Université Denis Diderot - IUP2 - Systèmes
Présentation du projet 2003

• **Introduction**

Le système UNIX propose différents mécanismes de communication entre processus (tubes anonymes ou nommés, files de messages) qui présentent l'inconvénient de nécessiter la recopie des données échangées entre les processus dans des *buffers* du système : un processus expéditeur prépare dans son espace d'adressage privé l'information qu'il souhaite transmettre, puis en demande la recopie par le système dans l'espace associé à un objet de communication au travers d'un appel système (`write` ou `msgsnd`) (tube ou file de messages). Inversement, un processus récepteur demande, par un appel système (`write` ou `msgrcv`) si des données sont disponibles en lecture dans un objet de communication donné : dans ce cas le système procède à la recopie de ces informations dans l'espace d'adressage du processus récepteur.

Ces différentes recopies sont bien évidemment pénalisantes du point de vue des performances.

L'objet du projet est d'implanter un mécanisme de communication permettant la communication de messages entre prcoessus sans que la recopie dans l'espace d'adressage du noyeu soit nécessaire. Le mécanisme proposé ici est fortement inspiré des **espaces de messages** introduits dans le système ChorusOS.

Il est demandé de de développer une bibliothèque de fonctions pemettant la communication de processus d'un même système au travers de ce mécanisme.

• **Principe général**

Avec l'approche proposé, un processus désirant transmettre des données sous forme d'un message commencera par demander un tampon en mémoire partagée pour lequel il obtiendra une adresse dans son propre espace d'adressage et il y préparera directement les données à transmettre. Une fois les données prêtes il en avisera le système.

Inversement un processus demandant à recevoir des données via un tel objet de communication recevra directement une adresse de son espace d'adressage qui lui permettra d'accéder immédiatement aux données constituant le message. Une fois les données exploitées, il restituera le *buffer* au système.

Pour réaliser l'implantation du service, on pourra utiliser les mécanismes de mémoire partagéee (segments de mémoire System V ou objets mémoires Posix), des mécanismes de sychronisation (sémaphores System V ou Posix) et des signaux Posix.

• **Présentation détaillée**

↪ On souhaite permettre à des processus d'échanger des **messages**, c'est-à-dire des informations structurées dont on souhaite préserver la structure : une lecture retournera exactement les données transmises lors d'une écriture.

↪ Le concept d'**espace de messages**

Un espace de messages est constitué de :

- ∘ un ensemble de *buffers*, chaque *buffer* étant susceptible de recevoir un message. Les *buffers* sont regroupés en *pools de messages* : les différents *buffers* d'un même *pool* ont la même taille.
- ∘ un **espace de messages** est constitué d'un ensemble de *pools* : le nombre de *pools* constituqnt un espace de messages et les caractéristiques de chacun de ces *pools* (nombre de *buffers* qu'il contient et taille commune de chacun de ces *buffers* sont fixées à la création de cet espace de messages (on pourra les regrouper dans un objet de type `msgPool` transmis en paramètre à la primitive de création d'un espace de messages).
- ∘ un ensemble de **files de messages**. Un message sera déposé dans une file de messages. Le nombre de files de messages sera également une caractéristique fixée lors de la création d'un espace de messages.

↪ **Identification d'un espace de messages**

- ∘ un espace de messages possède un **identifiant externe** de type `MsgSpaceId` : vous pouvez adopter comme tel identifiant ce que vous voulez, par exemple un entier ou une chaîne de caractères. Vous pourrez éventuellement prévoir un identifiant particulier `PRIVATEPOOL` permettant la création d'un espace de nommage anonyme (cet identifiant jouera le rôle de la clé `IPC_PRIVATE` des ipc System V).
- ∘ un espace de messages possède également un **identifiant interne** qui sera récupéré lors de la création ou de l'ouverture d'un espace de messages (voir `msgSpaceOpen`).

On gérera une **table des espaces de messages**, localisée en mémoire partagée afin d'être consultable et modifiable par différents processus, dont chaque entrée aura comme type `msgSpace` : une telle entrée contiendra tout ce qu'il faut pour pouvoir utiliser l'espace de messages correspondant à cette entrée. Toutes les opérations réalisées par un processus sur un espace de messages seront réalisées en utilisant cet identifiant interne : une solution consiste à adopter un pointeur (dans l'espace d'adressage du processus utilisateur) sur l'entrée correspondant à l'espace de messages considéré, ce pointeur ayant été obtenu lors de l'ouverture de l'espace de messages par le processus (ce principe est identique à celui utilisé par la bibliothèque standard identitiant localement les fichiers par un pointeur sur un objet de type `FILE`).

Attention : la table des espaces de messages étant susceptible d'être attachée dans l'espace d'adressage de différents processus, l'utilisation de pointeurs absolus dans ce type est à éviter !.

• **L'interface de programmation** est constituée des fonctions suivantes (les prototypes donnés sont des suggestions et vous pouvez évidemment en adopter d'autres) :

↪ `msgSpace *msgSpaceCreate(`
   `msgSpaceId` *`spaceId`*`,` // identifiant externe de l'espace de messages
   `int` *`queueNb`*`,` // nombre de files de messages
   `int` *`poolNb`*`,` // nombre de *pools* de *buffers*
   `msgPool` *`queueNb`*`[ ]` // caractéristiques des différents *pools*
   `)`

création et ouverture d'un espace de messages d'identification externe donnée. En cas d'erreur, la valeur `NULL` est renvoyée (table des espaces de messages pleine ou identification différente de `PRIVATEPOOL` déjà utilisée) ;

↪ `msgSpace *msgSpaceOpen(msgSpaceId` *`spaceId`*`)`
ouverture d'un espace de messages existant (*spaceId*≠`PRIVATEPOOL`). La valeur `NULL` est renvoyée si l'accès à l'espace de messages est impossible.

↪ `void *msgAllocate(`

```
msgSpace *space, // espace de message concerné
int pool, // numéro du pool où réaliser l'allocation
int taille, // si l'argument précédent est ANYPOOL, taille requise
int option // 0 ou NONBLOCK
)
```

allocation d'un *buffer* soit dans le *pool* spécifié (valeur différente de `ANYPOOL`), soit dans le *pool* de *buffers* non plein dont la taille des *buffers* est la plus proche (de valeur supérieure ou égale) de la taille demandée. La fonction renvoie l'adresse du *buffer* dans l'espace d'adressage du processus. La primitive est bloquante (sauf si l'option `NONBLOCK` est utilisée) dans le cas où il n'y a pas de *buffer* disponible. La valeur `NULL` est renvoyée en cas d'erreur (ou en cas de non-allocation à cause de `NONBLOCK`) ;

↪ `int msgPut(`

```
msgSpace *space, // espace de message concerné
int file, // numéro de file de messages
void *adresse // adresse du buffer
)
```

insére le *buffer* dont l'adresse est donnée (et qui est supposée avoir été obtenue préalablement lors d'un appel à `msgAllocate`) dans la file de numéro spécifié de l'espace de messages d'identité donnée. La fonction renverra `-1` en cas d'erreur et `0` en cas de réussite.
Remarque : un appel réussi de la primitive doit réveiller, s'il en existe, un processus en attente sur une lecture (primitive `msgGet`) sur la file de numéro donné.

↪ `void *msgGet(`

```
msgSpace *space, // espace de message concerné
int file, // numéro de la file de messages concernée
int option // 0 ou NONBLOCK
)
```

renvoie l'adresse du *buffer* en tête de la file de numéro donné dans l'espace de messages spécifié. Le message est effectivement extrait de la file (il ne pourra être relu que si il est réinséré explicitement dans une file). L'appel est bloquant si la file est vide à moins que l'option `NONBLOCK` ne soit utilisée (la valeur retournée sera alors `NULL`).
On pourra éventuellement donner la possibilité d'extraire un message sans spécifier de numéro de file (un argument `ANYFILE` pourra être utilisée, le choix de la file étant alors laissé à votre appréciation).

↪ `int msgFree(`
`msgSpace *`*`space`*`,` // espace de message concerné
`void *`*`adresse`* // adresse du *buffer*
`)`

restitue, au *pool* d'où il a été extrait (lors de l'appel à `msgAllocate`), le *buffer* d'adresse donnée. Cette fonction est typiquement appelée après qu'un processus ait exploité les informations contenues dans un *buffer* d'adresse renvoyée par `msgGet`.

Remarque Un appel réussi de la primitive doit réveiller, s'il en existe, un processus en attente d'allocation d'un *buffer* dans le *pool* spécifié sur la file de numéro donné (appel de la primitive `msgGet`)

↪ `int msgSpaceDelete(msgSpaceId `*`spaceId`*`)`

supprime l'espace de messages d'identification externe donnée.

Remarque : c'est à vous de décider si la suppression d'un espace de messages est immédiate ou différée jusqu'à la fermeture par tous les processus de l'espace de messages.

Dans le premier cas, la suppression d'un espace de messages devra provoquer le réveil de tous les processus bloqués sur une opération sur cet espace de messages.

Dans le deuxième cas, il est nécessaire d'ajouter une primitive `msgSpaceClose` de fermeture et de redéfinir une nouvelle primitive `Exit` exécutée à la fin d'un processus afin qu'il ferme les espaces de messages qu'il a ouverts.

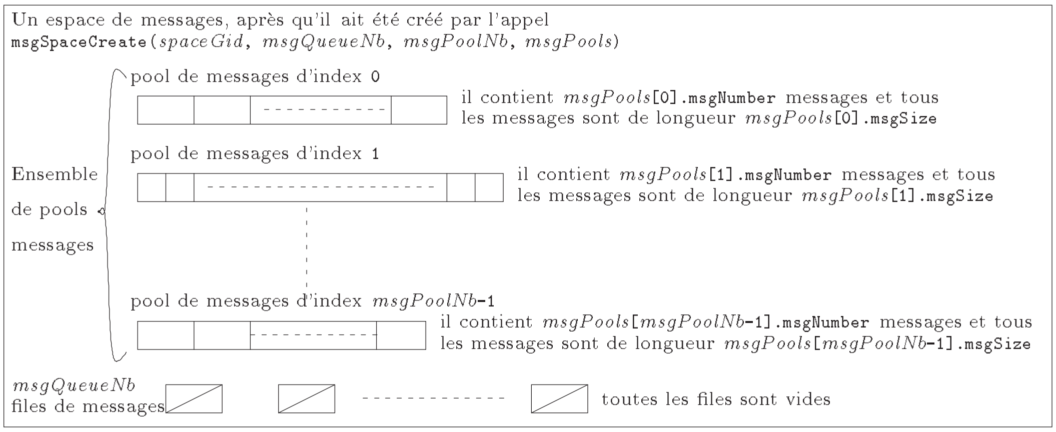
• **Conclusions**

- Bien qu'il soit possible de faire autrement, il pourra s'avérer plus simple de commencer toute application souhaitant utiliser les mécanismes de communication développés ici par un appel à une fonction d'initialisation `msgSpaceInit` réalisant en particulier l'attachement de la table des espaces de messages au processus appelant.

- Par ailleurs, il s'avérera probablement utile de disposer d'un utilitaire (commande) permettant l'examen de l'état d'un espace de messages.

• Quelques schémas explicatifs

↪ *un espace de messages immédiatement après sa création* :



↪ *les effets des primitives de manipulation des messages* :