


C++ Asynchronous Programming

Lyn Heo / C++ Korea FaceBook Group



Lyn Heo / C++ Korea FaceBook Group



오늘 내용

- 비동기 프로그래밍이 무엇인가
- 비동기 프로그래밍의 장단점
- 비동기 프로그래밍의 구현 방식
- 각각의 방식에 대한 장단점
- 앞으로는?

비동기 프로그래밍이란?

- 함수를 호출 하는 시점과 결과를 받아 오는 시점이 다른것
- 주로 파일 / 네트워크등의 IO나 대량의 연산을 할때 사용

1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μ s	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years

출처 : 코딩호러



비동기 프로그래밍의 장점

- 함수를 호출 해 두고 다른 작업을 할 수 있다
- 특정 Thread의 과부하를 방지 할 수 있다
 - 결국 같은 얘기
- 세밀한 Multi Thread Programming 없이도 추가 코어를 활용 가능하다
- ~~성능을 올릴 수 있다~~

비동기 프로그래밍의 단점

- 성능이 나빠진다
 - 뒤에서 다시 설명합니다
- CPU / Memory 소모량이 늘어난다
- 코드를 만들기 힘들고
- 디버깅하기 힘들다

그래도 비동기 프로그래밍을 해야 하는 이유

- 유저가 느끼는 퀄리티가 좋아진다
 - 사실 이게 모든 것이라고 봐도 틀리지 않다
- 비동기 프로그래밍이 강제되는 프로그래밍 언어가 있다
- 비동기 프로그래밍이 강제되는 플랫폼이 있다
 - Android, UWP, Etc...

비동기 프로그래밍의 필수 요소

- 호출한 작업을 대신 해 줄 Thread의 존재
- 호출한 작업의 결과를 받아올 방법의 존재
- 위 두가지를 위한 오버헤드 발생 → CPU를 더 많이 먹는다
- 결과를 처리할 시점까지 데이터가 존재해야 한다 → 메모리도 더먹는다

그런데 왜 성능이 더 좋다고?

- Client의 경우

- 대부분의 Platform 은 UI Thread 외에서의 UI 접근이 안전하지 않음(Windows, Android, etc.....). UI Thread는 항상 바쁨
- UI Thread의 부하를 다른 Thread로 넘길 수 있으므로 **사용자가 느끼기에 빨라 보임**

- Server의 경우

- 많은 IO 를 처리 하기 위해 루프에서 매번 체크하던가(non blocking), 아니면 IO 마다 Thread를 생성해서 기다려야 함
- 루프 or Thread Switching 의 부하가 비동기를 구현하기 위한 부하보다 큼

비동기 프로그래밍의 활용 목적

- UI 반응성을 올리기 위해
- Multi Thread 를 지원 하지 않는 언어에서 최대한 성능을 내기 위해
 - Javascript 기반의 Node.js가 대표적
- ~~유행하니까~~
- ~~그냥 멋있어보여서~~

Thread가 없는 언어는?

- 언어 스스로 별도의 코드 흐름을 만들 수 없다
 - 해결책 : 비동기 호출 부분을 별도의 언어로 구현해버린다
- node.js의 경우 libuv를 내장하여 비동기를 구현



비동기 프로그래밍의 방식

- (Windows의 경우) QueueUserAPC 를 호출 하여 대기중인 Thread에 작업을 임의로 할당
- OS에서 제공하는 비동기 IO 방법 이용(사용 용도가 한정적)
- Thread Pool을 직접 구현하여 작업을 실행
- Task 기반 프로그래밍
 - 라이브러리 기반(PPL, TBB)
 - 언어확장 기반 (Cilk+)
- async – await 를 이용한 프로그래밍(C++ 17 예정)

비동기 호출의 결과를 받아오는 방법

- Thread 내 / 외부에 공유하는 Queue에 넣고 나중에 체크한다
 - Windows의 IOCP가 이 방법으로 많이 씁니다.
- Message, Signal, Event 등의 알림을 받아 처리한다
 - 요즘은 잘 안쓰는 방법
- Callback 함수를 사용한다
 - 요즘 일반적인 방법. Node.js 가 이 방법을 사용

Thread를 직접 다루는것은 바보짓?

- 앞으로는 코어가 무한히 늘어날 것이므로 저레벨에서 Thread를 직접 프로그래밍은 바보짓이라는 이야기가 있습니다

병렬화의 이유

- 새로운 반도체 소자가 나오기 전까지는 더 이상 클럭 속도를 높이기 힘들다
- 무어의 법칙의 한계 -
- Multi-core에서 Many-core 시대로 바뀌고 있다.
(3~4년 내에 데스크탑 코어는 60개 까지 늘어날 것이다)
- 빠른 처리속도, 빠른 처리속도, 빠른 처리 속도를 얻기 위해서

- 현실에서는 코어가 전혀 늘어나지 않고 있습니다
 - 2007년 Q6600 4코어 → 2017년 i7-7700K 4코어
 - 클라우드의 VM은 2~4코어를 빌리는 경우가 대부분 많음
 - 스마트폰의 모바일 CPU 도 4코어가 한계
- 물론 코어가 많이 늘어난 하드웨어가 없는것은 아닙니다. 하지만 수평확장이 일반화 된 지금은 흔히 볼수 있는 물건이 아닙니다

Essentials

[Export specifications](#)

Product Collection	Intel® Xeon® Processor E7 v4 Family	Code Name	Products formerly Broadwell
Processor Number	E7-8894V4	Status	Launched
Launch Date ?	Q1'17	Lithography ?	14 nm
Recommended Customer Price ?	\$8898.00		

Performance

# of Cores ?	24	# of Threads ?	48
Processor Base Frequency ?	2.40 GHz	Max Turbo Frequency ?	3.40 GHz
Cache ?	60 MB	Bus Speed ?	9.6 GT/s QPI
# of QPI Links ?	3	TDP ?	165 W

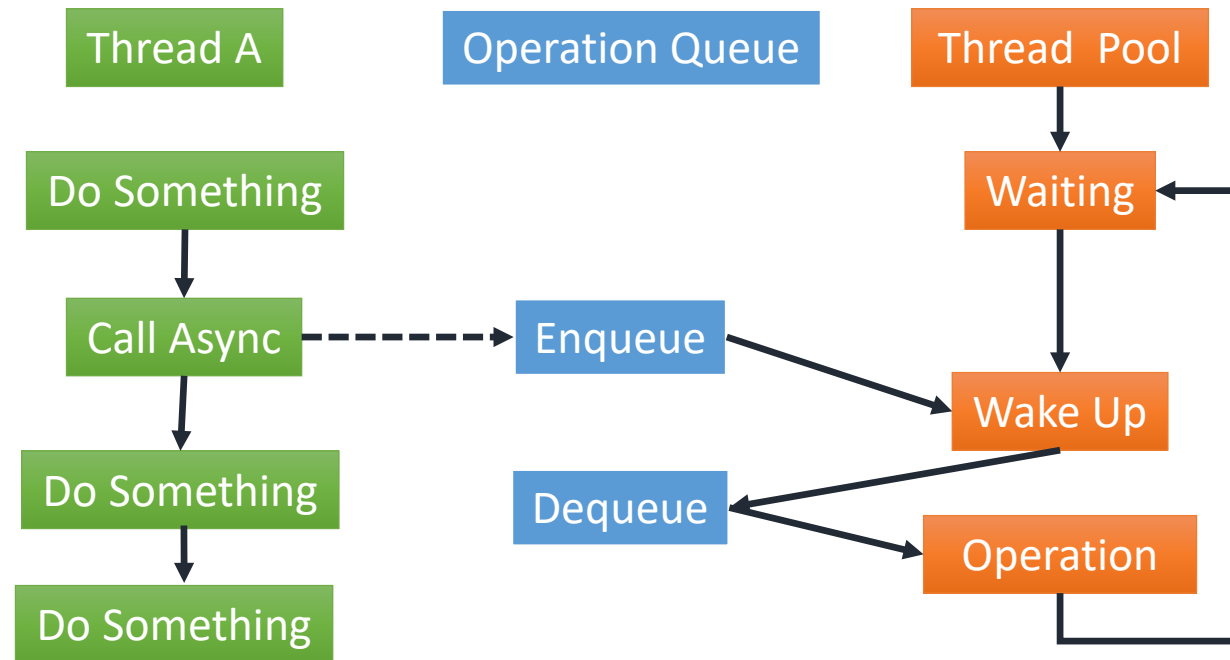
오히려 Task를 위한 잡다한 코드가 없는
비교적 심플한 형태의 코드가 사후디버깅에 좋을 수도 있습니다

C++는 선택지가 굉장히 많습니다

- 어떤 방식이 좋을지는 개인의 선택입니다.
- 유행하는 방법, 코딩이 편한 방법은 분명 존재합니다. 하지만 모든 방법에는 그에 해당하는 단점이 있습니다.
- 방법을 선택할 수 있는 것은 어찌 보면 그런 언어를 사용 하는 사람의 특권이기도 합니다.

Thread Pool을 이용한 구현

- 작업용 Thread를 미리 여러개(혹은 동적으로) 와 Thread 들에게 작업을 전달 하기 위한 Queue를 준비 한 후, 작업을 Queue에 넘기고 Thread들이 놀고있을때마다 꺼내가면서 처리
- 가장 일반적인 구현
- Node.js 가 사용하는 libuv 의 형태가 이런 식



QueueUserAPC

- 특정 Thread에 함수를 호출 하라고 요청
- 해당 Thread 는 Alertable 상태이어야 함
 - 이를 위해 해당 Thread를 Alterable 상태로 만들기 위한 특수 함수 존재
 - 현재 Thread에도 가능함(위험하지만)
- 특정 Thread 를 지정하여 작업을 대행시키는 방법이라 별도의 Thread 관리가 필요
- APC 호출 시 해당 Thread의 Stack을 공유하므로 재귀적으로 현재 Thread에 요청시 (당연하게도) Stack Overflow 의 위험이 있음
- (당연하게도) UI Thread에 오래 걸리는 작업을 요청 할 경우 블록됨

QueueUserAPC

```
void NTAPI asyncall(ULONG_PTR ptr)
{
    printf("Hello C++");
}

int main()
{
    QueueUserAPC(asyncall, GetCurrentThread(), 0);
    SleepEx(1000, TRUE);
    return 0;
}
```

Task 기반 비동기 프로그래밍

- 하나의 함수를 Task 라는 단위로 묶어서 처리하는 방법
- 리턴값까지 처리 해 주는 형태의 구현도 있음
 - 하지만 일반적이지 않음.
- 언어확장 / 라이브러리의 형태의 구현이 많음
- 병렬처리 기능과 병행되는 형태가 많음

Cilk+ 를 이용한 비동기 프로그래밍

- cilk_spawn 키워드로 함수를 호출
- cilk_sync 로 대기
- 전용 컴파일러가 필요하다는 것이 단점

```
void asyncall()
{
    printf("Hello C++");
}

int main()
{
    for (int i = 0; i < 4; ++i)
    {
        cilk_spawn asyncall();
    }
    cilk_sync;
}
```

TBB를 이용한 비동기 프로그래밍

- Cilk+ 를 만든 Intel 의 라이브러리여서 그런지 사용 방법이 비슷
- 비동기 작업이 진행 중인 상태에서 group 이 해제될 경우 문제 발생 - group 객체가 안전한곳에 보관되어야 함

- Library 형태라 멀티플랫폼 지원

```
int main()
{
    tbb::task_scheduler_init inits;
    tbb::task_group group;

    group.run([] { printf("Hello C++\n"); });
    group.run([] { printf("Hello Intel\n"); });
    group.run([] { printf("Hello Linux\n"); });
    group.wait();
}
```


C++ Standard Library를 이용한 비동기 프로그래밍

- C++ 표준에 구현된 방식
- C++11에 구현되었지만 C++14 에서 완성되었다고 보는 편이 좋음
- Thread / Future / Promise 의 합성으로 구현
- 표준이라는 것 외엔 그다지 17에선 개선이 있다고는 하는데

가장 기본적인 형태

- 객체가 3개나 필요하고 thread를 직접 생성해야 하는 불편함이 있음
- `std::packaged_task<T>`를 사용 할수도 있지만... 언급할 가치 없음

```
int main()
{
    using namespace std;

    promise<string> prom;
    future<string> fut = prom.get_future();
    thread thr([&prom](const char* say, const char* language)
    {
        printf("%s %s\n", say, language);
        prom.set_value(string("Seoul"));
    }, "Hello", "World");

    printf("%s", fut.get().c_str());
    thr.join();
}
```

- Thread를 직접 생성하고/ 통로 만들고 하는 과정 생략

```
int main()
{
    using namespace std;

    future<string> myfuture = std::async(std::launch::async,
        [](const char* say, const char* language)
        {
            printf("%s %s\n", say, language);
            return string("Seoul");
        }, "Hello", "World");

    printf("%s", myfuture.get().c_str());
}
```

C++ Standard Library 의 async 는 어떻게 구현되어 있을까요

- 당연히 컴파일러마다 구현은 다르지만... 제가 자주 쓰는 Visual C++ 를 한번 보겠습니다.future 헤더를 보면

```
1  // future standard header
2  #pragma once
3  #ifndef _FUTURE_
4  #define _FUTURE_
5  #ifndef RC_INVOKED
6
7  #ifdef _M_CEE
8      #error <future> is not supported when compiling with /clr or /clr:pure.
9  #endif /* _M_CEE */
10
11 #ifdef _RESUMABLE_FUNCTIONS_SUPPORTED
12     #include <experimental/resumable>
13 #endif /* _RESUMABLE_FUNCTIONS_SUPPORTED */
14
15 #include <functional>
16 #include <system_error>
17 #include <utility>
18 #include <chrono>
19 #include <mutex>
20 #include <condition_variable>
21 #include <ppltasks.h>
22
23 #include <thread>
24
25
```

```

template<class _Rx, bool _Inline>
class _Task_async_state
: public _Packaged_state<_Rx()>
{
    // class for managing associated synchronous state for asynchronous
    // execution from async
public:
    typedef _Packaged_state<_Rx()> _Mybase;
    typedef typename _Mybase::_State_type _State_type;

    template<class _Fty2>
    _Task_async_state(_Fty2&& _Fnarg)
        : _Mybase(_STD forward<_Fty2>(_Fnarg))
    {
        // construct from rvalue function object
        _Task = ::Concurrency::create_task([this]()
        {
            // do it now
            this->_Call_immediate();
        });

        this->_Running = true;
    }

```

PPL을 이용한 비동기 프로그래밍

- Microsoft 의 구현(~~TV에 나오는 제품 간접 광고 아닙니다~~)
- C# 의 Task<T> 와 비슷한 형태
- Task 체인과 리턴값의 처리를 지원
- UWP에 자주 사용(참고자료 : <http://www.slideshare.net/dgtman/f1-c-windows-10-uwp>)

```
int main()
{
    using namespace concurrency;

    auto helloTask = create_task([] { printf("Hello C++\n"); })
        .then([] { printf("Hello Microsoft\n"); })
        .then([] {
            printf("Hello Windows\n"); return "Lyn!";
        });

    printf("%s", helloTask.get());
}
```

그럼 다른 구현은?

- llvm 의 libstd++ 의 구현입니다.

```
template <class _Rp, class _Fp>
future<_Rp>
#ifdef _LIBCPP_HAS_NO_RVALUE_REFERENCES
__make_async_assoc_state(_Fp&& __f)
#else
__make_async_assoc_state(_Fp __f)
#endif
{
    unique_ptr<__async_assoc_state<_Rp, _Fp>, __release_shared_count>
    __h(new __async_assoc_state<_Rp, _Fp>(_VSTD::forward<_Fp>(__f)));
    _VSTD::thread(&__async_assoc_state<_Rp, _Fp>::__execute, __h.get()).detach();
    return future<_Rp>(__h.get());
}
```

지금까지 나온 방법의 공통점

- GC를 지원하지 않는 특성상 Task 작동 중에 Task 관련 객체가 사라질 수 있음
- 따라서 반드시 Task 의 종료시까지 Task 객체를 살려두어야 함
- 비동기 호출의 효율이 떨어지고 사용 가능한 상황이 한정됨
- 그럼 Callback 을 이용하는 방식은 어떨까요?

그럼 Callback 을 사용하는 방식은?

```
getData(function(a){
  getData(a, function(b){
    getData(b, function(c){
      getData(c, function(d){
        getData(d, function(e){
          ...
        });
      });
    });
  });
});
```

C++ 에는 추가적인 이슈도 있습니다

- Task 내에서 Callback 을 할 경우 작업을 요청 한 Thread 와 Callback 이 발생 하는 Thread가 다르기 때문
- 만약 Task의 완료 후 UI의 갱신이 필요하다면? 별도의 Thread 간 통신을 이용하면 UI Thread 로의 Switching 이 필요함
 - 사실 Node.js 도 마찬가지. Callback 을 javascript thread 에서 발생 시켜줘야 하는 이슈가 있음
- 이쯤 되면 대체 뭐가 편한지 의심이 가기 시작합니다

더 좋은 방법은 없나요?

C#의 async - await

- 개인적으로 가장 훌륭한 비동기 호출 구현
- C# 5.0에 추가
- 단 2가지 키워드 (async – await) 로 비동기 호출 구현
- Callback 방식이면서 마치 리턴을 받는 형태처럼 사용

C# 예제 코드

```
namespace ConsoleApplication11
{
    0 references
    class Program
    {
        1 reference
        private static async Task<string> asyncCall()
        {
            Console.WriteLine("Hello C#");
            return "Result";
        }

        1 reference
        private static async void foo()
        {
            string r = await asyncCall();
            Console.Write(r);
        }

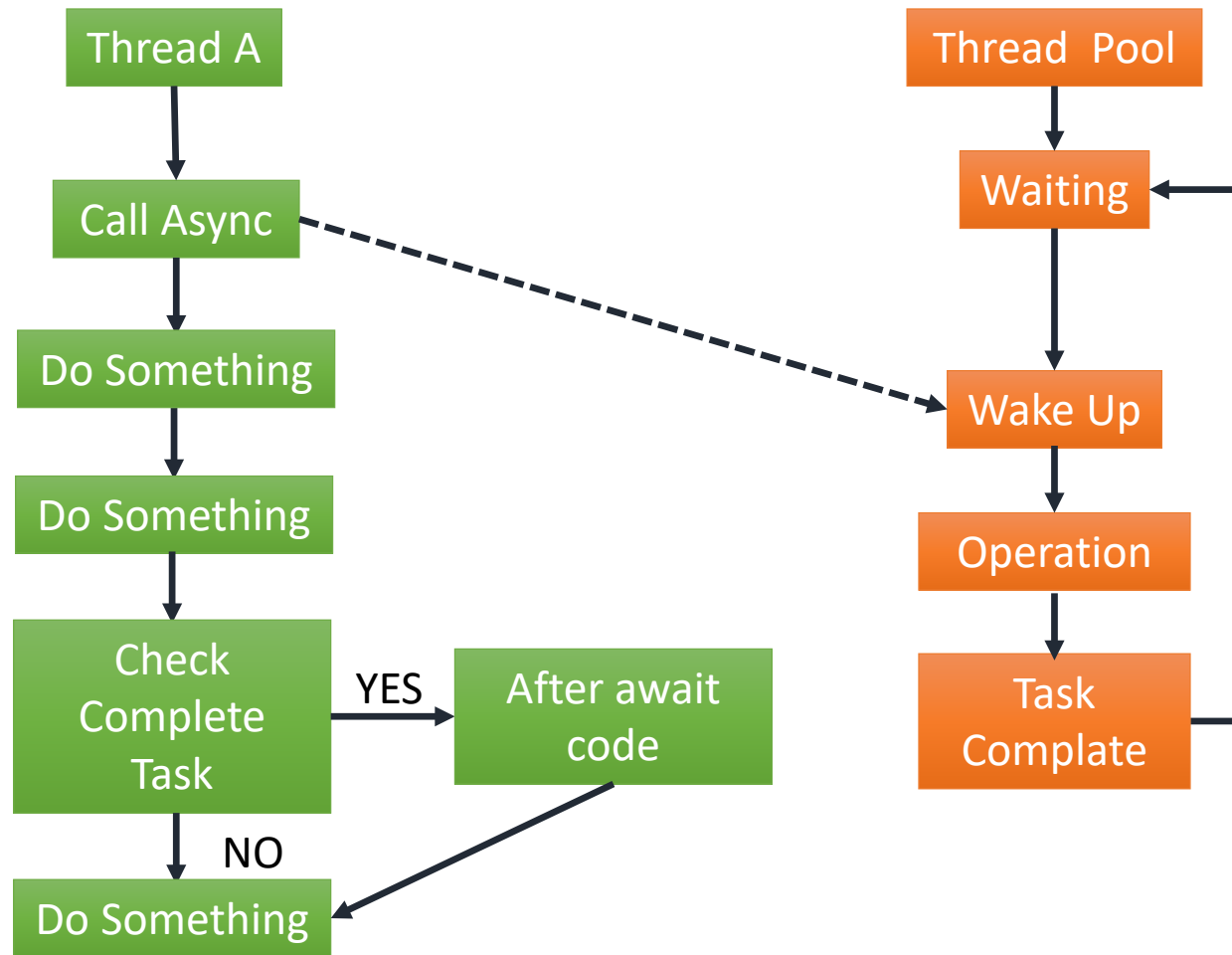
        0 references
        static void Main(string[] args)
        {
            foo();
        }
    }
}
```

```
namespace ConsoleApplication11
{
    0 references
    class Program
    {
        1 reference
        private static async Task<string> asyncCall()
        {
            Console.WriteLine("Hello C#");
            return "Result";
        }

        1 reference
        private static async void foo()
        {
            string r = await asyncCall();

            Console.Write(r);
        }

        0 references
        static void Main(string[] args)
        {
            foo();
        }
    }
}
```



C++ 에도 생깁니다 그런데...

- await / resumable 이라는 키워드의 추가로 지원 예정
- C++ 17 추가 예정(<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3722.pdf>)
- 아직 어떤 식으로 구현 될지는 모릅니다... C++17에 들어갈지 안들어갈지도 모릅니다. ~~사실 안들어갈 것 같습니다~~
- 테스트 해 볼 쓸만한 컴파일러도 없습니다
 - Visual C++도 버전따라 되다 안되다 합니다. 되는 경우도 64bit만 되는 경우도 있습니다
 - C++/CX 에서는 지원 되지만 다른 형태일 가능성도 있습니다

감사합니다