Multithreading:

What is thread.

- A thread is a basic unit of CPU utilization
- These are created within a process.
- It comprises of
    - thread ID
    - program counter
    - register set
    - Stack
- Other threads belonging to the same process shares
    - code section
    - data section
    - operating-system resources
        - such as open files
        - Signals

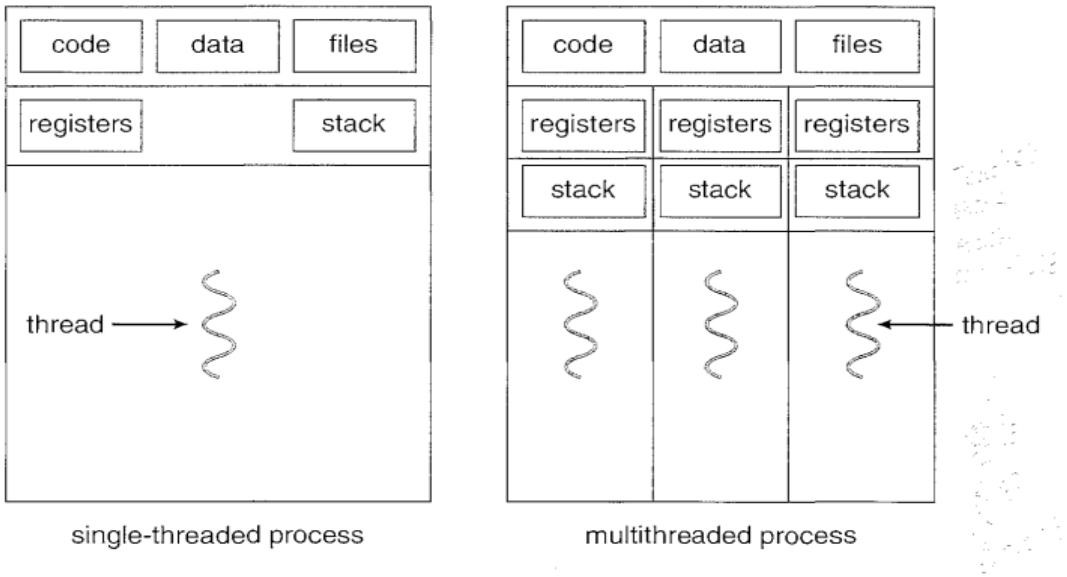


Figure 4.1 Single-threaded and multithreaded processes.

Multithreading

refers to the ability of an operating system to support multiple threads of execution within a single process.

Real life examples:

1. Multithreaded Server
   - Problem:
     i. The web server accepts client requests.
     ii. A busy Web server may have several (perhaps thousands of) clients concurrently accessing it.
     iii. If the Web server ran as a traditional single-threaded process, it would be able to service only one client at a time; a client might have to wait a very long time for its request to be serviced.
   - Solution using multiple processes
     i. When the server receives a request, it creates a separate process to service that request => but very heavy and lot of resources and time overload (Not used)
   - Solution using multithreading server (In use)
     i. If the Web-server process is multithreaded, the server will create a separate thread that listens for client requests.
     ii. When a request is made, rather than creating another process, the server will create a new thread to service the request
     iii. Resume listening for additional requests

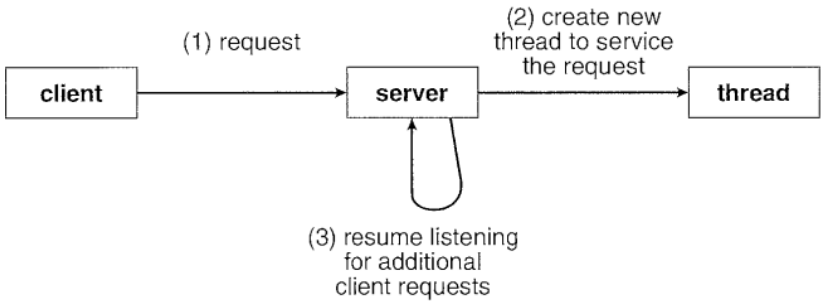


Figure 4.2 Multithreaded server architecture.

1. Foreground/Background

In a spreadsheet program, one thread could display menus and read user input, while another thread executes user commands and updates the spreadsheet.

2. Asynchronous Processing

Backing up in background

3. Faster Execution

Read one set of data while processing another set

What are Benefits Multithreaded systems?

Responsiveness: A program will continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

Example.

1. Code Editor (VS code). Write a code (1thread), error highlight (1thread), autosave (thread)
2. a multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.

Resource sharing. In Processes, you may only share resources through 1) shared memory 2) message passing. Such techniques must be explicitly arranged by the programmer.

But in threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Economy. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.

Example.

1. In Solaris for example, creating a process is about thirty times slower than creating a thread, and context switching is about five times slower.

Scalability. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one processor.

- Multithreading exploits multi-processor architectures very well
- Each thread may be running parallel on a different processor.
- A single threaded process can only run on a single CPU, no matter how many are available.
- Multithreading increases concurrency.

How is scheduling handled?

- Scheduling and dispatching

  are itself done at thread level.

- Suspension involves swapping of address space out of memory
- Termination of a process

  terminates all the threads in the process

States in which thread goes while scheduling process

- *Spawn* – Creating a new thread
- *Block* – Waiting for an event
- *Unblock* – Event happened, start new
- *Finish* – This thread is completed

→ Difference between Concurrency and parallelism:

Concurrency:

- Concurrency is when two or more tasks can start, run, and complete in overlapping time periods.
- It doesn't necessarily mean they will ever both be running at the same instant
- Eg. multitasking on a single-core machine

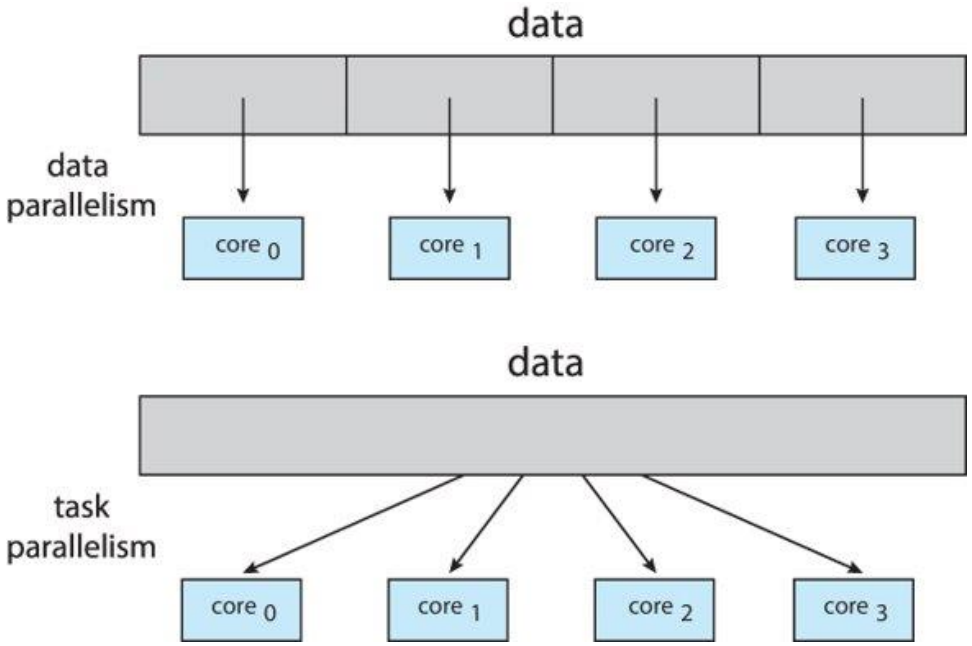
Parallelism:

- Parallelism is when tasks literally run at the same time,
- eg. on a multicore processor.

→ Multicore Programming

- The computer has multiple computing cores on a single chip, where each core appears as a separate processor to the operating system
- Multithreaded programming provides a mechanism for more efficient use of multiple cores and improved concurrency.
- Problems in Multicore Programming
  1. Dividing activities.
  2. Balance of amount of work done on each core
  3. Data splitting
  4. Data dependency (same as synchronization)
  5. Testing and debugging

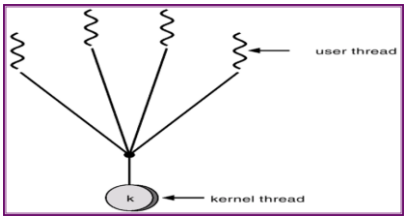
Just know the diagram.



Multithreading Models

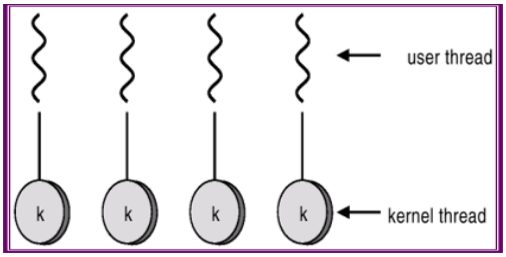
Many-to-One Model:

- Mapping of many user-level threads to one kernel thread
- Problems:
    - Entire process will block if a thread makes a blocking system call
    - Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors
- Eg. Green Threads a thread library available for Solaris



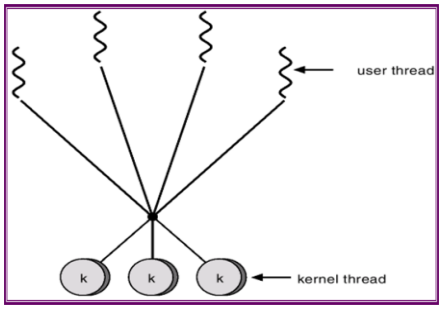
One-to-One Model:

- Mapping of each user thread to a kernel thread
- It provides more concurrency
- allows multiple threads to run in parallel on multiprocessors
- Problems:
    - creating a user thread requires creating the corresponding kernel thread -> Overhead of creating kernel threads.
    - There is upper limit on creation of Kernal threads
- EG. Linux, along with the family of Windows operating systems, implement the one-to-one model.



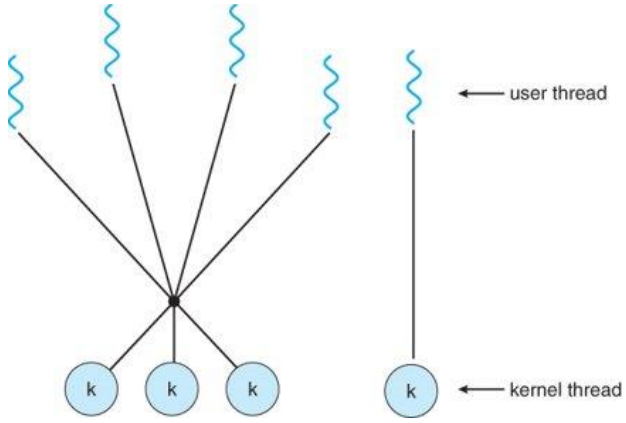
Many-to-Many Model:

- Mapping of multiplexed user-level threads to a smaller or equal number of kernel threads.
- create as many user threads as wishes
- Mapped kernel threads can run in parallel on a multiprocessor.
- When thread performs a blocking system call, the kernel can schedule another thread for execution
- No direct application
    - Used along with one-to-one model



Two level model:

- Uses best of both the models
- Eg.
    - IRIX



Multithreading Issues

- Fork(): creates new child process and Both parent and child processes are executed simultaneously
- Exec(parameter): replaces the current process image with another (different) one given in parameter. Control never returns to the original program unless there is an exec() error.

Problem with fork() and exec() system call

- Problem:

There is confusion when, one thread in a program calls fork(),

- Does the new process duplicate all threads?
- OR new process single-thread from where fork is called?

- Solution:

Use two versions of fork () system call

- If exec () is called immediately after forking => duplicating only the calling thread
- process does not call exec () after forking => duplicate all threads.

Thread Cancellation

Thread Cancellation terminating a thread before it has completed.

- Problem:
  - Two ways of thread cancellation
  - Asynchronous cancellation.
    - One thread immediately terminates the target thread.
    - canceling a thread asynchronously may not free a necessary system-wide resource.
  - Deferred cancellation.
    - The target thread periodically checks whether it should terminate
    - The thread has to check for a flag
    - The thread can perform this check at a at which it can be canceled safely. (lot of overhead)
- Solution
  - No solution discussed

Signal Handling

- Problem:
  - There is confusion in delivering signals, it is more complicated in multithreaded programs.
  - Where exactly must the signal be delivered?
    - 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.
- Solution
    - No solution discussed

Thread pool

- Problems:
    - The first issue concerns the amount of **time required to create the thread prior to servicing the request**, together with the fact that this thread will be discarded once it has completed its work.
    - The second issue: if we allow all concurrent requests to be serviced in a new thread, we have **not placed a bound on the number of threads concurrently active in the system**. Unlimited threads could exhaust system resources, such as CPU time or memory.
- Solution:
    - Use thread pool
        - create a number of threads at process startup and place them into a pool, where they sit and wait for work.
        - Once the thread completes its service, it returns to the pool and awaits more work.
        - Thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns

Thread Specific data:

in some circumstances, each thread need its own copy of certain data. We will call such data Thread specific data.

- Problem:
    - How to maintain same data's copy to each thread
- Solution:
    - To associate each thread with its unique identifier, we could use thread-specific data.