Examen de SEM AIR2 (durée 3h)

Simulation d'un parking de voiture

Présentation générale

Il s'agit d'écrire des programmes simulant *la journée* d'un parking en une petite dizaine de secondes. Il accueille M voitures durant la journée. Le parking est doté de N places libres à son ouverture le matin.

Le programme est réalisé au moyen de deux processus `voiture` et `simulateur`. Le processus `simulateur` créé le processus `voiture` qui lance les M unités d'exécutions correspondant aux différentes voitures, les voitures sont envoyées toutes les secondes. Ce processus `voiture` laisse rentrer chaque voiture en fonction du nombre de places. Chaque voiture reste garé un temps variable variant entre 1 et 6s puis sort du garage ce qui rend la place de parking disponible. Chaque unité d'exécution informe le processus `simulateur` du temps à attendre pour rentrer dans le parking et du temps resté sur le parking au moment où la voiture quitte le parking. Cela se fait au moyen d'un pipe protégé par un mutex.

N'utilisez pas l'unicode, modifiez les propriétés de chaque projet en indiquant pour le codage des caractères *non défini*.

A la fin de l'examen, vous rendez uniquement des fichiers .c et *non les solutions*. Chaque fichier doit porter exactement le nom indiqué dans le sujet. L'ensemble de ces fichiers doivent être compressé dans un document portant votre prénom et votre nom et transmis par mail.

1 Simulation avec une voiture

Créez une solution contenant un projet `voiture` et un fichier `voiture1.c`. Lorsqu'on écrit en ligne de commande

```
voiture
```

L'affichage obtenu avec les trois unités d'exécutions créés doit être

```
attente_entree_parking
debut_stationnement
quitte_parking
```

Complétez le fichier `voiture1.c` suivant :

```
#include <stdio.h>
#include <Windows.h>
#include <conio.h>
#include <time.h>
VOID test_erreur(LPSTR msg)
{
  DWORD erreur;
  LPSTR msg_erreur;
  erreur=GetLastError();
  if (0==erreur) return;
  FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,erreur,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPSTR)&msg_erreur,0,NULL);
  printf("%s\n Code erreur %X\n Message erreur : %s\n",
    msg,0x0000FFFF&erreur,msg_erreur);
  _getch();
  LocalFree(msg_erreur);
  ExitProcess(erreur);
}
VOID erreur(LPSTR msg)
{
  printf("%s\n",msg);
  ExitProcess(0);
}
INT hasard(VOID)
{
  srand((UINT)time(NULL)*GetCurrentThreadId());
  return 1+rand()%6;
}
DWORD WINAPI voiture(LPVOID param)
```

```
{
  //à ce stade, param n'est pas utilisé
  //affichage
  ...
  //attente d'un nombre de seconde égale à hasard()
  ...
  //affichage
}
INT main(VOID)
{
  HANDLE hVoiture;
  DWORD reponse;
  INT cpt,nb_voiture;
  //création d'autant d'une unité d'exécution
    ...
  //attente que cette unité d'exécution soit terminée
  ...
  //Fermeture du Handle
  ...
  _getch();
  return 0;
}
```

2 Simulation avec des voitures en supposant le parking infini

Remplacez le fichier `voiture1.c` par le fichier `voiture2.c` dans le projet `voiture`. Lorsqu'on écrit en ligne de commande,

```
voiture 3
```

l'affichage obtenu avec les trois unités d'exécutions créés doit être

```
attente_entree_parking
debut_stationnement
attente_entree_parking
debut_stationnement
attente_entree_parking
quitte_parking
quitte_parking
quitte_parking
```

Modifiez la fonction `main` suivante :

```
INT main(INT argc,CHAR * argv[])
{
  HANDLE * hVoiture;
  DWORD reponse;
  INT cpt,nbVoitures;
  //récupération du nombre de voiture entrée en ligne de commande
  if (2!=argc) erreur("voiture2:main:argc");
  ...
  //allocation dynamique avec malloc du tableau de Handle
  //correspondant au nombre de voiture
  ...
  for(cpt=0;cpt<nbVoitures;cpt++) {
    //création d'autant d'unité d'exécution qu'il y a de voitures
    ...
  }
  //attente que toutes les unités d'exécutions soient terminées
  ...
  //Fermeture des Handle et désallocation du tableau de Handle avec free
  ...
  _getch();
  return 0;
}
```

3 Réalisation de la simulation sans faire de statistique

Remplacez le fichier `voiture2.c` par le fichier `voiture3.c` dans le projet `voiture`. Lorsqu'on écrit en ligne de commande,

```
simulation 5 2
```

le processus `voiture` récupère le nombre de voitures 5 et le nombre de places de parking 2. Il créé un sémaphore non-nommé dont le Handle est transmis à chaque unité d'exécution. Ce sémaphore est incrémenté lorsque une voiture rentre dans le parking et décrémenté lorsqu'une voiture quitte le parking. L'affichage dans la fenêtre du processus `voiture` est similaire à ceci :

```
attente_entree_parking
debut_stationnement
attente_entree_parking
debut_stationnement
quitte_parking il reste 1 places
attente_entree_parking
debut_stationnement
quitte_parking il reste 1 places
quitte_parking il reste 2 places
attente_entree_parking
debut_stationnement
attente_entree_parking
debut_stationnement
quitte_parking il reste 1 places
quitte_parking il reste 2 places
```

La fonction `main` est ainsi modifiée :

```
INT main(INT argc,CHAR * argv[])
{
  \\Création d'un sémaphore non-nommé
  ...
  \\Transmission du Handle de sémaphore aux unités d'exécution
  ...
  \\Fermeture du Handle du sémaphore
}
```

La fonction `voiture` est ainsi modifiée :

```
DWORD WINAPI voiture(LPVOID param)
{
  \\Récupération du Handle du sémaphore
  ...
  \\Attente et décrémentation du sémaphore
  ...
  \\Incrémentation du sémaphore et lecture du nombre de places de parking
  ...
}
```

4 Calcul des temps d'attente et utilisation d'un mutex

Remplacez le fichier `voiture3.c` par le fichier `voiture4.c` dans le projet `voiture`. Modifiez la fonction `voiture` de façon à calculer le temps d'attente de cette voiture à l'entrée du parking et le temps de stationnement dans le parking. Affichez ces temps d'attente en protégeant cet affichage-là par un mutex nommé. Vérifiez le bon fonctionnement du mutex en introduisant un temps d'attente protégé par le mutex.

Pour le calcul du temps d'attente, vous utiliserez les instructions suivantes :

```
#include <time.h>
#include <stdio.h>
#include <windows.h>
#include <conio.h>
INT main(VOID)
{
  clock_t ctime;
```

```
  DWORD dwDuree;

  ctime=clock();
  Sleep(200);
  dwDuree=(DWORD)(clock()-ctime);
  printf("la duree est %lf\n",(double)dwDuree/1000);
  _getch();
  return 0;
}
```

5 Conversion de format de données

Les données transmises de `voiture` à `simulation` sont sous la forme suivante :

```
typedef enum Lieu {
  ENTREE=0,
  PARKING=1,
  FIN_TRANSMISSION=2,
} Lieu;
typedef struct Mot {
  Lieu lieu;
  DWORD temps;
} Mot;
```

Cependant pour utiliser le canal non-nommé, il convient de transmettre une chaîne de caractère plutôt qu'une donnée structurée. Créez les trois fonctions suivantes :

```
VOID conversion2str(CHAR str[],DWORD dwSize,Mot mot);
VOID conversion2mot(Mot * lpMot,const CHAR str[]);
BOOL v_conversion(VOID);
```

La première fonction convertit un mot de type Mot en chaîne de caractère. La deuxième fonction convertit une chaîne de caractère en un mot de type Mot. La troisième fonction vérifie sur un exemple qu'à partir d'un mot donné, on retrouve bien le même mot après l'avoir successivement converti en chaîne de caractère puis en Mot.

Rajoutez les trois fonctions et modifiez la fonction `voiture` dans le fichier `voiture4.c` en `voiture5.c` de façon à remplacer l'affichage des temps d'attente à l'entrée du parking et dans le parking par l'affichage d'une chaîne de caractère générée en utilisant la fonction `conversion2str`.

6 Réalisation de la simulation

Modifiez le fichier `voiture5.c` par le fichier `voiture6.c` dans le projet `voiture` et rajoutez le projet `simulation` contenant le fichier `simulation6.c`. Ces deux processus partagent un canal non-nommé (`pipe`), ce canal est utilisé en écriture par `voiture` et en lecture par `simulation`. Ce canal est créé par `simulation`, son Handle est transmis avec héritage à `voiture` puis aux différentes unités d'exécutions. L'accès en écriture dans les différentes unités d'exécutions doit être protégé par le mutex. Ce qui est écrit dedans est constitué exclusivement des chaînes de caractères générées précédemment. Rajoutez une écriture dans le canal par la fonction `main` du fichier `voiture6.c` de façon à informer de la fin de transmission en affichant la chaîne de caractère générée avec

```
const Mot mot_fin={FIN_TRANSMISSION,0};
```

Cette écriture doit avoir lieu quand toutes les unités d'exécutions ont terminées, il n'est donc plus nécessaire de la protéger par le mutex. A la fin de la communication, `simulation` vérifie qu'il a bien reçu toutes les données associées à l'ensemble des voitures (entrée parking et sortie de parking). Il calcule le temps moyen d'attente devant le parking en ajoutant le temps passé par chaque voiture et en divisant par le nombre de voiture. Il calcule de la même façon le temps moyen d'attente dans le parking.