**8. 작업 취소와 예외 처리가 객체 소멸에 미치는 영향을 이해할것**

**부모 Task 그룹과 자식 Task 그룹으로 구성된 병렬 작업 트리에서 부모 Task 그룹이 취소되면 자식 Task 그룹도 취소되어 더이상 실행되지 않는다는 특징을 가지고 있습니다.**이러한 메카니즘은 필요없는 연산을 수행하지 않도록 해주고 매끄럽게 병렬 작업 트리를 중단하는데 도움이 되지만, 만약 자식 Task가  리소스를 해제하는 등의 중요한 작업을 담당하고 있다면 문제가 발생하게 됩니다.

지금부터 설명할 예제에서는 리소스를 의미하는 Resource 클래스와 이 리소스를 담는 컨테이너 역할을 하는 Container 클래스를 선언하고 Container 클래스 소멸자에서 병렬적으로 리소스를 해제하는 코드를 구현해보도록 하겠습니다.

|  |
| --- |
| // parallel-resource-destruction.h #pragma once #include <ppl.h> #include <sstream> #include <iostream>  // 리소스를 구현한 클래스 class Resource { public:     Resource(const std::wstring& name)         : \_name(name)     {     }      // 리소스 해제     void cleanup()     {         // 리소스 해제 시뮬레이션         std::wstringstream ss;         ss << \_name << L": Freeing..." << std::endl;         std::wcout << ss.str();     } private:     // 리소스명     std::wstring \_name; };  // 리소스를 담는 컨테이너를 구현한 클래스 class Container { public:     Container(const std::wstring& name)         : \_name(name)         , \_resource1(L"Resource 1")         , \_resource2(L"Resource 2")         , \_resource3(L"Resource 3")     {     }      ~Container()     {         std::wstringstream ss;         ss << \_name << L": Freeing resources..." << std::endl;         std::wcout << ss.str();          // \_resource1과 \_resource2는 병렬 처리되어 동시에 해제 작업이 진행되고         // \_resource3는 \_resource1과 \_resource2가 모두 해제된 이후에 해제 작업 처리         **concurrency::parallel\_invoke(             [this]() { \_resource1.cleanup(); },             [this]() { \_resource2.cleanup(); }         );**          **\_resource3.cleanup();**     }  private:     // 컨테이너명     std::wstring \_name;      // 컨테이너가 관리하는 리소스     Resource \_resource1;     Resource \_resource2;     Resource \_resource3; }; |

Container 클래스의 소멸자 구현을 보면 parallel\_invoke 함수를 이용하여 \_resource1과 \_resource2를 동시에 해제하고 두 리소스가 모두 해제되면 \_resource3 의 해제 작업을 처리하는 것을 볼 수 있습니다. 이러한 방식의 구현은 일반적인 상황에서는 문제될 것이 없어보이지만 만약 Container 클래스가 다른 Task 내부에서 선언되어 있다면 얘기가 달라집니다.

그럼 이에 대한 예제로 Task 그룹을 하나 생성하고 이 그룹의 자식 Task에서 Container 객체를 생성하여 이 객체가 소멸되기 전에 작업을 취소하였을 경우 어떻게 동작하는지 알아보록 하죠. 여러분들은 아래에 구현된 코드를 보고 출력 결과가 어떻게 나올지 예측해보시기 바랍니다.

|  |
| --- |
| #include "parallel-resource-destruction.h"  using namespace concurrency; using namespace std;  static\_assert(false, "This example illustrates a non-recommended practice.");  int main() {     // 두개의 자식 Task를 실행하게될 Task 그룹을 생성     task\_group tasks;      // 두개의 Task의 동작을 동기화하기 위한 객체     event e1, e2;      // 두개의 Task를 동시에 실행하고 동기화 오브젝트를 이용하여 처리 순서를 보장     tasks.run([&tasks,&e1,&e2] {         // Container 객체 생성         **Container c(L"Container 1");**          // 두번째 Task의 대기 상태 해제         **e2.set();**          // Task 그룹의 작업 취소될 때까지 대기         **e1.wait();**     });      tasks.run([&tasks,&e1,&e2] {         // Container 객체가 생성될 때까지 대기         **e2.wait();**          // Task 그룹의 작업 취소         **tasks.cancel();**          // 첫번째 Task의 대기 상태 해제         **e1.set();**     });      // 자식 Task의 종료 대기     tasks.wait();      wcout << L"Exiting program..." << endl; }   // 실행결과  Container 1: Freeing resources... Exiting program... |

실제 출력 결과가 여러분들이 예측한 결과와 일치했나요? 아마도 실제 출력 결과가 많은 분들이 예측한 것과는 다르게 나왔으리라 생각됩니다. 결국 리소스 해제 작업은 하나도 처리되지 않았다는 얘기인데 왜 이러한 현상이 발생하는지 정리해보도록 하죠.

* Container 객체는 Task 그룹인 tasks 내부에서 생성되었기 때문에 parallel\_invoke 함수는 tasks의 자식 Task 그룹으로 간주됩니다. 부모 Task 그룹인 tasks가 이미 취소되었기 때문에 parallel\_invoke 함수가 처리하는 작업은 실행되지 않아 \_resource1과 \_resource2는 해제되지 않습니다.
* 부모 Task 그룹이 취소된 이후에 자식 Task 그룹이 실행되면 작업을 수행하지 않기 위해 내부적으로 Exception을 발생시킵니다. Container 클래스의 소멸자 안에서는 이런 Exception을 처리하는 코드가 없으므로 Exception은 상위단으로 전파되고, 이에 따라 소멸자의 나머지 루틴을 처리하지 못하여 \_resource3는 해제되지 않습니다.
* 소멸자에서 예외가 발생되어 처리되지 않으면 프로그램이 undefined 상태가 됩니다.

위에서 설명된 원인에 의해 이 예제는 우리가 의도했던 바와는 다르게 동작하여 문제가 발생합니다. **만약 Task가 취소되지 않는다는것을 보장받을 수 있는 경우라도 리소스를 해제하는 등의 중요한 작업은 병렬 처리하지 않은 것이 좋으며 소멸자에서 Exception을 발생시키지 않도록 주의해야 합니다.**

**9. 병렬 작업을 취소하기 전에 블러킹(Blocking) 연산을 수행하지 말것**

**Cancellation 메카니즘을 이용하여 병렬 작업을 취소하는 경우, 가능한 cancel 메서드를 호출하기 이전에 블러킹 되는 연산을 수행하지 않는 것이 좋습니다.**만약 병렬 작업을 취소하기 전에 블러킹 되는 연산을 수행한다면 블러킹 연산이 cancel 메서드의 호출을 지연시키고, 이에 따라 다른 Task 들은 하지 않아도 될 작업을 수행하게 됩니다. 그러므로 병렬 작업이 취소되야할 조건이 만족되었다면 가능한 빨리 cancel 메서드를 호출해야 합니다.

이에 대한 예제로 배열에서 주어진 조건 함수를 만족하는 원소를 병렬로 검색하는 parallel\_find\_answer 함수를 작성해보도록 합시다. 이 함수는 조건 함수가 true를 리턴할 경우 Answer 객체를 생성하고 작업을 중단합니다.

|  |
| --- |
| #include <windows.h> #include <ppl.h>  using namespace concurrency;  // 검색 결과를 나타내는 클래스 template<typename T> class Answer { public:     explicit Answer(const T& data)         : \_data(data)     {     }      T get\_data() const     {         return \_data;     }      // TODO: 추가 메서드 구현  private:     T \_data;      // TODO: 추가 데이터 구현 };  // 배열에서 주어진 조건 함수를 만족하는 원소를 검색 template<typename T, class Predicate> Answer<T>\* parallel\_find\_answer(const T a[], size\_t count, const Predicate& pred) {     // 검색 결과     Answer<T>\* answer = nullptr;     // 동시에 조건 함수를 만족했을 경우의 중복 처리 방지     volatile long first\_result = 0;      // parallel\_for 함수를 이용하여 배열 원소를 검색     structured\_task\_group tasks;     tasks.run\_and\_wait([&]     {         // 람다 함수 안에서 T 타입을 인식할 수 있도록 alias 선언         typedef T T;          parallel\_for<size\_t>(0, count, [&](const T& n) {             **if (pred(a[n]) && InterlockedExchange(&first\_result, 1) == 0)**             {                 // 검색 결과를 담는 Answer 객체 생성                 **answer = new Answer<T>(a[n]);**                 // 나머지 작업 취소                 **tasks.cancel();**             }         });     });      return answer; } |

병렬 루프 안에서 조건 함수 pred를 호출하여 리턴값이 true인 경우(중복 처리를 방지하기 위해 InterlockedExchange 사용) Answer 객체를 생성하고 cancel 메서드를 호출하여 작업을 중단합니다. new 연산자는 힙 영역에 메모리를 할당하는데, 이는 블러킹될 수 있는 연산이고 작업을 취소한 이후에 처리되어도 상관 없기 때문에 다음과 같이 처리 순서를 바꾸면 성능 향상에 도움이 됩니다.

|  |
| --- |
| parallel\_for<size\_t>(0, count, [&](const T& n) {     if (pred(a[n]) && InterlockedExchange(&first\_result, 1) == 0)     {         // 나머지 작업 취소         **tasks.cancel();**         // 검색 결과를 담는 Answer 객체 생성         **answer = new Answer<T>(a[n]);**     } }); |

**10. 가능한 False Sharing 문제를 피해갈것**

**False Sharing 이란 멀티 프로세서 환경에서 같은 캐시 라인에 위치한 메모리를 Write 하려고 할때 발생하는 성능 저하 문제입니다.** 두 개의 Task 가 병렬로 처리되어 같은 캐시 라인에 위치한 서로 다른 두 변수를 Write 한다고 가정해봅시다. 두 변수는 서로 공유되어 있지 않음에도 불구하고 같은 캐시 라인에 위치해 있기 때문에 둘 중 하나의 변수를 Write 하더라도 두 변수가 참조하는 캐시 라인이 무효화되어 각각의 프로세서는 이를 다시 로드하는 작업을 수행해야 하고 이러한 작업이 빈번하게 반복되면 성능이 감소하게 됩니다.

먼저 parallel\_invoke 함수를 사용한 병렬 작업에서 공유 변수를 사용하여 정해진 횟수만큼 값을 증가시키는 예제를 만들어보겠습니다.

|  |
| --- |
| **volatile long count = 0L;** concurrency::parallel\_invoke(     [&count] {         for(int i = 0; i < 100000000; ++i)             **InterlockedIncrement(&count);**     },     [&count] {         for(int i = 0; i < 100000000; ++i)             **InterlockedIncrement(&count);**     } ); |

각각의 Task에서 공유 변수에 접근하여 값을 증가시키기 위해 동기화 작업을 수행하기 때문에 성능이 크게 감소됩니다. 이 문제를 해결하기 위해 아래와 같이 카운터 변수를 두 개로 분리하고 병렬 작업이 모두 끝난 다음 결과값을 더하는 방식으로 구현할 수 있습니다.

|  |
| --- |
| **long count1 = 0L; long count2 = 0L;** concurrency::parallel\_invoke(     [&count1] {         for(int i = 0; i < 100000000; ++i)             **++count1;**     },     [&count2] {         for(int i = 0; i < 100000000; ++i)             **++count2;**     } ); **long count = count1 + count2;** |

하지만 count1, count2 변수가 인접해 있어 같은 캐시 라인에 위치할 확률이 높기 때문에 False Sharing 문제가 남아있게 됩니다. **False Sharing 문제를 제거하기 위해서는 두 개의 카운터 변수를 서로 다른 캐시 라인에 위치시켜야 하는데 이는 count1, count2 변수를 64-byte(캐시의 크기가 64-byte 이하라고 가정) 단위로 정렬시킴으로서 구현할 수 있습니다.**

|  |
| --- |
| **\_\_declspec(align(64)) long count1 = 0L; \_\_declspec(align(64)) long count2 = 0L;** concurrency::parallel\_invoke(     [&count1] {         for(int i = 0; i < 100000000; ++i)             ++count1;     },     [&count2] {         for(int i = 0; i < 100000000; ++i)             ++count2;     } ); long count = count1 + count2; |

위 예제처럼 변수의 정렬 크기를 변경함으로서 False Sharing 문제를 피해갈 수 있지만 그보다 더 좋은 방법은 combinable 클래스를 이용하는 것입니다. **combinable 클래스는 공유 데이터의 동기화로 인한 성능 감소를 Thread-Local Storage 를 통해 해결하고 내부적으로는 각각의 로컬 변수간의 False Sharing 문제 발생을 최소화 하도록 구현되어 있습니다.**

|  |
| --- |
| **combinable<int> counts;** concurrency::parallel\_invoke(     [&counts] {         for(int i = 0; i < 100000000; ++i)             **++counts.local();**     },     [&counts] {         for(int i = 0; i < 100000000; ++i)             **++counts.local();**     } ); **long count = counts.combine(plus<int>());** |