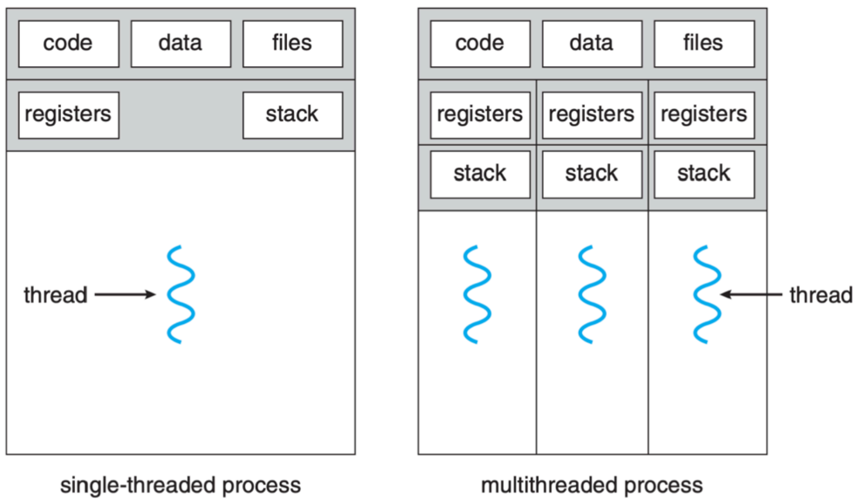
Thread Definition

- A **thread** is the basic unit of CPU utilization, consisting of:
    - Thread ID
    - Program counter
    - Register set
    - Stack
- Threads share with other threads in the same process:
    - Code section
    - Data section
    - OS resources (e.g., open files, signals)



- A **single-threaded process** has one thread of control, while a **multithreaded process** can perform multiple tasks simultaneously. Wel
- Most software applications that run on modern computers are **multithreaded**., 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.

Benefits of Multithreading

1. **Responsiveness**:
    - Allows a program to remain responsive even if part of it is blocked or performing lengthy operations (e.g., user interfaces).
2. **Resource Sharing**:

    - Threads share memory and resources of their process by default, unlike processes which require explicit mechanisms (e.g., shared memory or message passing).
3. **Economy**:
    - Thread creation and context-switching are more efficient than process creation.
4. **Scalability**:
    - Multithreading improves performance on multicore systems by allowing parallel execution across cores.

Multicore Programming

- **Multicore systems** place multiple computing cores on a single chip, appearing as separate processors to the OS.
- **Concurrency vs. Parallelism**:
    - **Concurrency**: Supports multiple tasks making progress (can occur on a single core via interleaving).
    - **Parallelism**: Performs multiple tasks simultaneously (requires multiple cores).
    - It is possible to have concurrency without parallelism by CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes in the system.

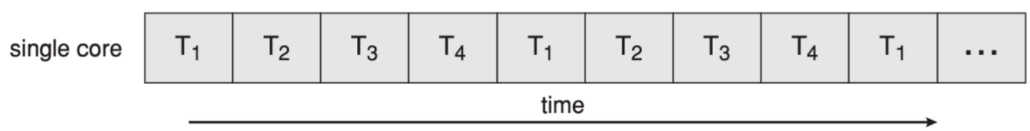


Figure 4.3 Concurrent execution on a single-core system.

-

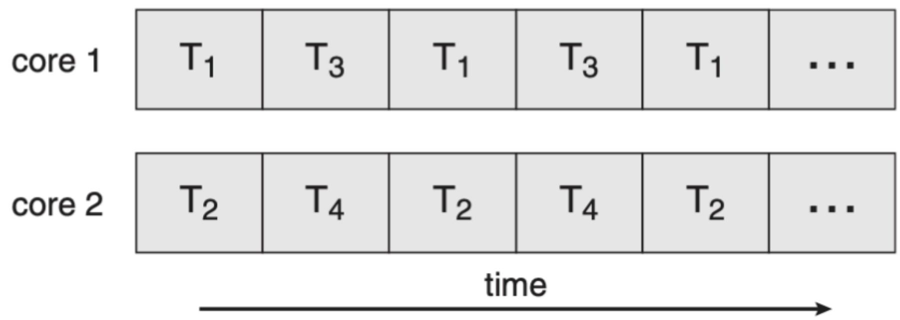


Figure 4.4 Parallel execution on a multicore system.

-

Programming Challenges

1. **Identifying Tasks**:
   - Find independent tasks that can run in parallel.
2. **Balance**:
   - Ensure tasks perform equal work of equal value.
3. **Data Splitting**:
   - Divide data for parallel execution.
4. **Data Dependency**:
   - Synchronize tasks to handle dependencies.
5. **Testing and Debugging**:
   - Concurrent programs are harder to test due to many possible execution paths.

Types of Parallelism

1. **Data Parallelism**:
   - Distributes subsets of the same data across cores and performs the same operation (e.g., summing an array).
   - Ex: On a single-core system, one thread would simply sum the elements [0] . . . [N − 1]. On a dual-core system, however, thread A , running on core0, could sum the elements [0] . . . [N/2 − 1] while thread B, running on

core1,could sum the elements [N/2] . . . [N − 1]. The two threads would be running in parallel on separate computing cores.

2. **Task Parallelism**:
   - Distributes tasks (threads) across cores, each performing unique operations (e.g., different statistical operations on the same data , . Different threads may be operating on the same data, or they may be operating on different data).
   - Ex: an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements.

- Most applications use a hybrid of both.

Multithreading Models

1. **Many-to-One Model**:
   - Maps many user-level threads to one kernel thread.
   - **Pros**: Efficient thread management in user space. allows the developer to create as many user threads as she wishes
   - **Cons**: Blocks entire process if a thread makes a blocking system call; no true parallelism. does not result in true concurrency, because the kernel can schedule only one thread at a time

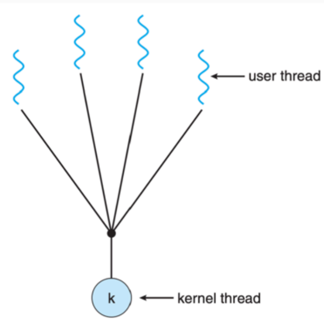


Figure 4.5 Many-to-one model.

2. **One-to-One Model**:
   - Maps each user thread to a kernel thread.
   - **Pros**: Allows concurrency and parallelism; handles blocking calls.
   - **Cons**: limited in the number of threads .Overhead due to kernel thread creation (e.g., Linux, Windows).

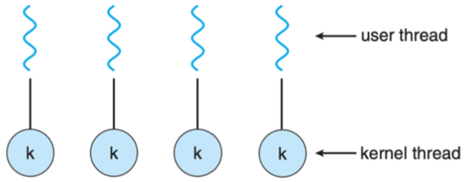


Figure 4.6 One-to-one model.

e-to-One Model

3. **Many-to-Many Model**:
    - Maps many user threads to a smaller/equal number of kernel threads.
    - **Pros**: Balances flexibility and efficiency; allows parallelism and handles blocking calls. developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.
    - **Cons**: More complex to implement.

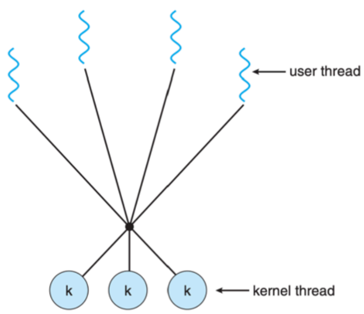


Figure 4.7 Many-to-many model.

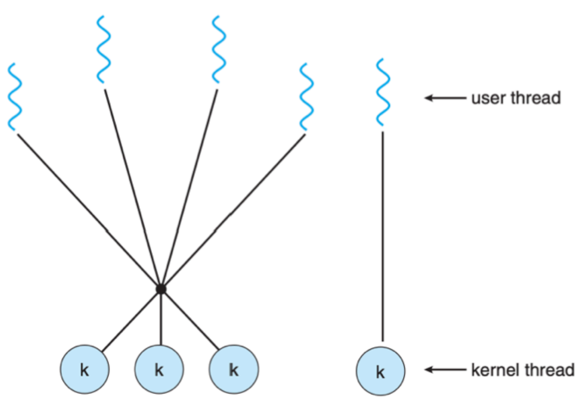


Figure 4.8 Two-level model.

Implicit Threading

- Transfers thread creation/management from developers to compilers/runtime libraries.

Ex problem: multithreaded web server

1. the server receives a request, it creates a separate thread to service the request.
2. problems. The first issue concerns the amount of time required to create the thread, thread will be discarded once it has completed its work.
3. 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 is Thread POOL

- **Thread Pools**:
  - o Pre-created threads wait for tasks. create a number of threads at process startup and place them into a pool, where they sit and wait for work.
  - o **Benefits**:
    1. Faster task servicing.
    2. Limits number of active threads.
    3. Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. Flexible task scheduling.
- **The number of threads in the pool** : set heuristically based on the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests.

Threading Issues

1. **fork() and exec() System Calls**:
   - o **fork()**: Behavior varies—can duplicate all threads or just the calling thread.
   - o **exec()**: Replaces the entire process (all threads) with a new program.
2. **Signal Handling**:

A signal is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously,

- o **Synchronous signals**: Delivered to the thread causing the event.
- o **Asynchronous signals**: Delivered to all threads or a specific thread.
- o 1. A signal is generated by the occurrence of a particular event.
- o 2. The signal is delivered to a process.
- o 3. Once delivered, the signal must be handled.

A signal may be handled by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

3. **Thread Cancellation**: terminating a thread before it has completed. example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled
   - **Asynchronous cancellation**: Immediate termination. One thread immediately terminates the target thread.
   - **Deferred cancellation**: Target thread checks periodically to terminate orderly. allowing it an opportunity to terminate itself in an orderly fashion.

Examples

- **Web Browser**: One thread displays content while another retrieves data.
- **Word Processor**: Threads for graphics, keystrokes, and spell-checking.
- **Server Design**: Multithreaded servers handle requests efficiently without process creation overhead.