Programmation Concurrente
Gestion d'un parking souterrain
TD – pthread

1. Quelques rappels sur les threads

1.1. Création

```
#include <pthread .h>
int pthread_create (
      pthread_t * p_tid,              /* Pointeur sur identite du thread */
      pthread_attr_t * p_attr,        /* NULL : attributs par defaut */
      void * (* fonction )( void *  arg),    /* Fonction executee par le thread */
      void *arg) ;              /* Parametre de << fonction >> */
```

La primitive `pthread_create` crée une nouvelle activié et renvoie son identité à l'adresse `p_tid` (il s'agît d'un numéro entier non signé qui servira par la suite à la gestion du thread). Son argument `attr` définit les attributs du thread (nous utiliserons toujours `NULL`, qui donne les attributs par défaut), `fonction` est un pointeur sur la fonction qui sera exécutée par l'activité (cette fonction retourne nécessairement un `void *` et prend nécessairement un unique argu- ment de type `void *`). Enfin, le dernier argument `arg` correspond à l'argument transmis à la fonction `fonction`. Cette primitive retourne 0 en cas de succès, un code d'erreur sinon.

1.2. Identité

Chaque processus a un numéro unique, le `pid`, qui est renvoyé par la primitive `getpid()`. Tout processus est lui-même décomposé en threads qui ont chacun leur identifiant unique pour un même processus, le `tid`. On notera l'activité `tid` (par exemple `5`) du processus `pid` (par exemple `1234`) sous la forme `pid.tid` (par exemple `1234.5`). Deux primitives sont dédiées à la manipulation des identités des activités :

```
#include <pthread .h>
pthread_t pthread_self(void);
int pthread_equal ( pthread_t tid_1 , pthread_t tid_2 );
```

`pthread_self()` retourne l'identificateur du thread courant dans le processus courant (le `tid`).

`pthread_equal(tid_1,tid_2)` retourne 0 si les deux identités transmises en argument sont identiques et une valeur non nulle sinon.

1.3. Terminaison

Tous les threads d'un même processus prennent fin si l'activité initiale prend fin ou si une des activités fait appel à la primitive `exit()` (ou `_exit()`). Une activité seule prend fin automati- quement quand la fonction passée en argument de la primitive `pthread_create` retourne. Les ressources allouées pour une activité ne sont jamais libérées automatiquement. Les primitives de terminaison d'un thread (sans affecter les autres activités) et de libération de ses ressources sont les suivantes :

```
#include <pthread .h>
void pthread_exit(void * p_status);
int pthread_detach ( pthread_t tid );
```

`pthread_exit(p_status)` termine l'activité qui l'a appelée en indiquant une valeur de retour à l'adresse `p_status`. Cette primitive ne peut jamais retourner dans le thread qui l'a appelée.

`pthread_detach(tid)` indique au système qu'il pourra récupérer les ressources al- louées au thread d'identifiant `tid` lorsqu'il terminera ou qu'il peut récupérer les res- sources s'il est déjà terminé. Cette primitive ne provoque pas la terminaison du thread appelant. Elle retourne 0 en cas de succès, un code d'erreur sinon.

1.4. Synchronisation

Les threads disposent dans les grandes lignes des mêmes outils de synchronisation que les pro- cessus. Le premier d'entre eux est l'attente passive de la fin d'une autre activité qui rappelle les primitives `wait()` ou `waitpid()` liées aux processus. La primitive permettant ce compor- tement est la suivante :

```
#include <pthread .h>
int pthread_join(pthread_t tid, void ** status);
```

Cette primitive suspend l'exécution de l'activité appelante jusqu'à la fin de l'activité d'iden- tifiant `tid`. Si l'activité d'identifiant `tid` est déjà terminée, cette primitive retourne immédia- tement. En cas de succès, elle retourne 0 et place à l'adresse `*status` la valeur de retour de l'activité attendue. En cas d'echec elle retourne un code d'erreur.

L'autre principal outil de synchronisation entre threads est une sorte de moniteur qui utilise un verrou et des variables conditions.

On manipule les verrous à l'aide des quatres primitives fondamentales suivantes qui retournent toutes 0 en cas de succès et un code d'erreur sinon :

```
#include <pthread .h>
int pthread_mutex_init(pthread_mutex_t * p_mutex, NULL); int pthread_mutex_lock (
pthread_mutex_t * p_mutex );
int pthread_mutex_unlock ( pthread_mutex_t * p_mutex );
int pthread_mutex_destroy(pthread_mutex_t * p_mutex);
```

`pthread_mutex_init(p_mutex,NULL)` réalise l'initialisation d'un verrou de type `pthread_mutex_t` dont on aura passé l'adresse en premier argument. Le second argument indique les attributs du verrou, on indiquera `NULL` pour avoir les attributs par défaut qui initialisent le verrou en mode libre.
`pthread_mutex_lock(p_mutex)` réalise l'opération 'prendre' sur le verrou dont l'adresse est passée en argument.
`pthread_mutex_unlock(p_mutex)` réalise l'opération 'rendre' sur le sémaphore dont l'adresse est passée en argument.
`pthread_mutex_destroy(p_mutex)` rend le verrou dont l'adresse est passée en argument inutilisable à moins d'un nouvel appel à `pthread_mutex_init(p_mutex,NULL)`.

La manipulation des variables condition se fait à l'aide des primitives suivantes :
Création :

```
static pthread_cond_t cond_stock = PTHREAD_COND_INITIALIZER;
```

Attente (equivalent du wait Java) :

```
int pthread_cond_wait (pthread_cond_t * cond, pthread_mutex_t * mutex);
```

Signalisation (equivalent du notify Java) :

```
int pthread_cond_signal (pthread_cond_t * cond);
```

Signalisation (equivalent du notifyAll Java) :

```
int pthread_cond_broadcast (pthread_cond_t * cond);
```

2. Gestion d'un parking souterrain

Le but de ce travail est d'implanter une simulation de gestion d'un parking souterrain. Le parking dispose d'un certain nombre de places initial. Des voitures peuvent se présenter à différentes bornes pour demander un ticket.

Dans le cadre de ce travail, chaque borne sera représentée par une thread.

2.1. Version 1

Dans cette version :

- Pour les entrée :
  - Si au moins une place est libre, un ticket est délivré et le nombre de places libres est décrémenté.
  - S'il n'y a plus de places disponibles, pas de décrémentation du compte et un message d'erreur est délivré
- Pour les sorties :
  - S'il y a au moins une place occupée, alors le nombre de places libres est incrémenté
  - S'il n'y a pas de place occupée, pas d'incrémentation du compte et un message d'erreur est délivré

Bien évidemment, la modification du nombre de places libres se fait en section critique

2.2. Version 2

Dans cette version :

- Pour les entrées
  - On remplace le message d'erreur par un blocage de la thread tant que la condition n'est pas remplie.
- Pour les sorties :
  - On remplace le message d'erreur par un blocage de la thread tant que la condition n'est pas remplie.

Version 2a : on réveille toutes les threads

- Comme en Java avec while / broadcast

Version 2b : on ne réveille plus toutes les threads

- Avec if / signal