API de communication entre threads

***Mots-clés** : Threads, communication, synchronisation.*

Spécifications du problème

Présentation

Dans le cadre d'une application multitâche, la communication entre tâches peut être mise en œuvre directement dans le code des tâches (cf. la communication réalisée dans le TD Serveur d'affichage). Mais le code à réaliser est dans ce cas plus lourd et les performances sont impactées par les synchronisations.

Une solution au problème de performance est d'avoir une tâche dédiée à la gestion de la communication. Celle-ci se chargera de récupérer les demandes de transfert et de d'assurer la délivrance de l'information à la tâche destinataire.

Pour simplifier le travail du programmeur, la solution est de lui fournir une API (bibliothèque de fonctions) qui pourra être utilisée dans le code des threads devant échanger des messages.

Travail à réaliser

On souhaite offrir un service de communication par messages aux threads d'une même application. Ce service sera basé sur le principe de « boîte à lettres » ou « file de messages ».

Pour utiliser ce service, un thread devra préalablement s'y abonner. Une fois abonné, il pourra émettre/recevoir des messages à destination/en provenance des autres threads abonnés au service.

Un thread pourra émettre des messages à destination d'un ou de tous les threads abonnés.

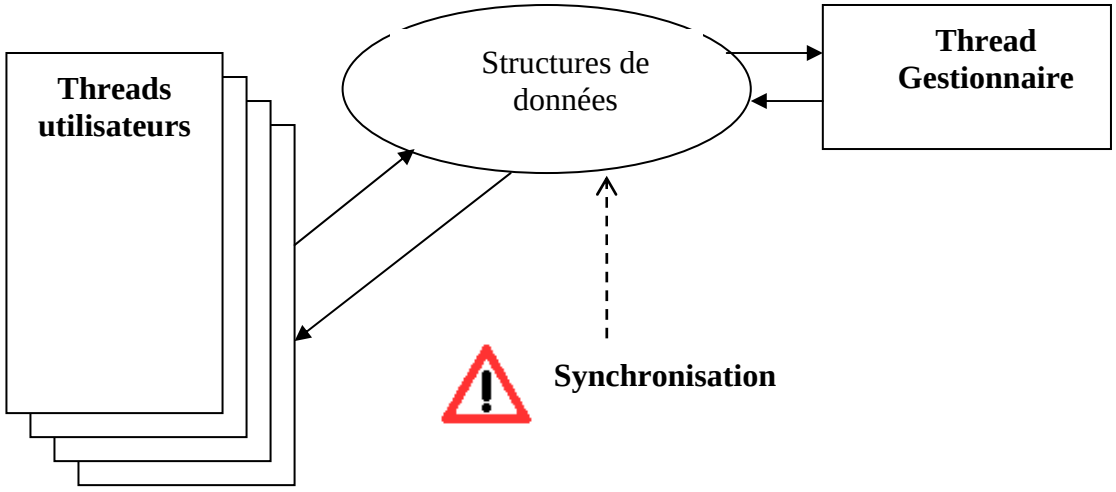
Un thread pourra récupérer les messages disponibles dans sa boîte à lettre (fonctionnement en FIFO).

Le format des messages est laissé à votre choix : chaîne de caractères, structure, ou format variable. On définira dans le code un nombre maximal de messages en attente pour un thread (taille de la boîte à lettres).

Le service sera mis en œuvre de la manière suivante :

- Dans le programme utilisateur, il sera utilisable via une API (ensemble de fonctions) qui sera invoquée par les threads désirant s'échanger des messages.
- Un thread supplémentaire sera chargé de la gestion de la communication. Il récupérera les demandes de transfert et assurera la délivrance de l'information à la tâche destinataire. Ce thread ne sera pas visible par le programmeur utilisateur de l'API.

Schéma général



L'API

La liste a minima des fonctions de l'API et de leurs fonctionnalités est fournie ci-après. Leur prototype n'est volontairement pas précisé, il dépendra de vos choix d'implémentation. Cependant, chaque fonction sera susceptible de renvoyer un code d'erreur.

1. **Initialisation du service : initMsg( )**

   Cette fonction lance le thread de gestion et effectue toutes les initialisations nécessaires. Elle doit être appelée une seule fois et avant toute autre fonction de l'API.

   Erreurs : service déjà lancé, erreurs « techniques » lors des initialisations.

2. **Abonnement d'une tâche au service : aboMsg( )**

   Cette fonction signale qu'une nouvelle tâche souhaite utiliser les services du gestionnaire de communication. Elle doit fournir un identifiant (nom ou numéro selon votre choix d'implémentation) permettant de la désigner lors des échanges ultérieurs. Elle doit être appelée une seule fois par fonction et avant tout échange de message par celle-ci.

   Erreurs : service non lancé, tâche déjà abonnée (= identifiant existant), nombre max. de tâches abonnées atteint, .erreurs « techniques », … …

3. **Emission d'un message : sendMsg( )**

   Cette fonction demande au thread gestionnaire de mettre un message à disposition d'une autre tâche (ou de toutes les tâches abonnées).

   Erreurs : service non lancé, tâche émettrice ou destinataire non abonnée (= identifiant inexistant), erreurs « techniques », … …

4. **Récupération d'un message : recvMsg( )**

   Cette fonction permet à une tâche de demander au thread gestionnaire s'il dispose d'un message pour elle. La récupération se fera en FIFO. Si aucun message n'est disponible, la fonction est bloquante.
   Elle devra aussi permettre d'informer une tâche du nombre de messages à sa disposition, sans effectuer de retrait. Dans ce cas, elle n'est pas bloquante et retourne le nombre de messages disponibles dans la boîte aux lettres.

   Erreurs : service non lancé, tâche émettrice ou destinataire non abonnée (= identifiant inexistant), erreurs « techniques », … …

5. **Désabonnement au gestionnaire : desaboMsg( )**

   Cette fonction informera le gestionnaire qu'une tâche ne souhaite plus disposer de ses services. A partir de ce moment, aucune autre tâche ne pourra lui envoyer de message. Les messages présents dans la boîte à lettres de la tâche sont alors perdus.

   Erreurs : service non lancé, tâche non abonnée (= identifiant inexistant, erreurs « techniques », messages en attente et pas de demande d'abandon, …

6. **Terminaison du gestionnaire : finMsg( )**

   La terminaison du gestionnaire ne devrait avoir lieu que si aucune tâche n'y est abonnée. Cependant, un flag permettra de forcer la terminaison.

   Erreurs : service non lancé, présence de tâches abonnées, erreurs « techniques », …

Implémentation

Forme de l'API

L'API se présentera comme un ensemble de fonctions. Vous prévoirez aussi un fichier d'entête qui contiendra les variables globales nécessaires à sa mise en œuvre.

Thread gestionnaire

Le gestionnaire de communication est mis en œuvre par un thread lancé par la fonction d'initialisation. Il pourra lui-même créer d'autres threads chargés du traitement d'une communication particulière afin d'améliorer les performances.

Mécanismes de communication

Dans la mesure où la communication se fait entre threads du même processus, vous utiliserez des variables globales comme zones de communication.

Mécanismes de synchronisation

Vous utiliserez les mécanismes à votre disposition dans l'API Posix, selon le besoin et vos choix d'implémentation. Pour mémoire, vous disposez de mutex, de sémaphores et de variables conditionnelles.

Remarque technique

Dans la mesure où le nombre de zones de communication et d'objets de synchronisation dépend du nombre de tâches abonnées, on se rappelle que le C permet d'allouer dynamiquement de la mémoire.

Par exemple :

```
// Allocation d'un tableau de 20 chaînes de 10 caractères chacune
char ** tabMessage :
int i ;
if ((tabMessage = (char **)malloc(20 * sizeof(char*))) == NULL) {
     printf("plus de memoire disponible\n");
     exit(-1) ;
}
for (i=0 ; i< 20 ; i++) {
     if ((tabMessage[i] = (char*)malloc(10*sizeof(char))) == NULL)
     printf("plus de memoire disponible\n");
     exit(-1) ;
}
```

```
// Allocation d'un mutex
pthread_mutex_t *m;
if ((m = malloc(sizeof(pthread_mutex_t))) == NULL) {
     printf("plus de memoire disponible\n");
     exit(-1) ;
}
pthread_mutex_init( &m, NULL );
// ...
pthread_mutex_destroy( &m);
```

```
// Allocation d'une variable conditionnelle
pthread_cond_t *v;
if ((v = malloc(sizeof(pthread_cond_t))) == NULL) {
     printf("plus de memoire disponible\n");
     exit(-1) ;
}
pthread_cond_init( &v, NULL );
// ...
pthread_cond_destroy( &v);
```

Tests de l'API

Vous devrez réaliser une application multitâche utilisant les fonctions de votre API.

Travail à rendre

Ce travail doit s'inscrire dans une démarche projet. Comme tout projet, il doit comporter une phase d'**analyse** (que fait l'application ?), de **conception** (comment va-t-on traiter techniquement le problème ?), de **réalisation** (codage) et de **tests** (permettant de vérifier que l'application fonctionne dans le cas nominal, les cas aux limites et les cas d'erreur).

Remarque : Un rebouclage sur les phases précédentes est à prévoir en cas de problème.

Rapport

Un rapport écrit (documents imprimés ou manuscrits acceptés) doit rendre compte des phases du projet précédemment citées.

Son plan idéal est donc le suivant:

1. **analyse du problème** : Vous disposez d'un cahier des charges (sujet du BE). Il est inutile de le réécrire, vous vous contenterez, si nécessaire, d'apporter des compléments.
2. **étude technique** (= document de conception) : doit détailler l'architecture de l'application (tâches et relations entre elles), décrire les structures de données utilisées, décrire les choix techniques effectués en les justifiant et en discutant leurs limites éventuelles, et comporter l'algorithme de chaque tâche/fonction. N'oubliez pas de traiter en particulier de la gestion des erreurs et des phases de lancement et de terminaison de l'application.
   Cette étude technique doit pouvoir être fournie à un programmeur et lui permettre de coder l'application sans information complémentaire.
3. **dossier de tests** : = liste des tests à effectuer pour valider le bon fonctionnement de votre code. Ces tests doivent être spécifiés **avant** de commencer le code. Il est aussi fourni au programmeur. Vous devrez écrire une application multithread permettant de tester toutes les fonctionnalités de votre API.
4. **résultats des tests** : Votre code fonctionne-t-il? Y a t-il des problèmes non résolus? D'où proviennent t-ils?
5. **conclusion** :

Les trois premiers items sont à rédiger **AVANT** de coder, même si d'éventuels problèmes techniques rencontrés lors du codage peuvent induire des modifications dans l'étude technique.

Fichiers source

Vous remettrez les fichiers source. Le code doit être lisible et commenté.

Organisation

- Travail en binôme : vous rendrez un rapport et un code par groupe.
- Encadrement : ce BE est à réaliser majoritairement en autonomie, cependant n'hésitez pas à poser des questions à vos enseignants :

  - sur le forum e-campus : https://e-campus.enac.fr/moodle/mod/forum/view.php?id=3192
  - par mail : joelle.luter@enac.fr
  - Alain Bannay : C125, Rémi Coudarcher, Joëlle Luter : C123
- - Délais : La date de remise du rapport est fixée au 30/11/2015, celle du code, des résultats des tests et du bilan au 14/12/2015. Tout retard sera pénalisé.