Optimized C++ Study
< Concurrency >

by 권태국 (http://taeguk.me)

Program Without Synchronization

- Event-oriented Programming
- Coroutines
- Message Passing
- Lock-free Programming

Event-oriented Programming

- scheduler가 싱글스레드로 동작하면서 event가 발생(완료)되면, 해당하는 event handler를 호출해준다.
- scheduler가 싱글스레드이므로, 동기화가 필요하지 않다.
- I/O작업의 완료를 기다리는 동안 다른 event들을 처리할 수 있으므로, 자원활용이 용이하다.
- Event Handler가 오래 실행되면, 다른 event들의 처리가 미뤄질 수 있음으로 주의해야한다.
- 보통 GUI Programming을 할 때 UI쪽에서 많이 쓰인다.
- Linux의 epoll, Java의 NIO

Coroutines

- non-preemptive한 multitasking을 위한것이다.
- 여러 개의 task들이 하나의 thread를 쪼개서 실행된다.
- 실행흐름을 explicit하게 다른 task로 위임해줘야한다.
- 내부적으로 싱글스레드이므로 동기화가 필요없다.
- 한개의 task가 오래실행되면, 다른 task들이 실행되지 못하므로, 이 점에 (반응성에) 주의해야한다.

Coroutines Example

```python
async def run(self):
    self._router = self._context.socket(zmq.ROUTER)
    self._router.bind(self._addr)
    Logger().log("slave router bind to {0}".format(self._addr))

    while True:
        msg = await self._router.recv_multipart()
        await self._process(msg)
```

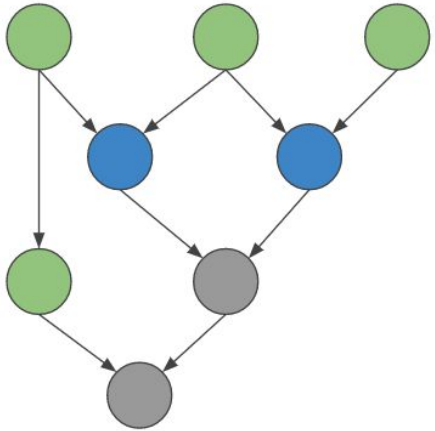
```python
async def run_heartbeat():
    while True:
        expired_slaves, leak_tasks = SlaveManager().purge()
        TaskManager().redo_leak_task(leak_tasks)

        for slave in expired_slaves:
            Logger().log("Expired Slave : {0}".format(str(slave)))

        for slave in SlaveManager().slaves:
            SlaveMessageDispatcher().dispatch_msg(slave, 'heart_beat_req', {})

        await asyncio.sleep(SlaveManager.HEARTBEAT_INTERVAL)
```

Python 예제라서 죄송합니다 ㅠㅜ

Event-oriented Programming **VS** Coroutines

VS

Message Passing



- 실행흐름이 일종의 그래프형태 or 파이프라인의 형태를 띄게 된다.
- 그래프의 각 node들이 (왠만하면 동기화없이) 독립적으로 동작하게 설계한다.
- 단순 multithreaded programming에서는 explicit한 동기화들이 많이 요구된다. message passing에서는 node간의 연결을 통해 implicit하게 동기화가 수행되며, message를 주고받음을 통해 공유상태를 최소화하는 방식으로 explicit한 동기화를 최소화한다.

Lock-free Programming

- 공유자원에 접근할 때 Lock이 필요없다.
- Lock-free 자료구조들은 내부적으로 CAS과 같은 atomic operation들을 활용해서 구현되어 있다.
- 결국 일반 자료구조 보다 더 많은 instruction이 실행되고, atomic한 instruction들은 결국 cpu/memory 내부적으로 동기화가 발생한다!
- 따라서 진정한 ‘lock-free’는 아니다. (thread-unsafe한 일반 자료구조보다는 느리다.)

Remove Code from Startup and Shutdown

- main() 함수 호출전에 수행되어야하는 staic variable들의 초기화같은 부분들을 가능하면 없애라.
- 큰 프로그램의 경우, 이러한 점들이 startup시간에 큰 영향을 줄수도 있다.

- 근데... 이게 concurrency랑 무슨 상관인거죠…?

Make Synchronization More Efficient

- Reduce the Scope of Critical Sections.
- Limit the Number of Concurrent Threads.
- Avoid the Thundering Herd.
- Avoid Lock Convoys.
- Reduce Contention.
- Don’t Busy-Wait on a Single-Core System.
- Don’t Wait Forever.
- Rolling Your Own Mutex May Be Ineffective.
- Limit Producer Output Queue Length.

Reduce the Scope of Critical Sections.

- 임계 영역은 작을수록 좋다. (당연한 소리…)
- I/O 작업은 임계영역 밖에서 하자. (Ex, 출력할 공유데이터를 미리 비공유 공간에 저장한 뒤, 임계영역 밖으로 빠져나와서 출력하기.)

Limit the Number of Concurrent Threads.

- Runnable threads의 개수는 processor core 수보다 작거나 같아야한다. (전체 thread 개수가 아니라 runnable의 개수다.)
- 이유 1. context switching의 overhead를 없애기위해.
- 이유 2. lock을 제때제때 release하고 acquire하지 못할 수 있다. (OS thread scheduler 때문에)
  - 보통 mutex는 (일정 시간동안의 spinlock) 를 수행하고, 그래도 lock을 취득하지 못하면, OS의 waiting queue로 들어가게 된다.
  - 만약, lock을 취득하고 있는 thread가 time sliced가 다되서, running queue에 들어가 있고, lock을 요구하는 thread가 spinlock을 수행하고 있으면, spinlock시간동안 lock은 절대 반환될 수 없다.
- The ideal number of threads contending for a brief critical section is two.

Avoid the Thundering Herd.

- 하나의 이벤트가 발생했을 때, 여러 개의 thread가 wakeup되면서, 한개를 제외한 thread들은 어차피 이벤트를 처리하지도 못하는데도 괜히 wakeup해가지고 다시 sleep해야하는 문제..
- Event 발생 시 thread를 애초에 하나만 깨우면 이러한 문제가 생기지 않겠지만, 상황이 복잡하여 thread를 여러 개 깨워야하는 경우 이러한 문제점이 발생할 수도 있을 것 같다.
- Windows에서 Manual Reset 모드의 Event를 여러 개의 thread가 기다릴 경우, 발생할 수 있다.
- 과거 Linux Kernel에서의 사례 : http://www.citi.umich.edu/projects/linux-scalability/reports/accept.html

Avoid Lock Convoys.

- time quantum 동안 여러번의 lock acquirement가 발생할 수 있고 lock을 붙잡고 있는 시간이 작은 경우에, 같은 우선순위를 가진 여러 개의 thread들이 동시에 lock을 두고 경쟁함에 따라 발생하는 문제.
- 문제점 1 : Context Switching이 너무 자주 발생해서 성능이 안좋아짐. (lock convoy에 속한 thread들이 time quantum을 다 소모하지 못하고 계속 blocking 되므로)
- 문제점 2 : Lock Convoy에 포함되지 않은 다른 thread들의 cpu 점유률이 높아짐. (공평성의 문제)
- https://blogs.msdn.microsoft.com/sloh/2005/05/27/lock-convoys-and-how-to-recognize-them/
- http://www.cnblogs.com/cobbliu/archive/2012/03/10/2388874.html

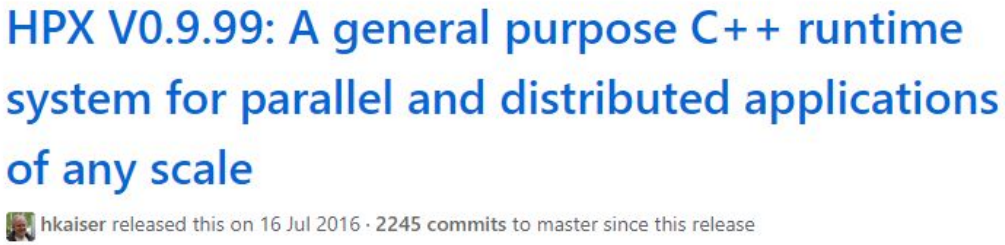
Reduce Contention.

- Be aware that memory and I/O are resources.
  - **Dynamically allocated memory. (Memory Manager에 대한 동시 호출은 동기화 됨에 유의해야 한다.)**
  - File I/O, Network I/O
- Duplicate resources.
- Partition resources.
- Fine-grained locking.
- Lock-free data structures.
- Resource scheduling.

ETC..

- Don’t Busy-Wait on a Single-Core System.
- Don’t Wait Forever.
  - 특정 조건을 무작정 계속 기다리는 것은 에러상황 발생 시 문제가 있을 수 있다. (ex, 특정 조건을 만들어주는 어떤 thread가 갑자기 죽은 경우, 조건을 기다리는 thread는 영원히 wait하게 된다.)
- Rolling Your Own Mutex May Be Ineffective.
  - OS나 아키텍처마다 그리고 상황마다 Mutex의 효율적인 구현이 달라질 수 있으므로, 자기만의 mutex를 구현하는 것은 특정 플랫폼의 특정 상황 외에는 효율적이지 않을 수 있다.
- Limit Producer Output Queue Length.
  - Producer가 자기혼자 컴퓨팅자원들을 독점하면서 생산만 계속하고, consumer는 그로인해 소비를 하지 못하고 있는 것은 ‘concurrency 하지 않다.’
  - **The queue only needs to be big enough to smooth any variation in the consumer’s performance.**

HPX 소개



- https://github.com/STEllAR-GROUP/hpx

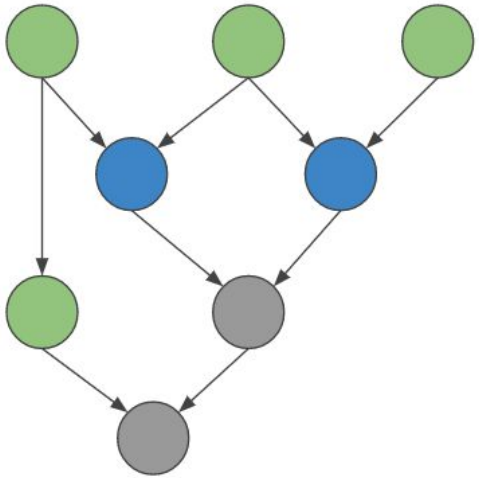
Future / Async

- Concurrency TS의 기능들도 구현되어 있음.
- Distributed 까지 확장.

```
hpx::future<std::uint64_t> fibonacci_future(std::uint64_t n)
{
    // if we know the answer, we return a future encapsulating the final value
    if (n < 2)
        return hpx::make_ready_future(n);
    if (n < threshold)
        return hpx::make_ready_future(fibonacci_serial(n));

    // asynchronously launch the creation of one of the sub-terms of the
    // execution graph
    hpx::future<std::uint64_t> f =
        hpx::async(&fibonacci_future, n-1);
    hpx::future<std::uint64_t> r = fibonacci_future(n-2);

    return hpx::async(&add, std::move(f), std::move(r));
}
```

Dataflow

- 데이터의 흐름으로서 병렬 알고리즘을 표현할 수 있다.
- Intel TBB의 tbb::flow 와 비슷한 기능.



```
RandIter mid = first;
std::advance(mid, mid_point);

hpx::future<RandIter> left = execution::async_execute(
    policy.executor(), *this, policy, first, mid,
    mid_point, f, proj, chunks);
hpx::future<RandIter> right = execution::async_execute(
    policy.executor(), *this, policy, mid, last,
    size - mid_point, f, proj, chunks);

return
    dataflow(
        policy.executor(),
        [mid](
            hpx::future<RandIter> && left,
            hpx::future<RandIter> && right
        ) -> RandIter
        {
            if (left.has_exception() || right.has_exception())
            {
                std::list<std::exception_ptr> errors;
                if(left.has_exception())
                    hpx::parallel::util::detail::
                    handle_local_exceptions<ExPolicy>::call(
                        left.get_exception_ptr(), errors);
                if(right.has_exception())
                    hpx::parallel::util::detail::
                    handle_local_exceptions<ExPolicy>::call(
                        right.get_exception_ptr(), errors);

                if (!errors.empty())
                {
                    throw exception_list(std::move(errors));
                }
            }
            RandIter first = left.get();
            RandIter last = right.get();

            std::rotate(first, mid, last);

            // for some library implementations std::rotate
            // does not return the new middle point
            std::advance(first, std::distance(mid, last));
            return first;
        },
        std::move(left), std::move(right));
```

Task Blocks

- N4411 을 기반으로 한 Task Blocks 구현. (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf)

```
template <typename Func>
int traverse(node& n, Func && compute)
{
    int left = 0, right = 0;
    define_task_block(
        [&](task_block<>& tr) {
            if (n.left)
                tr.run([&] { left = traverse(*n.left, compute); });
            if (n.right)
                tr.run([&] { right = traverse(*n.right, compute); });
        });

    return compute(n) + left + right;
}
```

Executors

- P0443R0/P0443R2를 구현(중) (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0443r0.html)
- 뭔가를 실행할 때 실행하는 방법/장소등을 담당하는 녀석을 executor라고 부른다.
- 기존의 C++ facility들과 함께 사용가능.
- **고도의 추상화 및 customization을 가능하게 함.**

```
class logging_executor
{
  public:
    logging_executor(logging_context& ctx) : context_(ctx) {}

    bool operator==(const logging_executor& rhs) const noexcept
    {
      return context() == rhs.context();
    }

    bool operator!=(const logging_executor& rhs) const noexcept
    {
      return !(*this == rhs);
    }

    const logging_context& context() const noexcept
    {
      return context_;
    }

    template<class Function>
    void execute(Function&& f) const
    {
      context_.log("executing function");
      f();
    }

  private:
    mutable logging_context& context_;
};
```

Parallel Algorithm (Parallel STL)

- C++17에 추가된 parallel STL들을 구현. (현재 몇 개를 제외한 대부분의 알고리즘들이 구현되어 있음. 남은 알고리즘들을 8월 말까지 다 구현하는 게 제 목표입니다.)
- 차세대 표준인 Ranges TS, Executors 등을 고려하여 구현.
- Distributed 환경까지 확장. (분산 자료구조에 대해)

```
hpx::parallel::sort(execution::par, c.begin(), c.end());
```

```
-------------- Benchmark Config --------------
seed            : 19960202
vector_size     : 1000000000
rand_fill range : 100000
base_num        : 50000
iterator_tag    : random
test_count      : 10
os threads      : 16
----------------------------------------------

* Preparing Benchmark...
tcmalloc: large alloc 4000006144 bytes == 0x1592000 @
tcmalloc: large alloc 4000006144 bytes == 0xefcb4000 @
* Running Benchmark...
--- run_partition_benchmark_std ---
--- run_partition_benchmark_seq ---
--- run_partition_benchmark_par ---
--- run_partition_benchmark_par_unseq ---

-------------- Benchmark Result --------------
partition (std) : 4.37529(sec)
partition (seq) : 4.56043(sec)
partition (par) : 0.371446(sec)
partition (par_unseq) : 0.368614(sec)
----------------------------------------------
```

Distributed Programming

- Concurrency & Parallel 을 위한 기능들의 상당수를 Distributed 환경에서도 쓸 수 있다.

```
hpx::partitioned_vector<T> c(size, policy);
iota_vector(c, T(1234));

const T v(42);
hpx::parallel::fill(fill_policy, c.begin(), c.end(), v);
```

```
hpx::future<std::uint64_t> fibonacci_future(std::uint64_t n)
{
    // if we know the answer, we return a future encapsulating the final value
    if (n < 2)
        return hpx::make_ready_future(n);
    if (n < threshold)
        return hpx::make_ready_future(fibonacci_serial(n));

    fibonacci_future_action fib;
    hpx::id_type loc1 = here;
    hpx::id_type loc2 = here;

    if (n == distribute_at) {
        loc2 = get_next_locality(++next_locality);
    }
    else if (n-1 == distribute_at) {
        std::uint64_t next = next_locality += 2;
        loc1 = get_next_locality(next-1);
        loc2 = get_next_locality(next);
    }

    hpx::future<std::uint64_t> f = hpx::async(fib, loc1, n-1);
    hpx::future<std::uint64_t> r = fib(loc2, n-2);

    return hpx::when_all(f, r).then(when_all_wrapper());
}
```

ETC...

- Support running HPX on top of MPI.
- Support Coroutines TS.
- Support percolation via CUDA & OpenCL.

```
hpx::future<int> fib1(int n)
{
    if (n >= 2)
        n = co_await fib1(n - 1) + co_await fib1(n - 2);
    co_return n;
}
```

HPX Summary

- Thread 생성 및 관리, 스케쥴링 / Distributed Node 간의 통신 등의 핵심 부분들을 담당하는 HPX Runtime System을 구현.
- future, dataflow, channel, lock 등 concurrency/parallel/distributed programming을 위한 utility들을 구현.
- HPX Runtime System 위에서 concurrency & parallel에 관련된 C++ 차세대 표준/proposal 들을 구현.
- 그리고 그러한 C++표준들을 확장. (예컨데, distributed 환경으로의 확장등.)

Welcome To Contribute

- (C++만 익숙하다면) 쉬운 일자리가 많습니다.
- Pull Request에 대해 관대합니다. (깐깐하지 않아서 생각보다 쉽게 merge될 수 있습니다.)
- 친절한 Community : IRC에 질문을 올리면 친절하게 답변해줍니다.
- C++ 현재/차세대 표준을 공부하고 직접 구현할 수 있는 기회.
- Template meta programming / Generic Programming 에 익숙해질 수 있습니다. (너무 극혐..)
- C++ 덕후들의 코드베이스를 공부할 수 있습니다.
- 때때로 코딩하는 시간보다 컴파일에러 잡는 시간이 더 깁니다…(template의 늪...ㅜㅜ)

<END>