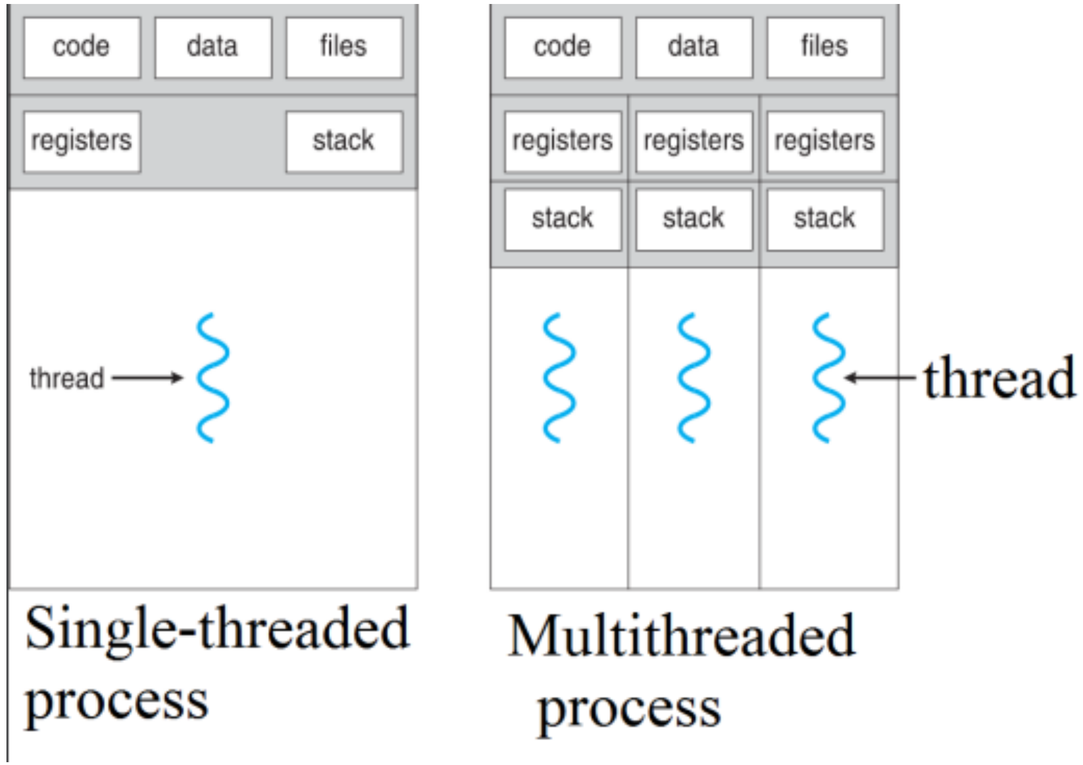
CPSC 5042: Week 6

Concurrency I

Multithreading

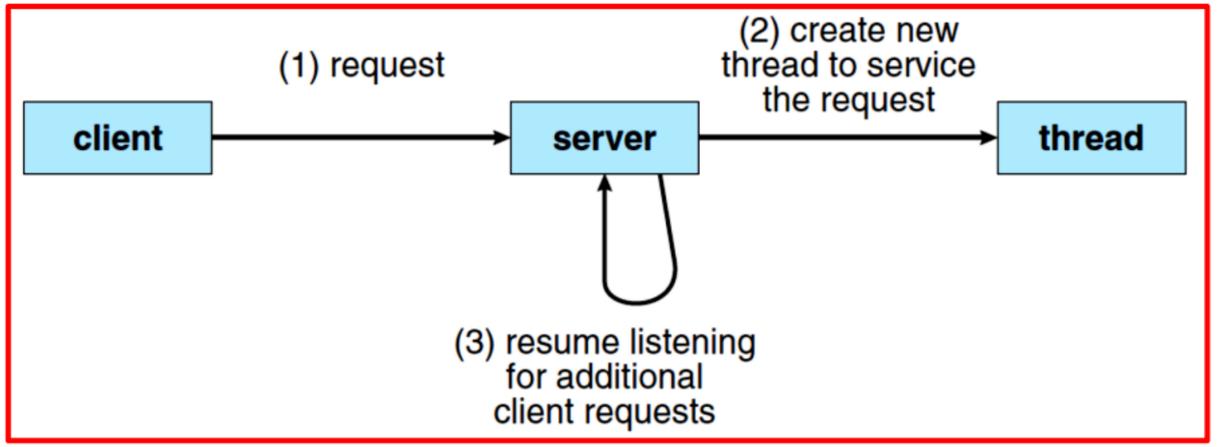
Let’s see what this is

Multithreading



- The primary difference is that threads within the same process run in a **shared memory space**, while processes run in separate memory spaces. Threads are not independent of one another like processes are, and as a result threads share with other threads their c**ode section, data section, and OS resources (like open files and signals).** But, like process, a thread has its own program counter (PC), register set, and stack space

Client - Server Model



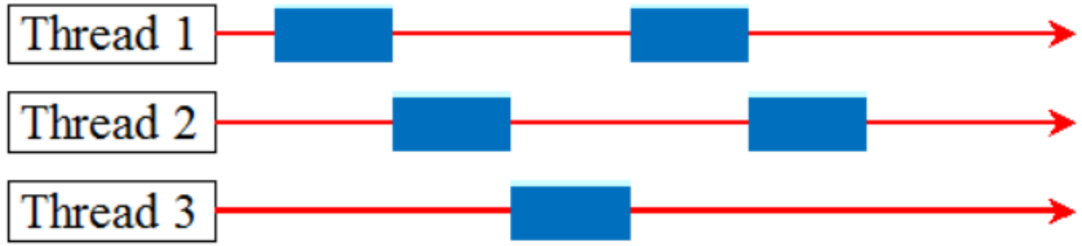
- Asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute **concurrently** and independently of one another.

- Because the threads are independent, there is typically little data sharing between them.

- Asynchronous threading is the strategy used in the multithreaded server illustrated in the above figure, and is also commonly used for designing responsive user interfaces.

Threads getting CPU

- Multiple Threads on Multiple CPUs



- Multiple Threads on Single CPUs

Benefits of Multithreading

- Most operating-system kernels are now multithreaded
- Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling
- For example, Solaris has a set of threads in the kernel specifically for interrupt handling;
  Linux uses a kernel thread for managing the amount of free memory in the system

Benefits of Multithreading

- **Responsiveness**: Multithreading an interactive application may allow a program to continue 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 by default

- **Economy**: thread creation consumes less time and memory than process creation

- **Scalability**:The benefits of multi-programming greatly increase in case of multiprocessor architecture, where threads may be running parallel on multiple processors. If there is only one thread then it is not possible to divide the processes into smaller tasks that different processors can perform. Single threaded process can run only on one processor regardless of how many processors are available. Multi-threading on a multiple CPU machine increases parallelism

How To

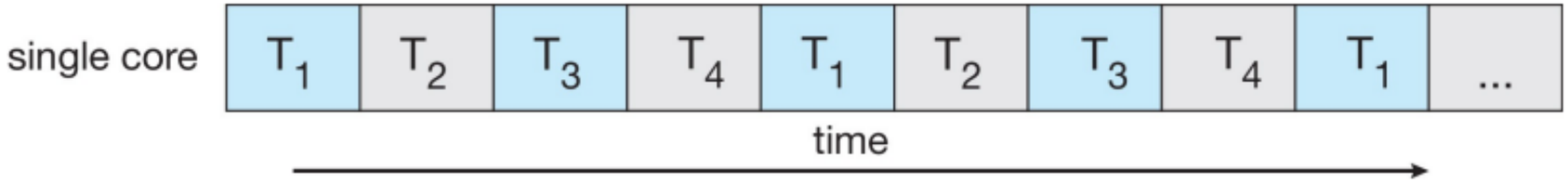
- Identifying tasks: Find areas that can be divided into separate, concurrent tasks
- Balance: Programmers must also ensure that the tasks perform equal work of equal value
- Data splitting: Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores
- Testing and debugging: Many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications

Multithreading vs Parallelism

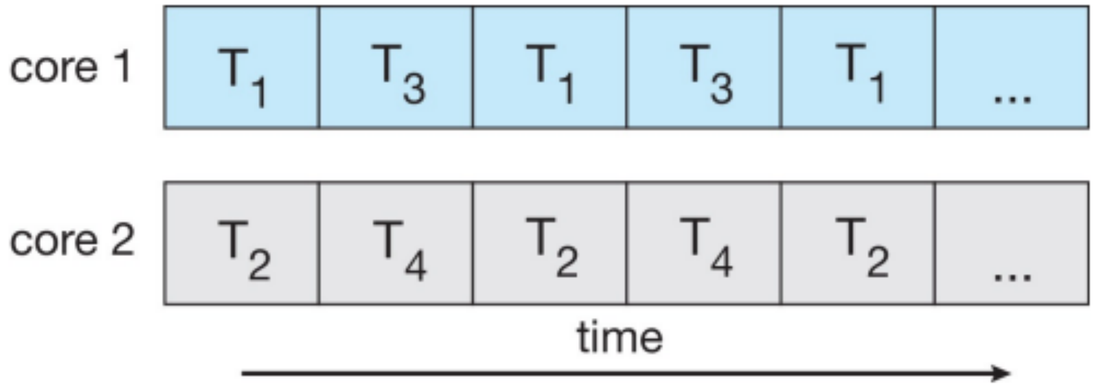
Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.
An application can be concurrent — but not parallel, which means that it processes more than one task at the same time, but no two tasks are executing at the same time instant

Multithreading vs Parallelism

- Concurrent Execution:

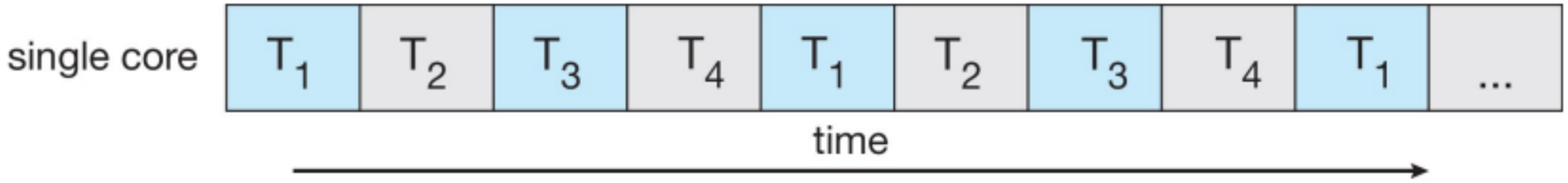


- Parallelism:

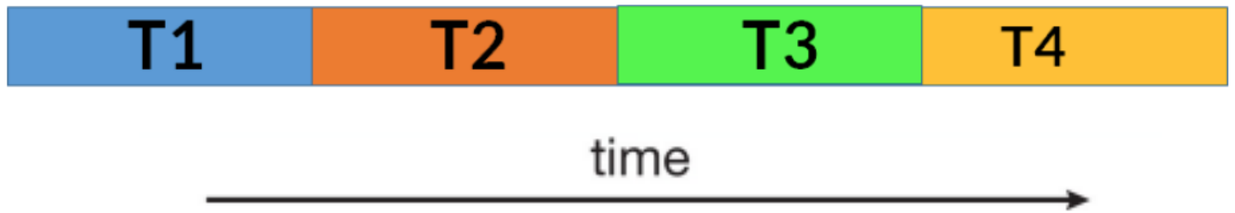


Note: Multithreading vs Parallelism

- Concurrent Execution:



- Sequential Execution:



Data vs Task Parallelism

- Data parallelism: Add N numbers. Divide the array into smaller portions and assign each portion to a core
- Task parallelism:
  - Core1: Find the mean value of an array of data
  - Core2: Find median value of an array of numbers.
- These are not mutually exclusive and some applications may contain both.
- Hardware threading helps make a processor core more power efficient through better utilization, something particularly attractive in servers but increasingly useful in all systems.
  - Imagine a system with 2 processors, each 8 core, and each core has 2 hardware threads. Such a machine will offer 32 hardware threads.

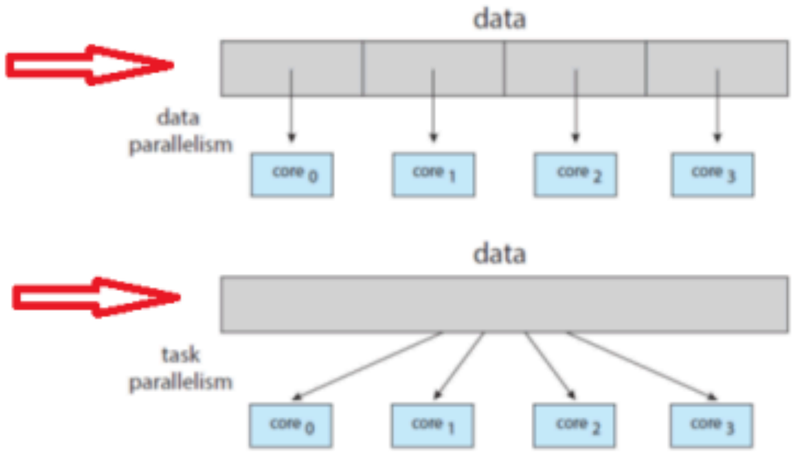
Data vs Task Parallelism

data parallelism

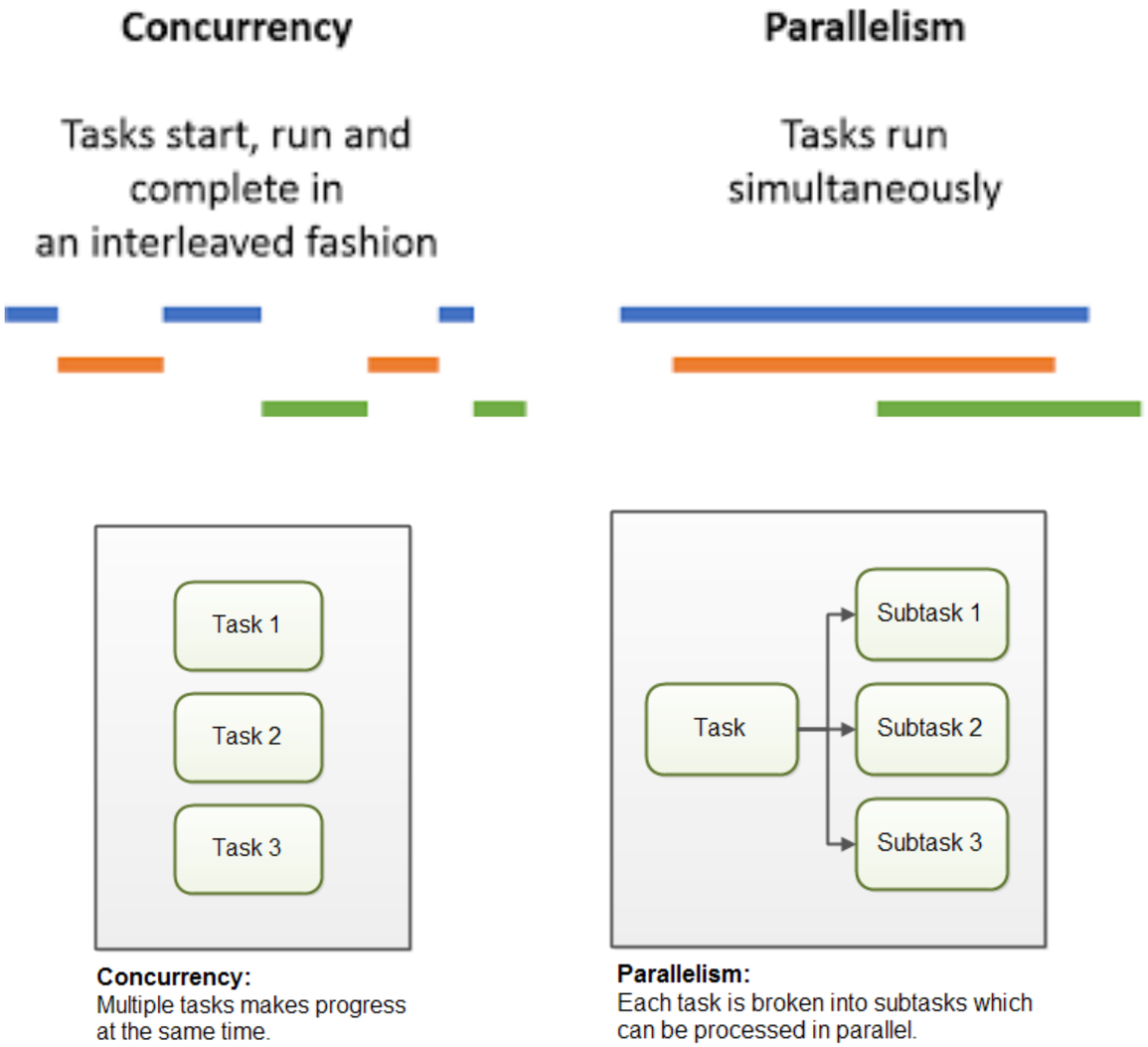
Add 1000 numbers: divide it into 4 parts

task parallelism

task1: find mean of these 1000 numbers
task2: find median of them
task3: sort them
task4: separate odds from evens



Summary: Concurrency vs Parallelism



- Concurrency means multiple tasks which start, run, and complete in overlapping time periods
- Parallelism is when multiple tasks or several parts of a task literally run at the same time (e.g. on a multi-core processor)

Amdahl's Law

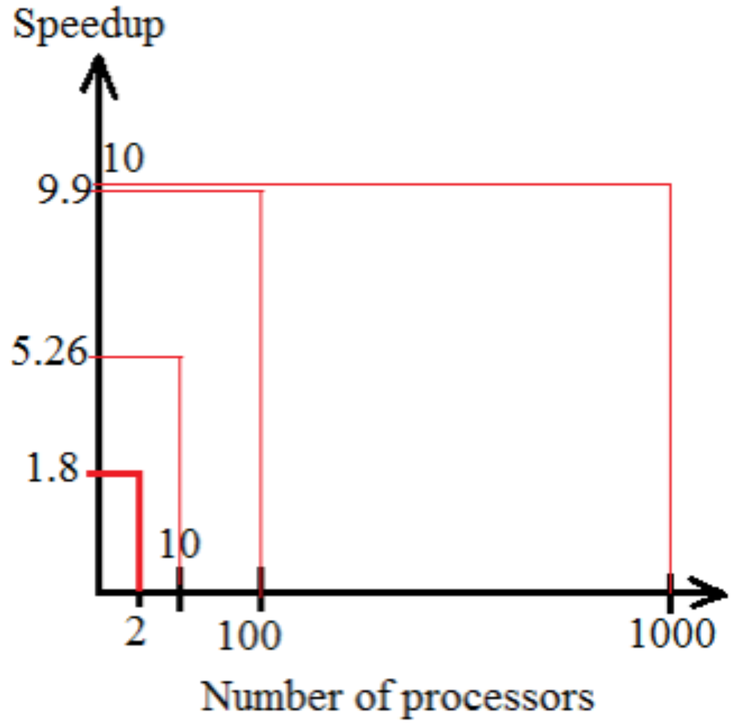
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores

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

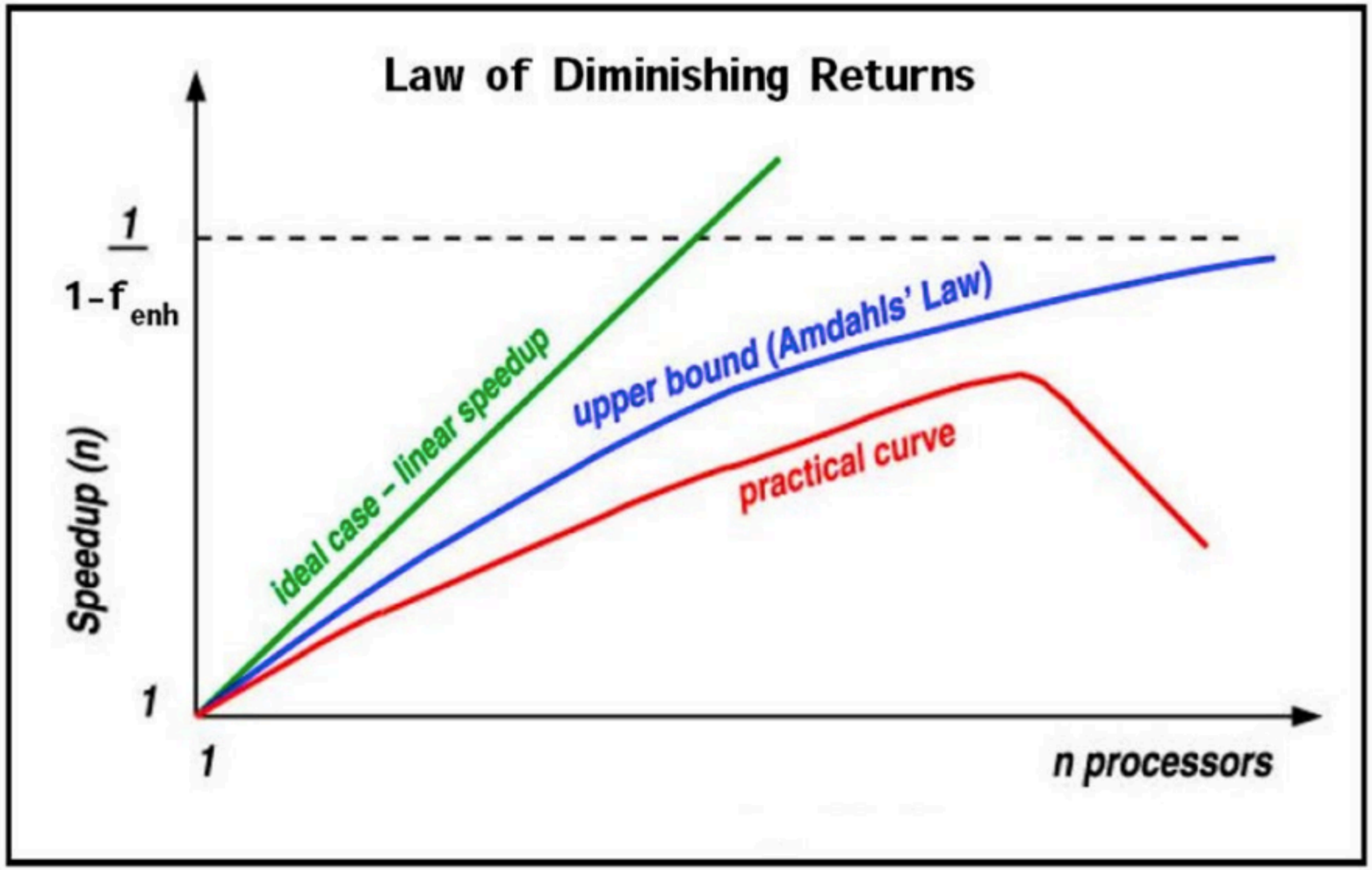
- That is, if application is 75% parallel and 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S

Amdahl's Law Example

- 90% of a program has codes that run in parallel. Also, 10% of the code is sequential
- What is the speedup using 2 processors?
- What is the speedup using 10 processors?
- What is the speedup using 100 processors?
- What is the speedup using 1000 processors?



Diminishing Returns in Amdahl's Law



Threads

Let’s see what this is

User Threads & Kernel Threads

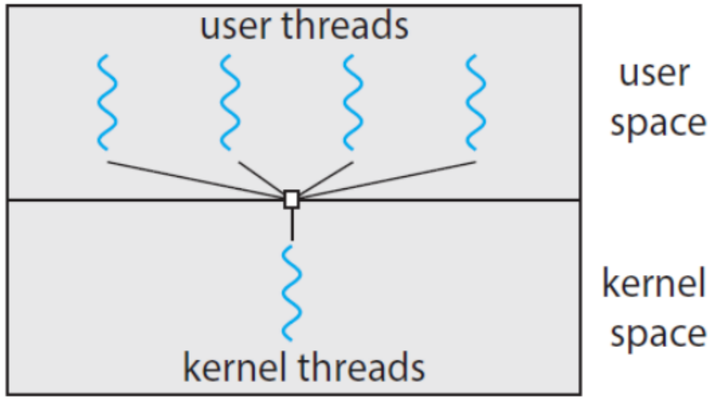
- A thread library provides the programmer with an API for creating and managing threads

- There are 2 primary ways of implementing a thread library

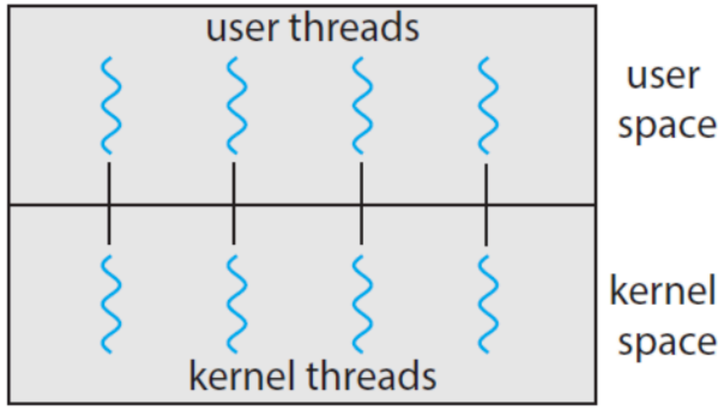
- The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

- The second approach is to implement a kernel level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

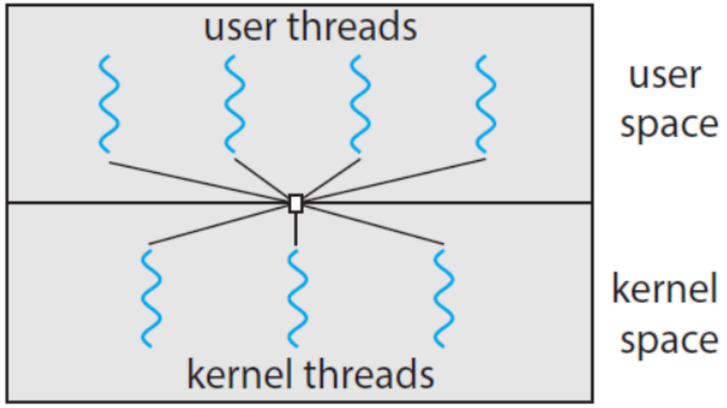
User Threads & Kernel Threads



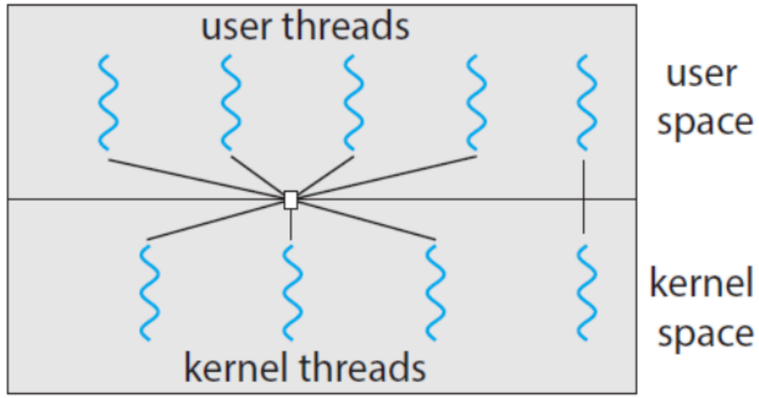
Many - To - One



One - To - One



Many - To - Many



Two Level Model

pthread

- 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)

pthread_create

- int pthread_create(
  pthread_t * thread, # pointer to unsigned integer that returns the thread id
  const pthread_attr_t * attr, # pointer to object that defines thread attributes. NULL default
  void * (*start_routine)(void *), # pointer to subroutine executed by thread
  void *arg); # pointer to arguments of start_routine

- The pthread_create() function starts a new thread in the calling process

- The new thread starts execution by invoking start_routine()

- argv is passed as the sole argument of start_routine()

- pthread_create() returns 0 on success, and it returns an error number when an error occurs

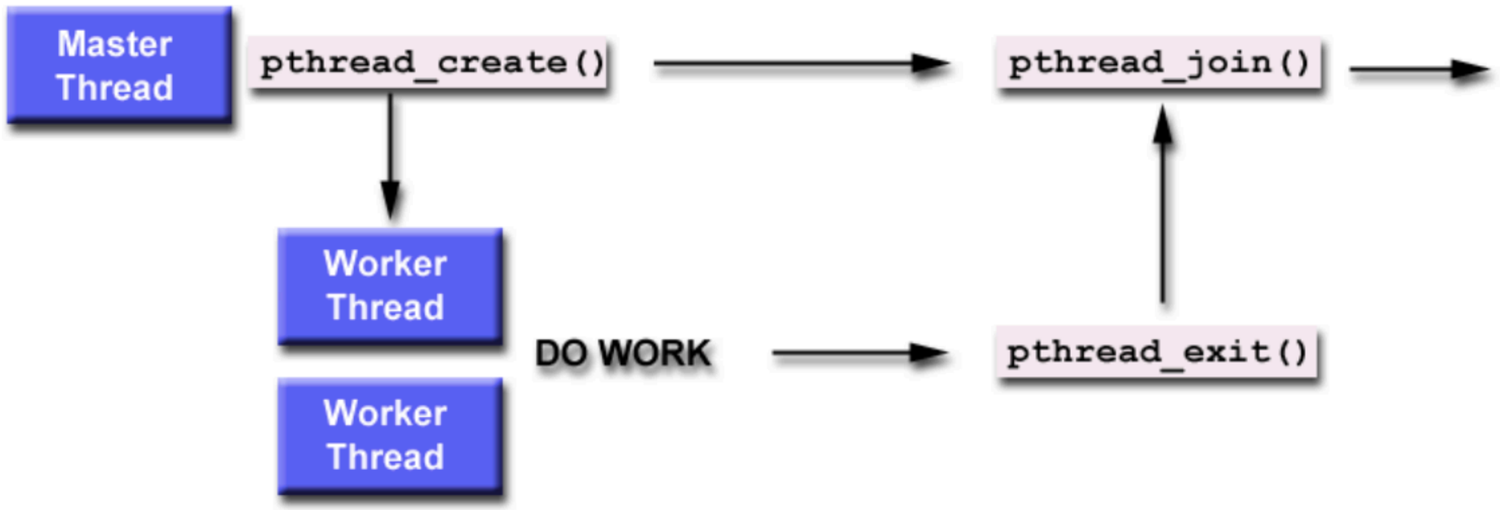
pthread termination

- Thread terminates in one of the following ways:
  - It calls pthread_exit(), specifying an exit status value
  - It returns from runner(). Equivalent to calling pthread_exit() with the value supplied in the return statement
  - It is canceled using pthread_cancel()
  - Any of the threads in the process calls exit()

pthread_exit

- void pthread_exit(void *retval);
- The pthread_exit() function is used to terminate a thread
- retval stores the return status of the thread terminated

Threads Hierarchy



pthread_join

- int pthread_join(
  pthread_t th, # thread id of the thread for which current thread waits
  void **thread_return # pointer where the exit status of thread th is stored
  );

- The pthread_exit() function is used to wait for the termination of a thread

- Blocks the calling thread until thread th has terminated

- retval stores the return status of the thread terminated

- Returns zero if successful, else error number

Implicit Threading

- Management of thousands of threads is difficult for programmers. This strategy, termed implicit threading, is a popular tool.
- Thread Pools
- Fork-Join
- OpenMP
- Grand Central Dispatch

Thread Pools

- Create a number of threads at start-up and place them into a pool, where they sit and wait for work
- Submit, awakened, queued
- Existing thread faster
- Limits on number of threads
- Use of threads, mechanics of threads

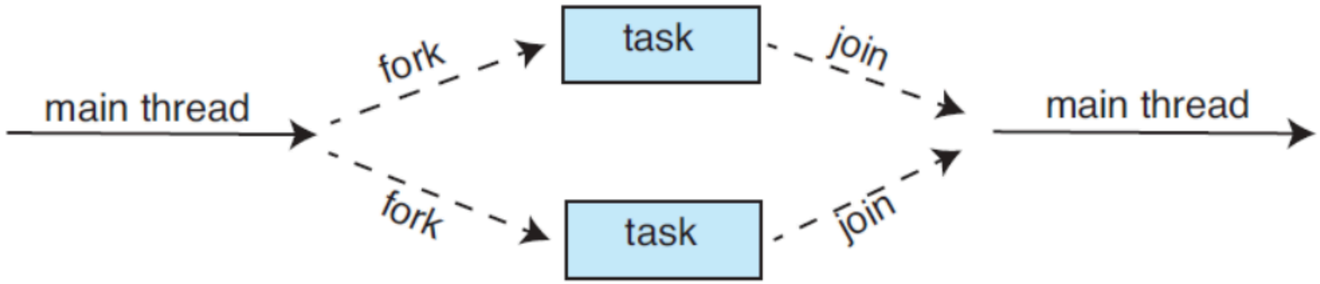
Thread Pools

- Servicing a request with an existing thread is often faster than waiting to create a thread

- A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads

- Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task.
  For example, the task could be scheduled to execute after a time delay or to execute periodically.

Fork-Join



- A fork gives you a brand new process, which is a copy of the current process, with the same code segments.
- As the memory image changes (typically this is due to different behavior of the two processes) you get a separation of the memory images (Copy On Write), however the executable code remains the same.
- Tasks do not share memory unless they use some Inter Process Communication (IPC) primitive

Output?

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

fork() output

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

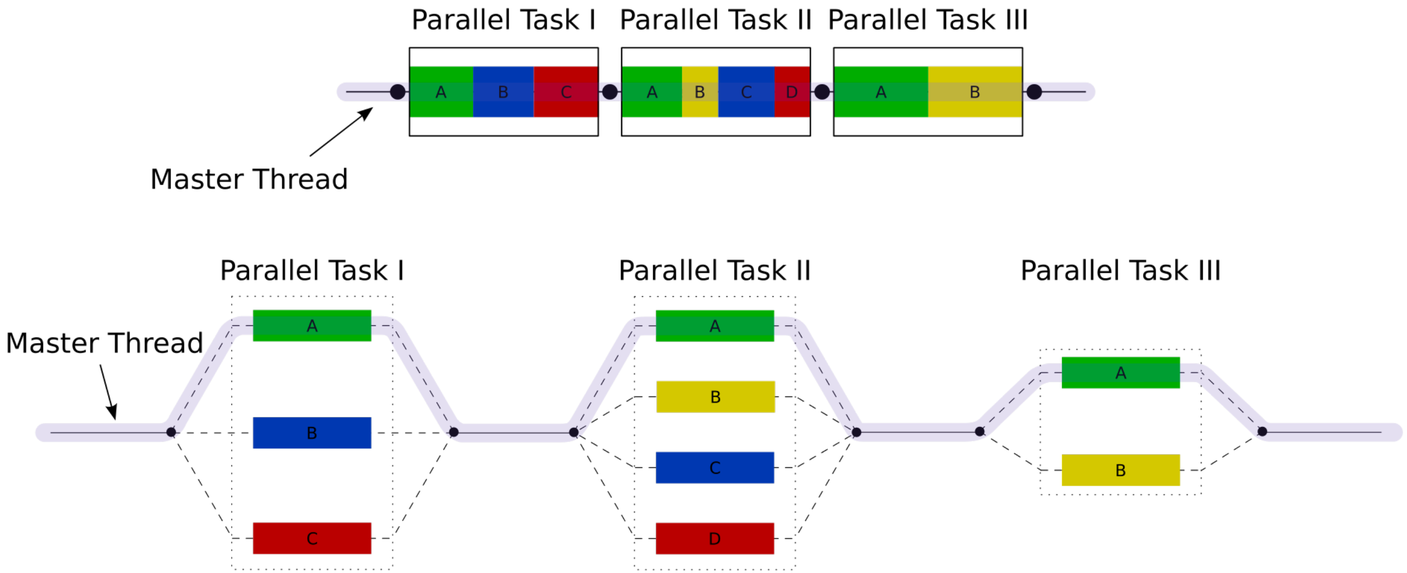
Output:

hello
hello
hello
hello
hello
hello
hello
hello

```
fork ();  // Line 1
fork ();  // Line 2
fork ();  // Line 3

     L1      // There will be 1 child process
   /   \     // created by line 1.
 L2     L2   // There will be 2 child processes
/ \   / \   //  created by line 2
L3 L3 L3 L3  // There will be 4 child processes
             // created by line 3
```

OpenMP



- OpenMP is a set of compiler directives as well as an API for programs that provides support for parallel programming in shared memory environments. OpenMP identifies parallel regions as blocks of code that may run in parallel.

https://en.wikipedia.org/wiki/OpenMP

Grand Central Dispatch

- 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"); }
- Blocks placed in dispatch queue
- Assigned to available thread in thread pool when removed from queue

https://en.wikipedia.org/wiki/Grand_Central_Dispatch