POSIX Threads (pthreads)

POSIX threads, also known as pthreads, are a standard for creating and managing threads in a Unix-like operating system, including Linux. Pthreads provide a way to create multiple threads of execution within a single process, allowing for concurrent programming and improved system utilization.

- It is an execution model
- Indepetently from a language
- Parallel execution model
- It allows a program to control multiple different flows of work that overlap in time.

Key Concepts:

Thread vs Process

- **Process**: Independent execution unit with its own memory space.
- **Thread**: Lightweight process within a process, sharing memory and file descriptors

1. **Thread Creation**: Creating a new thread using pthread_create().

2. **Thread Synchronization**: Coordinating access to shared resources using mutexes, condition variables, and semaphores.

3. **Thread Communication**: Exchanging data between threads using shared variables or message passing.

Main Pthread Functions (or) pthread API:

1. **pthread_create()**: Create a new thread.

2. **pthread_join():** Wait for a thread to finish.

3. **pthread_exit():** Terminate a thread.

4. **pthread_mutex_init()** : Initializes a mutex

5. **pthread_mutex_lock() and pthread_mutex_unlock():** Lock and unlock a mutex.

6. **pthread_mutex_destroy() :** Destroys a mutex.

7. **pthread_cond_wait() and pthread_cond_signal():** Wait on and signal a condition variable.

6. **pthread_self() :** Returns the calling thread's ID

Thread States:

A POSIX thread can exist in several states throughout its lifecycle:

a) Running:

The thread is currently executing on a CPU.

b) Ready (Runnable):

The thread is ready to execute but is waiting for a CPU to become available. It is in the scheduler's queue.

c)Blocked (or Waiting):

The thread is waiting for a specific event to occur, such as:

- A mutex lock to be released.
- An I/O operation to complete.
- A condition variable to be signalled.
- Another thread to complete its execution (e.g., using pthread_join ()).
- A timed wait (e.g., using pthread_cond_timedwait ()).

d)Terminated (Exited):

This is the final state of a thread, indicating that it has completed its execution. A thread can terminate voluntarily by returning from its entry point function or by calling pthread_exit(). It can also be terminated by another thread using pthread_cancel(). Once terminated, a thread cannot be restarted.

Thread joining

- Function: Thread joining (using `pthread_join()`) allows one thread to wait for another thread to complete its execution.

- Impact on thread state:
- When a thread calls `pthread_join()` on another thread, the calling thread will block and enter a "waiting" state until the target thread terminates.
- When the target thread exits (e.g., by returning from its start routine or calling `pthread_exit()`), the joining thread is unblocked and retrieves the target thread's exit status.
- Example: Ensuring that all worker threads have finished processing their tasks before the main thread continues with a final aggregation step.

  In essence, POSIX synchronization mechanisms introduce various states for threads, including running, ready, blocked, and waiting, depending on whether they can acquire a lock, are waiting for a condition to be met, or are waiting for another thread to finish. Understanding these mechanisms and their impact on thread states is crucial for designing and implementing correct and efficient multithreaded applications in the POSIX environment.

Advantage:

1. Improved responsiveness: Threads can handle tasks concurrently, improving system responsiveness.

2. Increased throughput: Threads can utilize multiple CPU cores, increasing overall system throughput.

3. Efficient resource use: Threads share the same memory space, reducing memory overhead.

Disadvantage:

1. Synchronization: Coordinating access to shared resources can be complex.

2. Communication: Exchanging data between threads can be error-prone.

3. Deadlocks: Threads can deadlock when waiting for each other to release resources.

Synchronization in Pthreads

- Use **mutexes** to avoid **race conditions** when multiple threads access shared data.
- Use **condition variables** to block a thread until a condition is true.

- **Feature** **Pthreads**

  Library - pthread.h

  Key Benefit - Efficient concurrency in shared memory

  Used For - Parallelism, synchronization, multi-threading in apps

Synchronization Mechanisms and Their Impact on Thread State

Synchronization mechanisms in an operating system are crucial for managing concurrent access to shared resources by multiple threads, preventing data corruption and ensuring orderly execution. They impact thread state by potentially blocking threads (putting them in a waiting state) while they contend for resources or waiting for a specific condition, and by allowing threads to transition from a waiting state to a runnable state once the resource becomes available or the condition is met.

Example:

Imagine multiple threads writing to a shared file. Without synchronization, they might overwrite each other's data, leading to data corruption. Using a lock, only one thread can hold the lock at a time, ensuring exclusive access to the file. Other threads attempting to write to the file will be blocked (transitioning to a waiting state) until the lock is released. Once the first thread finishes writing and releases the lock, another waiting thread can acquire the lock, transition to a running state, and continue writing.

POSIX Threads provide several mechanisms for synchronization, which directly influence thread state transitions:

a) Mutexes (Mutual Exclusion Locks):

Definition: Mutexes (mutual exclusions) are locks that enforce exclusive access to a shared resource or a section of code (a "critical section"). Only one thread can hold the mutex at a time, preventing other threads from accessing the protected resource or code section.

Mechanism: A thread wishing to access the protected resource first attempts to acquire the mutex lock. If the lock is available, the thread acquires it and enters the critical section. If another thread already holds the lock, the requesting thread blocks (pauses execution) until the lock is released. After finishing its work with the protected resource, the thread releases the mutex lock, allowing another waiting thread to acquire it.

Real-world Example: Imagine a shared counter in a multithreaded application. Without a mutex, multiple threads incrementing the counter simultaneously could lead to an incorrect final value. Using a mutex ensures that only one thread increments the counter at a time, guaranteeing data integrity.

Impact on Thread State: A thread attempting to acquire a locked mutex will transition from a "running" or "ready" state to a "waiting" or "blocked" state until the mutex is released by the owning thread.

b) Condition Variables:

Definition: Condition variables facilitate communication and synchronization among threads based on programmer-defined conditions. They allow threads to wait until a specific condition becomes true before proceeding.

Mechanism: Threads waiting for a condition to be met will wait on a condition variable, relinquishing the CPU. When another thread modifies the shared resource and makes the condition true, it signals the condition variable, waking up the waiting threads.

Real-world Example: Consider a producer-consumer scenario where one thread produces data and another consumes it. The consumer thread might wait on a condition variable until data is available in the shared buffer, and the producer thread might signal the condition variable after adding data, allowing the consumer to proceed.

Impact on Thread State: A thread calling pthread_cond_wait() will atomically release the associated mutex and enter a "waiting" or "blocked" state until another thread signals the condition variable using pthread_cond_signal() or pthread_cond_broadcast(). Upon being signalled, the thread reacquires the mutex and becomes "ready" to run.

c) Read-Write Locks:

Definition: Read-write locks are specialized locks that differentiate between read and write operations on a shared resource. Multiple threads can simultaneously acquire a read lock, allowing concurrent read access. However, only one thread can acquire a write lock at a time, providing exclusive write access and preventing any concurrent read or write operations.

Mechanism: When a thread wants to read the shared data, it requests a read lock. Multiple threads can hold read locks concurrently. When a thread wants to modify the data, it requests a write lock. This blocks any other threads from acquiring either a read or write lock until the write lock is released.

Real-world Example: A database or file system where data is frequently read but infrequently updated. Read-write locks would allow multiple threads to read data concurrently, enhancing performance, but would ensure exclusive access for any updates to maintain data consistency

Impact on thread state:

- When a thread acquires a read lock, it can read the shared resource. Multiple threads can hold read locks concurrently.
- If a thread tries to acquire a write lock while other threads hold either read or write locks, it will be blocked until all other locks are released.
- If a thread holding a read lock attempts to acquire a write lock on the same resource, the behavior is undefined according to the standard, highlighting the importance of careful usage.
- Example: A shared data structure accessed by many threads for reading and only occasionally updated by a single thread. Read-write locks allow for greater concurrency than a single mutex in such scenarios

d) Barriers:

Definition: Barrier synchronization forces a group of threads to wait at a specific point until all threads in the group have reached that point. Once all threads have arrived at the barrier, they are all released to continue execution.

Mechanism: A barrier is initialized with a specific count (the number of threads that need to reach it). Each thread, upon reaching the barrier, increments an internal counter and then waits. When the counter reaches the initial count, all waiting threads are released.

Real-world Example: In parallel processing, if multiple threads are working on different parts of a larger task, a barrier could be used to ensure all threads complete their current iteration before proceeding to the next one, guaranteeing data consistency or coordinating subsequent steps.

Impact on Thread State: Threads reaching a barrier will enter a "waiting" or "blocked" state until all participating threads have arrived at the barrier, at which point all waiting threads transition to a "ready" state.

Threads calling pthread_barrier_wait() will block until a specified number of threads have reached the barrier, at which point all blocked threads transition to Ready.

e) Semaphores:

Function: Semaphores are signaling mechanisms that can be used for more general synchronization than mutexes, allowing one or more threads to access a shared resource.

Impact on thread state:

- A semaphore has an internal counter. When a thread performs a "wait" operation (e.g., `sem_wait()`), it decrements the semaphore's count. If the count becomes negative, the thread is blocked and placed in a waiting queue associated with the semaphore.
- When a thread performs a "post" operation (e.g., `sem_post()`), it increments the semaphore's count. If any threads are blocked on the semaphore, one of them is unblocked and transitions to a "ready" state.

Threads performing a sem_wait() operation on a semaphore with a count of zero will transition to Blocked.

A sem_post() operation increments the semaphore count and can unblock a waiting thread, moving it to the Ready state.

f) Critical section:

- Definition: A section of code where shared resources are accessed. Only one process or thread should be allowed in the critical section at a time to prevent data corruption.
- Real-world analogy: Imagine a public restroom. Only one person can be inside at a time to maintain privacy and prevent conflicts. The act of entering, using, and exiting the restroom represents the critical section.

By understanding POSIX threads, developers can write more efficient and concurrent programs that take advantage of multi-core processors and improve system.

THE CRITICAL-SECTION PROBLEM

Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.

Important feature:

when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

- Each process must request permission to enter its critical section.
- The section of code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**.
- The remaining code is the **remainder section**.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Figure . General structure of a typical process P_i.