CSE2008: Operating Systems

L13 & L14: Multithreading Concepts

Dr. Subrata Tikadar

SCOPE, VIT-AP University

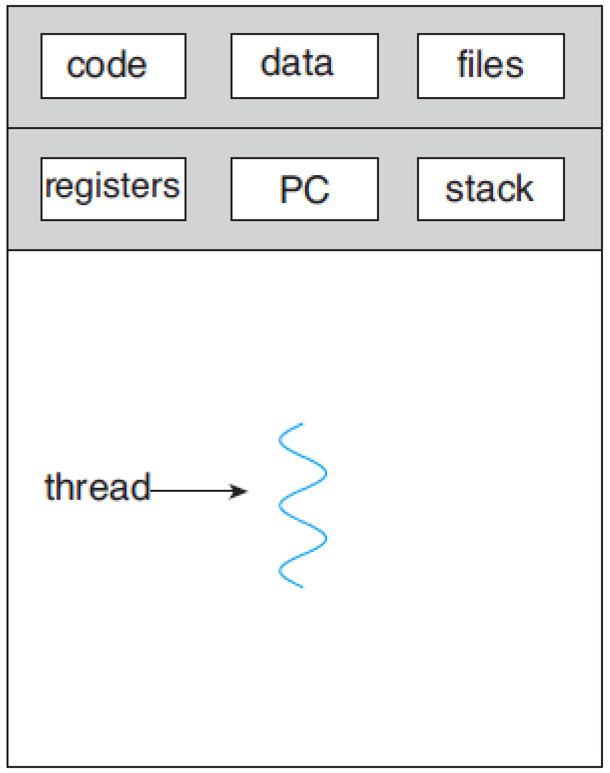
Recap

- Introductory Concepts
- Process Fundamentals
- IPC
- CPU Scheduling Algorithms

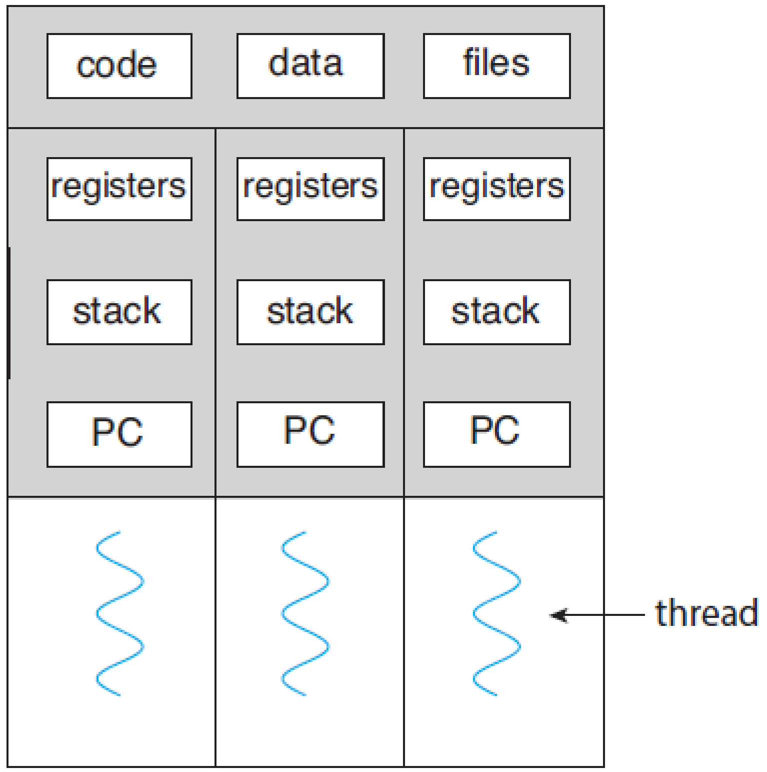
Outline

- Concepts of Threads
- Multithreading Models
- Thread Libraries

Concepts of Threads



single-threaded process



multithreaded process

Concepts of Threads

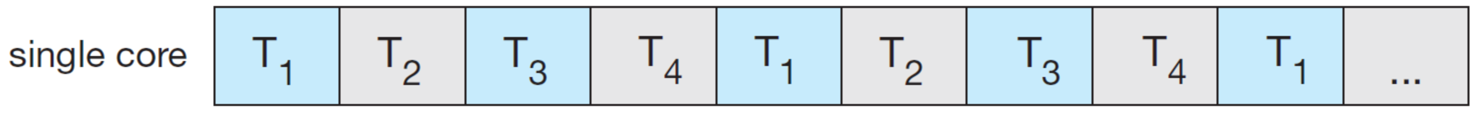
- Motivation for Multithreading
  - Most software applications that run on modern computers and mobile devices are multithreaded

    Examples:

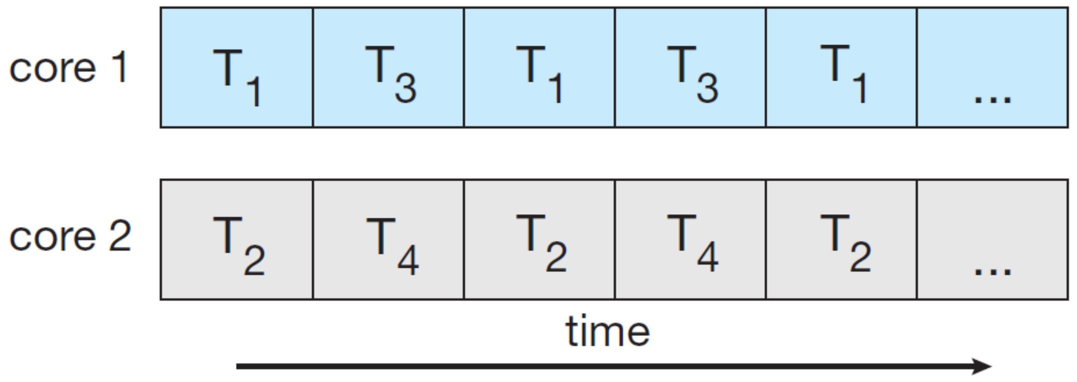
    1. A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background
    2. A web browser might have one thread display images or text while another thread retrieves data from the network
    3. An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image

Concepts of Threads

- Multicore Programming



Concurrent execution on a single-core system.



Parallel execution on a multicore system.

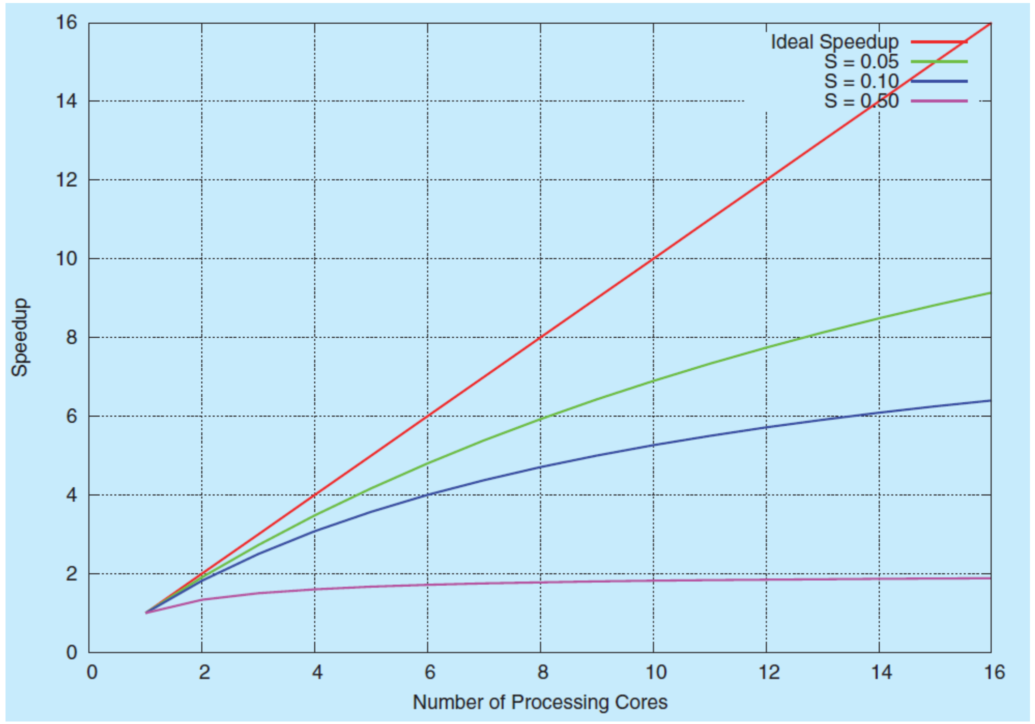
Concepts of Threads

- Programming Challenges
  - Identifying tasks
    - Separating independent and sequential tasks
  - Balance
    - It should be assured that parallel tasks perform equal work of equal value
  - Data splitting
    - Data accessed and manipulated by the tasks must be divided to run on separate cores
  - Data dependency
    - The data accessed by the tasks must be examined for dependencies between two or more tasks – in case of dependency, proper **synchronization technique** must be maintained

Concepts of Threads

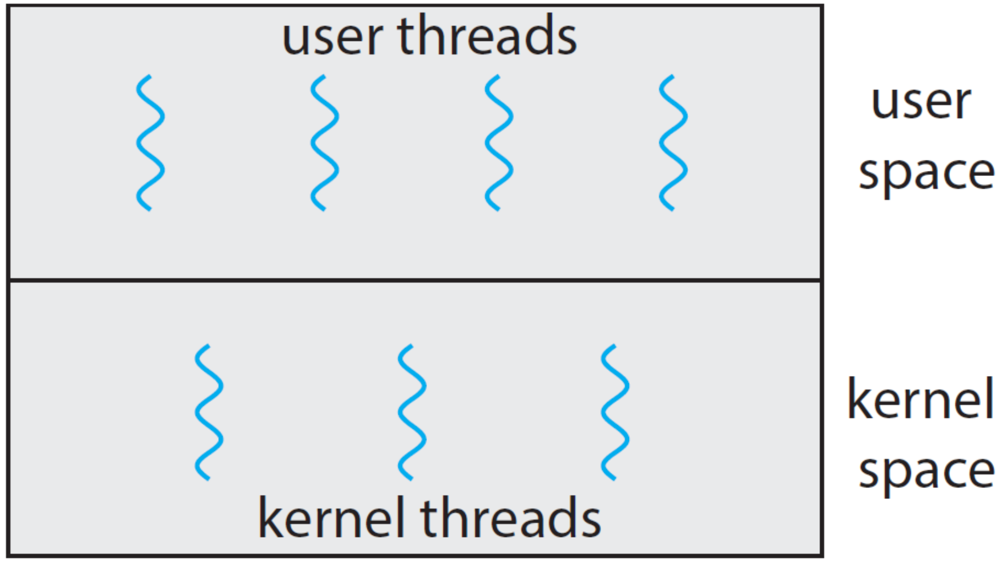
- AMDAHL'S LAW*
  - If S is the portion of the application that must be performed serially on a system with N processing cores, the formula appears as follows:

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



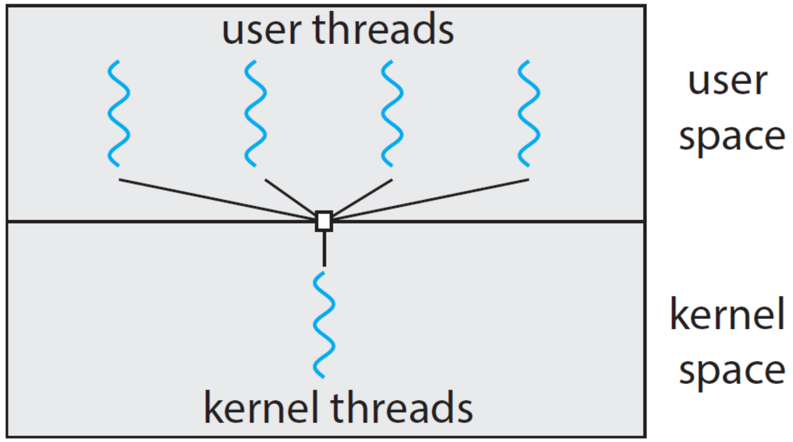
Multithreading Models

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



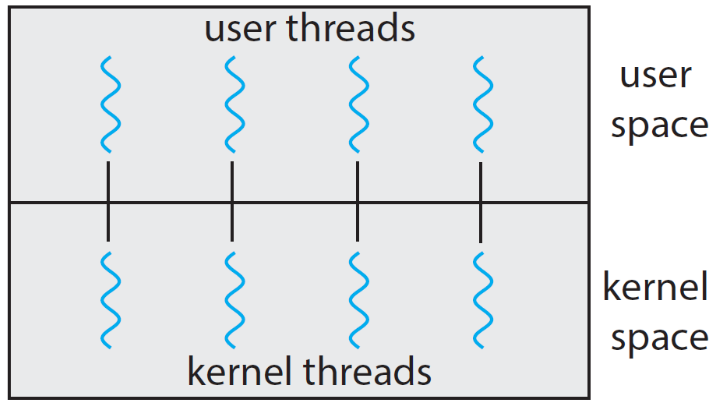
Multithreading Models

- Many-to-One Model
  - Maps many user-level threads to one kernel thread
  - Thread management is done by the thread library in user space, so it is efficient
  - However
    - the entire process will block if a thread makes a blocking system call
    - because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems



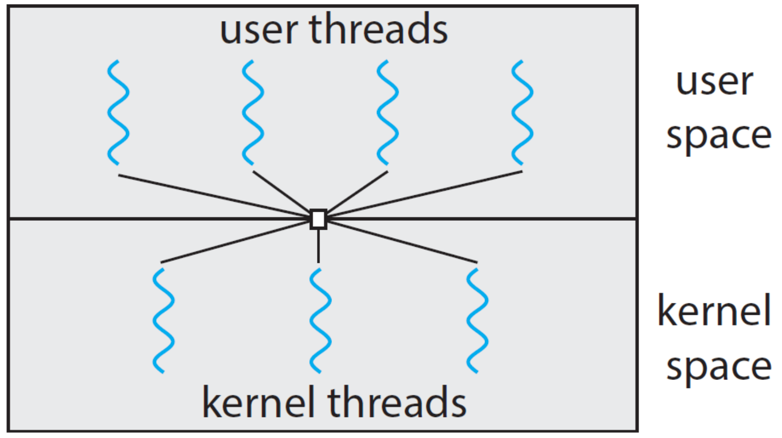
Multithreading Models

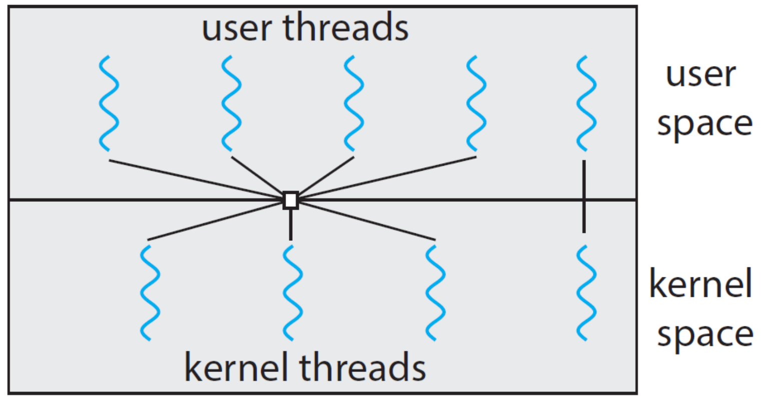
- One-to-One Model
  - Maps each user thread to a kernel thread
  - Provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call
  - Also allows multiple threads to run in parallel on multiprocessors
  - However
    - creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of a system



Multithreading Models

- Many-to-Many Model
  - Multiplexes many user-level threads to a smaller or equal number of kernel threads
  - The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a system with eight processing cores than a system with four cores)





Thread Libraries

- Provides the programmer with an API for creating and managing threads
- Two primary ways of implementing a thread library
  - to provide a library entirely in user space with no kernel support
  - to implement a kernel-level library supported directly by the operating system
- Three main thread libraries are in use now-a-days
  - Pthreads
  - Windows Threads
  - Java Threads

$$sum = \sum_{i=1}^{N} i$$

Thread Libraries

$$sum = \sum_{i=1}^{N} i$$

- 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; /* the thread identifier */
  pthread_attr_t attr; /* set of 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);
}
```

```
/* 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);
}
```

```
#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);
```

Thread Libraries

$$sum = \sum_{i=1}^{N} i$$

- Windows Threads

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

```
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* 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 */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

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

Thread Libraries

$$sum = \sum_{i=1}^{N} i$$

- Windows Threads
  - In situations that require waiting for multiple threads to complete, the WaitForMultipleObjects() function is used. This function is passed four parameters:
    1. The number of objects to wait for
    2. Apointer to the array of objects
    3. Aflag indicating whether all objects have been signaled
    4. A timeout duration (or INFINITE)

Example: If THandles is an array of thread HANDLE objects of size N, the parent thread can wait for all its child threads to complete with this statement:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

Thread Libraries

$$sum = \sum_{i=1}^{N} i$$

- Java Threads
  - Two techniques for explicitly creating threads in a Java program
    - Create a new class that is derived from the Thread class and to override its run() method
    - Define a class that implements the Runnable interface

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

```
Thread worker = new Thread(new Task());
worker.start();
```

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Thread Pool

- Create a number of threads at start-up and place them into a pool, where they sit and wait for work
- When a server receives a request, rather than creating a thread, it instead submits the request to the thread pool and resumes waiting for additional requests
- If there is an available thread in the pool, it is awakened, and the request is serviced immediately
- If the pool contains no available thread, the task is queued until one becomes free
- Once a thread completes its service, it returns to the pool and awaits more work

Thread Pool

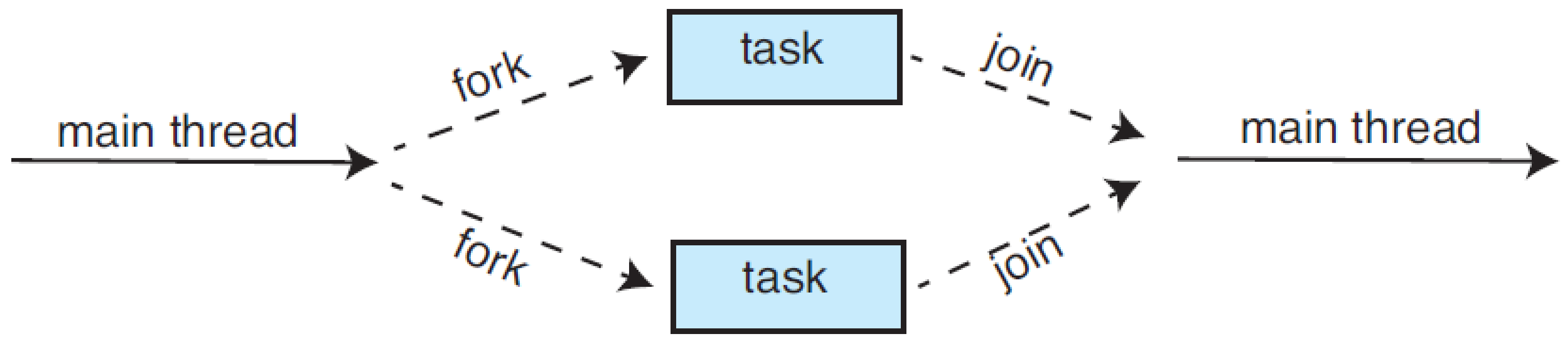
- Benefits:
  - Servicing a request with an existing thread is often faster than waiting to create a thread.
  - A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.
  - Separating the 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.

Thread Pool

- Java Thread Pools
    1. Single thread executor—newSingleThreadExecutor()—creates a pool of size 1.
    2. Fixed thread executor—newFixedThreadPool(int size)—creates a thread pool with a specified number of threads.
    3. Cached thread executor—newCachedThreadPool()—creates an unbounded thread pool, reusing threads in many instances.

Fork Join

- The strategy for thread creation covered is often known as the fork-join model
- The main parent thread creates (forks) one or more child threads and then waits for the children to terminate and join with it, at which point it can retrieve and combine their results



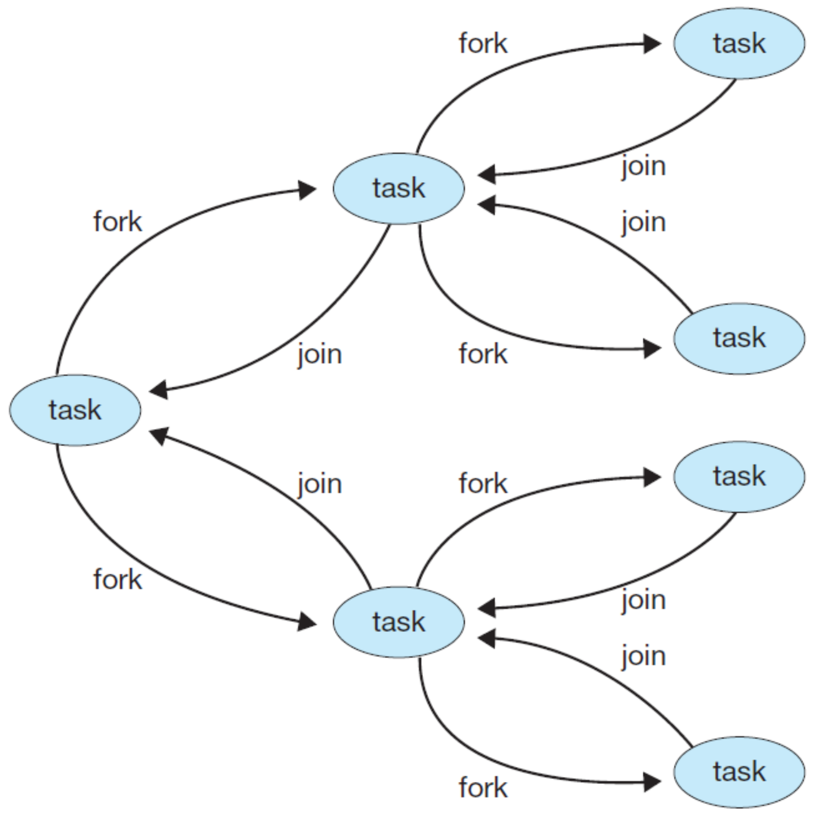
Fork Join

- Fork Join in Java

```
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
```



Self Study

- Threading Issues

Reference

- Abraham Silberschatz , Peter B. Galvin, Greg Gagne, “Operating System Concepts”, Addison Wesley, 10th edition, 2018
  - Chapter 4: Section 4.1 – 4.6

Next

- Module-3 [Process Coordination and Deadlock]

Thank You