FAR : TP IPC

1 Files de messages

On veut mettre en place une application composée de deux programmes : le premier, appelé *client*, génère et envoie des requêtes au second, appelé *calculatrice*. La calculatrice résout les requêtes et renvoie les réponses au client. Les requêtes sont des opérations mathématiques simples à effectuer (par exemple : $7 * 5 =?$). Les deux programmes (client et calculatrice) communiquent à l'aide d'une seule **file de messages**.

1. Proposer un protocole de communication (structure(s) des messages) entre le client et la calculatrice. Les requêtes possibles sont les opérations $+, -, *$ et / avec deux opérandes.
2. Réaliser les deux programmes client et calculatrice.
3. Modifier (après avoir gardé une copie) votre solution pour que la calculatrice puisse répondre aux requêtes de différents clients.
4. Optionnel : mettre en œuvre les situations suivantes :
   - Exécuter votre application de manière à constater l'existence de la file hors de la vie des processus engendrés (avec la commande `ipcs`).
   - Déposer des messages dans la file jusqu'à constater le blocage lorsque la file est pleine.
   - Exécuter l'application de manière à constater un blocage lorsque la file est vide.

2 Ensembles de sémaphores - Le rendez-vous

Il est souvent nécessaire de réaliser un rendez-vous entre processus. Pour lancer un jeu à plusieurs joueurs par exemple, pour synchroniser des calculs, etc.

Proposer une solution utilisant un sémaphore et permettant à n processus d'attendre jusqu'à ce que tous les processus soient présents et qu'ils soient arrivés à un point déterminé dit *point de rendez-vous* de leur code, ceci avant de poursuivre leur exécution.

Implémenter cette solution, en affichant la valeur du sémaphore à chaque arrivée d'un processus au point de rendez-vous (utiliser *semctl()*).

3 Ensembles de sémaphores et mémoire partagée

On envisage un traitement parallèle d'une image à effectuer par plusieurs programmes. Chaque programme a un rôle déterminé. Par exemple, un premier programme pourrait faire du lissage, un second, des transformations de couleurs ou de l'anti-crénelage, etc. Enfin, ces traitements doivent se faire dans un ordre bien déterminé.

Pour pouvoir réaliser les différents traitements en parallèle, l'idée est de diviser l'image en sous-ensembles ordonnés de points (pixels) et de permettre à chaque programme de travailler sur un sous-ensemble (appelé zone) différent. Le travail doit alors se faire de la manière suivante :

- chaque programme doit traiter, dans l'ordre, toutes les zones de l'image,
- avec garantie d'exclusivité : pendant qu'un programme travaille sur une zone, aucun autre programme ne doit pouvoir accéder à cette zone en même temps,
- sur chaque zone, les différents traitements doivent se faire dans un ordre déterminé : le programme P_1 doit passer en premier pour traiter cette zone, puis P_2, puis P_3, etc.

1. On se limite d'abord à deux traitements (donc deux programmes) et une seule zone (toute l'image). Proposer une solution permettant un fonctionnement correct. L'image sera représentée par un tableau avec un contenu de votre choix et sera stockée dans un segment de mémoire partagée.
2. Proposer une solution pour plusieurs zones.
3. Passer maintenant à trois traitements (et plus) et proposer une solution.

Remarques :

- les traitements sont à simuler avec la fonction sleep() permettant d'endormir un processus pendant un nombre de secondes passé en paramètre.
- le temps de simulation d'un traitement est à passer en paramètre du programme réalisant ce traitement. Il est souhaitable d'avoir des temps différents pour chaque traitement.
- choisir un temps de traitement suffisamment long (au moins 5 secondes), de sorte à pouvoir observer facilement le comportement à l'exécution, à montrer que l'ordre et l'exclusion mutuelle mis en place fonctionnent et à corriger les éventuelles erreurs.
- il n'est pas nécessaire d'écrire autant de programmes que de traitements : un programme à paramétrer pour les différents traitements peut suffire.