Programmation système et concurrente

API de communication entre threads

Un rapport de projet proposé par Hamadou Djibrilla et Pastre Guillaume.

Présentation :

La communication entre threads d'une application multitâche est possible et peut être directement dans le code des taches. Cependant, ce code est lourd, il faut bien penser aux synchronisations pour assurer des performances et ne pas impacter la protection des données.

C'est dans ce cadre que nous proposons de simplifier le travail du programmeur en lui apportant une bibliothèque de fonctions qui seront appelées dans le code des threads pour qu'ils puissent communiquer, faisant un code léger pour le programmeur tout en assurant synchronisation et protection des données.

Table des matières

Analyse

Cette étude a pour but de proposer une aide au programmeur pour faire communiquer ses threads, qu'ils aient une boîte aux lettres, gérée par un thread gestionnaire invisible par l'utilisateur.
Quoi qu'il en soit, en cas d'erreur, nous devons renvoyer l'erreur à l'utilisateur afin qu'il effectue les traitements en conséquence, nous ne fermerons pas brutalement l'application de l'utilisateur.
Un guide sera fourni pour savoir comment utiliser notre API, bien que nous cherchons à rendre l'utilisation la plus intuitive possible.

Fonctions que l'API doit apporter :

initMsg() :

Un utilisateur veut faire communiquer ses n threads avec des messages d'une certaine taille maximale, et fait alors appel à notre API pour mettre en route le système de communication. Dans le thread où il voudra démarrer la communication, il appellera la fonction initMsg().
Si le service est déjà lancé ou si les paramètres données sont trop ambitieux, une erreur sera renvoyée.

aboMsg() :

Pour qu'il puisse envoyer ou recevoir des messages, un thread doit au préalablement s'abonner au service de communication. Il faut pour cela que le système soit lancé (gestionnaire lancé), sinon il recevra une erreur. De la même façon si le thread est déjà abonné il en sera informé par une erreur, ainsi que s'il n'y a plus de place en mémoire pour l'abonner ou autre erreur technique.

sendMsg() :

Si un thread veut envoyer un message à un autre thread, il fera appel à notre fonction sendMsg(), donnant le thread destinataire et le message à transmettre. Notre gestionnaire se chargera de le transmettre à destination.
Le thread recevra une erreur si :

- Le gestionnaire n'est pas lancé
- Le thread expéditeur n'est pas abonné
- Le thread destinataire n'est pas abonné
- Un problème de communication survient

recvMsg() :

Lorsqu'un thread voudra récupérer un message ou savoir combien il en a reçu, il appellera notre fonction recvMsg().
Cette fonction pourra être utilisée pour :

- ✔ savoir combien de messages le thread a reçu (non bloquant)
- ✔ récupérer k messages parmi les n reçus (non bloquant si k<=n, bloquant si k>n (si a besoin du message pour continuer son exécution par exemple))

Une erreur sera renvoyée si le gestionnaire n'est pas lancé, si le thread appelant la fonction n'est pas abonné, et en cas de problèmes techniques de communication.

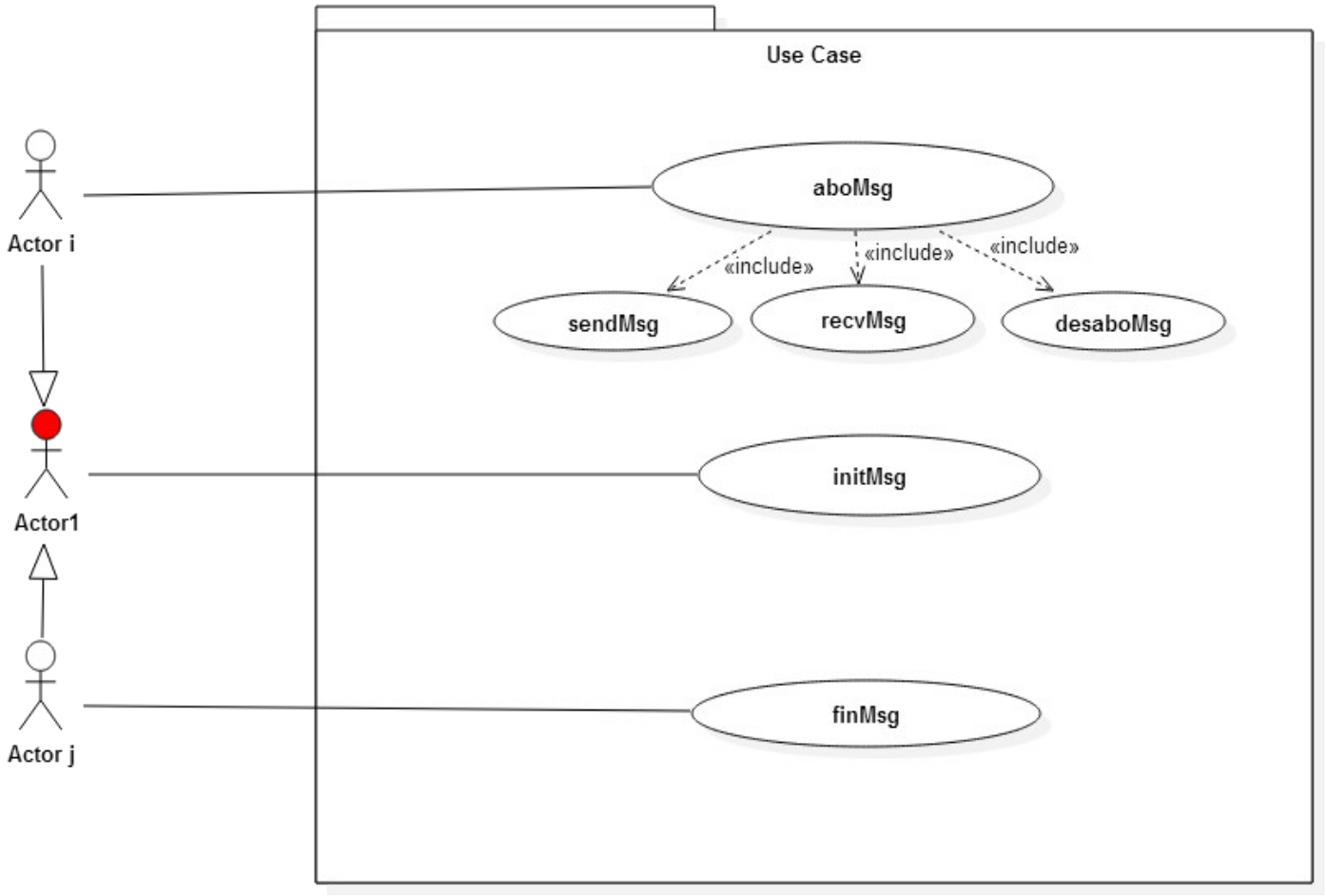
desaboMsg() :

Lorsqu'un thread abonné au système de communication ne souhaite plus recevoir ou envoyer de messages, il peut choisir de se désabonner grâce à la fonction desaboMsg(). Il perdra alors les éventuels messages en attente qu'il avait reçu, mais sera informé de leur perte. Dès lors, aucun autre thread ne pourra lui envoyer de message.
Une erreur sera renvoyée dans le cas où le thread cherchant à se désabonner n'était pas abonné, et si le gestionnaire n'était pas lancé.

finMsg() :

Lorsque l'utilisateur voudra mettre fin à notre service de communication entre threads, il pourra faire appel à la fonction finMsg(). Dans le cas nominal, aucun thread ne devrait être encore abonné lorsque l'on voudra mettre fin au gestionnaire. Mais dans le cas contraire, l'utilisateur pourra quand même le fermer sur présentation d'un flag, signalant l'urgence de la situation.
Une erreur sera renvoyée si le gestionnaire n'était pas lancé, et on renverra le nombre de threads encore abonnés lors de la fermeture brutale.

Voici un diagramme UML pour expliciter les interdépendances entre fonctions :



Toutes les fonctions ont effectivement besoin que le gestionnaire soit lancé, c'est à dire que la fonction initMsg() ait été appelée au préalable, et les fonctions sendMsg(), recvMsg() et desaboMsg() ont en plus besoin d'avoir au préalablement lancé aboMsg().

On remarque cependant qu'on pourra mettre fin au système de communication sans être au préalablement abonné, par exemple si un thread main lance ses threads de calcul qui communiqueront, il pourra le fermer sans lui-même avoir été abonné.

Voyons désormais les détails de conception.

Conception

Généralités

Structure des messages :

Le but de ce projet est de fournir un système de communication entre threads facile d'utilisation. Il nous faut tout d'abord définir le format de messages que les threads vont utiliser pour communiquer.
Comme nous avons choisi de faire en sorte que le thread gestionnaire, c'est à dire le thread que nous créons lors de l'appel à la fonction initMsg(), soit en charge de récupérer les messages que les threads envoient pour ensuite les redistribuer au thread destinataire, il était nécessaire d'avoir un identifiant du thread destinataire.

D'autre part, si par exemple l'utilisateur utilise différents threads pour effectuer du calcul parallèle, les threads attendront des résultats de calculs que d'autres threads ont effectués, mais il nous faut alors un identifiant du thread expéditeur dans le message pour savoir à quelle portion de calcul ce message correspond.

C'est donc dans cette optique que nous avons choisi d'utiliser comme message une structure comprenant l'identifiant du thread destinataire, l'identifiant du thread expéditeur, et le message en lui-même sous forme de chaîne de caractères. Structure message :

destinataire
expéditeur
msg

Structure de la table d'abonnés :

Rappelons que nous proposons un système de communication entre threads pour lequel il faut d'abord s'abonner, afin qu'ils puissent recevoir et envoyer des messages. Mais les threads ne doivent pas s'abonner plusieurs fois, il faut donc conserver une trace des threads abonnés pour empêcher ce type d'erreur et savoir qui peut discuter avec qui.
Nous avons donc pour cela créé une structure contenant l'identifiant du thread, le nombre de message qu'il a en attente (il s'incrémente lorsqu'on lui envoit un message, et se décrémente lorsqu'il lit un message), et la clef de sa file de message afin que l'on n'ait pas à la générer à nouveau à chaque fois.

Un tableau global d'abonnés sera créé à l'appel de la fonction initMsg(), pour lequel on réservera un espace mémoire avec le nombre maximum d'abonnés fourni par l'utilisateur (limité à 20 arbitrairement).
Structure abonne :

id_abonne
nbre_messages
id_file_desc

Protection des données et synchronisation

Comme nous utilisons des variables globales comme moyen de communication entre threads, nous devons les protéger afin de s'assurer qu'une seule tâche accède à la ressource en même temps, à la fois pour la consulter avant d'effectuer des traitements, et pour la modifier. C'est dans cette optique que nous avons choisi d'utiliser un mutex, les traitements n'étant pas très longs (et ne pouvant exploser avec les limites que nous imposons). La protection est donc assurée par exclusion mutuelle.
Par ailleurs, pour éviter que le gestionnaire ne monopolise le processeur inutilement à tester s'il y a du courrier à distribuer ou non, il sera réveillé par une variable conditionnelle.

Communication

Nous voulons mettre en place un système de boite aux lettres, où les threads pourront consulter quand ils le souhaitent les messages qu'ils ont reçus, en FIFO. Nous utiliserons donc des files de messages comme système d'arrivée des messages vers les threads. Il y aura donc une file de message dite « descendante » venant du gestionnaire vers chaque thread.
En revanche, pour envoyer les messages qui doivent d'abord passer par le gestionnaire, on utilisera une file de message commune à tous les threads, appelée file montante, dont la clef et l'identifiant sera connue par tous grâce à des variables globales. Les threads enverront donc leur message à envoyer contenant l'id du destinataire dans la file montante, le gestionnaire le récupérera, lira l'identifiant et le placera dans la file descendante correspondante.

Variables globales

Nous avons choisi d'utiliser des variables globales afin que les threads puissent s'abonner, se désabonner et recevoir leurs messages eux mêmes. En effet, lors de l'appel à initMsg(), un **tableau global d'abonnés** est initialisé vide, et chaque appel à aboMsg() se chargera de vérifier si le thread n'est pas déjà abonné, s'il reste encore de la place dans le tableau avant de s'insérer lui-même à la suite du tableau.
De la même façon, pour savoir jusqu'où le tableau et donc le nombre d'abonnés courant, une variable globale **nombre_abonnes** sera accessible et incrémentée à chaque abonnement et décrémentée à chaque désabonnement. L'abonnement sera limité par la variable globale **nombre_max_abonnes** qui sera établie lors de l'initMsg(). La **taille maximale des messages** et la **taille des boites aux lettres** seront aussi globales.
Pour la communication, la **clef et l'id de la file montante** seront globaux.
Le **mutex** et la **variable conditionnelle** seront également globaux car utilisés par plusieurs threads.
Des flags pourront être utilisés, notamment un **flag_gestionnaire**, qui se met à 1 lors de l'init(), et qui lorsque mis à 0 empêche l'appel aux fonctions nécessitant la présence du gestionnaire, et désabonnera tous les threads utilisateurs et terminera le thread gestionnaire afin de terminer le service.

Détail des fonctions

initMsg()

Notre fonction initMsg() sera appelée par un thread qui voudra lancer le système de communication. Concrètement, la fonction actualise les variables globales avec les paramètres que l'utilisateur lui aura donné (taille des messages, nombre maximum d'abonnés, nombre de messages maximum en attente) comme les utilisations du système peuvent être très différentes, et lance le thread gestionnaire qui sera chargé de distribuer le courrier (mettra le flag_gestionnaire à 1). Elle ouvrira également la file montante.
Regardons un peu désormais cette distribution de courrier.
Le code de notre thread gestionnaire sera une boucle infinie, qui consiste à distribuer le courrier tant qu'on ne lui demande pas d'arrêter. Il lira donc dans la file montante, s'il n'y a aucun message il attendra (variable conditionnelle), sinon il lira le destinataire et mettra la message dans sa file. Il gérera les erreurs et la demande de fermeture brutale.

aboMsg()

Appelée par un thread qui veut utiliser le service, aboMsg() lui demandera un identifiant en paramètre, a priori son identifiant thread obtenu par pthread_self() par exemple, mais pourra aussi être un entier donné au gré de l'utilisateur. La fonction regardera tout d'abord si le système est bien lancé, par exemple en testant si le flag_gestionnaire vaut 1. Il est nécessaire en effet d'avoir fait appel à initMsg() au préalable, comme le montre le diagramme UML dans la rubrique Analyse.
La fonction cherchera alors l'identifiant donné dans le tableau global d'abonnés courants (en ayant le mutex), et si elle le trouve cela voudra dire que le thread a déjà été abonné (ou l'identifiant est pris) et alors l'abonnement ne peut se faire.
Il faut donc qu'elle ne trouve pas l'identifiant pour vouloir ajouter le thread au tableau, mais il faut également que le nombre d'abonnés actuel ne soit pas égal au nombre maximal d'abonnés.
Alors la fonction ajoute le thread dans le tableau des abonnés (générant au préalable la clef de sa file pour la mettre dans le tableau) et incrémente le nombre d'abonnés.

sendMsg()

La fonction sendMsg() sera appelée par les threads désirant envoyer un message à un autre thread abonné.
Tout d'abord, le gestionnaire doit être lancé, un test sur le flag_gestionnaire sera mis en place.
Ensuite, les threads expéditeur et destinataire doivent être abonnés, il y aura donc une recherche dans le tableau global d'abonnés des deux threads, renvoyant une erreur si un des threads au minimum n'est pas abonné.
Ensuite, un test sur le nombre de messages en attente qu'a le thread destinataire sera mis en place, en effet l'utilisateur nous a fourni une taille maximale de boîte aux lettres, nous ne devons pas la dépasser. En cas de limite atteinte, le message ne sera pas envoyé, la fonction renverra une erreur.
Si tous ces tests sont passés, on génère le message en tant que structure décrite plus haut, avec l'identifiant du

thread destinataire, celui du thread expéditeur et le message en chaîne de caractères qu'on aura tronqué si nécessaire à la taille maximale de messages fourni par l'utilisateur.
Une fois ce message créé, la fonction le place dans la file montante, destinée au gestionnaire. On incrémente alors le nombre de messages en attente par le thread destinataire, et on envoie un signal sur la variable conditionnelle du gestionnaire.

rcvMsg()

Cette fonction sera appelée par les threads désireux de lire les messages qu'ils ont reçu, ou savoir combien de messages ils ont en attente. Et pour ce faire l'utilisateur doit le signaler à travers un flag passé en paramétre.
Comme pour sendMsg(), cette fonction a besoin que le gestionnaire soit lancé, et que le thread appelant soit abonné, donc les mêmes tests seront présents.
Il nous fallait une façon de différencier les deux fonctionnalités de la fonction, la vérification du nombre de messages étant non bloquante, et la récupération des messages devant pouvoir être bloquante, si jamais un thread avait besoin de récupérer un résultat avant de pouvoir continuer par exemple.
Nous avons donc établi que l'utilisateur donnerait en paramètre l'identifiant du thread appelant la fonction, mais également le nombre de messages qu'il voulait récupérer. Si l'utilisateur appelle rcvMsg() en demandant 0 messages (ou un nombre négatif), cela fera appel à la fonctionnalité de la lecture du nombre de messages en attente, dans le tableau des abonnés dans la case correspondant à son identifiant. Sinon, il peut demander k messages, alors s'il y avait k messages ou plus dans sa file, il les récupèrera et continuera son exécution, mais s'il y en avait moins, il récupérera les messages qui étaient là, et attendra les suivants jusqu'à faire un compte de k messages récupérés au total. A chaque récupération de message on décrémentera le compteur de messages dans le tableau.

desaboMsg()

La fonction desaboMsg() sera appelée par un thread abonné voulant se retirer du système de communication.
Le gestionnaire doit être lancé et le thread doit être abonné pour vouloir se désabonner. Les tests des deux fonctions précédentes seront encore effectués ici.
DesaboMsg prendra en paramètre l'identifiant du thread qui l'appelle, le cherchera dans la table d'abonnés, et renverra une erreur si le thread n'est pas abonné. Si le thread était abonné, il l'enlèvera du tableau global d'abonnés après avoir récupéré le nombre de messages qu'il avait en attente, et décrémentera la variable globale nombre_abonnes. Le tableau global d'abonnés sera alors reconstruit pour éviter les trous et avoir un bon indice.

finMsg()

Appelée par un thread quelconque, finMsg() aura pour mission de mettre fin au système de communication.
Le gestionnaire devra être lancé pour l'exécution de finMsg(), mais le thread appelant ne devra pas forcément être abonné. La fonction prendra en paramètre un flag, interprétant l'urgence de la situation. Si le flag est à 0, la fonction renverra une erreur si des threads sont encore abonnés au système, et ne le terminera pas. Si aucun thread n'est abonné, il fermera la file montante et mettra le flag_gestionnaire à 0, signalant au gestionnaire qu'il faut qu'il se termine. (un signal sur la variable conditionnelle sera alors à prévoir).
Si le flag donné en paramètre vaut 1, alors la fonction fermera toutes les files de messages (id dans le tableau d'abonnés), mettra le nombre d'abonnés à 0, fermera le gestionnaire et retournera le nombre de threads qui étaient encore abonnés lors de la fin brutale du système.

Tests

InitMsg() devra renvoyer une erreur si :

- le gestionnaire était déjà lancé
- pas assez de mémoire n'est disponible
- l'initialisation des moyens de communication s'est mal passée
- le lancement du thread gestionnaire s'est mal passé

AboMsg() renverra une erreur si :

- Le gestionnaire n'était pas lancé
- le thread appelant était déjà abonné
- le nombre maximal d'abonnés est atteint
- une erreur se produit lors de l'ouverture de sa file de messages

SendMsg() retournera une erreur dans les cas suivants :

- Le gestionnaire n'était pas lancé
- un des threads n'était pas abonné
- la boîte du destinataire est pleine
- une erreur est apparue lors de la communication

RcvMsg() devra être robuste si :

- Le gestionnaire n'était pas lancé
- le thread n'était pas abonné
- une erreur de communication survient

DesaboMsg() devra être prêt si :

- Le gestionnaire n'était pas lancé,
- le thread n'était pas abonné
- La fermeture de la file ne se passe pas correctement

FinMsg() nous fera part d'une utilisation anormale si :

- Le gestionnaire n'était pas lancé
- des threads étaient encore abonnés (si flag = 0)
- la fermeture des files de messages s'est mal passée

Manuel d'utilisation

Bienvenue dans le manuel d'utilisation de notre bibliothèque de communication entre threads. Vous trouverez ci-dessous tout ce qu'il faut savoir pour utiliser correctement notre bibliothèque de communication entre threads.

Avant toute chose, n'oubliez pas d'importer notre bibliothèque dans votre code.

Ensuite, lorsque vous jugez utile d'établir une communication entre threads que vous aurez créés, faites appel à notre fonction **initMsg()**, par exemple dans votre main, ou dans le premier thread qui voudra communiquer.

Cette fonction mettra en route le système de communication, lançant un thread gestionnaire qui sera invisible pour vous, mais qui permettra le bon fonctionnement de la communication.
Notre fonction initMsg() prend 3 paramètres : le nombre maximal de threads que vous aurez à faire communiquer (limité à 20), la taille (nombre de caractères) des messages que vous enverrez entre threads (le message sera transmis sous forme de chaîne de caractères, limité à 80) ainsi que le nombre maximal de messages qu'un thread peut avoir dans sa boîte aux lettres (limité à 10).
Ces paramètres nous seront utiles pour ajuster au mieux la mémoire nécessaire à la communication, en assurant des performances acceptables.

Vous avez déjà fait appel à initMsg() ? Félicitations ! Mais sachez que ce n'est pas encore fini ! En effet, pour communiquer entre threads, ceux-ci doivent se connaître. Il est donc nécessaire pour les deux threads de s'abonner au système de communication grâce à **aboMsg()** (veillez donc à indiquer que vous voulez faire communiquer au moins 2 threads lors de l'appel à initMsg!). Dès lors, les threads auront une visibilité l'un sur l'autre et pourront s'échanger des messages. Mais cette partie viendra juste après.
En faisant appel à aboMsg() dans vos threads, vous donnerez un identifiant du thread, par exemple son id_thread par la fonction pthread_self() de la bibliothèque pthread par exemple, ou juste un entier que vous noterez sur le coin d'un papier pour garder la correspondance.
Maintenant que vos threads sont abonnés, faisons les communiquer !

Rentrons dans le vit du sujet : Lorsque vous voulez faire communiquer vos threads, lorsque par exemple un thread a fini un calcul et doit transmettre le résultat à un autre, vous pourrez faire appel à notre fonction **sendMsg()**, à qui vous fournirez l'identifiant du thread destinataire, l'identifiant du thread expéditeur, et votre message en tant que chaîne de caractères. Rappelez-vous que vous avez limité vous-même la taille des messages lors de l'appel d'initMsg(), tenez-vous y, ou vos messages seront tronqués et vous perdrez de l'information ! Un mécanisme sophistiqué sera alors mis en route pour convoyer votre message jusqu'à votre thread destinataire, qui ne rêvait que de recevoir ce message.
Attention, n'oubliez pas que vous devez abonner vos threads avant de pouvoir utiliser sendMsg(), référez-vous plus haut si vous avez oublié comment faire.

Très bien, maintenant que vous savez envoyer un message (sous forme de chaine de caractères, n'oubliez pas), voyons un peu comment le récupérer afin de l'exploiter. Nous vous proposons un système de réception de messages avec deux facettes, pour faire face à toutes les situations. Mais pour éviter la profusion de fonctions et la confusion entre tous leur noms, ces fonctionnalités seront disponibles toutes les deux grâce à **rcvMsg()**.
Notre fonction rcvMsg() prend deux paramètres, l'identifiant du thread qui l'appelle, et le nombre de messages qu'il veut lire/recevoir. C'est là que réside toute l'astuce, alors accrochez-vous bien. Si vous voulez juste savoir combien de message ce thread a en attente, vous pourrez appeler rcvMsg() en demandant 0 message ! (ou un nombre négatif, mais c'est de suite moins joli). L'action sera alors non bloquante, et la valeur de retour de rcvMsg() sera la nombre de messages qu'on a destiné à ce thread mais qui n'ont pas encore été récupérés. Dès lors, vous pourrez récupérer le nombre de messages en attente ou moins en passant en paramètre ce nombre à rcvMsg(), et votre thread continuera son exécution après cette récupération. Mais si vous n'avez reçu que deux messages alors qu'il vous en fallait trois pour continuer votre exécution, n'ayez crainte ! Appelez rcvMsg() en donnant 3 en paramètre, votre thread lira les 2 premiers messages, et se mettra en attente du troisième, et ne se débloquera que lors de sa réception. Plutôt sympa, non ?
N'oubliez pas qu'il faut être abonné pour recevoir des messages, et qu'ils seront reçus sous forme de chaine de caractère, que vous devrez formater par vous-même s'il vous fallait un autre type.

Parce qu'il est parfois préférable de se taire, nous vous proposons de désabonner vos threads qui n'auront plus à communiquer dans la suite de votre utilisation de notre système. Pour cela, appelez **desaboMsg()** avec l'identifiant du thread à désabonner, il sera alors inconnu des autres threads (sauf peut-être si vous avez fait des collisions dans les noms de vos threads et avez alors supprimé le mauvais, mais alors c'est de votre faute, il fallait suivre nos conseils d'utiliser pthread_self()). Si le thread récemment désabonné avait des messages en attente qu'il n'a pas retiré, il ne pourra pas les récupérer, mais sera informé du nombre de messages perdus par la valeur de l'entier retour de desaboMsg().

Et comme toutes les bonnes choses ont une fin, même si cela vous déchire le cœur, il faut parfois mettre fin à notre système de communication. Mais croyez bien qu'il est plus propre de terminer le système en appelant **finMsg()** qu'en fermant violemment votre application. Dans l'idéal, désabonnez préalablement tous vos threads qui utilisaient le système de communication, et appelez ensuite finMsg() avec en paramètre 0. Si vous devez fermer en urgence en revanche, appelez finMsg() avec le paramètre 1, la fermeture sera brutale, mais vous saurez combien d'abonnés vous avez perdus dans la manœuvre par la valeur de retour de finMsg().
Enfin, n'essayez pas d'arrêter le système si vous ne l'avez pas lancé au préalable, ça ne peut pas marcher.

Liste des codes retour en cas d'erreur (ou non) :

- 0 : Tout s'est correctement passé (ou 0 message en attente)
- 1 : Le gestionnaire n'est pas lancé
- 2 : Un des threads au moins n'est pas abonné
- 3 : Le gestionnaire est déjà lancé alors qu'il n'aurait pas du
- 4 : Mémoire non disponible
- 5 : Erreur lors de la manipulation des files de messages
- 6 : Le thread est déjà abonné (et n'aurait pas du)

Voilà, vous savez tout pour bien utiliser notre bibliothèque, nous vous souhaitons une bonne programmation, en espérant vous avoir aidé dans votre tâche.
Bien cordialement,