

Windows 95/NT System
Programming



Interprocess Communication

Main Topics

- Shared Memory and Synchronization Techniques
 - Mutex Objects
 - Signals and semaphores
 - Critical Sections
 - Deadlock
- Pipes
 - Anonymous Pipes and Named Pipes
 - Reading and Writing Pipes

Wait Functions

- Win32 `wait` functions enable a thread to block its own execution
 - return when conditions specified in *wait* are satisfied
 - include timeout interval and handles to one or more synchronization objects

Wait Functions

- *WaitForSingleObject*
- *WaitForSingleObjectEx* (alertable wait)
- *WaitForMultipleObjects*
- *WaitForMultipleObjectsEx*
- *MsgWaitForMultipleObjects* (can return when specified type of input is available)

Synchronization Object Overview

- A `synchronization object` is an object whose handle can be specified in wait function to coordinate the execution of multiple threads
- Objects used exclusively for synchronization
 - `event`
 - `mutex`
 - `semaphore`
- Other synchronization objects:
 - `change notification, console input, process, thread`

Mutex Objects

- Signaled when not owned by any thread, non-signaled when owned by thread
- Only one thread at a time can own a mutex
- Created using `CreateMutex`
 - other threads with handle to mutex can own mutex object
- Thread obtains ownership by specifying object handle in `OpenMutex` wait function
- Release using `ReleaseMutex` function


Mutex Objects (cont'd)

One thread calls *CreateMutex*
Pass mutex handles to other cooperating threads
loop forever
 WaitForSingleObject
 ... critical section ...
 ReleaseMutex
end loop


Synchronization Example: Using Named Objects & Mutex

```
HANDLE hMutex;  
DWORD dwErr;  
hMutex = CreateMutex ( NULL,      /* no security descriptor */  
                     FALSE,      /* mutex not owned */  
                     "DatabaseMutex"); /* object name */  
if (hMutex == NULL)  
    printf ("CreateMutex error: %d\n", GetLastError() );  
else  
    if (GetLastError() == ERROR_ALREADY_EXISTS)  
        printf ("CreateMutex opened existing mutex\n");  
    else  
        printf("CreateMutex created new mutex\n");
```

One process
creates
mutex object



Another process
opens handle to
existing mutex



```
HANDLE hMutex;  
hMutex = OpenMutex (MUTEX_ALL_ACCESS,  
                   FALSE, /* handle not inheritable */  
                   "DatabaseMutex");  
if (hMutex == NULL)  
    printf ("OpenMutex error: %d\n", GetLastError() );
```


Synchronization Example: Using Named Objects & Mutex

```
dwWaitResult = WaitForSingleObject (hMutex, 5000L); /* 5-second timeout */
switch (dwWaitResult) {
    case WAIT_OBJECT_0: /* obtained mutex ownership */
        try {
            ... write to database ...
        }
        finally {
            if (!ReleaseMutex (hMutex)) { /* release mutex ownership */
                ... error handling ...
            }
            break;
        }
    case WAIT_TIMEOUT:
        return FALSE;
    case WAIT_ABANDONED: /* got ownership of abandoned mutex object */
        return FALSE;
}
```

Critical Section Objects

- Similar to mutex objects, but can be used only by threads of the same process
 - faster than mutex objects
- Can be owned by only one thread at a time
- Must declare `CRITICAL_SECTION` variable and initialize using `InitializeCriticalSection`
- Thread use `EnterCriticalSection` to request ownership and `LeaveCriticalSection` to release ownership
- Use `DeleteCriticalSection` to release system resources

Semaphore Objects

- Limits number of concurrent accesses to a shared resource
- Create using `CreateSemaphore`
- Other threads can open handle to existing semaphore object using `OpenSemaphore`
 - semaphore count decreases by one
- Use `ReleaseSemaphore` to increase semaphore count

Synchronization Example: Using Semaphore Objects

```
HANDLE hSemaphore;  
LONG cMax = 10;  
hSemaphore = CreateSemaphore (NULL, /* no security attributes */  
    cMax, /* initial count */  
    cMax, /* maximum count */  
    NULL); /* unnamed semaphore */  
if (hSemaphore == NULL)  
    ... check for error ...
```

Create
semaphore



Before accessing
shared resource



```
DWORD dwWaitResult;  
dwWaitResult = WaitForSingleObject (hSemaphore, 0L);  
switch (dwWaitResult) {  
    case WAIT_OBJECT_0:  
        ... semaphore signalled ...  
        break;  
    case WAIT_TIMEOUT:  
        ... semaphore nonsignalled ...  
        break;  
}
```

Synchronization Example: Using Semaphore Objects (cont'd)

```
if (!ReleaseSemaphore (  
    hSemaphore,    /* semaphore handle */  
    1,            /* increase count by one */  
    NULL) ) {    /* not interested in previous count */  
    ... handle error ...  
}
```

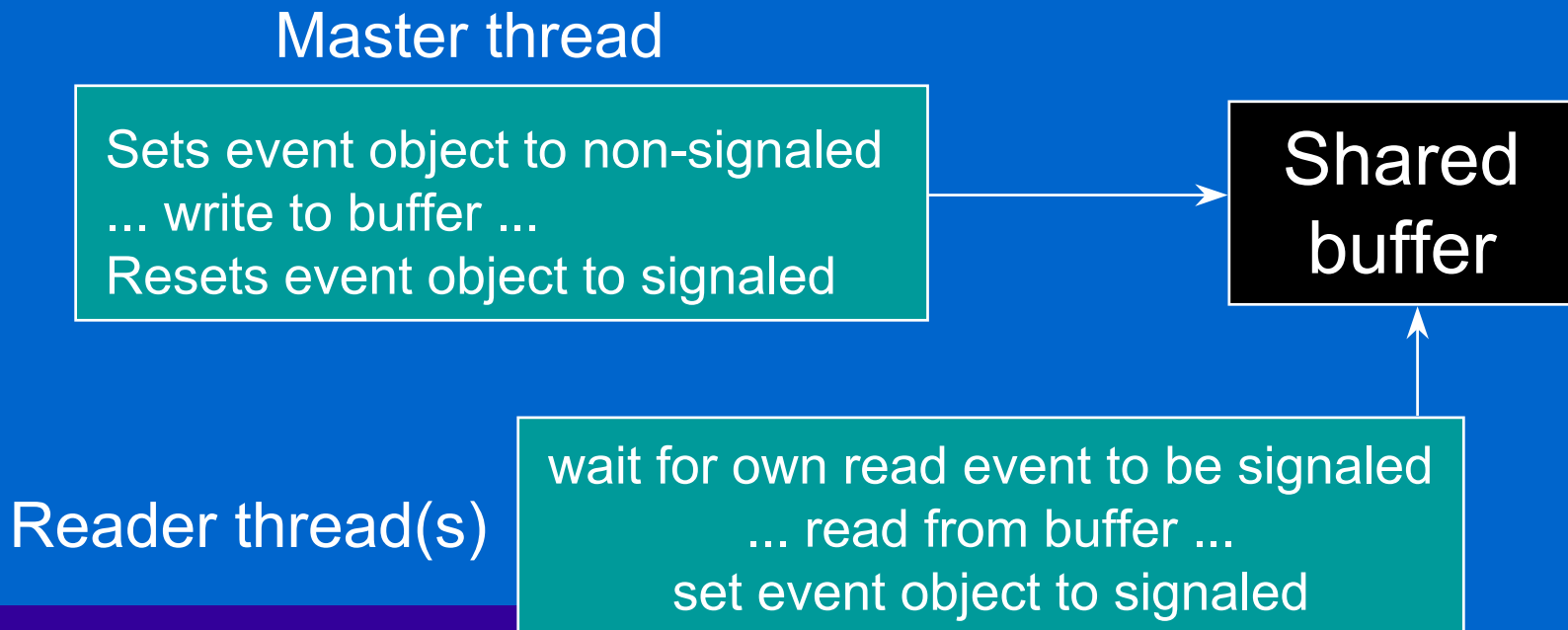
After accessing shared
resource, release semaphore

Event Objects

- Useful for signaling thread when a particular event has occurred
- State can be either signaled or non-signaled
- Create using `CreateEvent`
- Other threads can open a handle to existing event object using `OpenEvent`
- Use `PulseEvent` function to set event object's state to signaled and then reset it to non-signaled after releasing appropriate number of wait threads

Example: Using Event Objects

- Event objects used to prevent several threads from reading from shared memory buffer while master thread is writing to it



Using Event Objects: Master Thread

```
#define NUMTHREADS 4
HANDLE hGlobalWriteEvent;
void CreateEventsAndThreads (void) {
    HANDLE hReadEvents [NumThreads], hThread;
    DWORD i, IDThread;
    if ((hGlobalWriteEvent = CreateEvent (NULL, TRUE, TRUE, "WriteEvent")) == NULL)
        /* error exit */
    /* Create multiple threads and auto-reset event object for each thread */
    for (i=1; i<=NUMTHREADS; i++) {
        hReadEvents[i] = CreateEvent (NULL, FALSE, TRUE, NULL) ;
        if (hReadEvents[i] == NULL) {
            /* error exit */
        }
        hThread = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE) ThreadFunction,
            &hReadEvents[i], /* pass event handle */
            0, &IDThread);
        if (hThread == NULL) {
            /* error exit */
        }
    }
}
```


Master Thread (cont'd)

```
VOID WriteToBuffer(VOID) {
    DWORD dwWaitResult, i;
    if (! ResetEvent(hGlobalWriteEvent) ) {
        /* error exit */
    }
    dwWaitResult = WaitForMultipleObjects(
        NUMTHREADS, hReadEvents, TRUE, INFINITE);
    switch (dwWaitResult) {
        case WAIT_OBJECT_0:
            . /* Write to shared buffer */
            .
            break;
        default:
            printf("Wait error: %d\n", GetLastError());
            ExitProcess(0);
    }
}
```

Master Thread (cont'd)

```
if (! SetEvent(hGlobalWriteEvent) ) {  
    /* error exit */  
}  
for(i = 1; i <= NUMTHREADS; i++)  
    if (! SetEvent(hReadEvents[i]) ) {  
        /* error exit */  
    }  
}
```

Event Example: Reader Threads

```
VOID ThreadFunction(LPVOID lpParam) {
    DWORD dwWaitResult, i;
    HANDLE hEvents[2];

    hEvents[0] = (HANDLE) *lpParam; /* thread's read event */
    hEvents[1] = hGlobalWriteEvent;
    dwWaitResult = WaitForMultipleObjects(
        2, /* number of handles in array */
        hEvents, /* array of event handles */
        TRUE, /* wait till all are signaled */
        INFINITE); /* indefinite wait */
    switch (dwWaitResult) {
    case WAIT_OBJECT_0:
        /* ... Read from the shared buffer ... */
        break;
        /* An error occurred. */
    default:
        printf("Wait error: %d\n", GetLastError());
        ExitThread(0);
    }
    if (! SetEvent(hEvents[0]) ) {
        /* error exit */
    }
}
```

Interprocess Synchronization

- Multiple processes can have handles of same mutex, semaphore, or event object for IPC
- Processes can share object handles using named objects
- Child process created by `CreateProcess` can inherit handle of mutex, event, or semaphore object if `SECURITY_ATTRIBUTES` structure enables inheritance

Duplicating Handles

- DuplicateHandle function creates a duplicate handle that can be used by another specified process
 - creating process must pass handle to other process using interprocess communication

Overlapped I/O

- Win32 API can do synchronous and asynchronous I/O
- Synchronous I/O:
 - returns only when I/O completes
- Asynchronous (overlapped) I/O:
 - returns as soon as the call is issued
 - process is signaled when I/O completes
 - good for time-consuming I/O

Pipes

- A pipe is a communication conduit with two ends
 - a process with a handle to one end can communicate with a process having a handle to the other end
- Can be:
 - one-way: one end read-only; other end write-only
 - two-way: both ends of the pipe can read or write
- Can be anonymous (unnamed) or named

Anonymous Pipes

- Unnamed, one-way pipe that transfers data between related processes
 - parent and child process
 - two child processes of the same parent
- Used for local communication only
- Use `CreatePipe` to create an anonymous pipe with two handles
 - read handle
 - write handle
- After creating pipe, pass one end to another process; usually through inheritance

Anonymous Pipes (cont'd)

- Parent must communicate handle value to child:
 - Parent specifies pipe handle to `SetStdHandle` before creating child
 - Child uses `GetStdHandle` to retrieve handle value when it starts up
- Standard handles are
 - standard input
 - standard output
 - standard error

Reading From Anonymous Pipes

- To read from a pipe, use read handle in call to `ReadFile` function
- `ReadFile` returns when
 - an error occurs
 - when the specified number of bytes has been read
 - when the write end of the pipe is closed

Writing To Anonymous Pipes

- To write to a pipe, use write handle in call to `WriteFile`
- `WriteFile` returns when
 - an error occurs
 - when the specified number of bytes has been written
 - when the read end of the pipe is closed
- If pipe's buffer is full and there are still bytes to write, `WriteFile` does not return until some other process or thread reads from pipe, making more buffer space available

Asynchronous I/O On Pipes

- Asynchronous (overlapped) I/O not supported for anonymous pipes
 - cannot use `ReadFileEx` and `WriteFileEx`
 - overlapped parameter to `ReadFile` and `WriteFile` ignored

Redirecting Standard Output of Child to Pipe

- Call `GetStdHandle` to get current standard output handle
- Call `CreatePipe` to create anonymous pipe
- Call `SetStdHandle` to set standard output to write handle of pipe
- Call `CreateProcess` to create child process which inherits handles from parent
 - **child process use `GetStdHandle` to retrieve handles**
- Call `CloseHandle` to close parent's handle to write end of pipe
- Call `ReadFile` function to read from pipe
 - **parent reads data written to standard output by child process**

Wait Functions

- `WaitForSingleObject`
 - returns when state of specified object is signaled or when timeout elapses
- `WaitForMultipleObjects`
 - returns when state of one of specified objects is signaled or when timeout elapses
- `WaitForSingleObjectEx`,
`WaitForMultipleObjectsEx`
 - similar to first two, but can perform alertable wait when `fAlertable` parameter is `TRUE`
 - functions return when `ReadFileEx` or `WriteFileEx` completion routine is queued for execution