Dr. Dobb's Journal

February 2013

Task-Based Programming in Windows

Setting up a reliable parallel execution context

Next

ALSO INSIDE

UBM Tech

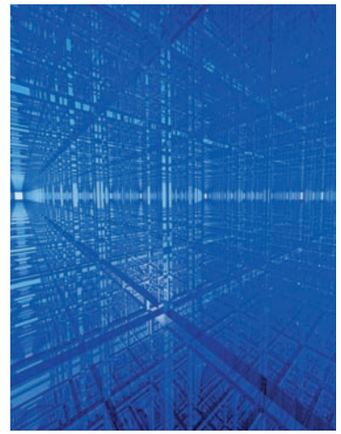
www.drdobbs.com

Previous Next

Dr. Dobb's Journal

CONTENTS

February 2013



FEATURES

GUEST EDITORIAL

More on DrDobbs.com

The Rise and Fall of Languages in 2012

The C++ renaissance didn't materialize, Java survived intact, and Objective-C kept on rolling. What else happened?

http://www.drdobbs.com/240145800

New Year's Resolution: No More Bugs

Someone once told Al Williams that his job (and all his colleagues) should be eventually eliminated because they were an unnecessary expense and it would be cheaper to just never have any failures.

http://www.drdobbs.com/240145857

Creating an Open Source Project

Creating an open source project can generate opportunities for everyone involved; more so than if the software sits dormant on a hard drive somewhere. But where do you start?

http://www.drdobbs.com/240145389

Isolating A Superbug

How could it possibly be that concatenating `preamble` and `output` would always work with one strategy but not another?

http://www.drdobbs.com/240145442

Review of Xamarin.Mac Professional

While a good IDE for porting Xamarin MonoTouch code (iOS-based C#) to the Mac, the long learning curve and poor documentation make it difficult to port code from Windows to the Apple desktop.

http://www.drdobbs.com/240145733

Mailbag

Rethinking copyright and more on methodologies

Unlocking Articles

In response to the editorial "The Crying Need to Punish Cyber Crime Fairly" (www.drdobbs.com/web-development/240146300), which discussed the excessively harsh penalties imposed for cyber crimes, we received this original suggestion regarding the initiating issue, which was the presence of articles written at taxpayer expense but locked behind a paywall:

"I propose that free-agent authors have the right to assign the rights to their work exclusively only for a very limited period, say two years, renewable for another term at both the authors' and assignees' option, with terms sequentially renewed for as long as both shall agree. Such assignment may come with whatever strings the parties agree to; particularly, an author may forbid re-assignment of various or any rights. Non-exclusive assignment may be done for a substantially longer term, perhaps even in perpetuity. Such exclusive assignments become non-exclusive at the end of the assignment term or following the death of the author.

This restriction on the 'right' of an author is really a restriction on assignees. Publishers, and newer forms thereof like patent trolls and JSTOR, need to be regulated just like any other public utility, and this is a counterbalance to their current overweening power.

Copyright law can remain substantially in force with the proviso that only the author or current assignees have standing. Protection from plagiarism and copying code is a good thing that needs to be preserved, but 'fair use' must be liberally interpreted.

The theft of works, whether exclusively held or not, remains a crime, of course. But the motivation for such theft may be reduced if such work may be otherwise accessible in the near future."

— Gary Knott

Not a Follower

In response to the editorial, "The Fruitlessness of No Methodology" (www.drdobbs.com/architecture-and-design/240144237), which expressed surprise that so many developers in our reader survey responded they used no methodology, we were smartly reminded of something we should have remembered.

"Just because 28% of developers answered your survey 'No development process,' you can't necessarily assume that they have no functioning methodology. Did the survey include a 'Custom/original methodology' choice, or were people effectively indicating 'None of the methodologies listed above'?

Let me explain what I mean by way of analogy. I was raised in a Protestant church and I live my life more-or-less according to Protestant values, but I have theological differences with mainstream religion and I am not a member of a religious community. I do not view myself as a 'subscriber' to any particular religion. If someone asked me to se-

lect my religion from a list of religions, I would probably select 'none.' But it would be incorrect to say that I live an unprincipled life.

I am instinctively opposed to anything that looks like 'group think.' I am skeptical of all religions, political parties, labor unions, fraternities and sororities, management theories, economic theories, and unsurprisingly, software methodologies. I am not saying that these groups or thought systems have nothing to offer. I am simply saying that I do not wish to be defined by any of them. I do not wish to 'wear the label' of any religion, political party, etc.

When asked for my affiliation, I will almost always say 'none.' But you have to be careful about what you read into that response."

— Bob Snyder

Improving Quality The Right Ways

Capers Jones, the noted software engineering methodologies expert, shared research in his observations on the cost of "cowboy" programming as discussed in the aforementioned editorial.

"The software industry spends about $0.50 out of every $1.00 expended for development and maintenance on finding and fixing bugs. Most forms of testing are below 35% in defect removal efficiency. That is, they remove only about one bug out of three. All tests together seldom top 85% in defect removal efficiency. About 7% of bug repairs include new bugs. About 6% of test cases have bugs of their own.

What happens before testing is even more important than testing itself. The most effective known methods of eliminating defects presently are requirements models; automated proofs; formal inspections of requirements, design, and code; and static analysis of code and text. These methods have been measured to top 85% in defect removal efficiency individually. Methods such as inspections also raise testing defect removal efficiency by more than 5% for each major test stage.

Unfortunately, testing has been the primary software defect removal method for more than 50 years. However, most forms of testing are only about 35% efficient as I mentioned above. Due to low efficiency, at least eight forms of testing are needed to achieve reasonably efficient defect removal efficiency. Pre-test inspections and static analysis are synergistic with testing and raise efficiency. Tests by certified test personnel using test cases designed with formal mathematical methods have the highest levels of test defect removal efficiency and can top 65%.

As to methodologies, some methods such as IBM's Rational Unified Process (RUP) and Watts Humphrey's Team Software Process (TSP) can be termed 'quality strong' because they lower defect potentials and elevate defect removal efficiency levels. Other methods such as Waterfall and the Cowboy development you refer to can be termed 'quality weak' because they raise defect potentials and have low levels of defect removal efficiency.

The software industry spends more money finding and fixing bugs than for any other known cost driver. This should not be the case. A synergistic combination of defect prevention, pre-test defect removal, and formal testing can lower software defect removal costs by more than 50% compared to 2012 averages, and raise defect removal efficiency from the current average of about 85% to more than 99%."

— Capers Jones

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

Why NoSQL Is Here To Stay

While they won't displace traditional RDBMSs, the easy scalability and programmability of NoSQL databases guarantees them a permanent place in the data center.

By Matt Asay

NoSQL databases have been the subject of considerable debate as developers and systems architects disagree on where they're most useful. Some technologists are skeptical that NoSQL will even fill a long-term need: They believe NoSQL databases will be limited to niche applications and that SQL databases will continue to be the dominant model. Such concerns ignore fundamental changes to how and why applications are developed. NoSQL is definitely here to stay, for a variety of reasons.

Scalability and Big Data

The days of gigabytes of data are gone; even modestly sized applications now must cope with terabytes or petabytes of data, much of it constantly changing and growing. As the data load continues to grow, IT organizations need a way to scale up quickly and flexibly without investing hundreds of thousands of dollars in building single-server solutions that become increasingly expensive with scale.

NoSQL databases make it easy to meet expanding needs because they're built to scale out, instead of scale up. By automatically functioning across pools of inexpensive commodity servers or cloud computing instances, NoSQL databases enable sites to increase (or decrease) capacity cost-effectively — you can have a high-performance system even if you don't necessarily have the largest budget.

And because they have fewer limits on scalability, NoSQL databases are much better equipped to handle the volume of data that modern applications need. The amount of data we use is not going to shrink, so the need for database management systems that can handle the growth is permanent.

Changing System Architectures

With the advent of cloud computing, the way that systems are set up has changed drastically, and will continue to change as new server types and technologies become available. A database instance might easily be spread across thousands of nodes, located around the world, and serving millions of widely dispersed customers simultaneously.

NoSQL databases natively work with modern system architectures, which means they're set up to scale across an unlimited number of

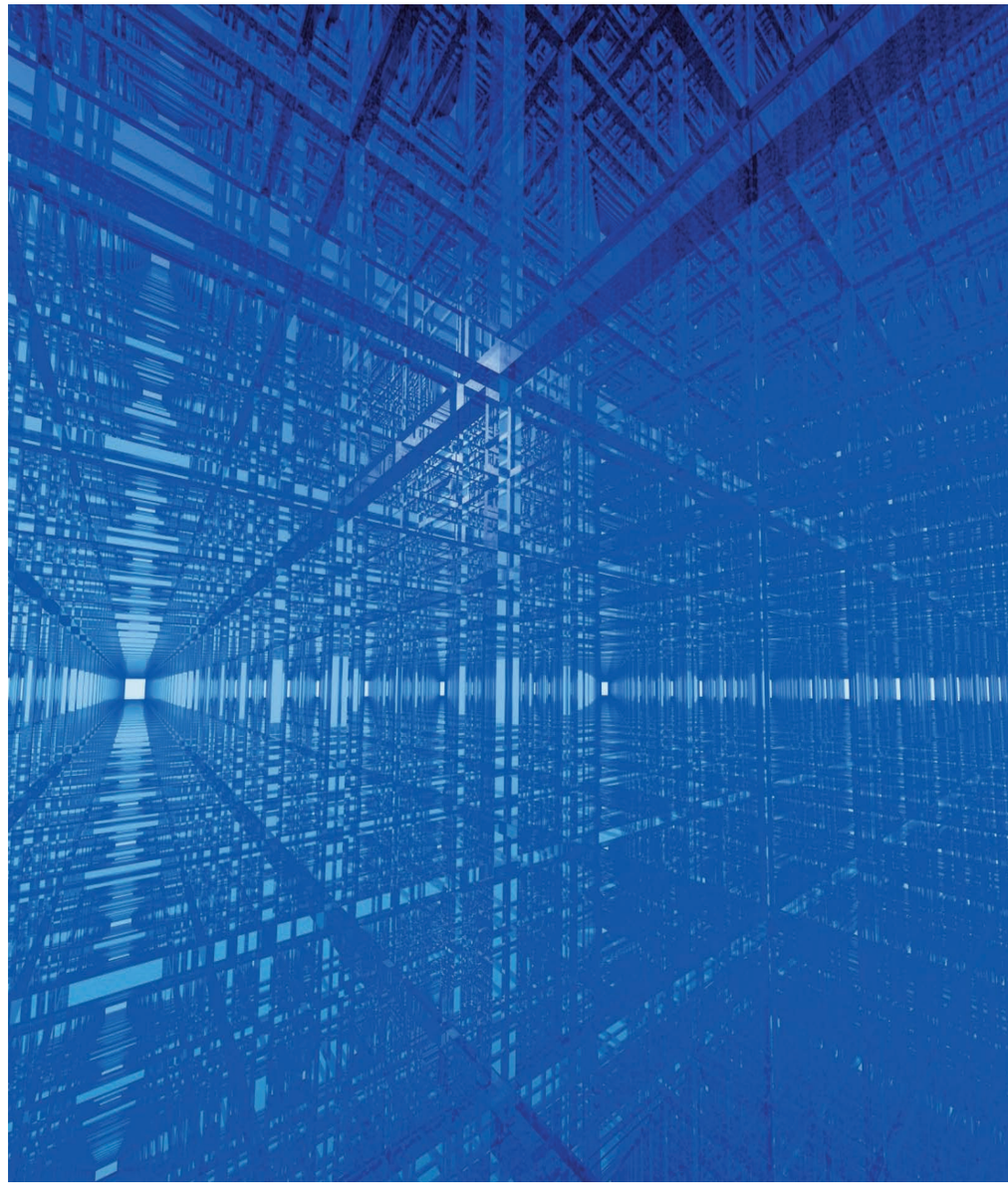
servers and continue functioning normally even as nodes are added and removed. This design dramatically reduces scaling costs as well as server administration and setup time. Many NoSQL databases are also location-aware, making it easy to accelerate data delivery to users based on geography.

Ease of Use

Because they don't have fixed schemas, NoSQL databases are known for being easy to work with. You can code applications quickly, without worrying about time spent migrating data to new schemas as your application evolves. In addition, document-oriented databases make it unnecessary to have object-oriented code transform requests into SQL queries, and then change the SQL results back into objects that fit in with the application logic. So will NoSQL databases displace RDBMS technology across the spectrum of applications? No. Not anytime soon, anyway. But for the vast majority of applications, NoSQL databases and particularly document databases offer the scale, flexibility, and ease-of-use that applications require.

— Matt Asay (http://is.gd/mUfjB6) is the Vice President of Corporate Strategy at 10gen (www.10gen.com/), the company behind MongoDB. Matt is a well-known open source advocate and former board member of the Open Source Initiative (OSI).

Comment



Task-Based Programming in Windows

Setting up a reliable parallel execution context so that programs can be converted into a series of independent executable tasks.

By John Revill

In 2005, Herb Sutter argued that the free lunch of Moore's Law was at an end (http://is.gd/DDDDuZ), and that multicore systems demanded alternative approaches to coding. His point was that we could not rely simply on languages and compilers to shield us from rethinking the way we structure our code.

Several years on, and despite the near ubiquity of multicore systems, the hardware potential seems to run well ahead of general practice in software development. When developers do turn, perhaps in desperation, to parallelization, it's still not unusual for them to end up writing apps with a thousand threads and an architecture based on what is sometimes described as the "synchronize and suffer " model.

This article discusses a different, rather straightforward approach I have found useful in building a number of multithreaded applications (server and client) over the years using existing features of the Windows API.

Architecture

The design consists of the following sequence:

1. From the main thread discrete pieces of work are handed off to an input queue. Each piece is discrete in that it is owned by only one thread at a time, which minimizes the amount of synchronization.
2. The input queue (or more generically an I/O queue) is a wrapper around a Windows I/O completion port.
3. A thread pool services the input queue. Each thread retrieves the next item from the queue and processes it.
4. Active tasks can be returned to the input queue. That is, they can cooperatively yield or they can run until the current work is exhausted. The thread then returns to the input queue for its next piece of work. If the input queue doesn't have enough work to

keep all the threads in the pool working, then threads are effectively parked until there is more work.

5. Output is posted to an output queue (another I/O queue instance), which may be on the main thread or another dedicated, special-purpose thread that acts as a coordinator, or collator of the output. For example, sorting or reassembling data which may have been processed out of order in the thread pool.
6. The collated output arrives at its final destination or sink (Figure 1).

There is nothing particularly new in this arrangement. It is a variation on a design previously presented in *Dr. Dobb's* (http://is.gd/eFDnqs). The goal rather is to show that the advantages in reliability and performance over "synchronize and suffer " can be obtained with only a modest effort.

Let's have a look at key pieces. The goal is for the application code to have minimal knowledge, if any, of its threading context. The threading should all be handled by the plumbing.

I/O Queues

Our input and output queues are instances of a generalized I/O queue, which is a fairly simple wrapper around a Windows I/O Completion Port (IOCP). IOCPs have been around since NT 4.0 and have been written about extensively (see "Multithreaded Asynchronous I/O & I/O Completion Ports"on page 22).

Several Windows subsystems use IOCPs for asynchronous operations including Winsock, Named Pipes, and file I/O. In addition to supporting Windows subsystems, IOCPs enable programmer-developed asynchronous operations. This is the capability we're interested in here. Being able to seamlessly integrate OS-supplied asynchronous features with our own code in a single architecture goes a long way to reducing the overall complexity of our app. The basic queue is shown in Listing One.

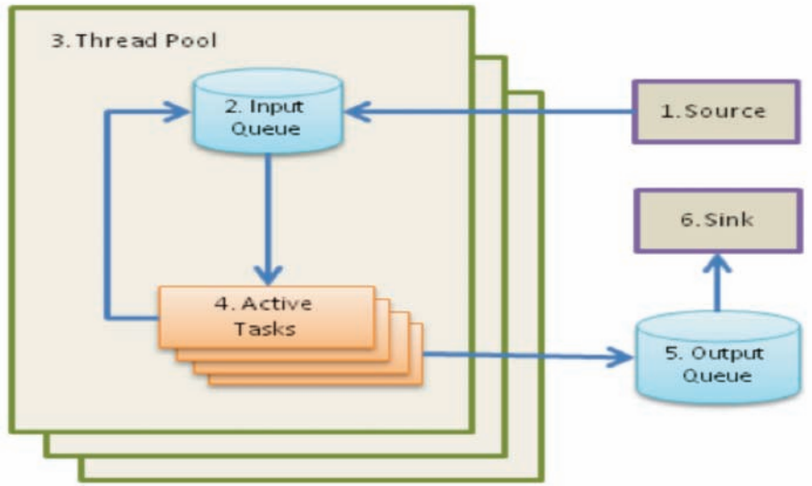


Figure 1: An overview of the process.

Listing One: The queue.

```
class Queue
{
private:
    HANDLE _hiocp;

    public:
        Queue();
        void Associate(HANDLE hAssoc);
        void Run();
        void Quit();
        void Post(IAsyncState *pState);
        void Fetch(IAsyncState **ppState, DWORD wait);
};
```

Work is posted to a queue on one thread and another thread can fetch the work item in FIFO order. When there is no work in the queue,

the fetching thread simply waits, either indefinitely or for a chosen interval. The latter might be useful, for example, for raising a notification indicating that a thread is idle, which could be fed into a thread scaling mechanism.

Thread Pool

Generally, a thread pool with a modest number of workers ends up being the most efficient. See some of the earlier link references for why this is so. As a guideline, we usually don't want to stray too far from a thread count that matches the available number of cores. This is what the example thread pool does. However, the pool constructor allows the size to be set explicitly if you're inclined to experiment.

If we are expecting larger numbers of blocking operations (for example, writing the results to a database), we could consider some form of scaling for the thread pool. This might be necessary if you end up dealing with some synchronous library code.

The example app has been built, intentionally, to run on WIndows XP. If you are targeting Windows 7 or later you could save some coding by using the built-in system thread pool, which is a considerable improvement over the version supplied in XP.

Once the pool is set up and fed, the worker thread function simply extracts items from the input queue and dispatches them via a generic mechanism, which I'll cover in a moment.

`Thread:GetInput` and `Thread::GetOutput` can be used anywhere to obtain pointers to these queues.

Windows supplies a top-level structured exception handler (SEH) for each thread (http://is.gd/nIggcF). If the top-level handler is invoked, that usually ends the whole program. Ideally, any runtime errors should be captured and dealt with in the application code and an error escaping into the thread function should be non-continuable. For this example (Listing Two), I add just enough support to make an attempt to pass some diagnostic data back to the main thread, which responds by shutting down the pool.

Listing Two: Operations with exception handling.

```
static unsigned _stdcall Function(void* param)
{
    assert(param);
    StructuredExceptionTranslator seh;

    try {
        Pool* pPool = reinterpret_cast<Pool*>(param);

        IO::Queue *pInput = pPool->GetInput();
        IO::Queue *pOutput = pPool->GetOutput();

        SetInput(pInput);
        SetOutput(pOutput);

        for(;;) {
            IO::IAsyncState* pState = NULL;
            pInput->Fetch(&pState, INFINITE);

            if(!pState) break;
                unsigned ret = pState->Dispatch();

            if(ret == IO::Queue::ERROR_YIELD)
                pInput->Post(pState);
        }

    } catch(StructuredException& e) {

        IO::Queue* pOutput = Thread::GetOutput();
        std::string msg =  "SEH Error ";
        AsyncError *pErr = new AsyncError(e, msg);

        //TODO:  copy exception record

        pOutput->Post(pErr);

    } catch(Win32Error& e) {
```

```
        IO::Queue* pOutput = Thread::GetOutput();
        std::string msg = e;
        AsyncError *pErr = new AsyncError(e, msg);
        pOutput->Post(pErr);

    } catch(std::exception& e) {

        IO::Queue* pOutput = Thread::GetOutput();
        std::string msg = e.what();
        AsyncError *pErr = new AsyncError(0xFFFFFFFF, msg);
        pOutput->Post(pErr);

    } catch(...) {

        IO::Queue* pOutput = Thread::GetOutput();
        std::string msg =  "Unexpected error ";
        AsyncError *pErr = new AsyncError(0xFFFFFFFF, msg);
        pOutput->Post(pErr);
    }

    _endthreadex(0);
    return 0;
}

};
```

There and Back Again

The plumbing for passing object instances around and invoking methods asynchronously are supplied by a pair of template classes: `AsyncStateT` and `AsyncMethodT`. Listing Three is simple class declaration:

Listing Three: The AsyncStateT class.

```
class MyAsyncClass: public AsyncStateT<MyAsyncClass>
{

    public:

        typedef AsyncMethodT<MyAsyncClass> Method;
        typedef AsyncStateT<MyAsyncClass> State;

        MyAsyncClass();


    private:

        unsigned Initialize();
        unsigned Step();
        unsigned Finalize();
        unsigned Report();

        unsigned _counter;

};
```

`AsyncStateT` implements an instance of `IAsyncState`, which is the interface required by `IO::Queue::Post()`. But before we can hand

off an instance of our new class, we need to setup the method to be invoked on the worker thread. This is handled in the constructor as follows:

```
MyAsyncClass() :_counter(0)
{
    State();
    State::SetInstance(this);
    State::SetMethod(new Method(&MyAsyncClass::Initialize));
}
```

`State::SetMethod` indicates the current method to be executed. `AsyncMethodT` handles the plumbing of the dispatch mechanism as well, ensuring (for example) that we can't do something like:

```
State::SetMethod(new Method(&NotMyClass::NotMyMethod));
```

without the compiler making a fuss. By convention, I use typedef of `unsigned(T::*Impl)();` for my async methods.

Now, we can hand off an instance to the input queue. Should you want to make sure that you won't be tempted to mess around with the instance while it's being processed, simply avoid keeping a reference:

```
IO::Queue *pInput = Thread::GetInput();
pInput->Post(new MyAsyncClass());
```

The arrival of this object causes the IOCP to wake up a worker thread, which has been parked since it called `pInput->Fetch`.

```
for(;;) {

    IO::IAsyncState* pState = NULL;
    pInput->Fetch(&pState, INFINITE);

    if(!pState)
        break;

    unsigned ret = pState->Dispatch();
    if(ret == IO::Queue::ERROR_YIELD)
          pInput->Post(pState);
}
```

With the retrieved pointer to an instance of `IAsyncState` in hand, the worker simply invokes `IAsyncState::Dispatch()`. `AsyncStateT`'s implementation of `Dispatch` hands the call off to the currently assigned instance of `IAsyncMethod`. `AsyncMethodT`'s implementation of `IAsyncMethod::Dispatch` is shown in Listing Four:

Listing Four: The dispatch method.

```
unsigned IO::IAsyncMethod::Dispatch(IO::IAsyncState*
pState)
{

    AsyncStateT<T>* pContext =
        dynamic_cast<AsyncStateT<T>*>(pState);
    assert(pContext);

    T* pInst = pContext->Instance();
    pInst->SetContext(pContext);
    unsigned ret = (pInst->*_impl)();
    pInst->SetContext(NULL);

    return ret;
}
```

First, we unpack the instance we passed in, and the combination of `dynamic_cast` and `assert` ensure that we haven't somehow managed to incorrectly map an instance of the generic `IAsyncState` interface. Actually, the `dynamic_cast` could accomplish that on its own by returning a `NULL` pointer that would result in an OS fault when we tried to use it but the `assert` will do it in a (slightly) nicer way. There's a little more pointer juggling until we get to:

```
unsigned ret = (pInst->*_impl)();
```

`_impl` is a reference to the method, which we set in our class constructor (which, you may recall, was `Initialize`).

```
unsigned Initialize()
{
    _counter = 10;
    State::SetMethod(new Method(&MyAsyncClass::Step));
    return IO::Queue::ERROR_YIELD;
}
```

This routine is now executing in a worker thread and we have assigned it the herculean task of setting the `_counter` member to 10.

Having completed initialization, let's move on to doing some work. `State::SetMethod(new Method(&MyAsyncClass::Step))` sets the current method to step; and returning `IO::Queue::ERROR_YIELD` instructs the worker thread to return the instance to the input queue.

Bear in mind that in the intervening period, any number of other items might have been posted, so our instance is at the back of the queue. When it is retrieved, possibly on a different thread, `Step()` will be executed (Listing Five):

Listing Five: The Step function.

```
unsigned Step()
{
    MyAsyncClass *pCopy = new MyAsyncClass(*this);
    pCopy->SetMethod(new Method
        (&MyAsyncClass::OnReportProgress));
    IO::Queue *pOutput = Thread::GetOutput();
    pOutput->Post(pCopy);
    _counter--;

    if(_counter == 0) {
        State::SetMethod(new Method(&MyAsyncClass::Finalize));
    }
    return IO::Queue::ERROR_YIELD;
}
```

`Step()` posts a copy of itself back to the output queue and in place of doing any real work decrements the counter. (In reality, this code would be an exercise in futility as the overhead of returning the instance to the input queue would far outweigh the utility of just decrementing a counter. For the moment we're just following the flow. A more realistic example is provided later.) We make a copy the current instance rather than post current instance. The state of our instance may have changed before it is fetched from the output queue, and if we want to ensure its integrity we would be back to needing to synchronize the access. Making a copy relieves us of this burden. If we have exhausted the work available (so, `counter == 0`), we arrange a transition to `Finalize` (Listing Six). Regardless of whether we have another step to take or need to finalize, we yield back to the input queue.

Listing Six: The Finalize function

```
unsigned Finalize()
{
    State::SetMethod(new Method(&MyAsyncClass::OnComplete));
    Thread::GetOutput()->Post(this);
    return;
}
```

In this case, `Finalize()` has little to do. Any required cleanup could be done here. We set the `OnComplete` handler and pass our instance to the output queue before returning `ERROR_SUCCESS` to indicate to the worker thread that we don't want this item posted again.

As our thread pool has been laboring to wind our counter back, the main thread has been fetching and dispatching our output queue items. And we've been receiving status reports:

```
unsigned OnReportProgress()
{
    printf( "%x %s %d\n ", this, __FUNCTION__, this->_counter);
    State::Delete();
    return ERROR_SUCCESS;
}
```

It helps to have a convention to keep the input and output sides of the fence separate in our minds. I've used a prefix of "On" for that purpose. Our progress reporter simply writes some state to the console and deletes itself. Remember this is a point-in-time copy of our working object. For a busy system, allocating and then deleting copies might become expensive and you will definitely run into some synchronization issues in the default heap allocator. In this case, a pool allocator with a private heap may be very useful.

The final output item is the return of our original instance to the output queue:

```
unsigned OnComplete()
{
    printf( "%x %s %d\n ", this, __FUNCTION__, this->_counter);
    State::Delete();
    return ERROR_SUCCESS;
}
```

Again, we just print some state and then delete the instance. Putting it all together, the output looks like this:

```
>>  DoDemoClass
3955c0 MyAsyncClass::OnReportProgress 10
396628 MyAsyncClass::OnReportProgress 9
3955c0 MyAsyncClass::OnReportProgress 8
396698 MyAsyncClass::OnReportProgress 7
396708 MyAsyncClass::OnReportProgress 6
396778 MyAsyncClass::OnReportProgress 5
396628 MyAsyncClass::OnReportProgress 4
3967e8 MyAsyncClass::OnReportProgress 3
396858 MyAsyncClass::OnReportProgress 2
3968c8 MyAsyncClass::OnReportProgress 1
395550 MyAsyncClass::OnComplete 0
<<  DoDemoClass
```

This has been a fairly detailed look at a trivial example with some excursions into some the internals of the plumbing classes. However, I hope it is clear that for the most part we don't have to worry much about the details, and we can focus on the application code. This is as it should be. Now for a more interesting example.

Example: Particle Swarm Optimization (PSO)

I've chosen a PSO (http://is.gd/9blJvv) as my example application because it's interesting in its own right, relatively simple to code, and a good fit for the discussion at hand. If you haven't been exposed to a PSO before, the idea is to randomly distribute a number of particles (or agents, if you prefer) across a search space and have the particles collaboratively search for an optimal solution.

The search space I'll use here is the Sombrero function (http://is.gd/U9O5Ji). As a search space this function holds few mysteries but it does effectively demonstrate the issue of local optima.

If the preceding paragraphs seem irretrievably vague, imagine a nest of ants wandering across the graph, shrouded in fog, searching for the highest point, or more typically for ants, the highest concentration of sugar. Each can sense its own altitude and can share that information. We can see that an individual ant could become stuck on the lower circular ridge. However, with a sufficiently large nest randomly but evenly scattered across the graph, the central spire would be quickly discovered. The analogy isn't entirely facetious; PSOs were inspired by animal behavior.

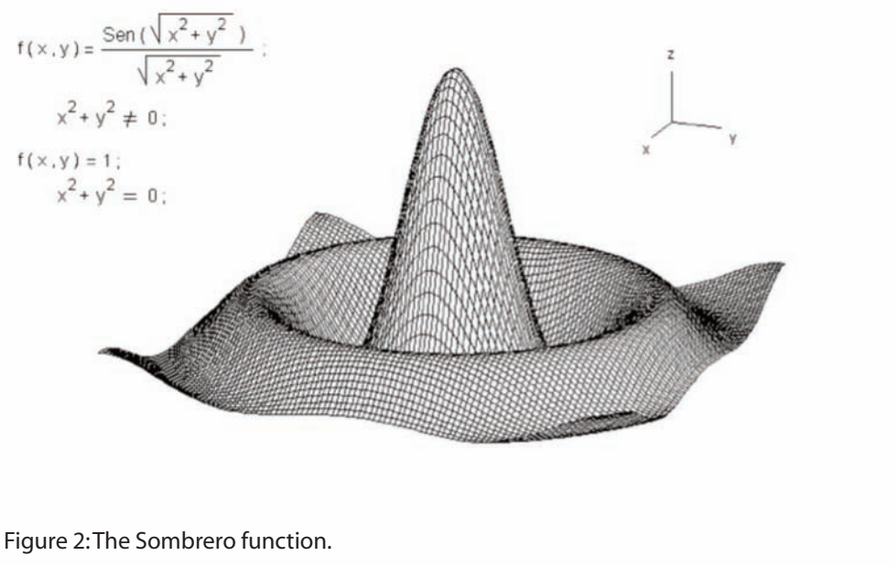


Figure 2: The Sombrero function.

PSOs can be used to hunt for solutions in domains that are combinatorially much more complex than the example. The example application implements two versions of a swarm: a synchronous and an asynchronous version invoked as shown in Listing Seven(a) and (b) respectively:

Listing Seven(a): The synchronized version of the principal PSO function.

```
unsigned DoSyncSwarm()
{
    std::cout <<  ">>   " << __FUNCTION__ << std::endl;
    DWORD start = ::GetTickCount();
    PSO::ISwarm * pSwarm = new PSO::Swarm();

    DoInitSwarm(1, pSwarm);
    pSwarm->Run();

    std::cout <<  "<<   "<< __FUNCTION__ <<  "
                  " << ::GetTickCount() - start <<
          std::endl << std::endl;

    return 0;
}
```

Listing 7(b): The async version of the principal PSO function.

```
unsigned DoAsyncSwarm()
{
    std::cout <<  ">>   " << __FUNCTION__ << std::endl;
    DWORD start = ::GetTickCount();
    Thread::Pool pool;

    PSO::ISwarm *pSwarm = new PSO::AsyncSwarm();
    DoInitSwarm(2, pSwarm);
    pSwarm->Run();

    pool.Run();

    std::cout <<  "<<   "<< __FUNCTION__ <<  "
                  " << ::GetTickCount() - start
          << std::endl << std::endl;

    return 0;
}
```

On the surface, differences between the routines are minor. `DoAsyncSwarm` creates a thread pool, and an `AsycSwarm` instance; then, after setting up and starting the swarm, it runs the pool, which returns when all particles have run to completion.

The synchronous form is useful for coming to grips with the working of the swarm, if it's not familiar, without getting tangled in the asynchronous implementation. It's also a useful reality check to determine whether our performance has really improved.

The naive approach to this problem — that is, the one I tried first — is to simply return particles to the input queue between each step. Similar to the aforementioned trivial example, the overhead of doing this meant that the asynchronous version was consistently slower.

My next approach was to run each particle to completion but this overlooked a subtlety of the way PSOs work. Each particle can have global influence by updating the best fitness and position values for the entire swarm. Running a couple of particles to completion before any others defeated the purpose of the swarm in the first place. In the end, I settled on creating an asynchronous helper class that could be assigned a number of particles from the parent swarm. I went with a simple round robin particle distribution across a collection of helpers, each of which is then queued. A randomized distribution may be a better approach in a production system, but I haven't tested that idea as yet.

A test on an Intel Core Duo 1.58 GHz machine with a run of 200 particles each stepping through 1000 iterations sees improvements ranging from 25%–75% using the asynchronous version. Occasionally, the asynchronous version will take the same amount of time or even longer. Bear in mind that the system is also servicing every other thread on the system as well and from time to time the thread pool in the app will just miss out.

Running in the IDE, you will regularly see `DoAsyncSwarm` coming in behind `DoSyncSwarm`. This is the Heisenberg effect at work — caused by the overhead of the IDE keeping track of the internal state of your program including threads. And the IDE itself has its own collection of threads to deal with. It's not too surprising that the timings are different. When you make a change, test your code in release mode to get a clear idea of the difference it will make.

Conclusion

I have used this architecture successfully in several of systems including: a mine scheduling system, a video camera sharing system in a facial image capture and recognition system, a package management system, and now a particle swarm optimization. While lunch may no longer be free, it can still be had at a quite reasonable price.

The complete source code for the example can be downloaded at http://is.gd/YQQ8sy.

— John Revill has more than 20 years' experience programming. He currently helps Queensland University of Technology in Australia move technology from the lab to the marketplace.

Comment

From the Vault

Improving Futures and Callbacks in C++ To Avoid Synching by Waiting

In C++, futures are a great way of decomposing a program into concurrent parts, but a poor way of composing those parts into a responsive and scalable program. This recent article from our vault shows how Microsoft's Parallel Pattern Library (PPL) provides a solution using tasks.

— *DDJ*

By Artur Laksberg

The C++11 standard provides several long-requested concurrency features such as the `std::thread`, `std::future`, and others. While those are a welcome addition to the language, in this article, I will show that they are not sufficient for all but the most basic concurrency needs. I will argue that the primitives in C++11 are particularly ill-suited for modern applications that must deal with the concurrency imposed by I/O operations and exploit multicore at the same time.

Fortunately, many of these limitations can be addressed by augmenting C++11 futures with continuations, based on experience with the Parallel Patterns Library (PPL) at Microsoft.

The reader is expected to have a working knowledge of C++11 and some experience writing parallel code, but familiarity with the PPL is optional.

Connected and Multicore

We take it for granted that the software we use daily is both connected to the Internet and able to harness multiple cores. It is natural to think of the Internet (or the cloud) and multiple cores not as two distinct capabilities of a program, but as a single elastic compute resource. As Herb Sutter puts it in *Welcome To The Parallel Jungle*, "The network is just another bus to more compute cores" (http://is.gd/GCObWm).

However, a developer building a modern connected multicore application faces two distinct challenges:

- Building a well-performing connected program requires dealing with the latency and unpredictability typical of I/O operations. This is difficult, and getting it right results in a an application that is responsive and scalable, but not necessarily faster.
- Building a well-performing multicore program is a parallel programming job — a different challenge altogether, often requiring a different tool set and different skills. When this is done right, the program runs faster, although its speed is usually orthogonal to the responsiveness and the scalability.

Taking the view of the cloud as a natural extension of multicore behooves us to find a programming model that, at the very minimum, gives us a way to efficiently compose the I/O operations and the multicore operations.

Concurrency in C++

In the last decade, the software industry has developed a many tools for multicore programming in C++. Libraries such as Intel's TBB (Threading Building Blocks) or Microsoft's PPL (Parallel Patterns Library) are the state of the art. These tools excel at parallel decomposition — partitioning serial code into multiple "chores" that run on multiple cores.

But there is more to being "connected and multicore" than just parallelism. Well-performing concurrent programs must combine the connected components, with their inherent latency and unreliability, with the parallel components. Put another way, if parallelism is about decomposing the program into independent parts, concurrency is about both decomposing and composing the program from the parts that work well individually and together.

I believe that it's in the composition of connected and multicore components where today's C++ libraries are still lacking.

The Dreaded Wait

Most of the concurrency primitives in C++11 are composed via waiting. One can spawn a thread (by creating an instance of std::thread), then wait for it to finish by calling the join method. Likewise, the result of a future object (represented by std::future) can be retrieved by calling the get method — during which the calling thread waits for the result to become available.

Why is this a problem?

Waiting on the GUI thread means that the user of the application is rewarded with the "hourglass" or the "spinning donut" while the thread is waiting for an operation to complete. This is bad enough for a CPU-bound operation, but the length of an I/O-bound call can be truly unpredictable and therefore very long.

Clearly, the GUI thread of the application is a scarce resource, and we want to return it to the message pump as soon as possible — but let's not kid ourselves by thinking that all we need to do is offload the long-running operation to another thread. If we did that, how would we synchronize the two threads without waiting?

The woes of composition-by-waiting are not limited to GUI programs. By default, at creation, a thread reserves 1 MB of stack space on Windows and 8 MB on Linux. This value is configurable, but reducing the stack size may break programs with deep call chains or multiple stack-allocated objects. In other words, not only GUI threads are ex-

pensive — all threads are. This can be felt acutely in a multithreaded server application. If many threads decide to block at the same time, waiting can bring the server to its knees very quickly.

Continuations

In C/C++, continuations are commonly known as "callbacks," and they are often used for asynchronous programming. This is not to say that the concept is unique to C++, but because the language has been a laggard in adopting mainstream functional programming features such as the lambda expressions, C++ libraries that use continuations consistently are still rare.

The concept of the continuation was pioneered by Scheme, which introduced the style of programming where instead of returning a value, a function takes an additional parameter — the continuation that is invoked to process the return value of the function. Naturally, the continuation itself is also a function that can take continuations, and so on.

Continuations make the flow of control explicit — instead of invoking a function and waiting for it to complete, a program written in a continuation-passing style specifies explicitly what to do with the return value when it is available.

For concurrent programs, continuations are a boon because they allow us to avoid blocking waits — which, as I stated above, greatly hinder responsiveness and scalability.

JavaScript has made continuations ubiquitous in Web programming. Because JavaScript is single-threaded, waiting for the server to produce the data would freeze the browser. Instead, JavaScript uses a technique known as AJAX, where the act of issuing a request to the server is separated from the act of handling the data retrieved from the server:

```
http.open("GET", "customer.html");
http.onreadystatechange = function() {
    if(http.readyState == 4) {
        var serverResponse = http.responseText;
        // process serverResponse here: ...
    }
}
```

More recently, Node.js has been very successful at capturing the mindshare of the developer community thanks to its use of continuations for server-side programming. In Windows Runtime, which powers the Metro-style apps in Windows 8, the concept of continuations is used holistically for all potentially long-running applications. Continuations are, in fact, the only way of working with asynchronous operations.

Tasks, Futures, and Promises

A beloved child has many names, as the saying goes. The concept of the "task" — also known as the future or the promise, depending on the language and library — represents a relatively straightforward idea.

Simply put, a task is a proxy for an eventual value. The value might (but doesn't have to) be produced on a different thread. In addition to the value, the task can produce a side effect, such as adding a record to the log file or printing a message on the screen. In that sense, a task can be thought of as a generalization of a function, where the act of invocation is separated from the act of retrieving the result. In C++11, the tasks are called futures. The future is a class template that can be used as follows:

```
std::future<int> f = std::async([] {
    return do_work();
});
// do useful work here
int result = f.get();
```

Here, we instantiate the future by calling an `std::async` function taking a lambda expression. Depending on your implementation, `std::async` will either launch a new thread or schedule the execution of the function on a thread pool. Because the creation of the future is separated from the retrieval of the result, some useful work can take place between these events, potentially resulting in parallel execution. We get more parallelism by running more futures concurrently, and we can build higher-level constructs on top of futures to do so.

A future in C++ doesn't have to be backed by a concurrently running function created by `std::async`. Another concept, called the promise, is used to communicate the value to the consumer of the future. For example, we can create a promise, pass it to another thread, and let it set the value on it. In the meantime, we can execute some useful work before retrieving the value from the future:

```
std::promise<int> p;
std::thread t([&]() {
    p.set_value(do_work());
});
std::future<int> f = p.get_future();
// do useful work
int result = f.get();
Don't Keep Me Waiting!
```

For many C++ programmers,the introduction of basic concurrency concepts in C++11 is exciting news. We no longer have to deal with non-standard, sometime proprietary, and platform-specific libraries for parallelism. By using the standard C++11 facilities, we can be sure that our code will compile and run on all platforms — if not now, then in the near future, when C++11 compilers become universally available on every platform.

I will disappoint many readers by stating bluntly that futures in C++, in their current form, are ill-suited for building modern connected multicore programs.

First of all, futures and the associated primitives in C++11 form a programming model that is too low-level for our everyday needs. While it is possible to build higher-level constructs such as `parallel_for` or `parallel_sort` on top of futures, commercial libraries such as TBB and PPL will always outperform our scrappy implementations. The way to solve this is to bring those industrial-strength implementations into the Standard.

For a software library to be low-level is not necessarily a bad thing, as long as the components of the library are composable. Unfortunately, this is not the case with futures. As Bartosz Milewski opined, "What was omitted from the standard, however, was the ability to compose futures" (http://is.gd/xxWPGr). I would add that the ability to compose futures in a wait-free way is particularly detrimental for connected multicore applications.

Specifically, consider the aforementioned example. When `f.get()` is called on the GUI thread, the thread blocks and is unable to process other messages sent to it by the event loop. As a result, the program stops responding to the user input — it freezes. Now imagine calling `f.get()` on the server side. A well-implemented thread pool will detect the blocking operation and spawn another thread that is able to do useful work. However, if many threads decide to block at the same time (imagine a Web service under heavy load), the thread pool will either spawn a large number of threads (which are expensive, as I mentioned earlier) or throttle thread creation, rendering the server temporarily unresponsive — even though the threads are sitting idle in the blocked operations doing no useful work. Things get worse when

we need to compose multiple futures — this simply cannot be done without blocking every one of them individually.

Towards a Better Future

Futures are a great way of decomposing a program into concurrent parts, but a poor way of composing these parts into a responsive and scalable program. At Microsoft, we hear consistently from C++ developers that writing concurrent code is hard, and writing code that composes concurrent I/O and CPU-bound operations efficiently is especially hard. As part of our attempt to address this, we extended the PPL with the concept of the task.

The task in PPL is conceptually similar to std::future, with one important addition — continuations. Like `std::future`, the `concurrency::task` is a class template that can be associated with a function that returns the payload of the task. Now, instead of calling get on the task to retrieve its value, we can call the then method to set up the continuation, in which we process the value:

```
task<int> t([]()
{
    return do_work();
});
task<void> t2 = t.then([](int n)
{
    std::cout << n;
});
```

The task is a very simple object that contains a placeholder for the eventual value and a list of function objects — the continuations to be invoked when the task transitions to the completed state. The `then` method doesn't block — all it does is add a continuation to that list. When the task completes, all its continuations are scheduled on the thread pool.

The default thread pool can be substituted with a custom scheduler, which is passed as a parameter to the then function. This is useful in several scenarios — for example, an event loop-based scheduler will run the work on a GUI thread, allowing it to access the thread-bound UI controls. A priority scheduler can run the work on a high-priority thread. A locality-aware scheduler will run the work close to a particular CPU core or a NUMA node, and so on.

The return value of the `then` is again a task, which can be continued, allowing for serial composition of operations. Multiple continuations off the same task can execute concurrently, giving us a way to do parallel composition of operations. Several tasks can be composed via the `join` or the `choice` operator, both non-blocking operations that produce a task that can be continued, or composed with others, and so on. For example, the `choice` operator can be used to produce a task that completes when the first of the argument tasks completes:

```
task<int> t1([]()
{
    return do_work();
});

task<int> t2([]()
{
    return do_some_other_work();
});

(t1 || t2).then([](int n) {
    cout << "The first value to arrive is " << n;
});
```

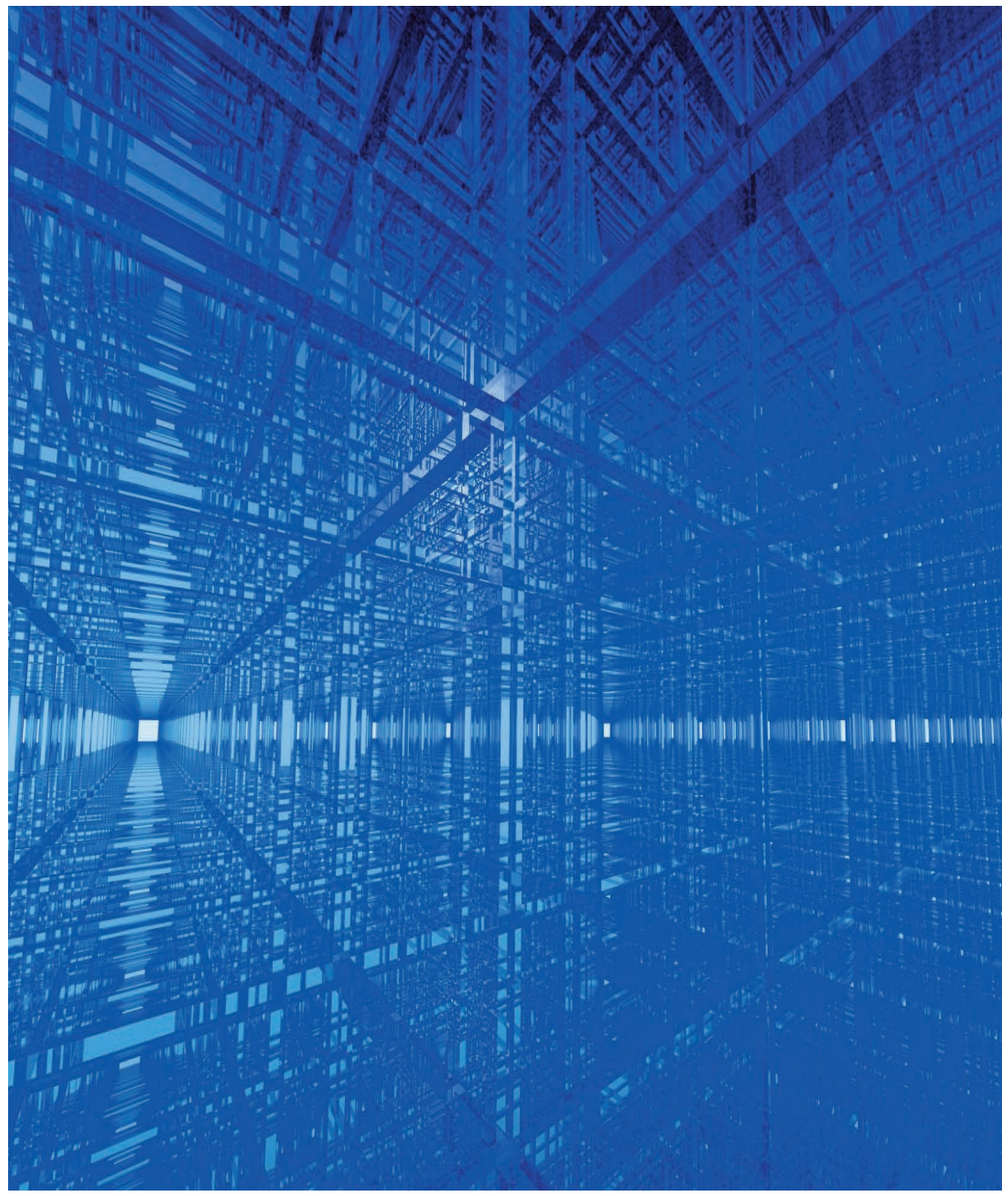
The ideas of tasks and continuations in PPL are influenced by the Task Parallel Library (http://is.gd/x8WCaZ), developed by Microsoft for

the .NET framework, as well as the Promises library of CommonJS (http://is.gd/wrQDjc), and (of course) `std::future` in C++11.

At Microsoft, we've been quite successful with this model so far. Windows Runtime has adopted PPL tasks as the model of asynchrony in C++, and we've started a new project called Casablanca (http://is.gd/FvyYyP) to build a modern C++ library for cloud-based client-server communication, in which all concurrent operations are exposed as PPL tasks. Today, PPL is a Microsoft library for targeting the Windows platform only, but we feel that the developer community at large can benefit from the ideas we developed in the PPL. That's why we've submitted a proposal to the C++ Standard's Committee to extend the std::future concept with continuations, along the lines of the continuations in the PPL tasks. The proposal is publicly available (PDF at http://is.gd/ZUHnMA), and we're currently in the process of building an experimental reference implementation compatible with Visual Studio 2012. We're a small team in Microsoft and not shy about asking for help from the community. If you're willing to volunteer constructive feedback, or test-drive our implementation, I promise that your voice will be heard.

— Artur Laksberg is a member of the Visual C++ team at Microsoft, working on concurrency libraries and tools. He can be reached at arturl@microsoft.com.

Comment



From the Vault

Multithreaded Asynchronous I/O & I/O Completion Ports

This article from 2007 discusses how I/O completion ports provide an elegant solution to the problem of writing scalable server applications that use multithreading and asynchronous I/O.

— *DDJ*

By Tom R. Dial

When developing server applications, it is important to consider scalability, which usually boils down to two issues. First, work must be distributed across threads or processes to take advantage of today's multiprocessor hosts. Second, I/O operations must be scheduled efficiently to maximize responsiveness and throughput. In this article, I examine I/O completion ports — an elegant innovation available on Windows that helps you accomplish both of these goals.

I/O completion ports provide a mechanism that facilitates efficient handling of multiple asynchronous I/O requests in a program. The basic steps for using them are:

- Create a new I/O completion port object.
- Associate one or more file descriptors with the port.
- Issue asynchronous `read/write` operations on the file descriptor(s).
- Retrieve completion notifications from the port and handle accordingly.

Multiple threads may monitor a single I/O completion port and retrieve completion events — the operating system effectively manages the thread pool, ensuring that the completion events are distributed efficiently across threads in the pool.

A new I/O completion port is created with the `CreateIoCompletionPort` API. The same function, when called in a slightly different way, is used to associate file descriptors with an existing completion port. The prototype for the function looks like this:

```
HANDLE CreateIoCompletionPort(
   HANDLE FileHandle,
   HANDLEExistingCompletionPort,
   ULONG_PTR  CompletionKey,
   DWORD NumberOfConcurrentThreads
   );
```

When creating a new port object, the caller simply passes `INVALID_HANDLE_VALUE` for the first parameter, `NULL` for the second and third parameters, and either zero or a positive number for the `ConcurrentThreads` parameter. The last parameter specifies the maximum number of threads Windows schedules to concurrently process I/O completion events. Passing zero tells the operating system to allow at least as many threads as processors, which is a reasonable default. For a discussion of why you might want to schedule more threads than available processors, see *Programming Server-Side Applications for Windows 2000* by Jeffrey Richter and Jason D. Clark (http://is.gd/tAcTzP).

Associating File Descriptors with a Port

Once a port has been created, file descriptors opened with the `FILE_FLAG_OVERLAPPED` (or `WSA_FLAG_OVERLAPPED` for sockets) may be associated with the port via another call to the same function. To associate an open file descriptor (or socket) with an I/O completion port, the caller passes the descriptor as the first parameter, the handle of the existing completion port as the second parameter, and a value to be used as the "completion key" for the third parameter. The completion key value is passed back when removing completed I/O requests from the port. The fourth parameter is ignored when associating files to completion ports; a good idea is to set this to zero.

Initiating Asynchronous I/O Requests: OVERLAPPED Explained

Once a descriptor is associated with a port, (and you may associate many file descriptors with a single I/O Completion Port), an asynchronous I/O operation on any of the descriptor(s) results in a completion event being posted to the port by the operating system. The same Windows APIs that let callers perform standard synchronous I/O have a provision for issuing asynchronous I/O requests. This is accomplished by passing a valid `OVERLAPPED` pointer to one of the standard functions. For example, take a look at `ReadFile`:

```
BOOL ReadFile(
  HANDLE    File,
  LPVOID    pBuffer,
  DWORD     NumberOfBytesToRead,
  LPDWORD   pNumberOfBytesRead,
  LPOVERLAPPED  pOverlapped
  );
```

For typical (synchronous) I/O operations, you've always passed `NULL` for the last parameter, but when doing asynchronous I/O, you need to pass the address of an `OVERLAPPED` structure in order to specify certain parameters as well as to receive the results of the operation. Asynchronous calls to `ReadFile` are likely to return `FALSE`, but `GetLastError` returns `ERROR_IO_PENDING`, indicating to the caller that the operation is expected to complete in the future.

A common mistake when using `OVERLAPPED` structures is to pass the address of an `OVERLAPPED` structure declared on the stack:

```
OVERLAPPED Gone;
// Set up 'Gone'..
ReadFile ( hFile, pBuf, Count,
          &NumRead, &Gone );
```

This just won't work because `ReadFile` returns immediately, and when the function containing the call to `ReadFile` exits, the stack will be unwound and the data pointed to by `&Gone` will become invalid. Thus, you should ensure that your program manages its `OVERLAPPED` structures (and any buffers you're using) carefully. The example em-

ploys a fairly common strategy that involves having a C++ class representing a connection derive from `OVERLAPPED` — which may offend some C++ purists, but is a practical solution to the problem. The connections are allocated on the heap, and when I/O operations are initiated, the connections' pointer is passed as the pointer to `OVERLAPPED`.

Retrieving Completed I/O Events from the Port

Now that we know how to create a completion port, associate descriptors to it, and initiate asynchronous I/O operations on the descriptors, it's on to retrieving completion events from the port. A thread removes an event from the port's queue by calling the `GetQueuedCompletionStatus` function:

```
BOOL GetQueuedCompletionStatus(
  HANDLE          CompletionPort,
  LPDWORD         pNumberOfBytes,
  PULONG_PTR      pCompletionKey,
  LPOVERLAPPED*   ppOverlapped,
  DWORD           Timeout
  );
```

Obviously, the first parameter to this function is the handle to the port object, followed by several pointers and a `Timeout` value. Once an operation has completed successfully, the variable pointed to by `pNumberOfBytes` contains the number of bytes written or read during the I/O completion, the `pCompletionKey` value contains the value of the completion key passed when associating the file descriptor to the port, and the `ppOverlapped` variable points to the `OVERLAPPED` pointer passed as the parameter to the asynchronous I/O function. The timeout value, which is specified in milliseconds, works just like other Windows functions in that the special value `INFINITE` may be passed to specify "wait forever."

Sending Your Own Events: PostQueuedCompletionStatus

Before we move on to a practical example, there's one more function to discuss:

```
BOOL PostQueuedCompletionStatus(
  HANDLE          CompletionPort,
  DWORD           NumberOfBytesTransferred,
  ULONG_PTR       CompletionKey,
  LPOVERLAPPED    pOverlapped
  );
```

This function lets you post completion events to the port. Typically, this function is used to send implementation-specific messages to the port. When you post a completion event to a port, one of the threads blocking on the port successfully returns from its call to `GetQueuedCompletionStatus` with copies of the parameters as they were posted.

This function is often used to notify worker threads of some global or application-wide event. Along those lines, the sample program presented in this article posts completion events with a special completion key value of `COMPLETION_KEY_SHUTDOWN` in order to tell the worker threads that the server is shutting down.

A Practical Example: The Fire Web Server

When I first sat down to write this article, I recalled that my own learning experience was made somewhat difficult by the relative dearth of real-world examples. The same was true of my experience learning

about some of the more esoteric Windows Sockets APIs. As these are both important Windows innovations, I decided to develop a simple, multithreaded web server that demonstrates the use of both. It is named in honor of my friend and colleague Ray Schraff, who often tells our customers that mankind has adopted the World Wide Web faster than any technology since the invention of fire. The Fire web server (available in a zip file at http://is.gd/JvStan) exploits I/O completion ports and the best features of Windows Sockets to deliver respectable performance in about 500 lines of C++ code.

The main() Event

All important initialization occurs in the main function, including the initialization of the Windows socket library, registration of an event handler to capture the user's request to stop the server via CTRL-C, and creation of the listener socket.

Next, a single I/O completion port is created, followed by the creation of a small pool of worker threads. Finally, a fixed number of `Connection` objects (each of which manages one socket) are created.

The Connection Class

The real meat of the program lies within the implementation of the `Connection` class. Its constructor creates a socket, associates it with the I/O completion port previously created in the `main` function, and finally, issues an asynchronous request to accept a client connection.

People familiar with the standard accept API may be confused by the fact that a client socket is created prior to the call to `AcceptEx`, so let me explain. `AcceptEx` requires that the client socket be created upfront, but this minor annoyance has a payoff in the end: It lets a socket descriptor be reused for a new connection via a special call to `TransmitFile`. This means that a server that deals with many short-lived connections can utilize a pool of allocated sockets without incurring the cost of creating new descriptors all the time.

The rest of the `Connection` class is a simple state machine; any given connection may be in any of four states:

1. `WAIT_ACCEPT`. Waiting for `AcceptEx` to complete.
2. `WAIT_REQUEST`. Waiting for the client request to be complete.
3. `WAIT_TRANSMIT`. Waiting for the response to be sent.
4. `WAIT_RESET`. Waiting for the client socket to be reset.

Here's how things get rolling: When the `Connection` objects are allocated in the main function, they all issue asynchronous accept calls on their sockets. This means that shortly after startup, all connection objects are in the `WAIT_ACCEPT` state, until a client actually connects and the operating system wakes one of the worker threads.

The handling worker thread takes advantage of the fact that the `Connection` class is derived from `OVERLAPPED`, casts the `OVERLAPPED` pointer into a connection object, and assuming the pointer checks out,

calls the `Connection`'s `OnIoComplete` function.

`OnIoComplete` implements the `Connection` class's state machine — essentially transitioning from one waiting state to another by calling the appropriate `CompleteXxx` function. For example, when a new client connects, the `CompleteAccept` method is called to perform the necessary steps to prepare the socket for actual use.

Likewise, each `CompleteXxx` function's last move is to issue another asynchronous I/O request, whether to read more data from the client, transmit a response, or ask that the socket be reset and ready to accept a new client.

At this point, several items merit mention when designing around I/O completion ports and asynchronous I/O. First, as has already been discussed, because asynchronous I/O functions typically return immediately, you must ensure that any buffers passed to the calls remain valid at least until the completion event is handled. This implies heap allocation, since buffers allocated on the stack in a function get junked on exit.

Second, in a server application such as Fire, any thread could handle any connection at any time. As soon as an asynchronous file operation is issued on a descriptor, it is up to the operating system to pick a thread to run the completion routine. Put differently, there is no guaranteed affinity between the thread issuing an asynchronous I/O call and the thread receiving the completion notification. For this reason, you must design your data structures carefully in order to ensure that threads don't tromp all over each other when trying to handle a request.

Last, you must be extremely careful to design your application to avoid races and the other classes of problems that arise when writing multithreaded programs. For example, when designing Fire, I spent a considerable portion of my development time convincing myself which states were necessary to consider. I was also careful to make sure that the various asynchronous I/O requests (read data, write data, reset the socket, and so on) were always the last operations performed in any of the completion handlers.

The reason is subtle, but clear: Were I to perform any other types of work in a handler function after issuing an I/O request, I would have a race condition — it would be possible for more than one thread to be operating on a connection object concurrently.

The benefit of all this careful planning, however, is that Fire does not require any mutual exclusion mechanism in its implementation.

The result should be that Fire scales well on multicore machines since individual worker threads are never competing for resources.

Conclusion

I/O completion ports provide an elegant solution to the problem of writing scalable server applications that use multithreading and asynchronous I/O. While it is important to design such applications carefully to avoid certain types of problems such as race conditions or excessive resource contention, the benefits of doing so far outweigh the costs, especially considering the world of multicore, multiprocessor servers in which we now reside.

Acknowledgments

Thanks to Dave Cutler, Len Holgate, Paul Lloyd, and especially Dad for his detailed review.

— At the time of original publication, Tom was a development team leader for Hyland Software.

Comment

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

ACCESS DATA WITH REST IN WINDOWS 8 APPS

Develop and consume a REST service that provides data to a Windows Store app that uses the Grid app — using C# and XAML.
http://www.drdobbs.com/240144594

DEBUGGING MEMORY IN IOS APPS

Dangling pointers, double frees, memory leaks, and hogs — when it comes to writing apps for iOS, you've got to avoid these common pitfalls. Here's how!
http://www.drdobbs.com/240144285

HOW TO DISTRIBUTE THE GAME OF LIFE

Do you have a very large grid that you want to use in a simulation and the grid doesn't fit into the memory of a single node? If so, learning how to develop a distributed memory version of the "Game of Life" may be a practical solution.
http://www.drdobbs.com/240145395

DEVELOPER READING LIST

New books on C, C#, Node, Win8 Apps, Perl, and Groovy.
http://www.drdobbs.com/240145159

BUILDING SCALABLE WEB ARCHITECTURE AND DISTRIBUTED SYSTEMS

Open source software has become a fundamental building block for some of the biggest websites. And as those websites have grown, best practices and guiding principles around their architectures have emerged. This article seeks to cover some of the key issues to consider when designing large websites, as well as some of the building blocks used to achieve these goals.
http://www.drdobbs.com/240142422

COMPARING OPENCL, CUDA, AND OPENACC [VIDEO]

Rob Farber takes you on a tour of the paths to massively parallel x86, MultiGPU, and CPU+GPU applications.
http://www.drdobbs.com/240145156

IN THIS ISSUE

Dr. Dobb's