420KH2 – TP00

Vous devez consommer les données provenant d'une source TR au sens strict, qui génère à un rythme constant des données *extrêmement* utiles.

Ces données doivent être prélevées à la source dès qu'elles sont rendues disponibles (respect d'une contrainte TR d'**immédiateté**). Le temps de prélèvement ne doit pas dépasser le délai entre la production de deux trames distinctes de données à la source (respect d'une contrainte TR de **brièveté**).

N'ayant pas encore obtenu l'appareil physique, vous avez mis la main sur un simulateur à bon marché sous forme d'une DLL. Celle-ci a, parmi ses bons côtés, celui de fonctionner à un rythme dicté par le code client.

Voir plus bas pour un exemple de code client, duquel vous pourrez vous inspirer, et pour des consignes quant à la manipulation de ce simulateur *très* sophistiqué.

Votre tâche sera d'écrire le code client capable de consommer **sans pertes** des données à la source, puis de les déposer dans un fichier texte, sous une forme susceptible d'être analysée à l'aide d'un tableur électronique tel qu'Excel.

Votre programme devra s'arrêter proprement suite à une demande de l'usager. Une fois le programme arrêté, on devra pouvoir ouvrir le fichier de données généré avec un tableur et voir la courbe graphique des données reçues.

Contraintes humaines et échéances

Organisation humaine	Équipes de deux personnes (pire cas : trois personnes, mais dans ce cas, il faut un travail impeccable!).
Format de la remise	Imprimé. Ne m'imprimez pas mon propre code (les fichiers que je vous fournis et que vous ne pouvez pas modifier); ce sont les sources résultant de vos efforts que je veux lire!
Date de remise	Au plus tard à la fin du cours du mardi 8 février 2017.

En détail

Le simulateur est une DLL du type « un peu tout croche », comme le sont souvent les outils de bas niveau ayant pour objectif de servir une fin précise.

Livrée sous forme binaire (TP00--fournisseur.dll), elle est accompagnée à la fois d'un fichier d'en-tête (fournisseur.h) et d'un programme de test tout simple (principal.cpp).

Le programme de test fonctionne… un peu, disons-le ainsi, mais ne respecte pas les contraintes TR imposées (donc, au sens d'un STR, *il ne fonctionne pas*).

Le fichier d'en-tête fournisseur.h est tel que décrit à droite. **Vous ne pouvez pas le modifier de quelque manière que ce soit**[1]. Vous remarquerez :

- le symbole FOURNISSEUR_API, défini au tout début. Ce symbole indique si les symboles de la DLL sont exportés ou importés. *Votre chic professeur vous expliquera l'idée derrière cette technique*;
- les fonctions debuter() et terminer() sont là pour les fins de la simulation (donc du TP00) et disparaîtront lorsque la DLL sera remplacée par le véritable appareil. *Pour la simulation, debuter() doit être appelé en début de parcours et terminer() doit être appelé à la fin du programme*;
- la fonction debuter() prend en paramètre deux entiers non-signés indiquant le rythme auquel les données seront générées, exprimé en millisecondes et en microsecondes.

```
#ifndef FOURNISSEUR_H
#define FOURNISSEUR_H
#ifdef FOURNISSEUR_EXPORTS
#define FOURNISSEUR_API __declspec(dllexport)
#else
#define FOURNISSEUR_API __declspec(dllimport)
#endif
void FOURNISSEUR_API __stdcall
   debuter(unsigned millis, unsigned micros);
void FOURNISSEUR_API __stdcall terminer();
struct raw_data {
   enum { BLOCK_SIZE = 1024 };
   double values[BLOCK_SIZE];
};
struct buffer_block {
   enum { NB_BUFFERS = 2 };
   int last_buffer_filled,
       fill_count;
   raw_data data[NB_BUFFERS];
};
int FOURNISSEUR_API __stdcall get_fill_count();
void FOURNISSEUR_API __stdcall
   get_buffer_blocks(buffer_block **);
#endif
```

[1] Je sais qu'il n'est pas joli; je l'ai préparé pour qu'il ressemble à ces trucs pas très jolis qu'on rencontre souvent en pratique. Ne craignez rien : je vous montrerai comment mieux préparer vos propres API, mais il vous faudra, en pratique, savoir interfacer avec ce genre de truc. Ça fait partie de la fonction.

Pour ce qui est de l'exécution de ce « très sophistiqué » simulateur :

- les données seront déposées dans des instances de `raw_data`. Ces `raw_data` sont placés dans des instances de `buffer_block`;
- la simulation remplira les `raw_data` d'un `buffer_block` de manière cyclique (le *zéro*-ième, puis le *un*-ième, puis… puis le `(NB_BUFFERS-1)`-ième, puis le *zéro*-ième, puis…);
- chaque fois qu'un `raw_data` aura été rempli, le `fill_count` du `buffer_block` auquel il appartient sera incrémenté. Le `last_buffer_filled` de ce `buffer_block` indiquera l'indice du plus récent `raw_data` rempli;
- la fonction `get_fill_count()` retournera la valeur du `fill_count` au moment de l'appel, alors que la fonction `get_buffer_blocks()` fera pointer le pointeur reçu en paramètre vers le `buffer_block` à utiliser.

Le code client donné en exemple est tel que décrit à droite.

En particulier, on y remarquera :

- des types représentant ceux des pointeurs de fonctions permettant d'invoquer les services de la `DLL`. *Cette syntaxe est lourde, mais si vous demandez gentiment à votre chic prof de vous expliquer comment faire, il vous montrera une technique pour l'alléger*;
- un chargement explicite, avec `LoadLibrary()`, de la `DLL`;
- une obtention explicite, avec `GetProcAddress()`, des pointeurs sur les fonctions de la `DLL`;

```
#include "fournisseur.h"
#include <iostream>
#include <string>
#include <windows.h>
using pf_debuter_t = void FOURNISSEUR_API (__stdcall *)(unsigned, unsigned);
using pf_terminer_t = void FOURNISSEUR_API (__stdcall *)();
using pf_get_fill_count_t = int FOURNISSEUR_API (__stdcall *)();
using pf_get_buffer_blocks_t = void FOURNISSEUR_API (__stdcall *)(buffer_block**);
int main() {
   using namespace std;
   const auto NOM_DLL = L"TP00--fournisseur.dll";
   HMODULE module = LoadLibrary(NOM_DLL);
   auto pf_debuter =
      reinterpret_cast<pf_debuter_t>(GetProcAddress(module, "debuter"));
   auto pf_terminer =
      reinterpret_cast<pf_terminer_t>(GetProcAddress(module, "terminer"));
   auto pf_get_fill_count =
      reinterpret_cast<pf_get_fill_count_t>(GetProcAddress(module, "get_fill_count"));
   auto pf_get_buffer_blocks =
      reinterpret_cast<pf_get_buffer_blocks_t>(GetProcAddress(module, "get_buffer_blocks"));
// ...
```

- un lancement de la simulation à un rythme lent ($\approx 10\ Hz$[2]);
- une attente active (`while`) de nouvelles données (ceci est quelque peu inefficace, consommant beaucoup de temps sur le processeur!);
- si une trame a été manquée, un *bip!* sonore est produit;
- les données sont saisies et projetées à la console (à titre d'exemple);
- le tout est, ici, répété cinq fois; par la suite, on trouve
- une fin de la simulation; et
- un déchargement explicite, avec `FreeLibrary()`, de la `DLL`.

Ce programme est un exemple, et ne convient pas tel quel à la tâche que vous devez accomplir.

```
// ...
   pf_debuter(100, 0);
   int cur_fill = pf_get_fill_count();
   enum { NESSAIS = 5 };
   for (int i = 0; i < NESSAIS; ++i) {
      int candidat;
      while (candidat = pf_get_fill_count(), candidat == cur_fill)
         ;
      if (candidat - cur_fill > 1)
         cout << '\a' << endl;
      buffer_block *p_block = nullptr;
      pf_get_buffer_blocks(&p_block);
      int last_buf_filled = p_block->last_buffer_filled;
      auto &data = p_block->data[last_buf_filled];
      for (int i = 0; i < raw_data::BLOCK_SIZE; ++i)
         cout << data.values[i] << ' ';
      cout << endl;
      cur_fill = candidat;
   }
   pf_terminer();
   FreeLibrary(module);
}
```

Vous pourrez toutefois vous en inspirer pour accomplir le boulot auquel vous êtes conviés. Vous remarquerez, entre autres, que si un rythme de $10Hz$, donc un délai de $100ms$ entre deux productions de données, suffit pour capturer les données, il est probable qu'aux alentours de $\approx 100\ Hz$, donc un délai de $\approx 10ms$, ou mieux, l'affichage à la console soit si lent que vous ne puissiez plus faire votre travail correctement.

Amusez-vous bien!

[2] À titre de rappel, 1 Hz signifie une fois par seconde.