**PPL 병렬 작업의 취소**

**PPL 에서는 병렬 작업을 수행하다가 임의의 상황에서 실행하던 작업을 취소할 수 있게 해주는 몇 가지 방법을 제공합니다.** 이번 강좌부터는 PPL이 제공하는 병렬 작업의 취소가 어떠한 방식으로 이루어지며 어떻게 동작하는지에 대해 알아보도록 하겠습니다. 일단 자세한 설명에 앞서 꼭 기억하고 있어야할 키포인트 몇 가지를 말씀드리고 넘어가도록 하죠.

* 병렬 작업의 취소는 즉시 이루어지지 않는다. 다시 얘기하자면 취소는 '취소 요청' 이지 '강제 취소' 가 아니다.
* 취소 작업은 취소를 요청하는 측과 취소 요청에 응답하는 측의 상호 협력적인 코드 작성으로 이루어진다.
* 병렬 작업의 취소는 크게 3가지 방식으로 구현이 가능하다.

세 번째 항목을 제외하고는 아직 무슨 얘기인지 이해가 잘 되지는 않으실지도 모르겠네요. 하지만 이어지는 강좌에서 예제 코드와 함께 설명을 들으시면 자연스럽게 이해가 되실거라 생각합니다. 위에서 언급한 사항은 중요한 개념이니 꼭 기억하시기 바라구요, 이제부터 본격적으로 어떤 방법으로 병렬 작업을 취소할 수 있는지에 대해서 알아보도록 하겠습니다.

**Cancellation Token**

첫 번째로 설명드릴 방법은 Cancellation Token을 이용하는 방법입니다. 이 방법은 PPL에서 병렬 작업을 취소하기 위한 가장 대표적인 방식이고 이것을 이용하여 Task와 Task 그룹의 작업을 취소할 수 있습니다. PPL에서 Cancellation Token은 다음과 같이 cancellation\_token\_source 클래스와 cancellation\_token 클래스를 통해 구현되어 있습니다.**Task 또는 Task 그룹은 cancellation\_token 을 구독(Subscribe) 하고 cancellation\_token\_source 클래스의 cancel 메서드를 호출하여 cancellation\_token 을 구독하는 모든 Task 또는 Task 그룹에게 취소를 요청하도록 합니다. 그리고 만약 작업이 아직 시작되지 않았다면 해당 작업은 실행되지 않습니다.**

서두에 설명한 키포인트중 두 번째 항목에서 '취소 작업은 취소를 요청하는 측과 취소 요청에 응답하는 측의 상호 협력적인 코드 작성으로 이루어진다' 라고 말씀 드렸었죠. cancellation\_token\_source 의 cancel 메서드 호출이 취소 요청이라면 취소 요청에 응답하는 코드는 어떻게 작성해야 할까요? 이는 Task 와 Task 그룹의 구현 방식이 약간 다르며 다음과 같이 구현할 수 있습니다.

* task 객체를 취소하려면 is\_task\_cancellation\_requested 함수로 취소 요청이 발생했는지 감시하고 cancel\_current\_task 함수를 호출하여 실행중인 작업을 취소합니다.
* task 그룹을 취소하려면 is\_current\_task\_group\_canceling 함수로 취소 요청이 발생했는지 감시하고 취소 요청 감지시 바로 task body가 종료되도록 구현하여 실행중인 작업을 취소합니다.

자...이제 Cancellation Token의 동작 방식이 이해가 되시나요? 아마도 이렇게 설명만 들어서는 아직 정확히 감이 오지는 않으실겁니다. 역시 백 번의 설명보다는 하나의 예제를 보는게 낫겠죠. 그럼 Cancellation Token을 이용하여 Task의 실행을 취소하는 간단한 예제를 작성해보도록 합시다.

|  |
| --- |
| #include <ppltasks.h> #include <iostream> #include <sstream>  using namespace concurrency; using namespace std;  // 작업 시뮬레이션 bool do\_work() {     wcout << L"Performing work..." << endl;     wait(250);     return true; }  // task\_status를 스트링으로 변환 std::wstring get\_status\_string(task\_status status) {     switch (status)     {     case not\_complete:         return L"not\_complete";      case completed:         return L"completed";      case canceled:         return L"canceled";     }     return L""; }  int wmain() {     **cancellation\_token\_source cts;     auto token = cts.get\_token();**      wcout << L"Creating task..." << endl;      // 취소되기 전까지 작업을 수행하는 task를 생성     auto t = create\_task([]     {         bool moreToDo = true;         while (moreToDo)         {             // 취소 요청 감시            **if (is\_task\_cancellation\_requested())**             {                 // TODO: 이곳에서 리소스 해제 작업 등을 수행                  // 현재 실행중인 작업을 취소                 **cancel\_current\_task();**             }             else             {                 // 작업 수행                 moreToDo = do\_work();             }         }     }**, token**);      // 1초 대기 후 취소 요청     wait(1000);      wcout << L"Canceling task..." << endl;     **cts.cancel();**      // 취소가 완료될 때까지 대기     wcout << L"Waiting for task to complete..." << endl;     **task\_status status = t.wait();**      wcout << L"Done." << endl;     wcout << L"Task status: " << get\_status\_string(status) << endl; }  // 실행결과  Creating task... Performing work... Performing work... Performing work... Performing work... Canceling task... Waiting for task to complete... Done. Task status: canceled |

메인 함수 시작 부분을 살펴보면 cancellation\_token\_source 객체를 생성하고 get\_token 메서드의 반환값인 cancellation\_token 객체를 create\_task 함수의 파라미터로 사용하여 Task를 생성합니다. 그러므로 해당 Task는 Cancellation Token을 구독하게 되며 Task 외부에서 cancel 메서드를 호출함으로서 Cancellation Token을 구독하는 Task에게 작업 취소를 요청합니다. 그리고 Task 내부를 살펴보면 루프를 돌때마다 is\_task\_cancellation\_requested 함수를 호출하여 취소 요청이 발생했는지 확인하고 취소 요청이 발생했을 경우 cancel\_current\_task 함수를 호출하여 실행중이던 작업을 취소하여 작업을 종료하게 됩니다.

**cancel\_current\_task 함수는 내부적으로 예외를 발생시키도록 구현되어있기 때문에 작업 함수를 종료하도록 코드를 작성하지 않아도(함수를 리턴시킨다던가, 루프를 빠져나오는 등의 코드) 예외 발생에 의하여 작업이 종료되며 Task 의 상태를 canceled 로 변경시킵니다.** 만약 cancel\_current\_task 함수를 호출하지 않고 작업 함수를 바로 리턴시키는 형태로 구현하였다면 작업의 취소는 이루어지지만 Task의 상태는 canceled 가 아닌 completed 로 되어버리기 때문에 올바른 방식이라고 할 수 없습니다.

**Task가 canceled 상태로 종료되었을 경우 task::wait 메서드는 canceled 를 리턴합니다. 하지만 task::get 메서드를 호출하는 경우에는 task\_canceled 예외를 발생시키므로 예외에 안전한 코드를 작성해야 합니다.** 또한 사용자 코드에서 임의로 task\_canceled 예외를 발생시키도록 구현하면 예기치 못한 상황이 발생할 수 있으니 주의하시기 바랍니다.

위의 예제에서는 is\_task\_cancellation\_requested 함수로 취소 요청을 감시하고 cancel\_current\_task 함수로 작업을 취소시켰지만 두 개의 함수 대신 다음과 같이 interruption\_point 함수를 사용하면 감시와 취소를 한 줄로 구현할 수도 있습니다.

|  |
| --- |
| auto t = create\_task([] {     bool moreToDo = true;     while (moreToDo)     {         // 취소 요청 감시 및 취소         **interruption\_point();**          // 작업 수행         moreToDo = do\_work();     } }, token); |

때때로 Cancellation Token 이 취소되는 시점에 리소스를 해제한다던가, UI를 갱신한다던가 하는 임의의 코드를 실행하고 싶은 경우가 생길 수 있는데 이런 경우 cancellation\_token::register\_callback 메서드를 이용하여 콜백 함수를 등록함으로서 구현할 수 있습니다.  아래 예제를 통해 작업 취소에 대한 콜백 함수를 등록하고 해제하는 방법을 살펴보도록 하죠.

|  |
| --- |
| #include <ppltasks.h> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     cancellation\_token\_source cts;     auto token = cts.get\_token();      // 이벤트 객체 생성     **event e;**      **cancellation\_token\_registration cookie;**     **cookie = token.register\_callback([&e, token, &cookie]()**     {         wcout << L"In cancellation callback..." << endl;         // 이벤트 객체 시그널         **e.set();**          // 불필요한 코드이지만 콜백 함수 등록을 해제하는 방법을 설명하기 위해 추가         **token.deregister\_callback(cookie);**     });      wcout << L"Creating task..." << endl;      // 이벤트 객체가 시그널되기를 기다리는 Task 생성     auto t = create\_task([&e]     {         **e.wait();**     }, token);      // 작업 취소 요청     wcout << L"Canceling task..." << endl;     **cts.cancel();**      // Task 종료 대기     t.wait();      wcout << L"Done." << endl; }  // 실행결과  Creating task... Canceling task... In cancellation callback... Done. |

먼저 이벤트 객체를 하나 생성하고 Task의 내부에서는 해당 이벤트 객체가 시그널 되기를 기다립니다. 그리고 cancellation\_token 객체의 register\_callback 메서드를 통해 콜백 함수로 등록한 람다 함수에서 이벤트 객체를 시그널 시켜줌으로서 Task 내부의 e.wait() 대기가 풀리게되고 Task 가 종료되어 t.wait() 종료 대기도 리턴되는 코드입니다. 코드의 흐름이 약간 정신없어 보이지만 차근 차근 살펴보면 다들 이해가 되실거라 생각합니다.