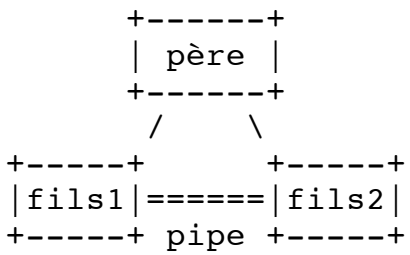
SR03 - Printemps 2014
TD 01
Révisions appels systèmes Unix
Communication par file de messages
IPC

Retour page d'accueil sr03.

Remarque préliminaire : ce TD fait appel à des notions vues dans l'UV SR02, UV dont les connaissances sont un prérequis de SR03.

TD 01 - A - forkpipe.c - Process communiquant par un pipe

Ecrire un programme qui crée DEUX sous-process communiquant par un pipe :



Le premier sous-process (fils1) écrit une suite de chaînes de caractères dans le pipe. Il trouve cette suite de caractères dans un fichier **input.txt**.

Celles-ci sont lues par l'autre sous-process (fils2) et imprimées sur stdout.

Nota: le buffer de lecture du sous-process fils2 sera de longueur 20.

"fils1" va envoyer sur le pipe des "messages" formatés ainsi :

```
[009][input.txt]
[nnn][ligne de nnn caractères]
...
```

"fils2" va lire le pipe, reconstituer les "messages" et reproduire sur stdout la suite des messages encadrés d'un marqueur :

```
reçu>>>input.txt<<<
reçu>>>[Fichier d'entrée du programme TD0 forkpipe.c].<<<
reçu>>><<<
```

...

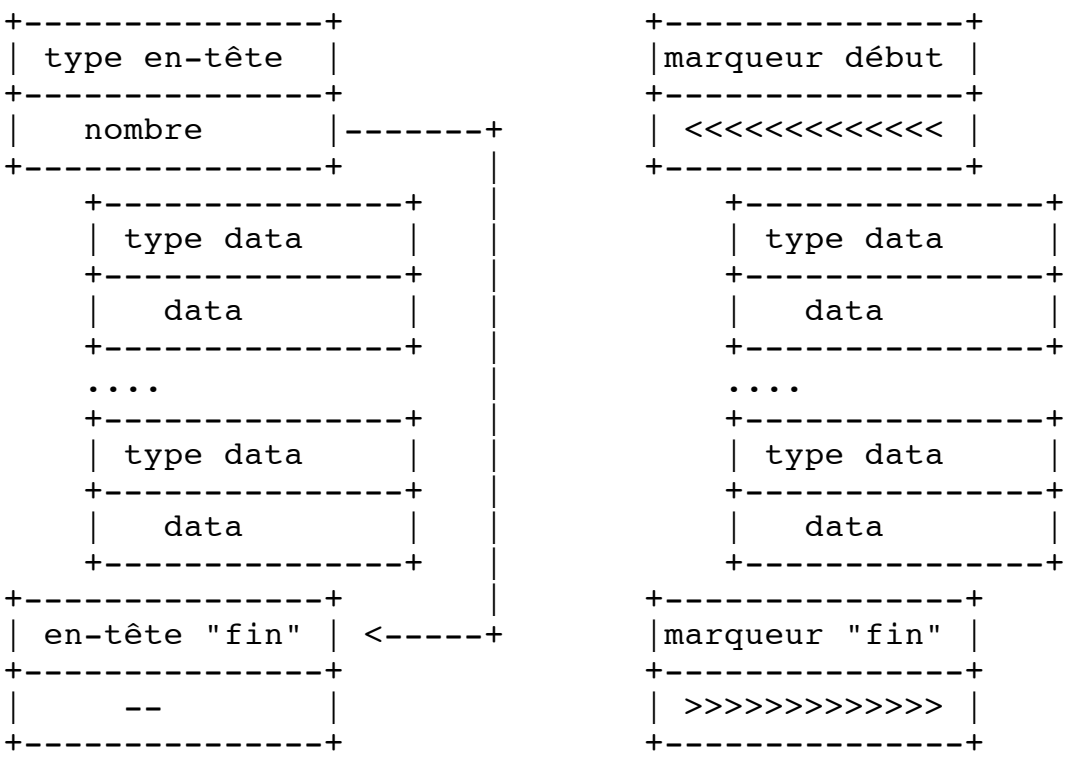
Ainsi, le premier fils envoie PLUSIEURS messages, tous de longueur INCONNUE du process lecteur, et le lecteur doit les reconstituer (séparer le premier message du deuxième, etc.).

Inventez et implantez une solution permettant cette reconstitution des messages.

Gérez les codes retour de lecture sur le pipe pour programmer correctement la terminaison du deuxième process.

Discussion sur le problème des messages dans un flux

```
On a deux grands choix :
 - travailler sur le contenant : encapsulation
   des données utiles dans des données de gestion;
 - travailler sur le contenu : utiliser des
   marqueurs de début et/ou de fin de messages. Ceci
   suppose que les marqueurs ne sont jamais présents
   dans les données, ce que l'on peut assurer par des
   techniques d'échappement.
```




```
L'important est de pouvoir assurer une extraction fiable,
depuis le flux, d'une série de messages de longueur
différentes, ces longueurs n'étant pas connues à l'avance,
mais découvertes au fur et à mesure de la lecture du flux.
```

TD 01 - B - Communication par file de messages IPC

Nous allons construire une simulation d'un dispositif clients-serveur.

Un serveur va conserver et gérer des objets informatiques représentant des stocks d'objets réels.

Des clients vont se connecter au serveur et pourront effectuer des opérations telles que :

- demander la création d'un panier d'achat,
- demander au serveur la liste des types d'objets disponibles,
- pour un type d'objets, demander l'état du stock et le prix,
- demander de prélever "n" objets d'un type et de les mettre dans le panier d'achat,
- demander d'enlever "k" objets du panier,
- vider le panier,
- acheter le contenu du panier.

Le serveur peut être interrogé simultanément par un nombre quelconque (ou presque) de clients.

Nous auront deux niveaux à considérer :

- le niveau de l'échange de données "applicatives" entre les clients et le serveur,
- le niveau du "transport" des données échangées entre les clients et le serveur.

Cette séparation des responsabilités entre deux niveaux (ont dit plutôt deux couches dans le vocabulaire des réseaux) est typique de la construction des applications utilisant des réseaux. On dit que la couche basse fournit un "service de transport" à la couche haute.

Dans ce TD, le mécanisme d'échange entre les clients et le serveur sera l'IPC Unix "File de Messages".

Ce mécanisme sera utilisé de la façon suivante :

- les clients et le serveur utiliseront la même file de messages,
- ni les clients, ni le serveur n'effectueront de lectures inutiles dans la file, grâce à l'utilisation astucieuse du champ "type" des messages posés dans une file de messages Unix.

Il faudra choisir un moyen d'identifier les requêtes des clients et les réponses du serveur vers les clients.

La difficulté principale vient du fait que tous les clients posent leur message de requête dans la même file et que le serveur pose tous les messages de réponse dans cette file également.

On aura donc n+1 process lisant des messages dans la même file : il leur faut un moyen pour ne lire QUE les messages qui leur sont destinés.

On va utiliser une particularité du fonctionnement de la file de messages Unix : on peut faire une lecture sur la file qui ne donne en réponse QUE les messages dont le champ "type" est égal à une valeur donnée.

Il suffit donc d'associer une valeur particulière de "type" pour les messages "requêtes", ainsi

seul le serveur lira ces messages.

Mais il faut aussi associer une valeur de type différente pour chaque client.

La difficulté principale vient du choix d'une méthode permettant d'affecter un muméro de type différent à chaque client.

C'est un problème, classique dans le cadre d'applications distribuées, de nommage.

On peut distinguer deux façons de résoudre ce problème de nommage (ici d'affectation d'un numéro unique à chaque client).

Les méthodes externes ci-dessous sont données à titre d'information, on utilisera dans ce TD la méthode "interne", plus simple dans ce cas.

Méthode "externe" :

On construit un mécanisme indépendant de la file de messages et du serveur. Chaque nouveau client exécute :
- demander un numéro de client unique
- utiliser le serveur pour une ou plusieurs transactions
- libérer le numéro de client.

Le mécanisme d'allocation/distribution des numéros uniques doit être accédé de manière ATOMIQUE (section critique), afin d'assurer que deux clients ne prennent le même numéro.

Ce mécanisme peut être :

- un fichier commun "connu" et accédé avec un verrou exclusif
- un vecteur de bits conservés par le système (par exemple un ensemble de sémaphores sous Unix),
- utiliser simplement le pid du process (tous différents sur la même machine), mais cette méthode ne fonctionne pas si on utilise des threads;
- créer un "serveur de numéros" qui écoute sur un socket et est implémenté comme un serveur concourant (1 seul client connecté à la fois);
- etc.

Méthode "interne" :

On fait faire ce travail par le serveur lui-même : on lui ajoute une requête spéciale "demande de numéro client". Quand il trouve une requête de ce type dans la file, le serveur choisit un numéro libre puis pose dans la file un message "nouveau client" contenant le numéro de client unique.
Le client qui a posé la requête n'a qu'à lire dans la file le premier message disponible de type "nouveau client".

On pourrait objecter "mais comment ce nouveau client sait-il que le message qu'il va lire est

pour lui (et non pas pour un autre nouveau client qui l'aurait demandé presque en même temps) ?".

Eh bien ! Il ne le sait pas, et PEU IMPORTE. Ce qui compte c'est qu'il va être le seul à obtenir ce numéro, puisque en le lisant, il retire le message de la file, personne d'autre ne l'aura. Ce qui compte ici, c'est l'UNICITE, l'ordre importe peu.

Nous disposons maintenant d'un moyen d'échange de messages entre les clients et le serveur.

Les messages posés dans une file de message ont la structure suivante (voir man msgop) :

```
struct msgbuf {
      long mtype;     /* message type, must be > 0 */
      char mtext[1];  /* message data */
};
```

Voir aussi dans "man msgop" l'utilisation du paramètre "msgtyp" de l'appel système msgrcv(2).

L'élément "mtype" des messages va nous permettre de distinguer les différents types de destinataires possibles :

- le serveur,
- les clients non identifiés,
- les clients identifiés.

Ensuite, les clients pourront envoyer et recevoir du serveur des messages de différentes sortes, chacun correspondant à une des opérations prévues au cahier des charges du serveur.

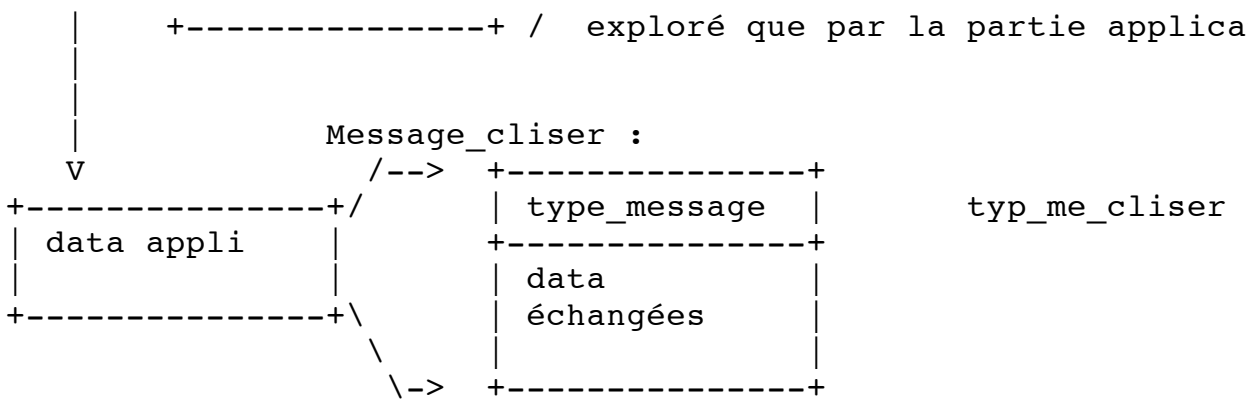
** Attention ** Dans la dernière phrase on a employé les mots "de différentes sortes" pour bien les distinguer des mots "différents types".

En effet, on manipule dans l'application deux "types" imbriqués :

- le type au sens file_mess_IPC (c'est-à-dire au sens du procédé de transport du message) qui va nous permettre de distinguer le destinataire (soit le serveur, soir UN client particulier, soit un client non identifié),
- le type au sens de l'application, par exemple :
  - message de type creer_panier,
  - message de type lister_objets,
  - message de type etat_objet, ...

Un message de type "etat_objet" au sens de l'application sera lui-même encapsulé dans un message de type (client->serveur) au sens de la file de messages.

```
Message_IPC :
        +---------------+  \
        | type_mess_IPC |   } en-tête utilisée par le mécanisme
        +---------------+  /  d'échange (de transport)
        | data appli    | \
   +--- |               |  } corps du message : le contenu n'est
```



En reprenant le cahier des charges du serveur, on peut lister tous les types de messages "application" susceptibles d'être échangés.

```
Echanges serveur <--> clients non-identifiés

   cli_noid ---[demande numéro client]---> serveur     typ_dem_num
   cli_noid <---[fournir numéro client]--- serveur     typ_new_num

Echanges serveur <--> clients identifiés
   cli_nnn ---[]---> serveur
   cli_nnn <---[]--- serveur

   cli_nnn ---[demande liste objets]---> serveur       typ_dem_list
   cli_nnn <---[fournir liste objets]--- serveur       typ_list_obj

   cli_nnn ---[demande état objet]---> serveur         typ_dem_obj
   cli_nnn <---[fournir état objet]--- serveur         typ_etat_obj

   cli_nnn ---[ajout n objets au panier]---> serveur   typ_aj_panier
   cli_nnn <---[fournir état panier]--- serveur        typ_panier

   cli_nnn ---[enlève n objets au panier]---> serveur  typ_en_panier
   cli_nnn <---[fournir état panier]--- serveur        typ_panier

   cli_nnn ---[vider panier]---> serveur               typ_vide_pan
   cli_nnn <---[fournir état panier]--- serveur        typ_panier

   cli_nnn ---[acheter panier]---> serveur             typ_achat
   cli_nnn <---[fournir facture]--- serveur            typ_facture
```

Ensuite, pour chaque type de message serveur <--> clients, il faut définir le contenu du message, c-à-d les data échangées. Par exemple :

```
        { typ_dem_list }
        { typ_list_obj, n, "id_obj1", "id_obj2",... , "id_objn"}
        { typ_dem_obj, "ident_objet"}
        { typ_etat_obj, struct obj }
```

Travail à faire

Ecrire un serveur **sermess.c** et un client **climess.c**

sermess.c va initialiser une table interne d'objets ("iniobj.h"), créer une file de messages, puis se mettre en attente de l'arrivée de requêtes. La programmation de ce TD est grandement facilitée par le fait que la directive **msgrcv(2)** peut être bloquante.

climess.c va demander un numéro de client, récupérer ce numéro, envoyer 2 ou 3 requêtes au serveur sur la file de messages et lire les réponses.

Il n'est pas obligatoire d'implémenter tous les types de requêtes définies dans l'analyse ci-dessus. On fera en fonction du temps disponible.

Pour simplifier la réalisation :

- choisir comme token un fichier local, par exemple "./mon_login_sr03"
- mettre comme drapeaux de création de la file : "IPC_CREAT|IPC_EXCL|0666"
- décider que le serveur traite, au maximum, 2 clients simultanément (NB_MAX_CLT_SIM_ = 2), avant de rejeter proprement les autres
- décider que le serveur traite, au maximum, 4 clients (NB_MAX_CLT = 4), avant de rejeter proprement les autres et de s'arrêter
- décider que le serveur gère 3 types d'objets (NB_MAX_TYP_OBJ = 3)

Ceci permet d'utiliser dans le serveur des structures statiques pour gérer les clients, les paniers et les objets. En effet on s'intéresse dans cette UV aux problèmes de **communication**, pas de gestion de données.
