Gamasutra.com

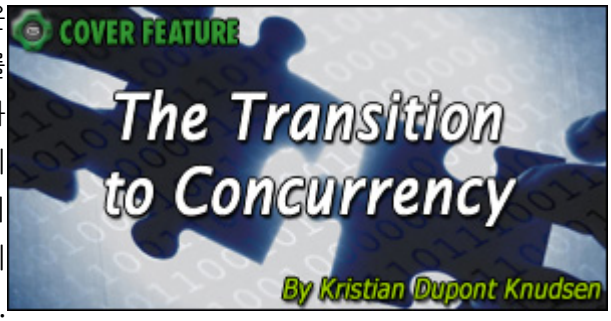
새로운 화두: 병행실행

Kristian Dupont Knudsen

2007년 4월 5일

http://gamasutra.com/features/20070405/knudsen_01.shtml

들어가며



병행실행(concurrency)은 가까운 미래에 핵심 화두로 떠오를 전망이다. 병행실행에 대한 논의가 수년간 진행되어 왔지만 이제 인텔과 AMD가 이쪽으로 움직이기 시작하고 있다는 사실은 병행실행에 뭔가가 있다는 것을 암시해준다. 최근들어 본 기사를 포함하여 병행실행과 관련된 수많은 글이 발표되었으므로 필자가 새삼 그 중요성을 반복할 필요는 없을 것이다. 그 대신 필자는 지금 당장 우리가 나아가야 할 방향을 제시하고자 한다.

독자가 C++ 프로그래머라면 세마포어 기반의 병행실행에 대해 이미 알고 있을 것이다. 뮤텍스를 비롯한 여타의 잠금(lock) 매커니즘들은 매우 어려운 사용법에도 불구하고 현재 병행실행에 가장 많이 사용하고 있다.

게다가 때로는 성능저하를 가져오는 방법으로 잠금 매커니즘을 사용하는 경우도 있다. 물론 이 잠금장치들을 완전히 버려야 한다는 것은 아니지만 병행실행 프로그램을 작성함에 있어서 세마포어가 최초이자 유일한 방법이어서는 안 될 것이다.

Erlang 과 그 친구들

병행실행에 접근하는 방법은 여러가지가 있는데 그 중에서도 적합한 프로그래밍 언어의 선택이 중요하다. 특히 Erlang은 병행실행 프로그래밍을 위해 제작된 함수형 프로그래밍 언어(functional programming language)이다.

일반적으로 함수형 프로그래밍 언어들은 병행실행에 탁월한 기능을 자랑하고 있는데, OCaml의 예를 들자면 단순히 컴파일러를 최적화시켜 우수한 성능 향상을 도모할 뿐만 아니라, 실제로 성능을 대폭 향상시켜주는 것으로 잘 알려져 있다. 다시 말해서, 병행실행 프로그램을 염두에 두고 있다면 Ocaml이야말로 최적의 선택이라는 뜻이다.

굳이 새로운 언어로 전환해야 하는가?

성능 향상에 관심이 있는 독자라면 필시 서버 프로그램이나 임베디드 프로그램, 혹은 본 필자처럼 게임 프로그램을 작성하는 일을 하고 있을 것이다. 만일 그렇다면 독자 여러분이 사용하는 언어는 C 나 C++일 것이며 이 외의 언어에는 별 관심이 없을 것으로 생각된다. 어찌됐든 C 와 C++은 매우 유동적인 언어들이니까. 그렇다면 여러분이 직접 구현할 수 없는 기능을 제공하는 프로그래밍 언어가 존재하기나 할까?

물론 그러한 생각이 틀린 것은 아니다. 프로그래머 스스로 기능을 구현해서 모든 문제를 해결할 수 있으니 말이다. 기존의 디자인 패턴을 이용하거나 새로운 디자인 패턴을 만들어서 이런 기능을 구현하는 것을 도울 수도 있다. 어찌됐든 디자인 패턴라는 것은 (아직) 구현되지 않은 언어의 기능으로 일컬어지기도 한다. 좀 복잡하긴 하지만 C 언어로 상속과 가상 메서드를 비롯한 모든 것을 직접 구현하여 객체지향 코드를 작성할 수도 있다. C 언어로 함수형 프로그래밍을 한다는 것은 고통스러운 작업이다 C++이라면 고통이 좀 줄어들지만 그래도 골치 아프긴 마찬가지이다.

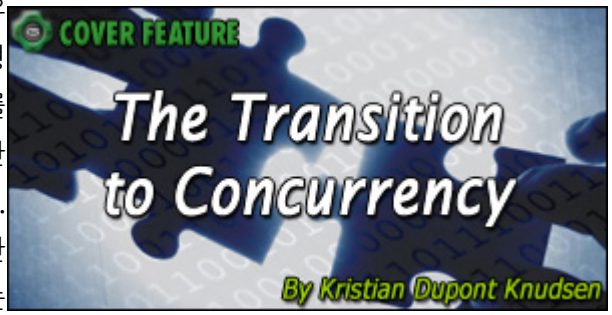
하지만 이러한 작업에 더 적합한 언어가 존재한다는 사실을 인정한다고 해도 그 언어를 실제로 사용하지는 못할 수도 있다. 아무리 뛰어난 프로그래머라 해도 바다 한가운데 고립된 섬처럼 독불장군 노릇을 할 수 없으며 다른 프로그래머들과 협력해야만 한다. 또한 상황을 보고해야 할 경영진이 있기 때문에 사용 언어를 바꾼다는 것은 사소한 결정이 아니다. 게다가 많은 플랫폼에서 C++ 컴파일러를 대체할 수 있는 것이 존재하지 않는다.

우리 Flux Studios에서는 마이크로소프트가 Xbox에 대한 지원을 제공할 것이라는 전제 하에 C#을 이용하여 스크립팅을 해 왔다. 물론 지원을 할 것으로 예상되긴 하지만 C#에 대해 문제를 일으킬 수 있는 C++/CLI에 대한 지원도 곧 이루어질 것인지는 분명하지 않다.

(단, 독자 여러분의 타겟 플랫폼이 자바 나 닷넷의 가상머신(VM)이라면 Scala, Nemerle 혹은 F#와 같은 하이브리드 언어를 실험적으로 이용해볼 수 있다. 이 언어들은 상당히 흥미롭게 보이는 동시에 자바 및 C#을 이용하는 프로그래머에게 나름대로 친숙하게 느껴질 것이다.)

OpenMP와 같은 API 덕분에 C++도 충분한 기능을 갖출 수 있게 되었지만 그렇다 해도 병행실행에 적합한 패러다임들을 이해하는 것이 가장 도움이 될 것으로 보인다.

상태(state)를 피하자: 변경불가 데이터형



일반적으로 적용하기에 매우 유용한 정책이기도 한 함수형 프로그래밍의 핵심은 특정 상태를 피하는 것이다. 특히 변경가능한 상태는 지양해야 할 대상이다. 소프트웨어에 있어서 변경가능한 상태라 함은 물리적으로 움직이는 하드웨어와 준하는 것으로, 열심히 작성한 코드가 외부 환경에 영향을 받는 것을 의미하며, 이는 코드를 유지보수하는 것을 어렵게 만든다.

함수형 프로그래밍 언어를 사용해 본 경험이 전혀 없는 독자라면 이 말을 이해하기가 다소 어려울 수 있으며, 혹은, 구체적인 차이를 인지하기가 어려울 수도 있다(필자의 경우였다).

순수한 함수형 프로그래밍에서는 모든 것이 변경불가상태, 즉 상수라는 것이다. C++ 형식의 변수를 변경불가상태로 선언하는 것은 불가능하지만 변수를 const 로 선언할 수는 있으며, 이 경우 지정된 데이터형의 변경불가상태형 버전으로 그 변수를 선언하는 것이다.

간단한 예제

아래와 같은 간단한 Color 클래스를 보자:

```
class Color
{
private:
        double red_;
        double green_;
        double blue_;

public:
        Color(double red, double green, double blue) :
       red_(red), green_(green), blue_(blue) {}

        double GetRed() const { return red_; }
        double GetGreen() const { return green_; }
        double GetBlue() const { return blue_; }

  double SetRed(double red) { red_ = red; }
        double SetGreen(double green) { green_ = green; }
        double SetBlue(double blue) { blue_ = blue; }

        void Multiply(double factor)
        {
                red_ *= factor;
                green_ *= factor;
                blue_ *= factor;
        }
```

```
};
```

상당히 흔한 형태로 보인다. private 필드에 읽고 쓰는 것이 모두 가능하긴 하지만 캡슐화가 잘 되어 있다. 하지만 접근자(accessor)와 Multiply 메서드에 체크를 추가함으로써 간단하게 컨트랙트를 삽입할 수도 있다.

이 클래스를 변경불가상태로 만들려면 아래와 같이 변경할 수 있을 것이다::

```
class Color
{
private:
        double const red_;
        double const green_;
        double const blue_;

public:
        Color(double red, double green, double blue) :
       red_(red), green_(green), blue_(blue) {}

        double GetRed() const { return red_; }
        double GetGreen() const { return green_; }
        double GetBlue() const { return blue_; }

        Color Multiply(double factor) const
        {
                return Color(red_ * factor, green_ * factor, blue_ * factor);
        }

private:
        Color& operator = (Color const& a) {}
```

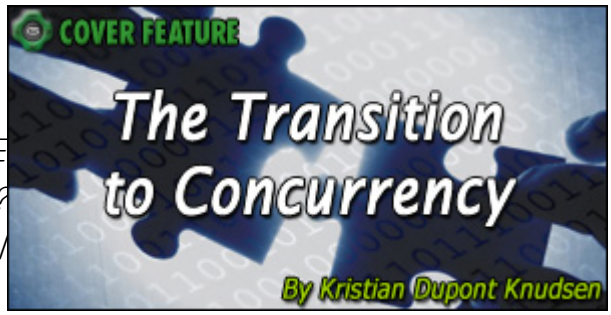
```
};
```

언뜻 보기에 별다른 차이는 없어 보인다. Set* 메소드를 제거하고 Multiply 메소드 부분에도 다소간의 변화를 주었지만 별건 아니다. 진짜로 바뀐 부분은 이 클래스가 이제 변경불가상태가 되었다는 사실이다.

객체가 한번 실행되고 난 다음에 이 객체의 내용을 변경하는 것은 절대 불가능하다. 매개변수 factor 를 통해 색상을 바꿀 수는 있겠지만 이는 완전히 새로운 인스턴스를 제공한다. 또한 대입 연산자도 private 으로 선언되는데 이는 묵시적인 방식으로라도 외부에서 대입 연산자를 호출할 수 없음을 의미한다. 이 덕분에 세마포어가 전혀 없음에도 이 클래스는 스레드로부터 위협을 받지 않게 된다.

여러분은 다중 스레드에서 효과를 볼 수 있는 방식으로 이 클래스를 써야만 하며 그렇게 해도 교착상태(deadlock)에 빠지거나 동시 액세스로 인한 데이터 불일치가 발생하지 않을 것이라고 확신해도 좋다. 이 버전에 컨트랙트를 삽입한다는 것은 생성자의 변수(하지만 값이 변하지 않는다)를 확인한다는 뜻이며, 단지 그 뿐이다.

복잡한 클래스와 데이터 구조



"일부의 값만 변경하려 할 때에도 전체를 복사하라는 말씀이신가요? 성능을 최대한 이끌어내는 것이 이 프로젝트의 목적 아니었습니까?"

전체 복사본을 생성하는 것은 많은 비용을 초래하는 작업이다. 특히 클래스가 복잡하며 전형적인 변경가능 데이터 구조에 의존하고 있다면 더욱 그러하다. 이것이야말로 여러분이 지속성 데이터 구조라고도 불리는 변경불가상태 데이터 구조를 고려해야 할 이유이다.

명령형 프로그래머들은 배열을 상당히 많이 사용하는 반면 함수형 프로그래머들은 링크된 리스트를 더 선호한다. 함수형 프로그래밍에 경험이 있다면 이 친구들이 프로그램의 머리와 꼬리, 즉 CAR 과 CDR 에 집착한다는 것을 이미 눈치챘을 것이다. 여기서 머리라 함은 링크된 리스트의 첫번째 요소이며 꼬리는 리스트의 나머지 부분을 의미한다.

이 간단한 연산들에는 중요한 특성이 있는데 이를테면 이 연산들을 통해 포인터를 복사함으로써 이전 값을 망치지 않고서도 새로운 값을 부여할 수도 있다. 변경 불가능한 리스트의 꼬리는 그 자체로 변경불가 리스트를 이루며 이 꼬리를 다른 머리에 연결시키면 이전 요소를 파괴하지 않고서도 새로운 요소를 갖춘 리스트를 만들 수 있는 것이다.

물론 링크된 리스트에는 잘 알려진 단점들도 없지 않은데 이 중 한가지는 랜덤 액세스가 불가능하다는 점이다. 이 경우에는 레드-블랙 트리와 같이 유용하게 활용할 수 있는 트리 구조가 많이 있다. 또한 현재로서는 관련 표준에 지속성 컨테이너에 대한 개념이 전혀 정립되어 있지 않으며 C++에 이러한 기능을 제공하는 범용 라이브러리를 아직 본 적도 없다. 그러므로 당분간 독자 여러분은 알아서 해결해야 하는 상황에 처해있다고 볼 수 있다. 또한 조만간 이 분야가 매우 활성화될 것으로 판단된다.

차세대 스타: 액티브 객체

데이터를 공유할 필요가 있을 때 이 데이터에 대한 접근이 병렬 스레드를 통해 이루어지더라도 우리는 이 접근을 순차적으로 만들 필요가 있다.

Erlang 은 C++에서 액티브 객체를 통해 시뮬레이션이 가능한 메시지를 사용함으로써 이를 가능하게 만들어준다. 액티브 객체란 객체와 스레드의 개념을 하나로 통합시켜주는 디자인 패턴으로, 객체가 생성될 때마다 새로운 스레드를 만드는 것이 Erlang 의 프로세스와 유사하다.

객체상에서 직접 메서드를 호출하는 대신에, 메시지를 전송하여 FIFO 큐에 합류시키면 그 객체가 가지고 있는 스케쥴러가 큐에 있는 메시지를 지속적으로 처리한다. 이에 따라 데이터에 실질적으로 접근하는(또는, 최소한, 이 데이터에 새로운 기록을 작성하는) 유일한 스레드는 소유자 스레드가 되며 다른 스레드는 모두 명령을 내리는 역할만 수행한다.

반환값이 필요하게 되는 경우가 발생하면 이는 퓨처(future) 패턴을 사용한다. 퓨처라 함은 준비된 결과를 포함하고 있는 객체를 말한다. C++에 액티브 객체 디자인 패턴을 적용시킨 예제에 대해 Paul Bridget이 작성한 문서를 이곳에서 볼 수 있다.

예외 안전성

병행실행을 거부감없이 받아들이는 것 외에 이러한 코딩 방식에 공통적으로 주목할 만한 점이 있다면 예외 안전성(exception safety)을 높여준다는 것이 있다.

예외 안전성을 확보한다는 것은 절대 사소한 문제가 아니며 게임 개발자들은 수행속도가 떨어질 것을 우려하여 예외처리를 회피하는 경향이 있는데 C++의 인라인 메커니즘이 이를 최소화 시켜주긴 하지만 이러한 우려는 합당한 것이다.

하지만 합리적인 불변변수를 형성하고 이 변수가 제대로 유지된다고 확신할 수 있다면, 익셉션을 방지할 수 있을 뿐만 아니라 전체적으로 프로그램의 코드를 안정적으로 만들 수가 있는 것이다. 또한 유닛 테스팅에서도 훨씬 좋은 결과를 얻을 수가 있게 된다. 필자의 판단에 따르면 병행실행 코드는 유닛 테스팅이 별다른 도움이 되지 못하는 분야이다. 적어도 세마포어 기반의 코드에서는 말이다.

다만 변동불가상태의 클래스에 대한 테스트를 쉽게 수행하고, 이 클래스가 정말로 변동불가상태라면, 병행실행능력에 대한 우려도 해소될 수 있을 것이다. 이러한 사실은 액티브 객체에도 대동소이하게 적용된다.

냄새를 쫓아라

"좋아. 일단 받아들이겠어. 하지만 지금 진행중인 프로젝트에서는 전체 클래스의 1~2 퍼센트에만 전환이 가능하단 말이지." 라고 독자 여러분은 생각하고 있을 수도 있다. (도대체 새로운 패러다임이나 시스템에 대한 소문을 들을 때마다 예외없이 떠오르는 생각이란, "음, 말은 되는데, 지금 내가 하고 있는 작업에 적용이 되려나?"와 같은 것일까?)

명령형 코드를 함수형 코드로 변환하는 것은 거대한 작업이며 독자 여러분에게는 현실성이 없을 수도 있다. 하지만 추후 완전히 처음부터 코드를 작성하는 경우가 생긴다면 생각을 해 볼 수도 있을 것이다. 이러한 경우에 인지하고 있어야 하는 '냄새'에 대해 알아보자.

1. 초기화 함수

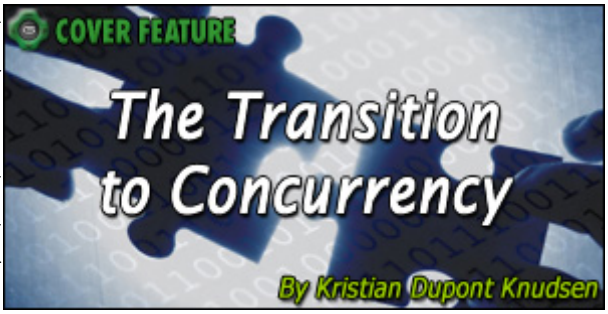
이 방법은 필자도 항상 적용하는 것으로, 필자의 판단에 이상적인 클래스란 생성자가 호출하거나 재초기화가 필요한 경우엔 프로그래머도 직접 호출할

수 있는 초기화 함수가 포함된 클래스이다. 필자는 클래스에 대해서 컨텐츠를 담고 있는 일종의 용기로 간주해왔지만 지금은 컨텐츠 자체를 나타내는 것으로 보고 있다.

클래스를 작성할 때 초기화가 단 한번만, 생성 단계에서 이루어지도록 하라. 이 말은, 만약 C++로 코딩하고 있다면, 그 자체로 변경불가능 상태인 레퍼런스의 힘을 잃지 않고서도 가능하다는 뜻이며(참조된 변수가 아닌 실제 레퍼런스를 말한다) 레퍼런스의 초기화는 생성자를를 통해서만 가능하다.

2. 디폴트 생성자

기본 생성자라 함은 정보를 전혀 내포하지 않은 상태에서 클래스가 초기화 되는 것을 말한다. 디폴트 생성자를 쓸 경우에는 보통 별도의 초기화 함수가 있거나 일련의 set 함수들이 있는 경우가 많으므로 디폴트 생성자가 쓸모가 있는 경우는 별로 없다. 인스턴스를 생성하였으나 초기화가 되지 않은 객체는 좀비 객체에 해당하며 문제를 발생시킬 소지가 다분하다. 따라서 필요한 모든 정보를 생성자에 제공하도록 하자.



만약 사용중인 클래스가 복잡하다면 생성자의 매개 변수 리스트가 길어질 수도 있다. 이 현상은 그 자체만으로 클래스가 지나치게 복잡해질 수 있음을 암시하는 '냄새'에 해당하지만 사실 정말 어쩔 수 없는 경우도 존재한다. 이러한 난관에 봉착했다면 Essence design pattern에서 도움을 구해 보도록 하자.

3. public 유효상태 함수

독자 여러분이 작성한 클래스에 *public bool IsValid()* 함수나 이와 유사한 것이 있다면 여러분은 컨트랙트에 대한 적용여부를 사용자에게 넘기는 것이나 다름없다. 이는 여러가지 이유로 인하여 좋지 못한 경우에 해당한다. 먼저 변경불가상태가 없는 클래스라면 생성 중을 제외하고 이러한 검사가 필요하지 않다. 만일 검사에서 실패한다면 예외를 발생시켜 클래스의 생성을 막아야하며, 예외를 사용하지 않는다면 단정(assert)을 이용하여 객체를 생성하기 전에 컨텐츠를 검사해야 할 것이다. 이렇게 하면 클래스가 불변식을 위반하는 경우는 발생하지 않게 된다.

만일 클래스에 변경 가능상태가 존재한다면 private 상태를 유지한다는 전제하에 이러한 검사를 하는 것이 좋은 생각일 수 있다. 이에 대한 자세한 내용은 contract programming 에서 확인할 수 있다.

4. 널 포인터 검사

방어적으로 프로그램을 작성하지 말 것. 널 포인터 검사는 대부분 컨트랙트 검증의 한가지 형태에 불과한데 이 말은 위의 3 번 항이 여기서도 적용됨을 의미한다. 아무래도 널 포인터가 들어갔을 것에 대해 불안하다면 단정을 이용해보자. 중요한 것은 IF 구문이 아니어야 한다는 점이다. (대개 프로그래머들은 널 포인터를 유효한 값으로 사용하여 "존재하지 않음"이란 의미로도 사용하는데 이것도 아주 좋은 방법이다.)

5. 상속

상속 자체에는 아무런 문제가 없으며 독자 여러분도 자신의 개발 업무에 상속을 잘 활용하고 있을 것이다. 하지만 일반적으로 상속은 함수형 프로그래밍 시스템에 잘 맞지 않는다. 특정 클래스가 변경불가상태라면 여기에 서브클래스를 만드는 것은 함수형 시스템에 잘 들어맞지 않거나 읽기 전용 메서드의 수를 늘리는 것밖에 되지 않는다.

6. 대형 클래스

대형 클래스 역시 복잡한 클래스일 수 있다는 뻔한 사실은 둘째치고, 대형 클래스를 병행실행 환경에서 이용할 경우 그보다 훨씬 큰 문제를 야기할 수 있다. 독자 여러분의 클래스가 변경불가상태라면 복사 가능한 정보의 양은 더욱 많다(하지만 만일 대형 클래스라면 변경불가상태는 아닐 가능성이 높다. 클래스의 크기와 변경가능성은 함께 어울리지 못하는 상호 배타성이 강하다). 그렇지 않다면, 불변식에 오류가 있거나 트랜잭션 및 세마포어 락 중에 지속성을 유지시켜야 할 데이터가 더 있다는 의미가 된다.

StateAid

세마포어가 없는 경우라 해도 병행실행 프로그램은 디버그 및 검증이 어렵다. 그 이유는 코드를 검사해도 이벤트의 순서를 판명할 수가 없기 때문이다.

이러한 이유를 바탕으로 본 필진은 StateAid라 하는 조그만 툴을 개발했다. StateAid는 비선형 방식으로 프로그램의 실행을 볼 수 있도록 도와준다.

맺음말

머지않아 병행실행이 수많은 개발자들의 삶에 일부분을 차지하게 될 것이며 C++ 및 그와 유사한 언어들은 현재 상태로는 병행실행에 잘 어울리지 않는다.

지금까지 개발자들은 날로 발전하는 CPU 아키텍처에 맞춰 코드를 훌륭히 최적화시켜주는 컴파일러에 너무 의존해왔다. 하지만 전형적인 명령형 코드는 자동으로 병행화하기가 어렵기 때문에 하드웨어의 새로운 발전에 대하여 앞으로도 컴파일러들이 제대로 역할을 수행할 것이라 기대하기도 어렵다.

세마포어는 일반적인 것이며 잘 알려져 있지만 이를 제대로 활용하기는 어렵다. 병행실행에 대해 제대로 준비하고 싶다면 변경실행상태, 지속성 데이터 구조 및 액티브 객체에 대해 익숙해짐으로써 이 변화를 주도할 수 있을 것이다.