Université Grenoble Alpes **Année 2025/2026**
L3 MIAGE

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

*Les exercices 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 mini-projet** (exercice 3).*

Exercice 1.

1. Rappel : de façon générale, l'ordre d'exécution et de mort d'un processus père et de ses fils est-il prévisible ?

2. Le **;** peut être utilisé pour signifier l'exécution d'une *séquence* de commandes (vous pouvez trouver plus de détails par exemple ici https://docstore.mik.ua/orelly/unix3/upt/ch28_16.htm).
Saisissez dans le shell :

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

et observer attentivement le résultat.

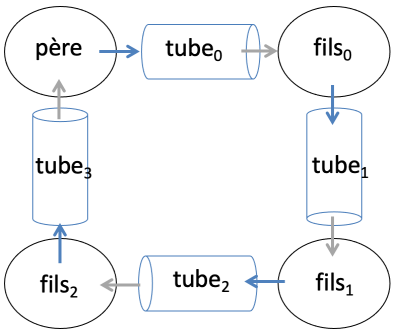
3. Analyser ce que devra faire un **programme C** pour réaliser l'équivalent de l'exécution de la ligne de commande de la question 2 : combien de processus fils devra-t-on créer ? que fera chacun d'eux ? et comment garantirons-nous la séquentialité ?

Rappel. *pwd est une commande **interne au shell** qui provoque **l'affichage de la variable d'environnement** correspondante (voir cours p.16-17). La fonction **getenv** permet de récupérer la valeur d'une variable d'environnement.*

4. Écrire ce programme C qui réalise l'équivalent de l'exécution de cette ligne de commande de la question 2.

Exercice 2.

Nous allons écrire un programme dans lequel NB processus (1 processus père et ses NB-1 processus fils) communiquent via NB tubes comme illustré ci-dessous pour NB=4 :



La valeur de NB sera passée en paramètre à la commande.
Le programme utilisera une variable globale *tableau_tubes* pour stocker un tableau (dynamiquement alloué) de NB tubes.

1. Ecrire une fonction

```
void creer_tubes(int n);
```

qui (alloue et) crée le tableau de *n* tubes *tableau_tubes*

2. Ecrire une fonction

```
void fermer_tubes(int i, int n);
```

qui se charge de fermer (dans le tableau de *n* tubes *tableau_tubes*) toutes les extrémités de

tubes qui seront non utilisées par le fils numéro *i*

3. L'application qu'on souhaite écrire vise à faire passer un entier K dans cet anneau de processus, il sera envoyé par le processus père et devra être récupéré par tous les processus fils :
Le processus père envoie K dans *tableau_tubes*[0] et attend qu'il revienne jusqu'à lui par le dernier tube. Chaque processus fils récupère K dans son tube d'entrée et le fait passer au fils suivant. Le dernier fils repasse K au processus père. Lorsque K est ainsi revenu et que tous les processus fils sont morts, le processus père meurt à son tour.

Ecrire une fonction

```
void passer_jeton(int i, int n);
```

qui sera exécutée par le fils numéro i pour récupérer l'entier dans son tube d'entrée et le passer au fils suivant.

Ecrire une fonction

```
void lancer_jeton(int n);
```

qui sera exécutée par le processus père pour envoyer un entier K dans *tableau_tubes*[0], attendre qu'il revienne jusqu'à lui par le dernier tube, et terminer correctement l'exécution.

Ecrire la fonction *main* pour créer les tubes et les processus fils et réaliser le travail décrit plus haut.

NB. Pour transmettre une simple donnée dans un tube (caractère, entier, nombre réel...), il suffit de **lire/écrire à l'adresse de la donnée elle-même** (pas besoin d'un "buffer" spécifique).

Exercice 3 - Projet.

Cet exercice constitue le mini-projet (à faire en binôme).

- **en séance le 10 octobre :** commencer à travailler sur le projet (analyse, réflexion, début de codage).
- travailler sur ce projet d'ici la séance suivante,
- **en séance le 17 octobre :** présenter l'état courant (idéalement, terminé !), poser des questions,...

Une **version finale** pourra être remise dans le moodle jusqu'au **24 octobre** soir. Pour tout code propre, lisible, très bien commenté et expliqué, un retour sera fait.

Le but de ce projet est de *programmer un mini-shell avec quelques fonctionnalités de base*.
A la fin du cours (page 45), on vous donne la structure extrêmement simplifiée d'un micro-shell ne pouvant prendre en charge, pour les commandes externes, qu'une seule commande en foreground, sans redirections et sans enchaînements. Vous partirez d'un squelette de ce style comme base de travail.

Votre programme devra inclure une *fonction permettant* ***d'analyser*** *(découper)* ***la ligne de commande*** (fonction *Lire_commande*, qui analyse la ligne *cmd*) afin d'y reconnaître la présence de paramètres et options, voire de plusieurs commandes.
Par exemple, pour :

```
head -12 fic.txt | tail -2
```

elle reconnait 2 commandes séparées par un tube, et
commande 1 est `head` avec présence des arguments/options `-12 fic.txt`
commande 2 est `tail` avec présence des arguments/options `-2`

Note. Afin de simplifier le traitement, cette fonction pourra faire l'hypothèse que tous les éléments de la ligne de commande sont séparés par des espaces, et simplement utiliser la fonction "strtok" pour découper la ligne.
(voir par exemple https://www.delftstack.com/fr/howto/c/strtok-in-c/)

Concernant les ***fonctionnalités du mini-shell***, tâchez de prendre en compte au minimum les éléments suivants :

• quelques **commandes internes** (par exemple cd, getenv et setenv) :
- la commande *cd* pourra être utilisée avec ou sans paramètre,
- la commande *getenv nom-de-variable* permettra d'obtenir la valeur associée à la variable d'environnement passée en paramètre,
- la commande *setenv* pourra être utilisée sans paramètre (affichage des noms et valeurs de toutes les variables d'environnement), ou sous la forme *setenv nom-de-variable valeur* pour positionner la valeur de la variable d'environnement.
  Voir plus d'informations sur la gestion des variables d'environnement ici :
  http://supertos.free.fr/supertos.php?page=950

• **redirections** d'E/S sur fichiers (< et >)

• communication entre commandes par **tubes** (au moins pour 2 commandes, et si possible pour un nombre quelconque). Pour gérer un nombre quelconque de commandes/tubes, il vous faudra inclure la création des processus fils dans une boucle, et utiliser un tableau de tubes.

Le shell devra en outre ***traiter la réception de*** ^C. En effet, l'utilisation de ^C dans le shell ne doit pas provoquer la **terminaison** du shell lui-même, mais de la (des) **commande(s) en cours** d'exécution (en foreground). Pour ce faire, le shell doit prévoir de *se dérouter sur un* ***handler*** *du signal correspondant*, dans lequel il renvoie le signal à tous ses processus fils (en foreground). Il pourra également afficher un message pour indiquer que le signal a été reçu.

Enfin, il sera aussi souhaitable de traiter l'exécution de commandes en background (**&**).
Dans ce cas, le processus fils ne doit pas être sensible à ^C, et le processus père ne doit pas attendre la terminaison du processus fils.
De ce fait votre code ne peut plus tout simplement inclure une attente de la mort des processus avant de re-afficher le prompt (comme c'est le cas pour la seule gestion de processus en foreground), cela afin d'éviter qu'une telle attente s'applique à un processus en background.
Le code devra ici être modifié pour ne prendre en compte la mort de tous les fils (background et foreground) que lorsqu'elle aura déjà eu lieu, via un *handler* pour le signal SIGCHLD. De plus, afin de gérer le cas où plusieurs processus fils pourraient mourir simultanément, on pourra prévoir une boucle, mais en utilisant *waitpid*, dont l'option WNOHANG permet au processus père de ne pas rester bloqué s'il n'y a plus de fils à attendre. Voir plus d'explications par exemple ici :
http://web.stanford.edu/class/archive/cs/cs110/cs110.1204/static/lectures/07-Signals/lecture-07-signals.pdf, pages 3-4.
Enfin, afin de ne revenir au prompt qu'à la mort de tous les processus en foreground, on réalisera une boucle d'attente active dont on ne sortira que lorsque le nombre de processus en foreground sera 0 (nombre à mettre à jour dans le *handler*).

==========

Nous vous recommandons de procéder progressivement comme suit :

1. Lire attentivement ci-dessus, saisir le squelette de shell donné dans le cours (assurez-vous de le comprendre parfaitement !), ajouter une définition pour la fonction *Lire_commande* qui découpe la commande *cmd* et place les différentes chaînes de caractères présentes sur la ligne dans le tableau *com*. Vérifier le bon fonctionnement de cette version de base.

2. Enrichir ce programme de base avec des constructions simples (quelques commandes internes, redirections d'E/S) et vérifier le bon fonctionnement de cette version.
Commenter soigneusement votre code, en particulier en ce qui concerne les choix techniques de votre mise en œuvre.

3. Une troisième version sera consacrée à la prise en compte des communications par tubes. A réaliser avec beaucoup de soin et, si possible, permettre l'utilisation de tubes entre un

nombre quelconque de commandes.
Commenter soigneusement pour expliciter les principes et solutions techniques utilisés.

4. Enfin prendre en compte les autres aspects mentionnés ci-dessus, et finaliser votre projet. Si votre application est modulaire, prévoir un *makefile*, même élémentaire.

Mentionner explicitement (via un Readme, ou des commentaires dans le code), ce qui a été implémenté et ce qui ne l'a pas été.

Tests

A titre indicatif, voici quelques exemples de commandes sur lesquelles votre mini-shell pourra être testé :

```
ls -la ..
wc < monshell.c > resultat
head -12 fic.txt | tail -2
getenv MANPATH
ps aux | more
ls | grep A | wc -l > res.txt
wc -c fic.txt &
ls | grep X  &
```

Remise

Remettre **un seul fichier nom1_nom2.tar.gz** où nom1 et nom2 sont les noms de famille des étudiants, qui sera produit à partir d'un répertoire lui-même appelé nom1_nom2 :

Placer vos fichiers constituant le projet dans un répertoire de nom nom1_nom2
Faites
    tar cvf nom1_nom2.tar nom1_nom2
puis
    gzip -9 nom1_nom2.tar

Les codes propres, lisibles, bien découpés en fonctions, bien indentés, très bien commentés et expliqués (et qui ne sont pas juste le produit d'une IA générative !) feront l'objet d'un retour (commentaires sur les points forts et faibles de la solution).