**중첩된 task**

지금까지 task의 then 메서드를 통한 task 연결 방법에 대하여 살펴보았습니다. 부모 task와 자식 task를 구성하는 방법, 즉 어떤 task를 먼저 실행하고 task가 종료되면 다음 task를 실행하는 방법에는 then 메서드를 이용한 방법 외에 또 한 가지 방법이 있습니다. **바로 중첩된 task(nested task)을 이용한 방법입니다.** 중첩된 task란 task 내부에서(outer task) 또 다른 task를 생성하여(inner task) 생성한 task를 리턴 하는 방식으로 **inner task는 outer task가 종료되면 실행**됩니다. 나머지는 예제를 통해서 자세히 알아보시죠.

|  |
| --- |
| #include <ppltasks.h> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     auto t = create\_task([]()     {         wcout << L"Task A" << endl;          // Task A 내부에서 Task B를 생성하여 리턴합니다.         **return create\_task([]() -> double**         {             wcout << L"Task B" << endl;             **return 0.2f;**         });     });      // Task C를 then으로 Task A에 연결합니다.     **t.then([](double d)**     {         wcout << L"Task C: " << d << endl;     }).wait(); }  // 실행결과  Task A Task B Task C: 0.2 |

 이번엔 이해를 돕기 위해 MSDN 예제를 조금 변형해봤습니다. 예제를 보시면 Task A를 생성하고 그 내부에서 Task B를 생성하여 바로 리턴 하는 코드가 작성이 되어 있습니다. 방금 설명 드린 중첩된 task에 의해**Task B는 Task A가 종료된 이후 실행됨을 보장받을 수 있죠.**그런데 그 아래 코드를 보면 Task A에는 중첩된 task인 Task B 뿐만 아니라 then으로 연결된 Task C가 존재하는군요. 그렇다면 Task A, B, C의 실행 순서는 어떻게 되는 것일까요? 실행 결과를 미리 보여드렸기 때문에 눈치를 채셨겠지만**task의 실행 순서는 Task A -> Task B -> Task C의 순서로 실행됩니다.** 다시 한 번 말씀드리지만 Task C는 Task B가 종료된 이후에 실행되는 것입니다. 고로 **'중첩된 task는 then으로 연결된 task보다 실행 우선순위가 높다'** 라고 생각하시면 됩니다. 이해가 되셨나요?

그럼 **task 연결 시 then 메서드를 이용하는 방법과 중첩된 task를 리턴 하는 방법의 차이점**에 대해 이야기해보겠습니다. Task B를 생성하는 코드를 보시면 아시겠지만 중첩된 task는 inner task가 outer task의 리턴 타입을 함수 파라미터로 사용하지 않습니다. **결국 inner task의 함수 파라미터는 항상 void 타입이라는 이야기가 됩니다.**그리고 또 한 가지 흥미로운 점은 **Task C의 함수 파라미터가 double 타입 이라는 점입니다.**Task C는 Task A 객체에 then으로 연결되었기 때문에 Task A의 리턴 타입을 함수 파라미터로 사용해야 할 것 같지만 실제로는 task의 실행 순서가 Task A -> Task B -> Task C 순서이기 때문에 **Task C는 중첩된 task인 Task B의 리턴 타입을 파라미터로 받게된다.**

**when\_all과 when\_any**

이번엔 task의 연결을 조금 더 업그레이드된 시나리오로부터 접근해보도록 하죠. 예를 들어 서로 종속성이 없는 여러 개의 task를 병렬로 실행하고 모든 task의 실행이 종료되면 또 다른 task를 실행하려면 어떻게 해야할까요? 또는 종속성이 없는 여러 개의 task를 병렬로 실행하고 하나의 task라도 먼저 실행이 종료 되면 또 다른 task를 실행하려면 어떻게 해야 할까요? 다들 눈치 채셨겠지만 이 질문에 대한 해답은 바로 위에 적어놓은 when\_all 함수와 when\_any 함수입니다.

**when\_all 함수는 컨테이너에 담긴 모든 task의 실행이 종료되기를 기다리며 모든 task가 종료되었을 경우 task<vector<T>> 타입의 task 객체를 리턴하는 함수입니다.** 결국 when\_all 함수는 task의 join을 구현한 함수라 할 수 있습니다. 아래 예제에서는 when\_all 함수를 이용하여 임의의 정수값을 리턴하는 3개의 task를 병렬로 실행하고 이 모든 task의 실행 종료를 기다려 이 값들의 합을 구하는 방법을 설명하도록 하겠습니다.

|  |
| --- |
| // 여러개의 task를 병렬로 실행 array<task<int>, 3> tasks = {     create\_task([]() -> int { return 88; }),     create\_task([]() -> int { return 42; }),     create\_task([]() -> int { return 99; }) };  // when\_all 함수를 이용하여 3개의 task의 실행 종료를 대기하고 // 모든 task가 종료되면 그 결과값들의 합을 구합니다. **auto joinTask = when\_all(begin(tasks), end(tasks)).then([](vector<int> results)** {     wcout << L"The sum is "               << accumulate(begin(results), end(results), 0)               << L'.' << endl; });  wcout << L"Hello from the joining thread." << endl;  // 모든 태스크가 종료될 때까지 대기 joinTask.wait();    // 실행결과  Hello from the joining thread. The sum is 229. |

**when\_all 함수를 사용하여 여러 task의 종료를 대기하려면 대기하려는 task의 리턴 타입은 모두 동일한 타입 이어야 합니다.**이 예제에서는 모두 int 타입을 리턴하는 task를 생성하고 있습니다. 그리고 이 예제에서 when\_all 함수의 리턴값은 task<vector<int>> 타입이며 vector 안에는 3개의 task의 리턴값인 88, 42, 99가 담기게 됩니다. when\_all 함수의 리턴값을 이용하여 then 메서드를 호출 하였으므로 vector<int> 타입의 파라미터가 사용되었으며 task 기반의 연결 방식으로 task<vector<int>> 타입을 사용할 수도 있습니다. 그리고 아래와 같이 && 연산자를 이용하면 좀 더 직관적인 코드로 when\_all 함수를 호출한 것과 동일한 결과를 얻을 수 있습니다.

|  |
| --- |
| auto t = t1 && t2; // when\_all 호출과 동일 |

**when\_any 함수는 컨테이너에 담긴 task 중 하나의 task라도 먼저 종료되면 가장 먼저 종료된 task의 리턴값과 컨테이너상의 인덱스를 담고있는 task<pair<T, size\_t>> 타입을 리턴하는 함수입니다.** when\_any 함수는 task의 select를 구현한 함수라 할 수 있으며, 여러 가지 방식으로 구현된 함수들 중 가장 빨리 처리된 함수를 선택하여 사용하는 방식 등에 응용될 수 있습니다. 아래 예제에서는 when\_any 함수를 이용하여 임의의 정수값을 리턴하는 3개의 task 병렬로 실행하고 이 task 중 가장 먼저 처리된 task의 결과값을 출력하는 방법을 설명하도록 하겠습니다.

|  |
| --- |
| #include <ppltasks.h> #include <array> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     // Start multiple tasks.     array<task<int>, 3> tasks = {         create\_task([]() -> int { return 88; }),         create\_task([]() -> int { return 42; }),         create\_task([]() -> int { return 99; })     };      // when\_any 함수를 이용하여 3개의 task중 가장 먼저 처리되는 task의 결과값을 출력합니다.    **when\_any(begin(tasks), end(tasks)).then([](pair<int, size\_t> result)**     {         wcout << "First task to finish returns "                   << result.first                   << L" and has index "                   << result.second                   << L'.' << endl;     }).wait(); }  // 실행결과  First task to finish returns 42 and has index 1. |

when\_all 함수와 마찬가지로 when\_any 함수가 select 하려는 task의 리턴 타입은 모두 동일한 타입이어야 합니다. **하지만 when\_all 함수와는 다르게 when\_any 함수의 리턴값은 task<pair<int, size\_t>> 타입이고 pair의 first 에는 가장 먼저 처리된 task의 리턴값이 담기게 되며 second에는 이 task의 컨테이너상의 index가 담기게 됩니다.** 위에서 출력값을 42와 인덱스 1을 표시하였지만 병렬 처리의 특성상 어떤 task가 가장 먼저 처리될지 판단할 수 없기 때문에 위 예제의 출력값은 실행할 때 마다 달라질 수 있습니다. when\_any 함수의 리턴값으로 then 메서드를 호출할 경우  task 기반의 연결 방식으로 task<pair<int, size\_t> 타입을 사용할 수도 있습니다. 그리고 when\_all 함수의 경우 처럼 || 연산자를 이용하면 좀 더 직관적인 코드로 when\_any 함수를 호출한 것과 동일한 결과를 얻을 수 있습니다.

|  |
| --- |
| auto t = t1 || t2; // when\_any 호출과 동일 |

**Lambda Expression 사용시 주의 사항**

일반적으로 task가 실행할 작업을 구현할 때 람다 함수를 주로 사용합니다. 람다 함수는 간결한 코드 작성을 가능하게 해주고 가독성이 좋게 만들어주지만 자칫 잘못하면 잘못된 코드를 작성하게 될 수도 있습니다. 그러므로 람다 함수를 이용하여 task를 구현할 경우에는 다음과 같은 사항을 주의해야 합니다.

* task는 백그라운드 스레드에서 병렬적으로 실행되기 때문에 람다 함수에서 캡쳐한 변수들의 사용에 유의해야 합니다. 값으로 캡쳐할 경우([=])에는 변수의 복사본이 사용되어 문제가 없지만 참조로 캡쳐할 경우([&]) 캡쳐한 변수는 반드시 task 실행 종료 시점보다 수명이 더 길어야 합니다.
* 앞서 설명한 항목과 같은 맥락으로 스택에 할당된 지역 변수는 캡쳐하지 말아야 합니다. 지역 변수로 선언된 객체의 멤버 변수 또한 마찬가지 입니다.
* [=] 또는 [&]로 모든 변수를 캡쳐하는것은 삼가하고, [v1] 또는 [&v1]와 같이 캡쳐할 변수와 캡쳐 방식을 명시적으로 적어줌으로서 좀 더 안전한 코드를 작성할 수 있습니다.

만약 여러 개의 연결된 task에서 변수의 값을 사용하고 변경하는 코드를 작성하려고 한다면 값을 변경해야 하기 때문에 참조 방식으로 캡쳐할 수 밖에 없고 위에서 언급한 문제가 발생하게 됩니다. 이런 경우에는 람다 함수에서 캡쳐할 변수를 shared\_ptr로 감싸서 스마트 포인터로 전달하는 방식으로 해결할 수 있습니다. 아래에서는 이에 대한 예제를 보여줍니다.

|  |
| --- |
| #include <ppltasks.h> #include <iostream> #include <string>  using namespace concurrency; using namespace std;  task<wstring> write\_to\_string() {     // shared\_ptr을 생성하여 변수 s의 수명이 변수 s를 사용하는     // 모든 task의 종료 시 까지 유지됩니다.     **auto s = make\_shared<wstring>(L"Value 1");**      return create\_task([s]     {         // Print the current value.         wcout << L"Current value: " << \*s << endl;         // 새로운 값을 할당         \*s = L"Value 2";      }).then([s]     {         // 현재 값 출력         wcout << L"Current value: " << \*s << endl;         // 새로운 값을 할당하고 리턴         \*s = L"Value 3";         return \*s;     }); }  int wmain() {     // 문자열을 사용하는 task 연결 체인을 생성     auto t = write\_to\_string();      // 작업 종료 대기 후 결과 값 출력     wcout << L"Final value: " << t.get() << endl; } |