Université Grenoble Alpes **Année 2023/2024**
L3 MIAGE

Introduction au Système
Planche pour les TP n°4 et 5

NOTE IMPORTANTE.
Les exercices d'entraînement 1 et 2 ne doivent pas prendre plus de la moitié de la séance de TP 4.

La suite de la séance est consacrée au **démarrage du travail sur le projet** : il s'agit aujourd'hui **d'analyser le problème et de commencer à concevoir l'architecture nécessaire**.
Il ne sera pas question de commencer à programmer quoi que ce soit avant d'avoir tout analysé, et bien **rédigé un document de conception** (qui fera partie du rapport à joindre au code).

Exercice 1.

1. Exécutez la commande :

```
who ; pwd ; ls -l
```

et observer attentivement le résultat.

2. Combien de processus fils doit-on créer pour réaliser l'équivalent de l'exécution de cette commande dans un programme C, comment s'exécutent-ils, que fait chacun d'eux ?

Rappel. *pwd est une commande **interne au shell** qui provoque l'affichage de la variable d'environnement correspondante (voir cours p.16). La fonction **getenv** permet de récupérer la valeur d'une variable d'environnement.*

3. Écrire ce programme C.

Exercice 2.

1. Exécuter dans votre shell les commandes :

```
ps ax | grep bash
```

et

```
ps ax | grep bash | wc -l
```

et assurez-vous de comprendre l'objectif de ces commandes.

2. Combien de processus fils doit-on créer pour réaliser l'équivalent de l'exécution de la deuxième commande dans un programme C, que fait chacun d'eux, comment communiquent-ils ?

3. Écrire ce programme C.

Exercice 3 - Projet. *LIRE ENTIEREMENT ET TRES ATTENTIVEMENT*

Cet exercice constitue le projet à rendre (en binôme). Vous devez :

- **en séance le 6 octobre : identifier votre binôme** pour ce projet auprès des encadrants, et **commencer à travailler sur le projet ensemble** (analyse, réflexion). Les binômes seront définis en séance et ne seront plus modifiables.
- **travailler sur ce projet d'ici la séance suivante**,
- et **présenter son état** courant **à la dernière séance**, le 13 octobre. Le binôme devra être au complet et le projet devra être **avancé au moins à l'étape 3** ci-dessous. Vous devrez au moins faire à votre encadrant une démonstration de la version finalisée de l'application à l'étape 3, et de l'état courant de l'étape 4.

Les dernières finitions, et le rapport, pourront être réalisés jusqu'au **20 octobre, date limite**.

Le but de ce projet est de programmer une application qui permet de **stocker et de consulter un dictionnaire de données**. Les données seront de type *chaîne de caractères*.

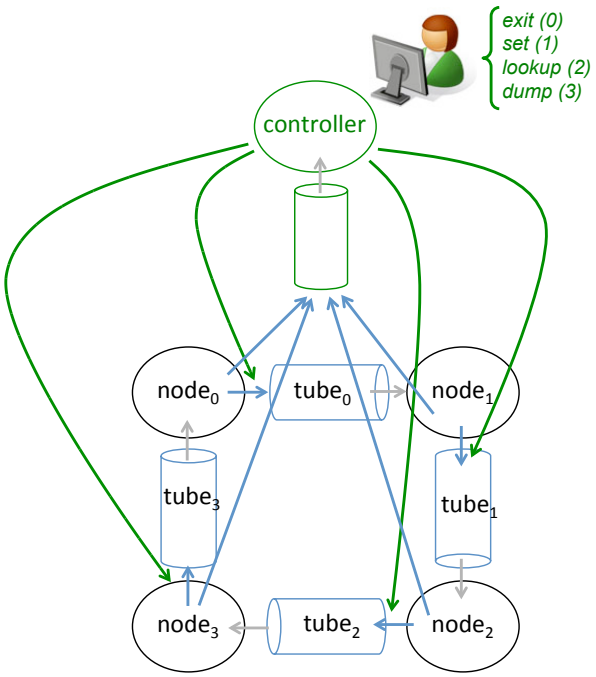
Le dictionnaire sera une collection associative, c'est-à-dire que les données seront stockées grâce à une ***association clé-valeur***, la clé de stockage étant un entier. Plus précisément,

nous utiliserons une liste chaînée de couples clé-valeur telle que définie en Annexe.

Le dictionnaire sera réparti entre **N processus *node***, N ≥ 2. Ce nombre de processus sera choisi au lancement, passé en argument à la commande. Chaque processus *node* numéro i prend en charge les valeurs de clés k = i, i+N, i+2N,…
Ces processus seront les processus fils d'un processus ***controller*** assurant l'interface avec l'utilisateur. Ce *controller* donnera la possibilité à l'utilisateur de saisir des commandes "set" (code commande 1), "lookup" (code commande 2), "dump" (code commande 3), ou "exit" (code commande 0).

- La commande "set" sert à placer une nouvelle donnée dans le dictionnaire, associée à une clé donnée k. Le *controller* va devoir adresser cette requête au bon processus fils (celui qui doit stocker la donnée de clé k). Pour ce faire, il va envoyer la requête au processus *node 0*, qui va la propager aux autres processus *node*, jusqu'à ce que le destinataire effectif soit atteint. C'est lui qui réalisera la commande, et en informera le *controller*.
- La commande "lookup" sert à récupérer dans le dictionnaire la donnée correspondant à une clé donnée k. Ici aussi, le *controller* va devoir adresser la requête au bon processus fils (celui qui stocke la donnée de clé k) : il va envoyer la requête au processus *node 0*, qui va la propager aux autres processus *node*, jusqu'à ce que le destinataire effectif soit atteint. C'est lui qui réalisera la commande, et renverra sa réponse directement au *controller*.
- La commande "dump" vous servira à des fins de débogage essentiellement, elle sert à provoquer l'affichage de la table complète de chaque processus. Ici le *controller* l'enverra directement à tous les processus *node* simultanément, et l'affichage des tables sera réalisé par les processus *node* eux-mêmes (pas par le processus père). Attention, vous devez proposer une solution de synchronisation (à <u>expliquer</u> impérativement dans le rapport) pour gérer les questions que cela va poser : d'une part le *controller* ne devra pas tenter d'interagir avec l'utilisateur tant que tous ces affichages ne sont pas terminés, et d'autre part il serait bon que les affichages des différents processus ne se mélangent pas.
- La commande "exit" est utilisée pour terminer. Ici aussi, le *controller* l'enverra directement à tous les processus *node* simultanément, et il terminera également lui-même.

Nous allons mettre en place une **communication par tubes** telle que décrite sur la figure ci-dessous (sur cette figure, pour 4 processus *node*) :



- les processus *node* sont connectés entre eux par une <u>structure de tubes en anneau</u> (ce qui leur permet de faire circuler une requête, jusqu'à ce qu'elle atteigne le bon processus). Le processus *controller* peut aussi écrire dans chacun de ces tubes.

- tout processus *node* peut écrire dans un unique tube servant à communiquer au *controller* la réponse à la requête reçue.

CE PROJET SERA À RÉALISER ET TESTER **IMPÉRATIVEMENT** SUR LES MACHINES DE L'UFR (machines de TP ou serveur mandelbrot).
LES DÉMOS DEVRONT ÊTRE FAITES SUR CES MACHINES **EXCLUSIVEMENT**

Pour la réalisation de cette application, **procédez progressivement comme suit** (et testez avec soin à chaque étape) :

1. *Analyse*
Lire attentivement ci-dessus, puis rédiger un document de conception dans lequel vous spécifiez avec soin quelles données seront transmises dans quel tube à quel moment (i.e., protocole d'échange, notamment entre *controller* et *nodes*).
Autant que possible, faire au moins valider cela par votre encadrant de TP dans la séance du 6 octobre.

2. *Gérer la création des processus.*
Écrire une application projet_0 qui crée N processus fils du processus initial. Assurez-vous que les N processus ont correctement été créés et qu'ils meurent tous correctement.
Attention, il s'agit ici d'utiliser une *boucle de création de N processus, tous fils* du processus initial (le processus créé au lancement de la commande). Prenez garde à ne pas créer de "petits-fils" (autrement dit, chaque fils créé ne doit pas à son tour créer de processus fils).
Vous pourrez facilement vous assurer de la bonne création des processus en utilisant getpid() et getppid().
Si vous avez des difficultés, vous pourrez ici ne faire créer que 2 processus fils, et finaliser ainsi jusqu'à l'étape 5. Puis revenir et reprendre avec une solution telle que souhaitée (i.e., générique) après finalisation de cette solution à 2 processus fils.

3. *Gérer la communication des processus.*
Reprendre sous la forme d'une application projet_1 qui crée N processus fils du processus initial (comme ci-dessus) et de plus met en place les N tubes tels que décrit plus haut + le tube de communication avec le processus père. Assurez-vous que les N processus arrivent à communiquer (en faisant par exemple circuler un entier dans l'anneau), et que la communication avec le processus père est également possible.
Assurez-vous également que tous les descripteurs inutilisés ont bien été fermés, que tous les processus peuvent mourir, qu'il ne reste pas de zombie !

4. *Se familiariser avec la collection associative.*
Écrire un programme de test testTable qui insère quelques éléments dans une liste, affiche la liste ainsi créée et effectue la recherche d'une valeur existante et d'une valeur absente.

5. *Assemblage des fonctionnalités.*
Vous pourrez alors passer à la réalisation finale du projet.
Écrire une **fonction *controller*** (et d'éventuelles fonctions auxiliaires) réalisant le travail qui sera effectué par le processus *controller* : saisie de la commande utilisateur et interaction avec les processus *node* pour satisfaire cette commande.

Attention, vous devez ici réfléchir au protocole d'échange :
- le controller doit envoyer une information claire aux processus fils (commande + paramètres éventuels) ;
- le *controller* doit dans tous les cas récupérer une information claire (quelqu'un a pris en compte la requête, et a pu la satisfaire ou a échoué dans cette tentative).

Nous vous suggérons de définir un format de message spécifique pour tous ces échanges (à expliquer dans le rapport). Le plus simple est probablement d'utiliser systématiquement des chaînes de caractères avec une syntaxe à définir.

Écrire une **fonction *node*** (et d'éventuelles fonctions auxiliaires) réalisant le travail qui sera effectué par chaque processus *node* : réception d'une requête, et satisfaction de cette requête

ou transmission au suivant.

Voici un exemple de session possible :

```
$ ./monprojet 5
Saisir commande (0 = exit, 1 = set, 2 = lookup, 3 = dump): 1
Saisir la cle (decimal number): 12
Saisir la valeur (chaine de caracteres, max 128 chars): Douze
Saisir commande (0 = exit, 1 = set, 2 = lookup, 3 = dump): 2
Saisir la cle (decimal number): 12
Valeur trouvee = Douze
Saisir commande (0 = exit, 1 = set, 2 = lookup, 3 = dump): 2
Saisir la cle (decimal number): 3
Pas de valeur trouvee
Saisir commande (0 = exit, 1 = set, 2 = lookup, 3 = dump): 3
process 3465 :
     12 Douze
Saisir commande (0 = exit, 1 = set, 2 = lookup, 3 = dump): 0
bye bye!
```

Note. Pour tester le comportement de votre application, noter que l'option `l` de la commande `ps` permet un affichage long (avec notamment l'identifiant du numéro de processus père de chaque processus, PPID), et que la commande `pgrep -P pid` affiche la liste des fils du processus de numéro pid.

Réalisation et remise. A LIRE ATTENTIVEMENT

Vous devrez fournir un fichier .tar.gz contenant :

- tous vos **sources** *correctement organisés et commentés* (les noms des auteurs devront figurer dans tous les sources),
- un **rapport** de présentation de votre travail au format **pdf** (de la taille d'environ une dizaine de pages, en 10 ou 11 points simple espacement) qui présente clairement :
  - l'objet du projet, et votre réflexion sur la conception de cette application,
  - vos choix de conception (notamment votre définition du protocole d'échange et tout ce qui concerne communication et synchronisation des processus) et de structures de données,
  - ainsi que la description de vos réalisations (voire limitations).

  Il permettra aussi de savoir comment tester votre projet : expliquer comment le compiler et l'exécuter.

  NB. Ce rapport n'est pas la concaténation des fonctions de votre projet !
- et un **Makefile** (même élémentaire) pour la compilation de vos sources.

*NB. Le fichier remis **doit** permettre d'identifier clairement les noms des étudiants du binôme.*
*Son identifiant devra être de la forme **nom1_nom2.tar.gz** où nom1 et nom2 sont les noms de famille des étudiants.*
*Attention, il devra être produit à partir d'un **répertoire lui-même appelé nom1_nom2**.*
→ *Placer votre hiérarchie de fichiers constituant le projet dans un répertoire nom1_nom2*
Faites tar cvf nom1_nom2.tar nom1_nom2
puis gzip -9 nom1_nom2.tar

Remise du projet :
Un seul fichier .tar.gz remis via le moodle avant le 20 Octobre 2023 soir, extrême limite.

Tout projet ne respectant pas **toutes** les consignes données ci-dessus (format, dénomination, organisation, date,…), ne sera **pas** pris en compte.

Annexe

Fichier table.h

```
#ifndef TABLE_H
#define TABLE_H

typedef struct _Table_entry {
   int key;                        // cle
   char* val;                      // valeur
   struct _Table_entry *next;      // suivant
} Table_entry;

typedef Table_entry *PTable_entry;

char* lookup(PTable_entry table, int k);
void store(PTable_entry *table, int k, char v[]);
void display(PTable_entry table);

#endif
```

Fichier table.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "table.h"

// Stocke dans table la valeur v sous la cle k
void store(PTable_entry *table, int k, char v[]) {
    PTable_entry te = (PTable_entry) malloc(sizeof(Table_entry));
    te->key = k;
    te->val = (char*) malloc(strlen(v) + 1);
    strcpy(te->val, v);
    te->next = *table;
    *table = te;
}

// Retourne la valeur se trouvant dans table sous la cle k, ou NULL si la
// cle n'existe pas
char* lookup(PTable_entry table, int k) {
    PTable_entry pe = table;
    while (pe != NULL && pe->key != k) {
        pe = pe->next;
    }
    if (pe != NULL)
        return pe->val;
    else
        return NULL;
}

// Affiche le contenu de la table
void display(PTable_entry table) {
    PTable_entry pe = table;
    while (pe != NULL) {
        printf("%d : %s\n", pe->key, pe->val);
        pe = pe->next;
    }
}
```