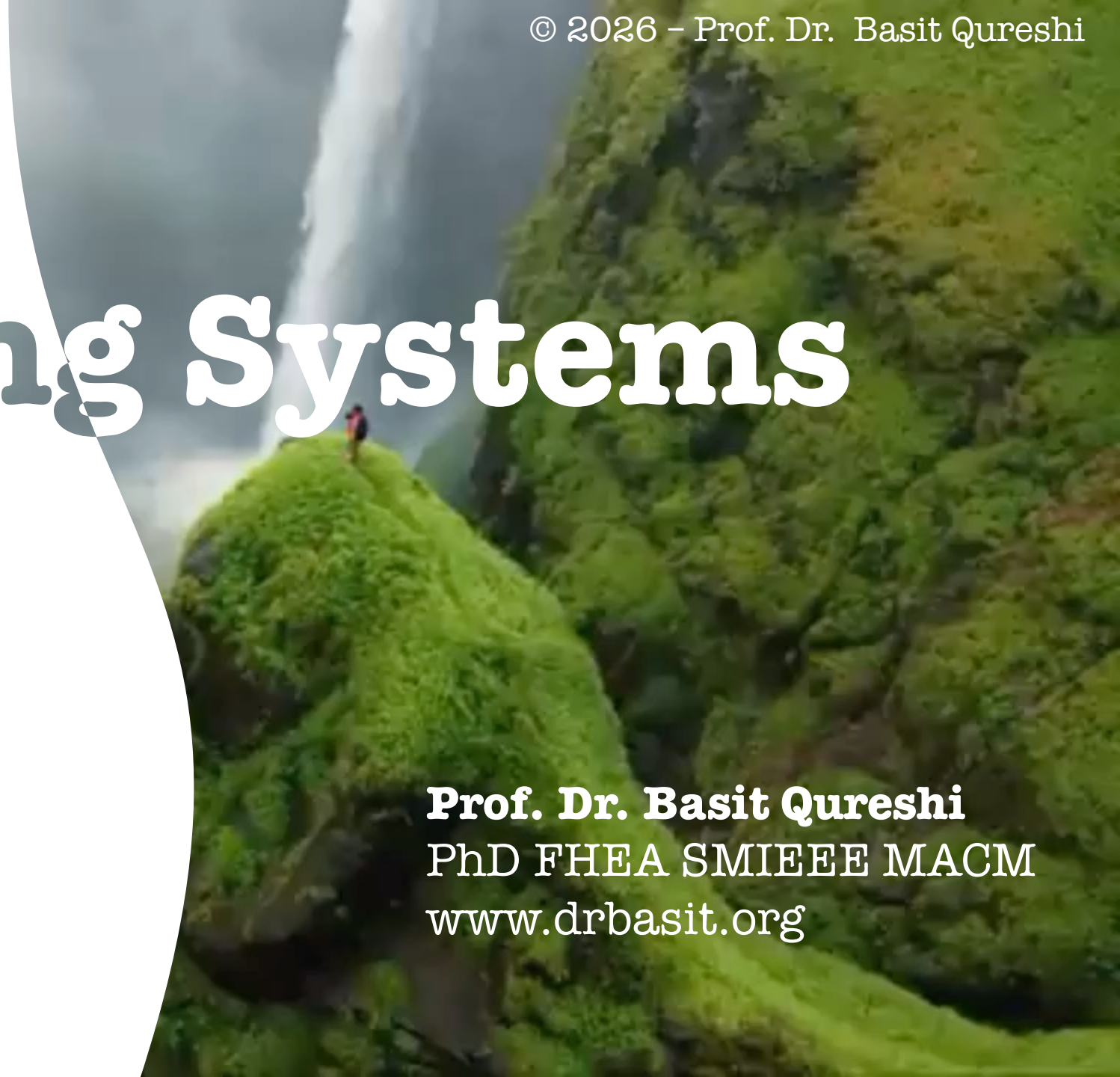
CS330 Operating Systems



THREADS

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

جامعة الأمير سلطان
PRINCE SULTAN UNIVERSITY

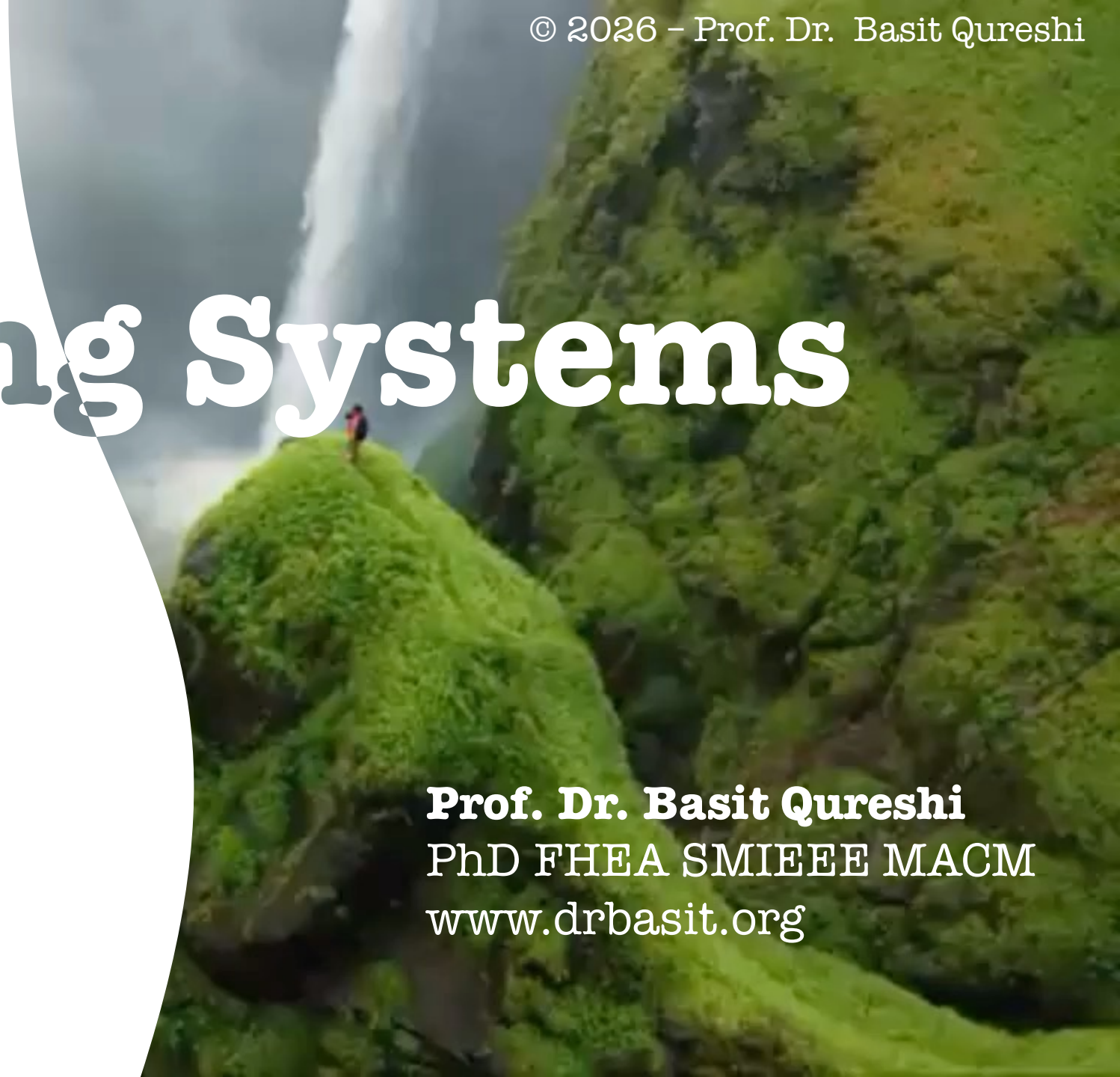
TOPICS

- Threads concept
- Types of threads
- Pthread library

Credit:
These notes are extracted from the following resources:

- Silberschatz, Galvin and Gagne, *Operating System Concepts*, 10 ed, Wiley, 2018

CS330 Operating Systems



THREADS CONCEPT

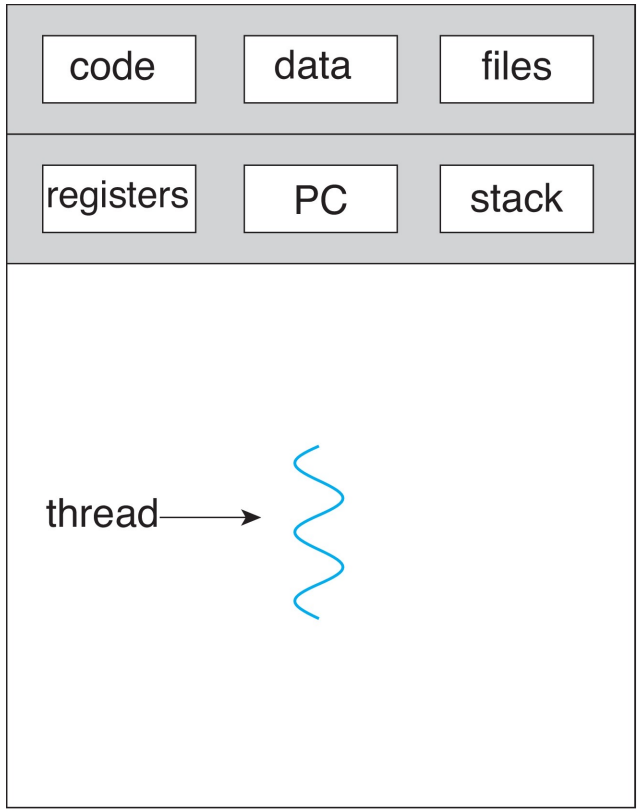
Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

جامعة الأمير سلطان
PRINCE SULTAN UNIVERSITY

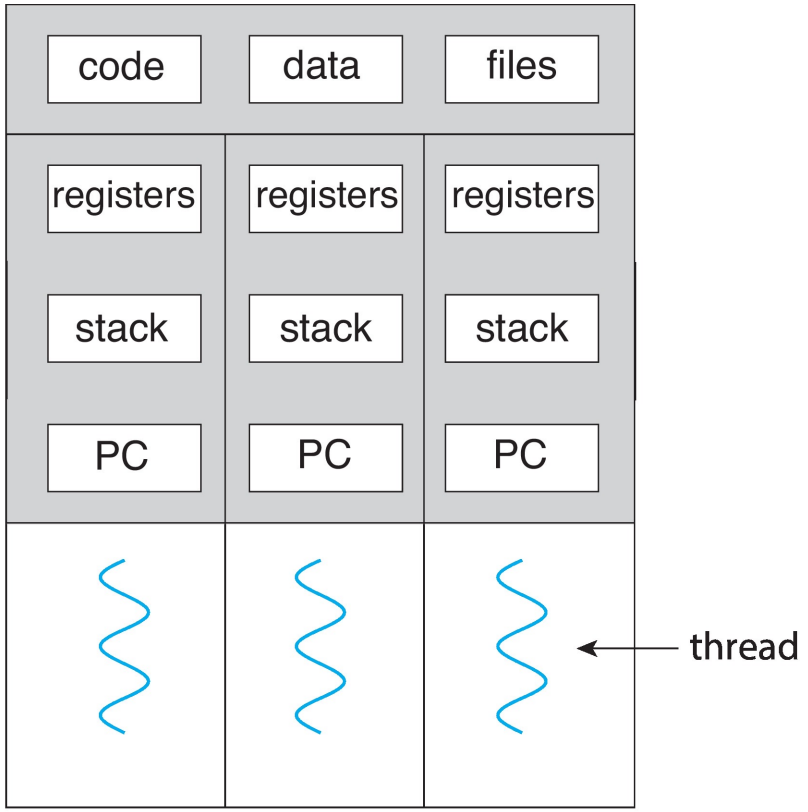
THREADS

- *Heavyweight process*: Traditional processes that can have only one thread
- *Multi-threaded processes:* A process that includes multiple threads that share the resources allocated to the process
- *Lightweight process:* another name for a ***thread,*** is a basic unit of CPU utilization; it is comprised of:
  - A thread ID
  - program counter
  - register set
  - stack space
- A thread shares with other threads in the same process its:
  - code section
  - data section
  - O/S resources

SINGLE AND MULTITHREADED PROCESSES



single-threaded process



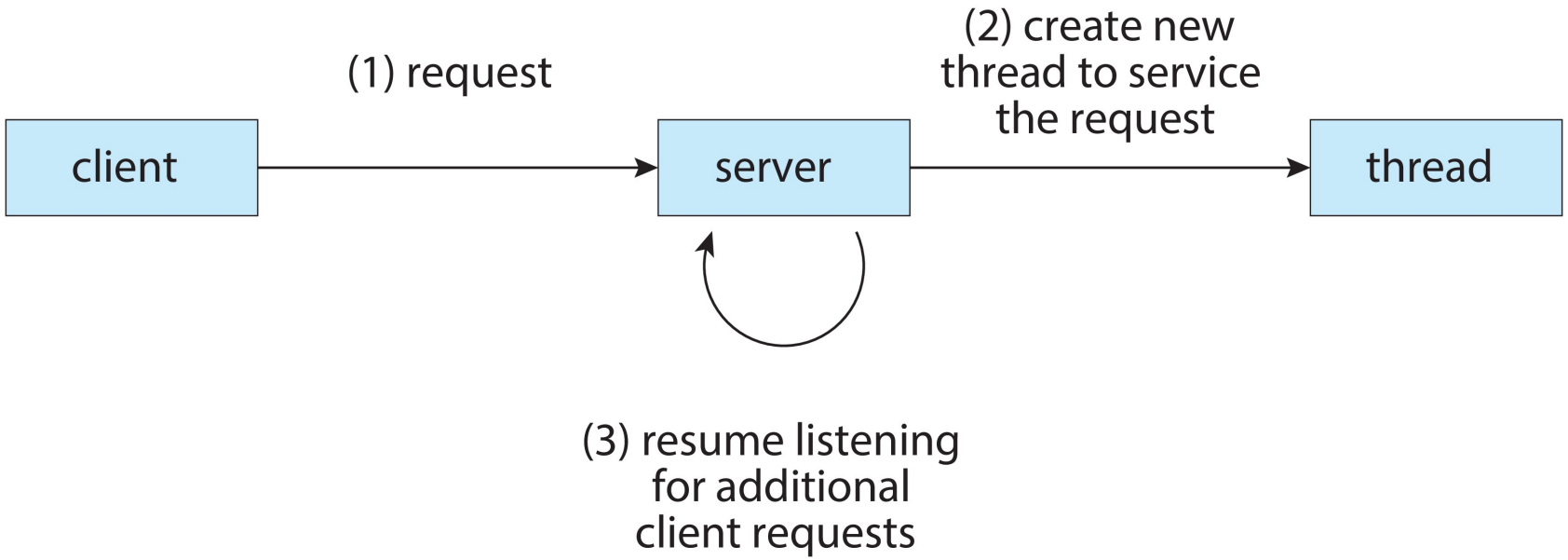
multithreaded process

EXAMPLES OF USING THREADS

Threads are important for any application with multiple tasks that can be run with separate threads of control.

- **A web server could produce a thread for each client**
  - Serve clients with multiple threads concurrently .
  - less overhead to use multiple threads than to use multiple processes.
- **A Word processor may have separate threads for:**
  - Reading user input
  - Display graphics
  - Performing Spell & grammar check

MULTITHREADED SERVER ARCHITECTURE



- Server will create a separate thread ------→listens for client requests.
- When a request is made, rather than creating another process, ---------->the server will create a new thread to service the request & resume listening for additional requests.

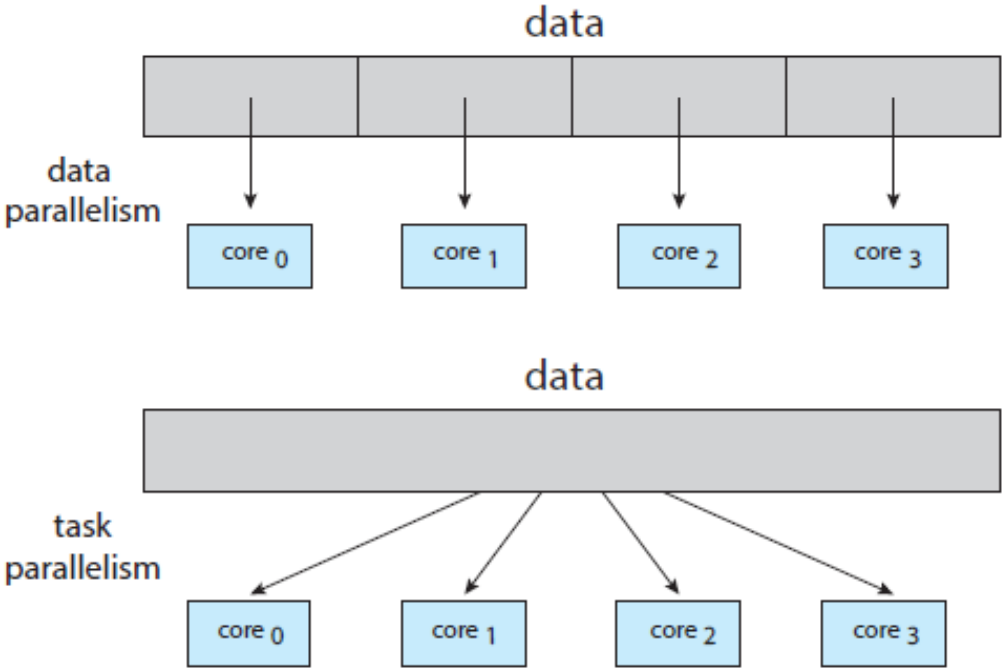
BENEFITS OF THREADS

- **Responsiveness:**
  - Threads allow a program to continue running even if part of it is blocked/ performing a lengthy operation. e.g: a multithreaded web browser can allow user interaction in 1 thread while loading an image in another thread.
- **Resource Sharing:**
  - Threads share memory and resources of the process they belong to.
- **Economy:**
  - Allocating memory and resources to a process is costly- threads share resources of the process**.**
  - Threads are **faster** to create and to **switch** between them than processes .(i.e. in Solaris, creating P is 30 times slower than creating thread).
- **Scalability- Utilization of Multiprocessor architectures:**
  - Multiple threads can run in parallel on different CPUs
  - A single threaded process can run only on 1 CPU no matter how many are available.
- Thread management could be done at either
  - User level
  - Kernel level

MULTICORE PROGRAMMING *CONT.*

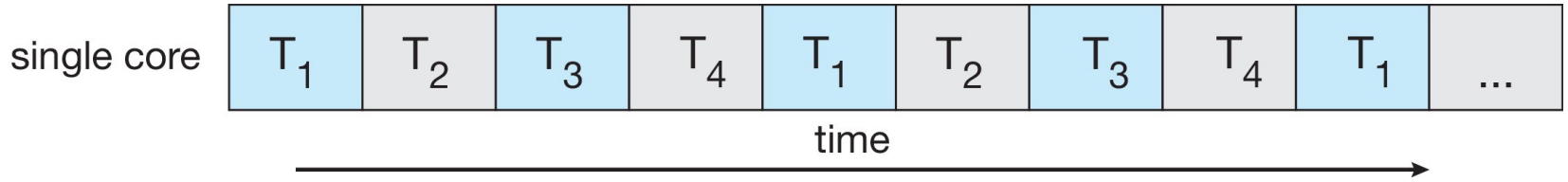
Models of multiprogramming:

- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task by allowing all of them to make progress
- 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

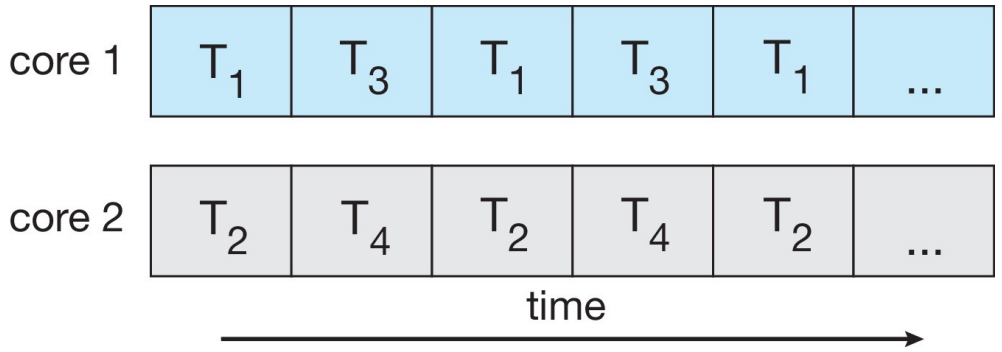


CONCURRENCY VS. PARALLELISM

- **Concurrent execution on single-core system:**
  - Concurrency means that execution of threads will be interleaved over time



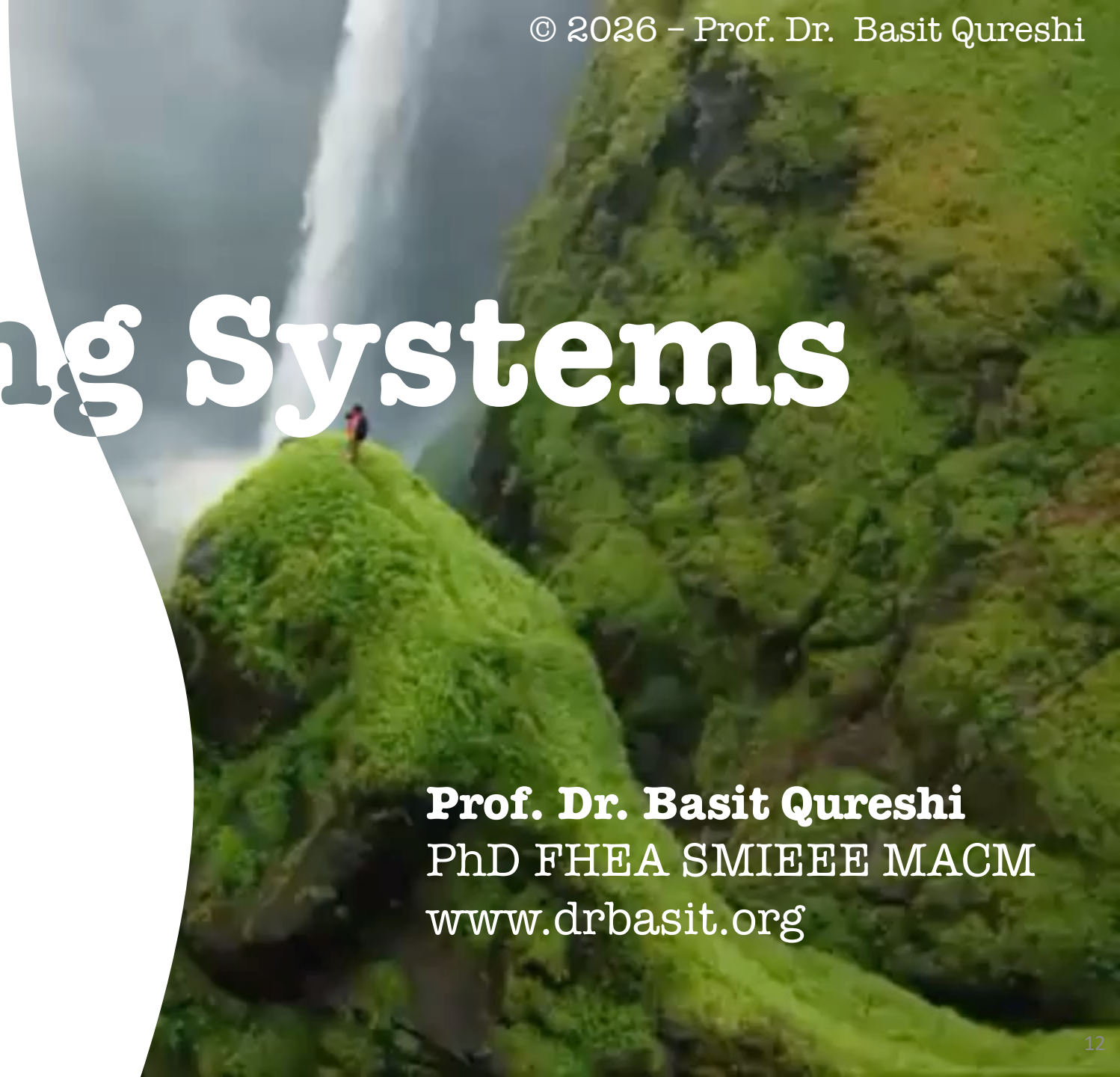
- **Parallelism on a multi-core system:**
  - Concurrency means that execution of threads can run in parallel because the system can assign a separate thread to each core.



MULTICORE PROGRAMMING: CHALLENGES

- **Dividing activities:** dividing work into equal parts.
- **Balance:** ensure that tasks perform equal work or equal value
- **Data splitting:** data accessed by the tasks must be divided to run on separate cores
- **Data dependency:** 1 task depend on data from another, the execution synchronized
- **Testing and debugging:** When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications

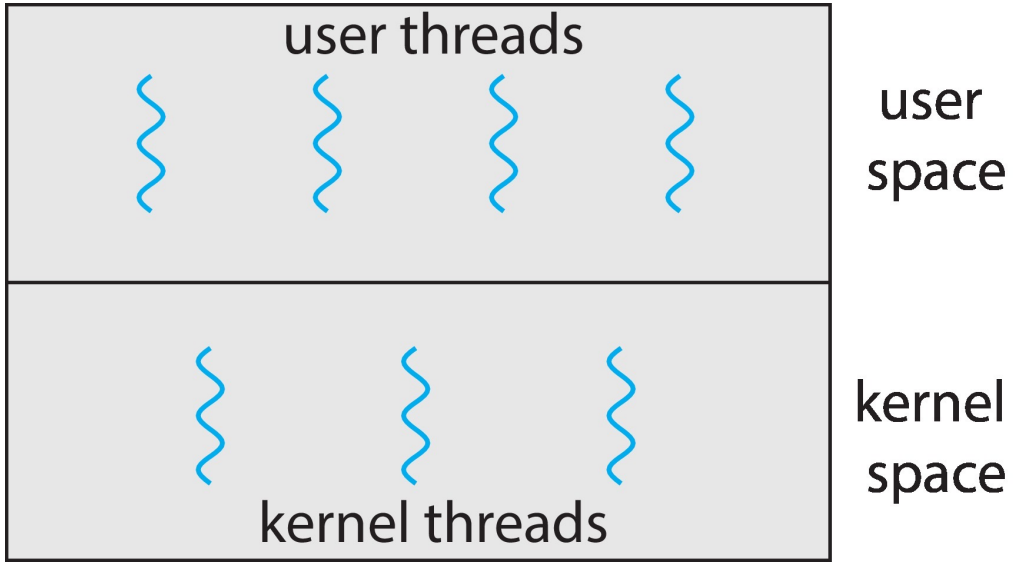
CS330 Operating Systems



TYPES OF THREADS

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

USER AND KERNEL THREADS



USER THREADS

- A thread is managed by an application (by user-level threads library) not in the kernel
- User threads are scheduled by the thread library
- Fast to create and manage: all thread creation and scheduling are done in user space without the need for kernel intervention
- Any user-level thread performing a **blocking system call** will cause the entire process to block if the kernel is single threaded.
- Examples of three primary thread libraries
  - POSIX *Pthreads*
  - Java threads
  - Win32 threads

KERNEL THREADS

- All thread management is done by the **kernel** itself
- Generally **slower** to create and manage than user threads
- If a thread is performing a blocking system, the kernel can **schedule** another thread in an application to be executed
- Kernel can **schedule** threads on different processors
- Kernel does creation, scheduling and management of threads.

- Virtually all general purpose operating systems has Kernel threads: Windows, Linux, Mac OS X, iOS, Android

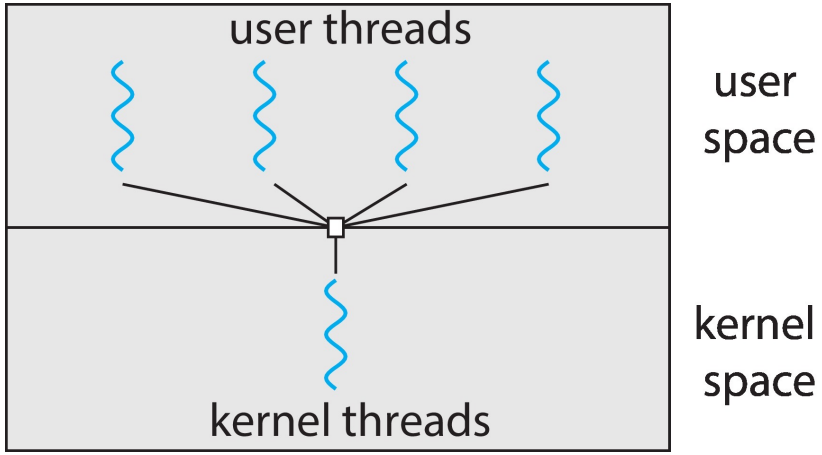
MULTITHREADING MODELS

What is the relationship between the user & kernel threads:

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

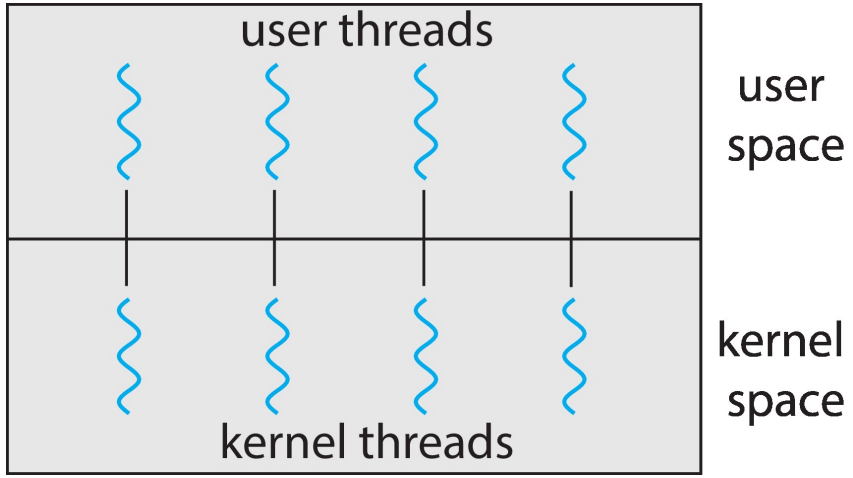
MANY-TO-ONE

- Many User threads are mapped onto a Single Kernel Thread.
- Used on Systems That Do Not Support Kernel Threads.
- Thread management is done outside the kernel--Efficient but
  - Entire process will block if a thread performs a blocking system call.
  - Only one thread can access the kernel at a time so multiple threads are unable to run in parallel on multiprocessors.
  - Few systems currently use this model
- Examples
  - Solaris Green Threads
  - GNU Portable Threads



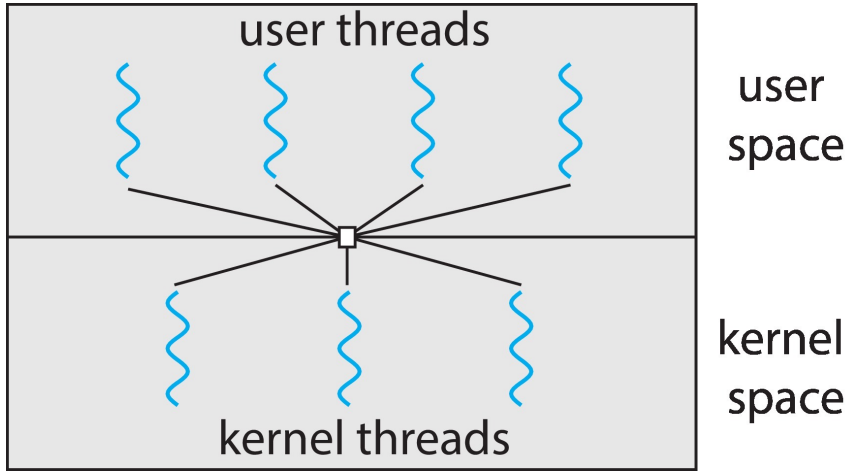
ONE-TO-ONE

- Each User-Level Thread is mapped onto its own single Kernel
- Thread giving more concurrency than many-to-one **by allowing other threads to run** if a thread performs a blocking system call.
- Threads don't block each other.
- Thread management is done by the kernel--**Slower**!
- Different threads can be run on different CPU's in an MP system.
- **Drawback**: overhead of creating so many kernel threads for an application so there is always a limit.
- Examples:
  - Windows
  - Linux



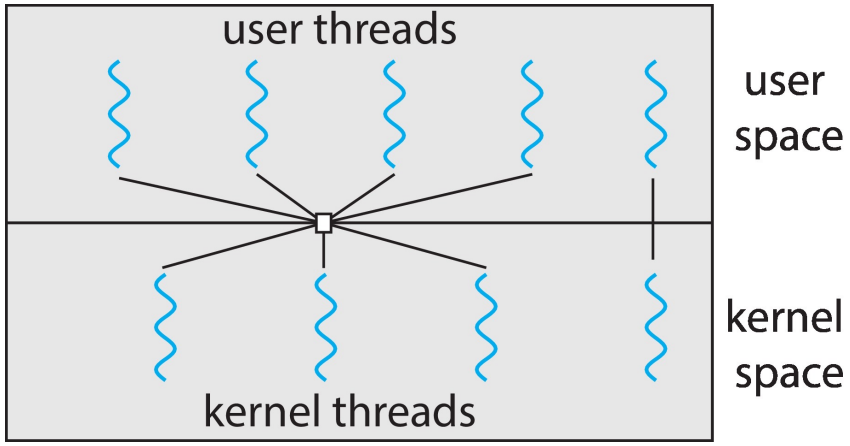
MANY-TO-MANY

- Many user-level threads may be mapped onto a smaller or equal number of kernel threads.
- Kernel threads run in parallel on a MP system. If a thread performs a blocking system call, the kernel can schedule another thread for execution
- Examples:
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package

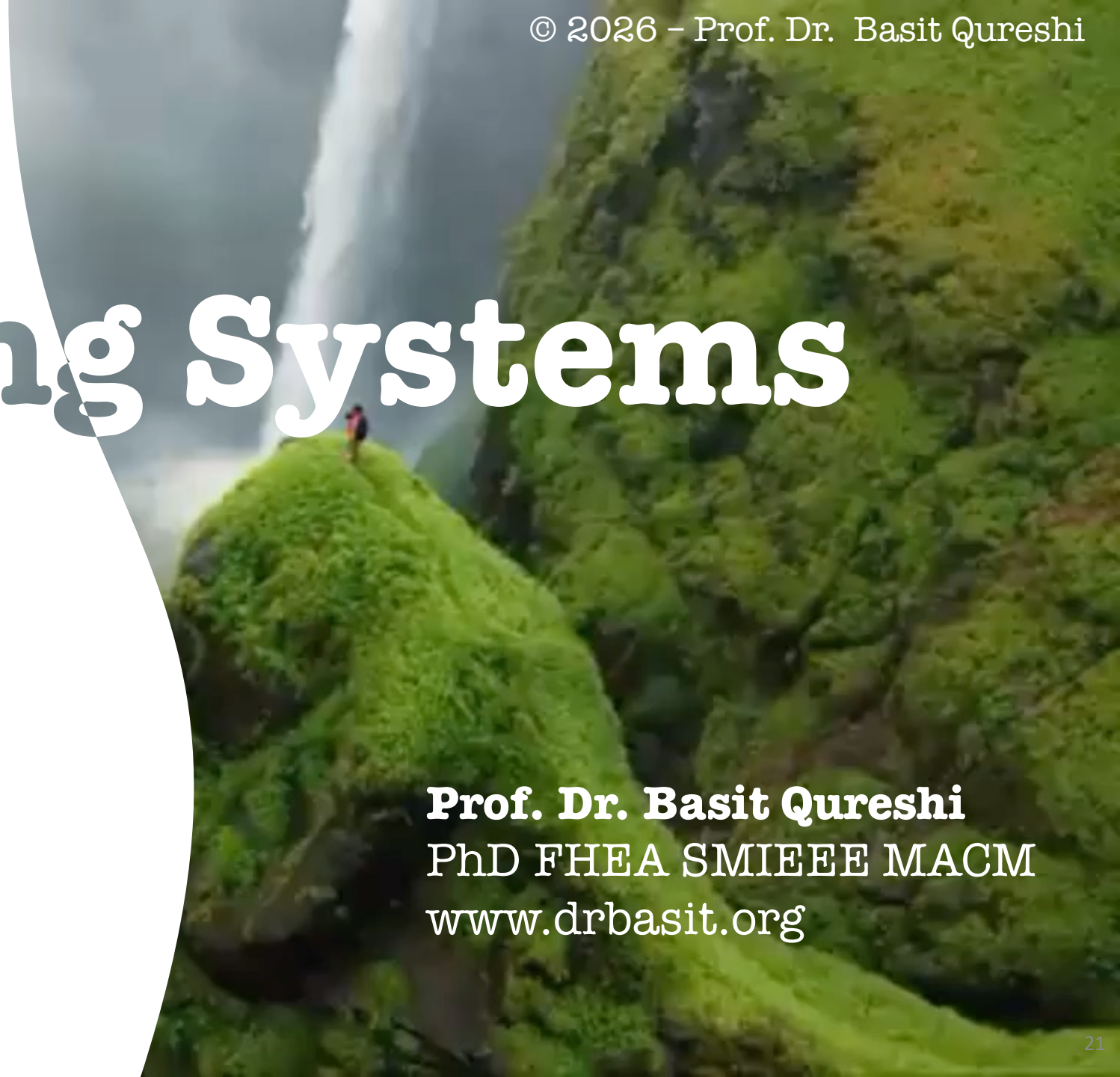


TWO-LEVEL MODEL

- 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



CS330 Operating Systems



PTHREAD

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

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 (Linux & Mac OS X)

THREADS EXAMPLE

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

void *printHello(void *arg)
{
    printf("Hello\n");
    return NULL;
}

int main()
{
    pthread_t thread;

    // Create a thread
    pthread_create(&thread, NULL, printHello, NULL);

    // Wait for the thread to finish
    pthread_join(thread, NULL);

    return 0;
}
```

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

void *printMessage(void *arg)
{
    char *message = (char *)arg;
    printf("%s\n", message);
    return NULL;
}

int main()
{
    pthread_t thread;
    char *message = "Hello from the thread!";

    // Pass message to the thread
    pthread_create(&thread, NULL, printMessage, message);

    // Wait for the thread to finish
    pthread_join(thread, NULL);

    return 0;
}
```

Shared=0

T1 increments

Shared=1

T2 increments

Shared=2

```
T1: shared = 1
T2: shared = 2
Final value of shared = 2
```

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

int shared = 0;

void *increment(void *arg){
    char *(message) = (char*) arg;
    shared ++;
    printf("%s: shared = %d\n", message, shared);
    return NULL;
}

int main(){
    pthread_t thread1, thread2;

    char *message1 = "T1";
    char *message2 = "T2";

    // Create two threads
    pthread_create(&thread1, NULL, increment, message1);
    pthread_create(&thread2, NULL, increment, message2);

    // Wait for both threads
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Final value of shared = %d\n", shared);

    return 0;
}
```

```
Hello from thread 0
Hello from thread 3
Hello from thread 1
Hello from thread 4
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 7
Hello from thread 8
Hello from thread 9
```

```
Hello from thread 0
Hello from thread 3
Hello from thread 4
Hello from thread 2
Hello from thread 5
Hello from thread 6
Hello from thread 1
Hello from thread 7
Hello from thread 8
Hello from thread 9
```

```c
#include <stdio.h>
#include <pthread.h>
void *printMessage(void *arg){
    int id = *(int *)arg;
    printf("Hello from thread %d\n", id);
    return NULL;
}

int main(){
    pthread_t threads[10];
    int thread_id[10];

    // Create 10 threads
    for (int i = 0; i < 10; i++){
        thread_id[i] = i;
        pthread_create(&threads[i], NULL, printMessage, &thread_id[i]);
    }

    // Wait for all threads to finish
    for (int i = 0; i < 10; i++)
    {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

```
Thread 1: Started
Thread 2: Started
Thread 3: Started
Thread 3: Woke up and quitting
Thread 1: Woke up and quitting
Thread 2: Woke up and quitting
Main: All threads have finished
```

Notice that the **order is determined by scheduling and sleep times**:

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

void *threadFunction(void *arg){
    int id = *(int *)arg;
    printf("Thread %d: Started\n", id);

    // Each thread sleeps for a different amount of time
    if (id == 1) sleep(3);
    else if (id == 2) sleep(5);
    else if (id == 3) sleep(2);

    printf("Thread %d: Woke up and quitting\n", id);
    pthread_exit(NULL);
}
int main(){
    pthread_t threads[3];
    int thread_id[3] = {1, 2, 3};

    // Create 3 threads
    for (int i = 0; i < 3; i++){
        pthread_create(&threads[i], NULL,
                       threadFunction, &thread_id[i]);
    }

    // Wait for all threads to finish
    for (int i = 0; i < 3; i++){
        pthread_join(threads[i], NULL);
    }

    printf("Main: All threads have finished\n");
    return 0;
}
```