

Re: WaitForMultipleObjects , threads and blocking function

Source: <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2008-04/msg00071.html>

- *From:* Joseph M. Newcomer <newcomer@xxxxxxxxxxxxx>
 - *Date:* Wed, 02 Apr 2008 08:20:42 -0500
-

See below...

On Wed, 02 Apr 2008 14:32:02 +0200, mosfet <john.doe@xxxxxxxxxxxxx> wrote:

Hi,

I would like some help about a problem about Threading and blocking function.

We have developed a c++ wrapper around wininet to post some specific data.

Here is the architecture :

A thread always running and waiting on two events, when the event m_hStartRequest is signaled the HTTP request is written to server and once it has finished it send an event to inform the caller that the request is done.

```
DWORD WINAPI CHttpAdapter::ThrRequestAction( IN LPVOID lpvThreadParam )
{
HANDLE WaitHnd[2] = {0};

CHttpAdapter* pThis = (CHttpAdapter*) lpvThreadParam;
if ( pThis == NULL ) {
return -1;
}

WaitHnd[0] = pThis->m_hStartRequest;
WaitHnd[1] = pThis->m_hExitThread;
```

Note that by placing the events in this order, you give preference to starting requests and not to shutting down. Thus, if there is a pending request, it will always be treated as the most important thing going on, and shutdown is considered secondary. The usual arrangement is to make the exit event be [0], so it always has priority.

Re: WaitForMultipleObjects , threads and blocking function

```
//  
while (1)  
{  
  
    DWORD dwRet = WaitForMultipleObjects( 2, WaitHnd, FALSE, INFINITE);  
    if ((dwRet == WAIT_OBJECT_0)) {
```

Already you are in trouble! The function can return several values, and therefore testing for just one of them is always poor coding style. It should be
switch(dwRet)

```
{  
case WAIT_OBJECT_0: // Shutdown event (see above comment)  
... deal with shutdown  
break;  
case WAIT_OBJECT_0 + 1: // request pending  
  
    // Start our http request  
    BOOL bSend = FALSE;  
    do {  
    NetStream& stream = pThis->m_pRequest->GetRequestStream();  
    DWORD dwWrite = stream.Write( pThis->m_pBufferIn, pThis->m_ulSizeIn );  
    ATLTRACE(_T("CHttpAdapter::ThrRequestAction: Sent %d bytes.\n"),  
dwWrite);  
    bSend = stream.Close();  
    while( !bSend );
```

and if the stream doesn't close, exactly what happens here? I can't figure out what the purpose of this code is. It does a synchronous read, but if the close fails, it tries to do the read again? I don't follow the logic at all

```
    }  
    pThis->m_pWebResp = (HttpWebResponse*)  
    pThis->m_pRequest->GetResponse();  
    ...  
  
    ::SetEvent( pThis->m_hRequestDone );
```

And who is blocked on this event? And why is it a single event? Why is it an event at all? Note that doing things like using Event to signal something-to-do and something-done is usually a bad design, because operations like semaphores should be used to handle queueing things up for service threads (otherwise there is a potentially fatal race condition, most of the time) and the sending thread should not be waiting for anything to happen; instead, it should be off doing something useful and from time to time receiving notifications that something has been finished. Common asynchronous interthread messaging systems include PostMessage, PostThreadMessage, and PostQueuedCompletionStatus. When I

Re: WaitForMultipleObjects , threads and blocking function

see bidirectional SetEvent, I get real suspicious about the correctness of the code

```
}  
}
```

```
Ret_t CHttpAdapter::SendDataAndParseResponse()
```

Already a design error. You are still thinking sequentially in an asynchronous world. There are two, only vaguely related, functions here: SendData, and ParseResponse. They should not be made into a single function. Otherwise, you have not demonstrated why there is a need for a secondary thread. You could have done this with a subroutine call. Just toss the request over the wall to the thread, and at some indefinite time in the future, the thread will initiate an action that notifies you that it has completed. Stop thinking sequentially.

```
{  
Ret_t nRet = TP_ERR_OK;  
EHttpRequestKind eHttpRequestKind;  
  
ATLTRACE(_T("CHttpAdapter::SendDataAndParseResponse()\n"));  
  
// Inform thread to send the request  
::SetEvent( m_hStartRequest );
```

This would mean it has been queued up somewhere. So why not use something like an I/O Completion Port, or a Semaphore, to indicate a non-empty queue?

```
DWORD dwRetType=::WaitForSingleObject(m_hRequestDone, 60000 );  
if (dwRetType == WAIT_OBJECT_0)
```

ALWAYS use a switch statement after a WFSO. Also, it is interesting to note that you seem to have only two kinds of completions: error completions, and error completions. Can't it ever complete successfully? You should at least have a default statement that says default:

```
ASSERT(FALSE);  
...set "internal error" code  
break;
```

but you are always at risk, especially during development, if you make the naive assumption that WFSO will ALWAYS return ONLY the one or two values you expect

```
eHttpRequestKind = eHttpRequest;  
else if (dwRetType == WAIT_TIMEOUT)
```

Re: WaitForMultipleObjects , threads and blocking function

```
eHttpErrKind = eTimeoutErr;  
  
nRet = CheckResponse( eHttpErrKind );  
  
return nRet  
}
```

The problem arises when the thread is trying to write to server and server do not respond.

In this case the stream.Write stay blocked and after 1 min the caller(SendDataAndParseResponse) falls in timeout and in this case I am trying to release it by calling the HttpEndRequest but it doesn't work ...

Yes, I'd expect that using synchronous communication. You need to do asynchronous HTTP if you want to recover from dead servers, but it has been far too many years since I looked into doing that. But the synchronous model you've programmed here is ill-suited for an asynchronous world, and using synchronous I/O for network sockets can be assumed to always be a design error. The problem is that you have two different timeouts going here, one of your own (60 seconds) and one of possibly unspecified delay, certainly longer, built into the HTTP layer protocol. There is a natural conflict, and the easiest way to avoid it is to use some form of asynchronous HTTP communication which will allow you to cancel pending I/O operations.

joe

Joseph M. Newcomer [MVP]

email: newcomer@xxxxxxxxxxxxx

Web: <http://www.flounder.com>

MVP Tips: http://www.flounder.com/mvp_tips.htm

.