Synchronisation
Parti I
C++

Rachid Kadouche

Définition

La synchronisation de processus est un mécanisme qui vise à bloquer l'exécution de certains processus à des points précis dans leur exécution, de manière que tous les processus se rejoignent à des étapes relais données, tel que prévu par le programmeur.

Exemples 1

Le *scope* d'une donnée

```
DWORD WINAPI Numero2(void* pParam)
{
  bool* pb;

  while (reinterpret_cast<bool*>(pParam))
  {pb = reinterpret_cast<bool*>(pParam);

    cout << *pb;
   }
  return 0;
}


DWORD WINAPI Numero1(void* pParam)
{
   bool b = true; // par ex.

   HANDLE hNo2;
   hNo2 = CreateThread(0, 0, Numero2, &b, 0, 0);
}
```

Que se passe-t'il si le thread «Numero1» ayant créé le thread «Numero2» se termine avant celui-ci?

Soit:

- Une «exception d'adressage» ou,
- Pointe sur «quelque chose»

Solutions:

1. Éviter l'utilisation d'une variable globale. (Risque de changement de cette variables)
2. Empêchant le thread «Numero1» de se terminer tant et aussi longtemps que le thread «Numero2» n'est est terminé son exécution. (Utiliser **WaitForSingleObject** et **WaitForMultipleObjects**)
3. L'allocation dynamique (avec **new**) de la donnée qui sera passée en paramètre au thread. (Il faut s'assurer de bien libérer cette mémoire)

Exemples 2

Conditions de course

Adapté des notes de cours de Patrice Roy

```
#include <windows.h>
class DivisionParZero {};
DWORD WINAPI Surprise (void *p)
// LE THREAD
{
   int *val = reinterpret_cast<int *>(p);
   while (*val > 0)
   {
      (*val)--;
      Sleep(1);  //  Sleep(1)  pour  s'assurer  que  les  autres  threads
travaillent
   }   // mais, en pratique, cela ne change rien au problème proposé ici
   return 0L;
}
int f (int &denominateur)   throw(DivisionParZero)
{
   const int NUM = 10;
   if (denominateur == 0)      // (a)
      throw DivisionParZero ();
   return NUM / denominateur; // (b)
}
int main ()
{
   int val = 10;
   HANDLE hSurprise = CreateThread (0, 0,
  Surprise,  &val, 0, 0);
   try
   {
      f (val);
   }
   catch (DivisionParZero) { }
   WaitForSingleObject (hSurprise, INFINITE);
   CloseHandle (hSurprise);
   return 0;
}
```

Les cas d'exécutions :

1. Lorsque le test (opération (a)) sera réalisé, la valeur de **dénominateur** soit différente de zéro, et qu'il en soit de même lors de la division (opération (b)). Dans ce cas, le programme se complétera sans erreur d'exécution.
2. Lorsque le test (opération (a)) sera réalisé, **dénominateur** soit égal à zéro. Il en serait alors de même pour la division (opération (b)), mais l'exception serait lancée. Pas d'erreur d'exécution
3. Il est possible qu'au moment du test (opération (a)), la variable ait une valeur différente de zéro, mais qu'elle ait le temps de diminuer à zéro entre cet instant et celui de la division (opération (b))

Solutions:

1. Éviter les situations où ce problème pourrait se produire ;-)
2. Coordonner sérieusement les accès en utilisant les mécanismes de synchronisation.

Exemples 3

Inter blocage (*Deadlock*)

Il se produit lorsque deux thread s'empêchent directement ou indirectement de fonctionner, par exemple:

- le thread A possède la ressource R1 et veut la ressource R2; alors que
- le thread B possède la ressource R2 et veut la ressource R1.

Les situations d'interblocage peuvent être complexes et se produire dès qu'il existe un **cycle d'obtention de ressources** dans un système. De tels problèmes peuvent faire «geler» une application. Ce sont en réalité des boucles d'attente infinies...

Solution:
Il faut repérer lors du design de l'application les situations où nous pourrions avoir de l'interblocage et s'assurer de l'éviter.

Mécanismes de synchronisation

Événements

Caractéristique:

Un **événement** est quelque chose qui peut être déclenché (ou lancé, un peu comme une exception) par un thread.

On utilisera un événement lorsqu'on voudra que un ou plusieurs threads (généralement plusieurs) restent en attente d'une condition particulière. Par exemple, on pourrait vouloir réaliser en parallèle plusieurs analyses financières sur une même série de données, dès que les données en question sont disponibles.

Comment utiliser les événements?

Les événements sont comme les threads identifiés par une variable de type HANDLE. Cette variable doit être accessible à tous les threads en ayant besoin.

- La **création** d'un événement se fait à l'aide de la fonction **CreateEvent**. Créer un événement *n'est pas la même chose* que le déclencher. (*** 1.)
- Un thread attendra une occurrence d'un événement à l'aide de la fonction **WaitForSingleObject**() , exactement comme nous l'avions vu pour la fin d'un thread . (*** 2.)
- **Pour provoquer un événement**, on utilise la fonction **SetEvent** avec le HANDLE qui représente l'événement à déclencher . (*** 3.)
- Une fois qu'un processus en a fini du HANDLE d'un événement, il doit le libérer avec **CloseHandle**, comme dans les cas du HANDLE d'un thread. (*** 4.)

Événements

```
#include <windows.h>
#include <iostream>
#include <string>
using namespace std;
const int MINUTES_MAX = 1;
const int SECONDES_MAX = 30;
class ParamAppl
{
public:
   ParamAppl()
   {
    Boucler1 = TRUE; Boucler2 = TRUE;
   }
public:
   BOOL Boucler1; BOOL Boucler2; HANDLE hEvent[2];
};
DWORD WINAPI Thread1( void* pParam )
{
ParamAppl* pParamAppl = reinterpret_cast<ParamAppl*>(pParam);
   while(pParamAppl->Boucler1 || pParamAppl->Boucler2 )
   {
    Sleep(1000);
    // *** 3. Déclencher les événements
    SetEvent(pParamAppl->hEvent[0]);
    SetEvent(pParamAppl->hEvent[1]);
   }
   return 0;
}

DWORD WINAPI Thread2( void* pParam )
{
int Minutes, Secondes;
ParamAppl* pParamAppl = reinterpret_cast<ParamAppl*>(pParam);
   Secondes = SECONDES_MAX;
   for(Minutes=MINUTES_MAX ; Minutes>=0 ; Minutes--)
   {
    for( ; Secondes>=0 ; Secondes--)
    {
       // *** 2. Attendre l'événement
       WaitForSingleObject(pParamAppl->hEvent[0], INFINITE);
       printf( "Duree restante :%0.2d:%0.2d\n\r", Minutes, Secondes
    }
    Secondes = 59;
   }
   pParamAppl->Boucler1 = FALSE; return 0;
}
```

```
DWORD WINAPI Thread3( void* pParam )
{
int Minutes, Secondes;
ParamAppl* pParamAppl = reinterpret_cast<ParamAppl*>(pParam);
   for(Minutes=0 ; Minutes<MINUTES_MAX ; Minutes++)
   {
    for(Secondes=0 ; Secondes<60 ; Secondes++)
    {
       // *** 2. Attendre l'événement
    WaitForSingleObject(pParamAppl->hEvent[1], INFINITE);
    printf("\t\t\tDuree ecoulee :%0.2d:%0.2d\n\r", Minutes, Secondes);
    }
   }
   for(Secondes=0 ; Secondes<=SECONDES_MAX ; Secondes++)
   {
    // *** 2. Attendre l'événement
    WaitForSingleObject(pParamAppl->hEvent[1], INFINITE);
    printf("\t\t\tDuree ecoulee :%0.2d:%0.2d\n\r", Minutes, Secondes);
   }
   pParamAppl->Boucler2 = FALSE; return 0;
}
int  main(int argc, char* argv[])
{
HANDLE ListeThreads[3]; ParamAppl param;

cout << "Synchronisation de deux threads avec un troisieme...\n\n\r";
   // *** 1. Créer les événements
   param.hEvent[0] = CreateEvent(NULL, FALSE, TRUE,
                                 L"Evenement pour Thread2"); // AutoReset
   param.hEvent[1] = CreateEvent(NULL, FALSE, TRUE,
                                 L"Evenement pour Thread3"); // AutoReset
   ListeThreads[0] = CreateThread(NULL, 0, Thread1, &param, 0, 0);
   ListeThreads[1] = CreateThread(NULL, 0, Thread2, &param, 0, 0);
   ListeThreads[2] = CreateThread(NULL, 0, Thread3, &param, 0, 0);
   WaitForMultipleObjects(3, ListeThreads, TRUE, INFINITE);
   // *** 4. Relacher les événements
   CloseHandle(param.hEvent[0]);CloseHandle(param.hEvent[1]);
   CloseHandle(ListeThreads[0]);CloseHandle(ListeThreads[1]);
   CloseHandle(ListeThreads[2]);
   cin.get();  return 0;
}
```

Sémaphores

Un **sémaphore** représente un droit d'accès à une ressource qu'on n'a qu'en nombre **limité**. Le sémaphore permet de **définir le nombre de processus ou de threads** pouvant accéder en même temps à une ressource.

Comment utiliser les sémaphores:

On utilise une variable de type **HANDLE** pour représenter le sémaphore.

- La création d'un sémaphore se fait à l'aide de la fonction **CreateSemaphore**. Créer un sémaphore n'y donne pas immédiatement accès. On spécifie à la création d'un sémaphore le nombre d'exemplaires disponibles pour la ressource représentée par ce sémaphore. (*** 1.)
- Un thread attendra pour obtenir un sémaphore à l'aide de la fonction **WaitForSingleObject**, exactement comme dans le cas où il attendrait la complétion d'un thread.(*** 2.)
- Le thread ayant obtenu le sémaphore doit le relâcher avec la fonction **ReleaseSemaphore**.(*** 3.)
- Une fois qu'un processus en a fini du HANDLE d'un sémaphore, il doit le libérer avec **CloseHandle**, comme dans les cas du HANDLE d'un thread.(*** 4.)

Sémaphores

Exemple sans synchronisation

```
#include <iostream>
#include <windows.h>
using namespace std;
DWORD WINAPI Thread( void* pParam )
{
char Chaine[50];
char i;
int T;
int* pVal;
  pVal = reinterpret_cast<int*>(pParam);
  sprintf_s(Chaine, "%s %d",
             "Je suis le thread numero", *pVal);
  i=0;
  while(Chaine[i] != '\0')
  {
       cout << Chaine[i];
       i++;
       T = rand()/100;
       Sleep(T);
  }
  return 0;
}
int main(int argc, char* argv[])
{
HANDLE hThread[3];
int Thread1Param = 1;int Thread2Param = 2; int Thread3Param = 3;
  cout << "Deux threads ecrivent a l'ecran  sans synchronisation..." << endl;
  hThread[0] = CreateThread(NULL, 0, Thread, &Thread1Param, 0, 0);
  hThread[1] = CreateThread(NULL, 0, Thread, &Thread2Param, 0, 0);
  hThread[2] = CreateThread(NULL, 0, Thread, &Thread3Param, 0, 0);
  WaitForMultipleObjects(3, hThread, TRUE, INFINITE);
  cin.get();
  return 0;
}
```

Sémaphores

```
#include <iostream>
#include <windows.h>
using namespace std;
HANDLE hSemaphore;

DWORD WINAPI Thread(void* pParam)
{
            char Chaine[50]; int i; int T; int* pVal;
            pVal = reinterpret_cast<int*>(pParam);
            sprintf_s(Chaine, "%s %d", "Je suis le thread numero", *pVal);
            // *** 2. Attente du semaphore
            WaitForSingleObject(hSemaphore, INFINITE);
            i = 0;
            while (Chaine[i] != '\0')
            {
                        cout << Chaine[i];        i++; T = rand() / 100; Sleep(T);
            }
            // *** 3. On n'a plus besoin du semaphore
            ReleaseSemaphore(hSemaphore, 1, NULL); return 0;
}
int main()
{
            HANDLE hThread[3];
            int Thread1Param = 1; int Thread2Param = 2; int Thread3Param = 3; LONG cMax = 2;
            cout << "Trois threads ecrivent a l'ecran avec semaphore..." << endl;
            // *** 1. Creation du semaphore
            hSemaphore = CreateSemaphore(NULL,   // Attributs de sécurité
                        cMax,   // Valeur de départ
                        cMax,   // Nombre maximum d'accès
                        NULL);  // Pas de nom pour le semaphore

            hThread[0] = CreateThread(NULL, 0, Thread, &Thread1Param, 0, 0);
            hThread[1] = CreateThread(NULL, 0, Thread, &Thread2Param, 0, 0);
            hThread[2] = CreateThread(NULL, 0, Thread, &Thread3Param, 0, 0);
            WaitForMultipleObjects(3, hThread, TRUE, INFINITE);
            // *** 4. Relâcher le handle
            CloseHandle(hSemaphore); CloseHandle(hThread[0]); CloseHandle(hThread[1]);
            CloseHandle(hThread[2]);
            cin.get();
            return 0;
}
```

Sémaphores

```
#include <iostream>
#include <windows.h>
using namespace std;
HANDLE hSemaphore;

DWORD WINAPI Thread( void* pParam )
{
char Chaine[50]; int i; int* pVal;
   pVal = reinterpret_cast<int*>(pParam);
   sprintf_s(Chaine, "%s %d", "Je suis le thread numero", *pVal);
   // *** 2. Attente du semaphore
  WaitForSingleObject(hSemaphore, INFINITE);
   i=0;
   while(Chaine[i] != '\0')
   {
    cout << Chaine[i];    i++; Sleep(500);
   }
   // *** 3. On n'a plus besoin du semaphore
   ReleaseSemaphore(hSemaphore, 1, NULL); return 0;
}
int main()
{
HANDLE hThread[3];
int Thread1Param = 1;int Thread2Param = 2; int Thread3Param = 3;LONG cMax = 2;
   cout << "Trois threads ecrivent a l'ecran avec semaphore..." << endl;
   // *** 1. Creation du semaphore
   hSemaphore = CreateSemaphore( NULL,   // Attributs de sécurité
                           1,   // Valeur de départ
                           cMax,   // Nombre maximum d'accès
                           NULL);  // Pas de nom pour le semaphore

   hThread[0] = CreateThread(NULL, 0, Thread, &Thread1Param, 0, 0);
   hThread[1] = CreateThread(NULL, 0, Thread, &Thread2Param, 0, 0);
   hThread[2] = CreateThread(NULL, 0, Thread, &Thread3Param, 0, 0);
   Sleep(5000);
   ReleaseSemaphore(hSemaphore, 1, NULL);
   WaitForMultipleObjects(3, hThread, TRUE, INFINITE);
   // *** 4. Relâcher le handle
   CloseHandle(hSemaphore);CloseHandle(hThread[0]);CloseHandle(hThread[1]);
   CloseHandle(hThread[2]);
   cin.get();
   return 0;
}
```

Mutex

Un **mutex** (pour ***Mut****ual* ***Ex****clusion*) représente un droit d'accès exclusif à une ressource dont on n'a **qu'un seul exemplaire**. Un mutex est un sémaphore de capacité 1. Le mutex est probablement le mécanisme le plus facile à utiliser pour synchroniser l'accès sur une donnée. Ceci explique qu'il s'agisse probablement de l'outil de synchronisation le plus fréquemment rencontré en pratique.

Comment utiliser les Mutex:

On utilise une variable de type HANDLE pour représenter le mutex.

- La création d'un mutex se fait à l'aide de la fonction **CreateMutex**.
- Un thread attendra pour obtenir un mutex à l'aide de la fonction **WaitForSingleObject**, exactement comme dans le cas où il attendrait la complétion d'un thread.
- On relâche un mutex préalablement obtenu avec la fonction **ReleaseMutex**.
- Une fois qu'un processus en a fini du HANDLE d'un mutex, il doit le libérer avec **CloseHandle**, comme dans les cas de nos autres HANDLE.

Mutex

```
#include <iostream>
#include <windows.h>
using namespace std;
HANDLE hMutex; // Variable globale mais pourrait être dans une classe

DWORD WINAPI Thread( void* pParam )
{
char Chaine[50];int i; int T; int* pVal;
   pVal = reinterpret_cast<int*>(pParam);
   sprintf_s(Chaine, "%s %d", "Je suis le thread numero", *pVal);
   i=0;
   // **** 2. Attente du mutex
   WaitForSingleObject(hMutex, INFINITE);
   while(Chaine[i] != '\0')
   {
    cout << Chaine[i];    i++; T = rand()/100; Sleep(T);
   }
   // **** 3. On n'a plus besoin du mutex
   ReleaseMutex(hMutex);  return 0;
}
int main(int argc, char* argv[])
{
HANDLE hThread[3];
int Thread1Param = 1;int Thread2Param = 2;int Thread3Param = 3;
LONG cMax = 2;
   cout << "Trois threads ecrivent a l'ecran avec mutex..." << endl;
   // **** 1. Creation du mutex
   hMutex = CreateMutex( NULL,   // Attributs de sécurité
                         FALSE,  // Pas de prise en charge
                         NULL);  // Pas de nom pour le mutex
   hThread[0] = CreateThread(NULL, 0, Thread, &Thread1Param, 0, 0);
   hThread[1] = CreateThread(NULL, 0, Thread, &Thread2Param, 0, 0);
   hThread[2] = CreateThread(NULL, 0, Thread, &Thread3Param, 0, 0);
   WaitForMultipleObjects(3, hThread, TRUE, INFINITE);
   // **** 4. Relâcher le handle
   CloseHandle(hMutex);   CloseHandle(hThread[0]);CloseHandle(hThread[1]); CloseHandle(hThread[2]);
   cin.get(); return 0;
}
```

Sections critiques

(*critical section*)

Une section critique est une **séquence d'opérations** pendant laquelle un seul thread à la fois doit avoir accès à la ressource.
Une section critique permet de mettre en place une synchronisation à exclusion mutuelle (un peu comme un **mutex**), à l'exception qu'une section critique n'est utilisable que par les threads d'un même processus. L'utilisation de sections critiques est un peu plus rapide que celle de mutex alors dans les cas où les deux solutions seraient applicables, nous devrions favoriser celle des sections critiques.
Tout comme un mutex, la section critique ne peut être possédée que par un seul thread à la fois ce qui est évidemment pratique pour éviter qu'une ressource partagée soit accédée simultanément par deux threads.

Comment utiliser les sections critiques:

- On utilise une variable de type **CRITICAL_SECTION** pour représenter le droit d'entrée dans la section critique. Cette variable doit évidemment être accessible par les threads concernés.
- La création et la mise en place d'une section critique se fait à l'aide de la fonction **InitializeCriticalSection** .Voir ***** 1.**
- Un thread voulant obtenir le droit de travailler dans la section critique fera un appel à **EnterCriticalSection**. Voir ***** 2.**
- Tout thread ayant appelé **EnterCriticalSection** doit faire un appel à **LeaveCriticalSection** une fois son passage dans la section critique complété. Voir ***** 3.**
- Une fois qu'un processus en aura fini de la section critique, au sens où plus un seul thread n'en aura besoin, il devra la libérer avec **DeleteCriticalSection**. Voir ***** 4.**

Sections critiques

(*critical section*)

```
#include <iostream>
#include <windows.h>
using namespace std;
// Variable globale mais pourrait
// être dans une classe
CRITICAL_SECTION SectionCritique;

DWORD WINAPI Thread( void* pParam )
{
   char Chaine[50];char i;int T;int* pVal;
   pVal = reinterpret_cast<int*>(pParam);
   sprintf_s(Chaine, "%s %d", "Je suis le thread numero", *pVal);
   // *** 2. Entrée dans la section critique
    EnterCriticalSection(&SectionCritique);

   i=0;
   while(Chaine[i] != '\0')
   {
        cout << Chaine[i];i++; T = rand()/100; Sleep(T);
   }
    // *** 3. Sortie de la section critique.
    LeaveCriticalSection(&SectionCritique);return 0;
}
int main()
{
   HANDLE hThread[2]; int Thread1Param = 1;int Thread2Param = 2;
   cout << "Deux threads ecrivent a l'ecran avec section critique..." << endl;
   // *** 1. Initialisation de la section
   InitializeCriticalSection(&SectionCritique) ;
   hThread[0] = CreateThread(NULL, 0, Thread, &Thread1Param, 0, 0);
   hThread[1] = CreateThread(NULL, 0, Thread, &Thread2Param, 0, 0);
   WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
   // *** 4. Relâcher les ressources
   DeleteCriticalSection(&SectionCritique);CloseHandle(hThread[0]);CloseHandle(hThread[1]);
   cin.get();
   return 0;
}
```