Pthreads

POSIX Threads commonly knows as pthreads. Portable operating system interface(POSIX) is an execution model that exists independently from a programming language. A process is an instance of a running (or suspended) program. Threads are lightweight process. In a shared memory program a single process may have multiple threads of control is called pthreads. They provide a way to create and manage multiple threads within a single process.

For compile the program

```
# gcc -g -Wall -o pth_hello pth_hello.c -lpthread
```

For execution

```
# ./pth_hello 4
```

- -lpthread tells the compiler that we want to link in the pthreads library.

What are pthreads?

Pthreads, short for POSIX threads, are a set of standard C programming language thread programming interfaces. They provide a way to create and manage multiple threads within a single process. Pthreads are commonly used in Unix-like operating systems, including Linux, macOS, and various flavors of Unix.

Pthreads allow developers to write concurrent programs by dividing the program's execution into multiple threads that can run concurrently. Each thread represents an independent flow of control within the program, allowing different parts of the program to be executed simultaneously. This can lead to improved performance and responsiveness in multi-threaded applications.

The pthreads API provides functions for creating and terminating threads, managing thread attributes such as stack size and scheduling parameters, synchronizing threads using mutexes (mutual exclusion objects) and condition variables, and controlling thread-specific data.

Here are some key concepts related to pthreads:

1. Thread Creation: The pthread_create() function is used to create a new thread within a process. It takes a function pointer to the starting routine of the thread and any required arguments.
2. Thread Termination: Threads can be terminated explicitly using pthread_exit() or by returning from the thread's start routine. Terminating the main thread automatically terminates all other threads within the process.

3. Synchronization: Pthreads provide synchronization mechanisms like mutexes and condition variables to coordinate access to shared resources and enable thread communication.

4. Mutexes: Mutexes (short for mutual exclusion) are used to protect critical sections of code, ensuring that only one thread can execute that section at a time.

5. Condition Variables: Condition variables provide a way for threads to wait until a certain condition is met. They are typically used in conjunction with a mutex to implement thread synchronization.

6. Thread Joining: The pthread_join() function allows a thread to wait for the termination of another thread. This is useful when a thread needs to obtain the result or status of another thread.

Pthreads provide a powerful and widely used mechanism for concurrent programming in C. They offer flexibility and control over thread management and synchronization, allowing developers to write efficient multi-threaded applications.

Why pthreads?

Pthreads, or POSIX threads, provide a standardized thread programming interface in the C programming language. There are several reasons why pthreads are commonly used for concurrent programming:

1. Portability: Pthreads are designed to be portable across different operating systems and platforms. They are supported on Unix-like systems, including Linux, macOS, and various flavors of Unix. This portability allows developers to write thread-based applications that can run on different systems without significant modifications.
2. Standardization: Pthreads are part of the POSIX standard (IEEE Standard 1003.1c), which defines a set of APIs for operating system interfaces. Being a standard API, pthreads offer a consistent and well-defined interface for thread programming across different platforms. This makes it easier for developers to write portable code and reduces the need for platform-specific thread APIs.
3. Efficiency: Pthreads provide a lightweight and efficient mechanism for creating and managing threads. The overhead of thread creation and context switching is relatively low compared to some other threading models. This efficiency is crucial for achieving good performance in multi-threaded applications where multiple threads need to execute concurrently.
4. Flexibility and Control: Pthreads offer a high degree of control over thread management and synchronization. Developers have fine-grained control over thread creation, termination, synchronization, and communication through mechanisms like mutexes and

condition variables. This flexibility allows developers to optimize their code for specific requirements and tailor thread behavior to suit their application's needs.

5. Familiarity: Pthreads have been widely used for many years and have become a de facto standard for thread programming in C. Many developers are familiar with the pthreads API, making it easier for them to understand and maintain code that uses pthreads. Additionally, there are numerous resources, tutorials, and examples available for pthreads, making it easier for developers to learn and use the API effectively.

Overall, pthreads provide a reliable and widely supported threading API for concurrent programming in C. They offer portability, standardization, efficiency, flexibility, and familiarity, making them a compelling choice for developers working on multi-threaded applications.

Designing threaded programs in Pthreads

Designing threaded programs in Pthreads involves careful consideration of various aspects to ensure proper concurrency, synchronization, and thread management. Here are some key steps and considerations for designing threaded programs using Pthreads:

1. Identify Parallelizable Tasks: Analyze your application to identify tasks that can be executed concurrently. These tasks should be independent or have minimal dependencies on each other to enable effective parallel execution.
2. Determine Thread Structure: Decide how many threads you need and how they will be organized. Consider factors such as workload distribution, load balancing, and the number of available resources. The number of threads should be based on the available hardware resources (e.g., the number of CPU cores) and the nature of the tasks being performed.
3. Thread Creation: Use pthread_create() to create threads. Pass the appropriate function pointer as the thread's start routine and any necessary arguments. Ensure that the thread function is properly designed to execute the intended task independently or with minimal dependencies.
4. Synchronization and Data Sharing: Carefully manage shared data among threads to avoid race conditions and ensure consistent results. Use mutexes with pthread_mutex_lock() and pthread_mutex_unlock() to protect critical sections where multiple threads access shared resources. If there are dependencies between threads, consider using condition variables (pthread_cond_wait() and pthread_cond_signal()) to coordinate their execution.
5. Avoiding Deadlocks and Starvation: Be mindful of potential deadlocks and resource starvation situations. Deadlock occurs when threads are stuck waiting for resources that are held by other threads, resulting in a deadlock state. Resource starvation happens when a thread is perpetually denied access to resources it requires. Design your synchronization mechanisms and resource allocation strategies to avoid these issues.
6. Thread Joining and Termination: Use pthread_join() to wait for thread completion and retrieve any return values if needed. Properly handle thread termination using

pthread_exit() or by returning from the thread function. Make sure to join or detach all threads before the main thread exits to avoid resource leaks.

7. Error Handling: Check the return values of pthread functions for errors and handle them appropriately. Use error handling techniques, such as error codes or errno, to identify and react to potential issues during thread creation, synchronization, or other operations.
8. Performance Optimization: Consider performance optimizations like load balancing, minimizing thread synchronization, reducing overhead, and exploiting parallelism. Profile your threaded program, identify bottlenecks, and apply appropriate optimizations based on specific requirements.
9. Testing and Debugging: Thoroughly test your threaded program to ensure correctness and validate its behavior under different scenarios and workloads. Debugging multi-threaded programs can be challenging, so consider using debugging tools and techniques specifically designed for concurrent code.
10. Documentation and Maintenance: Document the design choices, synchronization mechanisms, and any assumptions made in the threaded program. This documentation will help understand and maintain the code in the future.

Designing threaded programs in Pthreads requires a good understanding of concurrency concepts, synchronization techniques, and the Pthreads API itself. It's essential to carefully plan the thread structure, data sharing, synchronization mechanisms, and error handling to ensure correct and efficient execution of the concurrent program.

pthreads APIs

The Pthreads API (POSIX threads API) provides a set of functions and types for creating, managing, and synchronizing threads in C-based programs. Here are some commonly used Pthreads APIs:

1. Thread Creation and Termination:
   a. `pthread_create()`: Creates a new thread and starts its execution.
   b. `pthread_exit()`: Terminates the calling thread and returns a value to the joining thread.
   c. `pthread_join()`: Waits for the termination of a specific thread and retrieves its return value.
2. Mutexes (Mutual Exclusion):
   a. `pthread_mutex_init()`: Initializes a mutex object.
   b. `pthread_mutex_destroy()`: Destroys a mutex object.
   c. `pthread_mutex_lock()`: Locks a mutex, blocking if it is already locked by another thread.
   d. `pthread_mutex_unlock()`: Unlocks a mutex, allowing other threads to lock it.
3. Condition Variables:
   a. `pthread_cond_init()`: Initializes a condition variable.

   b. `pthread_cond_destroy()`: Destroys a condition variable.
   c. `pthread_cond_wait()`: Atomically releases a mutex and waits on a condition variable until signaled.
   d. `pthread_cond_signal()`: Signals a condition variable, waking up one waiting thread.
   e. `pthread_cond_broadcast()`: Signals a condition variable, waking up all waiting threads.

4. Thread-Specific Data:
   a. `pthread_key_create()`: Creates a thread-specific data key.
   b. `pthread_key_delete()`: Deletes a thread-specific data key.
   c. `pthread_getspecific()`: Retrieves the value associated with a thread-specific data key.
   d. `pthread_setspecific()`: Sets the value associated with a thread-specific data key.
5. Thread Attributes:
   a. `pthread_attr_init()`: Initializes a thread attributes object.
   b. `pthread_attr_destroy()`: Destroys a thread attributes object.
   c. `pthread_attr_setdetachstate()`: Sets the thread's detach state (joinable or detached).
   d. `pthread_attr_getdetachstate()`: Retrieves the thread's detach state.
6. Thread Scheduling:
   a. `pthread_setschedparam()`: Sets the scheduling parameters (policy and priority) of a thread.
   b. `pthread_getschedparam()`: Retrieves the scheduling parameters of a thread.

These functions are part of the Pthreads API and can be used to create, manage, synchronize, and control the behavior of threads within a program. They allow developers to implement multi-threaded programs with proper synchronization and coordination between threads.

It's important to consult the Pthreads documentation and reference manual for detailed information on each function, including their return values, error handling, and usage guidelines.

Creating and Terminating threads in pthreads

In the pthreads library, you can create and terminate threads using the following functions:

1. `pthread_create`: This function is used to create a new thread. It takes four parameters: a pointer to the thread identifier, thread attributes (can be set to NULL for default attributes), a function pointer to the start routine of the thread, and an optional argument to pass to the start routine. The return value of this function indicates whether the thread creation was successful.

Example:

```c
#include <pthread.h
void* thread_function(void* arg)
{
   // Thread code goes here
   return NULL;
}
int main()
{
   pthread_t thread_id;
   int res = pthread_create(&thread_id, NULL, thread_function, NULL);
   if (res != 0) {
      // Error handling
   }
   // Rest of the main code
   return 0;
}
```

2. `pthread_join`: This function is used to wait for a specific thread to terminate. It takes two parameters: the thread identifier and a pointer to a location where the exit status of the thread will be stored. If you don't care about the exit status, you can pass NULL as the second parameter.

Example:

```c
#include <pthread.h>
void* thread_function(void* arg)
{
   // Thread code goes here
   return NULL;
}
int main()
{
   pthread_t thread_id;
```

```
    int res = pthread_create(&thread_id, NULL, thread_function, NULL);
    if (res != 0)
{
        // Error handling
}
    // Wait for the thread to terminate
    res = pthread_join(thread_id, NULL);
    if (res != 0)
{
        // Error handling
}
    // Rest of the main code
    return 0;
}
```

3. `pthread_exit`: This function is called from within a thread to terminate itself and return a value. It takes a single parameter that represents the exit status of the thread.

Example:

```c
#include <pthread.h>
void* thread_function(void* arg)
{
    // Thread code goes here
    pthread_exit(NULL);
}
int main()
{
    pthread_t thread_id;
    int res = pthread_create(&thread_id, NULL, thread_function, NULL);
    if (res != 0) {
        // Error handling
}
```

```
    // Rest of the main code
    return 0;
}
```

4. `pthread_cancel`: This function is used to request cancellation of another thread. It takes a single parameter which is the thread identifier of the thread to be canceled. The target thread must have created with the attribute `PTHREAD_CANCEL_ENABLE` for it to be able to be canceled.

Example:

```c
#include <pthread.h>
void* thread_function(void* arg)
{
    while (1) {
        // Thread code goes here
    }
    return NULL;
}
int main()
{
    pthread_t thread_id;
    int res = pthread_create(&thread_id, NULL, thread_function, NULL);
    if (res != 0) {
        // Error handling
}
    // Cancel the thread
    res = pthread_cancel(thread_id);
    if (res != 0)
{
        // Error handling
}
    // Rest of the main code
    return 0;
}
```

Note that thread cancellation should be used with caution as it can lead to resource leaks and other issues if not handled properly in the target thread.

Passing argument to threads in pthreads

In POSIX threads (pthreads), you can pass arguments to threads by using the `pthread_create` function. The argument is passed as a void pointer, so you can pass any type of data by casting it to a void pointer.

Here's an example of how you can pass an argument to a thread using pthreads:

```c
#include <pthread.h>
#include <stdio.h>
// Function executed by the thread
void* thread_function(void* arg) {
    // Cast the argument back to its original type
    int value = *(int*)arg;

    // Perform thread operations using the argument
    printf("Thread argument: %d\n", value);
    // Exit the thread
    pthread_exit(NULL);
}
int main() {
    pthread_t thread;
    int argument = 42;
    // Create the thread and pass the argument
    pthread_create(&thread, NULL, thread_function, (void*)&argument);
    // Wait for the thread to finish
    pthread_join(thread, NULL);
    return 0;
}
```

`` ``` ``

In this example, we define a thread function `thread_function` that takes a void pointer as an argument. Inside the thread function, we cast the argument back to an integer and use it. In the main function, we create a thread using `pthread_create` and pass the address of the `argument` variable as the argument to the thread. Finally, we wait for the thread to finish using `pthread_join`.

Remember to take care of the memory management when passing arguments between threads. If the lifetime of the argument is longer than the thread execution, you should ensure the data remains valid until the thread finishes using it.

Joining and detaching threads in pthreads

In POSIX threads (pthreads), you can use the functions `pthread_join` and `pthread_detach` to control the lifecycle of threads.

1. `pthread_join`: This function allows a calling thread to wait for the termination of another thread. The syntax is as follows:

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

- `thread`: The thread identifier of the thread to join.

- `retval`: A pointer to a location where the exit status of the joined thread will be stored (if needed). It can be set to `NULL` if the exit status is not required.

The `pthread_join` function blocks the calling thread until the specified thread terminates. If the thread has already terminated, `pthread_join` returns immediately. After joining a thread, its resources are released back to the system.

Here's an example that demonstrates the usage of `pthread_join`:

```c
#include <pthread.h>

#include <stdio.h>

void* thread_function(void* arg) {

    printf("Thread executing\n");

    pthread_exit(NULL);

}

int main() {
```

```
    pthread_t thread;
    pthread_create(&thread, NULL, thread_function, NULL);
    // Wait for the thread to finish
    pthread_join(thread, NULL);
    printf("Thread joined\n");
    return 0;
}
```

2. `pthread_detach`: This function allows a thread to be detached from the calling thread, enabling it to continue execution independently. The syntax is as follows:

```c
int pthread_detach(pthread_t thread);
```

- `thread`: The thread identifier of the thread to detach.

Once a thread is detached, its resources are automatically released when it terminates, and you cannot join or obtain its exit status anymore.

Here's an example that demonstrates the usage of `pthread_detach`:

```c
#include <pthread.h>
#include <stdio.h>
void* thread_function(void* arg) {
    printf("Thread executing\n");
    pthread_exit(NULL);
}
int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_function, NULL);
    // Detach the thread
    pthread_detach(thread);
    printf("Thread detached\n");
    return 0;
}
```

~~~
```
~~~

In this example, the thread is detached immediately after creation. The main thread does not wait for the detached thread to finish, and the detached thread continues execution independently.

It's important to note that detached threads cannot be joined, so you must ensure that a detached thread doesn't access any shared resources that may be modified or deallocated by other threads.