Synchronization

Synchronization (Part B)

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Objectives

- Windows advanced synchronization techniques include
  - Events
  - Semaphore throttles
  - Thread pools

Used with "lots" of threads, when thread context switching is very time consuming

Chapters 8 and 9 of J. M. Hart

Events

- Events are kernel synchronization objects
- They
  - Are useful in sending a signal to a thread indicating that a particular event has occurred
    - Remind, UNIX signals and condition variables
  - Have an additional capability with respect to previous synch strategies as they can release multiple threads from a wait **simultaneously** when a single event is signaled
    - Remind, UNIX condition variable broadcast

Events

- Events objects must be
  - Created using **CreateEvent**
    - When they are created their type, either even or pulse, is selected
  - Signalled using **PulseEvent** or **SetEvent**
  - Reset with **Automatic** or **Manual Reset**
  - Waited-for using WFSO or WFMO
- Overall, there are four combinations with very different behavior

Events

1 T is released

From 1 to n Ts are released

Ts are released in t=[0,∞]

	Auto-Reset	Manual-Reset
SetEvent	Exactly one thread is released. If none are currently waiting on the event, the next thread to wait will be released. 	All currently waiting threads are released. The event remains signaled until it is explicitly reset by some thread. 
PulseEvent	Exactly one thread is released, but only if a thread is currently waiting on the event. 	All currently waiting threads are released. Then the event is automatically reset. 

Ts are released in t=[0]

CreateEvent

```
HANDLE CreateEvent (
  LPSECURITY_ATTRIBUTES lpsa,
  BOOL fManualReset,
  BOOL fInitialState,
  LPTCSTR eventName
);
```

- Create a new event object
- Parameters
  - lpsa
    - Often NULL
  - fManualReset
    - TRUE, for **manual-reset** event
    - FALSE, otherwise (auto-reset event)

CreateEvent

- fInitialState
  - The event is initially set to signaled if it is TRUE
  - Often FALSE
- eventName
  - Name of the even (named event)
  - It is possible to use **OpenEvent** to open a named event, possibly created by another process

```
HANDLE CreateEvent (
  LPSECURITY_ATTRIBUTES lpsa,
  BOOL fManualReset,
  BOOL fInitialState,
  LPTCSTR eventName
);
```

SetEvent

```
BOOL SetEvent (HANDLE hEvent);
```

- With SetEvent
  - If the event is manual-reset, the event remains signaled until some thread calls **ResetEvent** for that event
    - A **ResetEvent** put the event explicitly to the non-signaled state
    - Any number of waiting threads, or threads that subsequently begin wait operations for the specified event object by calling one of the wait function, can be released while the object's state is signaled

SetEvent

- If the event is automatic-reset, the event object remains signaled until a single waiting thread is released
  - When a thread is released the system **automatically** sets the state to non-signaled
  - If no threads are waiting, the event object's state remains signaled until a thread is realeased

```
BOOL SetEvent (HANDLE hEvent);
```

ResetEvent

```
BOOL ResetEvent (HANDLE hEvent);
```

- The ResetEvent function is used primarily for manual-reset event objects
  - Manual-reset event must be set explicitly to the non-signaled state
  - Auto-reset event objects do not need ResetEvent
    - They are **automatically** changed from signaled to non-signaled after a single waiting thread is released
- Return value
  - Non-zero, if the function succeeds
  - Zero, if the function fails

PulseEvent

```
BOOL PulseEvent (HANDLE hEvent);
```

- PulseEvent allows you to release all threads currently waiting on a manual-reset event
  - The event is then automatically reset to the non-signaled state after releasing the appropriate number of waiting threads
  - If no threads are waiting, or if no thread can be released immediately, **PulseEvent** simply sets the event object's state to non-signaled and returns

PulseEvent

- Return value
  - Non-zero, if the function succeeds
  - Zero, if the function fails

```
BOOL PulseEvent (HANDLE hEvent);
```

Wait for Events

- Events are waited for using the general functions WFSO and WFMO
  - Be careful when using **WaitForMultipleObjects** to wait for **all** events to become signaled
    - A waiting thread will be released only when **all** events are simultaneously in the signaled state
    - Unfortunately, some signaled events might be **reset** before the thread is released

Example

- Use an event object to prevent several threads from reading from the same shared memory buffer the same value
- More specifically
  - A **boss** writing thread
    - Writes a new data field into a buffer
    - Sets the event object to the signaled state when it has finished writing
  - Several **worker** reader threads
    - Wait for the data to be ready
    - Only one of them, reads the data field from the buffer

Example

Main program

```
HANDLE writeEvent;
...
writeEvent = CreateEvent (
  NULL,                 // default security attributes
  TRUE,                 // manual-reset event
  FALSE,                // initial state is nonsignaled
  TEXT("WriteEvent") // object name
);
if (writeEvent == NULL) {
  printf("CreateEvent failed (%d)\n", GetLastError());
  return;
}

... Create 1 Writer thread and n Reader threads ...
... Wait for all Threads ...

CloseHandle (writeEvent);
```

WriteToBuffer

ReadFromBuffer

Example

1 Writer Thread

The event objects is used to prevent **several** threads from reading from the shared memory buffer

```
void WriteToBuffer (VOID) {
  ... Write to the shared buffer
  if (!SetEvent(writeEvent)) {
    printf("SetEvent failed (%d)\n", GetLastError());
    return;
  }
}
```

Example

N Reader Threads

```
void ReadFromToBuffer (VOID) {
  DWORD waitResult;

  waitResult = WaitForSingleObject (writeEvent, INFINITE);
  switch (waitResult) {
    // Event object was signaled
    case WAIT_OBJECT_0:
      ... Read from the shared buffer ...
   // An error occurred
   default:
     printf("Wait error (%d)\n", GetLastError());
     return;
 }
 ...
 return;
}
```

Warnings

- There are numerous subtle problems using events
  - Setting an event that is already set has no effect
    - There is no memory
    - Multiple SetEvent may be lost
  - Resetting an event that is already reset has no effect

Warnings

- PulseEvent is **unreliable** and **should not be used**
  - The event may be lost
  - A thread waiting on a synchronization object can be
    - Momentarily removed from the wait state by a kernel-mode APC (Asynchrnous Procedure Call)
      - For example a completion notification
    - Then returned to the wait state after the APC is complete
  - If the call to **PulseEvent** occurs during the time when the thread has been removed from the wait state, the thread will not be released because **PulseEvent** releases only those threads that are waiting at the moment it is called
  - It exists mainly for backward compatibility

Synch Primitives Comparison

	CS	Mutex	Semaphore	Event
Named, Securable Synchronization Object	No	Yes	Yes	Yes
Accessible from Multiple Ps	No	Yes	Yes	Yes
Synchronization	ECS	Wait	Wait	Wait
Release	LCS	Release or owner terminates	Any thread can release	Set or Pulse
Ownership	One T at a time. Recursive	One T at a time. Recursive	N/A	N/A
Effect of Release	One waiting T can enter	One waiting T can gain ownership after last release	Multiple Ts can proceed, depending on release count	One or several waiting Ts will proceed after a Set or Pulse

Semaphore Throttles

- Scenario
  - **N** worker Ts contend for a shared resource
    - They may use a CS, a mutex or a semaphore
  - Performance degradation is severe when **N** increases and contention is **high**
- Target
  - Improve performance
  - Retain the simplicity of the original approach
- "Semaphore throttles"
  - Use a semaphore to **fix** the **maximum** amount of running Ts

Semaphore Throttles

- The boss T
  - Creates a semaphore
  - Sets the maximum value to a "reasonable number"
    - Example: 4, 8, 16
    - Its value depends on the number of core or processors
    - It is a tunable value
- Worker Ts must get a semaphore unit before working
  - Wait on the semaphore throttles
  - Then, wait on the CS or mutex or semaphore, etc. (to access critical section areas)

Semaphore Throttles

Worker loop

WFSO = Wait For Single Object

```
while (TRUE) {
  WFSO (hSem, Infinite);
  ...
  WFSO (hMutex, Infinite);
  ...
  ReleaseMutex (hMutex);
  ...
  ReleaseSemaphore (hSem, 1, NULL);
}
```

Semaphore Throttles

Standard Critical Section

If the max count for hSem is 1 (at most 1 worker T), hMutex is useless

Semaphore Throttles

- Variations
  - Some workers may acquire multiple units
    - The idea is that workers than use more resources wait more on the throttles
  - Caution
    - Pay attention to deadlock risks
- The boss T may tune dynamically the worker Ts behavior
  - Decreases or increases the number of active workers
    - By waiting or releasing semaphore units
    - Anyhow, the maximum number of Ts allowed is set once and only once at initialization

Set it to be "large enough"

Thread Pools

Available from NT 6:
Windows 7, Vista, Server 2008, etc.

- Thread pool concepts
  - The user
    - Initializes a "thread pool" queue
    - Creates "work objects" (or "tasks") rather than threads
      - Each task is a callback function (equivalent to a thread function)
      - Each callback function has a unique parameter
    - Inserts tasks into the queue

Thread Pools

- Windows automatically
  - Manages a small number of worker threads
    - Windows may automatically adjust the number of workers
  - Assigns a task to a worker thread that will work on the task
  - When it completes it may be assigned to a new task
- Worker threads
  - Can run concurrently on separate processors
  - Invokes all callbacks without stopping
    - There is no context switching
  - Callback functions **should not use** ExitThread or _endthreadex
    - This call would terminate the worker thread

Thread Pools

- Phases
  - Define a new callback environment of type TP_CALLBACK_ENVIRON
  - InitializeThreadpoolEnvironment
    - Initialization call to the environment
  - CreateThreadPoolWork
    - Creates a work objects
  - SubmitThreadPoolWork
    - Submit work objects to the thread pool

"Equivalent" to **CreateThread**
but we do not explicitly run threads, we just submit task to the thread pool (queue)

Thread Pools

- WaitForThreadpoolWorkCallbacks
  - Block the boss (or calling) thread until all calls to the work object complete
- CloseThreadpoolWork
  - Replace CloseHandle

"Equivalent" to **WFSO**

Thread Pools

```
void InitiazeTreadpooldEnvironment (
  PTP_CALLBACK_ENVIRON pcbe
);
```

To next function

- This function initializes a callback environment
  - The structure TP_CALLBACK_ENVIRON defines the callback environment to initialize
  - This function must be called before using any the API functions reported in the following pages
- There is no return value

Thread Pools

To next function

pwk

```
PTP_WORK CreateThreadpoolWork (
  PTP_WORK_CALLBACK pfnwk,
  PVOID pv,
  PTP_CALLBACK_ENVIRON pcbe
);
```

From previous function

- This system call is "equivalent" to CreateThread
- It creates a new work object within the callback environment pcbe
  - It must be called once for every task (i.e., thread) we want to solve

Thread Pools

- Parameters
  - **pfnwk** is the callback function (the "thread" function)
  - **pv** is the parameter of this function
  - **pcbe** is the enviroment
- Return value
  - The task **pwk**, in case of success
    - This work object is not ready to run yet as it must be submitted to the thread pool
  - NULL, in case of failure

```
PTP_WORK CreateThreadpoolWork (
  PTP_WORK_CALLBACK pfnwk,
  PVOID pv,
  PTP_CALLBACK_ENVIRON pcbe
);
```

Thread Pools

```
VOID SubmitThreadpoolWork (
  PTP_WORK pwk
);
```

From previous function

- This system call submits a new work object pwk to the thread pool
  - **pwk** is the work object
    - It was returned by function CreateThreadPoolWork
  - The callback function associated with **pwk** will be called once for every SubmitThreadpollWork call
    - This call should never fail if **pwk** is valid as all resources have been previously allocated
    - The kernel decides which thread to use

Thread Pools

```
VOID WaitForThreadpoolWorkCallbacks (
  PTP_WORK pwk,
  BOOL fCancelPendingCallbacks
);
```

- This function allows the boss thread to wait for all submitted work objects to be completed
  - It is usually called by the boss thread once for each submitted work objects
  - It also allows the boss thread to cancel still pending work objects
- This function does not have a timeout
  - It is not possible to try-wait on the pool

Thread Pools

- Parameters
  - **pwk** is the work object
    - It was returned by function CreateThreadPoolWork
  - It **fCancelPendingCallbacks** is true it is possible to cancel a work object
    - Only work objects not yet started can be cancelled, all others run to completion

```
VOID WaitForThreadpoolWorkCallbacks (
  PTP_WORK pwk,
  BOOL fCancelPendingCallbacks
);
```

Thread Pools

```
VOID CloseThreadpoolWork (
  PTP_WORK pwk
);
```

- This function releases the specifies work object
  - **pwk** is the work object
    - It was returned by function CreateThreadPoolWork
  - The work object is
    - Freed immediately, if there are no outstanding callbacks
    - Freed asynchronously, after the outstanding callbacks complete
- There is no return value

Thread function

- With thread pools the thread function assumes the name of "callback " function
  - Callback functios, as standard thread functions, must have a pre-defined prototype
- Windows will call this function once for every SubmitThreadpoolWork invocation

Thread function

```
VOID CALLBACK WorkCallback (
  PTP_CALLBACK_INSTANCE Instance,
  PVOID Context,
  PTP_WORK pwk
);
```

- Parameter
  - **Instance** identifies a specific callback instance
    - It may be passed on to other functions, such as SetEventWhenCallbackReturns, …
    - It provides Windows with specific information about this instance that may help in the scheduling
      - The instance may execute for a short or a long time
      - Often, callback functions are expected to execute quickly

Thread function

- **Context** it the parameter of the function
  - It was specified during the call to CreateThreadpoolWork
- **pwk** is the work object
  - It was returned by function CreateThreadPoolWork

```
VOID CALLBACK WorkCallback (
  PTP_CALLBACK_INSTANCE Instance,
  PVOID Context,
  PTP_WORK pwk
);
```

Example

- Solve a concurrent problem using a thread pool to run threads instead of running them explicitly
  - Initialize the thread pool
  - Submit work objects to the thread pool
  - Wait for all tasks to be completed
  - Clean-up the pool

Define a new callback environment

Initialize the environment

```
TP_CALLBACK_ENVIRON cbe;
...
InitializeThreadpoolEnvironment (&cbe);
```

Example

Main (boss thread)

```
for (i=0; i<nThread; i++) {
  ...
  pThreadArg->objectNumber = i;
  ...
  pWorkObjects[i] = CreateThreadpoolWork (
    Worker, pThreadArg, &cbe);
  if (pWorkObjects[i] == NULL)
    ... error ...
  SubmitThreadpoolWork (pWorkObjects[i);
}

for (i=0; i<nThread; i++) {
  WaitForThreadpoolWorkCallbacks (
    pWorkObjects[i], FALSE);
  CloseThreadpoolWork (pWorkObjects[i]);
}
```

Create work objects

Submit work to the pool

Thread pool work objects are running ...

Wait for thread and destroy work objects

Example

Thread callback function

```
VOID CALLBACK Worker (
  PTP_CALLBACK_INSTANCE Instance,
  PVOID Context,
  PTP_WORK Work)
{
  THARG *threadArgs;

  threadArgs = (THARG *)Context;
  _tprintf (_T("Worker Thread Number: %d.\n"),
    threadArgs->objectNumber);

  ...

  return;
}
```