Using fibers to simplify enumerators, part 3: Having it both ways

devblogs.microsoft.com/oldnewthing/20041231-00

December 31, 2004

Raymond Chen

As we discovered in the previous two entries [second], the problem with enumeration is that somebody always loses.

Now we will use fibers to fight back. Before you decide to use fibers in your programs, make sure to read the **dire warnings** at the end of this article. My goal here is to show one use of fibers, not to say that fibers are the answer to all your problems. Fibers can create more problems than they solve. We'll come back to all the dire warnings later.

As with most clever ideas, it has a simple kernel: Use a fiber to run both the caller and the enumerator each on their own stack.

```
#include <windows.h>
#include <shlwapi.h>
#include <stdio.h>
#include <strsafe.h>

enum FEFOUND {
 FEF_FILE,          // found a file
 FEF_DIR,           // found a directory
 FEF_LEAVEDIR,      // leaving a directory
 FEF_DONE,          // finished
};

enum FERESULT {
 FER_CONTINUE,      // continue enumerating
                    // (if directory: recurse into it)
 FER_SKIP,          // skip directory (do not recurse)
};

class __declspec(novtable) FiberEnumerator {
public:
 FiberEnumerator();
 ~FiberEnumerator();

 FEFOUND Next();
 void SetResult(FERESULT fer) { m_fer = fer; }
 void Skip() { SetResult(FER_SKIP); }

 virtual LPCTSTR GetCurDir() = 0;
 virtual LPCTSTR GetCurPath() = 0;
 virtual const WIN32_FIND_DATA* GetCurFindData() = 0;

protected:
 virtual void FiberProc() = 0;

 static void DECLSPEC_NORETURN WINAPI
    s_FiberProc(void* pvContext);

 FERESULT Produce(FEFOUND fef);

protected:
 void* m_hfibCaller;
 void* m_hfibEnumerator;
 FEFOUND  m_fef;
 FERESULT m_fer;
};

FiberEnumerator::FiberEnumerator()
 : m_fer(FER_CONTINUE)
{
 m_hfibEnumerator = CreateFiber(0, s_FiberProc, this);
}
```

```
FiberEnumerator::~FiberEnumerator()
{
 DeleteFiber(m_hfibEnumerator);
}

void DECLSPEC_NORETURN FiberEnumerator::
    s_FiberProc(void *pvContext)
{
 FiberEnumerator* self =
    reinterpret_cast<FiberEnumerator*>(pvContext);
 self->FiberProc();

 // Theoretically, we need only produce Done once,
 // but keep looping in case a consumer gets
 // confused and asks for the Next() item even
 // though we're Done.
 for (;;) self->Produce(FEF_DONE);
}
```

This helper class does the basic bookkeeping of fiber-based enumeration. At construction, it remembers the fiber that is consuming the enumeration, as well as creating a fiber that will produce the enumeration. At destruction, it cleans up the fiber. The derived class is expected to implement the `FiberProc` method and call `Produce()` every so often.

The real magic happens in the (somewhat anticlimactic) `Produce()` and `Next()` methods:

```
FERESULT FiberEnumerator::Produce(FEFOUND fef)
{
 m_fef = fef; // for Next() to retrieve
 m_fer = FER_CONTINUE; // default
 SwitchToFiber(m_hfibCaller);
 return m_fer;
}

FEFOUND FiberEnumerator::Next()
{
 m_hfibCaller = GetCurrentFiber();
 SwitchToFiber(m_hfibEnumerator);
 return m_fef;
}
```

To `Produce()` something, we remember the production code, pre-set the enumeration result to its default of `FER_CONTINUE`, and switch to the consumer fiber. When the consumer fiber comes back with an answer, we return it from `Produce()`.

To get the next item, we remember the identity of the calling fiber, then switch to the enumerator fiber. This runs the enumerator until it decides to `Produce()` something, at which point we take the production code and return it.

That’s all there is to it. The `m_fef` and `m_fer` members are for passing the parameters and results back and forth across the fiber boundary.

Okay, with that groundwork out of the way, writing the producer itself is rather anticlimactic.

Since we want to make things easy for the consumer, we use the interface the consumer would have designed, with some assistance from the helper class.

```
class DirectoryTreeEnumerator : public FiberEnumerator {
public:
 DirectoryTreeEnumerator(LPCTSTR pszDir);
 ~DirectoryTreeEnumerator();

 LPCTSTR GetCurDir() { return m_pseCur->m_szDir; }
 LPCTSTR GetCurPath() { return m_szPath; }
 const WIN32_FIND_DATA* GetCurFindData()
    { return &m_pseCur->m_wfd; }

private:
 void FiberProc();
 void Enum();

 struct StackEntry {
   StackEntry* m_pseNext;
   HANDLE m_hfind;
   WIN32_FIND_DATA m_wfd;
   TCHAR m_szDir[MAX_PATH];
 };
 bool Push(StackEntry* pse);
 void Pop();

private:
 StackEntry *m_pseCur;
 TCHAR m_szPath[MAX_PATH];
};

DirectoryTreeEnumerator::
 DirectoryTreeEnumerator(LPCTSTR pszDir)
 : m_pseCur(NULL)
{
 StringCchCopy(m_szPath, MAX_PATH, pszDir);
}

DirectoryTreeEnumerator::~DirectoryTreeEnumerator()
{
 while (m_pseCur) {
   Pop();
 }
}

bool DirectoryTreeEnumerator::
      Push(StackEntry* pse)
{
 pse->m_pseNext = m_pseCur;
 m_pseCur = pse;
 return
  SUCCEEDED(StringCchCopy(pse->m_szDir,
               MAX_PATH, m_szPath)) &&
  PathCombine(m_szPath, pse->m_szDir, TEXT("*.*")) &&
  (pse->m_hfind = FindFirstFile(m_szPath,
       &pse->m_wfd)) != INVALID_HANDLE_VALUE;
}

void DirectoryTreeEnumerator::Pop()
{
 StackEntry* pse = m_pseCur;
 if (pse->m_hfind != INVALID_HANDLE_VALUE) {
  FindClose(pse->m_hfind);
 }
 m_pseCur = pse->m_pseNext;
}

void DirectoryTreeEnumerator::FiberProc()
{
 Enum();
}

void DirectoryTreeEnumerator::Enum()
{
 StackEntry se;
 if (Push(&se)) {
  do {
   if (lstrcmp(se.m_wfd.cFileName, TEXT(".")) != 0 &&
       lstrcmp(se.m_wfd.cFileName, TEXT("..")) != 0 &&
       PathCombine(m_szPath, se.m_szDir, se.m_wfd.cFileName)) {
    FEFOUND fef = (se.m_wfd.dwFileAttributes &
                    FILE_ATTRIBUTE_DIRECTORY) ?
                    FEF_DIR : FEF_FILE;
    if (Produce(fef) == FER_CONTINUE && fef == FEF_DIR) {
     Enum(); // recurse into the subdirectory we just produced
    }
   }
  } while (FindNextFile(se.m_hfind, &se.m_wfd));
 }
 Produce(FEF_LEAVEDIR);
 Pop();
}
```

As you can see, this class is a mix of the two previous classes. Like the consumer-based class, information about the item being enumerated is obtained by calling methods on the enumerator object. But like the callback-based version, the loop that generates the objects themselves is a very simple recursive function, with a call to `Produce` in place of a callback.

In fact, it's even simpler than the callback-based version, since we don't have to worry about the FER_STOP code. If the consumer wants to stop enumeration, the consumer simply stops calling `Next()`.

Most of the complexity in the class is just bookkeeping to permit abandoning the enumeration prematurely.

Okay, let’s take this fiber out for a spin. You can use the same `TestWalk` function as last time, but for added generality, change the first parameter from `DirectoryTreeEnumerator*` to `FiberEnumerator*`. (The significance of this will become apparent next time.)

A little tweak needs to be made to the main function, though.

```
int __cdecl main(int argc, char **argv)
{
 ConvertThreadToFiber(NULL);
 DirectoryTreeEnumerator e(TEXT("."));
 TestWalk(&e);
 ConvertFiberToThread();
 return 0;
}
```

Since the enumerator is going to switch between fibers, we’d better convert the thread to a fiber so it’ll have something to switch back to!

Here’s a schematic of what happens when you run this fiber-based enumerator:

`ConvertThreadToFiber`		
Main fiber		
construct `DirectoryTreeEnumerator`		**Enumerator fiber**
`CreateFiber`		(not running)
initialize variables		
`Next(CONTINUE)`		
`SwitchToFiber()`	→	starts running
		`FindFirstFile` etc
		`Produce(FILE)`
`Next(CONTINUE)` “returns” FILE	←	`SwitchToFiber()`
use the result		
`Next(CONTINUE)`		
`SwitchToFiber()`	→	`Produce(FILE)` “returns” CONTINUE
		`FindNextFile` etc
		`Produce(DIR)`

`Next(CONTINUE)` “returns” DIR	←	`SwitchToFiber()`
use the result		
`Next(CONTINUE)`		
`SwitchToFiber()`	→	`Produce(DIR)` “returns” CONTINUE
	and so on… until…	
		`Produce(DONE)`
`Next(CONTINUE)` “returns” DONE	←	`SwitchToFiber()`
cleanup		
`DeleteFiber`		
destruct `DirectoryTreeEnumerator`		
`ConvertFiberToThread`		

Observe that from each fiber’s point of view, the other fiber is just a subroutine!

Coding subtlety: Why do we capture the caller’s fiber each time the `Next()` method is called? Why not capture it when the `FiberEnumerator` is constructed?

Next time, we’ll see how this fiber-based enumerator easily admits higher-order operations such as filtering and composition.

Dire warnings about fibers

Fibers are like dynamite. Mishandle them and your process explodes.

The first dire warning is that fibers are expensive in terms of address space, since each one gets its own stack (typically a megabyte).

And since each fiber has its own stack, it also has its own exception chain. This means that if a fiber throws an exception, only that fiber can catch it. (Same as threads.) That’s a strong argument against using an STL std::stack object to maintain our state: STL is based on an exception-throwing model, but you can’t catch exceptions raised by another fiber. (You also can’t throw exceptions past a COM boundary, which severely limits how much you can use STL in a COM object.)

One of the big problems with fibers is that everybody has to be in cahoots. You need to decide on one person who will call the `ConvertThreadToFiber` function since fiber/thread conversion is not reference-counted. If two people call ConvertThreadToFiber on the same thread, the first will convert it, and so will the second! This results in two fibers for the same thread, and things can only get worse from there.

You might think, "Well, wouldn't the `GetCurrentFiber` function return NULL if the thread hasn't been converted to a fiber?" Try it: It returns garbage. (It's amazing how many people ask questions without taking even the slightest steps towards figuring out the answer themselves. Try writing a test program.)

But even if GetCurrentFiber told you whether or not the thread had been converted to a fiber, that still won't help. Suppose two people want to do fibrous activity on the thread. The first converts, the second notices that the thread is already a fiber (somehow) and skips the conversion. Now the first operation completes and calls the `ConvertFiberToThread` function. Oh great, now the second operation is stranded doing fibrous activity without a fiber!

Therefore, you can use fibers safely only if you control the thread and can get all your code to agree on who controls the fiber/thread conversion.

An important consequence of the "in cahoots" rule is that you have to make sure all the code you use on a fiber is "fiber-safe" – a level of safety even beyond thread-safety. The C runtime library keeps information in per-thread state: There's errno, all sorts of bonus bookkeeping when you create a thread, or call various functions that maintain state in per-thread data (such as strerror, _fcvt, and strtok).

In particular, **C++ exception handling is managed by the runtime**, and the runtime tracks this data in per-thread state (rather than per-fiber state). Therefore, if you throw a C++ exception from a fiber, **strange things happen**.

(Note: Things may have changed in the C runtime lately; I'm operating from information that's a few years old.)

Even if you carefully avoid the C runtime library, you still have to worry about any other libraries you use that use per-thread data. None of them will work with fibers. If you see a call to the `TlsAlloc` function, then there's a good chance that the library is not fiber-safe. (The fiber-safe version is the `FlsAlloc` function.)

Another category of things that are not fiber-safe are windows. Windows have thread affinity, not fiber affinity.

Raymond Chen

Follow