FOUNTAINHEAD

NEW YORK
2ND MAR
2016

Baruch College

EXPERT EXCEL

~ Asynchronous UDFs ~

Excel Asynchronous UDFs

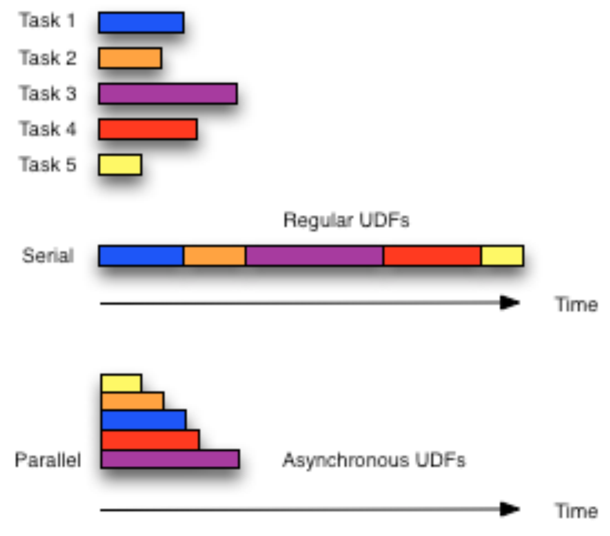
Asynchronous UDFs were introduced in Excel 2010. They are part of the Excel 2010 XLL SDK.

Asynchronous UDFs in Excel have the potential to dramatically improve the calculation speed of workbooks that have long-running UDF functions.

Regular UDFs are typically dispatched in serial fashion; that is to say, the first UDF is dispatched and has to finish before the next UDF is dispatched and has to finish … and so on for all the UDFs in a workbook.

Asynchronous UDFs on the other hand are dispatched onto separate calculation threads (outside Excel's calculation engine) and run in parallel, which is to say that the UDFs run concurrently and are scheduled by the operating system.

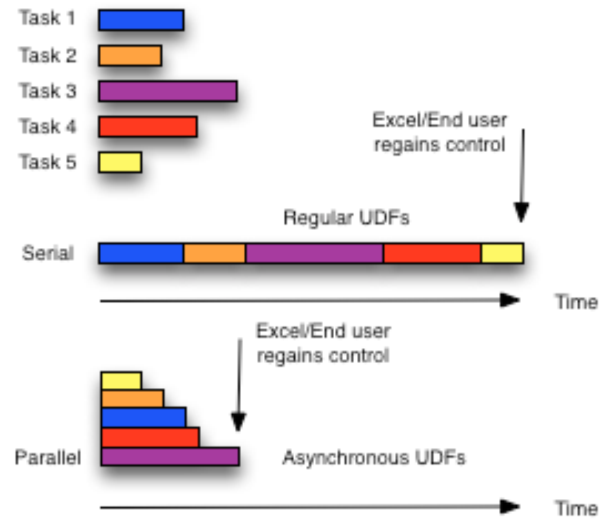
Clearly, UDFs running in parallel are better than UDFs running in sequence.



Regular Versus Asynchronous UDFs

Asynchronous? Really?

Actually, the term asynchronous is somewhat misleading, and perhaps parallel UDF would be a better name. Fact is that control is not handed back to the workbook—enabling the user to continue working—until the longest running UDF finishes.



When is control handed back to Excel?

So for UDFs that take minutes or hours to run the overall run time will likely be shorter than for regular UDFs, but the end user experience won't be great as Excel won't be usable in the meantime.

This is in contrast to the case where an RTD server is used as a compute server for long-running computations, because Excel remains usable throughout.

That said, asynchronous UDFs do provide significant performance gains, and in the ideal case, for N tasks each taking time T to run, the overall run time would be just T, and not N x T as for regular UDFs. But if T is long, say many seconds, then asynchronous UDFs still make for a less than great user experience.

Excel 2010 XLL SDK

To build asynchronous UDFs you will need to code them as an XLL.

An XLL is just a Windows DLL with certain entry points (functions) pre-defined. To have a working XLL requires that these functions are implemented:

- `xlAddInManagerInfo()`
- `xlAddInManagerInfo12()`
- `xlAutoAdd()`
- `xlAutoClose()`
- `xlAutoFree()`
- `xlAutoFree12()`
- `xlAutoOpen()`
- `xlAutoRegister()`
- `xlAutoRegister12()`
- `xlAutoRemove()`

The functions with the "12" postfix are called by Excel 2007 and later.

Essential what these functions do is perform a "handshake" between Excel and the XLL add-in. By "handshake" I mean the functions are called and data is exchanged with Excel so it can register the UDF functions within the XLL and call them from cells on worksheets.

XLOPER12

Fundamental to understanding Excel and Excel UDFs is that data structure `XLOPER12` (and prior to Excel 2007, just `XLOPER`). It is the fundamental data element within Excel. For example, the data in a single cell is an `XLOPER12` and because the `XLOPER12` structure is a union of data types it is polymorphic (meaning that cells in Excel can contain many different types of data: numbers, strings, range references, etc.).

```
/*
** XLOPER12 structure
**
** Excel 12's fundamental data type:
** can hold data of any type. Use "U"
** as the argument type in the
** REGISTER function.
**/
typedef struct xloper12
{
	union
	{
		double num;
		XCHAR *str;
		BOOL xbool;
		int err;
		int w;

		struct
		{
			WORD count;
			XLREF12 ref;
		} sref;
		struct
		{
			XLMREF12 *lpmref;
			DWORD idSheet;
		} mref;
		struct
		{
			struct xloper12
				*lparray;
			RW rows;
			COL columns;
		} array;
		struct
		{
			union
			{
				int level;
				int tbctrl;
				DWORD idSheet;
			} valflow;
			RW rw;
			COL col;
			BYTE xlflow;
		} flow;
		struct
		{
			union
			{
				BYTE *lpbData;
				HANDLE hdata;
			} h;
			long cbData;
		} bigdata;
	} val;
	DWORD xltype;
} XLOPER12, *LPXLOPER12;
```

When XLL UDFs are called by Excel, what's passed back and forth are typically XLOPER12's or pointers to XLOPER12's

Asynchronous UDF. What's different?

Asynchronous UDFs are different from regular XLL UDFs in two ways: 1) how they are registered, and 2) how they are written.

Let's first look at how registration of asynchronous UDFs is different from regular UDFs. Then let's look at how to write an asynchronous UDF.

Registering XLL Asynchronous UDFs

The standard way to register XLL UDF functions is when the XLL is loaded by Excel; that is when `xlAutoOpen()` is called.

```
int WINAPI xlAutoOpen(void)
{
	Reg(L"UDFFuncName",
	    L"QQ",
	    L"UDFFuncName",
	    L"Param");
	Reg(L"UDFAsync",
	    L">QX",
	    L"UDFAsync",
	    L"Param");
	return 1;
}
```

The above function registers two XLL UDFs: `UDFFuncName` which is a regular UDF, and `UDFAsync` which is an asynchronous UDF.

The function `Reg()` is a helper function that simplifies the registration process.

```
void Reg(XCHAR* udf,
         XCHAR* type,
         XCHAR* udfName,
         XCHAR* param)
{
     XLOPER12 xResult;
     XLOPER12 xModuleText;
     Excel12(xlGetName,
             &xModuleText, 0);
     LPXLOPER12 pxudf =
        TempStr12(udf);
     LPXLOPER12 pxtype =
        TempStr12(type);
     LPXLOPER12 pxudfName =
        TempStr12(udfName);

     LPXLOPER12 pxparam =
        TempStr12(param);

     Excel12(xlfRegister,
             &xResult, 5,
             &xModuleText,
              pxudf,
              pxtype,
              pxudfName,
              pxparam);

      Excel12(xlFree,
              &xResult, 1,
              &xModuleText);

      xlAutoFree12(pxudf);
      xlAutoFree12(pxtype);
      xlAutoFree12(pxudfName);
      xlAutoFree12(pxparam);
}
```

In terms of registration, this is all pretty standard stuff, except for the ">" and "X" in the registration of the asynchronous UDF.

The ">" marks the function as asynchronous.

The "X" takes a little more explanation. "X" designates an XLOPER of type `xltypeBigData` and represents the asynchronous callback handle, which refers to a function you write and which runs on a separate thread and calls back to Excel with the result when that result is ready.

So the registration string ">QX" should be interpreted as "register an asynchronous UDF that takes an XLOPER12 as a parameter and will return a result through a hidden (from the user) handle to a function that runs on a separate thread". It's a bit of a mouthful, but if you break it into pieces it makes sense.

Implementing an Asynchronous UDF

Unlike regular UDFs, asynchronous UDFs come in two parts. A synchronous function that is called by Excel when the function is invoked from a cell, and a asynchronous function that is dispatched on a background thread.

First the asynchronous function.

```
DWORD WINAPI
AsyncFibonacci(LPVOID args)
{
	LPXLOPER12* opers =
	    (LPXLOPER12*)args;
	XLOPER12 xlResult;
```

```
     // Code for Fibonacci.
     // Result is fibnum.

     if (opers[0]->xltype &
         xltypeNum)
     {opers[0]->val.num = fibnum;

     int retval =
     Excel12(xlAsyncReturn,
            &xlResult, 2,
             opers[1], opers[0]);

     delete opers;

     ExitThread(0);
     return 0;
}
```

And next the synchronous function that launches the asynchronous function.

```
void WINAPI
Fibonacci(LPXLOPER12 oper,
          LPXLOPER12 asyncHandle)
{
     LPXLOPER12* argsArray =
        new LPXLOPER12[2];
     if (argsArray == NULL)
     {
AsyncStubFailHelper(asyncHandle);
        return;
     }
     argsArray[0] =
        TempOper12(oper);
     if (argsArray[0] == NULL)
     {
        delete argsArray;
AsyncStubFailHelper(asyncHandle);
     }
     argsArray[1] =
        TempOper12(asyncHandle);
     if (argsArray[1] == NULL)
     {
     xlAutoFree12(argsArray[0]);
     delete argsArray;
AsyncStubFailHelper(asyncHandle);
     }
     if (CreateThread(NULL, 0,
         AsyncFibonacci,
         argsArray, 0, NULL)
         == NULL)
     {
     xlAutoFree12(argsArray[1]);
     xlAutoFree12(argsArray[0]);
     delete argsArray;
AsyncStubFailHelper(asyncHandle);
     }
}
```

Calling Asynchronous UDFs

When all is said and done, from the end user's perspective, an asynchronous UDF looks like any other Excel function, albeit with a slightly different behavior behind the scenes.

	A	B	C
1	=UDFAsync(42)		
2			
3			

returns …

	A	B	C
1	165,580,141		
2			
3			

When to Use Asynchronous UDFs

As each asynchronous UDF launches on a separate thread—and threads, especially in large numbers, are not without overhead from an operating system point of view—you don't want to write asynchronous UDFs that are likely to be called from more than 10's of cells in Excel. But if you are calling some service that takes some time to return just because of round-trip latency, for example, then that sort of function would be a good candidate for an asynchronous UDF; in effect, instead of suffering the round-trip latency T, N times for N calls, you would just suffer it once—that's called latency hiding.

However, if you have hundreds or thousands of calls to long-running UDF functions you would do well to consider handing that off the an RTD compute server.

Resources

You will find these links useful:

- *Writing Asynchronous User-Defined Functions in Excel 2010* (http://msdn.microsoft.com/en-us/library/ff796219.aspx).
- *Asynchronous User-Defined Functions* (http://msdn.microsoft.com/en-us/library/ff475859.aspx).
- *Developing Excel 2010 XLLs* (http://msdn.microsoft.com/en-us/library/bb687911.aspx).
- *Excel 2010 SDK: Excel 2010 XLL Software Development Kit* (http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=23242).