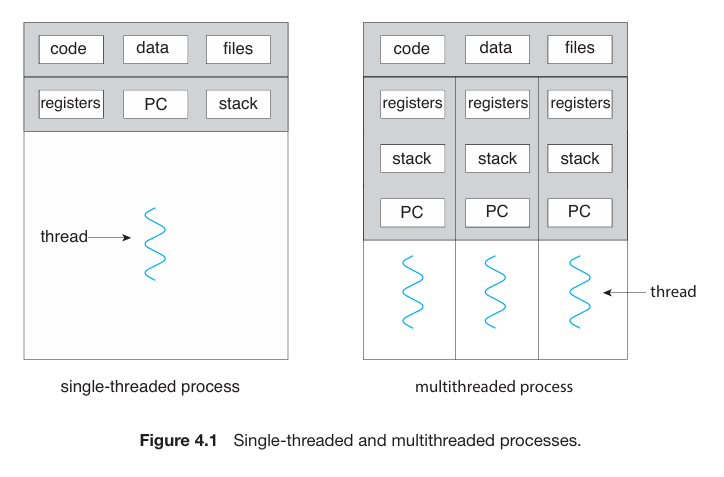
**Threads and Concurrency**

**Thread** – It is the basic unit of CPU utilization. Thread comprises of a thread ID, a program counter, a register set and a stack.

Thread shares its code and data sections as well as other OS resources like open files and signals with other threads belonging to the same process.

A traditional process has single thread of control. If it has multiple threads of control, it can perform multiple tasks at a time.

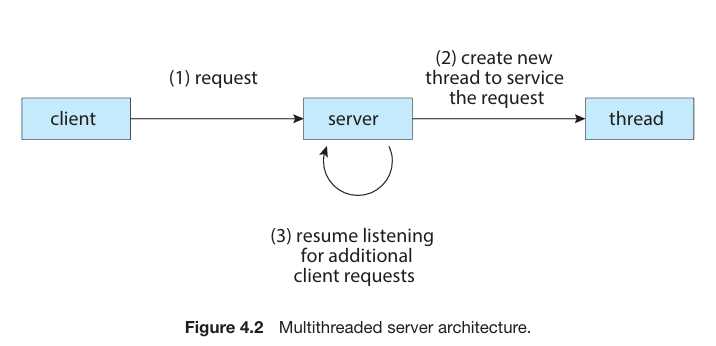
|  |  |
| --- | --- |
| **Multithreaded Process** | **Single-threaded Process** |
| A process having multiple threads of execution is called multithreaded process.  Each process can execute independently. This allows parallelism and concurrent execution of tasks. | A process having single thread of execution is called single-threaded process.  The process executes one instruction after another sequentially. |



In some situations a single application may be required to perform several similar tasks like a web server which takes client requests for data (web pages, images, sound etc.). Now there may be several clients accessing that server concurrently. If the web server was a single-threaded process, it would be able to service one client at a time and the other clients would have to wait for a long period of time for its request to be serviced.

One solution for the above problem is to have the web server as a single process which upon client request creates a separate process to service that request. But creating a new process is time consuming and resource intensive (overhead). Hence it is more efficient to use a process with multiple threads. Now the server will create a separate thread for a request instead of creating another process.

Most operating system kernels are also typically multithreaded. As an example, during system boot time on Linux systems, several kernel threads are created. Each thread performs a specific task, such as managing devices, memory management, or interrupts handling.



**Benefits**

* **Responsiveness** – Multithreading an interactive program may allow a program to keep on running even if a part of it is blocked or is performing a lengthy task increasing responsiveness to the user.
* **Resource sharing** – Threads share resources of the process they belong to. This allows application to have different threads of execution within the same address space.
* **Economy** – As threads share the resources of the process they belong to so it is more economical to have threads rather than separate processes. Threads consume less time and memory than process creation and context switching is faster in between threads as compared to processes.
* **Scalability** – A multithreaded process can benefit more in a multiprocessor architecture where threads may be running in parallel on different processing core.

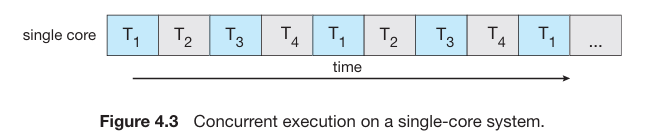
**Multicore Programming**

**Multicore** – Placing multiple cores on a single processing chip where each core appears as a separate CPU to the OS.

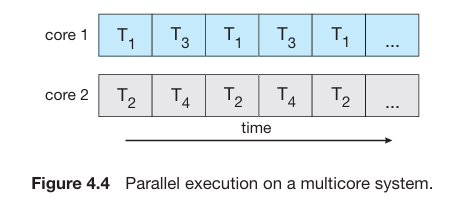
Multithreaded programming provides a mechanism for more efficient use of multiple programming cores and improved concurrency.

On a single core system, concurrency means threads will be interleaved over time whereas in multicore system, it means that some threads can run in parallel as a thread can be assigned to a separate core.

A concurrent system supports more than one task by allowing all the tasks to make progress. In contrast, a parallel system can perform more than one task simultaneously. Thus it is possible to have concurrency without parallelism.



Before multicore or multiprocessing systems, most computers had only one processor and the CPU schedulers gave the illusion of parallelism by rapidly switching processes allowing each process to make progress. Such processes were running concurrently not in parallel.



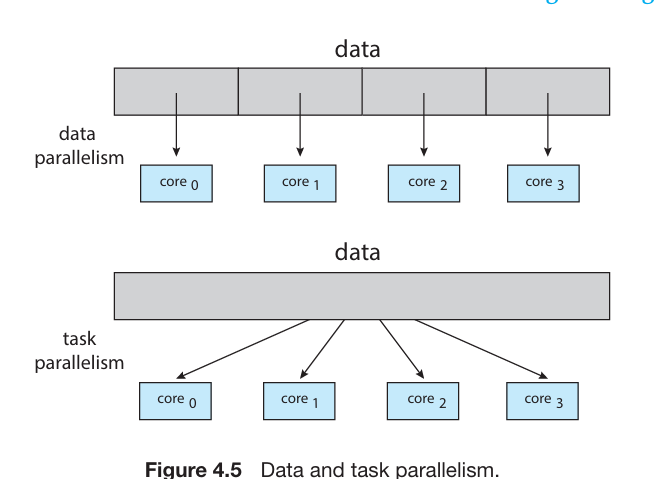
**Programming Challenges**

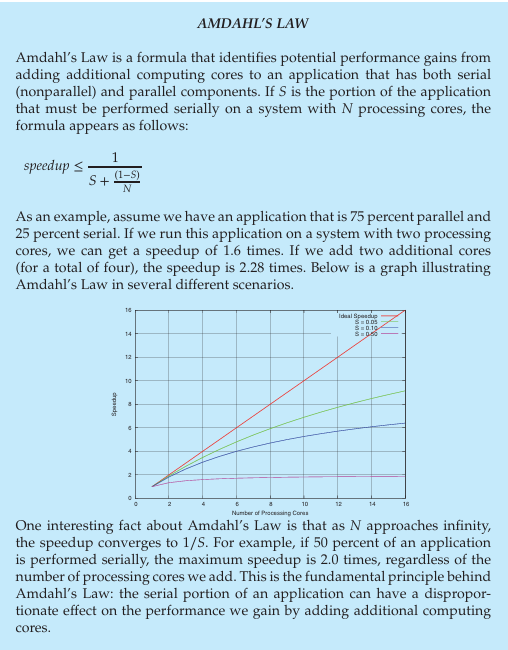
In general, these five areas present challenges in programming for multicore systems:

* **Identifying tasks** - This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.
* **Balance** - While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.
* **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.
* **Data dependency** - The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.
* **Testing and debugging** - When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

**Types of Parallelism**

|  |  |
| --- | --- |
| **Data** | **Task** |
| It focuses on distributing subset of same data across multiple programming cores and performing the same operation on each core.  Fundamentally, data parallelism involves distribution of data across multiple cores. | It focuses on distributing tasks (threads) across multiple programming cores. Each thread is performing unique operation. Different threads maybe operating on the same or different data.  Fundamentally, task parallelism involves distribution of tasks across multiple cores. |



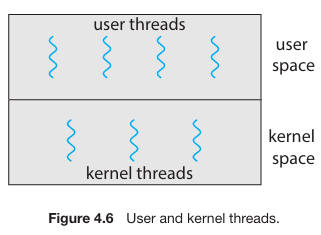


**Multithreading Models**

Support for threads is provided at *user* and *kernel* level.

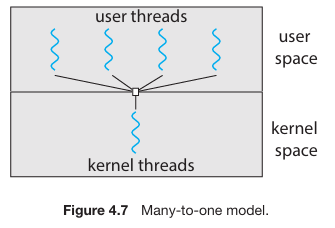
**User level** – Threads which are supported above the kernel and managed without the help of OS.

**Kernel level** – Threads supported and managed directly by the OS.



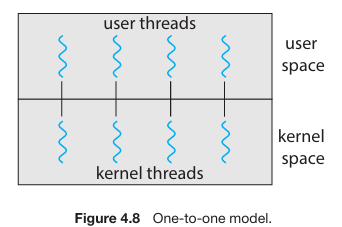
**Many-to-One Model**

In this model, many user-level threads are mapped to a single kernel thread. Thread management occurs in user space, making it efficient. However, if a thread makes a blocking system call, the entire process blocks. Also, only one thread can access the kernel at a time, limiting parallelism, especially on multicore systems. While this model was used in systems like Solaris and early Java, it's less common today due to its inefficiency in utilizing multiple processing cores.



**One-to-One Model**

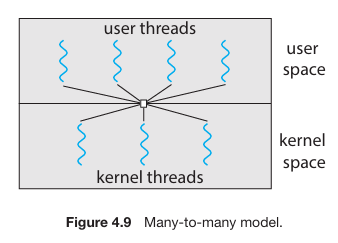
This model maps each user thread to a corresponding kernel thread, providing more concurrency compared to the many-to-one model. When a thread makes a blocking system call, another thread can run. Multiple threads can execute in parallel on multiprocessors. However, creating many user threads may burden the system with a large number of kernel threads. Operating systems like Linux and Windows implement this model.

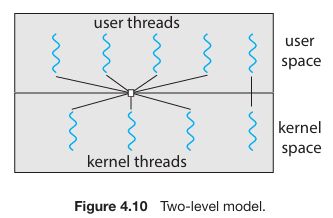


**Many-to-Many Model**

This model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may vary depending on the application or machine configuration. It offers flexibility in creating user threads while allowing parallel execution on multiprocessors.

There's a variation of this model, the two-level model, which also allows user-level threads to be bound to kernel threads. However, implementing the many-to-many model is challenging, and with the increasing number of processing cores in systems, limiting kernel threads has become less important. Despite its flexibility, most operating systems now use the one-to-one model due to practical considerations.





**Thread libraries**

The concept of thread libraries provides programmers with *APIs* to manage threads in their programs. There are two primary ways to implement thread libraries: *user-space* and *kernel-level*. In user-space libraries, all code and data structures exist in user space and a invoking a function in the library will result in a local function call, while in kernel-level libraries, they exist in kernel space and invoking a function in the API for the library will result in a system call.

**User-space** - Memory space where user programs run, separate from the operating system's kernel space.

**Kernel-level** - Memory space where the core of the operating system resides, including device drivers and core functionality.

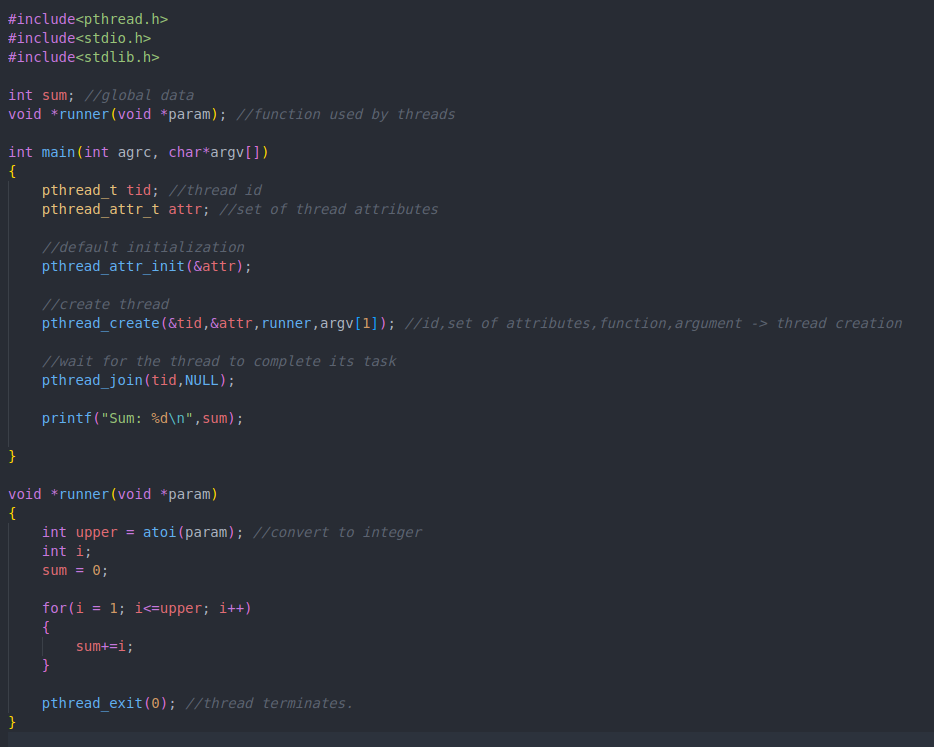
Three main thread libraries are widely used: POSIX Pthreads, Windows threads, and Java threads. POSIX Pthreads can be implemented as either user-level or kernel-level libraries. Windows threads are kernel-level libraries specific to Windows systems, while Java threads are implemented using the host system's thread library (such as Pthreads on UNIX-like systems or Windows API on Windows).

*Global data* declared outside of any function is shared among all threads belonging to the same process in POSIX and Windows threading. In Java, shared data must be explicitly arranged between threads due to the lack of global data concept.

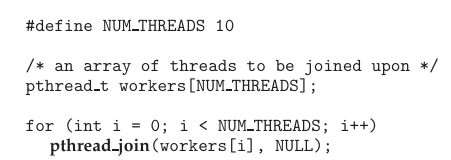
**Global Data** - Data declared outside of any function that is accessible to all threads belonging to the same process.

|  |  |
| --- | --- |
| **Asynchronous** | **Synchronous** |
| Strategy where the parent thread creates child threads that execute concurrently and independently. Parent doesn’t wait for its children to execute. As threads are independent there is little data sharing between them. | Strategy where the parent thread creates child threads and waits for them to terminate before continuing execution. It involves significant data sharing among threads. |

**Pthreads**

****

The provided example demonstrates the basic usage of the Pthreads API to create a multithreaded program for calculating the summation of a non-negative integer. Threads in a Pthreads program begin execution in a specified function. The example illustrates the creation of a second thread using pthread\_create () and shares global data between threads. At this point, the program has two threads: the initial (or parent) thread in main () and the summation thread (child thread) performing the summation operation in the runner () function. The program follows a thread create/join strategy, where the parent thread waits for the child thread to terminate using pthread\_join (). Upon termination of the child thread, the parent thread outputs the calculated sum.



Above code illustrates joining 10 threads using Pthreads library;

**Threading Issues**

**The fork () and exec () system calls**

Some UNIX systems offer two versions of the fork () system call:

1. One version duplicates only the thread that invoked the fork () system call.
2. Another version duplicates all threads.

The choice between these two versions depends on the application's requirements:

* If the program immediately calls exec () after forking, then duplicating all threads is unnecessary because the new program specified in the exec () call will replace the entire process, including all threads. In this case, duplicating only the calling thread is appropriate to avoid unnecessary overhead.
* However, if the separate process does not call exec () after forking, it should duplicate all threads to ensure that the new process maintains the same multi-threaded environment as the parent process.

**Signal Handling**

Signals in UNIX systems are notifications of events that can occur *synchronously* or *asynchronously*. Synchronous signals are generated by actions within a process, like illegal memory access, while asynchronous signals originate externally, like from user input or timer expiration.

**Signal** - A notification in UNIX systems indicating the occurrence of a particular event.

**Synchronous** **Signal** - A signal generated by an event occurring within the process itself.

**Asynchronous** **Signal** - A signal generated by an external event not directly tied to the running process.

Signals are delivered to processes and can be handled by default or user-defined handlers.

**Default Signal Handler** - The kernel-defined action taken when handling a signal if a user-defined handler is not specified.

**User-defined Signal Handler** - A custom function defined by the user to handle a specific signal.

In single-threaded programs, signal handling is straightforward, but in multithreaded programs, it's more complex as signals can be delivered to specific threads or all threads in a process. The method of delivering signals depends on the type of signal. Standard UNIX function for delivering signals is; kill (pid\_t PID, int signal)

In multithreaded UNIX environments, threads can specify which signals they will accept and which they will block. This means that an asynchronous signal might only be delivered to threads that are not blocking it. However, since signals need to be handled only once, typically the signal is delivered to the first thread encountered that is not blocking it. POSIX Pthreads provides the function pthread\_kill (pthread\_t tid, int signal) to deliver a signal to a specified thread by its ID (tid). Windows doesn't have explicit signal support but uses asynchronous procedure calls (APCs) for similar functionality, delivered to specific threads.

**Thread Cancellation**

Thread cancellation involves terminating a thread before it completes its task. This is useful in scenarios like database searches or web page loading, where canceling unnecessary threads improves efficiency.

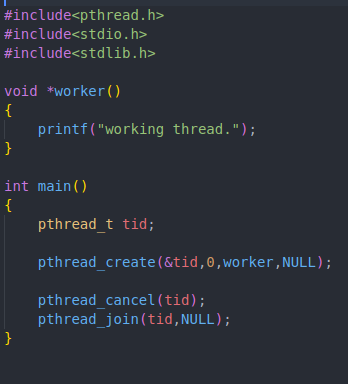
**Thread Cancellation** - The termination of a thread before it completes its execution.

There are two main approaches to thread cancellation: *asynchronous* and *deferred*. **Asynchronous** cancellation immediately terminates the target thread, without giving it a chance to clean up or complete its task. **Deferred** cancellation allows the target thread to periodically check for cancellation requests, enabling it to terminate itself safely and in an orderly fashion.

**Target thread** – It refers to the thread that is intended to be canceled.

However, cancellation can pose challenges, particularly with resource management and data consistency, especially with asynchronous cancellation.

In Pthreads (POSIX threads), thread cancellation is initiated using the pthread\_cancel () function, which takes the identifier of the target thread as a parameter. The following code illustrates the described method:

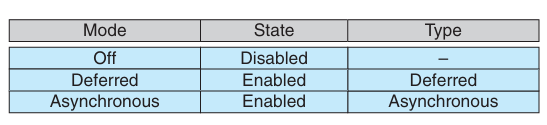


**Explanation**

* pthread\_create (&tid, 0, worker, NULL). This line creates a new thread with the identifier stored in tid, executing the worker function. The NULL parameter represents optional arguments passed to the thread (in this case, none are passed).
* pthread\_cancel (tid). This line sends a cancellation request to the target thread identified by tid. However, the actual cancellation depends on how the target thread is set up to handle the request.
* pthread\_join (tid, NULL). This line waits for the target thread identified by tid to terminate. It ensures that the main thread waits until the worker thread has finished executing before the program exits.

Invoking pthread\_cancel () indicates only a request to cancel the target thread, however; actual cancellation depends on how the target thread is set up to handle the request. When the target thread is finally canceled, the call to pthread\_join () in the canceling thread is returned.

Regarding cancellation modes, Pthreads supports three cancellation modes, each defined by a state and a type.



Threads can set their cancellation state and type using appropriate APIs. While cancellation can be disabled for a thread, cancellation requests remain pending, allowing the thread to later enable cancellation and respond to the requests.

Deferred cancellation is the default cancellation type in Pthreads. In deferred cancellation, a thread periodically checks for cancellation requests and can respond to them at specific points in its execution, known as cancellation points.

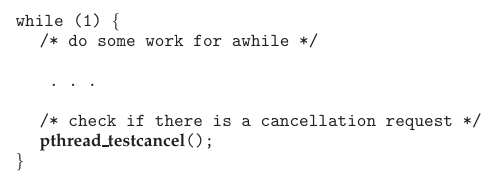
**Cancellation points** - predefined points in the execution flow of a thread where it checks for pending cancellation requests.

Most blocking system calls in the POSIX and standard C library are defined as cancellation points. These include functions like read (), write (), pthread\_join (), and many others. When a thread is blocked in one of these functions, it is actively checking for cancellation requests, allowing it to be canceled if necessary.

In Pthreads, the pthread\_testcancel () function is used to establish a cancellation point within a thread's execution flow. It checks if a cancellation request is pending for the current thread. If a cancellation request is pending, the function will not return, and the thread will terminate. Otherwise, if no cancellation request is pending, the function will return, and the thread will continue its execution.

**Cleanup handler** - A cleanup handler is a function that can be registered to be invoked automatically when a thread is canceled. This function allows a thread to release any resources it may have acquired before it is terminated due to a cancellation request.

The following code illustrates how a thread would respond to cancellation request using deferred cancellation:



**Thread Local Storage (TLS)**

Thread-local storage (TLS) refers to a mechanism in multithreaded programming where each thread maintains its own copy of certain data, allowing threads to access and modify their own instance of the data without interfering with other threads' copies.

TLS is useful when threads need their own instance of data that should not be shared with other threads. For example, in a transaction-processing system where each transaction is handled by a separate thread, each thread might need its own unique identifier associated with the transaction.

**Local variables and TLS**

Local variables are visible only within the scope of a single function invocation, while TLS data are visible across multiple function invocations within the same thread.

TLS data persist across function calls within the same thread, unlike local variables which are created and destroyed each time a function is invoked.

Pthreads (POSIX threads) includes the pthread\_key\_t type, which provides a key specific to each thread for accessing TLS data.

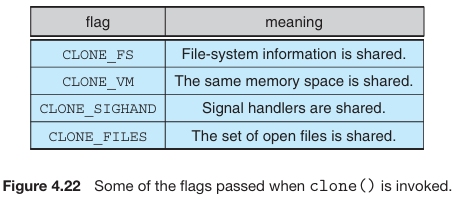
An example of declaring a thread-local variable in C using the gcc compiler:



**Linux Threads**

In Linux, the clone() system call is used for creating tasks, which can serve as either processes or threads, depending on the flags passed to it. Unlike traditional systems that distinguish between processes and threads, Linux uses the term "task" to refer to both. The level of sharing between the parent and child tasks is determined by the flags provided to clone ().

Flags such as CLONE\_FS, CLONE\_VM, CLONE\_SIGHAND, and CLONE\_FILES determine the extent of sharing between parent and child tasks. For example, sharing file-system information, memory space, signal handlers, and open files can be controlled using these flags.



When clone () is invoked with specific flags set, the parent and child tasks share various resources, resembling the behavior of creating a thread. If no sharing flags are set, invoking clone () behaves similarly to the fork () system call, where a new task is created without sharing resources with the parent task.

Linux represents tasks using a unique kernel data structure (struct task\_struct), which contains pointers to other data structures holding task-specific information, such as open files, signal handlers, and memory management.

When clone() is called, a new task is created, and depending on the flags passed, it either points to the data structures of the parent task or creates separate copies.

The flexibility of clone () extends to containerization concepts. Flags passed to clone () can control the behavior of creating tasks within a container, allowing for the creation of isolated Linux systems (containers) under a single Linux kernel.