Threads

Thread Essentials

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Objectives

- Upon completion of this unit you will be able to run and control threads
  - Run and end threads
    - CreateThread, _beginthreadex, ExitThread, TerminateThread
  - Wait for threads
    - WaitForSingleObject (WFSO), WaitForMultipleObjects (WFMO)
  - Other thread control functions
    - GetCurrentThread, GetCurrentThreadId,
    - SuspendThread, ResumeThread, etc.

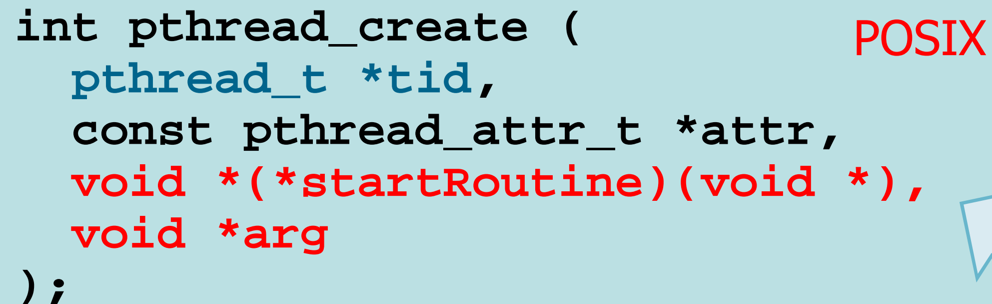
Create Thread

```
HANDLE CreateThread (
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddr,
  LPVOID lpThreadParm,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

- This function allows you to run a thread and it specifies
  - The thread's start address within the process's code
  - A pointer to a thread argument
    - Each thread has a permanent ThreadId and it is usually accessed by a HANDLE

Create Thread

- Return value
  - The handle of the thread, if success
  - A NULL handle, otherwise



POSIX

```
int pthread_create (
  pthread_t *tid,
  const pthread_attr_t *attr,
  void *(*startRoutine)(void *),
  void *arg
);
```

Windows API

```
HANDLE CreateThread (
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddr,
  LPVOID lpThreadParm,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

Create Thread

- Parameters
  - lpsa
    - Security attributes structure
    - Often equal to NULL
  - dwStackSize
    - Byte size for the new thread's stack
    - Use zero to default to the primary thread's stack size (often 1 MB)

```
HANDLE CreateThread (
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddr,
  LPVOID lpThreadParm,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

Create Thread

- lpStartAddr
  - Points to the **function** (within the calling process) to be executed
  - The function accepts a single pointer argument and returns a 32-bit DWORD exit code
- lpThreadParm
  - The pointer passed as the **thread argument**
  - The thread can interpret the argument as a poiner to a structure

```
HANDLE CreateThread (
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddr,
  LPVOID lpThreadParm,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

Create Thread

.NET and Java separate thread creation from thread start. Pthreads does not

- dwCreationFlags
  - If zero, the thread is immediately ready to run
  - If CREATE_SUSPENDED, the new thread will be in the suspended state, requiring a ResumeThread function call to move the thread to the ready state
- lpThreadId
  - Points to a DWORD that receives the new thread's identifier
  - It can be NULL

```
HANDLE CreateThread (
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddr,
  LPVOID lpThreadParm,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

Thread Function

```
DWORD WINAPI ThreadFunction (LPVOID);
```

- Prototype of the thread function
  - It receives a single argument (long poiner to void)
  - It returns a DWORD value

Exit Thread

```
VOID ExitThread (DWORD exitCode);
```

- **ExitThread** is the preferred technique to exit a thread in C language
  - The thread's stack is deallocated on termination
  - All handles referring the thread are signaled
- A thread
  - Will remain in the system until the last handle to it is closed (using **CloseHandle**)
    - Only after Closehandle the thread will be deleted
  - Any other thread can retrieve its exit code **exitCode**
    - See **GetExitCodeThread** for details

Exit Thread

- A common alternative for a thread to exit, is to **return** from the thread function
  - The exit code exitCode can be returned with return
- When the last thread in a process terminates, so does the process itself
- You can terminate a different thread with **TerminateThread**, but this is
  - Dangerous
    - Thread's resources may not be deallocated (e.g., handler not called)
  - Better to let the thread terminate itself

Example

```
HANDLE *threadH;
thread_t *threadD;
DWORD *threadId;

... Allocate thread, threadD, threadId ...

for (i=0; i<N; i++) {
  threadD[i]. ... = ...;
  threadH[i] = CreateThread (NULL, 0,
    (LPTHREAD_START_ROUTINE) threadFunction,
    &threadD[i], 0, &threadId[i]);
  if (threadH[i] == NULL) {
    ExitProcess(0);
  }
}

... Wait for threads ...

for (i=0; i<N; i++) {
  CloseHandle (threadH[i]);
}
```

Array of handles

Parameters embedded into a C structure

Run threads

It can be NULL

Wait for threads

Close handles

Example

```
DWORD WINAPI threadFunction (LPVOID lpParam) {
  thread_t *data;

  data = (thread_t *) lpParam;

  ... Thread body ...

  ExitThread (0);
}
```

Thread function

Data type conversion (cast)

Can be captured by **GetExitCodeThread**

Create Thread

- Nearly all programs (and thread functions) use the C library
- C and C++ make available functions
  - _beginthreadex
  - _endthreadex
- They have the same parameters as CreateThread and ExitThread
- Please remind to include
  - #include <process.h>

Wait for Threads

- Functions WaitForSingleObject (WFSO) and WaitForMultipleObjects (WFMO) allow to wait for thread termination
- These functions, are general purpose
  - They wait for many different types of objects
    - Wait for one or more handles to become "signaled"
    - The handle/handles can represent processes, threads, semaphores, etc.
    - The meaning of "signaled" varies among object types
  - It is possible to specify an optional timeout period

Wait for Threads

```
DWORD WaitForSingleObject (
  HANDLE hObject,
  DWORD dwTimeOut
);
```

- Functions WFSO awaits for a **single** object
  - A single handle, **hObject**, to wait for
  - A timeout limit (**dwTimeOut**) to indicate the timeout in milliseconds
    - Zero means that the function returns immediately after testing the state of the specified objects
    - INFINITE indicates no timeout
      - Wait forever for an “object” to terminate

Wait for Threads

```
DWORD WaitForMultipleObjects (
  DWORD nCount,
  LPHANDLE lpHandles,
  BOOL fWaitAll,
  DWORD dwTimeOut
);
```

- Functions WFMO awaits for **multiple** objects
  - The set of handles in the array **lpHandles** of **nCount** size
  - The handles do not need to be of the same type (e.g., processes, threads, etc.)
    - The number of objects **nCount** should not exceed MAXIMUM_WAIT_OBJECTS (i.e., usually **64**)
    - If the parameter **fWaitAll** if TRUE, WFMO waits for **all** objects to be signaled rather than only **one**

Wait for Threads

- WFSO and WFMO have the following possible return values
  - WAIT_OBJECT_0
    - For WFSO (or WFMO) the (a) single handle is signaled
    - For WFMO, **all** handles are signaled when **fWaitAll** is TRUE
  - WAIT_OBJECT_0 + n (where $0 \leq n < nCount$)
    - With WFMO it is possible to determine which handle was signaled by subtracting WAIT_OBJECT_0 from the return value

```
DWORD WaitForSingleObject (HANDLE hObject, DWORD dwTimeOut);

DWORD WaitForMultipleObjects (DWORD nCount,
   LPHANDLE lpHandles, BOOL fWaitAll, DWORD dwTimeOut);
```

Wait for Threads

- WAIT_TIMEOUT
  - The timeout period elapsed before the wait could be satisfied by a signal
- WAIT_FAILED
  - The call to WFSO or WFMO failed
- WAIT_ABANDONED_0
  - Not possible with processes or threads, used for mutex handles

```
DWORD WaitForSingleObject (HANDLE hObject,
    DWORD dwTimeOut);

DWORD WaitForMultipleObjects (DWORD cObjects,
    LPHANDLE lphObjects, BOOL fWaitAll, DWORD dwTimeOut);
```

Example

Wait for threads one at a time

```
for (i=0; i<N; i++) {
  WaitForSingleObject (threadH[i], INFINITE);
  CloseHandle (threadH[i]);
}
```

Forcing an order in the waiting process may be inefficient

Wait for threads all togheter

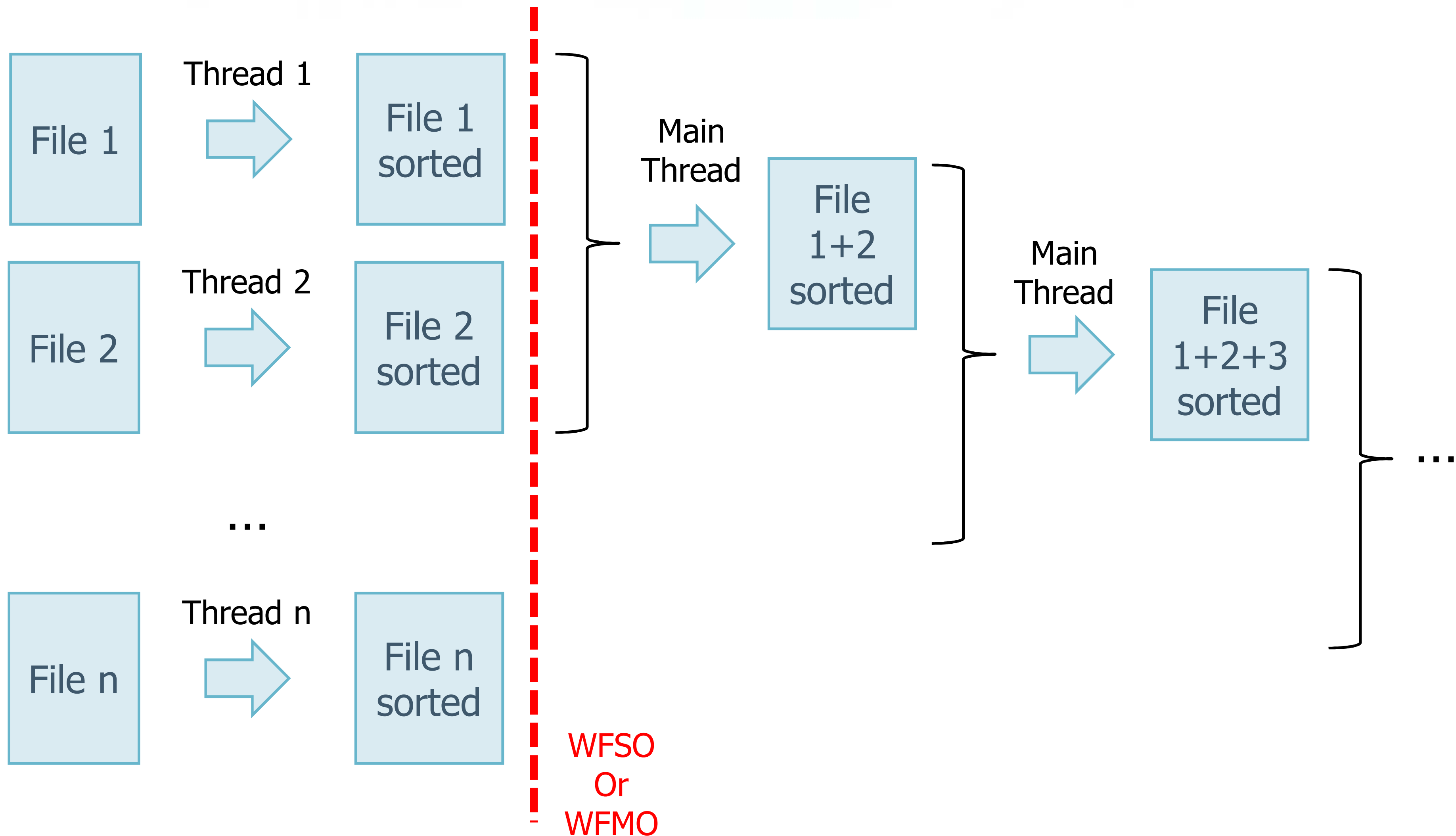
```
WaitForMultipleObjects (N, threadH, TRUE, INFINITE);
for (i=0; i<N; i++) {
  CloseHandle (threadH[i]);
}
```

Can wait for at most MAXIMUM_WAIT_OBJECTS object

Example

Laboratory 04
Exercise 01

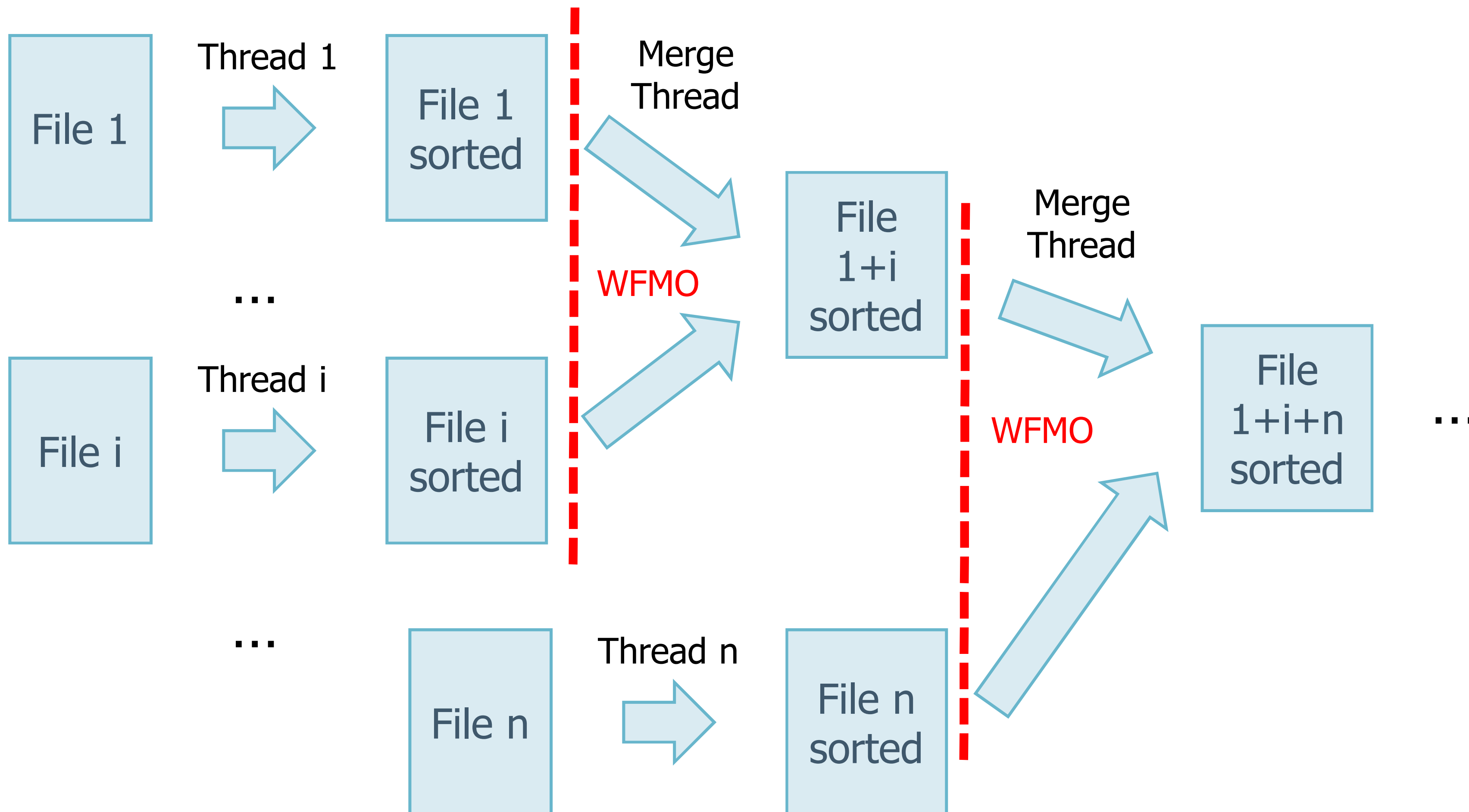
- File sorting (through sort and merge)



Example

Laboratory 04
Exercise 02

- File sorting (through sort and merge)



Example

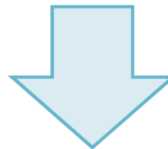
- How can we use WFMO to
  - Wait for more than **MAXIMUM_WAIT_OBJECTS** handles?
  - Wait (and act) for a single thread within a large group of threads

Example

Wait for all threads in a group, then move into the next group

```
WaitForMultipleObjects (N, threadH, TRUE, INFINITE);
```

Wait for a large number of threads



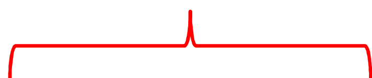
Usually 64

```
for (i=0; i<N; i+=MAXIMUM_WAIT_OBJECTS) {
  WaitForMultipleObjects (
    min (MAXIMUM_WAIT_OBJECTS, N-i),
    &threadH[i], TRUE, INFINITE);
}
```

Reamining Threads

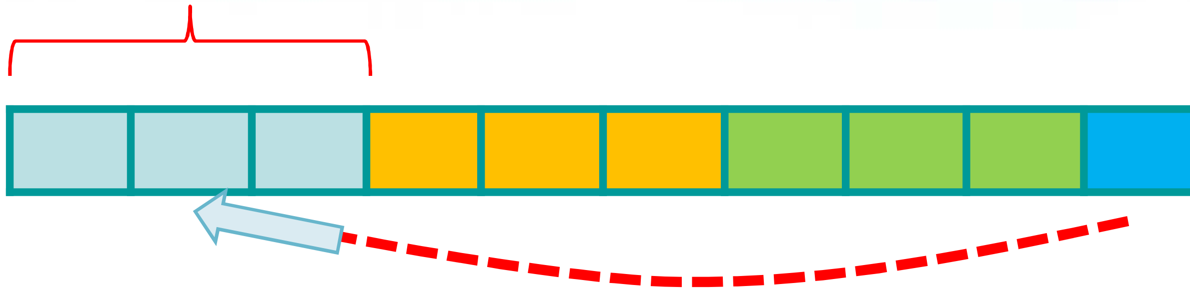
Wait for all (within the same goup) Then move to the next group

MAXIMUM_WAIT_OBJECTS



Example

Wait for a single thread within the first group, then re-arrange the groups




```
while (N > 0) {
  INDEX = WaitForMultipleObjects (
    min (MAXIMUM_WAIT_OBJECTS, N),
    threadH, FALSE, INFINITE);
  index = (int) INDEX - (int) WAIT_OBJECT_0;
  CloseHandle (threadH[index]);
  ...
  threadH[index] = threadH[N-1];
  ... Free threadData[index] ...
  threadData[index] = threadData[N-1];
  N--;
}
```

Wait for a single thread

Re-arrange groups (handles and data)

Thread Identifiers

```
BOOL GetExitCodeThread (
  HANDLE lThread,
  LPDWORD lpExitCode
);
```

- A terminated thread will exist until the last handle to it is closed (by Closehandle)
- Any other thread can retrieve its exit code
  - The code will be returned into **lpExitCode**
  - The value **STILL_ACTIVE** will be returned is the thread is still running

Thread Identifiers

```
HANDLE GetCurrentThread (VOID);

DWORD GetCurrentThreadId (VOID);

DWORD GetThreadId (HANDLE threadHandle);
```

- These functions are use to obain
  - GetCurrent Thread the thread handles
  - GetCurrentTheradId the thread identifiers
  - GetTheradId the thread's ID from its handle

Resume & Suspend Threads

```
DWORD ResumeThread (HANDLE hThread);

DWORD SuspendThread (HANDLE hThread);
```

- Every thread has a suspend count
  - A thread can execute only if this count is zero
- A thread can be created in the suspended state
- One thread can
  - Increment the suspend count of another thread (resume)
  - Decrement the suspend count of another thread (suspend)

Resume & Suspend Threads

- Return value
  - Both functions return previous suspend count
  - The value 0xFFFFFFFF, in case of failure
- Useful in preventing "race conditions"
  - Do not allow threads to start until initialization is complete
- Unsafe for general synchronization

Thread's Priority

```
DWORD SetThreadPriority (
  HANDLE hThread, DWORD dwPriority);

DWORD GetThreadPriority (HANDLE hThread);
```

- Change or determine a thread's priority
  - For itself
  - For another process, security permitting
- Thread priorities are relative to the process base priority (the priority class)
  - See **SetPriorityClass** and **GetPriorityClass** for further details

Thread's Priority

- Use constant values as **dwPriority**
  - `THREAD_PRIORITY_LOWEST`, `THREAD_PRIORITY_BELOW_NORMAL`, etc.

- Modify the priority with cautions
  - Use high thread priorities with caution
  - Avoid real time priorities for user processes
    - User threads may preempt executive threads
  - Assure fairness
    - All threads should run eventually
    - Real time priorities may prevent fairness
      - "Priority inversion"
      - "Thread starvation"