

Unit OS3: Concurrency

3.4. Windows APIs for Synchronization and Inter-Process Communication

Windows Operating System Internals - by David A. Solomon and Mark E. Russinovich with Andreas Polze

Roadmap for Section 3.4.

- Windows API constructs for synchronization and interprocess communication
- Synchronization
 - Critical sections
 - Mutexes
 - Semaphores
 - Event objects
- Synchronization through interprocess communication
 - Anonymous pipes
 - Named pipes
 - Mailslots

3

Critical Sections

```
VOID InitializeCriticalSection( LPCRITICAL_SECTION sec );
VOID DeleteCriticalSection( LPCRITICAL_SECTION sec );

VOID EnterCriticalSection( LPCRITICAL_SECTION sec );
VOID LeaveCriticalSection( LPCRITICAL_SECTION sec );
BOOL TryEnterCriticalSection ( LPCRITICAL_SECTION sec );
```

Only usable from within the same process

- Critical sections are initialized and deleted but do not have handles
- Only one thread at a time can be in a critical section
- A thread can enter a critical section multiple times - however, the number of Enter- and Leave-operations must match
- Leaving a critical section before entering it may cause deadlocks
- No way to test whether another thread is in a critical section

4

Critical Section Example

```
/* counter is global, shared by all threads */
volatile int counter = 0;
CRITICAL_SECTION crit;
InitializeCriticalSection ( &crit );

/* ... main loop in any of the threads */
while (!done) {
    _try {
        EnterCriticalSection ( &crit );
        counter += local_value;
        LeaveCriticalSection ( &crit );
    }
    _finally { LeaveCriticalSection ( &crit ); }
}
DeleteCriticalSection( &crit );
```

5

Synchronizing Threads with Kernel Objects

```
DWORD WaitForSingleObject( HANDLE hObject, DWORD dwTimeout );  
  
DWORD WaitForMultipleObjects( DWORD cObjects,  
                              LPHANDLE lpHandles, BOOL bWaitAll,  
                              DWORD dwTimeout );
```

The following kernel objects can be used to synchronize threads:

- Processes
- Threads
- Files
- Console input
- File change notifications
- Mutexes
- Events (auto-reset + manual-reset)
- Waitable timers

6

Wait Functions - Details

- **WaitForSingleObject():**
 - hObject specifies kernel object
 - dwTimeout specifies wait time in msec
 - dwTimeout == 0 - no wait, check whether object is signaled
 - dwTimeout == INFINITE - wait forever
- **WaitForMultipleObjects():**
 - cObjects <= MAXIMUM_WAIT_OBJECTS (64)
 - lpHandles - pointer to array identifying these objects
 - bWaitAll - whether to wait for first signaled object or all objects
 - Function returns index of first signaled object
- **Side effects:**
 - Mutexes, auto-reset events and waitable timers will be reset to non-signaled state after completing wait functions

7

Mutexes

```
HANDLE CreateMutex( LPSECURITY_ATTRIBUTE lpsa,
                   BOOL fInitialOwner, LPTSTR lpszMutexName );

HANDLE OpenMutex( LPSECURITY_ATTRIBUTE lpsa,
                 BOOL fInitialOwner, LPTSTR lpszMutexName );

BOOL ReleaseMutex( HANDLE hMutex );
```

Mutexes work across processes

- First thread has to call `CreateMutex()`
- When sharing a mutex, second thread (process) calls `CreateMutex()` or `OpenMutex()`
- `fInitialOwner == TRUE` gives creator immediate ownership
- Threads acquire mutex ownership using `WaitForSingleObject()` or `WaitForMultipleObjects()`
- `ReleaseMutex()` gives up ownership
- `CloseHandle()` will free mutex object

8

Mutex Example

```
/* counter is global, shared by all threads */
volatile int done, counter = 0;
HANDLE mutex = CreateMutex( NULL, FALSE, NULL );

/* main loop in any of the threads, ret is local */
DWORD ret;
while (!done) {
    ret = WaitForSingleObject( mutex, INFINITE );
    if (ret == WAIT_OBJECT_0)
        counter += local_value;
    else /* mutex was abandoned */
        break; /* exit the loop */
    ReleaseMutex( mutex );
}
CloseHandle( mutex );
```

9

Comparison - POSIX mutexes

- POSIX pthreads specification supports mutexes
 - Synchronization among threads in same process
- Five basic functions:
 - pthread_mutex_init()
 - pthread_mutex_destroy()
 - pthread_mutex_lock()
 - pthread_mutex_unlock()
 - pthread_mutex_trylock()
- Comparison:
 - pthread_mutex_lock() will block - equivalent to WaitForSingleObject(hMutex);
 - pthread_mutex_trylock() is nonblocking (polling) - equivalent to WaitForSingleObject() with timeout == 0

10

Semaphores

```
HANDLE CreateSemaphore( LPSECURITY_ATTRIBUTE lpsa,
                        LONG cSemInit, LONG cSemMax,
                        LPTSTR lpszSemName );

HANDLE OpenSemaphore( LPSECURITY_ATTRIBUTE lpsa,
                     LONG cSemInit, LONG cSemMax,
                     LPTSTR lpszSemName );

HANDLE ReleaseSemaphore( HANDLE hSemaphore,
                        LONG cReleaseCount, LPLONG lpPreviousCount );
```

- Semaphore objects are used for resource counting
 - A semaphore is signaled when count > 0
- Threads/processes use wait functions
 - Each wait function decreases semaphore count by 1
 - ReleaseSemaphore() may increment count by any value
 - ReleaseSemaphore() returns old semaphore count

11

Events

```
HANDLE CreateEvent( LPSECURITY_ATTRIBUTE lpsa,
                  BOOL fManualReset, BOOL flnitialState
                  LPTSTR lpszEventName );
BOOL SetEvent( HANDLE hEvent );
BOOL ResetEvent( HANDLE hEvent );
BOOL PulseEvent( HANDLE hEvent );
```

- Multiple threads can be released when a single event is signaled (barrier synchronization)
 - Manual-reset event can signal several thread simultaneously; must be reset manually
 - PulseEvent() will release all threads waiting on a manual-reset event and automatically reset the event
 - Auto-reset event signals a single thread; event is reset automatically
- flnitialState == TRUE - create event in signaled state

12

Comparison - POSIX condition variables

- pthread's condition variables are comparable to events
 - pthread_cond_init()
 - pthread_cond_destroy()
- Wait functions:
 - pthread_cond_wait()
 - pthread_cond_timedwait()
- Signaling:
 - pthread_cond_signal() - one thread
 - pthread_cond_broadcast() - all waiting threads
- No exact equivalent to manual-reset events

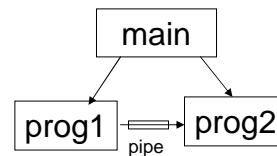
13

Anonymous pipes

```
BOOL CreatePipe( PHANDLE phRead,  
                PHANDLE phWrite,  
                LPSECURITY_ATTRIBUTES lpsa,  
                DWORD cbPipe )
```

Half-duplex character-based IPC

- cbPipe: pipe byte size; zero == default
- Read on pipe handle will block if pipe is empty
- Write operation to a full pipe will block
- Anonymous pipes are oneway



14

I/O Redirection using an Anonymous Pipe

```
/* Create default size anonymous pipe, handles are inheritable. */  
if (!CreatePipe (&hReadPipe, &hWritePipe, &PipeSA, 0)) {  
    fprintf(stderr, "Anon pipe create failed\n"); exit(1);  
}  
/* Set output handle to pipe handle, create first processes. */  
StartInfoCh1.hStdInput = GetStdHandle (STD_INPUT_HANDLE);  
StartInfoCh1.hStdError = GetStdHandle (STD_ERROR_HANDLE);  
StartInfoCh1.hStdOutput = hWritePipe;  
StartInfoCh1.dwFlags = STARTF_USESTDHANDLES;  
  
if (!CreateProcess (NULL, (LPTSTR)Command1, NULL, NULL, TRUE,  
    0, NULL, NULL, &StartInfoCh1, &ProcInfo1)) {  
    fprintf(stderr, "CreateProc1 failed\n"); exit(2);  
}  
CloseHandle (hWritePipe);
```

15

Pipe example (contd.)

```
/* Repeat (symmetrically) for the second process. */
StartInfoCh2.hStdInput = hReadPipe;
StartInfoCh2.hStdError = GetStdHandle (STD_ERROR_HANDLE);
StartInfoCh2.hStdOutput = GetStdHandle (STD_OUTPUT_HANDLE);
StartInfoCh2.dwFlags = STARTF_USESTDHANDLES;

if (!CreateProcess (NULL, (LPTSTR)targv, NULL, NULL, TRUE, /* Inherit
handles. */
    0, NULL, NULL, &StartInfoCh2, &ProcInfo2)) {
    fprintf(stderr, "CreateProc2 failed\n"); exit(3);
}
CloseHandle (hReadPipe);

/* Wait for both processes to complete. */
WaitForSingleObject (ProcInfo1.hProcess, INFINITE);
WaitForSingleObject (ProcInfo2.hProcess, INFINITE);
CloseHandle (ProcInfo1.hThread); CloseHandle (ProcInfo1.hProcess);
CloseHandle (ProcInfo2.hThread); CloseHandle (ProcInfo2.hProcess);
return 0;
```

16

Named Pipes

- Message oriented:
 - Reading process can read varying-length messages precisely as sent by the writing process
- Bi-directional
 - Two processes can exchange messages over the same pipe
- Multiple, independent instances of a named pipe:
 - Several clients can communicate with a single server using the same instance
 - Server can respond to client using the same instance
- Pipe can be accessed over the network
 - location transparency
- Convenience and connection functions

17

Using Named Pipes

```
HANDLE CreateNamedPipe (LPCTSTR lpszPipeName,  
                        DWORD fdwOpenMode, DWORD fdwPipMode  
                        DWORD nMaxInstances, DWORD cbOutBuf,  
                        DWORD cbInBuf, DWORD dwTimeOut,  
                        LPSECURITY_ATTRIBUTES lpsa );
```

- lpszPipeName: \\.\pipe\[path]pipename
 - Not possible to create a pipe on remote machine (. – local machine)
- fdwOpenMode:
 - PIPE_ACCESS_DUPLEX, PIPE_ACCESS_INBOUND, PIPE_ACCESS_OUTBOUND
- fdwPipeMode:
 - PIPE_TYPE_BYTE or PIPE_TYPE_MESSAGE
 - PIPE_READMODE_BYTE or PIPE_READMODE_MESSAGE
 - PIPE_WAIT or PIPE_NOWAIT (will ReadFile block?)

Use same flag settings for all instances of a named pipe

18

Named Pipes (contd.)

- nMaxInstances:
 - Number of instances,
 - PIPE_UNLIMITED_INSTANCES: OS choice based on resources
- dwTimeOut
 - Default time-out period (in msec) for WaitNamedPipe()
- First CreateNamedPipe creates named pipe
 - Closing handle to last instance deletes named pipe
- Polling a pipe:
 - Nondestructive – is there a message waiting for ReadFile

```
BOOL PeekNamedPipe (HANDLE hPipe,  
                   LPVOID lpvBuffer, DWORD cbBuffer,  
                   LPDWORD lpcbRead, LPDWORD lpcbAvail,  
                   LPDWORD lpcbMessage);
```

19

Named Pipe Client Connections

- CreateFile with named pipe name:
 - \\.\pipe\[path]pipename
 - \\servername\pipe\[path]pipename
 - First method gives better performance (local server)
- Status Functions:
 - GetNamedPipeHandleState
 - SetNamedPipeHandleState
 - GetNamedPipeInfo

20

Convenience Functions

- WriteFile / ReadFile sequence:

```
BOOL TransactNamedPipe( HANDLE hNamedPipe,  
                        LPVOID lpvWriteBuf, DWORD cbWriteBuf,  
                        LPVOID lpvReadBuf, DWORD cbReadBuf,  
                        LPDWORD lpcbRead, LPOVERLAPPED lpa);
```

- CreateFile / WriteFile / ReadFile / CloseHandle:
 - dwTimeOut: NMPWAIT_NOWAIT, NMPWAIT_WIAT_FOREVER,
NMPWAIT_USE_DEFAULT_WAIT

```
BOOL CallNamedPipe( LPCTSTR lpszPipeName,  
                   LPVOID lpvWriteBuf, DWORD cbWriteBuf,  
                   LPVOID lpvReadBuf, DWORD cbReadBuf,  
                   LPDWORD lpcbRead, DWORD dwTimeOut);
```

21

Server: eliminate the polling loop

```
BOOL ConnectNamedPipe (HANDLE hNamedPipe,  
LPOVERLAPPED lpo );
```

- lpo == NULL:
 - Call will return as soon as there is a client connection
 - Returns false if client connected between CreateNamed Pipe call and ConnectNamedPipe()
- Use DisconnectNamedPipe to free the handle for connection from another client
- WaitNamedPipe():
 - Client may wait for server's ConnectNamedPipe()
- Security rights for named pipes:
 - GENERIC_READ, GENERIC_WRITE, SYNCHRONIZE

22

Comparison with UNIX

- UNIX FIFOs are similar to a named pipe
 - FIFOs are half-duplex
 - FIFOs are limited to a single machine
 - FIFOs are still byte-oriented, so its easiest to use fixed-size records in client/server applications
 - Individual read/writes are atomic
- A server using FIFOs must use a separate FIFO for each client's response, although all clients can send requests via a single, well known FIFO
- Mkfifo() is the UNIX counterpart to CreateNamedPipe()
- Use sockets for networked client/server scenarios

23

Client Example using Named Pipe

```
WaitNamedPipe (ServerPipeName, NMPWAIT_WAIT_FOREVER);
hNamedPipe = CreateFile (ServerPipeName, GENERIC_READ | GENERIC_WRITE,
    0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
if (hNamedPipe == INVALID_HANDLE_VALUE) {
    fptinf(stderr, Failure to locate server.\n"); exit(3);
}

/* Write the request. */
WriteFile (hNamedPipe, &Request, MAX_RQRS_LEN, &nWrite, NULL);

/* Read each response and send it to std out. */
while (ReadFile (hNamedPipe, Response.Record, MAX_RQRS_LEN, &nRead, NULL))
    printf ("%s", Response.Record);

CloseHandle (hNamedPipe);
return 0;
```

24

Server Example Using a Named Pipe

```
hNamedPipe = CreateNamedPipe (SERVER_PIPE, PIPE_ACCESS_DUPLEX,
    PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE | PIPE_WAIT,
    1, 0, 0, CS_TIMEOUT, pNPSA);
while (!Done) {
    printf ("Server is awaiting next request.\n");
    if (!ConnectNamedPipe (hNamedPipe, NULL)
        || !ReadFile (hNamedPipe, &Request, RQ_SIZE, &nXfer, NULL)) {
        fprintf(stderr, "Connect or Read Named Pipe error\n"); exit(4);
    }
    printf("Request is: %s\n", Request.Record);
    /* Send the file, one line at a time, to the client. */
    fp = fopen (File, "r");
    while ((fgets (Response.Record, MAX_RQRS_LEN, fp) != NULL))
        WriteFile (hNamedPipe, &Response.Record,
            (strlen(Response.Record) + 1) * TSIZE, &nXfer, NULL);
    fclose (fp);
    DisconnectNamedPipe (hNamedPipe);
} /* End of server operation. */
```

25

Windows IPC - Mailslots

Mailslots bear some nasty implementation details; they are almost never used

- Broadcast mechanism:
 - One-directional
 - Multiple writers/multiple readers (frequently: one-to-many comm.)
 - Message delivery is unreliable
 - Can be located over a network domain
 - Message lengths are limited (w2k: < 426 byte)
- Operations on the mailslot:
 - Each reader (server) creates mailslot with CreateMailslot()
 - Write-only client opens mailslot with CreateFile() and uses WriteFile() – open will fail if there are no waiting readers
 - Client's message can be read by all servers (readers)
- Client lookup: *\mailslot\mailslotname
 - Client will connect to every server in network domain

26

Locate a server via mailslot

Mailslot Servers

App client 0

```
hMS = CreateMailslot(
    "\\.\mailslot\status");
ReadFile(hMS, &ServStat);
/* connect to server */
```

App client n

```
hMS = CreateMailslot(
    "\\.\mailslot\status");
ReadFile(hMS, &ServStat);
/* connect to server */
```

Message is sent periodically

Mailslot Client

App Server

```
While (...) {
    Sleep(...);
    hMS = CreateFile(
        "\\.\mailslot\status");
    ...
    WriteFile(hMS, &StatInfo);
}
```

27

Creating a mailslot

```
HANDLE CreateMailslot(LPCTSTR lpszName,  
    DWORD cbMaxMsg,  
    DWORD dwReadTimeout,  
    LPSECURITY_ATTRIBUTES lpsa);
```

- `lpszName` points to a name of the form
 - `\\.mailslot[path]name`
 - Name must be unique; mailslot is created locally
- `cbMaxMsg` is msg size in byte
- `dwReadTimeout`
 - Read operation will wait for so many msec
 - 0 – immediate return
 - `MAILSLOT_WAIT_FOREVER` – infinite wait

28

Opening a mailslot

- `CreateFile` with the following names:
 - `\\.mailslot[path]name` - retrieve handle for local mailslot
 - `\\host\mailslot[path]name` - retrieve handle for mailslot on specified host
 - `\\domain\mailslot[path]name` - returns handle representing all mailslots on machines in the domain
 - `*\mailslot[path]name` - returns handle representing mailslots on machines in the system's primary domain: max msg. len: 400 bytes
 - Client must specify `FILE_SHARE_READ` flag
- `GetMailslotInfo()` and `SetMailslotInfo()` are similar to their named pipe counterparts

29

Further Reading

- Mark E. Russinovich and David A. Solomon, Microsoft Windows Internals, 4th Edition, Microsoft Press, 2004.
 - Chapter 3 - System Mechanisms
 - Synchronization (from pp.149)
 - Named Pipes and Mailslots (from pp. 804)
- Jeffrey Richter, Programming Applications for Microsoft Windows, 4th Edition, Microsoft Press, September 1999.
 - Chapter 10 - Thread Synchronization
 - Critical Sections, Mutexes, Semaphores, Events (from pp. 315)
- Johnson M. Hart, Win32 System Programming: A Windows® 2000 Application Developer's Guide, 2nd Edition, Addison-Wesley, 2000.

30

Source Code References

- Windows Research Kernel sources
 - \base\ntos\ke – primitive kernel support
 - eventobj.c - Event object
 - mutntobj.c – Mutex object
 - semphobj.c – Semaphore object
 - timerobj.c, timersup.c – Timers
 - wait.c, waitsup.c – Wait support
 - \base\ntos\lex – executive object (layered on kernel support)
 - Event.c – Event object
 - Mutant.c – Mutex object
 - Semaphore.c – Semaphore object
 - Timer.c – Timer object
 - \base\ntos\inc\ke.h, ex.h – structure/type definitions

31