Multithreading in C: Understanding and Managing Threads

Introduction to Threads

A thread is a basic unit of CPU utilization, which consists of a program counter, a set of registers, and a stack space. Unlike traditional processes, threads share: - Code section - Data section (variables) - Open files and signals

[[**Threads vs. Processes**]]

- **Process**: An independent program with its own memory space.
- **Thread**: A "lightweight" process that shares its memory space with other threads in the same process.

Key Advantage: The primary benefit of using threads over creating new processes is the efficiency in resource sharing. Since threads share the same memory space, they are quicker to create and switch between than processes.

Thread Management in C

To use threads in C, you need to use the [[POSIX threads]] (pthreads) library. This involves including the `pthread.h` library and using specific functions to create and manage threads. Here's a summary of the main functions used in managing threads:

Including the Library

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

Thread Creation

The basic syntax for creating a thread is:

```
pthread_t tid;
pthread_create(&tid, NULL, thread_function, argument);
```

- `pthread_t tid`: Declares a thread identifier.
- `&tid`: Pointer to store the thread ID.
- `NULL`: Specifies default thread attributes.
- `thread_function`: Function that the thread will execute.
- `argument`: A pointer to arguments for the thread function (can be `NULL`).

Important Thread Functions

- **`pthread_self()`**: Returns the thread ID of the calling thread.
- **`pthread_exit(void *status)`**: Terminates a thread and optionally returns a status.

- `pthread_cancel(pthread_t tid)`: Cancels a thread.
- `pthread_join(pthread_t tid, void **status)` : Makes the calling thread wait for another thread to terminate.
- `pthread_detach(pthread_t tid)`: Marks a thread as detached, meaning its resources are automatically released upon termination.

Example Code: Basic Thread Creation

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

int var = 0;

void* thread_function(void* arg) {
    printf("Thread %ld: started\n", pthread_self());
    sleep(1);
    var = 10;
    printf("Thread %ld: var updated to %d\n", pthread_self(), var);
    sleep(2);
    printf("Thread %ld: finished\n", pthread_self());
    return NULL;
}

int main() {
    pthread_t tid;
    printf("Main thread %ld: start (var = %d)\n", pthread_self(), var);
    pthread_create(&tid, NULL, thread_function, NULL);
    printf("Main thread %ld: created thread %ld\n", pthread_self(), tid);
    sleep(3);
    printf("Main thread %ld: end (var = %d)\n", pthread_self(), var);
    return 0;
}
```

In this example: - A thread is created using `pthread_create` to execute `thread_function`. - The main thread continues its execution while the newly created thread runs `thread_function` in parallel. - Shared variable `var` is modified by the thread.

[[More Examples on threads]] ## Exercises

Exercise 1: Parallel Calculation

Write a program that accepts four decimal arguments `a`, `b`, `c`, and `d`, and calculates `a * b + c * d` using multithreading: - Create two threads: - **Thread 1** calculates `a * b`. - **Thread 2** calculates `c * d`. - The main thread waits for the results of both threads and sums them.

Exercise 2: Struct with Random Values

Create a `TypeTable` structure with the following fields: - An integer array. - Number of elements in the array. - An integer `x`.

Write a program that: - Creates a thread to fill the array with random integers between `0` and `99`. - Reads an integer `x` from the user in the main thread. - After the array is filled, create another thread to check if `x` is in the array and return `1` if found, `0` otherwise.

Concepts to Remember

- **Concurrency**: Threads can execute concurrently, allowing a program to make better use of CPU resources by performing multiple operations at the same time.
- **Synchronization**: When multiple threads share data, proper synchronization mechanisms like mutexes or semaphores are necessary to avoid race conditions.
- **Race Conditions**: When two threads access shared data simultaneously, leading to unpredictable results.
- **Thread Safety**: Ensuring that shared resources are accessed safely by multiple threads.

These notes and exercises should help you solidify your understanding of threading in C. If you need further explanations or more advanced concepts like synchronization, feel free to ask!