Primitives pour la programmation concurrente sur Windows NT

Prof. Dr. J.-D. Decotignie

decotignie@ieee.org

Plan

- threads
- synchronisation
- entrées et sorties superposées
- communication

csem

Threads

- création

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttr,
  DWORD dwStackSize,
  LPTHREAD_START_ROUTINE LpSrartAddress,
  LPVOID lpParameterm,
  DWORD DwCreationFlags,   /* permet de démarrer suspendu
  LPDWORD lpThreadId);
```

  - retourne une poignée si tout va bien
  - retourne FALSE sinon (GetLastError donne la cause)

csem

Objets du noyau

- il existe de nombreux objets définis dans le noyau
  - process, threads, files, events, semaphores, mutexes, pipes
- peuvent avoir plusieurs propriétaires
- le noyau garde la trace des utilisations (nb de poignées)
- la destruction s'opère quand ce compte passe à 0
- quand on crée un thread, le compte vaut 2 (créateur et thread lui même)
- quand un thread se termine, on décrémente
- pour détruire, il faut faire CloseHandle();

csem

Fin d'un thread

- sortie

```
VOID ExitThread(DWORD dwExitCode);
```

  - pas nécessaire
- sortie du thread primaire d'un process
  - pas par ExitThread car seul ce thread sera terminé
  - soit
    - par ExitProcess
    - en retournant de la fonction qui l'implante

csem

Etat d'un thread

- teste si le thread a terminé

```
BOOL GetExitCodeThread(
  HANDLE poignée,
  LPDWORD LpExitCode);
```

  - retourne TRUE si le thread a terminé
  - lpExitCode donne la valeur de fin ou STILL_ACTIVE
- attend la fin d'exécution d'un thread

```
DWORD WaitForSingleObject(
  HANDLE poignée,
  DWORD durée);     /* INFINITE est utilisable
```

  - retourne WAIT_OBJECT_0, WAIT_TIMEOUT ou WAIT_ABANDONED (Mutex)

csem

Etat des objets du noyaux

- signalé ou non
- signalé si
  - Thread, Process: terminé
  - Event: quand l'événement est activé
  - Mutex: pas pris par un thread
  - Semaphore: compte plus grand que zéro

csem

Priorités d'un thread

- modification et lecture

```
BOOL SetThreadPriority(
  HANDLE poignée,
  int Priority);

int GetThreadPriority(HANDLE poignée);
```

- on démarre souvent suspendu

```
BOOL ResumeThread (
  HANDLE poignée);

LONG SuspendThread ( // donne le nombre de suspensions
  HANDLE poignée);
```

csem

Attente multiple

- il existe parfois plusieurs conditions d'attente

```
DWORD WaitForMultipleObjects(
  DWORD nCount,
  CONST HANDLE *lpHandles,
  BOOL WaitAll,
  DWORD durée);     /* INFINITE est utilisable
```

  - nCount est au max. MAXIMUM_WAIT_OBJECTS (64)
  - on peut attendre un seul ou tous
  - return_value – WAIT_OBJECT_0 (ou WAIT_ABANDONED_0) indique le déclencheur
  - WAIT_FAILED si erreur (GetLastError donne les détails)

csem

Attente multiple (2)

- cas d'un thread qui attend un message

```
DWORD MsgWaitForMultipleObjects(
  DWORD nCount,
  CONST HANDLE *lpHandles,
  BOOL WaitAll,     // un ou tous
  DWORD durée,      // INFINITE est utilisable
  DWORD dwWakeMask);
```

  - dwWaitMask indique le type de message attendu (hot key, mouse, key, mouse button, timer, post message, send message, ...)

csem

Synchronisation

- sections critiques

```
VOID Initialize/Delete/Enter/LeaveCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
```

  - permettent la création, la destruction d'une section critique
  - permettent d'entrer et sortir d'une section critique
  - n'est pas un objet du noyau mais une variable
    - n'est pas détruit si le thread est détruit
    - attente multiple impossible
    - seulement utilisable pour des threads dans le même process

csem

Synchronisation (2)

- mutex

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES LpMutexAttributes,
  BOOL InitialOwner,
  LPCTSTR lpName);  /* nom de désignation
```

  - OpenMutex pour obtenir une poignée sur un existant
  - (Msg)WaitForSingle/MultipleObject(s) pour entrer dans la section critique
  - ReleaseMutex pour en sortir
  - CloseHandle();
  - passe automatique en non signalé en cas d'arrêt du thread sans faire ReleaseMutex

csem

Synchronisation (3)

- Sémaphores

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES LpMutexAttributes,
  LONG InitialCount,
  LONG MaxCount,
  LPCTSTR lpName);  /* nom de désignation
```

  - OpenSemaphore pour obtenir une poignée sur un existant
  - (Msg)WaitForSingle/MultipleObject(s) bloque si à 0, décrémente sinon
  - ReleaseSemaphore incrémente compte si <MacCount
  - CloseHandle();

Synchronisation (4)

- event

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES LpMutexAttributes,
  BOOL ManualReset,
  BOOL InitialState,
  LPCTSTR lpName);  /* nom de désignation
```

  - manualReset indique si après avoir réveillé un thread, l'événement se remet à zéro (non signalé)
  - open, closeHandle comme avant
  - Set(Reset)Event pour passer à l'état signalé (non signalé)
  - PulseEvent (avant de se remettre en non signalé)
    - ManualReset: réveille tous les threads en attente
    - Auto reset: réveille un seul thread en attente

Synchronisation (5)

- interlocked variables

```
LONG InterlockedIncrement(
  LPLONG lpTarget); /* signe rc = signe variable
LONG InterlockedDecrement(
  LPLONG lpTarget); /* signe rc = signe variable
LONG InterlockedExchange(
  LPLONG lpTarget
  LONG Valeur);     /* retourne ancienne valeur
```


Fin d'un thread (encore)

- terminer un autre thread

```
BOOL TerminateThread(
  HANDLE hThread, DWORD dwExitCode);
```

  - dangereux
- utilise un fanion ou un événement

Entrées et sorties superposées

- permet de faire une demande de lecture ou d'écriture sans bloquer en attendant la réponse
- s'applique sur
  - fichiers, ports, named pipes, etc.

```
HANDLE CreateFile(
  LPCTSTR lpFileName,      // pointeur sur le nom
  DWORD dwDesiredAccess,   // mode d'accès (rd,wr)
  DWORD dwShareMode,       // partage
  LPSECURITY_ATTRIBUTES lpSecAttr,
  DWORD dwCreationDistribution,
  DWORD dwFlagsAndAttr,    // ici: FILE_FLAG_OVERLAPPED
  HANDLE hTemplateFile);   // poignée vers fichier
```


Entrées et sorties superposées (2)

- lecture (retourne false en principe)

```
BOOL ReadFile(
  HANDLE hfile,
  LPVOID lpBuffer,
  DWORD NbByteToRead,      // nombre voulus
  LPDWORD lpNbByteRead,    // nombre lus
  LPOVERLAPPED lpOverlapped);     // structure
```

- écriture (retourne false en principe)

```
BOOL WriteFile(
  HANDLE hfile,
  LPVOID lpBuffer,
  DWORD NbByteToWrite,     // nombre voulus
  LPDWORD lpNbByteWritten, // nombre écrits
  LPOVERLAPPED lpOverlapped);     // structure
```


Entrées et sorties superposées (3)

- attente WaitForObject avec poignée
- résultat (comme si appel read/write normal)

```
BOOL GetOverlappedResult(
  HANDLE hfile,
  LPOVERLAPPED lpOverlapped,
  LPDWORD lpNbByteTransferred,   // nombre lus ou écrits
  BOOL bWait
);
```

- plusieurs requêtes peuvent être faites en parallèle

```
typedef struct _OVERLAPPED {
  DWORD Internal; DWORD InternalHigh;
  DWORD Offset;
  DWORD OffsetHigh;
  HANDLE hEevent; } OVERLAPPED;
```


Attente multiple

- utiliser la variable event de OVERLAPPED
- utiliser WriteFileEx ou ReadFileEx
  - on donne alors une routine de callback (APC)

```
VOID WINAPI FileIOCompletionRoutine(
  DWORD dwErrorCode,
  DWORD dwNbBytesTransferred,
  LPOVERLAPPED lpOverlapped);
```

  - quand l'opération est terminée, cette routine est appelée
    - seulement si le thread est dans l'état "alertable" (SleepEx, yyyWaitForxxxObjectEx, SignalObjectAndWait)
- IO Completion ports

Communication

- pipes
- message queues
- mailslots
- mémoire partagée
- fichiers partagés

Named Pipes

- uni ou bidirectionnel
- création

```
HANDLE CreateNamedPipe(
  LPCTSTR lpPipeName,      // pointeur sur nom
  DWORD fdwOpenMode,       // lecture, écriture
  DWORD fdwPipeMode,       // bloquant, car/msg
  DWORD nMaxInstances,     // nombre de clients
  DWORD cbOutBuf,          // taille des tampons
  DWORD cbInBuf,
  DWORD dwTimeOut,         // timeout en accès
  LPSECURITY_ATTRIBUTES lpsa);
```

- attente de connexion

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


Named Pipe - client

- connexion par CreateFile
- vérification de l'existence par WaitNamedPipe(name,timeout)
- accès par ReadFile ou WriteFile
  - supporte entrées et sorties superposées
- PeekNamePipe pour voir s'il y a quelque chose

Message queues

- seulement vers un thread qui gère une fenêtre
- seulement pour les messages définis dans le système
  - on ne passe que son numéro
- PostMessage: envoi non bloquant
- SendMessage: envoi bloquant
  - idem + timeout
  - idem + call back
- GetMessage: réception bloquante

Waitable Timers

- synchonisation timer
  - associé à une fonction de call back
- manual reset timer
  - comme un autre événement

```
HANDLE CreateWaitableTimer(
  LPSECURITY_ATTRIBUTES lpta,
  BOOL bManualReset,
  LPCTSTR lpTimerName);
```

- initialement inactif

Waitable Timers (2)

- mise en route

```
BOOL SetWaitableTimer(
  HANDLE htimer,
  const LARGE_INTEGER *pDueTime,
  long lperiod,     // 0 si un seul coup
  PTIMERAPCROUTINE pfnCompletionRoutine,
  LPVOID lpArgToCompletionRoutine,
  BOOL fresume);
```

- arrêt CancelWaitableTimer

Mémoire

- on peut bloquer des pages en mémoire physique
  - éviter la pagination (page sur disque)
  - indispensable pour le temps réel
  - deux primitives
    - VirtualLock
    - VirtualUnlock

Références

- J. Hart, "Win32 System Programming", Addison Wesley, 2001, 0-201-70310-6
- J. Beveridge, R. Wiener, "Multithreaded Application in Win32", Addison Wesley, 1997, 0-201-44234-5
- bibliothèque Visual C++