**병렬 알고리즘 취소**

parallel\_for 와 같은 병렬 알고리즘을 사용할 때 for 루프에서 break 를 사용하여 루프를 중단하고 빠져나오는 것처럼 자연스럽게 작업을 취소하려면 어떻게 해야할까요? 만약 우리가 병렬 작업의 취소 방법에 대해 모르고 있었다면 다음과 같은 방법으로 구현해 볼 수 있을겁니다.

|  |
| --- |
| #include <ppl.h> using namespace concurrency;  int wmain() {     // 취소 플래그     **bool canceled = false;**      parallel\_for(0, 100, [&](int i) {         // i 값이 50일 경우 취소 플래그 설정         if (i == 50)         {             **canceled = true;**         }          // 취소 플래그가 설정되지 않을 경우 작업 수행         **if (!canceled)**         {             // TODO: 작업 수행         }     }); } |

하지만 이런 방식으로 구현하면 취소 플래그 설정 이후 나머지 작업을 수행하지는 않겠지만 루프가 완전히 종료되는것은 아니기 때문에 쓸데없는 오버헤드가 발생하게 됩니다. 그럼 어떻게 해야 병렬 알고리즘을 자연스럽게 취소할 수 있을까요? 그 해답은 병렬 작업의 취소에 대한 지난 강좌에 모두 포함되어있고 이를 조금만 응용하면 쉽게 구현할 수 있습니다.

PPL에서 제공하는 병렬 알고리즘은 내부적으로 Task 그룹을 이용하여 구현되어 있습니다. 그러므로 Task 그룹을 취소했던 3가지 방식을 적용하면 병렬 알고리즘을 자연스럽게 취소할 수 있죠. 그럼 지금부터 3가지 방식을 어떻게 응용해서 구현해야 할지 설명하도록 하겠습니다.

**첫번째 방법은 Cancellation Token과 run\_with\_cancellation\_token 함수를 사용하는 방법입니다.**이 함수는 첫번째 파라미터로 실행할 작업을 전달받고 두번째 파라미터로 Cancellation Token을 전달받아 작업을 실행합니다. 구체적인 구현 방식은 바로 예제를 통해서 살펴보도록 하겠습니다.

|  |
| --- |
| #include <ppl.h> #include <iostream> #include <sstream>  using namespace concurrency; using namespace std;  int wmain() {     // Cancellation Token을 이용하여 run\_with\_cancellation\_token 호출     **cancellation\_token\_source cts;**     **run\_with\_cancellation\_token([&cts]()**     {         **parallel\_for(0, 20, [&cts](int n)**         {             // n 값이 11일 경우 작업 취소             if (n == 11)             {                 **cts.cancel();**             }             // 현재값 출력             else             {                 wstringstream ss;                 ss << n << endl;                 wcout << ss.str();             }         });     **}, cts.get\_token());** }  // 실행결과  18 3 9 0 19 10 1 5 |

run\_with\_cancellation\_token 함수가 실행할 작업에 parallel\_for 함수를 포함시키고, for 루프를 break 하여 빠져나오듯이 Cancellation Token의 cancel 메서드를 호출하여 병렬 루프의 작업을 취소하는 것을 볼 수 있습니다. run\_with\_cancellation\_token 함수는 내부적으로 structured\_task\_group 객체를 생성하여 run\_and\_wait 메서드를 호출하도록 구현되어있고, parallel\_for 함수 역시 그 내부에서 structured\_task\_group 객체를 생성하도록 구현되어있습니다. 그러므로 위 예제를 풀어서 적어보면 다음과 같은 구조로 표현될 수 있습니다.

|  |
| --- |
| cancellation\_token\_source cts;  // run\_with\_cancellation\_token **structured\_task\_group tg1(cts.get\_token());** **tg1.run\_and\_wait([]()** {     // parallel\_for     **structured\_task\_group tg2;**     for (int i = 0; i < 20; ++i)     {         tg2.run(...);     }     tg2.wait(); }); |

**Task 그룹의 취소에 있어서 중요하게 설명했던것 중에 하나는 바로 부모 Task 그룹이 취소되면 자식 Task 그룹도 취소된다는 점입니다.** 이 예제에서는 Cancallation Token의 cancel 메서드의 호출로 인해 Task 그룹 tg1, 즉 run\_with\_cancellation\_token 함수를 통해 만들어진 부모 Task가 취소되면 parallel\_for 함수에 해당하는 tg2 도 자동적으로 취소되어 나머지 작업을 수행하지 않게 되는것이죠. 물론 cancel 메서드를 호출하여 취소 요청을 하더라도 즉시 종료되지는 않고 이미 실행중이던 작업은 마저 수행하고 나서 종료됩니다.

**두번째 방법은 structured\_task\_group 클래스 객체와 run\_and\_wait 메서드를 직접 사용하는 방법입니다.** Cancellation Token을 명시적으로 사용하지 않기 때문에 Task 그룹의 cancel 메서드를 이용하여 병렬 루프의 작업을 취소하며 동작 원리는 위에서 설명한 것과 같습니다.

|  |
| --- |
| #include <ppl.h> #include <iostream> #include <sstream>  using namespace concurrency; using namespace std;  // task\_group\_status를 스트링으로 변환 std::wstring get\_status\_string(task\_group\_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() {     // Task 그룹을 생성하고 내부에서 parallel\_for 함수를 호출     **structured\_task\_group tg;**      **task\_group\_status status = tg.run\_and\_wait([&] {**         **parallel\_for(0, 100, [&](int i) {**             // i 값이 50일 경우 작업 취소             if (i == 50)             {                 **tg.cancel();**             }             else             {                 // TODO: 작업 수행             }         });     });      // Task 그룹 상태 출력     wcout << L"The task group status is: " << get\_status\_string(status) << endl; }  // 실행결과  The task group status is: canceled |

첫번째 예제와 마찬가지로 부모 Task 그룹에 해당하는 tg가 취소되면 parallel\_for 함수가 수행하는 작업도 취소되어 더이상 실행되지 않고 부모 Task 그룹은 canceled 상태를 출력하게 됩니다.

**세번째 방법은 예상하셨다시피 Exception을 이용한 방법입니다.** 이는 parallel\_for 함수를 try-catch 블럭으로 묶어줌으로서 다음과 같이 쉽게 구현이 가능합니다.

|  |
| --- |
| #include <ppl.h> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     **try     {         parallel\_for(0, 100, [&](int i) {**             // i 값이 50일 경우 Exception을 발생시켜 작업 취소             if (i == 50)             {                 **throw i;**             }             else             {                 // TODO: 작업 수행             }         });     **}     catch (int n)     {**         wcout << L"Caught " << n << endl;     **}** }  // 실행결과  Caught 50 |

Task 그룹에서 Exception을 이용한 작업 취소 역시 아직 시작되지 않은 작업을 더이상 실행하지 않기 때문에 parallel\_for 함수의 작업은 취소됩니다.

지금까지 병렬 알고리즘을 취소하는 세가지 방법에 대하여 알아보았습니다. 이 방법들은 그 마다의 장단점이 있기때문에 주어진 상황과 본인의 코딩 스타일에 맞춰서 한가지 방법을 선택하여 사용하시면 됩니다. 그리고 병렬 작업의 취소를 시기 적절하게 활용한다면 효율적인 코드 작성에 큰 도움이 되지만 어떤 경우에는 잘못된 동작을 야기할 수도 있습니다. **예를들어 어떤 Task가 다른 Task의 블러킹 상태를 깨워주거나 할당된 리소스를 해제하는 등의 중요한 역할을 담당할 경우 작업이 취소되어 해당 작업이 실행되지 않으면 데드락, 리소스 누수 등의 심각한 상황이 발생할 수 있으니 주의하시기 바랍니다.**

자, 이제 병렬 작업의 취소에 대한 강좌를 마지막으로 PPL에 대한 전반적인 모든 기능들을 살펴보았습니다. 다음 강좌부터는 지금까지 배운 PPL을 어떤 상황에서 활용해야 할지, 사용시 어떤 점을 주의해서 코드를 작성해야 할지 등의 내용을 정리하여 'PPL 활용 십계명' 이라는 제목으로 찾아 뵙도록 하겠습니다.