Guida alla Programmazione Concorrente con Esempi in C++

Introduzione ai Thread

I thread sono componenti essenziali della programmazione concorrente, permettendo l'esecuzione parallela di più istruzioni all'interno di un programma. Questo è utile in contesti di sistemi multi-processore o multi-core, dove ogni core può eseguire un thread separato, migliorando le prestazioni complessive e la reattività delle applicazioni.

Vantaggi principali dei thread:

1. Parallelismo: Eseguire diverse parti di un programma simultaneamente, sfruttando i sistemi multi-core.
2. Reattività: Mantenere l'interfaccia utente reattiva eseguendo operazioni lunghe o bloccanti in background.
3. Semplicità di design: Alcuni algoritmi e modelli di programmazione sono più facilmente espressi con i thread.
4. Isolamento dei compiti: Suddivisione di compiti complessi in unità più gestibili eseguite da thread separati.

Esempi di Codice in C++

Esempio 1: Creazione e gestione di thread

```
#include <iostream>
#include <thread>

// Funzione eseguita dal thread
void threadFunction(int id) {
```

```
    std::cout << "Thread " << id << " is running
";
}

int main() {
    std::thread t1(threadFunction, 1);
    std::thread t2(threadFunction, 2);

    t1.join();
    t2.join();

    std::cout << "Threads have finished execution
";
    return 0;
}
```

Esempio 2: Sincronizzazione con mutex

```
#include <iostream>
#include <thread>
#include <mutex>

int counter = 0;
std::mutex mtx;
```

```
void increaseCounter() {
    for (int i = 0; i < 100000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);
        ++counter;
    }
}

int main() {
    std::thread t1(increaseCounter);
    std::thread t2(increaseCounter);

    t1.join();
    t2.join();

    std::cout << "Counter value: " << counter << '
';
    return 0;
}
```

Esempio 3: Problema produttore-consumatore

```
#include <iostream>
#include <thread>
#include <queue>
#include <mutex>
```

```
#include <condition_variable>

const int bufferSize = 10;
std::queue<int> buffer;
std::mutex mtx;
std::condition_variable cv;

void producer() {
    int item = 0;
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [] { return buffer.size() < bufferSize; });
        buffer.push(item++);
        std::cout << "Produced: " << item << std::endl;
        cv.notify_all();
        lock.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

void consumer() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [] { return !buffer.empty(); });
        int item = buffer.front();
```

```
        buffer.pop();
        std::cout << "Consumed: " << item << std::endl;
        cv.notify_all();
        lock.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(150));
    }
}

int main() {
    std::thread prod(producer);
    std::thread cons(consumer);

    prod.join();
    cons.join();

    return 0;
}
```

Soluzioni degli Esercizi

Esercizio 1: Calcolare il complemento a 1 di un numero binario memorizzato in un array di 100 posizioni parallelizzando il calcolo attraverso 5 thread

```
#include <iostream>
#include <thread>
#include <vector>
```

```
const int arraySize = 100;
const int numThreads = 5;
int binaryArray[arraySize] = { /* array inizializzato con valori binari */ };

void complementSection(int start, int end) {
    for (int i = start; i < end; ++i) {
        binaryArray[i] = ~binaryArray[i] & 1; // Calcola il complemento a 1
    }
}

int main() {
    std::vector<std::thread> threads;
    int sectionSize = arraySize / numThreads;

    for (int i = 0; i < numThreads; ++i) {
        int start = i * sectionSize;
        int end = (i == numThreads - 1) ? arraySize : start + sectionSize;
        threads.emplace_back(complementSection, start, end);
    }

    for (auto& t : threads) {
        t.join();
    }
```

```
    std::cout << "Completato il complemento a 1 dell'array.
";
    return 0;
}
```

Esercizio 2: Programma con 5 procedure A, B, C, D, E tali che A e B vengono eseguite in parallelo prima di C, e D ed E vengono eseguite in parallelo dopo C

```
#include <iostream>
#include <thread>

void procedureA() { std::cout << "Esecuzione di A
"; }
void procedureB() { std::cout << "Esecuzione di B
"; }
void procedureC() { std::cout << "Esecuzione di C
"; }
void procedureD() { std::cout << "Esecuzione di D
"; }
void procedureE() { std::cout << "Esecuzione di E
"; }

int main() {
    std::thread tA(procedureA);
    std::thread tB(procedureB);
```

```
    tA.join();
    tB.join();

    std::thread tC(procedureC);
    tC.join();

    std::thread tD(procedureD);
    std::thread tE(procedureE);

    tD.join();
    tE.join();

    std::cout << "Completato l'esecuzione delle procedure.
";
    return 0;
}
```

Esercizio 3: Comportamento del programma multithreaded

```
#include <iostream>
#include <thread>

int A[3] = {0, 0, 0};
int i = 0;
```

```
int k = 0;


void T1() {
    while (i < 2) {
        A[k] = 1;
        i = i + 1;
    }
}


void T2() {
    A[i] = 2;
    k = k + 1;
}


void T3() {
    A[k] = 3;
}


int main() {
    std::thread t1(T1);
    std::thread t2(T2);
    std::thread t3(T3);


    t1.join();
    t2.join();
```

```
    t3.join();


    std::cout << "Array A: ";
    for (int j = 0; j < 3; ++j) {
        std::cout << A[j] << " ";
    }
    std::cout << "
";
    return 0;
}
```