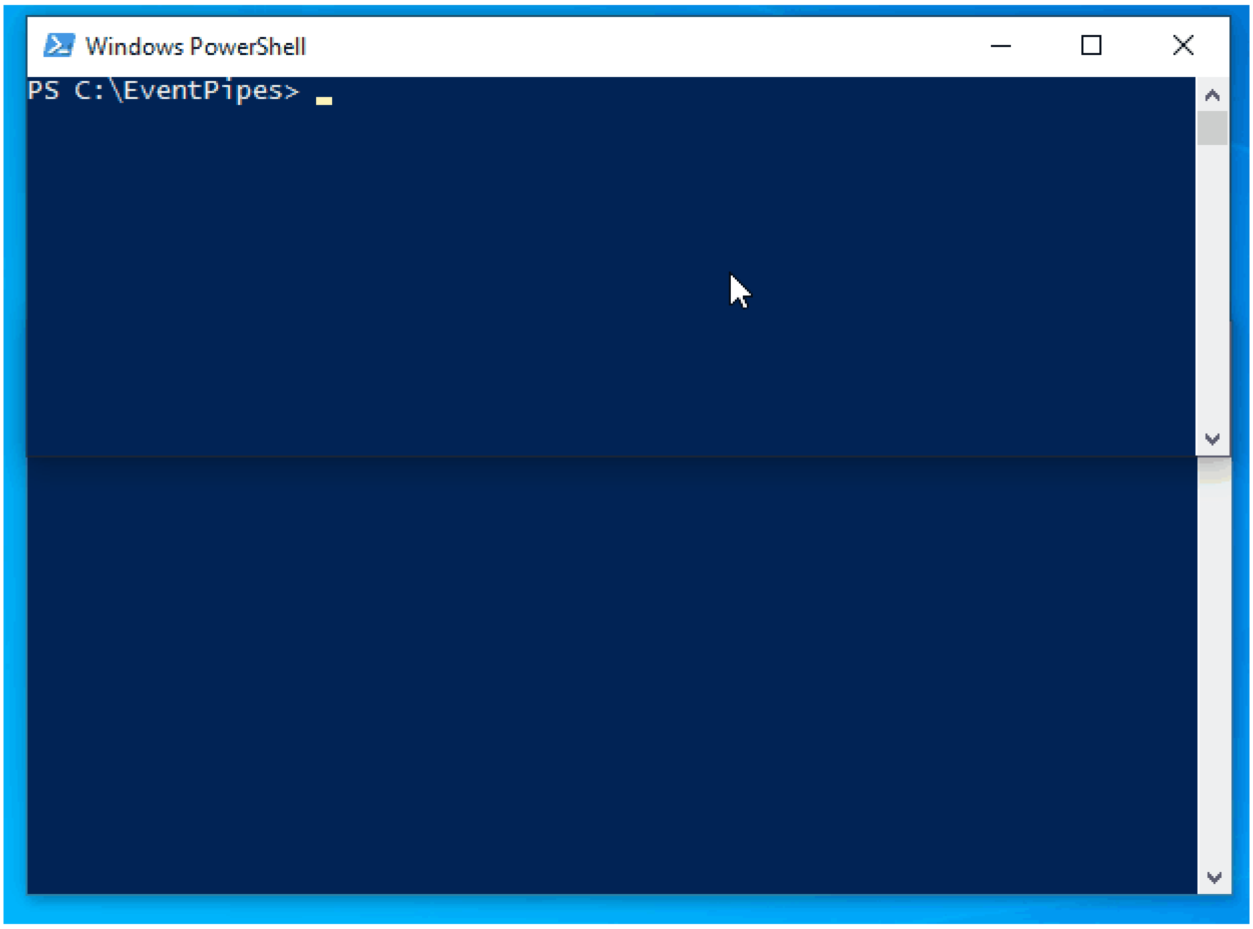
x86matthew - EventPipe - An IPC method to transfer binary data between processes using event objects

web.archive.org/web/20220405165724/https://www.x86matthew.com/view_post

EventPipe - An IPC method to transfer binary data between processes using event objects

This post demonstrates an idea that I came up with for covert inter-process communication.



```
PS C:\EventPipes> _
```

When two processes on a system need to communicate with each other, they would generally use common methods such as named pipes or shared memory. However, these are fairly easy to detect and monitor - I decided to come up with a new method using Windows event objects.

Event handles have only two states - on and off. They are often used for general multi-threading synchronisation, and usually wouldn't warrant any further investigation.

My basic theory is as follows:

1. Create 256 event objects in the receiver program using CreateEvent.
2. Call WaitForMultipleObjects on all of the above event handles in the receiver program.
3. The sender program opens all 256 event objects created by the receiver program using

OpenEvent.
4. Call SetEvent in the sender program to trigger the corresponding event for each byte to transmit (0-255).

There are a couple of issues with the theory above. As most Windows developers will already know, the WaitForMultipleObjects function only allows a maximum of 64 handles. The other problem is synchronisation - we need to know the exact sequence in which the events were originally triggered.

To get around the above issues, the following changes are required:

1. Instead of using 256 event objects (one for each byte), we will split each byte into two events - we will call this "high" and "low" in base-16. For example, when sending 0x95, we would trigger the 0x9 "high" event, and the 0x5 "low" event. This means we only need to wait for 16 event handles at any one time using WaitForMultipleObjects, rather than the full 256. As a side-effect, this also means the total number of event objects is reduced to 32 (16 high + 16 low).
2. To fix the synchronisation issues, we will create one extra event handle called the "acknowledgement" flag. This event will be triggered in the receiver program after each byte has been processed. The sender program will only send the next byte after the acknowledgement flag has been set.

I have created a library of functions to demonstrate this concept:

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

#define EVENT_DATA_HIGH_COUNT 16
#define EVENT_DATA_LOW_COUNT 16

struct EventPipeObjectStruct
{
HANDLE hEventDataHigh[EVENT_DATA_HIGH_COUNT];
HANDLE hEventDataLow[EVENT_DATA_LOW_COUNT];
HANDLE hEventAck;
};

DWORD EventPipe_GetHandles(char *pName, DWORD dwCreate,
EventPipeObjectStruct *pEventPipeObject)
{
char szEventName[512];
EventPipeObjectStruct EventPipeObject;

// create "high" data event handles
for(DWORD i = 0; i < EVENT_DATA_HIGH_COUNT; i++)
```

```
{
// set current event name
memset(szEventName, 0, sizeof(szEventName));
_snprintf(szEventName, sizeof(szEventName) - 1, "EventPipe_%s_H%u", pName, i);

if(dwCreate == 0)
{
// open existing object
EventPipeObject.hEventDataHigh[i] = OpenEvent(EVENT_ALL_ACCESS, 0,
szEventName);
}
else
{
// create new object
EventPipeObject.hEventDataHigh[i] = CreateEvent(NULL, 0, 0, szEventName);
}

// check for errors
if(EventPipeObject.hEventDataHigh[i] == NULL)
{
return 1;
}
}

// create "low" data event handles
for(i = 0; i < EVENT_DATA_LOW_COUNT; i++)
{
// set current event name
memset(szEventName, 0, sizeof(szEventName));
_snprintf(szEventName, sizeof(szEventName) - 1, "EventPipe_%s_L%u", pName, i);

if(dwCreate == 0)
{
// open existing object
EventPipeObject.hEventDataLow[i] = OpenEvent(EVENT_ALL_ACCESS, 0,
szEventName);
}
else
{
// create new object
EventPipeObject.hEventDataLow[i] = CreateEvent(NULL, 0, 0, szEventName);
}

// check for errors
if(EventPipeObject.hEventDataLow[i] == NULL)
{
return 1;
}
}

// create acknowledgement event
memset(szEventName, 0, sizeof(szEventName));
_snprintf(szEventName, sizeof(szEventName) - 1, "EventPipe_%s_A", pName);
if(dwCreate == 0)
{
// open existing object
EventPipeObject.hEventAck = OpenEvent(EVENT_ALL_ACCESS, 0, szEventName);
}
else
{
// create new object
EventPipeObject.hEventAck = CreateEvent(NULL, 0, 0, szEventName);
}

if(EventPipeObject.hEventAck == NULL)
{
return 1;
}

// store data
memcpy((void*)pEventPipeObject, (void*)&EventPipeObject, sizeof(EventPipeObject));

return 0;
}

DWORD EventPipe_CreateReceiver(char *pName, EventPipeObjectStruct
*pEventPipeObject)
{
// create event handles
if(EventPipe_GetHandles(pName, 1, pEventPipeObject) != 0)
{
return 1;
}

return 0;
}

DWORD EventPipe_Open(char *pName, EventPipeObjectStruct *pEventPipeObject)
{
// open event handles
```

```
if(EventPipe_GetHandles(pName, 0, pEventPipeObject) != 0)
{
return 1;
}

return 0;
}

DWORD EventPipe_RecvRawByte(EventPipeObjectStruct *pEventPipeObject, BYTE
*pByte)
{
DWORD dwEventDataHighValue = 0;
DWORD dwEventDataLowValue = 0;
BYTE bByte = 0;

// wait for "high" data value
dwEventDataHighValue = WaitForMultipleObjects(EVENT_DATA_HIGH_COUNT,
pEventPipeObject->hEventDataHigh, 0, INFINITE);
if(dwEventDataHighValue >= EVENT_DATA_HIGH_COUNT)
{
return 1;
}

// wait for "low" data value
dwEventDataLowValue = WaitForMultipleObjects(EVENT_DATA_LOW_COUNT,
pEventPipeObject->hEventDataLow, 0, INFINITE);
if(dwEventDataLowValue >= EVENT_DATA_LOW_COUNT)
{
return 1;
}

// calculate byte value from high/low value
bByte = (BYTE)((dwEventDataHighValue * EVENT_DATA_LOW_COUNT) +
dwEventDataLowValue);

// set acknowledgement event
if(SetEvent(pEventPipeObject->hEventAck) == 0)
{
return 1;
}

// store byte value
*pByte = bByte;

return 0;
```

```
}

DWORD EventPipe_RecvRawData(EventPipeObjectStruct *pEventPipeObject, BYTE
*pData, DWORD dwLength)
{
BYTE *pCurrPtr = NULL;

// receive all requested bytes
pCurrPtr = pData;
for(DWORD i = 0; i < dwLength; i++)
{
// get current byte
if(EventPipe_RecvRawByte(pEventPipeObject, pCurrPtr) != 0)
{
return 1;
}

// increase ptr
pCurrPtr++;
}

return 0;
}

DWORD EventPipe_RecvDataBlock(EventPipeObjectStruct *pEventPipeObject, BYTE
*pData, DWORD dwMaxLength, DWORD *pdwDataLength)
{
DWORD dwBlockLength = 0;

// get block length
if(EventPipe_RecvRawData(pEventPipeObject, (BYTE*)&dwBlockLength,
sizeof(DWORD)) != 0)
{
return 1;
}

// validate length
if(dwBlockLength > dwMaxLength)
{
return 1;
}

// get block data
if(EventPipe_RecvRawData(pEventPipeObject, pData, dwBlockLength) != 0)
{
```

```
return 1;
}

if(pdwDataLength != NULL)
{
// store block length
*pdwDataLength = dwBlockLength;
}

return 0;
}

DWORD EventPipe_SendRawByte(EventPipeObjectStruct *pEventPipeObject, BYTE
bByte)
{
// set "high" data event
if(SetEvent(pEventPipeObject->hEventDataHigh[bByte / EVENT_DATA_LOW_COUNT])
== 0)
{
return 1;
}

// set "low" data event
if(SetEvent(pEventPipeObject->hEventDataLow[bByte % EVENT_DATA_LOW_COUNT])
== 0)
{
return 1;
}

// wait for acknowledgement
if(WaitForSingleObject(pEventPipeObject->hEventAck, INFINITE) != 0)
{
return 1;
}

return 0;
}

DWORD EventPipe_SendRawData(EventPipeObjectStruct *pEventPipeObject, BYTE
*pData, DWORD dwLength)
{
BYTE *pCurrPtr = NULL;

// send all requested bytes
pCurrPtr = pData;
```

```
for(DWORD i = 0; i < dwLength; i++)
{
// send current byte
if(EventPipe_SendRawByte(pEventPipeObject, *pCurrPtr) != 0)
{
return 1;
}

// increase ptr
pCurrPtr++;
}

return 0;
}

DWORD EventPipe_SendDataBlock(EventPipeObjectStruct *pEventPipeObject, BYTE
*pData, DWORD dwLength)
{
// send block length
if(EventPipe_SendRawData(pEventPipeObject, (BYTE*)&dwLength, sizeof(DWORD)) !=
0)
{
return 1;
}

// send block data
if(EventPipe_SendRawData(pEventPipeObject, pData, dwLength) != 0)
{
return 1;
}

return 0;
}
```

Receiver program:

```
int main()
{
EventPipeObjectStruct EventPipeObject;
char szRecvString[512];
DWORD dwDataLength = 0;

memset((void*)&EventPipeObject;, 0, sizeof(EventPipeObject));
if(EventPipe_CreateReceiver("x86matthew", &EventPipeObject;) != 0)
{
return 1;
}

for(;;)
{
memset(szRecvString, 0, sizeof(szRecvString));
if(EventPipe_RecvDataBlock(&EventPipeObject;, (BYTE*)szRecvString,
sizeof(szRecvString) - 1, &dwDataLength;) != 0)
{
return 1;
}

printf("Received %u bytes: '%s'\n", dwDataLength, szRecvString);
}

return 0;
}
```

Sender program:

```
int main(int argc, char *argv[])
{
EventPipeObjectStruct EventPipeObject;

if(argc != 2)
{
return 1;
}

memset((void*)&EventPipeObject;, 0, sizeof(EventPipeObject));
if(EventPipe_Open("x86matthew", &EventPipeObject;) != 0)
{
return 1;
}

if(EventPipe_SendDataBlock(&EventPipeObject;, (BYTE*)argv[1], strlen(argv[1])) != 0)
{
return 1;
}
return 0;
}
```

Potential future improvements: - Communication is currently one-way only - add duplex support.
- The event objects are named to allow us to open the handles in the remote process - make them "anonymous" and duplicate the handles into the remote process.