Title: NTU IM Operating-System HW 02
Student ID: R12631070
Name: 林育新

使用方式

```
Make
./hw2_Q1 <線程數量> <每個線程當中有幾個點>
./hw2_Q2 <數列長度>
./hw3_Q3 // 要去cpp程式內 修改輸入陣列
```

作業處理邏輯

hw2_Q1.c 對應到 4.17 (Q1)

題目: 蒙特卡羅方法估算圓周率 π (使用 Pthreads)

1. **算法解釋**：
   蒙特卡羅方法: 用大量隨機資料來模擬問題的解答，並從中估算出結果。
   假設有一個半徑為1的圓(面積為π)，內接於一個邊長為2的正方形(面積為4)中。
   隨機產生大量的點落在這個正方形內，統計有多少點落在圓內。
   估計的π值：π ≈ 4 × (圓內點數量) / (總點數量)； 概念相當於 正方形面積 * (圓形面積/正方形面積)。

2. **子線程邏輯**：

   - 生成隨機點並計算圓內點數，結束前用mutex更新global的point數量。

```
void *generate_points(void *arg)
{
    // 把輸入參數強制轉型成所需格式，獲取每個線程點的數量。(只能傳入一個指標，
如果想要傳入多個，只能用物件或結構體包起來。)
    unsigned long long num_points = *((unsigned long long *)arg);

    // 建立每個線程單獨的計數器
    unsigned long long local_count = 0;

    // 設置隨機數生成器
    std::random_device rd;                                  // 真隨機數設備
    std::mt19937 gen(rd());                                 // 使用Mersenne
Twister算法
    std::uniform_real_distribution<> dis(-1.0, 1.0);  // 均勻分布在
[-1,1]

    // 生成隨機點並檢查是否在圓內
    for (unsigned long long i = 0; i < num_points; ++i)
    {
```

```
            double x = dis(gen); // 隨機x座標
            double y = dis(gen); // 隨機y座標

            // 檢查點是否在單位圓內 (x² + y² ≤ 1)
            if (x * x + y * y <= 1.0)
            {
                local_count++; // 如果在圓內，本地計數加1
            }
        }

        // 使用互斥鎖安全地更新全局計數
        pthread_mutex_lock(&mutex);          // 上鎖
        points_in_circle += local_count;     // 更新圓內點數
        total_points += num_points;          // 更新總點數
        pthread_mutex_unlock(&mutex);        // 解鎖

        return nullptr;
    }
```

3. **主線程邏輯：**

   - pthread_create 創建線程。
   - pthread_join 等待所有子線程完成以後，計算π的估計值。

```
pthread_t threads[num_threads]; // 創建線程陣列

// 創建並啟動所有線程
for (int i = 0; i < num_threads; ++i)
{
    // pthread_create的參數：線程ID、線程屬性、線程函數、傳遞給線程函數的參
數。(參數只能傳入一個指標，如果想要傳入多個，只能用物件或結構體包起來。)
    pthread_create(&threads[i], nullptr, generate_points,
&points_per_thread);
}

// 等待所有線程完成
for (int i = 0; i < num_threads; ++i)
{
    pthread_join(threads[i], nullptr);
}

// 計算並輸出π的估算值
double pi_estimate = 4.0 * points_in_circle / total_points;
std::cout << "估算的π值: " << pi_estimate << std::endl;
std::cout << "總點數: " << total_points << std::endl;
std::cout << "圓內點數: " << points_in_circle << std::endl;
```

hw2_Q2.c 對應到 4.21 (Q2)

題目: 類似上次作業，印出斐波那契序列。但把fork改為multi-thread來做。主線程僅等待子線程結束，不參與計算，僅印出子線程返回的結果。

- 斐波那契數列前兩項為0, 1
- 後續計算方式為 a[n] = a[n-1] + a[n-2]

1. **子線程邏輯：**

```
void *generate_fibonacci(void *arg)
{
    ThreadData *data = (ThreadData *)arg;

    // 前兩項特例
    if (data->length >= 1)
    {
        data->sequence.push_back(0); // fib₀ = 0
    }
    if (data->length >= 2)
    {
        data->sequence.push_back(1); // fib₁ = 1
    }

    // 從第三項開始計算斐波那契數列: fibₙ = fibₙ₋₁ + fibₙ₋₂
    for (int i = 2; i < data->length; ++i)
    {
        long long next = data->sequence[i - 1] + data->sequence[i - 2];
        data->sequence.push_back(next);
    }

    return nullptr;
}
```

2. **主線程邏輯：**

   - 事先準備一個struct，用於傳遞子線程計算的斐波那契結果。

   ```
   struct ThreadData
   {
       int length;                       // 要生成的斐波那契數列長度
       std::vector<long long> sequence; // 儲存生成的數列
   };
   ```

   - pthread_create 創建線程。

   - pthread_join 子線程完成以後，印出計算的斐波那契結果。

   ```
   // atoi函數將CLI輸入的字串轉換為整數
   int length = std::atoi(argv[1]);
   ```

```
if (length < 0)
{
    std::cerr << "錯誤: 長度必須是非負整數\n";
    return 1;
}

// 準備線程資料
ThreadData data;
data.length = length;

pthread_t thread;

// 創建線程來生成斐波那契數列
if (pthread_create(&thread, nullptr, generate_fibonacci, &data) != 0)
{
    std::cerr << "錯誤: 無法創建線程\n";
    return 1;
}

// 等待線程完成
if (pthread_join(thread, nullptr) != 0)
{
    std::cerr << "錯誤: 無法等待線程\n";
    return 1;
}
```

hw2_Q3.c 對應到 Project 2 - multithreaded sorting application (Q3)

題目: 輸入一個列表，將列表分為一半，丟到兩個線程中進行排序。
然後再用一個線程，合併兩個已排序的列表。
父線程僅在最後印出結果。

1. **子線程邏輯：**
    - 使用 std::sort (通常是quick sort) 對分半的陣列排序。

```
// 排序線程函數
void *sort_thread(void *arg)
{
    SortParams *params = (SortParams *)arg;

    // 使用 std::sort (通常是 quick sort) 對指定範圍的子陣列進行排序
    std::sort(original_array.begin() + params->start_index,
              original_array.begin() + params->end_index + 1);

    return nullptr;
}
```

    - 使用 merge sort 把兩個線程的陣列合併。

```
// 合併線程函數
void *merge_thread(void *arg)
{
    size_t mid = original_array.size() / 2 - 1;
    size_t i = 0;       // 左半部分索引
    size_t j = mid + 1; // 右半部分索引
    size_t k = 0;       // 合併陣列索引

    // 合併兩個已排序的子陣列
    while (i <= mid && j < original_array.size())
    {
        if (original_array[i] <= original_array[j])
        {
            sorted_array[k++] = original_array[i++];
        }
        else
        {
            sorted_array[k++] = original_array[j++];
        }
    }

    // 複製剩餘元素
    while (i <= mid)
    {
        sorted_array[k++] = original_array[i++];
    }

    while (j < original_array.size())
    {
        sorted_array[k++] = original_array[j++];
    }

    return nullptr;
}
```

2. **主線程邏輯：**

   - 主線程負責創建線程與等待，印出最初的陣列、部分排序後的陣列、最終完成排序的陣列。

```
// 計算分割點
size_t mid = original_array.size() / 2 - 1; // size_t 是無號整數 用於表示
索引

// 創建排序線程參數
SortParams params1 = {0, mid};
SortParams params2 = {mid + 1, original_array.size() - 1};

pthread_t sort_thread1, sort_thread2, merge_thread1;

// 創建並啟動排序線程
pthread_create(&sort_thread1, nullptr, sort_thread, &params1);
```

```
    pthread_create(&sort_thread2, nullptr, sort_thread, &params2);

    // 等待排序線程完成
    pthread_join(sort_thread1, nullptr);
    pthread_join(sort_thread2, nullptr);

    // 創建並啟動合併線程
    pthread_create(&merge_thread1, nullptr, merge_thread, nullptr);

    // 等待合併線程完成
    pthread_join(merge_thread1, nullptr);
```