Pthreads: A Comprehensive Guide

Pthreads, or POSIX threads, provide a standardized API for creating and managing threads in C/C++ programs. This guide covers essential concepts, functions, and synchronization mechanisms.

Introduction to Pthreads

A thread is a separate flow of control within a process. Pthreads allow you to create multiple threads that execute concurrently, enabling parallelism and improving performance.

Key Concepts

- **Thread:** An independent execution unit within a process.
- **Process:** A program in execution, containing one or more threads.
- **Concurrency:** Multiple threads making progress simultaneously.
- **Parallelism:** Multiple threads executing at the same time on different cores.

Creating Threads: `pthread_create`

The `pthread_create` function creates a new thread.

Function Signature

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

- `thread`: A pointer to a `pthread_t` variable that will store the ID of the new thread.
- `attr`: A pointer to a `pthread_attr_t` structure specifying thread attributes (can be `NULL` for default attributes).
- `start_routine`: A pointer to the function that the new thread will execute.
- `arg`: A pointer to the argument that will be passed to the `start_routine`.

Example Implementation

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>

using namespace std;

void *thread_function(void *arg) {
    int thread_id = *(int *)arg;
    cout << "Thread " << thread_id << ": Started" << endl;
    sleep(2);
    cout << "Thread " << thread_id << ": Finished" << endl;
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t threads[3];
    int thread_ids[3] = {1, 2, 3};

    for (int i = 0; i < 3; i++) {
        int result = pthread_create(&threads[i], NULL,
thread_function, &thread_ids[i]);
        if (result) {
            cerr << "Error creating thread " << i << ": " << result <<
endl;
            return 1;
        }
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    cout << "Main: All threads finished" << endl;
    return 0;
}
```

Critical Points

- Always check the return value of `pthread_create` to ensure the thread was created successfully. A return value of 0 indicates success. Any other value indicates an error. Common errors include:
  - `EAGAIN`: The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process was exceeded.
  - `EINVAL`: Invalid settings in attr.
  - `EPERM`: No permission to create a thread with the scheduling policy and parameters defined in attr.
- The `start_routine` must have a specific signature: `void *function_name(void *arg)`.
- The `arg` parameter allows you to pass data to the thread function. It's common to pass a pointer to a structure containing multiple values.

Joining Threads: `pthread_join`

The `pthread_join` function waits for a thread to terminate.

Function Signature

```
int pthread_join(pthread_t thread, void **retval);
```

- `thread`: The ID of the thread to wait for.
- `retval`: A pointer to a pointer that will store the return value of the thread (can be `NULL` if the return value is not needed).

Example Implementation

See the example in the `pthread_create` section.

Critical Points

- Calling `pthread_join` is essential to prevent memory leaks. When a thread terminates, its resources are not fully released until `pthread_join` is called.
- If a thread is detached (using `pthread_detach`), it is not joinable, and calling `pthread_join` will result in an error.
- `pthread_join` is a blocking call. The calling thread will wait until the target thread terminates.
- The `pthread_join` function returns 0 on success. Otherwise, it returns an error number. Common errors include:
  - `EINVAL`: The thread is not joinable or another thread is already waiting to join this thread.
  - `ESRCH`: No thread with the ID thread could be found.
  - `EDEADLK`: A deadlock was detected (e.g., the calling thread is trying to join itself).

Passing Arguments to Threads

Passing arguments to threads involves passing a pointer to the `pthread_create` function.

Example Implementation

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>

using namespace std;

struct ThreadArgs {
    int thread_id;
    string message;
};

void *thread_function(void *arg) {
    ThreadArgs *args = (ThreadArgs *)arg;
    cout << "Thread " << args->thread_id << ": " << args->message <<
endl;
    pthread_exit(NULL);
}

int main() {
    pthread_t thread;
    ThreadArgs args;
    args.thread_id = 1;
    args.message = "Hello from thread 1";
```

```
    int result = pthread_create(&thread, NULL, thread_function,
&args);
    if (result) {
        cerr << "Error creating thread: " << result << endl;
        return 1;
    }

    pthread_join(thread, NULL);
    cout << "Main: Thread finished" << endl;
    return 0;
}
```

Critical Points

- When passing arguments, ensure that the data pointed to by `arg` remains valid until the thread function accesses it. Avoid passing pointers to local variables that go out of scope.
- Consider using dynamically allocated memory (e.g., using `new`) to create the argument structure, and then free the memory within the thread function after the data has been used.

Mutexes: `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_unlock`

Mutexes (mutual exclusion locks) are used to protect shared resources from concurrent access.

Function Signatures

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- `pthread_mutex_lock`: Locks the mutex. If the mutex is already locked, the calling thread will block until the mutex becomes available.
- `pthread_mutex_trylock`: Tries to lock the mutex. If the mutex is already locked, it returns immediately with an error code (`EBUSY`).
- `pthread_mutex_unlock`: Unlocks the mutex.

Example Implementation

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>

using namespace std;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int shared_data = 0;

void *thread_function(void *arg) {
    int thread_id = *(int *)arg;
    pthread_mutex_lock(&mutex);
```

```
    shared_data++;
    cout << "Thread " << thread_id << ": Shared data = " <<
shared_data << endl;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[3];
    int thread_ids[3] = {1, 2, 3};

    for (int i = 0; i < 3; i++) {
        int result = pthread_create(&threads[i], NULL,
thread_function, &thread_ids[i]);
        if (result) {
            cerr << "Error creating thread " << i << ": " << result <<
endl;
            return 1;
        }
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    cout << "Main: All threads finished, Shared data = " <<
shared_data << endl;
    return 0;
}
```

Critical Points

- Always unlock a mutex after you are finished with the shared resource. Failing to do so can lead to deadlocks.
- Use `pthread_mutex_trylock` when you need to avoid blocking. Check the return value to see if the lock was acquired successfully. `pthread_mutex_trylock` returns 0 if it successfully acquires the lock, and `EBUSY` if the mutex is already locked.
- Ensure that the same thread that locks a mutex unlocks it.
- Recursive locking is not allowed by default. If you need recursive locking, use a recursive mutex (created with appropriate attributes).
- `pthread_mutex_lock` and `pthread_mutex_unlock` return 0 on success. Otherwise, they return an error number. Common errors include:
  - `EINVAL`: The mutex was not initialized.
  - `EAGAIN`: The system lacked the necessary resources (for `pthread_mutex_lock`).
  - `EPERM`: The calling thread does not own the mutex (for `pthread_mutex_unlock`).

- EDEADLK: The current thread already owns the mutex (for `pthread_mutex_lock`).

Condition Variables: `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`

Condition variables are used to signal threads waiting for a specific condition to become true.

Function Signatures

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- `pthread_cond_wait`: Atomically unlocks the mutex and waits on the condition variable. The thread is blocked until another thread calls `pthread_cond_signal` or `pthread_cond_broadcast`. Upon being signaled, the thread re-acquires the mutex before returning.
- `pthread_cond_signal`: Signals one thread waiting on the condition variable.
- `pthread_cond_broadcast`: Signals all threads waiting on the condition variable.

Example Implementation

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>

using namespace std;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int shared_data = 0;
bool data_ready = false;

void *producer_thread(void *arg) {
    pthread_mutex_lock(&mutex);
    shared_data = 100;
    data_ready = true;
    cout << "Producer: Data ready" << endl;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

void *consumer_thread(void *arg) {
    pthread_mutex_lock(&mutex);
    while (!data_ready) {
        cout << "Consumer: Waiting for data" << endl;
        pthread_cond_wait(&cond, &mutex);
    }
```

```
    cout << "Consumer: Data received = " << shared_data << endl;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

int main() {
    pthread_t producer, consumer;

    pthread_create(&producer, NULL, producer_thread, NULL);
    pthread_create(&consumer, NULL, consumer_thread, NULL);

    pthread_join(producer, NULL);
    pthread_join(consumer, NULL);

    cout << "Main: All threads finished" << endl;
    return 0;
}
```

Critical Points

- Always use a mutex in conjunction with a condition variable.
- The `pthread_cond_wait` function must be called with the mutex locked. It atomically unlocks the mutex and waits for the signal.
- When a thread is signaled and returns from `pthread_cond_wait`, it re-acquires the mutex.
- Use a loop to check the condition after returning from `pthread_cond_wait` to handle spurious wakeups.
- Use `pthread_cond_broadcast` when multiple threads may be waiting for the same condition.
- `pthread_cond_wait`, `pthread_cond_signal`, and `pthread_cond_broadcast` return 0 on success. Otherwise, they return an error number. Common errors include:
  - `EINVAL`: The value specified by cond or mutex is not valid.
  - `EPERM`: The mutex was not owned by the current thread (for `pthread_cond_wait`).

Barriers: `pthread_barrier_init`, `pthread_barrier_wait`, `pthread_barrier_destroy`

Barriers are used to synchronize multiple threads at a specific point in their execution.

Function Signatures

```
int pthread_barrier_init(pthread_barrier_t *barrier, const
pthread_barrierattr_t *attr, unsigned count);
int pthread_barrier_wait(pthread_barrier_t *barrier);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- `pthread_barrier_init`: Initializes a barrier with a specified count. The barrier is released when `count` threads have called `pthread_barrier_wait`.
- `pthread_barrier_wait`: Blocks the calling thread until `count` threads have called `pthread_barrier_wait`. When the last thread calls `pthread_barrier_wait`, all threads are unblocked, and the barrier is reset.
- `pthread_barrier_destroy`: Destroys a barrier.

Example Implementation

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>

using namespace std;

pthread_barrier_t barrier;
const int NUM_THREADS = 3;

void *thread_function(void *arg) {
    int thread_id = *(int *)arg;
    cout << "Thread " << thread_id << ": Before barrier" << endl;
    pthread_barrier_wait(&barrier);
    cout << "Thread " << thread_id << ": After barrier" << endl;
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS] = {1, 2, 3};

    pthread_barrier_init(&barrier, NULL, NUM_THREADS);

    for (int i = 0; i < NUM_THREADS; i++) {
        int result = pthread_create(&threads[i], NULL,
thread_function, &thread_ids[i]);
        if (result) {
            cerr << "Error creating thread " << i << ": " << result <<
endl;
            return 1;
        }
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_barrier_destroy(&barrier);
    cout << "Main: All threads finished" << endl;
    return 0;
}
```

Critical Points

- The `count` parameter in `pthread_barrier_init` specifies the number of threads that must call `pthread_barrier_wait` before the barrier is released.
- All threads that participate in a barrier must call `pthread_barrier_wait`.
- Barriers are reusable. After all threads have passed the barrier, it is automatically reset, and threads can use it again.
- Ensure that the barrier is properly destroyed using `pthread_barrier_destroy` when it is no longer needed.
- `pthread_barrier_init`, `pthread_barrier_wait`, and `pthread_barrier_destroy` return 0 on success. Otherwise, they return an error number. Common errors include:
  - `EINVAL`: The value specified by barrier is not valid.
  - `EBUSY`: The barrier is in use (for `pthread_barrier_destroy`).
  - `EAGAIN`: The system lacked the necessary resources to initialize another barrier.

Conclusion

Pthreads provide a powerful API for creating and managing threads in C/C++ programs. Understanding the concepts and functions discussed in this guide is essential for writing concurrent and parallel applications. Proper synchronization using mutexes, condition variables, and barriers is crucial to avoid race conditions and ensure correct program behavior.