Multi-threading in C Using Pthreads Part 2

Ahmed Métwalli

December 20, 2024

Objective

- Understand the concept of threads and how they work.
- Learn how to create and manage threads in C.
- Observe concurrency, thread dependencies, and interleaving of outputs.

Code Snippet 1: Basic Thread Management

Steps to Implement the Code

1. Include the necessary headers: `stdio.h`, `stdlib.h`, `unistd.h`, `time.h`, and `pthread.h`.
2. Define a structure `Targs` to encapsulate thread arguments (`label` and `limit`).
3. Write a thread routine `my_routine` that:
   - Accepts a pointer to `Targs`.
   - Iterates up to `limit` and prints the thread label and progress.
   - Introduces a random delay using `usleep()`.
4. In the `main` function:
   - Seed the random number generator using `srand(time(0))`.
   - Create an array of threads and arguments.
   - Loop to initialize `label` and `limit`, then create threads using `pthread_create`.
   - Run a loop in the main thread to simulate its task.
   - Wait for all threads to complete using `pthread_join`.

Description

This code demonstrates basic thread creation and management in C. Each thread is assigned a unique task, and all threads run concurrently with the main thread. The main thread waits for all threads to complete before terminating.

Code

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

// Define the number of threads to create
```

```c
#define NUM 6

// Define a structure to hold thread-specific arguments
typedef struct {
    char label; // A character label to identify the thread (e.g., 'A', 'B', etc.)
    int limit;  // The number of iterations the thread will perform
} Targs;

// Thread routine executed by each thread
void* my_routine(void* raw_args) {
    // Cast the void* argument back to a Targs* to access the thread-specific data
    Targs* args = (Targs*) raw_args;

    // Loop up to the limit specified in the Targs structure
    for (int i = 0; i < args->limit; i++) {
        // Print the thread's label, current iteration, and total iterations
        printf("%c: %d / %d\n", args->label, i + 1, args->limit);

        // Introduce a random delay to simulate variable task durations
        usleep(rand() % (1000 * 1000)); // Sleep for up to 1 second
    }

    return NULL; // Return NULL to indicate successful execution
}

int main() {
    // Seed the random number generator with the current time
    srand(time(0));

    // Array to hold thread IDs
    pthread_t tids[NUM];

    // Array to hold thread arguments
    Targs args[NUM];

    // Create NUM threads
    for (int t = 0; t < NUM; t++) {
        // Assign a unique label to each thread ('A', 'B', etc.)
        args[t].label = 'A' + t;

        // Assign a random limit (number of iterations) between 1 and 5
        args[t].limit = 1 + rand() % 5;

        // Create the thread, passing its specific arguments
        pthread_create(&tids[t], NULL, my_routine, &args[t]);
    }

    // Main thread performs its own task
    for (int i = 0; i < 4; i++) {
        // Print the main thread's progress
        printf("main: %d\n", i);

        // Introduce a random delay to simulate work
        usleep(rand() % (1000 * 1000)); // Sleep for up to 1 second
    }

    // Wait for all threads to finish
```

```
    for (int i = 0; i < NUM; i++) {
        pthread_join(tids[i], NULL); // Block until thread i finishes
    }

    return 0; // Indicate successful program termination
}
```

Expected Output

The output will vary due to random delays introduced by `usleep()`. A sample output might look like this:

```
A: 1 / 5
C: 1 / 4
B: 1 / 2
D: 1 / 4
main: 0
E: 1 / 3
D: 2 / 4
B: 2 / 2
C: 2 / 4
main: 1
E: 2 / 3
A: 2 / 5
C: 3 / 4
A: 3 / 5
D: 3 / 4
C: 4 / 4
E: 3 / 3
main: 2
A: 4 / 5
D: 4 / 4
A: 5 / 5
main: 3
```

Observations

- Threads run concurrently, and their outputs interleave based on the operating system's scheduling and random delays.
- Each thread completes its iterations independently and stops.
- The main thread operates in parallel with the worker threads and completes its task.
- The `pthread_join` ensures the main program waits for all threads to finish before exiting.

Code Snippet 2: Threads with Dependencies

Steps to Implement the Code

1. Include the necessary headers: `stdio.h`, `stdlib.h`, `unistd.h`, `time.h`, and `pthread.h`.
2. Define a structure `Targs` to encapsulate thread arguments (`label`, `limit`, and `prerequisite`).
3. Write a thread routine `my_routine` that:
    - Accepts a pointer to `Targs`.
    - Waits for the prerequisite thread to finish using `pthread_join`.

- Iterates up to `limit` and prints the thread label and progress.
- Introduces a random delay using `usleep()`.

4. In the `main` function:
   - Seed the random number generator using `srand(time(0))`.
   - Create an array of threads and arguments.
   - Loop to initialize `label`, `limit`, and `prerequisite`, then create threads using `pthread_create`.
   - Set dependencies such that odd-indexed threads depend on the previous thread.
   - Wait for all threads to complete using `pthread_join`.

Description

This code extends the concept of multi-threading by introducing thread dependencies. Certain threads must wait for others to finish before starting their execution, demonstrating synchronization between threads.

Code

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

// Define the number of threads to create
#define NUM 6

// Structure to hold thread-specific arguments
typedef struct {
    char label;             // A unique label to identify the thread (e.g., 'A', 'B', etc.)
    int limit;              // The number of iterations the thread will perform
    pthread_t prerequiste; // Thread dependency: the thread that must complete before this one starts
} Targs;

// Thread routine executed by each thread
void* my_routine(void* raw_args) {
    // Cast the void* argument to a Targs* to access thread-specific data
    Targs* args = (Targs*) raw_args;

    // Wait for the prerequisite thread to finish, if any
    pthread_join(args->prerequiste, NULL);

    // Perform the thread's work
    for (int i = 0; i < args->limit; i++) {
        // Print the thread's label and progress
        printf("%c: %d / %d\n", args->label, i + 1, args->limit);

        // Introduce a random delay to simulate work
        usleep(rand() % (1000 * 1000)); // Sleep for up to 1 second
    }

    return NULL; // Indicate successful thread execution
}

int main() {
```

```
38      // Seed the random number generator
39      srand(time(0));
40
41      // Array to hold thread IDs
42      pthread_t tids[NUM];
43
44      // Array to hold thread arguments
45      Targs args[NUM];
46
47      // Create NUM threads
48      for (int t = 0; t < NUM; t++) {
49          // Assign a unique label to each thread ('A', 'B', ..., 'F')
50          args[t].label = 'A' + t;
51
52          // Assign a random limit (number of iterations) between 1 and 5
53          args[t].limit = 1 + rand() % 5;
54
55          // Define thread dependencies: odd-indexed threads depend on the
               completion of the previous thread
56          if (t % 2 != 0) { // For threads 1, 3, 5
57              args[t].prerequiste = tids[t - 1]; // Dependency on the thread at
                   index t-1
58          } else {
59              args[t].prerequiste = (pthread_t)0; // No dependency for even-indexed
                    threads
60          }
61
62          // Create the thread, passing its specific arguments
63          pthread_create(&tids[t], NULL, my_routine, &args[t]);
64      }
65
66      // Main thread performs its own task
67      for (int i = 0; i < 4; i++) {
68          // Print the main thread's progress
69          printf("main: %d\n", i);
70
71          // Introduce a random delay to simulate work
72          usleep(rand() % (1000 * 1000)); // Sleep for up to 1 second
73      }
74
75      // Wait for all threads to finish
76      for (int i = 0; i < NUM; i++) {
77          pthread_join(tids[i], NULL); // Block until thread i finishes
78      }
79
80      return 0; // Indicate successful program termination
81  }
```

The output will vary due to random delays introduced by `usleep()`.

Observations

- Threads with dependencies wait for their prerequisite threads to finish before starting.
- Even-indexed threads start immediately, while odd-indexed threads depend on the preceding thread.
- Threads run concurrently wherever possible, and their outputs interleave due to random delays.
- The `pthread_join` within the thread routine ensures synchronization between dependent threads.

Discussion

This example demonstrates how to introduce thread dependencies in multi-threaded programs. It shows how:

- Threads can be synchronized using `pthread_join`.
- Dependencies can be dynamically assigned to control execution order.
- Concurrency can still be achieved for independent threads.

Thread dependencies are useful for tasks that require specific sequences of execution while leveraging the benefits of multi-threading.