Building C++

Architecting systems with modern C++

Work Contracts - a simple, wait free, asynchronous task management system

`https://www.buildingcpp.com/documents/work_contracts`

`https://github.com/buildingcpp/system.git`

The basics:

A `work_contract` represents a repeatable task which can be executed an arbitrary number of times. The execution of the task is performed asynchronously and is guaranteed to be thread safe.

`work_contract`s are associated with a parent `work_contract_group` and are created using `work_contract_group::create_contract()`.

To exercise a `work_contract` it must first be 'invoked'. Invoking a work contract is done using `work_contract::invoke()`. This flags the `work_contract` as ready to be executed. Executing an invoked `work_contract` is achieved by worker threads which call `work_contract_group::execute_next_contract()` on the `work_contract_group` which created that `work_contract`. This worker thread is then responsible for executing the task associated with that `work_contract`. Once the task has been executed the `work_contract` returns to the non-invoked state.

The following demonstrates the creation of a `work_contract_group`, a `work_contract`, invoking the `work_contract` and executing that `work_contract`. For the sake of simplicity, the main thread acts as the worker thread.

```
#include <library/system.h>
#include <iostream>


void foo(){std::cout << "foo\n";}


int main()
{
    using namespace bcpp::system;

    static auto constexpr max_work_contracts = (1 << 20);
    work_contract_group workContractGroup(max_work_contracts);
    auto workContract = workContractGroup.create_contract(foo);

    workContract.invoke();
    workContractGroup.execute_next_contract();

    return 0;
}
```

[output]

```
foo
```

`work_contract`s can also optionally define a one-shot surrender task. As with the primary task, a `work_contract`'s surrender task is performed asynchronously and is guaranteed to be thread safe. Surrender tasks are also executed by worker threads which call `work_contract_group::execute_next_contract()`.

Furthermore, the surrender task is guaranteed to be the final task for the `work_contract`. Once invoked, neither the surrender task, nor the primary work task shall ever be executed again. It is accurate to think of the surrender task as the 'destructor' for the `work_contract`. The surrender task can be explicitly invoked via `work_contract::surrender()` or, optionally, when the `work_contract` is destroyed.

```
#include <library/system.h>
#include <iostream>


void foo(){std::cout << "foo\n";}
void bar(){std::cout << "bar\n";}


int main()
{
    using namespace bcpp::system;

    static auto constexpr max_work_contracts = (1 << 20);
    work_contract_group workContractGroup(max_work_contracts);
    auto workContract = workContractGroup.create_contract(foo, bar);

    workContract.invoke();
    workContractGroup.execute_next_contract();

    workContract.surrender();
    workContractGroup.execute_next_contract();

    return 0;
}
```

[output]

```
foo
bar
```

Simple example #1:
The basics

The following example demonstrates the basics of creating work contracts and executing them. However, since the main thread is both invoking and executing the work contract it is not particularly useful.

Work contracts are intended to be a BYOT (bring your own threads) system. This design allows the user to retain complete control over which thread, which cpu and when a work contract is actually executed.

```
#include <library/system.h>
#include <iostream>


int main()
{
    // create a work_contract_group
    static auto constexpr max_contracts = (1 << 20);
    bcpp::system::work_contract_group workContractGroup(max_contracts);

    // create a work_contract
    auto workContract = workContractGroup.create_contract(
            [](){std::cout << "contract executed\n";},
            [](){std::cout << "contract surrendered\n";});

    // invoke the work_contract
    workContract.invoke();

    // execute invoked work_contract
    workContractGroup.execute_next_contract();

    // surrender the work_contract
    workContract.surrender();

    // execute invoked work_contract
    workContractGroup.execute_next_contract();
}
```

[output]

```
contract executed
contract surrendered
```

Simple example #2:
work contracts are repeatable

Work contracts are not “tasks” in the traditional sense. Work contracts can be executed many times.

```
#include <library/system.h>
#include <iostream>


int main()
{
    // create a work_contract_group
    static auto constexpr max_contracts = (1 << 20);
    bcpp::system::work_contract_group workContractGroup(max_contracts);

    // create a work_contract
    auto counter = 0;
    auto workContract = workContractGroup.create_contract(
            [&](){std::cout << "contract executed: “ << ++counter << "\n";},
            [](){std::cout << "contract surrendered\n";});

    for (auto i = 0; i < 5; ++i)
    {
        // invoke the work_contract
        workContract.invoke();

        // execute invoked work_contract
        workContractGroup.execute_next_contract();
    }

    // surrender the work_contract
    workContract.surrender();

    // execute invoked work_contract
    workContractGroup.execute_next_contract();
}
```

[output]

```
contract executed: 1
contract executed: 2
contract executed: 3
contract executed: 4
contract executed: 5
contract surrendered
```

Simple example #3:

invoking an already invoked work contract does nothing

A work contract is in one of two states. It is either invoked or it is not. Once a work contract is executed this state transitions from invoked to not invoked. Repeated invocations of an already invoked work_contract does nothing to update its state once it is already in the invoked state.

The following example invokes the work contract multiple times prior to executing the work contract. The result is a single execution of the work function.

```
#include <library/system.h>
#include <iostream>


int main()
{
    // create a work_contract_group
    static auto constexpr max_contracts = (1 << 20);
    bcpp::system::work_contract_group workContractGroup(max_contracts);

    // create a work_contract
    auto counter = 0;
    auto workContract = workContractGroup.create_contract(
            [&](){std::cout << "contract executed: " << ++counter << "\n";},
            [](){std::cout << "contract surrendered\n";});

    for (auto i = 0; i < 5; ++i)
        workContract.invoke();  // invoke the work_contract repeatedly

    // execute invoked work_contract
    workContractGroup.execute_next_contract();

    // surrender the work_contract
    workContract.surrender();

    // execute invoked work_contract
    workContractGroup.execute_next_contract();
}
```

[output]

```
contract executed: 1
contract surrendered
```

Simple example #4:

The surrender routine:

- The surrender routine is executed asynchronously.
- The surrender routine is executed exactly once.
- Once surrendered, a work contract can no longer by invoked nor re-surrendered.
- A work contract which is surrendered while in the invoked but non-executed state will be have its surrender routine executed next.
- A work contract can be explicitly surrendered via `work_contract::surrender()` or when the work_contract is destroyed.

```
#include <library/system.h>
#include <iostream>


int main()
{
    // create a work_contract_group
    static auto constexpr max_contracts = (1 << 20);
    bcpp::system::work_contract_group workContractGroup(max_contracts);

    // create a work_contract
    auto workContract = workContractGroup.create_contract(
            [](){std::cout << "contract executed\n";},
            [](){std::cout << "contract surrendered\n";});

    // invoke the work_contract
    workContract.invoke();

    // surrender the work_contract before executing it
    workContract.surrender();

    // execute invoked work_contract
    workContractGroup.execute_next_contract();
}
```

[output]

```
contract surrendered
```

Example #5:

Where's the async?!

So far, for the sake of clarity, the examples have not been asynchronous. The work contract system is a "bring your own threads" system. This is done for the following reasons:

- User can determine which threads will do the work of executing and surrendering contracts.
- User can determine when threads will do this work
- User can determine how many threads will do this work

Note: the work contract system is lock free and thread safe. Therefore more than one thread can invoke a `work_contract_group::execute_next_contract()` simultaneously thereby processing more than one work contract simultaneously.

NOTE: A work contract's work routine is guaranteed to be executed by only one thread at a time regardless of how many threads invoke `work_contract_group::execute_next_contract()` simultaneously.

```
#include <library/system.h>
#include <iostream>

int main()
{
    // create a work_contract_group
    static auto constexpr max_contracts = (1 << 20);
    bcpp::system::work_contract_group workContractGroup(max_contracts);

    // create a worker thread do to the asnyc work
    std::jthread workerThread([&](auto const & s)
            {while (!s.stop_requested())
            workContractGroup.execute_next_contract();});

    // create a work_contract
    auto workContract = workContractGroup.create_contract(
            [](){std::cout << "contract executed\n";},
            [](){});

    // invoke the work_contract
    workContract.invoke();

    // just a demo so give the worker a chance to do the work
    std::this_thread::sleep_for(std::chrono::seconds(1));
}
```

[output]

```
contract executed
```

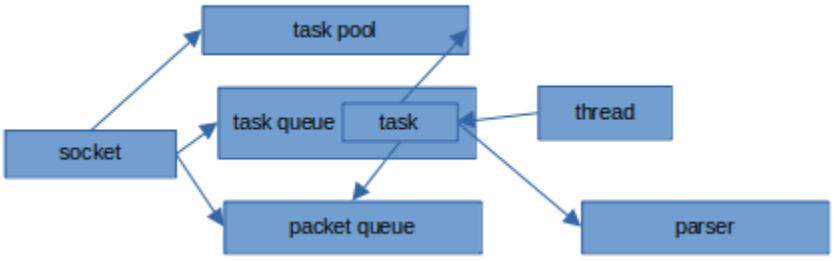
Architecture, better designs and motivations for work contracts:

The work contract system contains no queues. It is not intended to be used with numerous short lived or one time tasks. It is intended to provide a simple way to connect components with a 'set it and forget it' mentality such that one component can easily trigger asynchronous activity in another component. It is a system which encourages separation of concerns, push based and event driven architectures. It encourages a 'tell, don't ask' approach to designing software.

As an example, consider a system consisting of two components: a socket (which receives packets) and a parser (which parses the data within those packets). The two components are connected by a queue which contains packets received by the socket but which have not yet been parsed by the parser.

A naive 'task queue' centric approach might be to create a base 'task' class with virtual functions. Then create a 'parse task' which implements those virtual functions to pop a packet from the queue and then invoke `parser.parse(packet)`. This approach could easily require:

- the implementation of task classes
- repeated new/delete of those tasks or otherwise require a pool and allocator
- tasks queues; often containing tasks of many kinds - not just the task at hand.
- synchronization for the queues
- encourages a design where the task needs to know how to find the correct parser
- likely can produce one 'task' per packet
- offers no protection from multiple 'task's being processed on the same parser simultaneously



The above diagram attempts to sketch out the relationships between the various components. Arrows indicate which objects require knowledge of other objects. For instance, the socket would need to know (somehow) about the task pool to efficiently allocate a 'parse task'. It would also need to know about the task queue to enqueue the task as well as the packet queue itself. The task would require knowledge about the packet and parser at execution time. etc.

Admittedly, the above example is a bit of a straw man but such designs are not entirely uncommon. Die instance, it could be argued that the packet is placed within the task thereby eliminating the packet queue entirely. But this would create an even more specialized task as well as mandate the the 'task queue' now be sufficiently sized to contain as many packets as might be expected. It would also further enforce the idea of one task per packet.

Again, the above example could be viewed as a straw man but such designs are not entirely uncommon.

In contrast, work contracts encourage complete separation of concerns between the socket and the parser. Neither requires knowledge of the other.

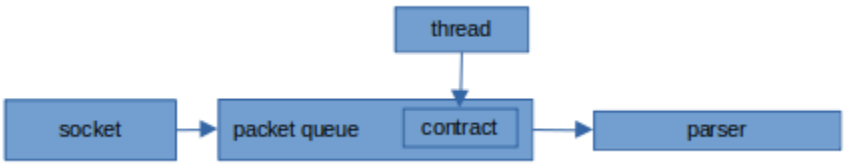
It is very likely that the socket and parser are long lived objects and, furthermore, the relationship between these two objects is very deterministic. That is, we know at design time that when a packet is placed in the queue the parser should remove the packet and parse it. Work contracts are an excellent replacement for the above (admittedly) naive solution.

In a work contract based solution the socket and the parser are completely decoupled. The socket simply pushes packets into the provided queue. It has no concern for when, where, who, nor what the fate of those packets might be.

The parser creates the queue and places within it a work contract which is invoked (note: not executed) whenever the queue is not empty.

This work contract, when executed, then dequeues a packet and calls parser.parse(packet).

Such a design would look like this:



- The socket does not create tasks
- The socket has does not require knowledge of the system beyond the queue
- the queue knows nothing of who enqueues packets
- invoking the work contract requires nothing more than an atomic compare/swap of a single bit
- the work contract ensures single threaded execution and therefore ensures the thread safe use of parser.parse(packet).

Where’s the beef?!

A simple demo of this entire system can be implemented with a single page of code:

```
using namespace bcpp::system;

struct packet{};
struct queue
{
    queue(work_contract workContract):workContract_(std::move(workContract)){}
    void push(packet p){queue_.push(p); workContract_.invoke();}
    packet pop(){auto p = queue_.front(); queue_.pop(); return p;}
    work_contract workContract_;
    std::queue<packet> queue_;
};
struct parser
{
    parser(work_contract_group & wcg):
    queue_(std::make_shared<queue>(wcg.create_contract([this](){parse();}))){}
    void parse(){auto packet = queue_->pop(); std::cout << "parsing packet\n";}
    std::shared_ptr<queue> queue_;
};
struct socket
{
    socket(std::shared_ptr<queue> q):queue_(q){}
    void receive(){std::cout << "receiving packet\n"; queue_->push(packet{});}
    std::shared_ptr<queue> queue_;
};
int main()
{
    work_contract_group wcg(256);
    parser p(wcg);
    socket s(p.queue_);
    std::jthread workerThread([&](std::stop_token stopToken)
        {while (!stopToken.stop_requested()) wcg.execute_next_contract();});
    s.receive();
    std::this_thread::sleep_for(std::chrono::seconds(1));
    workerThread.request_stop();
    return 0;
}
```

[output]

```
receiving packet
parsing packet
```

Performance, scheduling and fairness:

The work contract system is lock free. At its core, a `work_contract_group` manages a collection of work_contracts using a binary heap structure with the work_contracts at the leaf nodes. Each non-leaf node is an atomic count representing how many 'child' leaf nodes are in the 'invoked' state. Invoking a work contract involves traversing the binary tree from the leaf to the root, incrementing the count along the way. Similarly, locating an 'invoked' work contract involves navigating from the root to a leaf node while decrementing the count along the way.

Therefore, both invoking and executing a work contract each require log^2(n) successful atomic compare exchange calls to complete. Contention arises toward the root of the tree where it is more likely that multiple threads are competing to adjust the same atomic counters.

Each call to `work_contract_group::execute_next_contract()` increments an 'inclination' counter who's bits are used to create a preference for choosing to traverse to a left or right child node in the case where both child nodes have non zero counts. This monotonically increasing 'inclination' counter ensures that all leaf nodes will be fairly selected and that starvation can not occur. To increase fairness, work contracts are inserted into the binary heap in such a way as to create a more balanced binary heap than that which existed prior to the work contract's insertion into the heap. However, work contracts can be surrendered which remove them from the binary heap and rapid removal of work contracts could, if unfortunately selected, lead to a significantly unbalanced binary heap. Future work will involve identifying when the struct is significantly unbalanced and will balance as needed by relocating work contracts within the heap as they are executed. Re-balancing is a very lightweight process and should not be required very often given the longevity of work contracts and because it is unlikely that the binary heap will unbalanced in general.

With regards to performance the main goals of the work contract system are:

- high throughput
- extremely reliable fairness

To measure these the throughput a benchmark was created which times how many times a trivial task can be invoked within a fixed duration of time. The benchmark runs 10 times. Each run increases the number of worker/consumer threads by one. Each task is immediately re-scheduled as the task in executed. Additionally, each unique task is associated with a counter which tracks how many times that unique task has been executed during the test. This metric is used to calculate the mean number of task executions, the standard deviation as well as coefficient of variation. A lower coefficient of variation indicates that tasks were executed more evenly (fairly). A higher coefficient of variation indicates that some tasks were executed more often (less fairly) than other tasks.

The benchmark compares the results for the tests using work contracts as well more a traditional mpmc task queue design. In this case, ConcurrentQueue from MoodyCamel is used to provide a well documented and efficient lock free mpmc queue to compare against work contracts.

The test runs for 10 seconds per pass, increases the number of worker threads by one with each pass and operates on 256 concurrently active tasks throughout each pass. The test machine is a AMD Ryzen 9 3900 12-Core Processor. Hyper-threading has been disabled and each worker thread has its cpu affinity set to a unique cpu.

The task itself is designed to consume only a small amount of CPU time (around 1 usec) and avoid using any resources which could cause cache contention.

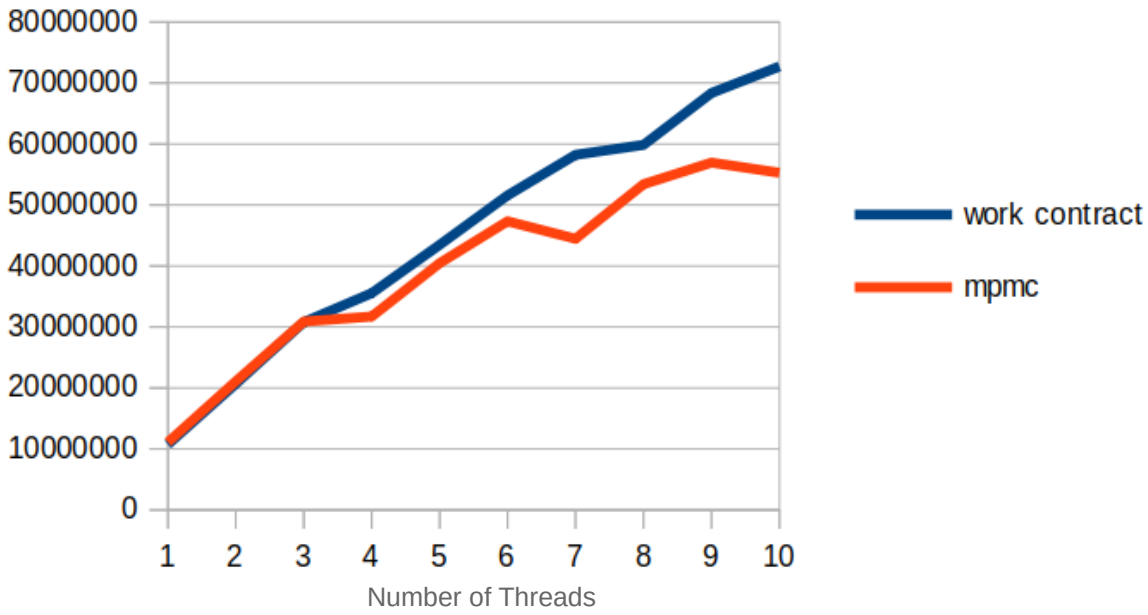
The task:

```
std::int32_t work_function()
{
    auto t = 0;
    for (auto i = 0; i < (1 << 8); ++i)
        t += std::to_string(i).size();
    return t;
};
```

work contracts					
thread count	total tasks	tasks per thread per second	mean	standard deviation	coefficient of variation
1	10,734,100	1,073,400	41,930	0.367	0.000
2	20,665,131	1,033,246	80,723	4,545.304	0.056
3	30,784,074	1,026,128	120,250	1,346.214	0.011
4	35,543,546	888,582	138,841	1,762.597	0.013
5	43,474,273	869,478	169,821	3,906.835	0.023
6	51,572,160	859,528	201,453	2,325.401	0.012
7	58,215,048	831,636	227,402	3,879.622	0.017
8	59,854,444	748,174	233,806	3,281.332	0.014
9	68,355,238	759,495	267,012	2,954.879	0.011
10	72,702,103	727,014	283,992	2,851.247	0.010

MPMC queue *(using MoodyCamel ConcurrentQueue)*					
thread count	total tasks	tasks per thread per second	mean	standard deviation	coefficient of variation
1	11,088,294	1,108,819	43,313	43,398	1.002
2	21,081,395	1,054,061	82,349	58,172	0.706
3	30,888,815	1,029,620	120,659	137,082	1.136
4	31,717,938	792,943	123,898	118,731	0.958
5	40,462,459	809,243	158,056	152,179	0.963
6	47,324,498	788,736	184,861	178,127	0.964
7	44,450,647	635,004	173,635	164,720	0.949
8	53,400,419	667,501	208,595	197,903	0.949
9	56,937,117	632,630	222,410	210,336	0.946
10	55,269,915	552,695	215,898	198,539	0.920

Tasks per second for 256 concurrent repeating tasks:



Fairness: Coefficient of variation for 256 concurrent repeating tasks *(smaller is better):*