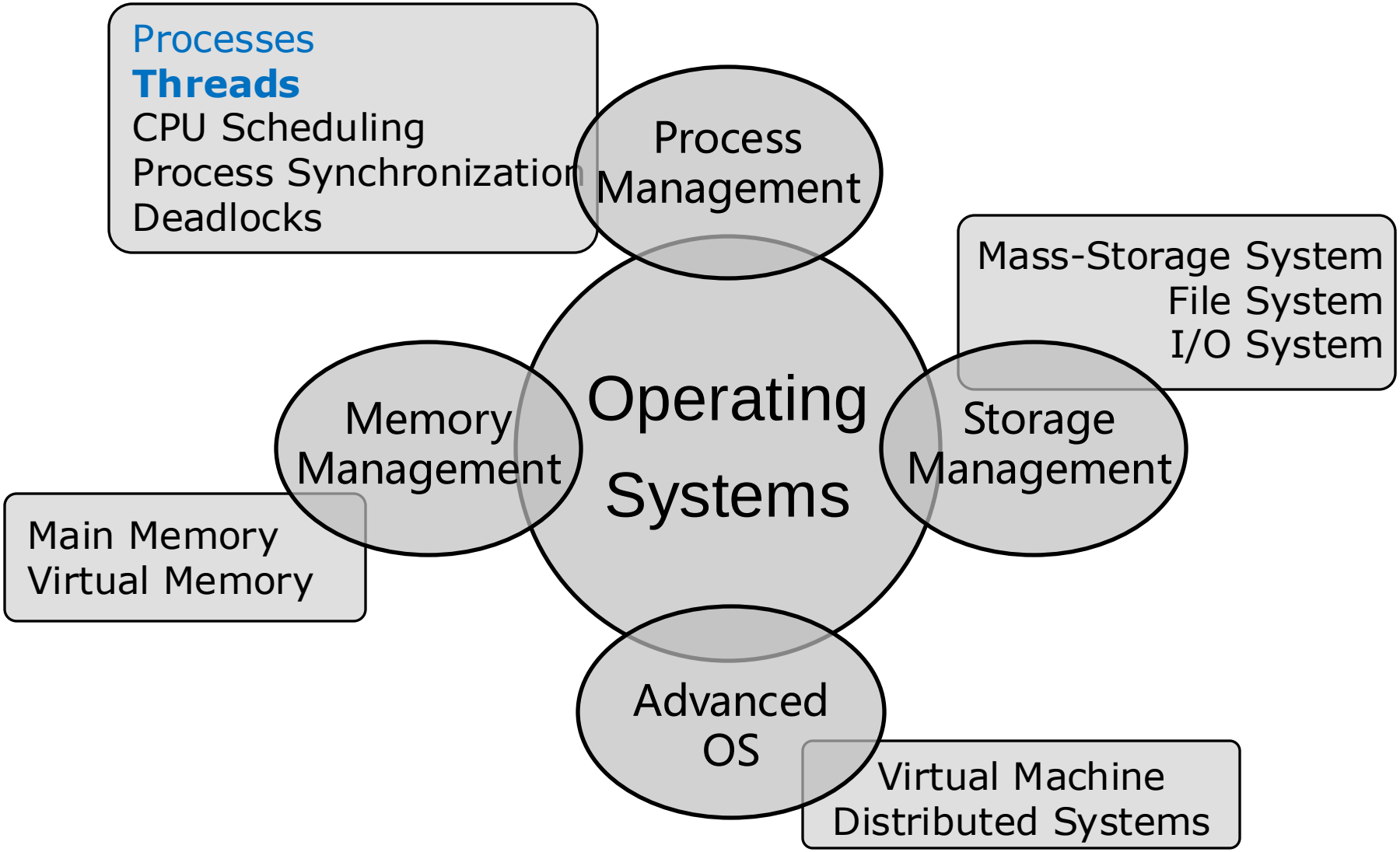
Threads & Concurrency

Shengzhong Liu
Department of Computer Science and Engineering
Shanghai Jiao Tong University

Operating System Topics



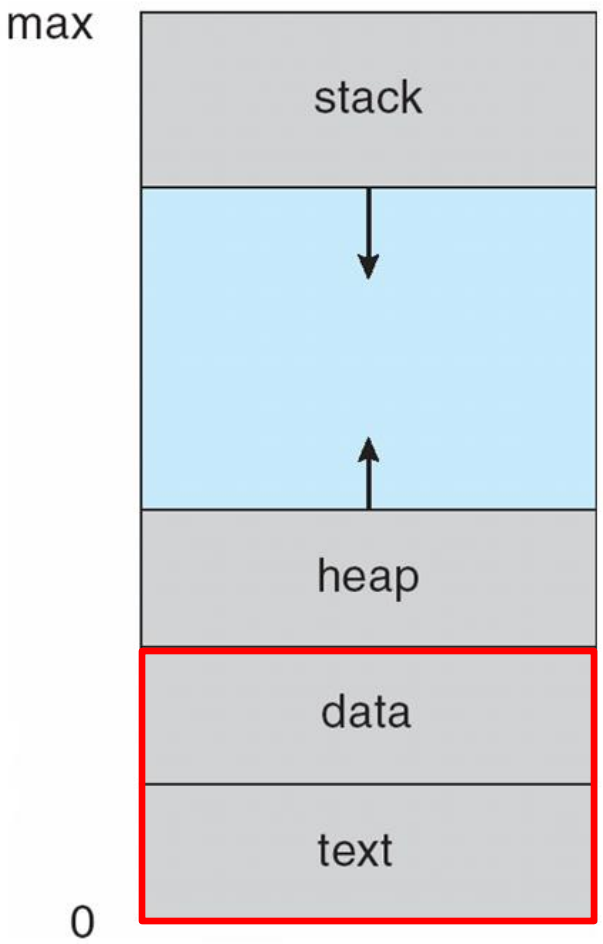
Outline

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

Thread Concept

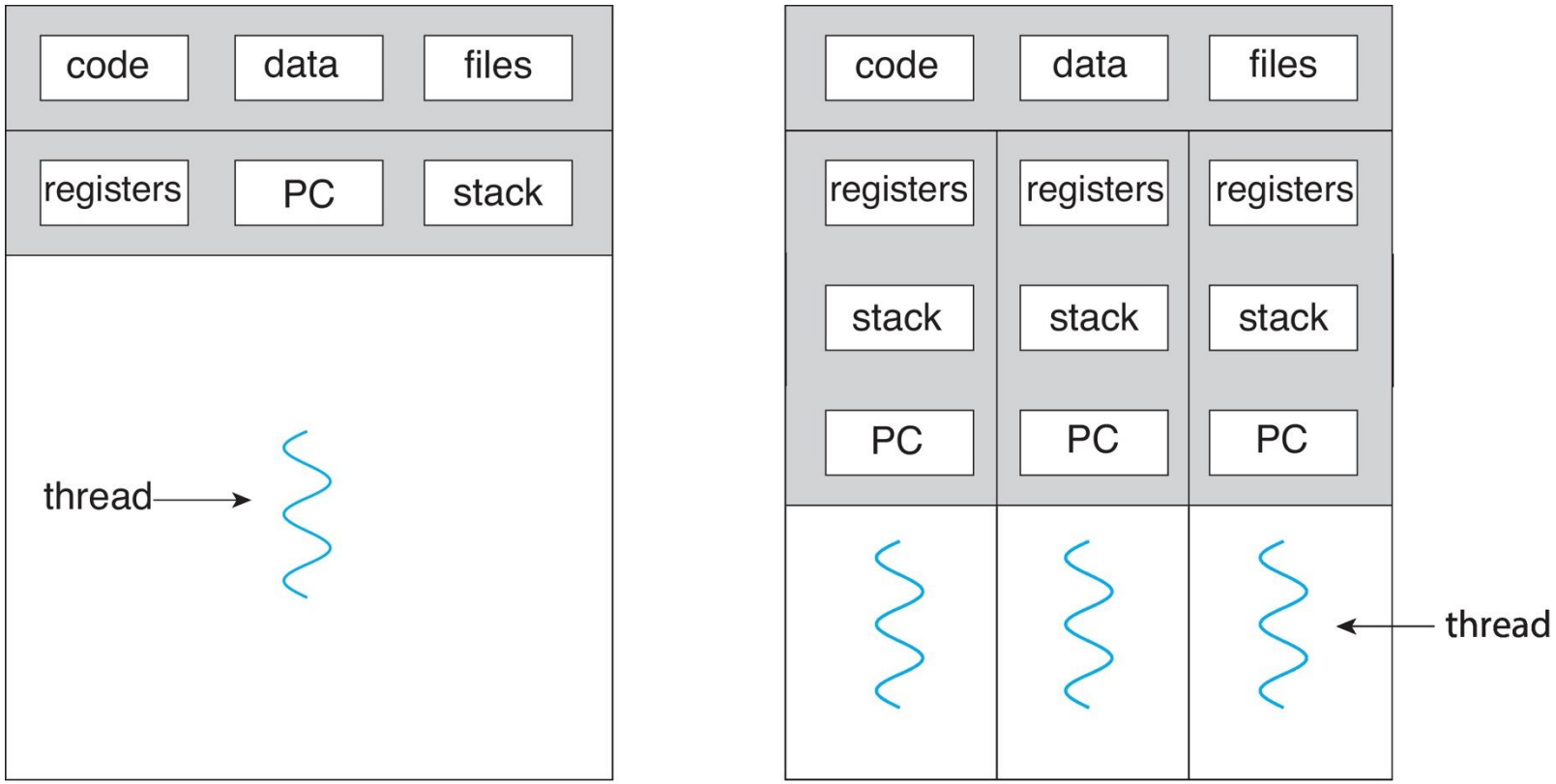
Thread Definition

- A thread is a basic unit of CPU utilization
  - A thread comprises
    - a thread ID,
    - a program counter,
    - a register set,
    - a stack
  - A thread shares with other threads belonging to the same process
    - code section
    - data section
    - other operating-system resources, such as open files



Process memory layout

Single and Multithreaded Processes



single-threaded process

multithreaded process

Process vs. Thread

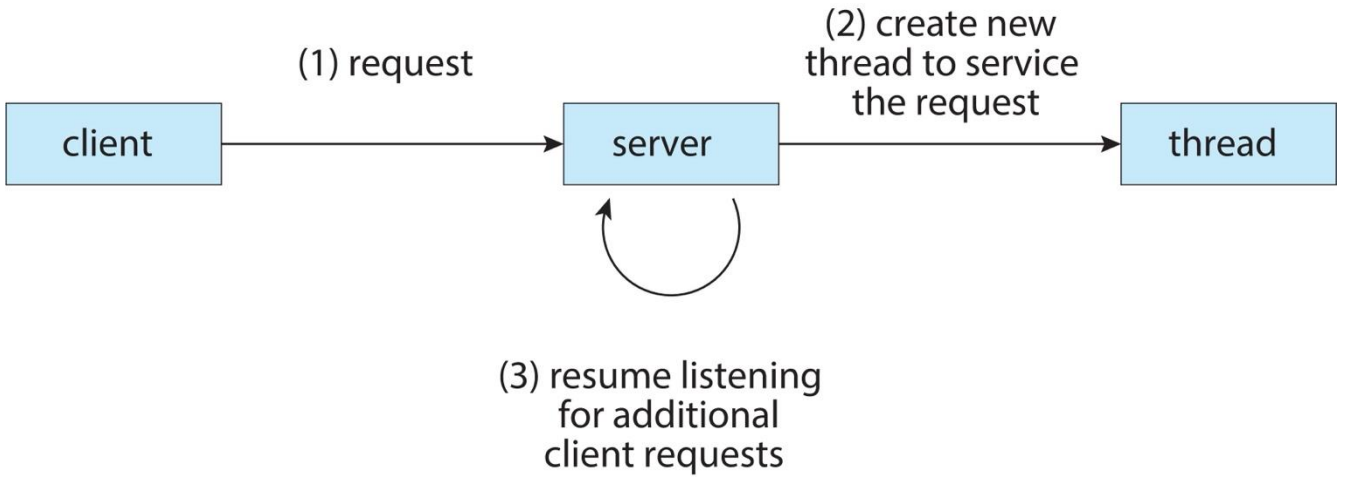
	Process	Thread
Unit	Independent	Exists as subsets of a process
State Information	Carries considerably more state information	Shares process state as well as memory and other resources
Address Space	Has separate address space	Shares process's address space
Communication	Interact only through IPC	More ways to communicate
Context Switch	Relatively slow	Context switching in the same process is typically faster

Motivation

- Two application scenarios:
  - A single application is sometimes required to perform several similar tasks
  - An application wants to leverage processing capabilities on multicore systems
- Process creation is heavy-weight while thread creation is light-weight
- Multi-threaded web server - Multi-tasks with the application can be implemented by separating threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request

Motivation: (1) Multithreading for Multi-Tasks

- Example 1: Client-server applications use multi-threading to improve efficiency

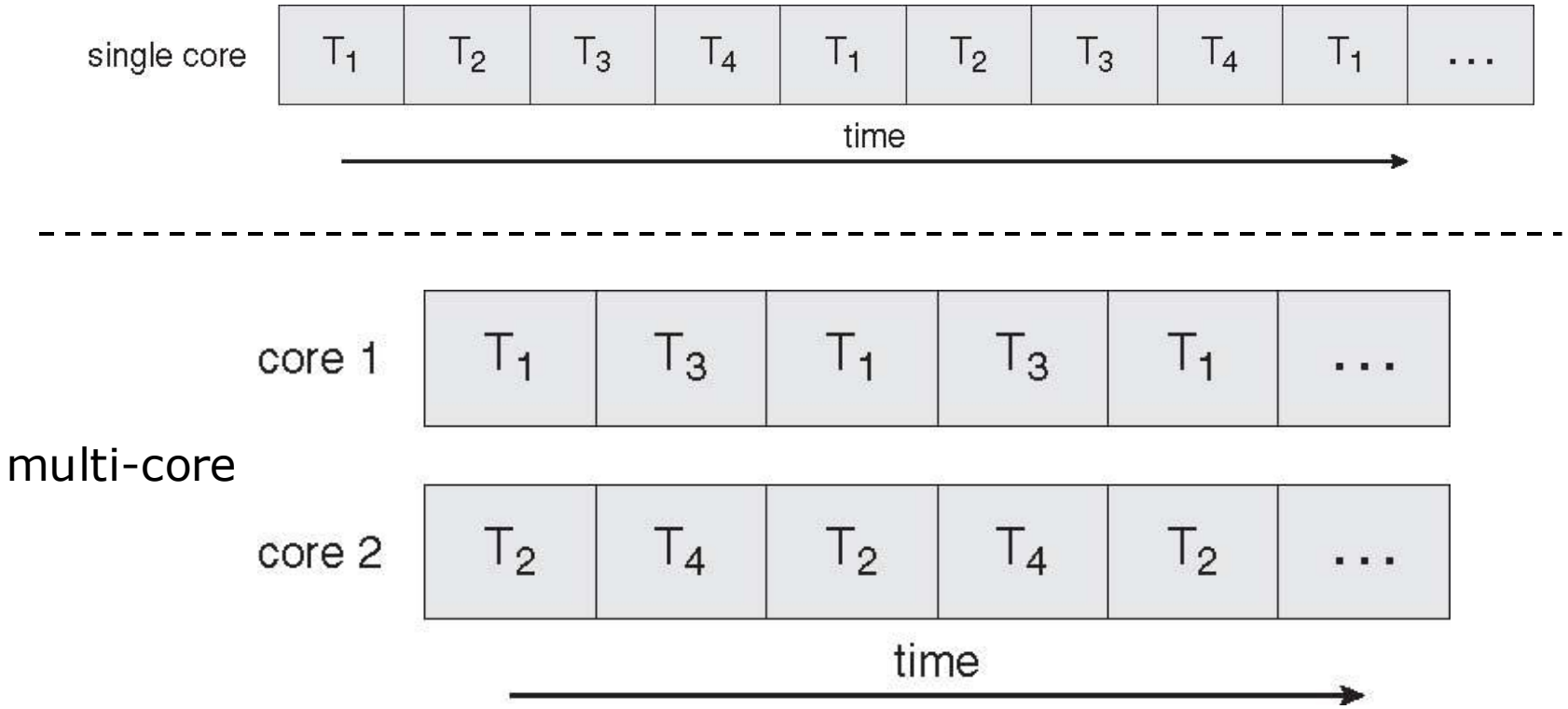


- Example 2: Most operating system kernels are multithreaded.
  - In Linux, use **`ps -ef | grep kthreadd`** to find the parent thread of all kernel threads

```
(nvitop) admin@Shengzhong-PX:~/Shengzhong$ ps -ef | grep kthreadd
root           2       0  0 Feb19 ?        00:00:05 [kthreadd]
admin    1187782 1185436  0 14:40 pts/0    00:00:00 grep --color=auto kthreadd
```

Motivation: (2) Multithreading for Multi-Core Execution

Parallel execution on a multi-core system



Benefits of Multithreading

- **Responsiveness**
  - A program continues running even if part of it is blocked or is performing a lengthy operation
- **Resource Sharing**
  - Threads share the memory and the resources of the process to which they belong
  - IPC techniques are not needed
- **Economy**
  - Creating a thread is much faster than creating a process
  - Thread switching lower overhead than context switching
- **Scalability**
  - Process can take advantage of multicore architectures

Drawbacks of Multithreading

- Make the programming more complicated
- Make the debugging harder
- Possible error when threads concurrently access the shared resources
- Poorly divided jobs can cause even worse system performance
- .....

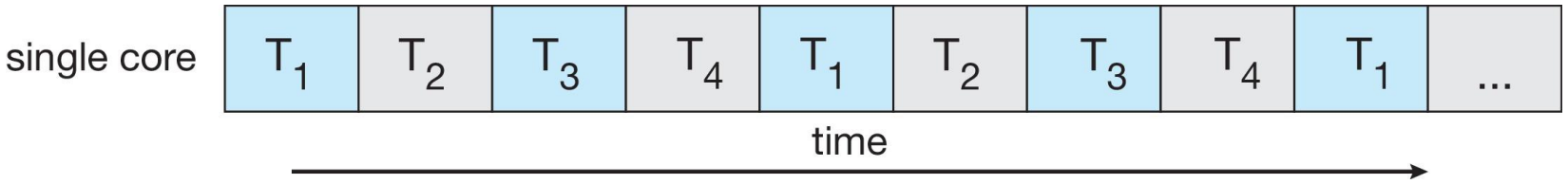
Multicore Programming

Multicore Programming

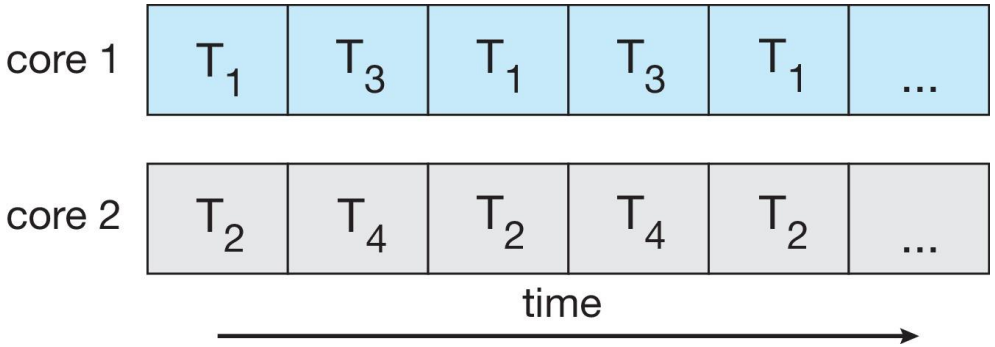
- Two types of multicore programming
  - ***Concurrency***（并发）supports more than one task making progress
    - Single processor / core, scheduler providing concurrency
    - A more generalized form of parallelism includes time-slicing for virtual parallelism.
  - ***Parallelism*** (并行) implies a system can perform more than one task simultaneously
    - Multiple tasks running on multiple processors / cores
    - A condition that at least two threads are executing simultaneously.
- Relations:
  - Concurrency defines a problem: two things need to happen together.
  - Parallelism gives a solution: two processor cores execute two things simultaneously.
  - Interleaved processing (through time-sharing) is another concurrency solution.

Concurrency vs. Parallelism

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



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

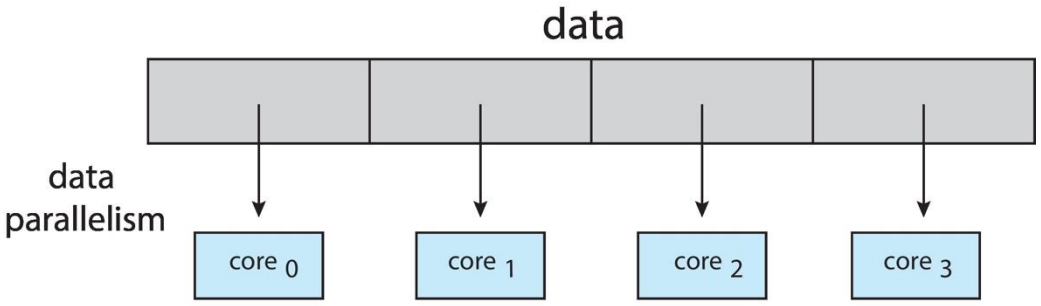


Concurrency vs. Parallelism

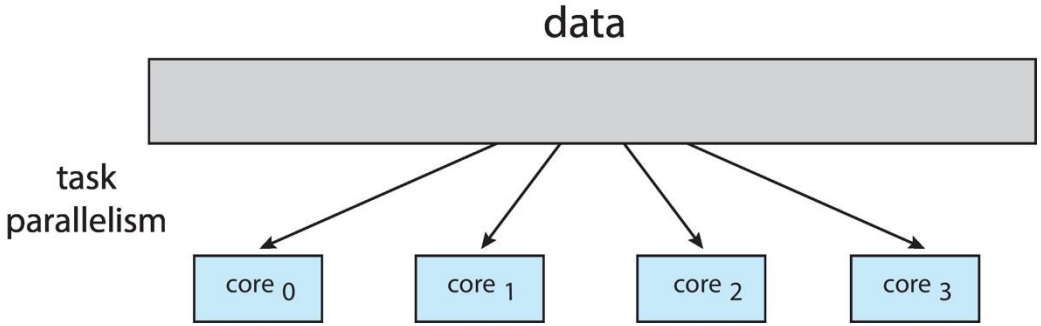
	Concurrency	Parallelism
Basic	Managing and running multiple computations at the same time	Running multiple computations simultaneously
Implementation	Interleaving Operation	Using multiple CPUs
Benefits	Increased amount of work accomplished at a time	Improved throughput, computational speed-up
Make use of	Context switching	Multiple CPUs for operating multiple processes.
Required Processing Units	Probably single	Multiple
Example	Running multiple applications at the same time	Running web crawlers on a cluster

Types of Parallelism

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each

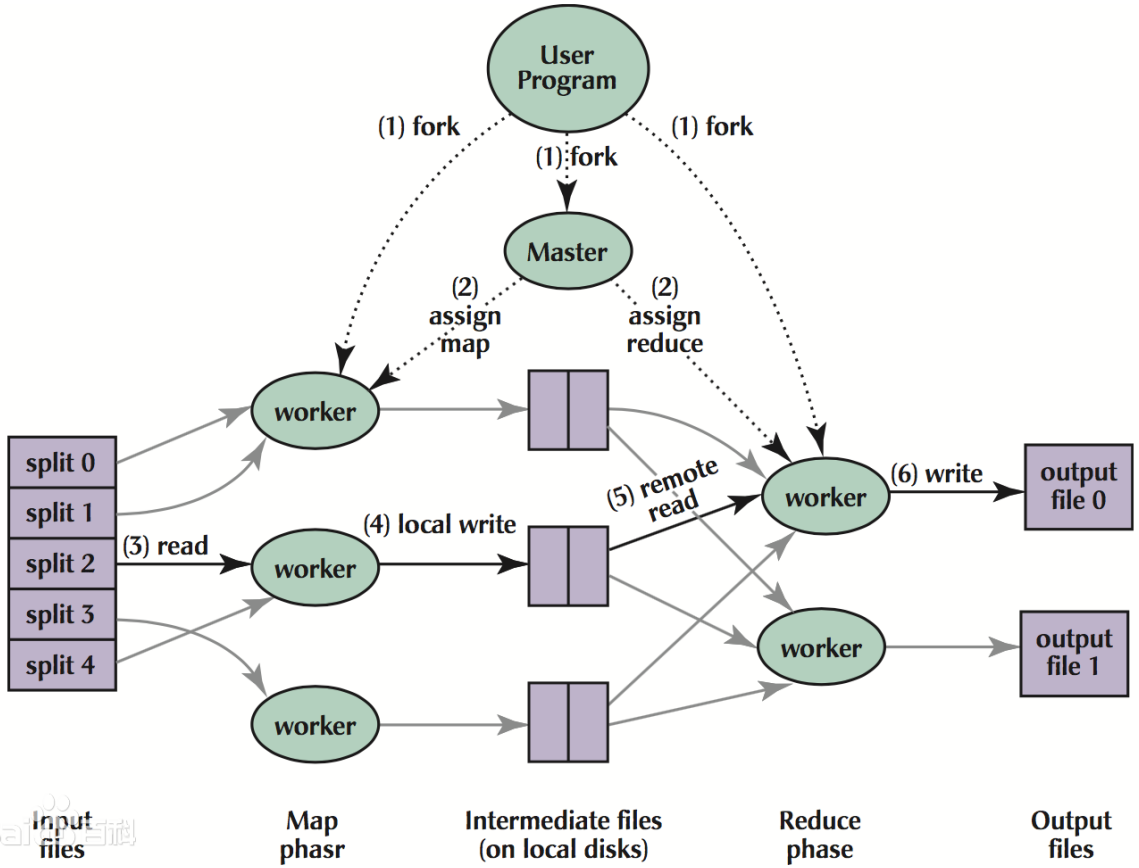


  - **Task parallelism** – distributing threads across cores, each thread performing unique operation



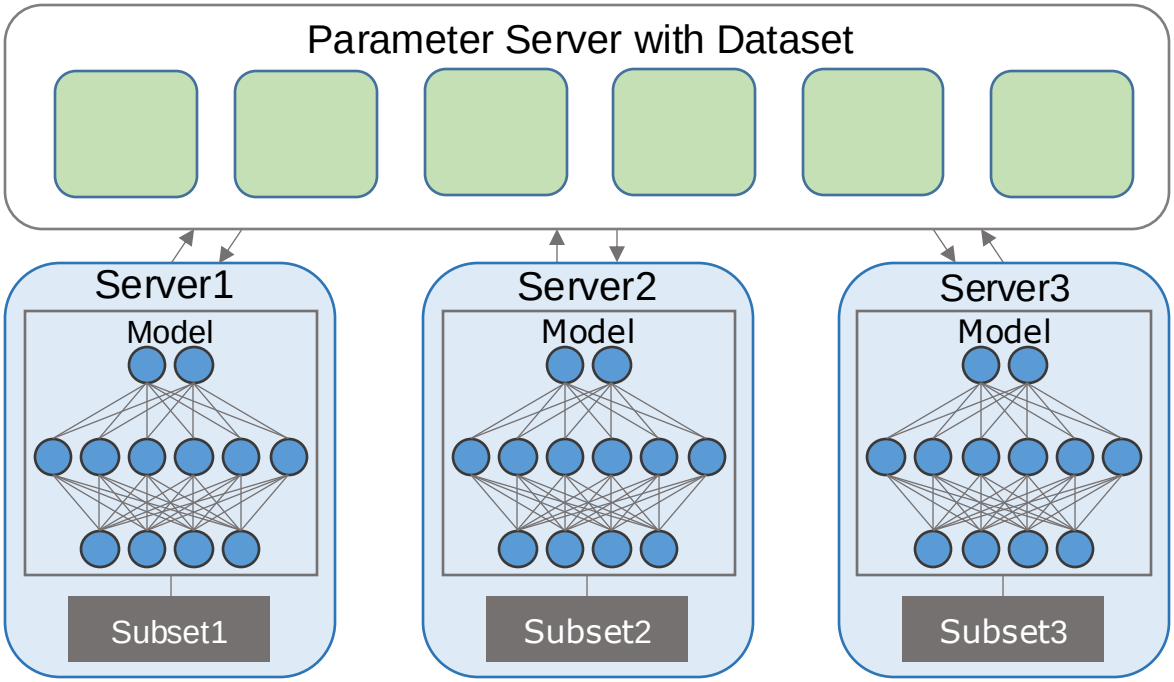
Example: Data Parallelism - MapReduce

- MapReduce is a programming model or pattern within the Hadoop framework that is used to access big data stored in the Hadoop File System (HDFS).

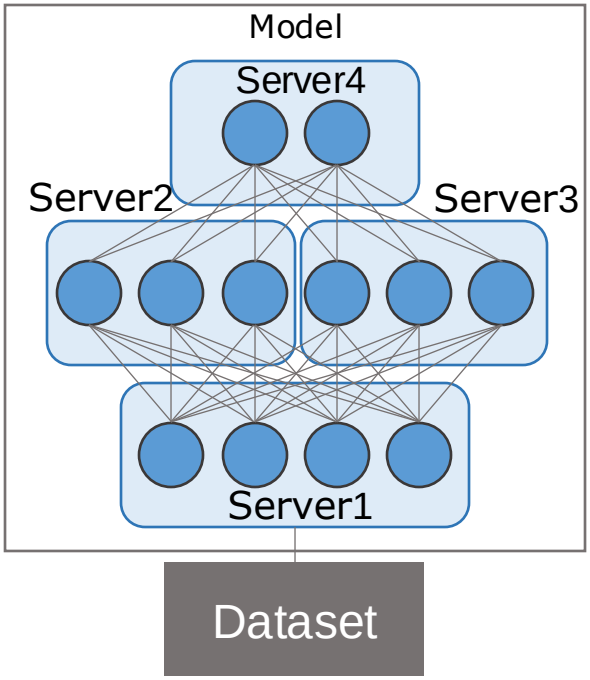


Example: Distributed Machine Learning

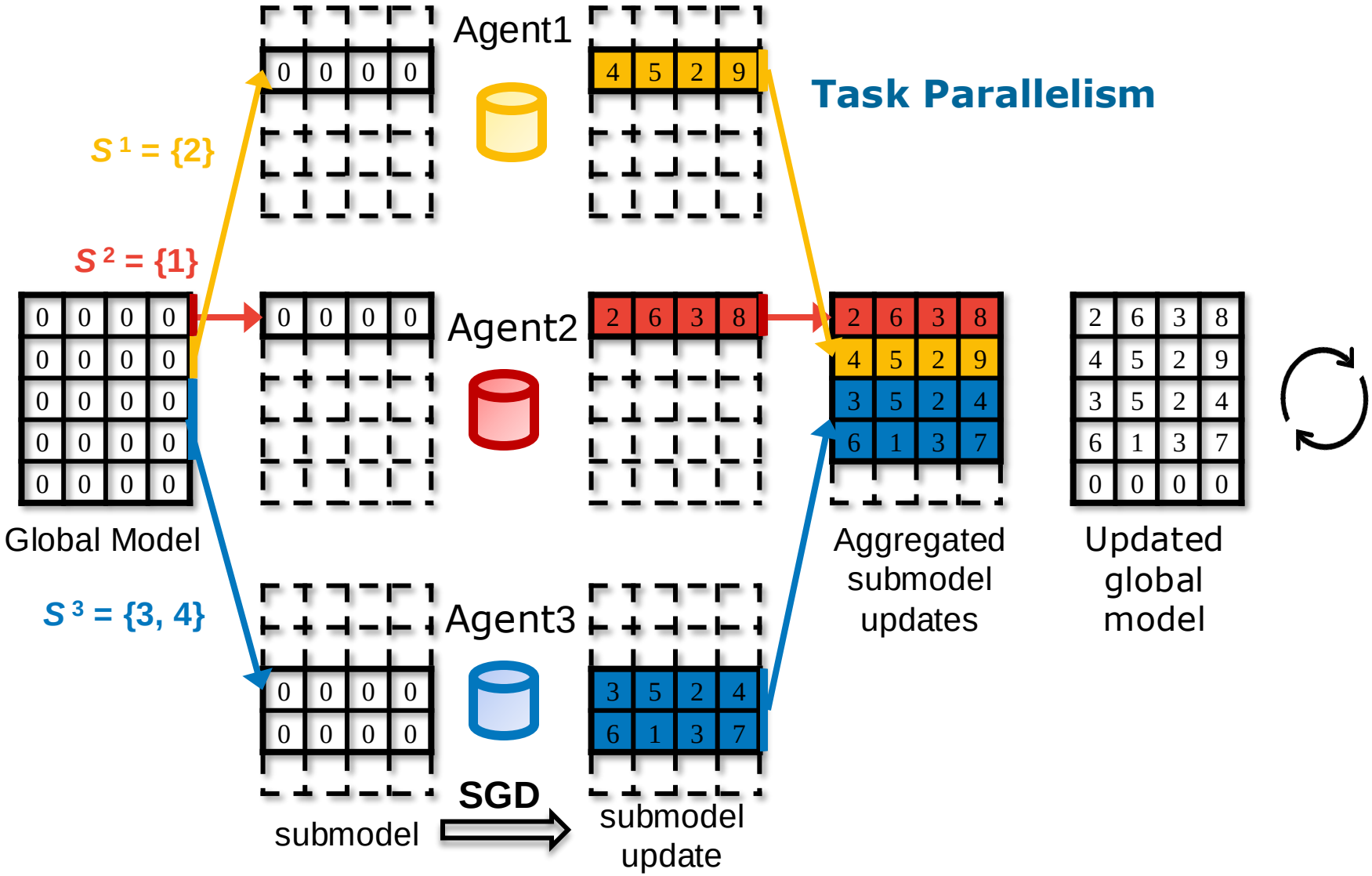
Data Parallelism



Task Parallelism



Example: Federated Submodel Learning



Amdahl's Law

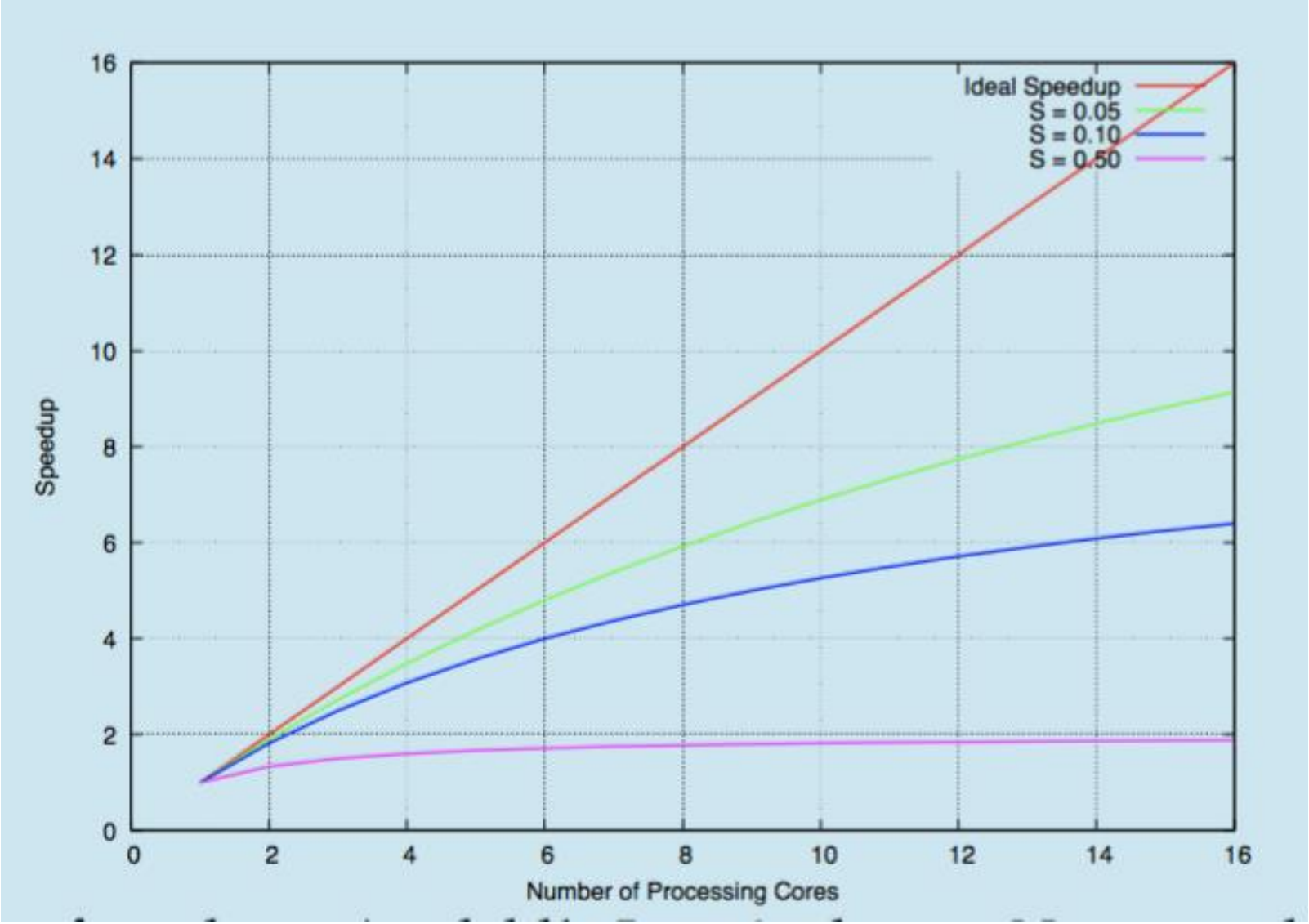
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
  - *S* is the serial portion
  - *N* processing cores

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

- 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

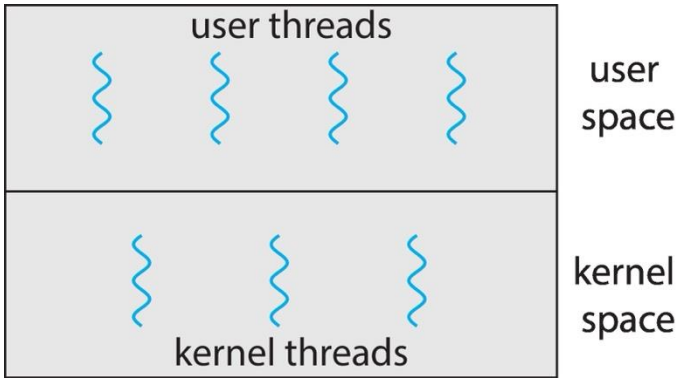
Amdahl's Law


Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

Multithreading Models

Kernel Threads vs. User Threads

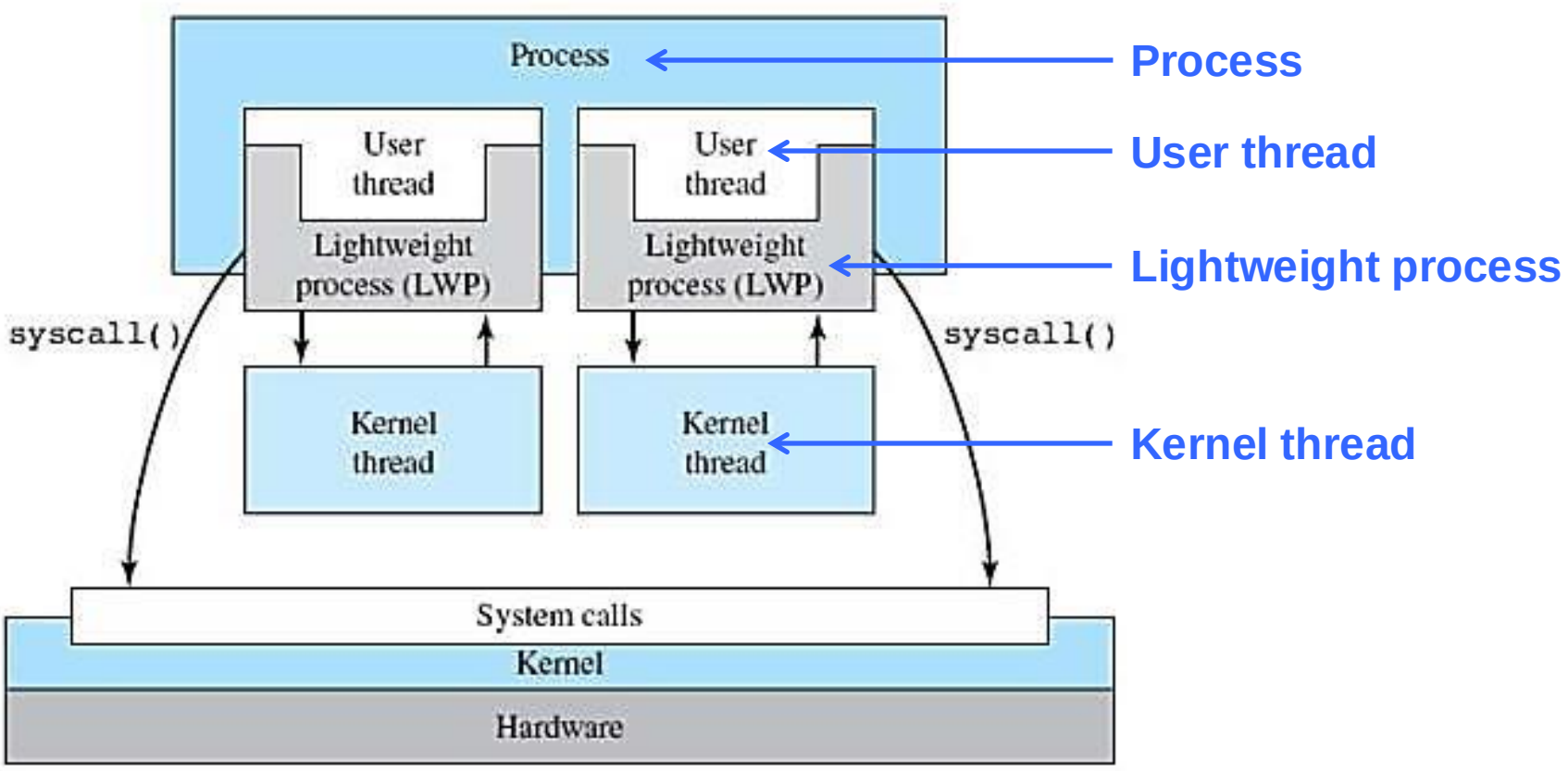
- **Kernel Threads**
  - Supported by the OS kernel
  - Examples - virtually all general-purpose operating systems support kernel threads
    - Windows
    - UNIX, Linux
    - Mac OS X, iOS
    - Android



- **User Threads**
  - Thread management done by user-level threads library. Unwared by the kernel.
  - Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads

- User threads rely on kernel threads for execution.

Thread Model



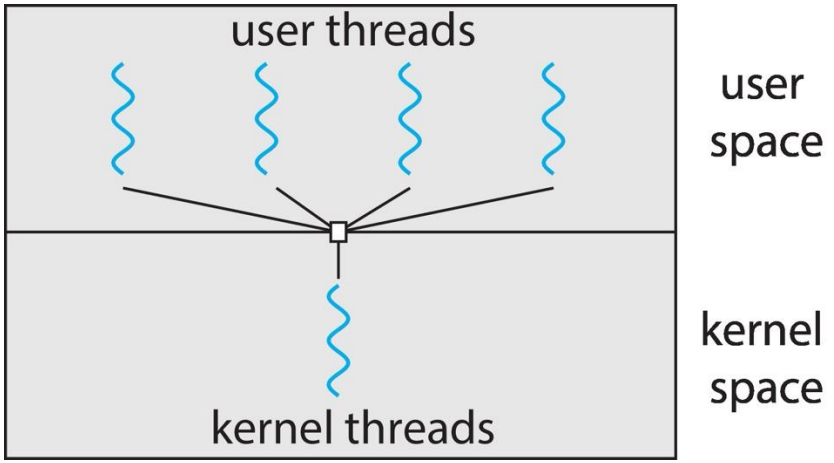
Lightweight process (LWP): A mapping between user threads and kernel threads

Multithreading Models

- Four common connections between user threads and kernel threads
  - Many-to-One
  - One-to-One
  - Many-to-Many
  - Two-Level Model

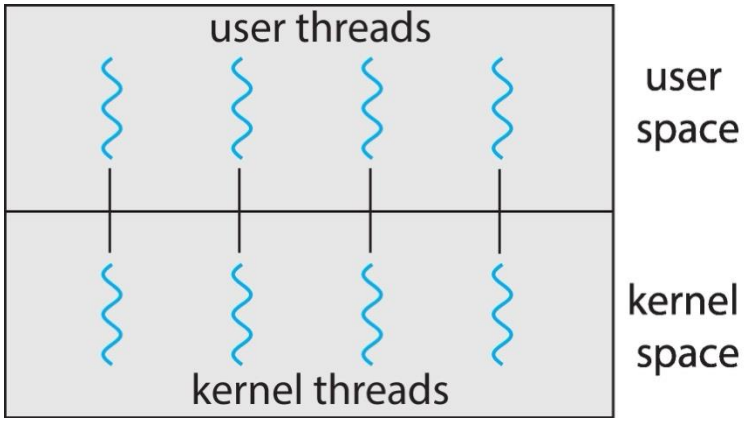
Multithreading Model: Many-to-One

- Many user-level threads are mapped to a single kernel thread
- Strength
  - Multiple threads are hidden by user-level thread library
  - The user-level thread operations are fast without kernel involvement
- Weaknesses
  - The entire process will be blocked if a thread makes a blocking system call
  - Multiple threads are unable to run in parallel on multiprocessors
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



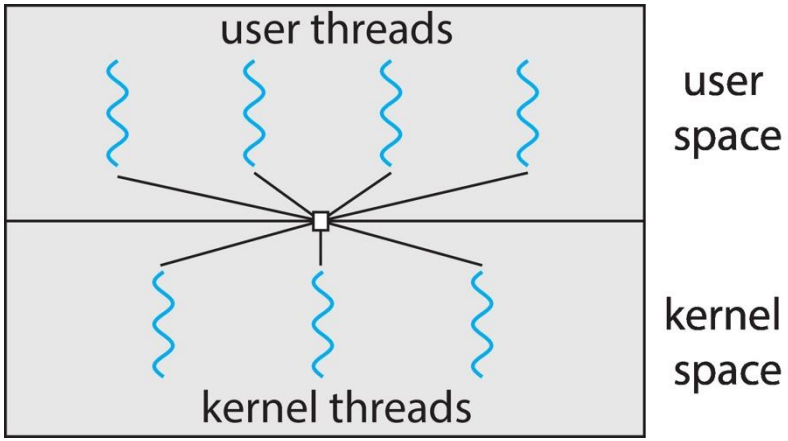
Multithreading Model: One-to-One

- Each user-level thread is mapped to a kernel thread
- Strength
  - More concurrency than many-to-one
- Weakness
  - High overhead: Creating a user thread requires creating the corresponding kernel thread, incurring overhead
  - The OS may restrict how many kernel threads can be created.
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



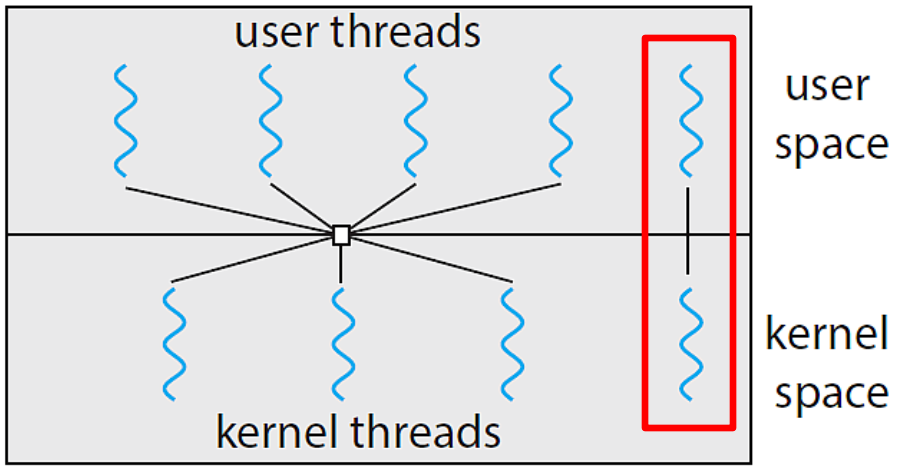
Multithreading Model: Many-to-Many

- Allows many user-level threads to be mapped to many kernel threads
  - Typically, the #user threads are more than the #kernel threads.
- Strength:
  - Flexibility: Can have a large number of user threads without creating too many kernel threads.
  - True parallelism: Can effectively utilize multiple CPU cores.
- Weakness:
  - Complex implementation: The user-level library and kernel must coordinate carefully.
  - May lead to load imbalance between kernels.
- Examples
  - Windows NT/2000 with the *ThreadFiber* package



Two-Level Model

- Similar to Many-to-Many, except that it allows a user thread to be **bound** to a kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Thread Libraries

Thread Libraries

- **Thread library** provides the programmer with API for creating and managing threads
- Two primary ways of implementation
  - **User-level thread library**
    - All codes and data structures for the library exist in user space
    - Invoking a library function → Local function call in user space
  - **Kernel-level thread library**
    - Code and data structures for the library exist in kernel space
    - Invoking a library function → System call to the kernel
- Three primary thread libraries:
  - POSIX Pthreads, Win32 threads, Java threads

Pthreads

- Is provided either in user-level or kernel-level
- This is a specification for thread behavior, not an implementaiton.
  - A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - 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)

Example Using Pthreads

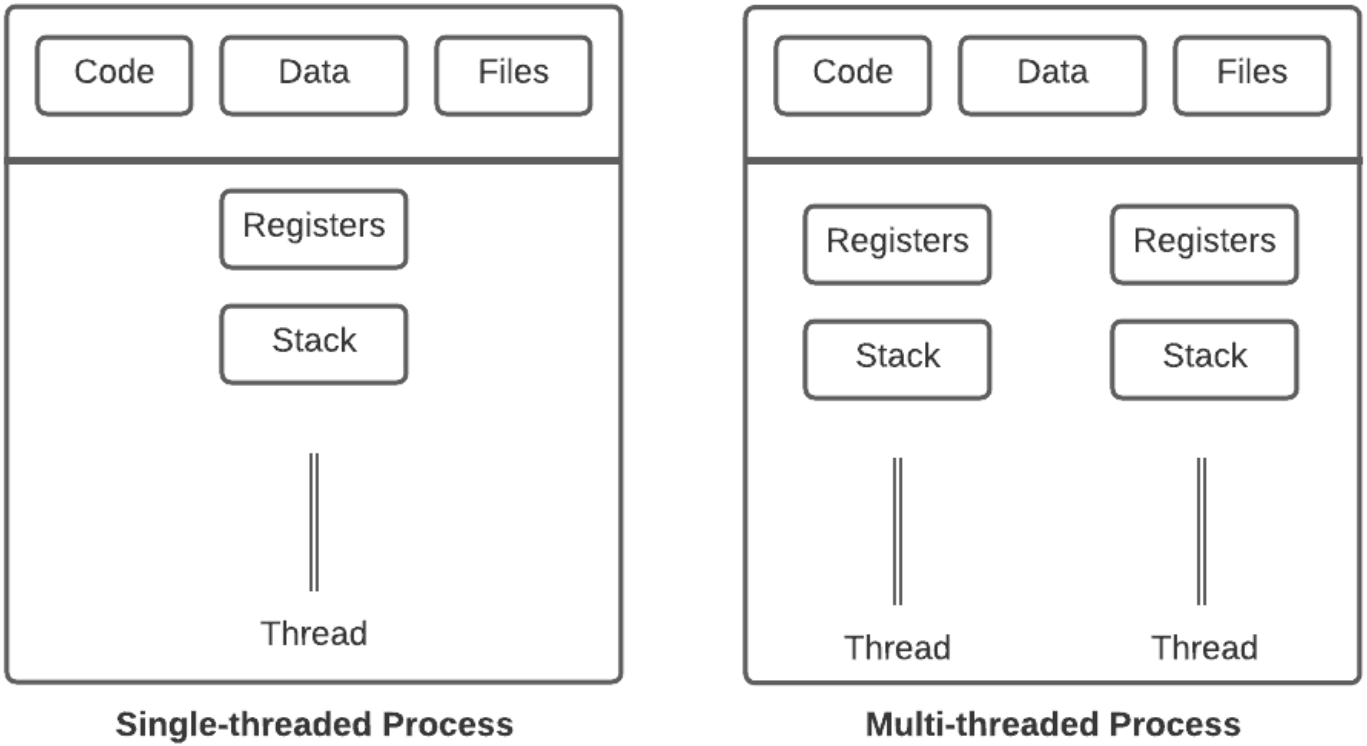
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

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; /* create thread identifier */
    pthread_attr_t attr; /* create thread attributes */
    /* set the default attributes of the thread */
    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);
}
```

Example Using Pthreads (Cont.)

```
/* The thread will execute 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);
}
```

`atoi()` function converts a string integer to its equivalent integer value

Process and Thread Differences



Process has its own memory space, and threads share code, data, files

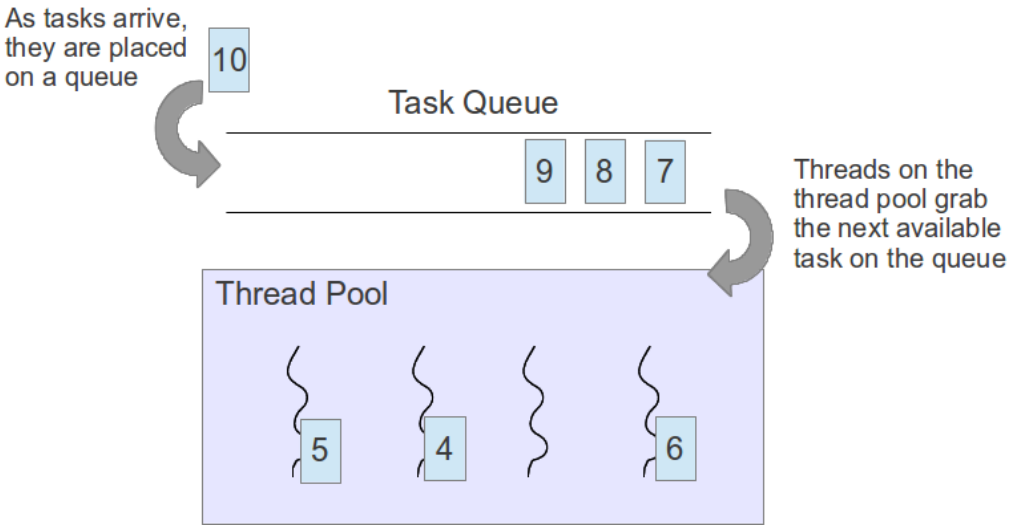
Implicit Threading

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
  - Hundreds to thousands of threads are hard to manage explicitly
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Advantage:
  - Developers only need to identify parallel tasks, and the libraries determine the specific details of thread creation and management.
- Five methods explored
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks

Implicit Threading: (1) Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Slightly faster to service a request with an existing thread than creating a new thread
  - Allows the number of threads in the application(s) to be bound to pool size
  - 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



Implicit Threading: (1) Thread Pools

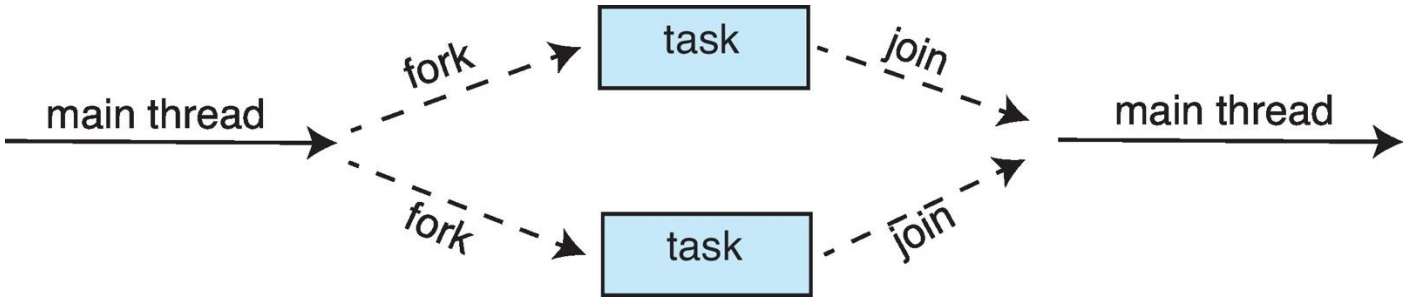
- Tread pool examples:
  - Windows - Windows API supports thread pools:

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

  - Java thread pools:
    - `static ExecutorService newSingleThreadExecutor()`
    - `static ExecutorService newFixedThreadPool(int size)`
    - `static ExecutorService newCachedThreadPool()`

Implicit Threading: (2) Fork-Join Parallelism

- General algorithm for fork-join strategy:
  - Threads are not constructed directly during the fork stage
    - Insetad, parallel tasks are designated
  - A library
    - manages the number of threads that are created
    - responsible for assigning tasks to threads.

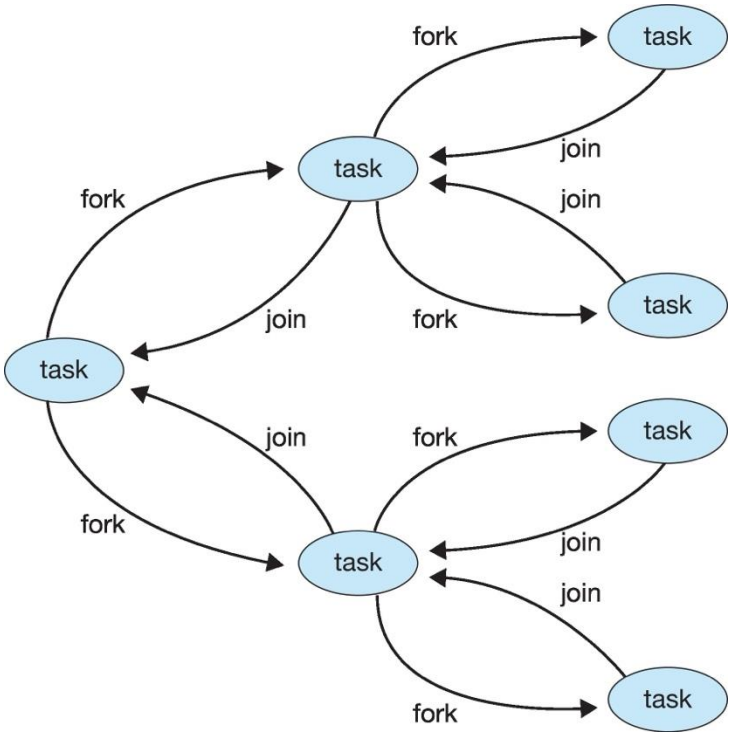


Implicit Threading: (2) Fork-Join Parallelism

- Java introduced a fork-join library in Version 1.7 for recursive (递归) divide-and-conquer algorithms such as Quicksort and **Mergesort**.
  - Separate tasks are forked during the divide step and assigned smaller subsets of the original problem.
  - At some point, the problem is small enough to be solved directly.

```
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
```

General recursive algorithm behind Java's fork-join model



Implicit Threading: (3) OpenMP

- Set of compiler directives (指令) 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 #cores
  - 【双核】Dual-core system → Two threads
  - 【四核】Quad-core system → Four threads

```
#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;
}
```

Implicit Threading: (4) Grand Central Dispatch

- GCD is an Apple technology for macOS and iOS operating systems
  - Extensions to C, C++ and Objective-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"); }
```

- Schedule tasks/blocks for execution by placing them in **dispatch queue**
  - Task assigned to available thread in thread pool when removed from queue
- Two types of dispatch queues:
  - **Serial** – Blocks removed in FIFO order but only **one block** removed at a time
  - **Concurrent** – Blocks removed in FIFO order but **several blocks** can be removed at a time

Implicit Threading: (5) TBB

- TBB – Intel Threading Building Blocks
  - Template library for designing parallel C++ programs
  - No need for a special compiler or language support

- A serial version of a simple for loop

```
for (int i = 0; i < n; i++) {
   apply(v[i]);
}
```

- The same for loop written using TBB with **`parallel_for`** statement:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```

Iteration Space Lambda Function Body

Threading Issues

Threading Issue

- Semantics of **fork()** and **exec()** system calls with multi-threading
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Threading Issues: (1) Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIX OSes have two versions of fork
    - One version duplicates all threads
    - One version duplicates only the thread that invoked `fork()` system call

- `exec()` usually works as normal if it is called by a thread
  - The program specified in the parameter to exec will replace the running process including all threads

Threading Issues: (2) 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 an event and delivered to a process
  2. 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 the default handler
- In single-thread program, signal is always delivered to the process
- Signal delivery options in multithreaded programs:
  - 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

Threading Issues: (3) Thread Cancellation

- Thread cancellation: Terminate a thread before it has finished
  - Thread to be canceled is called **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:

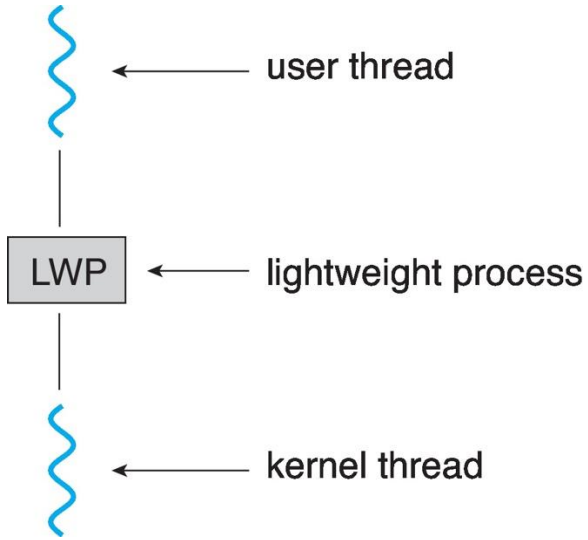
```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Threading Issues: (4) 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 should be visible across function invocations
- Similar to **`static`** data
  - TLS is unique to each thread

Threading Issues: (5) Scheduler Activations

- Both Many-to-Many and Two-level models require communication to maintain the appropriate #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 to the user-thread library to schedule the user thread to run
  - Each LWP attached to a kernel thread
- Scheduler activations provide **upcalls** - from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct #kernel threads



Operating System Thread Examples

Operating System Examples

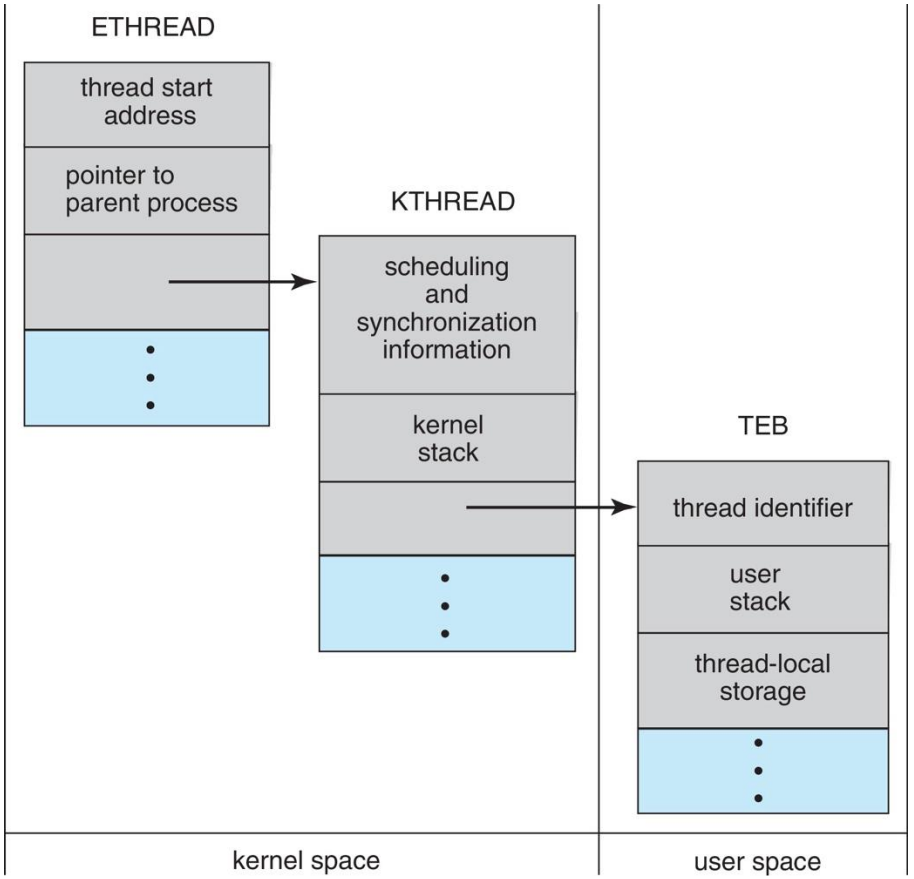
- Windows Threads
- Linux Threads

OS Example: (1) Windows Threads

- Windows API – primary API for Windows applications
  - 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

OS Example: (2) Windows Threads

- 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



OS Example: (2) 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`** stores pointers to process data structures (shared or unique)
  - Do not copy all data structures

Summary

- A thread represents a basic unit of CPU utilization. Threads of the same process share many process resources, including code and data.
- Discussed the benefits of multithreaded applications and the challenges in their implementation.
- Multicore programming:
  - Concurrency vs. Parallelism
  - Data Parallelism vs. Task Parallelism
- Multithreading models differ in their mappings between user threads and kernel threads.
- A thread library provides an API for creating and managing threads. We introduced Pthread as an example.
- Implicit threading involves identifying tasks—not threads—and allowing languages or API frameworks to create and manage threads.
- Reviewed threading issues and OS threading examples.

Homework

- Reading:
  - Chapter 4