2. POSIX THREADS

POSIX Threads (pthreads):

- POSIX threads, or Pthreads, are a **standard set of C language programming interfaces for creating and managing threads** on UNIX-like systems.
- Threads allow multiple paths of execution within the same process to **share resources such as memory, file descriptors, and data**.

2.1. Introduction to Threads:

- **Thread:** The smallest unit of execution within a process.
- **Lightweight Process:** Threads are sometimes called lightweight processes because they share the same address space and resources of the parent process.
- **Benefits of Threads:**
  - **Responsiveness:** Programs remain responsive even during blocking operations.
  - **Resource Sharing:** Threads share memory and resources efficiently.
  - **Economy:** Less overhead compared to creating full processes.
  - **Scalability:** Multiple threads can run on multiple processors.

2.2. POSIX Threads:

- POSIX threads are **standardized by IEEE POSIX 1003.1c**.
- Supported on UNIX and Linux systems.
- POSIX Threads API includes:
  - **Thread management**
  - **Synchronization mechanisms**
  - **Thread-specific data**

2.3. Thread Functions:

2.3.1 Thread Creation and Termination:

- **pthread_create()**
  - creates a new thread and makes it runnable, taking a thread ID, attributes, function, and argument.
  - Threads start executing the **function specified in the start routine**.
  -

- **Syntax:**

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

    - thread: pointer to thread identifier
    - attr: thread attributes (or NULL for default)
    - start_routine: function the thread will execute
    - arg: argument to the function
- Returns 0 on success.

- **pthread_exit()**
    - Terminates the calling thread.
    - Allows returning a value to another thread:
    - Proper termination ensures **no memory leaks and orderly thread cleanup.**
    - void pthread_exit(void *retval);
- **pthread_join()**
    - waits for a thread to complete and collects its return value. Threads can be **joinable** (resources reclaimed manually) or **detached** (resources freed automatically).
    - Syntax:
    - int pthread_join(pthread_t thread, void **retval);
        - Blocks calling thread until thread terminates.
        - retval receives the return value of the terminated thread.

2.3.2 Thread Attributes:

- pthread_attr_t
- Thread attributes define **behavior and properties** of a thread. Common attributes include **detach state, stack size, and scheduling policy**.
- **Detach state** determines whether a thread can be joined or auto-freed. **Scheduling policies** like SCHED_OTHER, SCHED_FIFO, or SCHED_RR control CPU access.
- Attributes can be initialized using pthread_attr_init() and set with specific functions.

- Default attributes are applied if NULL is passed during thread creation.

2.3.3 Thread Cancellation:

- A thread can be canceled by another thread:
    - pthread_cancel(thread_id);
    - Thread can set a cancellation point using pthread_testcancel().
- Useful for terminating long-running threads safely.

2.4. Thread Synchronization:

- Synchronization mechanisms help an operating system manage shared resources safely.
- They prevent multiple threads from accessing the same resource at the same time.
- This avoids data corruption and ensures correct program execution.
- When a thread requests a busy resource, it is blocked and moved to a waiting state.
- Once the resource becomes free, the waiting thread becomes runnable again.

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:

- **Mutexes** allow only one thread to enter a critical section at a time.
- **Condition variables** enable threads to wait for specific events or signals.
- **Read-Write locks** allow multiple readers but one writer to access shared data.
- **Semaphores** control access to limited resources using counting mechanisms.
- Synchronization ensures **data consistency, correctness, and thread safety**.

2.4.1 Mutexes (Mutual Exclusion):

- 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.

Functions:

- pthread_mutex_init(&mutex, NULL);
- pthread_mutex_lock(&mutex);
- pthread_mutex_unlock(&mutex);
- pthread_mutex_destroy(&mutex);

2.4.2 Condition Variables

- 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 that calls pthread_cond_wait() releases the associated mutex automatically.
- ➢ The thread then enters a waiting or blocked state. It remains blocked until another thread signals the condition variable.
- ➢ The signal can be sent using pthread_cond_signal() or pthread_cond_broadcast().
- ➢ Upon being signaled, the thread reacquires the mutex and becomes "ready" to run.

Functions:

- ○ pthread_cond_init(&cond, NULL);
- ○ pthread_cond_wait(&cond, &mutex);
- ○ pthread_cond_signal(&cond);
- ○ pthread_cond_broadcast(&cond);
- ○ pthread_cond_destroy(&cond);

2.4.3 Read-Write Locks:

- ➢ 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.

Functions:

- ✓ pthread_rwlock_init()
- ✓ pthread_rwlock_rdlock()
- ✓ pthread_rwlock_wrlock()
- ✓ pthread_rwlock_unlock()
- ✓ pthread_rwlock_destroy()

2.4.4. Semaphores:

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

2.5. Thread-Specific Data:

- **Thread-Specific Data** also called as **Thread-Local Storage.**
- It is a mechanism that allows **each thread to have its own private copy of a variable**, even though the variable is globally declared.
- Thread-specific data enables threads within the same process to store and access **independent data values**, preventing interference between threads.

Functions:

- ✓ pthread_key_create() – A **key** is created.
- ✓ pthread_setspecific() – Each thread sets its own value.
- ✓ pthread_getspecific() – Threads retrieve their values.
- ✓ pthread_key_delete() – Frees the **key** associated with thread-specific data.

2.6. Example: Creating Threads:

```
#include <pthread.h>
#include <stdio.h>
void* print_message(void* arg)
{
   char* msg = (char*) arg;
   printf("%s\n", msg);
   pthread_exit(NULL);
}
int main()
{
   pthread_t thread1, thread2;
   pthread_create(&thread1, NULL, print_message, "Hello from thread 1");
   pthread_create(&thread2, NULL, print_message, "Hello from thread 2");
   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);
   return 0;
}
```

2.7. Advantages of Pthreads:

- ✓ Standardized across UNIX systems.
- ✓ Efficient resource sharing.
- ✓ Supports fine-grained concurrency.

2.8. Disadvantages / Limitations:

- ✓ Complexity in writing thread-safe code.
- ✓ Risk of deadlocks if mutexes are mismanaged.
- ✓ Debugging multithreaded programs is harder.