Event

Un'altro oggetto per la sincronizzazione dei trhead è l'oggetto event. Gli oggetti Event sono utilizzati quando si deve notificare ad un thread in attesa che è avvenuto evento.

Gli event sono utilizzati per I/O overlapped su file o nell'utilizzo di device di comunicazione come le porte seriali o anche per fare in modo che dei thread partono contemporaneamente.

Le funzioni per creare e gestire un oggetto event sono:

- CreateEvent
- OpenEvent
- SetEvent
- ResetEvent
- PulseEvent

CreateEvent

La funzione CreateEvent crea un nuovo oggetto Event o ne apre uno esistente , e la sua sintassi è la seguente:

```
HANDLE CreateEvent(
LPSECURITY_ATTRIBUTES lpMutexAttributes
BOOL bManualReset, // flag per il reset manual
BOOL bInitialState, // stato initial
LPCTSTR lpName // Nome
);
```

parametri sono:

lpMutexAttributes

Puntatore ad una struttura di tipo SECURITY_ATTRIBUTES che determina se l'handle restituito può essere passato ad un processo figlio,se è NULL, l'handle non può essere passato ad un processo figlio.

bManualReset

Specifica se si sta creando un event manual-reset o un auto-reset. Se TRUE per settare lo stato dell'oggetto a non segnalato si deve usare la funzione ResetEvent

Se FALSE, il sistema setta automaticamente lo stato a non segnalato dopo che una funzione di attesa di un thread è stata rilasciata.

bInitialState

Specifica il valore iniziale del event.

Se True lo stato iniziale è segnalato

Se False lo stato iniziale è non segnalato

lpName Il nome dell'event

OpenEvent

La funzione OpenEvent apre un named event esistente:

```
HANDLE OpenEvent(
DWORD dwDesiredAccess, // access flag
BOOL bInheritHandle, // inherit flag
LPCTSTRlpName // nome
);
```

dwDesiredAccess

Specifica l'accesso richiesto per l'oggetto Event, esso può essere la combinazione in OR dei seguenti valori:

EVENT_ALL_ACCES tutti i flag

EVENT_MODIFY_STATE abilita la possibilità di modificare il valore dell'event attraverso le funzioni SetEvent e ResetEvent.

SYNCHRONIZE abilita l'uso del'event in una delle wait functions

bInheritHandle

Specifica se l'handle restituito può essere passato ad un processo figlio,se è NULL, l'handle non può essere passato ad un processo figlio.

lpName Il nome del event

SetEvent

La funzione SetEvent setta lo stato dell' event a segnalato.

```
BOOL SetEvent(
HANDLE hEvent, // handle del event
);
```

ResetEvent

La funzione ResetEvent setta lo stato dell' event a non segnalato.

```
BOOL ResetEvent(
HANDLE hEvent, // handle del event
);
```

PulseEvent

La funzione PulseEvent provvede in una singola operazione a settare a segnalato lo stato dell'event e a risettarlo immediatamente dopo a non segnalato dopo il rilascio di tutti gli event da parte delle funzione di waiting, una sorta di reset totale.

```
BOOL PulseEvent(
HANDLE hEvent, // handle del event
);
```

Nel seguente esempio due thread aspettano un evento da programma principale per attivarsi

```
#include <stdio.h>
#include <windows.h>
////////////////////////////////////////
DWORD ThreadFunc1 (LPDWORD lpdwParam);
DWORD ThreadFunc2 (LPDWORD lpdwParam);
////////////////////////////////////////
VOID main (VOID)
{HANDLE hEvent, hThreads[2];
  DWORD dwThreadId1, dwThreadId2;
  SECURITY_ATTRIBUTES shouldInherit;
  shouldInherit.nLength = sizeof (SECURITY_ATTRIBUTES);
  shouldInherit.lpSecurityDescriptor = NULL;
  shouldInherit.bInheritHandle = TRUE;
  printf ("Main thread, Creating event object\n");
  hEvent=CreateEvent(&shouldInherit,TRUE,FALSE,"TestEvent");
  if (hEvent==NULL)
   printf ("Main thread: CreateEvent error: %d\n",
GetLastError());
 else
  {if (GetLastError()==ERROR_ALREADY_EXISTS)
```

```
        printf ("Main thread: CreateEvent opened existing
event\n");
    else
        printf ("Main thread: CreateEvent created new event\n");
   }
 printf ("Main thread: Creating secondary thread 0\n");
 hThreads[0]=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)
ThreadFunc1, NULL, 0, &dwThreadId1);
 if (hThreads[0]==NULL)
   fprintf(stderr,"Main thread: CreateThread1 error\n");
 printf ("Main thread: Creating secondary thread 1\n");
 hThreads[1]=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)
ThreadFunc2, NULL, 0, &dwThreadId2);
  if (hThreads[1]==NULL)
    fprintf(stderr,"Main thread: CreateThread2 error\n");
/* main program, sleeping for 3 seconds */
  printf ("Main thread, sleeping for 3 seconds\n");
  Sleep (3000L);
/* Main program, calling PulseEvent */
  printf ("Main thread, calling PulseEvent\n");
  PulseEvent (hEvent);
  printf ("Main thread, waiting for secondary threads to
finish\n");
```

```
  WaitForMultipleObjects (2,hThreads, TRUE, INFINITE);
  Sleep (1000L);
  printf ("End of event test program\n");
}
////////////////////////////////////////
DWORD ThreadFunc2 (LPDWORD lpdwParam)
{HANDLE hEvent;
 DWORD dwWaitResult;
 printf ("ThreadFunc2 start\n");
 hEvent=OpenEvent(EVENT_MODIFY_STATE|SYNCHRONIZE,
FALSE,"TestEvent");
 if (hEvent==NULL)
    printf ("ThreadFunc2: OpenEvent error:%d\n",GetLastError());
 else
    printf("ThreadFunc2: Successfully opened event object\n");
 dwWaitResult=WaitForSingleObject(hEvent,INFINITE);
 if (dwWaitResult==WAIT_OBJECT_0)
   printf("ThreadFunc2: event signalled, terminating\n");
 else
   printf("ThreadFunc2:Error WaitForSingleObject,
%d\n",GetLastError());
 return 0;
}
```

```
//////////////////////////////////////
DWORD ThreadFunc1 (LPDWORD lpdwParam)
{HANDLE hEvent;
 DWORD dwWaitResult;
 printf ("ThreadFunc1 start\n");
 hEvent=OpenEvent(EVENT_MODIFY_STATE|SYNCHRONIZE,
FALSE,"TestEvent");
 if (hEvent==NULL)
    printf ("ThreadFunc1: OpenEvent error:%d\n",GetLastError());
 else
    printf("ThreadFunc1: Successfully opened event object\n");
 dwWaitResult=WaitForSingleObject(hEvent,INFINITE);
 if (dwWaitResult==WAIT_OBJECT_0)
   printf("ThreadFunc1: event signalled, terminating\n");
 else
   printf("ThreadFunc1:Error WaitForSingleObject,
%d\n",GetLastError());
 return 0;
}
```

Come esercitazione vi propongo il barbiere che dorme.

In un negozio di barbiere abbiamo

- un barbiere
- una poltrona per il cliente servito
- N sedie per clienti in attesa

Se non vi sono clienti nel negozio il barbiere dorme sulla poltrona, il primo cliente che entra nel negozio vuoto sveglia il barbiere.

I clienti che entrano trovando la poltrona occupata si mettono in attesa su una sedia.

Il cliente che non trova una sedia libera va via.

Cosa serve

- Un thread barbiere
- I Thread Clienti
- Un mutex per la poltrona `hPoltrona`
- Un event per avvertire il cliente che il barbiere ha finito di tagliare i capelli `hFinetaglio`
- Un mutex per sapere se il barbiere dorme o no `BarbiereDorme`
- Un event per svegliare il barbiere `hSveglia`
- Un array di N mutex per le sedie `hchair[N]`
- Una variabile che indica se il barbiere ha servito tutti i clienti presenti `tuttiserviti` se =0 ha ancora clienti se = 1 non ci sono più clienti posso andare a dormire di nuovo

Il thread Barbiere quindi agisce così

1. Finché giornata finita
   a. Inizialmente dorme quindi si pone la variabile `tuttiserviti=0` e acquisisce il mutex `BarbiereDorme` e il mutex `hPoltrona` usando la funzione `WaitForSingleObject`
   b. Aspetta che qualcuno lo svegli quindi cerca di acquisire l'event `hSveglia` sempre con una `WaitForSingleObject`
   c. Appena acquisito vuol dire che un cliente a mandato l'event con un `PulseEvent(hSveglia);` quindi ci sono clienti da servire, si pone `tuttiserviti=1` si rilasciano i mutex `hPoltrona` e `BarbiereDorme`
   d. Finché ci sono clienti
      I. Taglia i capelli `Sleep`
      II. Avverte che a finito `PulseEvent(hFinetaglio)`

III. Attende un attimo e poi vede se la poltrona è di nuovo occupata

```
Sleep(10);
ris=WaitForSingleObject(hPoltrona, 0);
if(ris!=WAIT_TIMEOUT) tuttiserviti=1;
else ReleaseMutex(hPoltrona);
```

IV. Fine ciclo Finché ci sono clienti

e. Fine ciclo Finché giornata finita

I Clienti agiscono così

1. Cercano di acquisire una delle sedie entro un certo tempo con l'istruzione `ris=WaitForMultipleObjects(NCHAIR,hchair, FALSE,1000L);`
   a. Se l'acquisisce `if(ris!=WAIT_TIMEOUT)`
      I. Vede se il barbiere dorme cercando di acquisire il mutex `BarbiereDorme, ris1=WaitForSingleObject(BarbiereDorme,0);`
      II. Se dorme lo sveglia `if(ris1==WAIT_TIMEOUT)PulseEvent (hSveglia); else ReleaseMutex(BarbiereDorme);`
      III. Attende un attimo in modo che il barbiere rilasci la poltrona e vi si siede `WaitForSingleObject(hPoltrona,..`
      IV. Rilascia la sedia
      V. Aspetta che il barbiere avverte che ha finito il taglio con un `WaitForSingleObject` su `hFinetaglio` (il Barbiere lo fa con un `PulseEvent(hSveglia)`)
      VI. Rilascia la poltrona e va via

La differenza tra il codice per Unix/Linux e il codice per Windows sta principalmente nel fatto che un su Unix/Linux se dichiarati fast e quindi di default possono essere rilasciati da un altro thread.

Su Window “Un thread rilascia un mutex chiamando il relativo metodo ReleaseMutex. I mutex presentano affinità di thread, ossia possono essere rilasciati solo dal thread che ne è proprietario. Se un thread rilascia un mutex di cui non è proprietario, nel thread viene generata un'eccezione ApplicationException.”

Citazione dalla pagina

http://msdn.microsoft.com/it-it/library/hw29w7t1.aspx

viceversa

“Similarly, only the thread that owns a mutex can successfully call the **ReleaseMutex** function, though any thread can use **ReleaseSemaphore** to increase the count of a semaphore object.”

Citazione dalla pagina

http://msdn.microsoft.com/It-it/library/windows/desktop/ms685129(v=vs.85).aspx

Quindi se si usano i semafori in windows è possibile decrementarli da un altro trhead.

Nel nostro caso invece di utilizzare un event per dire al Thread che possiede il mutex di rilasciarlo , con i semafori basterà rilasciarlo.