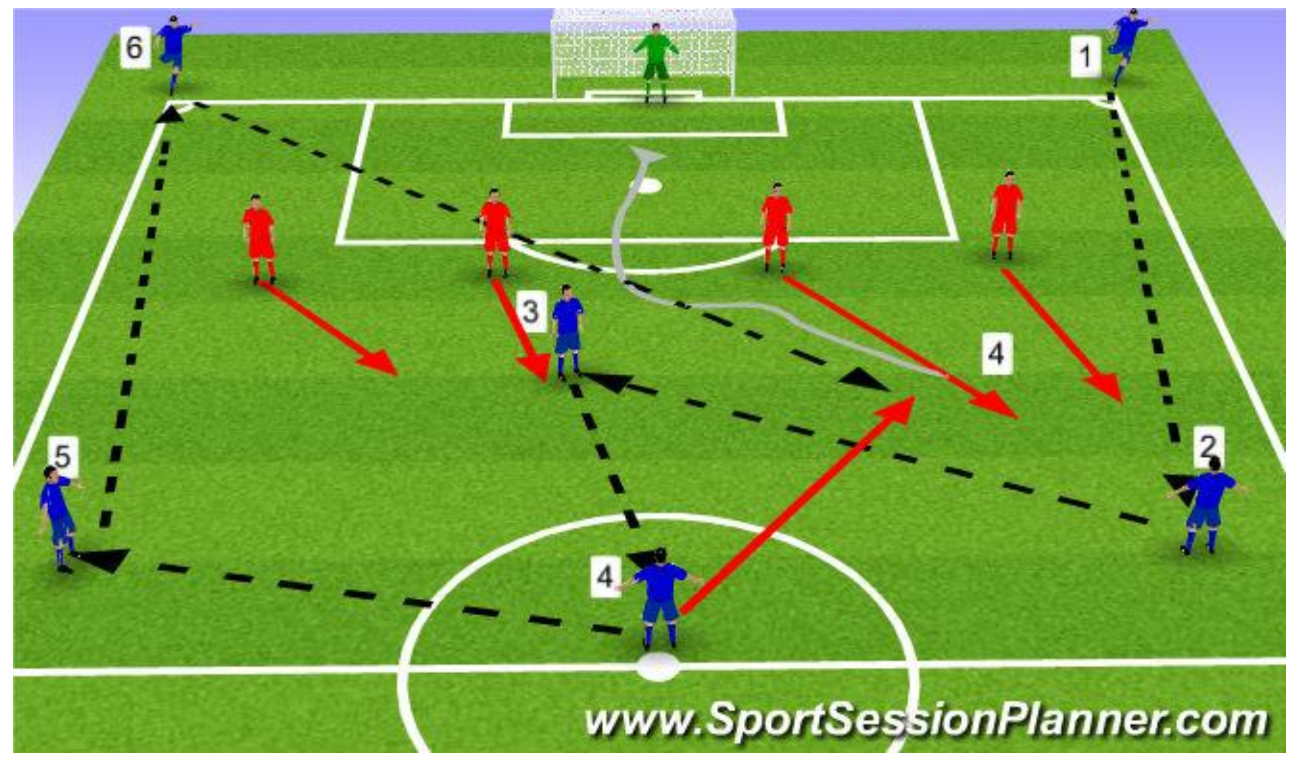
Data vs Task Parallelism

Data Parallelism

- Single Instruction Multiple data
- Multiple workers all doing same things

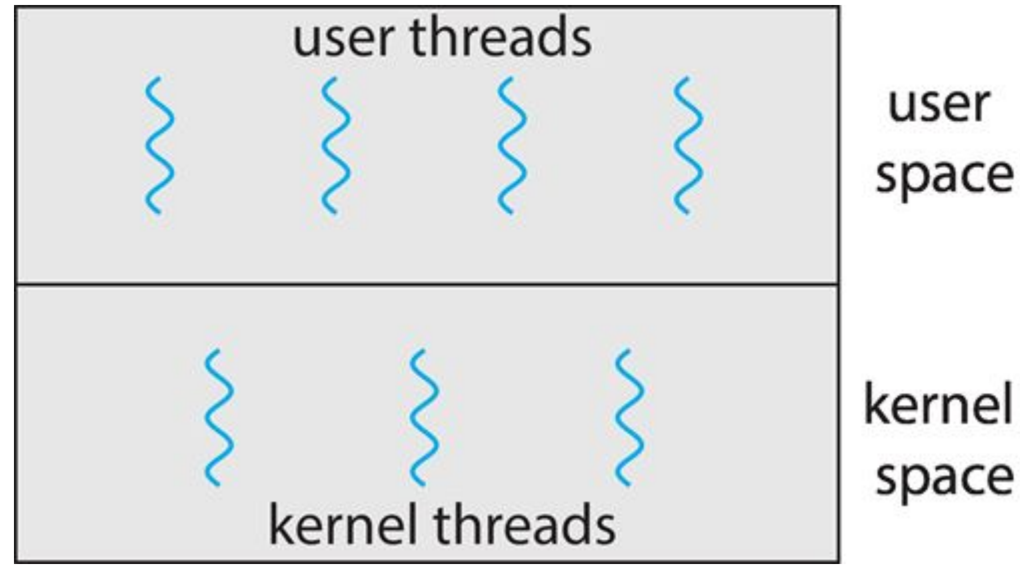
Task Parallelism

- Multiple instruction same data or Multiple Instruction Multiple Data
- Works with same objective



Types of Threads

- There are two types of threads to be managed in a modern system: User threads and kernel threads.
- In a specific implementation, the user threads must be mapped to kernel threads.



User Threads

Thread management done by user-level threads library
User Threads are supported above the Kernel and are managed without kernel support.
Kernel knows nothing about User thread

Kernel Threads

Thread directed supported and managed by the OS
Many OS kernels are now multi-threaded; several threads operate in the kernel and each thread performs a specific task. Kernel level threads, allowing the kernel to perform multiple simultaneous tasks and to service multiple kernel system calls simultaneously.

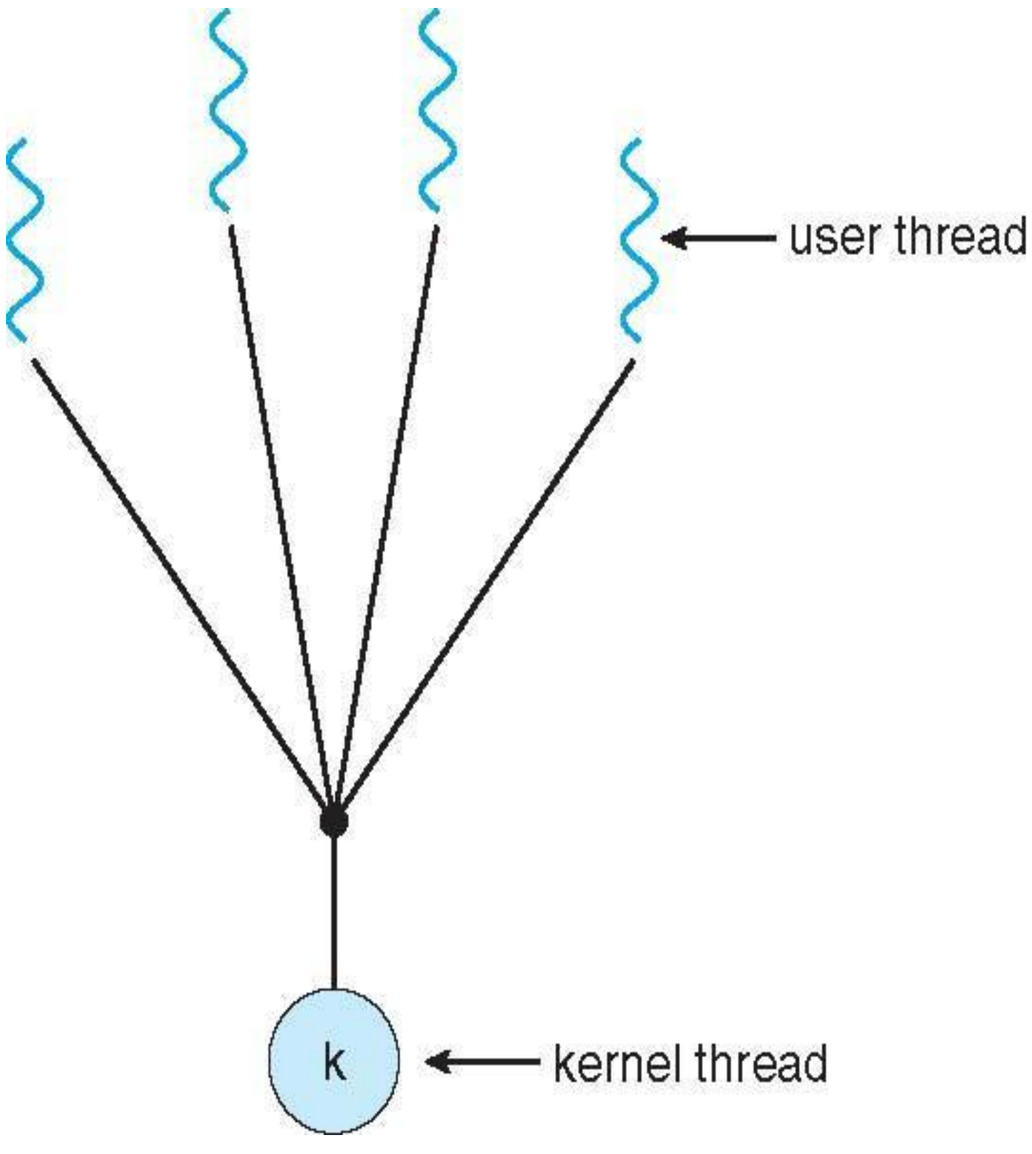
For functioning both types of threads there must exists a relationship between them.

Multithreading Models

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

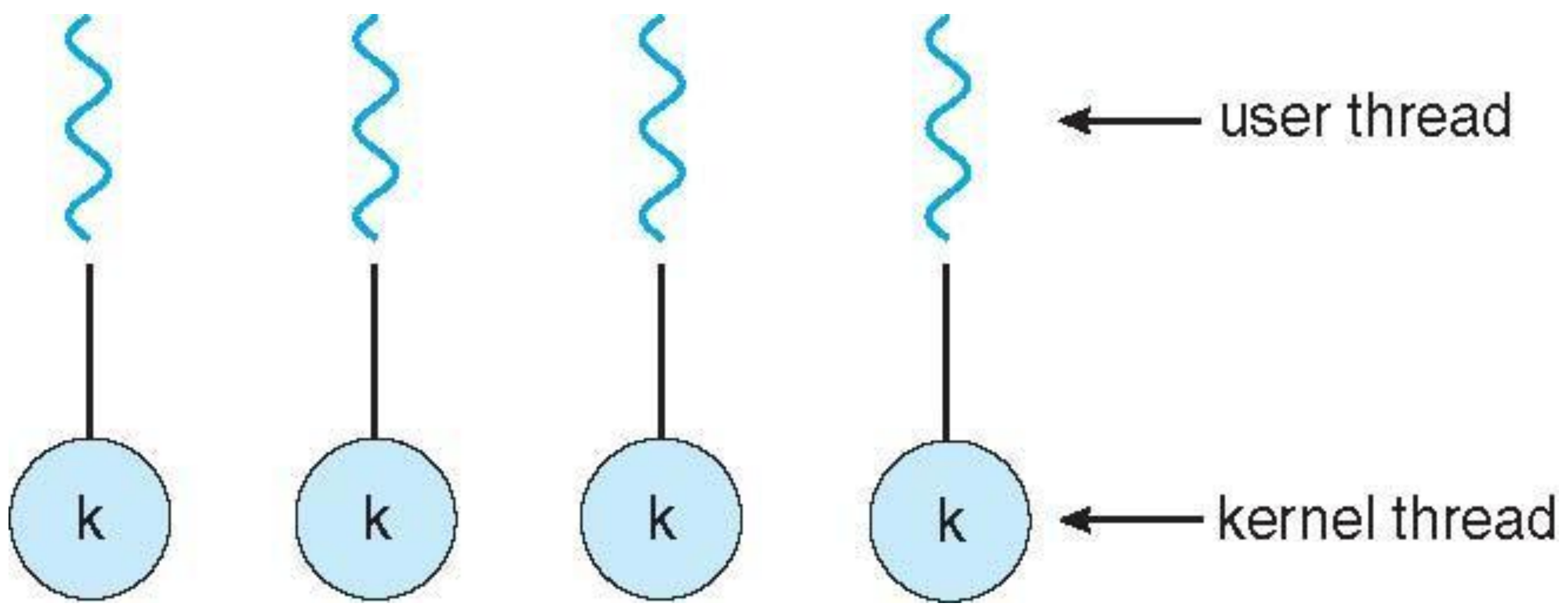
Many-to-One Model

- Many user-level threads mapped to single kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- If one thread makes a blocking system call causes the entire process block.
- Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



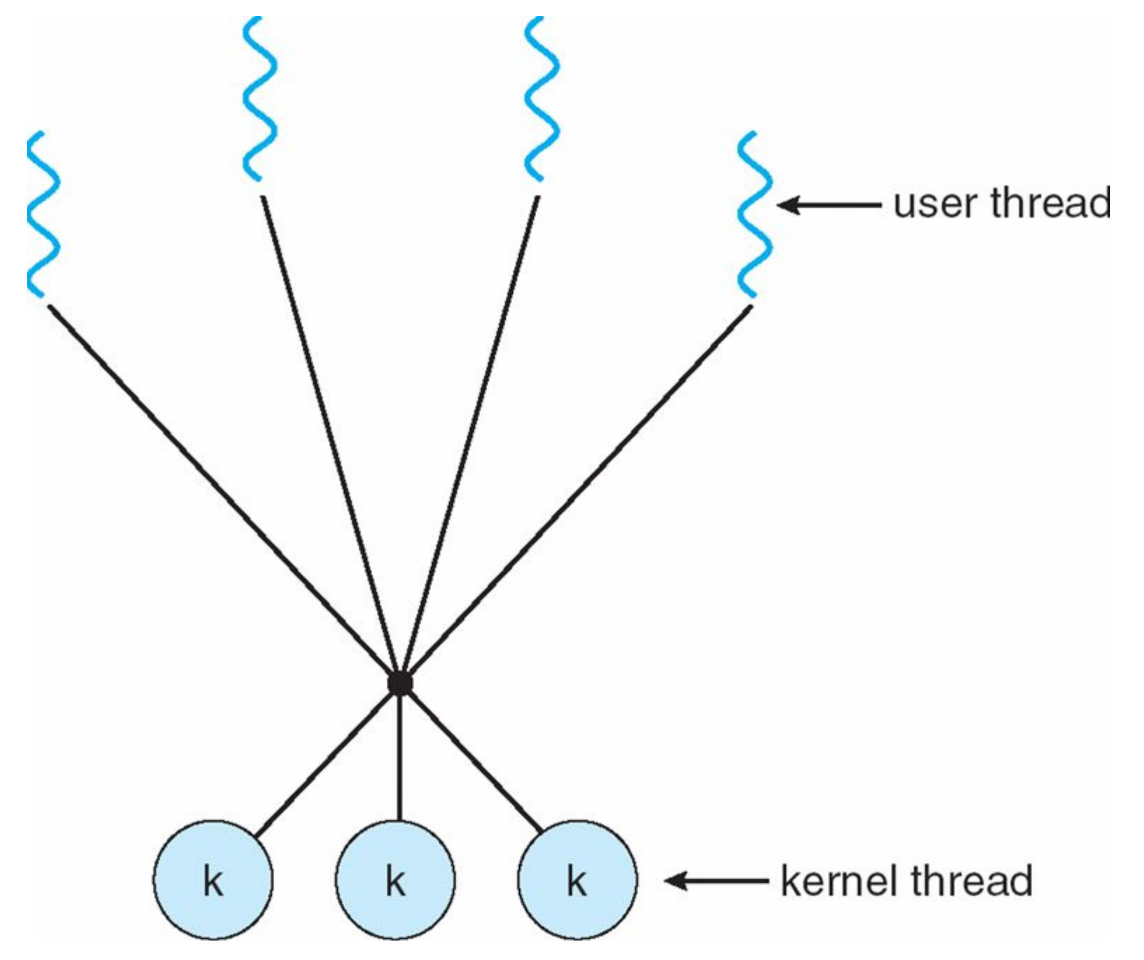
One-to-one Model

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- Provides more concurrency than the many to one model by allowing another thread to run when a thread makes a blocking system call
- Also allows multiple-threads to run in parallel on multi-processors
- Creating a user thread requires creating the corresponding kernel thread
- Because the overhead of creating Kernel threads can Barden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Examples: Windows, Linux, Solaris 9 and later



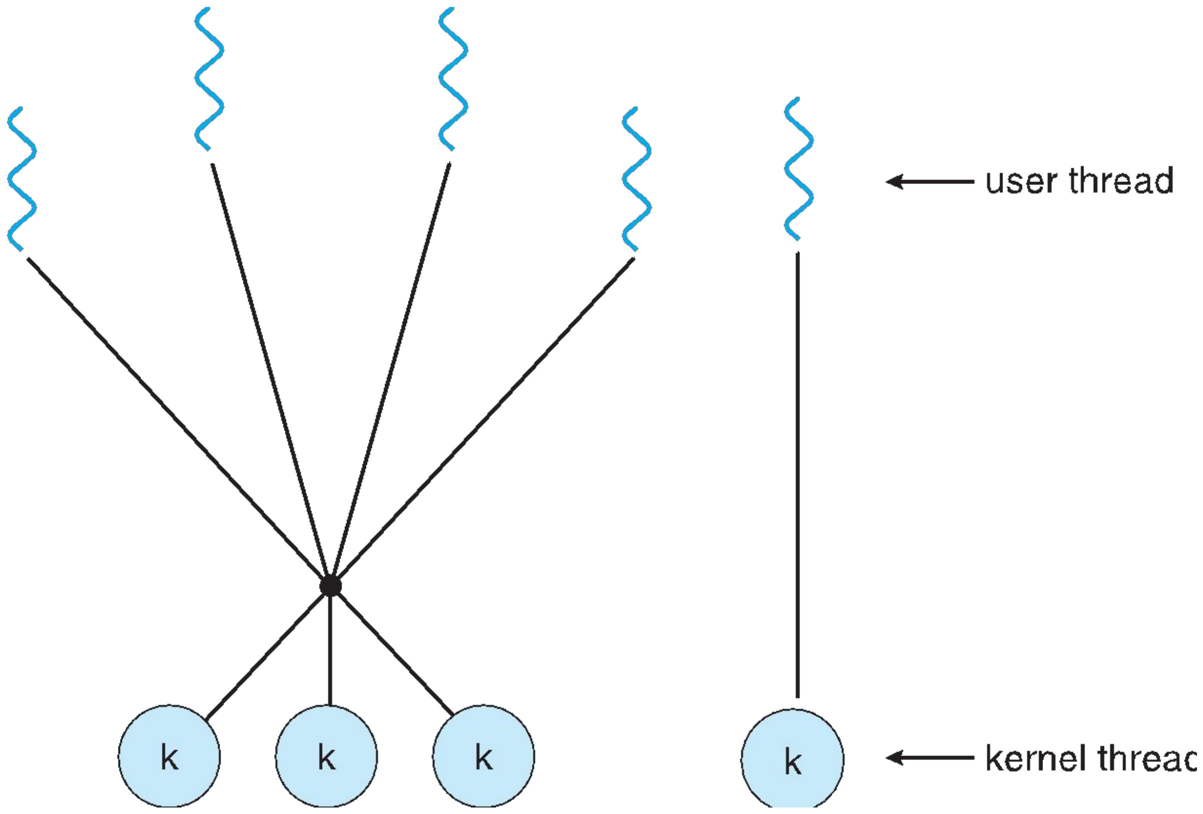
Many-to-Many Model

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



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 a thread library:
  - First approach is to implement a user level library with no kernel support. All code and data structures for the library (threads) exist in user space.
    - This means that invoking a function in a library results in a local function calling user space that is not a system call (library entirely in user space) .
  - Second approach is to implement a kernel level library supported directly by the OS. In this case code and data structures for the library (threads) exist in kernel space (library supported by OS kernel level) .
    - This means that invoking a function in the library results in a system call to the kernel.

- There are three main thread libraries in use today:
  - POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
  - Win32 threads - provided as a kernel-level library on Windows systems.
  - 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.

Thread Libraries

- P threads
  - pthreads, is the extension of the POSIX standard may be provided either as user-level or kernel-level library, which defining an API for thread creation and synchronization.
  - The POSIX standard (IEEE 1003.1c) defines the specification for pThreads, not the implementation It is up to developer or OS designer of the library.
  - Common in UNIX operating systems (Solaris, Linux, Mac OS X)
  - Global variables are shared amongst all threads.
  - One thread can wait for the others to rejoin before continuing.
  - p Threads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.

- Win32
  - Win32 is a kernel level library available in windows system
  - These threads are created in the Win32 API using the ***Create Thread()***
  - Just as Pthreads, a set of attributes or parameters is passed to this function

Thread Libraries

- Java Threads
  - Java threads API (library) allow JAVA programs directly to creates and manage the threads
  - Typically implemented using the threads model provided by underlying OS.
  - Java threads are managed by the JVM
  - Even in single JAVA program which contains only a ***main()*** method runs as a single thread as the JVM
  - There are two techniques for crating threads in JAVA program
    - One approach is to create a new class that is derived from the thread class and to override its ***run()*** method
    - An alternative and more commonly used technique is to define a class that implements the ***Runnable interface***. The runnable interface is defined as follows:

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

pthread library

- Create a thread in a process

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

Thread identifier (TID) much like

Pointer to a function, which starts execution in a different thread

Arguments to the function

- Destroying a thread

```
void pthread_exit(void *retval);
```

Exit value of the thread

- Join : Wait for a specific thread to complete

```
int pthread_join(pthread_t thread, void **retval);
```

TID of the thread to wait for

Exit status of the thread

Example

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

unsigned long sum[4];

void *thread_fn(void *arg){
  long id = (long) arg;
  int start = id * 2500000;
  int i=0;

  while(i < 2500000){
     sum[id] += (i + start);
     i++;
  }
  return NULL;
}

int main(){
  pthread_t t1, t2, t3, t4;

  pthread_create(&t1, NULL, thread_fn, (void *)0);
  pthread_create(&t2, NULL, thread_fn, (void *)1);
  pthread_create(&t3, NULL, thread_fn, (void *)2);
  pthread_create(&t4, NULL, thread_fn, (void *)3);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  pthread_join(t4, NULL);
  printf("%lu\n", sum[0] + sum[1] + sum[2] + sum[3]);
  return 0;
}
```

Create Thread using pthread in C:

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

void *funcThread(void *arg);
int main(){
    pthread_t t1;
    pthread_create(&t1,NULL,funcThread,NULL);
    pthread_join(t1,NULL);

    return 0;
}
void *funcThread(void *arg){
    printf("Entered thread:\n");
    for(int i=0;i<3;i++){
        printf("thread: %d\n",i);
    }
    printf("Done with thread ....\n");
}
```

Output:

```
Entered thread:
thread: 0
thread: 1
thread: 2
Done with thread ....
```

Return values from thread function using pthread in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
int *func_thread(int *v);
int *t_ret;
int main(){
    pthread_t t1;
    int n=5;
    pthread_create(&t1,NULL,func_thread,&n);
    pthread_join(t1,&t_ret);
    printf("Thread returned: %d\n",t_ret);

    return 0;
}
int *func_thread(int *v){
     *v=*v*5;
    return *v;
}
```

Output

```
Thread returned: 25
```

Return values from thread function by cancelling the thread using pthread in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
void *func_thread(int *n);
void *t_ret;
int num=5;
int main(){
    pthread_t t1;
    pthread_create(&t1,NULL,(void *)func_thread,&num);
    pthread_join(t1,&t_ret);
    printf("Thread returned: %d\n",(int *)t_ret);

    return 0;
}

void *func_thread(int *n){
    printf("Entered in Thread:\n");
    if(*n % 2==0){
        pthread_exit(*n * *n);
        printf("Operation completed\n");
    }
    else{
        pthread_exit(*n * *n * *n);
        printf("Operation completed\n");
    }
}
```

Output

```
Entered in Thread:
Thread returned: 125
```

Implicit Threading

- Implicit multi-threading is concurrent execution of multiple threads extracted from single sequential program.
- Shifts the burden of addressing the programming challenges outlined earlier (Dividing activities, Balance, Data splitting, Data dependency, Testing and debugging) , from the application programmer to the compiler and run-time libraries.
- 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
- Five methods explored:
    - Thread Pools
    - Fork-Join
    - OpenMP
    - Grand Central Dispatch
    - Intel Threading Building Blocks

Thread Pools

- If a web server receives a request, it creates a separate tread to serve the request. The issue arise:
  - Thread creation time
  - No bound on concurrent execution of threads
- **Thread pools:**
- Create a number of threads in a pool where they await for 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
- Windows API supports thread pools:

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

Threading Issues

- ***fork() and exec() System Calls:*** *Duplicate all the threads or not?*
- ***Thread cancellation:*** *Thread cancellation is the task of terminating a thread before it has completed.*
- ***Signal Handling:*** *Where should a signal be delivered?*
- ***Thread Pool:*** *Create a number of threads at the process start-up.*
- ***Thread Specific data:*** *Each thread might need it's own copy of certain data.*

The fork() and exec() system calls

- fork()
  - The fork() system call is used to create a separate, duplicate process
- exec()
  - When a exec() system call is invoked, the program specified in the parameter to exec() will replace the entire process –including all threads

- **Issue:** If one thread in a program calls fork(), does the new process duplicate all threads or the new process single-threaded?
- **Solution:** Some UNIX systems have chosen to 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() to use and when?

Which version of fork() will to be used depends on the application.

- If exec() is called immediately after forking then duplicating all threads is unnecessary.
- If the separate process does not call exec() after forking, the separate process should duplicate all threads

Thread Cancellation

- Thread cancellation is the task of Terminating a thread before it has completed.
- Thread to be canceled is **target thread**
- Two general approaches:
- **Asynchronous cancellation:** one thread terminates the target thread immediately.
  - There is an issue if a thread is cancelled while in the midst of updating data it is sharing with other threads.
- **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
  - Cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not. So it should be canceled at a point when it can be cancelled safely.

Signal handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- Synchronous and asynchronous received signal.
- A **signal handler** is used to process signals
  1. A signal is generated by the occurrence of a particular event
  2. A generated signal is delivered to a process
  3. Once delivered, the signal must be handled
- Every Signal may be handled by one of two possible handlers:
  1. A default signal handler
  2. A user-defined signal handler

Signal handling

- Signals are always delivered to a process but delivering signals is more complicated in multi-threaded program. In that case following options exist
  1. Deliver the signal to the thread to which the signal applies (Synchronous cancellation)
  2. Deliver the signal to every thread in the process (Asynchronous cancellation)
  3. Deliver the signal to certain threads in the process
  4. Assign a specific thread to receive all signals for the process

Thread Local Storage

- Threads belonging to a process share the data of the process. In some circumstances, each thread might need its own copy of certain data.

  - Create Facility needed for data private to thread is called **Thread-local storage** (**TLS**) or thread-specific data
  - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

End of Chapter 4