Université Toulouse III – Paul Sabatier
118 route de Narbonne
31062 Toulouse cedex 9

# Travaux pratiques – n°3
# Utilisation de classe C++ permettant d'implanter une synchronisation de type « moniteur de Hoare »

## Documentation

Deux classes C++ (HoareMonitor et HoareCondition) sont mises à votre disposition sous Moodle afin de réaliser une synchronisation comparable à celle des moniteurs de Hoare étudiés en CTD : exclusion mutuelle entre les opérations « publiques » du moniteur assurée, possibilité de réaliser un blocage avec priorité sur une variable condition, accès à la longueur de la file d'attente associée à une variable condition.

Pour vous aider dans la compréhension de ces classes un exemple d'application est à votre disposition dans l'archive ProdConso_v1_MHoare.zip. Le fichier Makefile associé permet de visualiser les dépendances et la manière dont l'application (ainsi que les classes relatives aux « moniteurs de Hoare ») doit être compilée.

Les classes relatives aux « moniteurs de Hoare », ainsi que les sources dont elles dépendent, sont disponibles sous le répertoire MONITEURS_HOARE de cette archive.

## Exercice 1 – Lecteurs-rédacteurs avec priorité selon l'ordre des demandes

On souhaite gérer les accès parallèles à un fichier partagé selon le modèle des lecteurs-rédacteurs en adoptant une priorité FIFO sur les demandes d'accès (dernière version du TD).

- Implanter ce moniteur à l'aide des classes C++ fournies.
- Écrire une application dans laquelle N threads représentant des lecteurs et M threads représentant des rédacteurs synchronisent leurs accès à un fichier partagé grâce aux opérations proposées par le « moniteur de Hoare » précédemment défini.

Remarque : Vous pouvez repartir du code mis au point lors du TP2.

*[Code à déposer sous Moodle]*

## Exercice 2 – Gestion d'accès à des isoloirs

On veut simuler le comportement de NBE électeurs partageant les accès à NBI isoloirs (avec NBE très supérieur à NBI). Le comportement d'un électeur consiste à arriver au bureau de vote, à entrer dans un isoloir, à y préparer son enveloppe puis à ressortir de l'isoloir pour la placer dans l'urne.

Les contraintes sont les suivantes :

- Un isoloir ne peut être utilisé que par un électeur à la fois.
- Certains électeurs ont une priorité d'accès à ces NBI isoloirs (par exemple, un accès handicapé). Ceci leur permet, en cas d'affluence, de passer devant les électeurs « non prioritaires ».

❖ Proposer une spécification pour un moniteur gérant les accès concurrents de NBE électeurs à ces NBI isoloirs.

❖ Implanter ce moniteur à l'aide des classes C++ fournies.

❖ Écrire une application dans laquelle NBE threads électeurs utilisent les opérations de ce moniteur pour synchroniser leurs accès aux NBI isoloirs existant.

Remarque : Un squelette de code pour les threads électeurs est fourni sous Moodle dans le fichier isoloirs_base.cpp.

**Version 1 –** Proposer une solution utilisant les classes implantant les « moniteurs de Hoare »

*[Code à déposer sous Moodle]*

**Version 2 –** Proposer une solution utilisant les notions vues lors des deux premières séances de TP [« condition » (pthread_cond_t) et « mutex » (pthread_mutex_t)] [**Examen TP de novembre 2018**]

**Remarque** : L'examen de TP n'utilisera pas les classes C++ mais les notions vues lors des deux premières séances de TP. La partie synchronisation du CT portera sur les moniteurs de Hoare.

---

**Rappel** : Si les affichages sont trop rapides, il est possible de temporiser l'exécution d'un thread pendant quelques microsecondes ou nanosecondes à l'aide des primitives :

int usleep (useconds_t usec) ;

int nanosleep(const struct timespec *req, struct timespec *rem) ;

Voir le manuel en ligne pour leur utilisation (man 3 usleep ou man 2 nanosleep).

On peut utiliser une valeur générée aléatoirement (voir les fonctions srand et rand) pour varier les délais d'attente d'un thread à un autre.

Mais, **attention**, la temporisation n'est pas là pour résoudre les problèmes d'accès concurrents à des variables partagée. En d'autres termes : toute exécution d'une application parallèle doit donner un résultat cohérent **sans** temporisation !