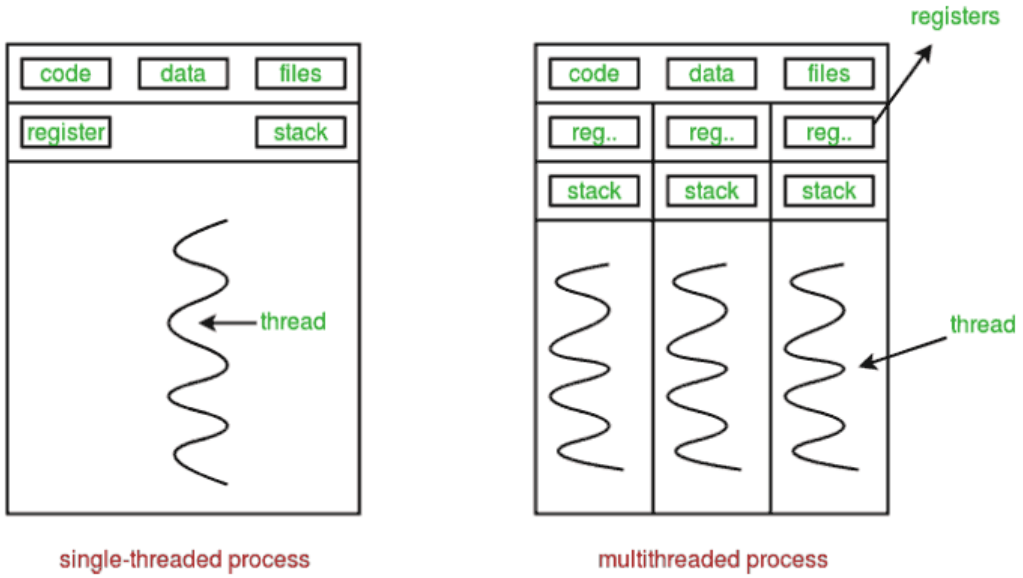
Multithreading in C

Last Updated : 23 Jul, 2025

A thread is a single sequence stream within a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. But unlike processes, threads are not independent from each other unlike processes. They share with other threads their code section, data section and OS resources like open files and signals. But, like processes, a thread has its own program counter (PC), a register set, and a stack space.



Multithreading is a programming technique where a process is divided into multiple smaller units called threads, which can run simultaneously. Multithreading is commonly used in applications like web servers, games, and real-time systems to handle simultaneous tasks like user input, background processing, and other I/O operations simultaneously.

Multithreading in C

In C programming language, we use the POSIX Threads (**pthreads**) library to implement multithreading, which provides different components along with thread management functions that create the foundation of a multithreaded program in C.

The pthread library is defined inside **<pthread.h>** header file. Generally, we don't need to explicity specify to the linker that we are using this library but if the program shows error, then compile it with the following flags:

```
gcc sourceFile.c -lpthread
```

Let's see how to perform multithreading in our C program.

Creating a Thread

The first step is to create a thread and give it a task. To create a new thread, we use the **pthread_create()** function provided by the thread library in C. It initializes and starts the thread to run the given function (which specifies its task).

Syntax

```
pthread_create(thread, attr, routine, arg);
```

where,

- **thread** : Pointer to a **pthread_t** variable where the system stores the ID of the new thread.
- **attr** : Pointer to a thread attributes object that defines thread properties. Use NULL for default attributes.
- **routine**: Pointer to the function that the thread will execute. It must return void* and accept a void* argument.
- **arg**: A single argument passed to the thread function. Use NULL if no argument is needed. You can pass a struct or pointer to pass multiple values.

Example

```c
#include <pthread.h>
#include <stdio.h>

void* foo(void* arg) {
    printf("Created a new thread");
    return NULL;
 }
int main() {

    // Create a pthread_t variable to store
    // thread ID
    pthread_t thread1;

    // Creating a new thread.
    pthread_create(&thread1, NULL, foo, NULL);
    return 0;
}
```

Output

```
Created a new thread
```

In the above program, there is a possibility that the **main** thread may end before the execution of the created thread **thread1** and it may lead to unexpected behaviour of the program. So, there is a functionality in C to wait for the execution of the particular thread.

Wait for Thread to Finish

pthread_join() function allows one thread to wait for the termination of another thread. It is used to synchronize the execution of threads.

Example

```c
#include <pthread.h>
#include <stdio.h>

void* foo(void* arg) {
    printf("Thread is running.\n");
    return NULL;
}

int main() {
    pthread_t thread1;
```

```
    pthread_create(&thread1, NULL, foo, NULL);

    // Wait for thread to finish
    pthread_join(thread1, NULL);

    return 0;
}
```

Output

```
Thread is running
```

Explicitly Terminate Thread

pthread_exit() function allows a thread to terminate its execution explicitly. **pthread_exit()** is called when a thread needs to terminate its execution and optionally return a value to threads that are waiting for it.

Example

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

void* foo(void* arg) {
    printf("Thread is running.\n");

    // Explicity terminate thread
    pthread_exit(NULL);

    printf("This will not be executed.\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, foo, NULL);

    // Wait for created thread to finish
    pthread_join(thread, NULL);

    return 0;
}
```

Output

```
Thread is running.
```

Requests Cancellation of Thread

The **pthread_cancel()**function is used to request the cancellation of a thread. It sends a cancellation request to the target thread, but the actual termination depends on whether the thread is in a cancellable state and if it handles cancellation.

Example

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void* myThreadFunc(void* arg) {
    while(1) {
        printf("Thread is running...\n");
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, myThreadFunc, NULL);
    sleep(5);
    //Requesting to cancel the thread after 5 seconds.
    pthread_cancel(thread);
    // Wait for the thread to terminate
    pthread_join(thread, NULL);

    printf("Main thread finished.\n");
    return 0;
}
```

Output

```
Thread is running...
Thread is running...
Thread is running...
Thread is running...
Thread is running...
Thread is running...
Main thread finished.
```

Getting ID of a Thread

pthread_self()function returns the thread ID of the calling thread. This is useful when you need to identify or store the ID of the current thread in multi-threaded programs.

Example

```c
#include <pthread.h>
#include <stdio.h>

void* foo(void* arg) {

    // Get current thread ID
    pthread_t thisThread = pthread_self();
    printf("Current thread ID: %lu\n",
        (unsigned long)thisThread);
    return NULL;
}

int main() {
```

```
    pthread_t thread1;
    pthread_create(&thread1, NULL, foo, NULL);
    pthread_join(thread1, NULL);
    return 0;
}
```

Output

```
Current thread ID: 134681321096896
```

Common Issues in Multithreading

Multithreading can greatly improve the performance of the program by running multiple tasks simultaneously. But it also comes with several threading issues and challenges that need to be handled properly to ensure efficient running of a multithreaded program Some of the common issues in multithreading in c are as follows:

Race Conditions

The race condition issue occurs when multiple threads try to access a shared resource at the same time and the output depends on the order of the execution of those threads. This issue can lead to unpredictable behavior of the program and cause the program to generate different results each time when it is executed.

Example: If two threads try to access a shared counter variable where both are trying to change the value based on the current value, then the program may not generate the expected result because both threads may read and write the counter at the same time.

Deadlocks

A deadlock condition arises when multiple threads are blocked forever as they are waiting for each other to release the occupied resource. It generally arises in situations where thread acquire some resources initially and request for more resources midway during their execution. Which leads to a cycle of dependencies.

Example: Thread A has occupied Resource 1 and 2 arewaiting for resource 3 while thread B has occupied resource 3 and is waiting for resource 2 for its completion.

Starvation

The starvation condition arises when a thread is denied access for the requested resource for indefinite amount of time. This situation commonly occurs with priority-based scheduling algorithms when they are biased toward certain threads.

Example: A thread with low priority will have to wait indefinitely as higher priority threads are continuously available.

Thread Synchronization

Thread synchronization is a process that is used to ensure that multiple threads can work with shared resources without causing any issues like race conditions, deadlocks, or data corruption. Thread synchronisation techniques control how threads interact or modify shared data or resources and also ensures that certain critical sections of code are executed in a controlled manner.

Mutex-Based Synchronization

A mutex is the most basic synchronization mechanism. It ensures that only one thread can access a shared resource at a time, preventing race conditions.

Semaphores

A semaphore is a signaling mechanism that controls access to a shared resource using a counter. It can be either a binary semaphore (with values 0 or 1) or a counting semaphore (with values greater than 1), allowing multiple threads to access a resource concurrently up to a specified limit.

Conditional Variables

Condition variables allow threads to wait for certain conditions to be met before proceeding. A condition variable is always used along with a mutex lock. A thread will wait on the condition variable and release the mutex until it is notified by another thread.

Barrier Synchronization

Threads can be synchronized at specific points in execution using a barrier. Barriers are synchronization mechanisms that block threads until all threads in a group reach a specific point in their execution. Once all threads reach the barrier, they are allowed to proceed.

Read-write Locks

Read-Write locks allow multiple threads to read from a shared resource or data at the same time but ensure that only one thread can write to the resource at a time. This improves performance when the number of readers is more than writers.

Need for Multithreading

Multithreading is used to improve a program's efficiency by allowing it to perform multiple tasks in parallel. It enhances CPU utilization, reduces idle time, and makes applications faster and more responsive, especially in tasks like file handling, user interaction, and background processing.

For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.

Advantages of Multithreading

- **Improved Performance:** Multithreading enables programs to perform multiple tasks simultaneously by running multiple threads concurrently which can improve the performance and speed up the execution of programs.
- **Better CPU Utilisation:** Multithreading also improves CPU utilisation as multiple threads run concurrently on different cores of CPU ensuring that all cores are put to work which help in reducing CPU idling thus utilising the complete potential of the CPU.
- **Responsiveness and Improved User Experience**: Multithreading also enhances the responsiveness of the program as different threads can handle different tasks making the program more responsive and faster.
- **Efficient Resource Sharing**: Compared to separate processes threads can easily share data as they share the same memory space. Which reduces the time required for interposes communication and makes information sharing faster.

- **Scalability**: Applications that use multithreading are easier to scale as workloads can be easily divided among different threads. Also, a greater number of cores can be added to further handle larger workloads.

Limitations of Multithreading

- **Complexity in Programming**: Using Multithreading increases the complexity of the program as developers also have to handle thread synchronisation, thread safety and other concurrency issues.
- **Concurrency Issues:** Multithreading programs also have to deal with different concurrency issue like Race Conditions, Deadlocks and starvation.
- **Increased Overheads in Thread Management:** Multithreading also increases the overheads as it has to manage the creation, synchronisation and context switching between the threads. Which can lead to increases memory usage. Also creating too many threads than the available CPU cores can lead to performance degradations.
- **Increased Risk of Bugs**: Multithreaded programs are more prone to bugs and are hard to debug.