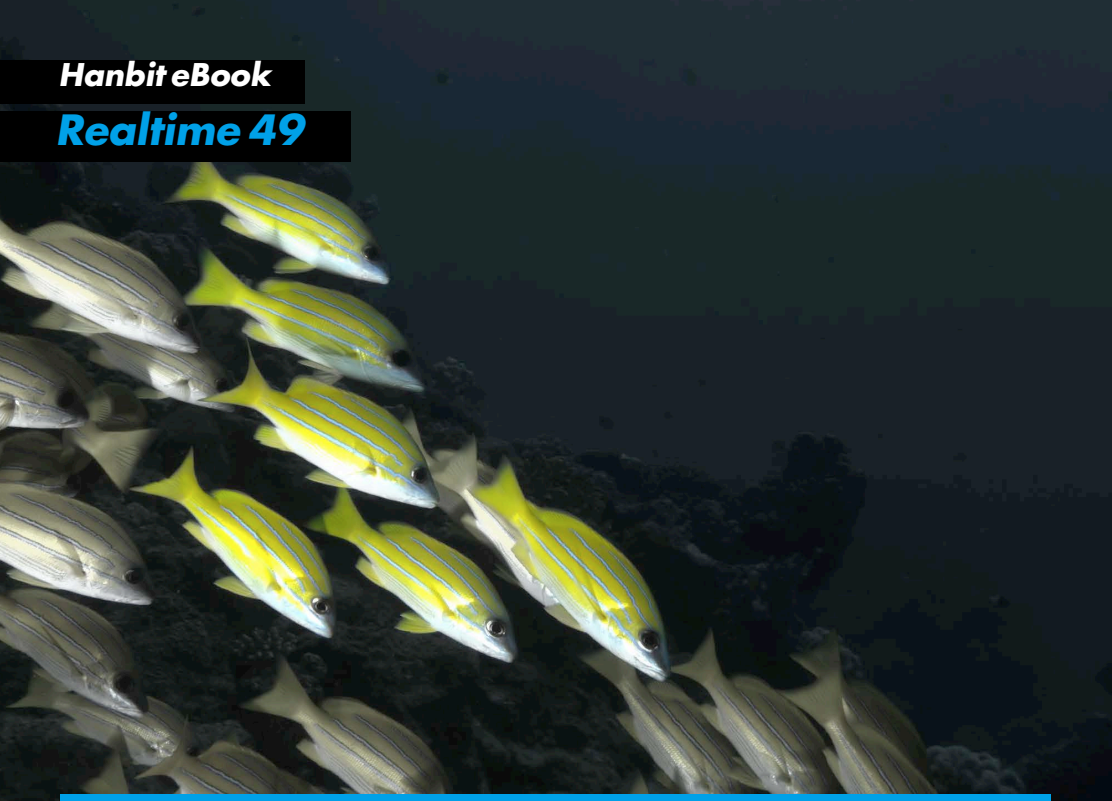
Hanbit eBook
Realtime 49



Thinking About

PPL을 이용한 VC++ 병렬 프로그래밍

김경진 지음

한빛미디어 Hanbit Media, Inc.

Thinking About:

PPL을 이용한 VC++ 병렬 프로그래밍

지은이_ **김경진**

새로운 것을 만들기 좋아하고 도전을 즐기는 대한민국의 평범한 개발자다. 아직은 부족한 점이 많지만, 소프트웨어 개발 분야의 장인이 되기 위해 열심히 노력 중이다. 현재 Microsoft Visual C++ MVP로 활동하면서, 본인이 가진 지식을 보다 많은 개발자들과 공유하기 위해 힘쓰고 있다.

- 이메일: devmachine@naver.com
- 블로그: http://devmachine.blog.me
- 트위터: @DevMachine

Thinking About: **PPL을 이용한 VC++ 병렬 프로그래밍**

초판발행 2013년 11월 29일
2판발행 2014년 03월 17일

지은이 김경진 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-654-8 15000 / **정가** 12,000원

책임편집 배용석 / **기획** 김병희 / **편집** 안선화
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 김현미
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 서은옥, 김옥현

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

저자 서문

이제는 스마트폰 같은 휴대용 기기에도 멀티 코어가 기본으로 탑재되는 바야흐로 '멀티 코어 시대'다. 하지만 이러한 하드웨어를 제대로 활용하는 소프트웨어가 과연 얼마나 될까? 멀티 코어를 효율적으로 활용하려면 병렬 프로그래밍이 필수지만, 소프트웨어 개발자들에게 병렬 프로그래밍은 여전히 어렵기만 하다.

필자 또한 여느 소프트웨어 개발자들과 마찬가지로 병렬 프로그래밍 때문에 골머리 썩었던 게 한두 번이 아니다. 병렬 프로그래밍은 도저히 친해지려야 친해질 수 없는 존재와도 같았다. PPL(Parallel Patterns Library)을 접하기 전까지는 말이다.

PPL은 Visual C++ 환경에서 병렬 프로그래밍을 도와주는 획기적인 도구다. 이 도구를 이용하면 병렬 프로그래밍을 수행하기 위해 더 이상 복잡한 코드를 작성할 필요가 없으며, 간단히 코드 몇 줄 수정하는 것만으로 기존 프로그램의 성능을 대폭 향상시킬 수도 있다. PPL은 시대에 뒤처지지 않기 위해 억지로 배워야 할 신기술이 아니라 개발자가 좀 더 편하게 일하도록 도와주는 든든한 지원군이다. 그러니 새로운 것을 배워야 한다는 부담감은 잠시 내려놓고 가벼운 마음으로 이 책을 읽어보기 바란다.

PPL에 관한 경험을 조금 더 많은 개발자들과 공유하고 싶어서 블로그에 연재하기 시작했던 강좌가 이렇게 책으로까지 출간되니 감회가 새롭다. 필자는 '어떻게 하면 독자들이 PPL을 쉽게 이해하고 편하게 사용할 수 있을까?'라는 고민을 수도 없이 거듭하면서 이 책을 집필했는데, 이러한 필자의 노력이 독자들에게 조금이나마 전달되었으면 하는 바람이다.

마지막으로 이 책이 출간될 수 있도록 도와주신 한빛미디어 관계자 여러분께 감사의 마음을 전한다. 특히, 첫 집필이라 부족한 점이 많음에도 불구하고 칭찬과 격려를 아끼지 않으신 편집자 김병희 님께 진심으로 감사하다는 말을 전하고 싶다.

아울러 집필하는 동안 많은 시간을 함께 보내지 못했지만 언제나 뒤에서 묵묵히 응원해준 아내와 곧 태어날 나의 소중한 아들에게 고맙고 사랑한다는 말을 꼭 전하고 싶다.

집필을 마치며
지은이 김경진

대상 독자 및 예제 파일

초급 **초중급** 중급 중고급 고급

이 책은 Parallel Patterns Library(PPL)를 이용하여 누구나 쉽게 병렬 프로그래밍을 구현하는 것을 목표로 한다. 내용의 신뢰성을 높이고자 MSDN 문서를 참조하였으며, 본문 내용과 예제는 MSDN 문서를 기반으로 집필했다. 전체적인 내용은 예제 위주로 진행되며, 주요 요소에 그림을 첨부하여 독자들이 쉽게 이해할 수 있도록 작성했다.

이 책의 전반적인 내용은 C++ 초중급 이상의 개발자라면 누구나 이해할 수 있는 수준이다. 하지만 STL과 C++11 표준(auto 키워드, 람다 표현식)에 대한 사전 지식이 없다면 다소 어려울 수 있으니 먼저 알아두는 것이 좋다. 지금까지 PPL에 관련된 국내 자료가 전혀 없었는데, 이 책을 통해 많은 개발자가 병렬 프로그래밍에 친숙해졌으면 하는 바람이다. 이 책은 다음과 같은 독자들을 대상으로 한다.

- 윈도우 응용 프로그램 개발자
- 윈도우 서버 개발자 및 게임 개발자
- 이미지/영상 처리 애플리케이션 개발자
- 제품의 성능 개선을 목표로 하는 모든 Visual C++ 개발자

이 책에 포함된 소스 코드는 다음 주소에서 다운받을 수 있다.

- http://www.hanbit.co.kr/exam/2654

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. **eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.**

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. **무료로 업데이트되는, 전자책 전용 서비스입니다.**

   종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

INTRO

멀티 코어 시대, 병렬 프로그래밍합시다

■ CPU의 발전과 멀티 코어

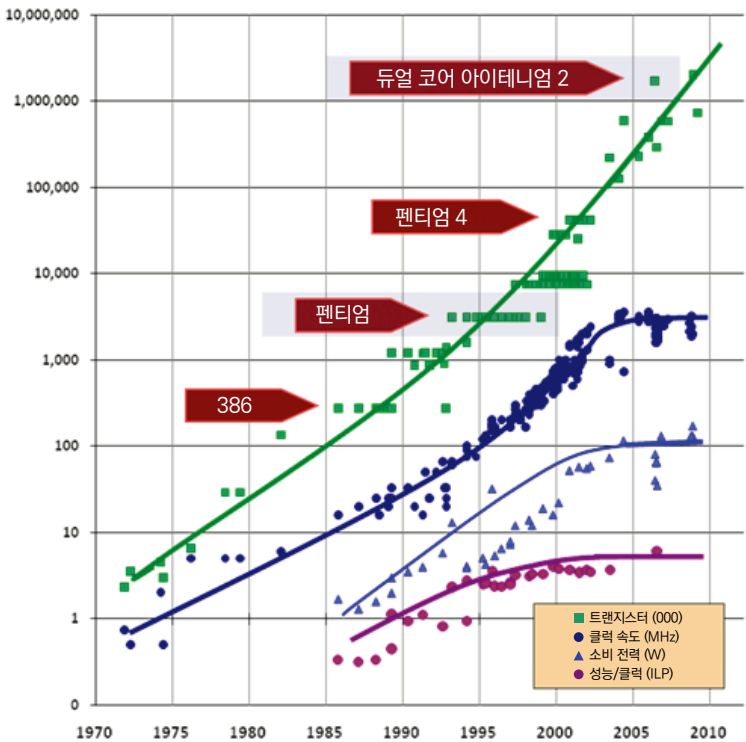
CPU의 역사는 세계 최초의 상용 마이크로프로세서가 출시된 1971년으로 거슬러 올라간다. 인텔은 1971년 11월 15일에 '인텔 4004'라는 이름으로 4비트 CPU를 출시했는데, 약 2,300개의 트랜지스터가 사용된 것으로 최고 클럭 속도가 740kHz에 불과했다. 이후 인텔은 반도체 집적 회로의 성능이 18개월마다 2배로 증가한다는 '무어의 법칙Moore's Law'을 꾸준히 실현하면서 프로세서의 성능을 기하급수적으로 발전시켜나갔다.

90년대부터 2000년대 초반까지는 CPU 제조사들이 클럭 속도를 조금이라도 올리기 위해 치열하게 경쟁하던 시대였다. CPU의 클럭 속도를 올리면 컴퓨터의 처리 속도 역시 그에 비례하여 빨라지기 때문에 싱글 코어 환경에서는 클럭 속도가 곧 컴퓨터의 성능을 가늠하는 척도였다. 이러한 시대적 상황을 대변하듯 당시에는 오버클럭이 꽤나 유행했고, 필자 역시 CPU의 클럭 속도를 조금이라도 더 끌어올리기 위해 고가의 쿨러까지 동원하며 오버클럭에 매진하던 시절이 있었다.

이후에도 많은 사람이 CPU의 클럭 속도가 계속 올라갈 것이라 예상했지만, 실제로 이러한 흐름은 그리 오래 지속되지 못했다. 이론적으로 CPU의 클럭 속도를 2배 올리려면 8배가량의 전력이 더 필요한데, 소비 전력의 증가는 발열과 직결된다. 이와 같은 문제는 90년대 초반까지만 해도 무시할 수 있는 수준이었다. 하지만 시간이 지날수록 고전력, 고발열 문제가 점점 더 심각해졌고 펜티엄 4에 이르러서는 더 이상 해결하기 어려운 지경에 도달했다. 무한정 치솟을 것만 같았던 클럭 속도 증가에 제동이 걸린 것이다.

결국 CPU 제조사들은 2005년을 기점으로 클럭 속도를 올리는 대신 코어 개수를 늘리는 쪽으로 방향을 선회했다. 비로소 싱글 코어 시대를 마감하고 멀티 코어의 시대가 열린 것이다.

그림 I-1 인텔 CPU의 발전 동향



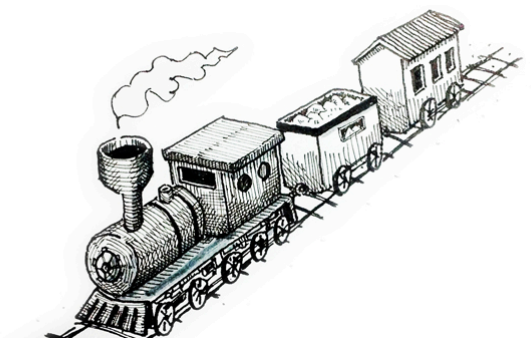
프로세서의 집적도는 아직 무어의 법칙을 따라 꾸준히 증가하고 있으며, 집적도의 증가는 전력 효율이 높은 여러 개의 코어를 제한된 공간 안에 배치하는 데 활용되고 있다. 최근에는 스마트폰 같은 휴대용 기기의 사용이 늘어나면서 저전력, 저발열의 중요성이 다시 한 번 강조되는 추세다. 이러한 흐름이라면 멀티 코어의 시대는 꽤나 오랫동안 유지될 것으로 예상되며, 싱글 코어 시대로 다시 되돌아가는 것

은 사실상 불가능해 보인다. 그러므로 멀티 코어 시대에 발맞춰 소프트웨어 개발의 패러다임도 함께 변화해야 할 것이다.

■ 병렬 프로그래밍은 선택이 아닌 필수

하나의 선로에서 목적지를 왕복하며 석탄을 실어 나르는 수송 열차가 있다고 가정해보자. 만약 정해진 양의 석탄을 조금 더 빠른 시간 안에 목적지까지 운송하려면 어떻게 해야 할까?

그림 I-2 하나의 선로를 이용한 석탄 운송 작업



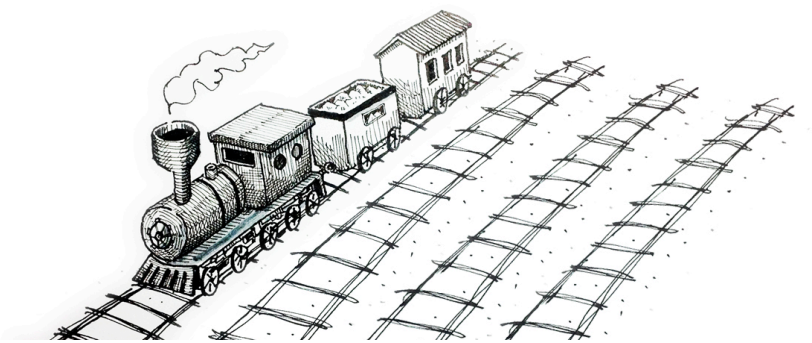
이 문제에 대한 해답은 생각보다 단순하다. 바로 수송 열차의 운행 속도를 올리는 것이다. 열차의 운행 속도가 올라가면 석탄 운송 작업의 속도 역시 자연스럽게 증가하며, 같은 양의 석탄을 운송하기 위해 소요되는 시간은 그만큼 단축될 것이다.

하나의 선로를 사용하는 석탄 운송 작업은 싱글 코어 CPU에서 동작하는 소프트웨어에 비유할 수 있다. 열차의 운행 속도가 곧 CPU 클럭 속도를 의미하며, 석탄 운송 작업은 소프트웨어가 처리하는 작업을 의미한다. 그러므로 위와 같은 규칙을 적용했을 때, CPU 클럭 속도가 올라가면 소프트웨어의 처리 속도 역시 그에 비례하여 증가한다는 결론을 얻을 수 있다. 실제로 클럭 속도가 꾸준히 증가하던 싱글 코어 시대에는 소프트웨어 개발자들이 별다른 노력을 하지 않고도 소프트웨어의 성

능이 저절로 향상되는 혜택을 얻을 수 있었다.

하지만 오늘날과 같은 멀티 코어 시대에는 소프트웨어가 하드웨어의 발전에 편승하는 게 더는 어려워졌다. 멀티 코어 CPU에서 동작하는 소프트웨어는 마치 여러 개의 선로를 사용하는 석탄 운송 작업과 같은데, 선로가 여러 개라고 해서 저절로 여러 개의 수송 열차가 운용되는 것은 아니다. 멀티 코어를 고려하지 않고 기존 방식대로 소프트웨어를 개발한다면 아래 그림과 같이 여러 개의 선로를 두고도 하나의 선로밖에 활용하지 못하는 반쪽짜리 소프트웨어가 될 수밖에 없다.

그림 I-3 여러 개의 선로 중에 하나의 선로밖에 활용하지 못하여 선로가 낭비되는 모습



여러 개의 선로, 즉 여러 개의 코어를 낭비하지 않고 효율적으로 활용하려면 여러 개의 작업을 동시에 처리하는 '병렬 프로그래밍'을 도입해야 한다. 이러한 변화를 두고 『Exceptional C++』의 저자 허브 셔터Herb Sutter는 "공짜 점심의 시대는 끝났다"라는 명언을 남겼는데, 이는 멀티 코어 시대의 흐름에 맞춰 소프트웨어 개발자도 변화해야 한다는 점을 역설한 말이다. 멀티 코어 CPU를 제대로 활용하기 위한 유일한 방법이 바로 병렬 프로그래밍이며, 이는 앞으로 소프트웨어 개발자들의 필수 지식이 될 것이다.

병렬 프로그래밍 도구

병렬 프로그래밍이란 하나의 프로그램에서 여러 개의 작업을 동시에 처리하도록 구현하는 프로그래밍 방법을 말한다. 병렬 프로그래밍을 잘 활용하면 소프트웨어의 성능을 향상시킬 수 있다는 사실을 누구나 잘 알고 있지만, 순차적인 프로그래밍 방식에 비해 어렵다는 단점 때문에 실제로는 쉽사리 적용하지 못하는 경우가 많다. 인간의 두뇌는 아직까지 동시에 여러 개의 작업을 처리하는 데 익숙하지 않은 모양이다.

병렬 프로그래밍의 가장 대표적인 방식은 '멀티 스레드 프로그래밍'이며, 개발자라면 누구나 한 번쯤은 스레드를 생성하여 작업을 병렬 처리하도록 구현해본 경험이 있을 것이다. 하지만 스레드를 직접 다루어 병렬 프로그래밍을 구현하는 것은 생각보다 쉬운 일이 아니다. 각각의 스레드에서 처리할 작업을 효율적으로 분배하는 일부터 공유 자원 접근으로 인한 동기화와 데드락 등의 여러 가지 문제가 우리의 발목을 잡는다. 멀티 스레드를 이용한 프로그램을 버그 없이 안정적으로 구현하는 것도 물론 어렵지만, CPU 코어 개수가 늘어날 때마다 성능이 향상되도록 확장성 있게 구현하는 것은 더더욱 어렵다.

■ 병렬 프로그래밍 언어와 라이브러리

이렇게 어렵기만 한 병렬 프로그래밍은 소프트웨어 개발자들이 풀어야 할 영원한 숙제인 것일까? 다행히도 병렬 프로그래밍을 쉽게 구현하는 방법은 이미 오랫동안 연구되어 왔고, 최근 들어서는 병렬 프로그래밍을 돕는 도구가 점점 늘어나고 있다. 컴파일러 또는 언어적인 차원에서 지원되는 것은 OpenMP, CilkPlus, X10, Erlang, Haskell 등이 대표적이며, 라이브러리 차원에서 지원되는 것은 TBB[Threading Building Blocks], TPL[Task Parallel Library] 등이 있다. 이와 같이 다양한 형태의 도구들을 활용하면 이전보다 훨씬 더 쉽게 병렬 프로그래밍을 구현할 수 있을 것이다.

■ Parallel Patterns Library(PPL)

PPL[Parallel Patterns Library, 병렬 패턴 라이브러리]은 마이크로소프트사에서 제작한 병렬 프로그래밍 라이브러리로 Visual C++ 환경에서 사용한다. 인텔 TBB를 모티브 삼아 만들었기 때문에 인터페이스가 상당히 유사하며, 기능은 대부분 TBB와 호환한다. 사실 PPL이 처음 제공되기 시작한 Visual Studio 2010 버전 때만 해도 PPL의 모든 기능은 TBB에 이미 포함된 것들이었다. 하지만 Visual Studio 2012 버전부터는 PPL에서 제공하는 기능이 대폭 늘어나, TBB와는 독자적으로 발전해나가고 있다.

PPL은 C++ 템플릿 라이브러리 형태로 제공되며, STL[Standard Template Library]과도 많이 닮았다. STL과 유사한 스타일의 제네릭 알고리즘과 컨테이너를 제공하며, 기존의 STL 알고리즘 및 컨테이너와도 잘 호환한다. 병렬 프로그래밍이 아직 익숙하지 않은 개발자라 하더라도 STL을 사용해본 경험이 있다면 PPL 역시 쉽게 사용할 수 있을 것이다.

■ 왜 PPL인가?

이 책을 구독하는 독자들은 대부분 Visual C++ 환경에서 애플리케이션이나 게임 등을 개발하는 개발자일 것이다. Visual C++ 환경에서 병렬 프로그래밍을 위해 선택할 수 있는 도구는 OpenMP, TBB, PPL 등이 있는데, 어떤 도구를 선택하는 것이 가장 적합할까?

OpenMP는 컴파일러 지시자 형태로 제공되는 병렬 프로그래밍 도구로 C, C++, 포트란 언어에서 사용할 수 있다. 멀티 플랫폼을 지원한다는 것이 큰 장점이지만 아무래도 각 플랫폼에 최적화된 라이브러리를 사용하는 것보다는 성능이 다소 떨어진다. 그리고 C++ 라이브러리가 아닌 컴파일러 지시자 형태로 제공되기 때문에, 이에 대한 문법을 새로 익혀야 한다는 부담과 가독성이 떨어진다는 단점도 있다. 반면에 PPL은 C++ 템플릿 라이브러리 형태로 제공되므로 C++ 개발자

라면 누구나 친숙하게 사용할 수 있고, C++11 표준인 auto 키워드 및 람다 표현식과 함께 사용하면 아주 간결한 코드를 작성할 수 있다. 물론 필요에 따라 PPL과 OpenMP를 혼용하여 사용하는 것도 가능하다.

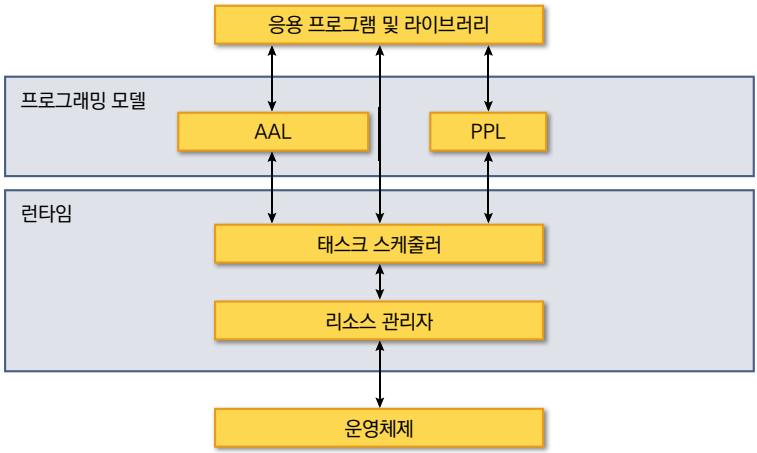
TBB는 병렬 프로그래밍을 돕기 위해 인텔에서 제작한 C++ 템플릿 라이브러리로, 멀티 플랫폼을 지원하며 현시점에서는 PPL보다 조금 더 다양한 기능을 제공한다. 하지만 직접 라이브러리를 설치하여 개발 환경을 구축해야 하는 불편함이 있으며, 가장 치명적인 단점은 바로 라이선스 문제이다. TBB의 오픈 소스 버전은 GPLv2 라이선스를 따르기 때문에 TBB를 이용하여 상용 소프트웨어를 개발하려면 상용 버전 라이선스를 별도로 구매해야 한다. 반면에 PPL은 Visual Studio 2010 버전 이상부터 기본적으로 내장되어 있기 때문에 별도의 라이브러리를 설치하는 수고 없이 간편하게 사용할 수 있다.

물론 병렬 프로그래밍 도구들은 각각이 서로 다른 장단점을 가지고 있지만 위와 같은 내용을 종합한 결과, Visual C++ 환경에서는 PPL을 사용하는 것이 가장 적합하다는 결론을 내릴 수 있다. PPL은 정체된 기술이 아니기 때문에 앞으로도 꾸준히 진화할 것으로 예상되며, Visual C++ 개발자에게 더 편리한 기능과 향상된 성능을 제공해줄 것이라 기대해본다.

동시성 런타임

동시성 런타임Concurrency Runtime은 동시적 프로그래밍을 위한 여러 가지 도구를 묶어 놓은 프레임워크다. PPL은 동시성 런타임에 포함되는 4가지 구성 요소 중 하나며, 상위 계층인 프로그래밍 모델에 위치하여 응용 프로그램 및 라이브러리와 상호작용한다.

그림 I-4 동시성 런타임의 구조



하위 계층인 런타임에는 리소스 관리자와 태스크 스케줄러가 위치해 있으며, 이들은 운영 체제와 프로그래밍 모델 사이에서 동시에 실행되는 작업들을 효율적으로 처리하도록 도와준다. 이 중 리소스 관리자는 프로세서와 메모리 같은 연산 자원을 추상화하여 관리하며, 태스크task 스케줄러는 여러 개의 자원에 작업을 효율적으로 분배하는 역할을 수행한다. 이처럼 보이지 않는 곳에서 런타임이 고군분투하는 덕분에 병렬 처리의 최적화에 대한 고민 없이도 PPL을 통해 효율적인 병렬 프로그래밍을 구현할 수 있는 것이다.

시작하기 전에

■ 개발 환경 및 운영 체제 요구 사항

PPL은 Visual Studio 2010 버전 이상에서 사용할 수 있다. 하지만 Visual Studio 2012 버전에서 새로운 기능이 많이 추가되었으니, 가능하면 Visual Studio 2012 버전 이상을 사용하길 권장한다. 참고로 이 책에 포함된 모든 예제를 오류 없이 컴파일하기 위해서는 Visual Studio 2012 버전 이상을 사용해야 한다.

PPL을 사용하여 개발된 응용 프로그램은 Windows XP SP3, Windows Vista SP2, Windows 7 이상의 운영 체제에서 실행할 수 있다. 그리고 서버 군에서는 Windows Server 2003 SP2, Windows 2008 SP2, Windows 2008 R2 버전 이상에서 실행할 수 있다.

■ 요구되는 사전 지식

이 책은 C++ 초중급 이상의 개발자라면 누구나 볼 수 있지만, 다음과 같은 사전 지식을 가지고 있다면 책의 내용을 이해하기가 훨씬 쉬울 것이다.

- STL 컨테이너와 알고리즘
- auto 키워드와 람다 표현식

STL 컨테이너와 알고리즘에 대한 개념이 아직 부족하다면, 한빛 미디어 홈페이지에서 무료로 다운로드 받을 수 있는 eBook[01]을 가볍게 한번 읽어보기를 추천한다. 그리고 C++11 표준에 정식으로 추가된 auto 키워드[02]와 람다 표현식[03]은 MSDN 사이트에 잘 설명되어 있으니 이 자료를 참고하기 바란다.

---

01 『Thinking About: C++ STL 프로그래밍』(http://www.hanbit.co.kr/ebook/look.html?isbn=9788979149937)
02 auto Keyword (http://msdn.microsoft.com/en-us/library/dd293667.aspx)
03 Lambda Expressions in C++ (http://msdn.microsoft.com/en-us/library/dd293608.aspx)

1 | PPL 시작하기

전통적인 방식으로 병렬 프로그래밍을 구현하려면 여러 개의 스레드를 생성하고, 각 스레드에서 처리할 프로시저를 작성해야 한다. 하지만 스레드는 개발자보다는 하드웨어와 운영체제에 가까운 저수준Low-Level의 개념으로, 이를 이용하여 효율적인 병렬 프로그램을 구현하는 것은 생각보다 까다롭고 신경 써야 할 것도 많다.

이러한 단점을 보완하기 위하여 'PPLParallel Patterns Library'에서는 스레드 대신 병렬 작업을 추상화시킨 '태스크task' 개념을 사용한다. 병렬 작업을 '스레드 단위'가 아닌 '논리적 작업 단위'로 표현함에 따라 스레드를 직접 다루는 것에 대한 부담에서 해방되고[01] 비즈니스 로직 구현에 집중할 수 있게 된다.

병렬로 실행되는 모든 작업은 태스크에서부터 시작되는 만큼, 태스크는 PPL을 이용한 병렬 프로그래밍의 핵심이다. 이제부터 태스크를 이용하여 작업을 구현하고 이를 병렬 처리하는 방법에 대해서 알아보자.

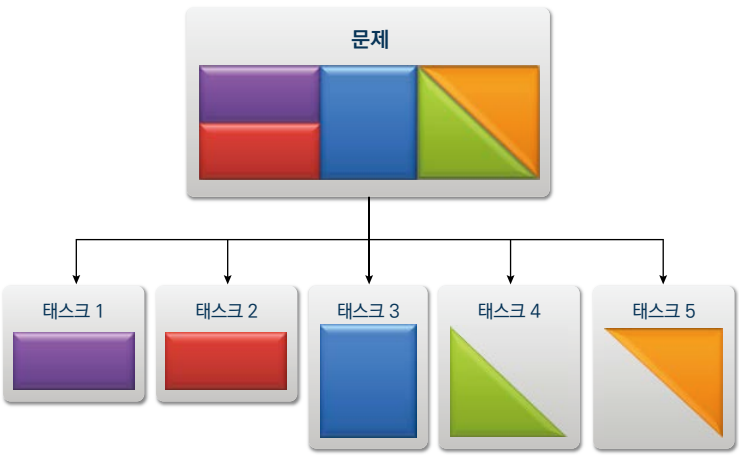
1.1 문제를 태스크로 분리하자

PPL을 이용한 병렬 프로그래밍은 동시에 처리할 수 있는 문제를 찾아 그것을 여러 개의 태스크로 분리하는 것부터 시작한다. 분리된 태스크는 각자의 역할에 따라 작업을 수행하는데, 이들은 병렬 처리되기 때문에 실행 순서가 보장되지 않는다. 따라서 각 태스크를 서로에 대한 의존성 없이 독립적으로 동작할 수 있도록 분리하는 것이 중요하다.

01 스레드를 생성하고 관리하는 역할은 동시성 런타임에 포함된 리소스 관리자와 태스크 스케줄러가 담당한다.

NOTE_ 문제에서 병렬성을 찾아 태스크로 분리하는 일은 전적으로 개발자의 몫이다. 수학 연산이나 이미지 프로세싱과 같이 병렬성이 풍부한 문제도 있지만, 병렬성이 부족하여 병렬화하기 어려운 문제가 더 많다. 모든 문제를 병렬화시킬 수 있는 것은 아니지만 자료 구조를 개선하면 어느 정도 병렬성을 높일 수 있을 것이다.

그림 1-1 주어진 문제를 여러 개의 독립적인 태스크로 분리한 모습



문제를 논리적인 작업 단위인 태스크로 분리했다면 실제 코드에서는 이를 어떻게 표현해야 할까? 이제부터 그 방법을 알아보자.

1.1.1 태스크를 추상화한 task_handle 클래스

PPL에서는 병렬 프로그래밍의 논리적 작업 단위를 표현하는 도구, 즉 태스크를 표현하기 위하여 태스크가 실행할 병렬 작업을 캡슐화한 'task_handle' 클래스를 제공한다. task_handle 클래스는 매우 단순한 구조의 템플릿 클래스로, 실행할 작업을 함수 포인터, 함수 객체Functor, 람다 표현식Lambda Expression 등의 형태로[02] 전달받

02 이후부터는 함수처럼 호출이 가능한 모든 형태를 일컬어 '함수 타입'이라 부를 것이다.

아 객체를 생성한다. 전달된 작업 함수는 이후 operator() 메소드를 통해 실행된다.

표 1-1 task_handle 클래스 메소드

메소드명	설명
생성자	작업 함수를 전달하여 객체를 생성
operator()	작업 실행

이제 여러 가지 함수 타입을 이용하여 task_handle 객체를 생성하고 이를 실행하는 예제를 작성해보자. PPL에서 제공하는 함수와 클래스를 사용하려면 'ppl.h' 헤더 파일을 포함시키고 'Concurrency' 네임스페이스를 사용해야 한다.

예제 1-1 세 가지 타입의 작업 함수를 전달받아 task_handle 객체를 생성하고 이를 실행하는 코드

```
#include <ppl.h>
#include <iostream>

using namespace Concurrency;
using namespace std;

void task_function()
{
   wcout << L"Task1" << endl;
}

class task_functor
{
public:
   void operator()() const
   {
       wcout << L"Task2" << endl;
```

```
    }
};
int main()
{
    // 함수 객체
    task_functor functor;

    // 함수 포인터, 함수 객체, 람다 표현식을 이용한 task_handle 객체 생성
    task_handle<function<void(void)>> task1(task_function);
    task_handle<function<void(void)>> task2(functor);
    task_handle<function<void(void)>> task3([](){
        wcout << L"Task3" << endl;
    });

    // 태스크 실행
    task1();
    task2();
    task3();

    return 0;
}
```

예제 1-1의 실행 결과

```
Task1
Task2
Task3
```

예제 1-1에서는 task_handle 객체를 생성하기 위해 세 가지 타입의 작업 함수를 사용했다. 이 중에서도 람다 표현식은 가독성이 좋고 코드를 간결하게 만들어주기 때문에 간단한 태스크를 정의하는 경우에 많이 사용된다. 앞으로 살펴볼 예제에서

는 작업 함수를 구현할 때 람다 표현식을 사용할 것이다.

여기서 잠깐_ 람다 표현식 Lambda Expression

람다 표현식은 C++11에서 새롭게 추가된 문법으로, 함수의 몸체는 있지만 이름은 없는 익명의 함수 객체다. 상태를 유지할 수 있다는 함수 객체의 장점을 가지면서도 별도의 클래스나 구조체를 정의해야 하는 번거로움이 없어서 간단한 함수 객체를 정의할 때 편리하게 사용할 수 있다. 또한 함수 객체를 정의하는 곳과 사용하는 곳이 분리되어 있지 않기 때문에, 가독성이 좋고 코드를 간결하게 작성할 수 있다는 것도 장점이다. 이 책에 포함된 예제를 이해하려면 반드시 람다 표현식에 대한 사전 지식이 필요하므로 이에 대해 충분히 숙지하고 나머지 과정을 진행하길 바란다. 람다 표현식에 대한 자세한 내용을 알고 싶다면 MSDN 사이트의 'Lambda Expressions in C++'(http://msdn.microsoft.com/en-us/library/dd293608.aspx) 문서를 참고하기 바란다. Visual C++ 환경에서 람다 표현식은 Visual Studio 2010 버전 이상에서 사용할 수 있다.

생성된 task_handle 객체는 operator() 메소드를 통해 함수처럼 호출될 수 있는데, 이때에는 생성자로 전달받은 작업 함수가 실행된다. 사실 예제 1-1처럼 태스크 객체의 작업 함수를 직접 호출하는 일은 없으며 task1, task2, task3는 병렬 처리되지 않고 순차적으로 실행된다. 이 부분에서는 task_handle 객체의 작업 함수가 이러한 방식으로 호출된다는 것만 이해하고 넘어가자.

예제 1-1에서 가장 까다로운 작업은 task_handle 클래스의 템플릿 파라미터를 기술하는 것이다. 작업 함수의 정확한 프로토타입대로 파라미터를 기술하지 않으면 컴파일 오류가 발생하는데, 템플릿에 익숙하지 않은 개발자에게 이것은 생각보다 어렵고 귀찮은 작업이다. 다행히도 PPL에서는 task_handle 객체를 생성해주는 make_task 함수를 제공한다. 예제 1-2와 같이 make_task 함수와 auto 타입을 사용하면 간단하게 태스크를 생성할 수 있다.

예제 1-2 make_task 함수를 이용한 태스크 객체 생성

```
auto task3 = make_task([](){
   wcout << L"Task3" << endl;
});

task3();
```

1.2 태스크 그룹을 이용하여 태스크를 병렬 처리하자

문제를 여러 개의 작업 함수로 분리하고 이를 이용하여 task_handle 객체를 생성했다면, 마지막으로 남은 것은 태스크를 병렬로 실행하는 일이다. PPL에서는 태스크 컬렉션을 구성하고 태스크의 실행 및 관리를 담당하는 태스크 그룹을 제공한다. 태스크를 태스크 그룹에 추가하여 실행하면 동시성 런타임에 포함된 태스크 스케줄러에 의해 스케줄링되어 병렬로 실행된다. 여기서는 내부적으로 태스크가 스레드를 몇 개 사용하여 실행되는지 신경쓰지 않아도 된다. 태스크 스케줄러가 CPU 코어 개수와 현재 리소스 사용률 등을 고려해 최적화된 방법으로 태스크를 실행하기 때문이다.

그림 1-2 태스크 그룹에 태스크를 추가하여 병렬로 실행하는 모습

PPL에서 태스크 그룹은 두 개의 클래스(structured_task_group 클래스와 task_group 클래스)로 구분되어 있다. 기본적인 동작과 사용 방법은 같지만 몇 가지 중요한 차이점이 있으니, 주어진 환경에 맞게 둘 중 하나를 선택하기 바란다. 지금부터 structured_task_group 클래스와 task_group 클래스 관해서 자세히 알아보자.

1.2.1 structured_task_group 클래스

PPL에서 말하는 태스크 그룹은 일반적으로 'structured_task_group 클래스'를 의미한다. 앞으로 병렬 프로그래밍을 구현하기 위한 대부분의 상황에서 직접 또는 간접적으로 structured_task_group 클래스를 이용하게 될 것이다. 이제, structured_task_group 클래스를 이용하여 태스크를 병렬로 실행하는 가장 기본적인 코드를 작성해보자.

예제 1-3 structured_task_group 클래스를 이용한 병렬 작업 실행

```
#include <ppl.h>
#include <iostream>

using namespace Concurrency;
using namespace std;

int main()
{
   // task_handle 객체 생성
   auto task1 = make_task([] {
       wcout << L"Task1" << endl;
   });
   auto task2 = make_task([] {
       wcout << L"Task2" << endl;
   });
   auto task3 = make_task([] {
```

```
        wcout << L"Task3" << endl;
    });

    // structured_task_group 객체를 생성
    structured_task_group tasks;

    // 태스크를 병렬로 실행하고 종료 대기
    tasks.run(task1);
    tasks.run(task2);
    tasks.run(task3);
    tasks.wait();

    return 0;
}
```

예제 1-3의 실행 결과

```
Task2
Task1
Task3
```

병렬 작업을 나타내는 task_handle 객체를 생성하고 태스크 그룹을 나타내는 structured_task_group 객체를 생성했다면, 작업을 병렬로 실행할 준비가 모두 완료된 것이다. 그다음 task_handle 객체를 파라미터로 전달하여 structured_task_group::run 메소드를 호출하면, 태스크가 태스크 그룹에 포함되고 런타임에 의해 병렬로 실행된다.

런타임은 최적화된 스케줄링 방식으로 태스크를 병렬 처리하므로 이것저것 신경 쓸 필요 없이 작업 구현에만 집중하면 된다. 이것이 바로 PPL을 이용한 병렬 프로그래밍의 핵심이며, PPL을 사용하는 이유다.

예제 1-3의 마지막 부분에서는 wait 메소드를 호출하여 실행한 모든 태스크가 종료될 때까지 대기한다. structured_task_group 객체를 사용하면 태스크를 모두 실행한 후에 반드시 wait 메소드를 호출해야 한다. 병렬 작업의 특성상 예제 1-3의 출력 결과는 순서를 예측할 수 없으며 실행할 때마다 달라질 수 있다.

태스크 그룹의 run 메소드와 wait 메소드를 각각 호출하는 대신, run_and_wait 메소드를 사용하면 태스크를 태스크 그룹에 포함시켜 실행함과 동시에 모든 태스크가 종료될 때까지 대기하도록 한 번에 구현할 수 있다. 예제 1-4는 run_and_wait 메소드를 이용하여 태스크를 실행하고 대기시키는 예제다.

예제 1-4 run_and_wait 메소드를 이용한 태스크 실행과 대기

```
tasks.run(task1);
tasks.run(task2);
tasks.run_and_wait(task3);
```

run_and_wait 메소드는 태스크의 실행과 대기를 한 번에 처리하기 때문에 태스크는 별도의 스레드에서 실행될 필요가 없다. 따라서 run_and_wait 메소드로 실행된 태스크는 항상 호출된 스레드에서 실행된다. 그리고 run_and_wait 메소드는 task_handle 타입 이외에도 함수 타입을 파라미터로 전달받는 오버로드 함수를 가지고 있다. 그러므로 task_handle 객체를 명시적으로 생성하지 않고 작업 함수를 전달하여 태스크를 실행할 수 있다.

예제 1-5 람다 표현식을 사용하여 run_and_wait 메소드 호출

```
tasks.run_and_wait([] {
   wcout << L"Task3" << endl;
});
```

표 1-2는 structured_task_group 클래스가 제공하는 모든 메소드와 그에 관한 설명을 표로 나타낸 것이다. 태스크 그룹 취소 시 사용하는 cancel 메소드와 is_canceling 메소드에 대해서는 "5장 병렬 작업의 취소"에서 살펴보겠다.

표 1-2 structured_task_group 메소드

메소드명	설명
run	전달받은 태스크를 실행
wait	그룹에 포함된 모든 태스크가 종료될 때까지 대기
run_and_wait	전달받은 태스크를 실행하고 모든 태스크가 종료될 때까지 대기
cancel	태스크 그룹 실행 취소
is_canceling	태스크 그룹이 현재 취소 상태에 있는지 확인

1.2.2 task_group 클래스

task_group 클래스는 structured_task_group 클래스와 마찬가지로 태스크를 병렬로 실행하는 역할을 담당하며, 제공하는 메소드 역시 structured_task_group 클래스와 동일하다. 그럼 task_group 클래스와 structured_task_group 클래스에는 어떤 차이점이 있을까?

- 첫 번째 차이점은 메소드의 스레드 세이프[03] 여부다. 이는 structured_task_group 클래스와 task_group 클래스를 구분하는 가장 큰 차이점이다. structured_task_group 클래스는 스레드 세이프하지 않기 때문에 객체를 생성한 스레드 안에서만 메소드를 호출해야 한다는 제약이 따른다. 반면 task_group 클래스는 스레드 세이프하기 때문에 서로 다른 스레드(태스크)에서 동시에 메소드를 호출할 수 있다.

  task_group 클래스를 사용하면 태스크 내부에서 또 다른 태스크를 태스크 그룹

03 '스레드 세이프하다'는 것은 여러 스레드에서 동시에 사용해도 안전하다는 것을 의미한다.

에 추가하는 것처럼, 조금 더 유연하고 자유로운 코드를 작성할 수 있다. 그런 반면, task_group 클래스에는 스레드 세이프한 동작을 위한 동기화 코드가 포함되어 있어서 structured_task_group 클래스에 비해 성능이 다소 떨어진다.

스레드 세이프 규칙에는 한 가지 예외 사항이 있는데, cancel 메소드와 is_canceling 메소드는 두 클래스 모두 다른 스레드에서 호출해도 안전하다는 점이다. 이러한 예외 규칙에 덕분에 structured_task_group 객체가 실행하는 태스크 내부에서도 cancel 메소드를 호출하여 태스크 그룹을 취소할 수 있다.

- 두 번째 차이점은 wait 메소드를 호출하여 태스크 종료를 대기하는 도중에 다른 태스크를 태스크 그룹에 추가할 수 있는지 여부다. 이는 사실 첫 번째 항목의 연장선상에 있는 내용이다. structured_task_group 클래스는 객체를 생성한 스레드 안에서만 메소드를 호출할 수 있기 때문에 이러한 동작이 불가능하다. 반면 task_group 클래스는 wait 메소드로 대기하는 중에도 다른 스레드(태스크)에서 run 메소드를 호출하여 새로운 태스크를 태스크 그룹에 추가할 수 있다.

- 세 번째 차이점은 run 메소드의 파라미터 타입이다. structured_task_group 클래스는 run 메소드의 파라미터로, 오직 task_handle 타입만을 사용할 수 있다. 반면 task_group 클래스는 run 메소드의 파라미터로, task_handle 타입뿐만 아니라 함수 타입을 추가적으로 사용할 수 있다. 만약 함수 타입을 전달하여 run 메소드를 호출했다면 내부적으로 task_handle 객체를 생성하여 관리한다. 참고로 run_and_wait 메소드는 structured_task_group 클래스와 task_group 클래스에서 모두 task_handle 타입 또는 함수 타입을 파라미터로 사용할 수 있다.

위에서 설명한 세 가지 차이점을 종합하여 structured_task_group 클래스 조건에 만족하는 경우 structured_task_group 클래스를 사용하는 것이 좋으며, 그렇지 않을 경우에만 task_group 클래스를 사용할 것을 권장한다. 그럼, 설명한 내용을 토대로 task_group 클래스를 사용하는 코드를 작성해보자.

예제 1-6 structured_task_group 클래스와 task_group 클래스 비교

```
#include <ppl.h>
#include <sstream>
#include <iostream>

using namespace Concurrency;
using namespace std;

// 메시지를 화면에 출력
template<typename T>
void print_message(T t)
{
   wstringstream ss;
   ss << L"Message from task: " << t << endl;
   wcout << ss.str();
}

int wmain()
{
   // structured_task_group 객체와 task_group 객체를 각각 생성
   structured_task_group tg1;
   task_group tg2;

   auto task1 = make_task([&tg2]()
   {
      // 태스크가 병렬로 실행되므로 다른 스레드에서 동작할 수 있음
      tg2.run([] {
         print_message(L"Hello");
      });
      // run 메소드를 task_handle 객체가 아닌 람다 표현식을 이용하여 호출
      tg2.run([] {
         print_message(42);
```

```
        });
    });
    auto task2 = make_task([&tg2]()
    {
        // 태스크가 병렬로 실행되므로 다른 스레드에서 동작할 수 있음
        tg2.run([] {
            print_message(3.14);
        });
    });

    // tg1을 실행하고 모든 태스크가 종료될 때까지 대기
    tg1.run(task1);
    tg1.run_and_wait(task2);

    // tg2의 모든 태스크가 종료될 때까지 대기
    tg2.wait();
}
```

예제 1-6의 실행 결과

```
Message from task: Hello
Message from task: 3.14
Message from task: 42
```

예제 1-6에서 주목해야 할 것은 task_group 객체인 'tg2'이다. 'tg1'은 메인 스레드에서 차례대로 메소드를 호출하지만 'tg2'는 병렬로 실행되는 두 개의 태스크 내부에서 run 메소드를 호출한다. 그래서 run 메소드가 두 개의 스레드에서 동시에 호출될 수도 있지만, task_group 객체는 이러한 동작에 안전하다. 그리고 'tg2'는 run 메소드를 호출할 때 명시적으로 task_handle 객체를 생성하지 않고 람다 표현식을 이용하여 호출하고 있는데, 이것 역시 task_group 클래스의 특징이다. 예제 1-6의 출력 결과는 순서를 예측할 수 없으며 실행할 때마다 다를 수 있다.