Threads

Moumita Patra
July-Nov 2019
Ref:- Galvin, Gagne

Thread

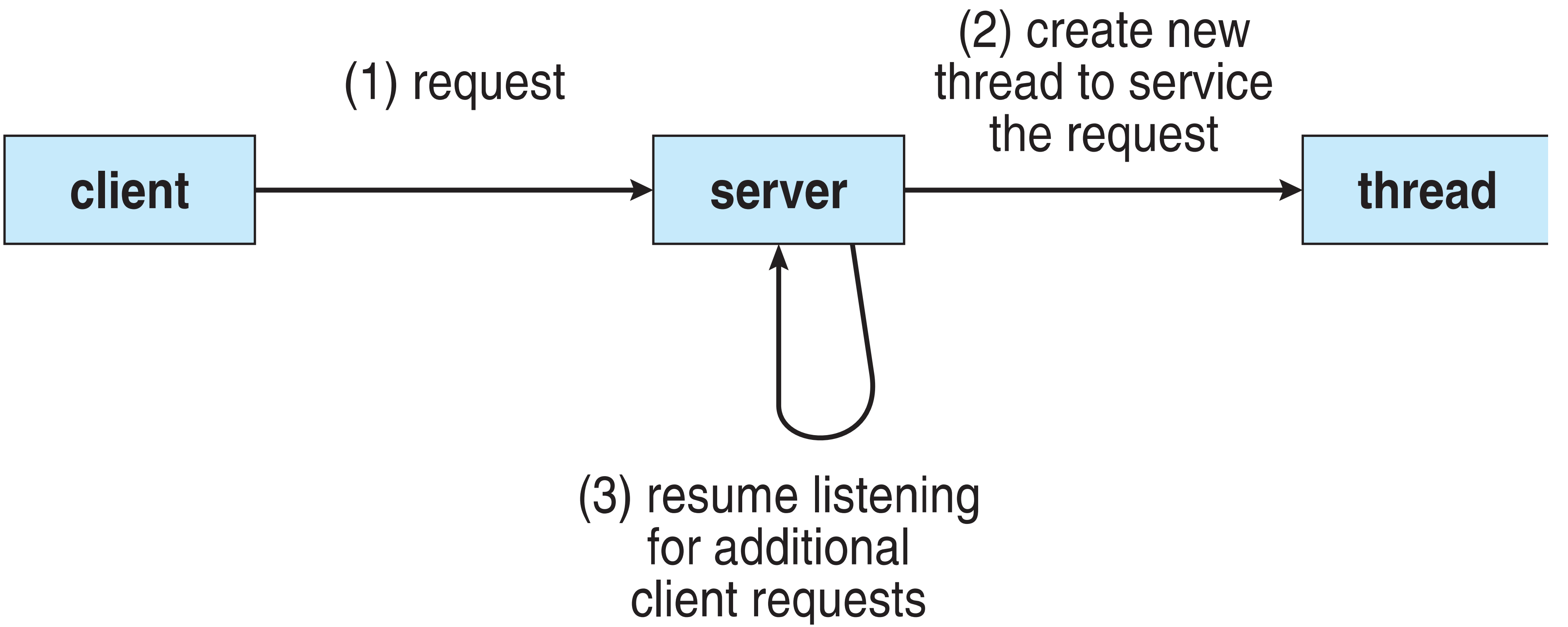
- A thread is a basic unit of CPU utilization.
- It comprises of-
  - Thread ID
  - A program counter
  - A register set
  - A stack
- It shares with other threads belonging to the same process its code section, data section, and other resources, such as open files and signals.

Single and Multithreaded Processes

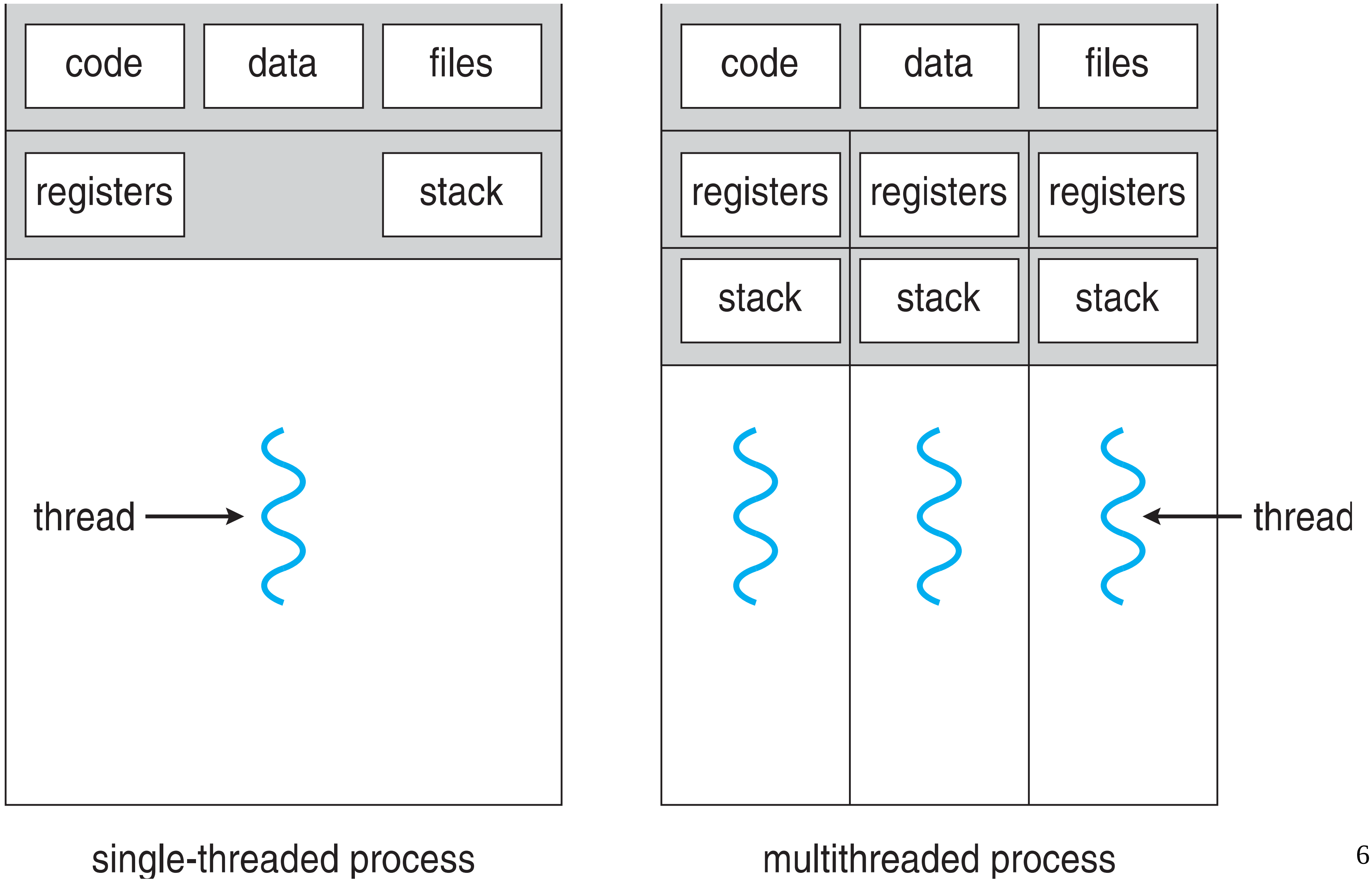
Multithreading

- If a process has multiple threads of control, it can perform more than one task at a time.
- Benefits:-
  - Responsiveness
  - Resource sharing
  - Economy
  - Scalability

Multithreaded Server Architecture



Single and Multithreaded Processes



Multicore Programming

- Multiple computing cores across CPU chips or within CPU chips
- Provides a mechanism for more efficient use of multiple computing cores and improved concurrency.

Multicore Programming

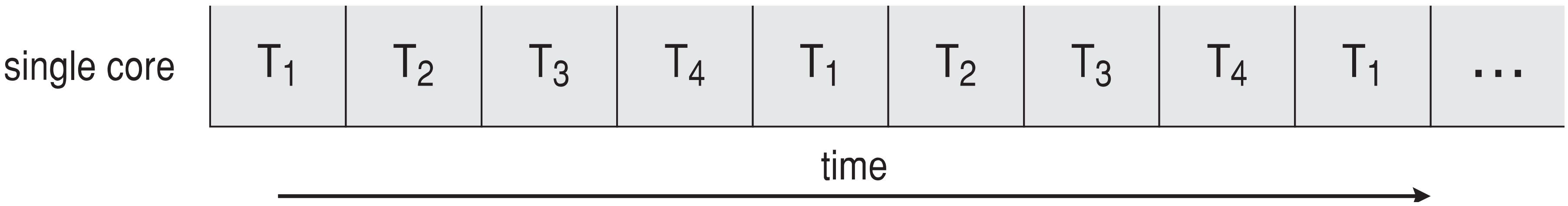
- Challenges in programming for multicore systems:
  - Identifying the tasks
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

Concurrency and Parallelism

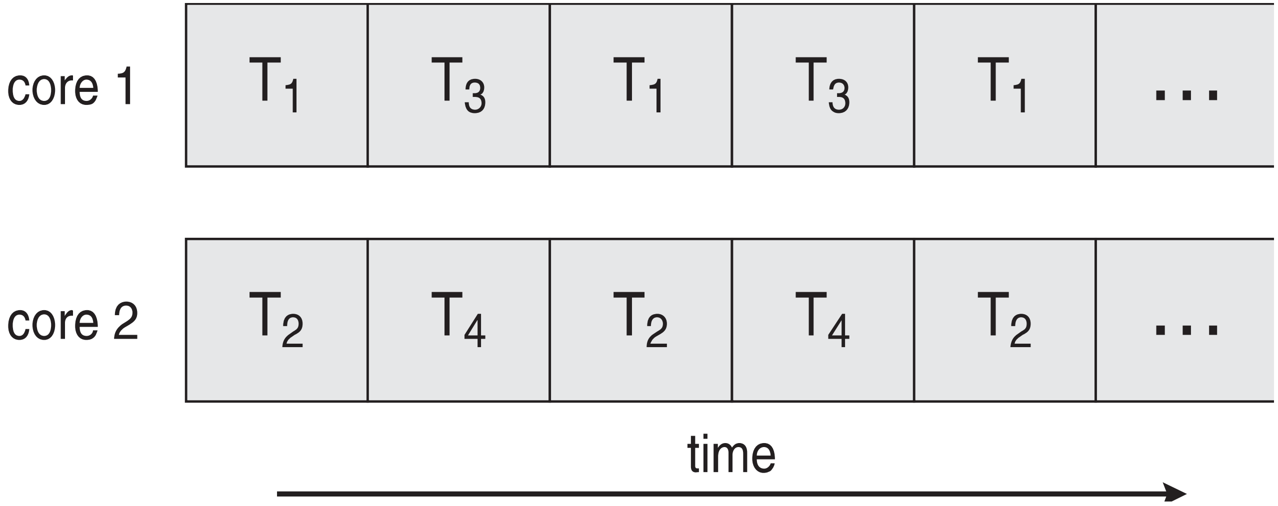
- **Parallelism**- a system that can perform more than one task simultaneously.
- **Concurrency**- supports more than one task by allowing all the tasks to make progress.

Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:



Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism-** distributing subsets of the same data across multiple cores, same operation on each
  - **Task parallelism-** distributing threads across cores, each thread performing unique operation
- As number of threads grows, so does architectural support for threading

Multithreading Models

User Threads and Kernel Threads

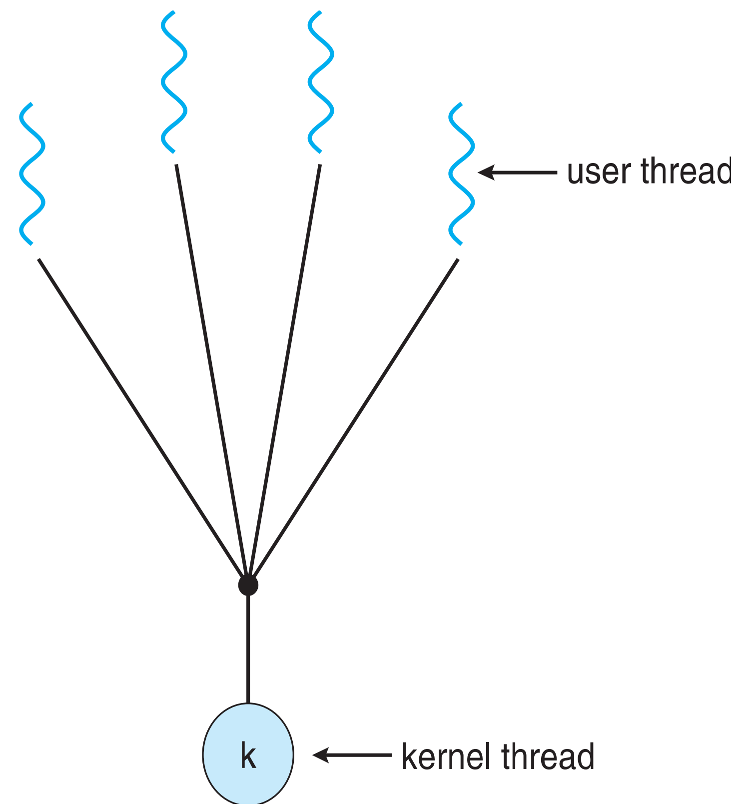
- User threads- management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads
- Kernel threads- Supported by kernel
- Examples- virtually all general purpose OS, such as:
  - Windows
  - Solaris
  - Linux
  - Mac OS X

Multithreading Models

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

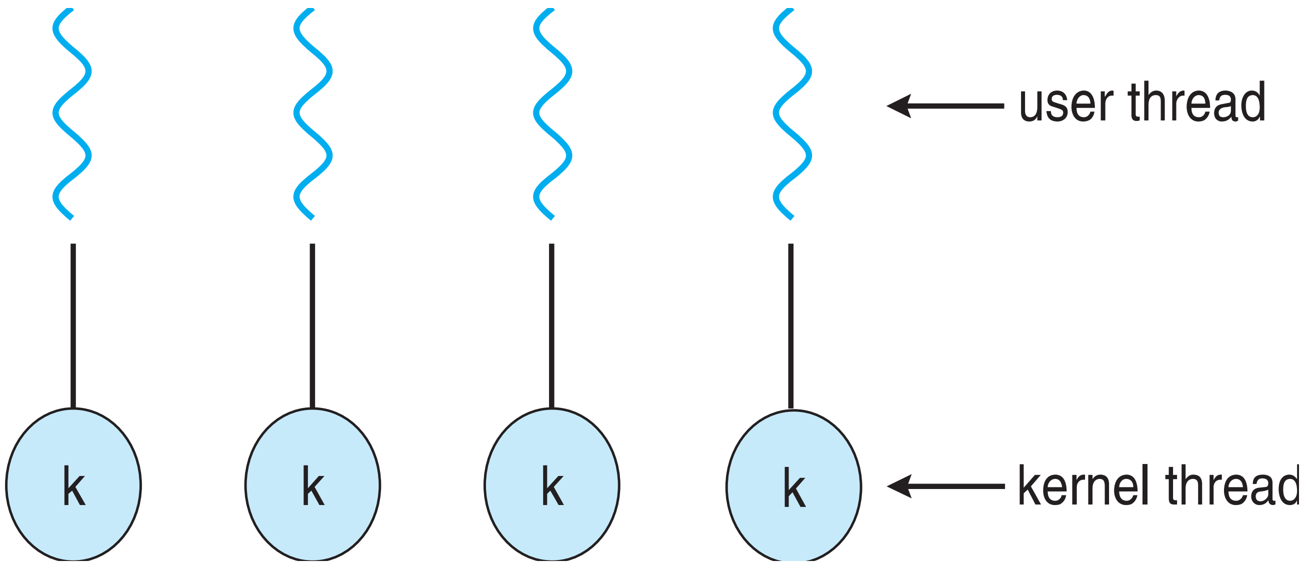
Many-to-One

- Many user-level threads mapped to single kernel thread
- If one thread is blocked, it causes all to block
- Multiple threads may not run in parallel on multicore system becasue only one may be in kernel at a time
- Examples- Solaris Green Threads
- GNU Portable Threads



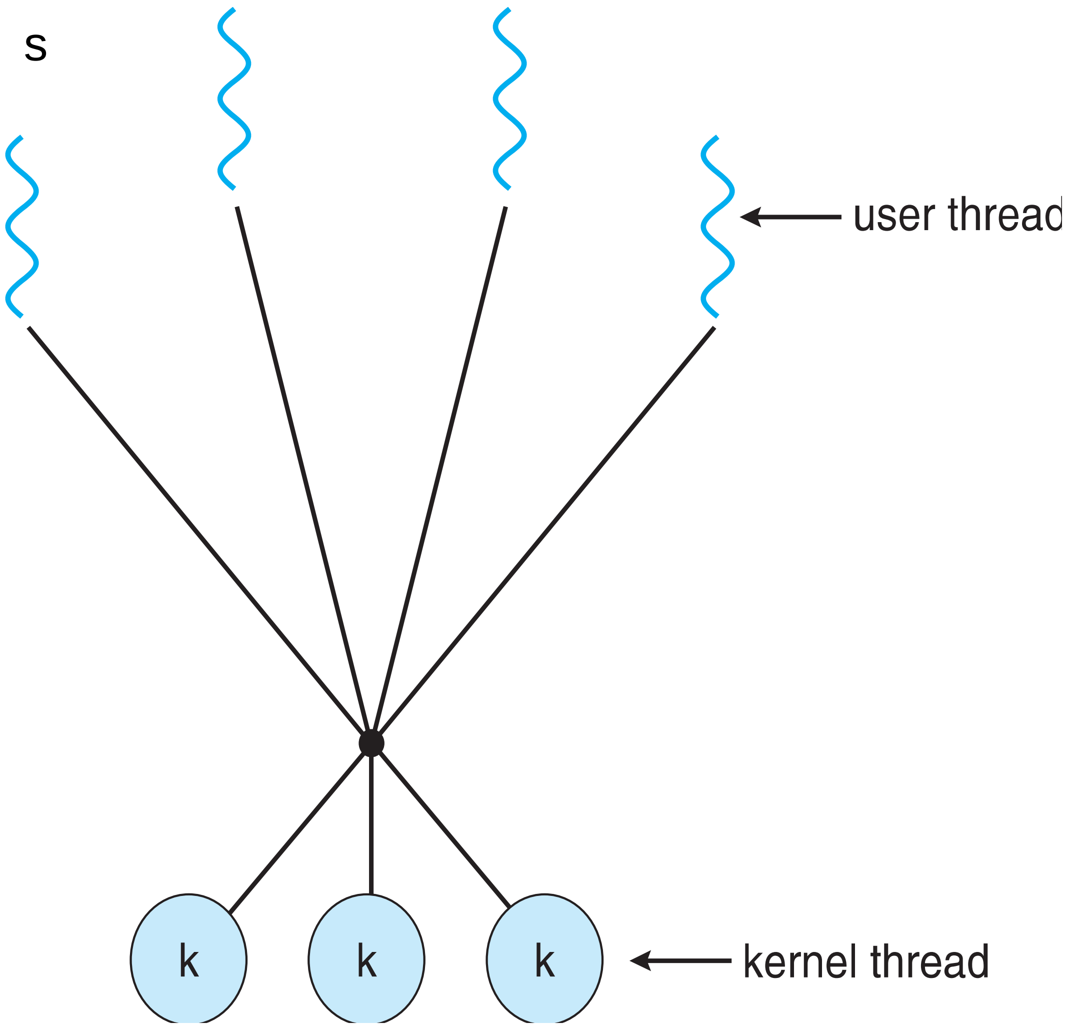
One-to-One

- Each user-level thread maps to kernel thread
- Creating user-level thread creates a kernel thread
- More concurrency than Many-to-One
- Number of threads per process is sometimes restricted due to overhead
- Examples:-
- Windows
- Linix
- Solaris



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allow OS to create a sufficient number of kernel threads
- Example- Solaris prior version of 9



Thread Librares

- 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

- A POSIX standard API for thread creation and synchronization
- May be provided as either user-level or kernel-level
- Specification for thread behaviour, not an implementation
- API specifies behaviour of the thread library, implementation is up to development of the library
- Common in UNIX OS

Pthreads Example

```
#include <pthread.h>
#include <stdio.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 */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

```
    /* get the default attributes */
    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 begin control 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);
}
```

Pthread code for joining 10 threads

```
#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);
```

Windows Multithreaded C Program

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

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

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

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Windows Multithreaded C Program (Cont.)

```
  /* 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 */

  if (ThreadHandle != NULL) {
      /* now wait for the thread to finish */
     WaitForSingleObject(ThreadHandle,INFINITE);

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

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

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS

Java threads may be created by:

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

→ Extending Thread class

→ Implementing the Runnable interface

Java Multithreaded Program

```
class Sum
{
  private int sum;

  public int getSum() {
   return sum;
  }

  public void setSum(int sum) {
   this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }

  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setSum(sum);
  }
}
```

Java Multithreaded Program (Cont.)

```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0)
       System.err.println(args[0] + " must be >= 0.");
      else {
       Sum sumObject = new Sum();
       int upper = Integer.parseInt(args[0]);
       Thread thrd = new Thread(new Summation(upper, sumObject));
       thrd.start();
       try {
          thrd.join();
          System.out.println
                    ("The sum of "+upper+" is "+sumObject.getSum());
       } catch (InterruptedException ie) { }
      }
     }
     else
      System.err.println("Usage: Summation <integer value>"); }
}
```

Implicit Threading

- For supporting multithreaded applications
- **Implicit threading**- creation and management of threads by compilers and run-time libraries
- Growing in popularity because, as numbers of threads increase, maintaining program correctness becomes more difficult with explicit threads
- Three methods:
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch

Thread Pools

- Create a number of threads in a pool where they wait for work
- When a server receives a request, it awakens a thread from this pool and passes it to the request for service
- Once the thread completes its service, it returns to the pool and awaits for more work.
- If the pool contains no available thread, the server waits until one becomes free.

Benefits-

- Servicing a request with an existing thread is faster than waiting to create a thread
- A thread pool limits the number of threads that exist at any point

  Important on systems that cannot support a large number of concurrent threads
- Different strategies can be used for running the task as creation is separate
- Number of threads in the pool can be set heuristically based on factors such as- number of CPUs in the system, amount of physical memory, and the expected number of concurrent client requests

OpenMP

- Set of compiler directives and API for programs written in C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies parallel regions as blocks of code that may run in parallel.
- Application developers insert compiler directives into their code at parallel regions.
- These directives instruct the openMP run-time library to execute the region in parallel.

Grand Central Dispatch (GCD)

- Technology for Mac OS X and iOS operating systems.
- Combination of extensions to C language, an API, and a run-time library that allows application developers to identify sections of code to run in parallel.
- Extensions are known as blocks.
- GCD schedules blocks for run-time execution by placing them on a dispatch queue.
- When it removes a block from a queue, it assigns the block to an available thread from the thread pool it manages.

Grand Central Dispatch

Two types of dispatch queue:

- **Serial**- blocks placed on a serial queue are removed in FIFO order
- Each process has its own queue
- Once a block has been removed from the queue, it must complete execution before another block is removed
- **Concurrent**- blocks removed in FIFO order but several blocks may be removed at a time
- There are 3 system-wide concurrent dispatch queues distinguished according to priority- low, default, and high

Threading Issues

- Semantics of fork() and exec() system calls
- Signal handling
- Thread cancellation
- Thread-local storage
- Scheduler activations

Semantics of fork() and exec()

- Does fork() duplicate only the calling thread or all threads?- some OS have two versions of fork()
- If thread invokes exec(), the program specified in the parameter to exec() will replace the entire process including threads
- Which version of fork() to use depends on the application and when exec() is called

Signals

- Signals are used in UNIX systems to notify a process about the occurence of a particular event
- Two types-
- Synchronous- delivered to the same process that performed the operation that caused the signal
- Asynchronous- when a signal is generated by an event external to a running process, that process receives the signal asynchronously

Signal Handling

- A signal handler is used to process signals
  - ➢ Signal is generated by particular event
  - ➢ Signal is delivered to a process
  - ➢ Signal is handled by either- default or user-defined handler
- Every signal has default handler that kernel runs when handling signal
- User-defined signal handler can override default

Challenges

Where should a signal be delivered for multi-threaded system?

- 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

Thread Cancellation

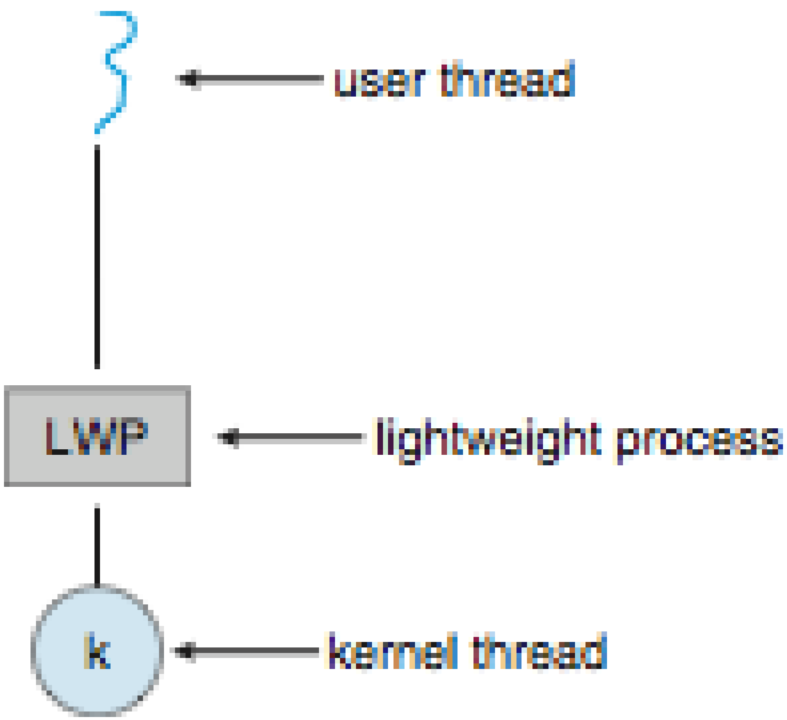
- Terminating a thread before it has completed
- Thread to be canceled is referred as target thread
- Two approaches:
  - Asynchronous cancellation- One thread immediately terminates the target thread
  - Deferred cancellation- the target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion

Thread-Local Storage (TLS)

- Data that is local to a thread
- TLS allows each thread to have its own copy of data
- Useful when you do not have control over thread creation process, i.e, in case of thread pool
- Different from local variables
  - ➢ Local variables are visible only during a single function invocation
  - ➢ TLS data are visible across function invocations but are unique to each thread (similar to static data)

Scheduler Activations

- Both many-to-many and two-level models require communication to maintain the appropriate number of 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 on which process can schedule user thread to run
  - Each LWP attached to kernel thread



Scheduler Activations

- Scheduler activations provide upcalls- a communication mechanism from the kernel to the upcall handler in the thread library
- This communication allows an application to maintain the correct number of kernel threads

NOTE:

A thread is a flow of control within a process