Gestion des processus

Le but de ce projet est d'écrire un shell simplifié permettant de lancer des processus en ligne de commande dans un environnement Unix.

1 Rendu

Cette section décrit très succinctement le contenu attendu dans ce projet. Cette section constitue une base de travail et ne se substitue pas aux consignes données en cours par votre enseignant.

Pour ce projet, vous devrez mettre en avant et vous serez évalués sur vos capacités à :

— comprendre du code avancé fourni ;
— restituer les différents éléments pratiques et théoriques que vous avez vus en cours ;
— mettre en œuvre ces éléments dans un code efficace, ¡¡ propre ¿¿ et documenté ;
— être curieux face à des techniques et des sujets nouveaux.

Deux rendus sont attendus à l'issu de ce projet : un rapport et le code source réalisé pour mener à bien les expériences.

Le rapport

Cette section décrit très succinctement le contenu attendu du rapport de projet.

Cette section constitue une base de travail et ne se substitue pas aux consignes données en cours par votre enseignant.

Considérations générales

Sachez qu'une part non négligeable de votre note sera liée à la forme du rapport, au discours que vous y développerez et au respect des présentes consignes. Ces points sont plus importants que les résultats bruts que vous pourriez placer dans le rapport.

Voici quelques points généraux que vous devez prendre en compte [1] :

— (pénalité : note = 0 – conseil de discipline) Votre rapport est un travail de binôme **uniquement**. Tout partage d'éléments significatifs avec un autre binôme sera fortement sanctionné (pour les deux groupes concernés sans distinction). Il en sera de même quant à l'utilisation abusive de ressources externes (plagiat Internet ou d'ouvrages).
— (pénalité : -1 par « remplissage ») Ce rapport doit être présenté comme un rapport technique. Avant d'inclure un élément dans votre rapport, demandez-vous systématiquement s'il aide le lecteur à **comprendre les mécanismes** que vous observez ou expliquez. En particulier, évitez les captures d'écrans inutiles (une page pour afficher une ligne de commande sans intérêt).
— (pénalité : -4) Supposez que le lecteur de votre rapport n'a pas accès à ce sujet et ne le connaît pas. Les sections précédentes ne sont pas des questions et sont là pour vous guider dans les différents points à aborder et les sujets à traiter. Il sera jugé **inadmissible** que votre rapport comporte des passages tels que : « Pour la question 3.5 1, nous avons utilisé une boucle for ».
— (pénalité : -2) Respectez au maximum la langue française pour être compréhensible : relisez-vous et faîtes-vous relire ! De même, vous êtes de futurs ingénieurs donc utilisez un vocabulaire précis et technique : un programme ne « plante » pas et n'a pas de « bug » qui le font « crasher » et un ordinateur ne « rame » pas !

Structure du rendu (pénalité : -5)

En plus des points précédents, un rapport technique, comme tout document, doit s ?articuler autour de trois parties :

— l ?introduction qui présente le contexte, la problématique et le contenu du document ;
— le développement qui correspond au coeur du rapport ;

1. Le non respect de ces consignes sera sanctionné. Les pénalités indiquées correspondent au nombre de points retirés de la note de projet. Ce ne sont que des valeurs indicatives et elles ne sont pas exhaustives.

— la conclusion qui synthétise le contenu du document et présente des perspectives ouvertes par les travaux présentés.

En particulier, l'introduction doit présenter le contexte **technique** de votre rapport (et pas que vous êtes à l'ENSSAT et que vous êtes en projet de systèmes d'exploitation). Il faut ensuite définir la problématique de votre étude, le but recherché et enfin annoncer le plan du rapport qui doit être lié à la structure de votre discours.

Quant à la conclusion, elle doit comporter une synthèse « globale » (c'est un pléonasme volontaire) du contenu du rapport et porter une analyse générale sur ce qui a été fait et observé. Elle doit se terminer en présentant des perspectives ouvertes par votre travail (impacts, points à confirmer, développement complémentaires possibles, ...). Votre avis sur ce que vous avez appris à titre personnel, votre satisfaction ou insatisfaction personnelle à faire ce projet ou à suivre ce cours, vos difficultés d'organisation personnelle (« le chat a mangé mon ordinateur ») ou toute autre considération sur l'âge du capitaine et le sens du vent sont superflus dans un rapport technique.

Forme du rapport (pénalité : note/2)

La forme de votre rapport est importante. Comme indiqué précédemment, les règles de grammaire et de typographie de la langue dans laquelle est écrit le rapport doivent être respectées. Le texte doit être justifié. L ?utilisation des outils LaTeX est un bon moyen d'assurer uniformité et respect de la typographie (mais n'est pas obligatoire).

Quels que soient les moyens utilisés pour produire votre document, vous devez fournir un rendu lisible sur un maximum de supports. Le fichier contenant votre rapport devra donc **obligatoirement** être au format PDF. Le fichier sera nommé comme suit : « SE_IMR_NomBinome1_NomBinome2.pdf » où NomBinome1 et NomBinome2 correspondent aux noms des étudiants par ordre alphabétique. Malgré la présence de vos noms dans le nom du fichier, il est nécessaire de les inclure **aussi** dans la première page du rapport !

Le code (pénalité : note/2)

L'ensemble du code utilisé devra être fourni au sein d'un unique fichier compressé et nommé sur le modèle suivant : « SE_IMR_NomBinome1_NomBinome2.zip » où NomBinome1 et NomBinome2 correspondent aux noms des étudiants par ordre alphabétique. Ce code devra être organisé, commenté, correctement indenté et compilable. La présence d'un fichier README décrivant comment utiliser vos sources est également souhaitée. Ces consignes sont primordiales et leur non respect impactera lourdement la notation finale.

Organisation

Ce projet est à réaliser par binôme. Le rapport sera à rendre pour fin-janvier (la date exacte sera précisée ultérieurement) et devra contenir une description des modifications apportées au code fourni pour répondre à toutes les spécifications. Il n'y a pas d'heures encadrées prévues pour ce projet.

Vous devrez annoncer à votre encadrant de projet la constitution des binômes par e-mail avant les vacances d'hiver. En cas de problème avec votre binôme (absence, non participation au travail), vous devrez prévenir votre encadrant le plus rapidement possible, et pas l'indiquer par une note dans votre rapport, afin qu'il puisse réagir en conséquence.

2 Décomposition en modules

L'ensemble des fichiers fournis dans l'archive `.tar` propose une séparation des fonctions en plusieurs modules avec un regroupement en rapport avec le rôle des fonctions :

1. Le fichier `shell.c` devra contenir le code de la fonction `main`. Son rôle est de mettre en place toutes les initialisations nécessaires, puis de lancer la boucle principale de lecture des lignes de commandes.

2. Le module `parse` (fichiers `.c` et `.h`) contient les fonctions réalisant le parcours de la ligne de commande, et l'initialisation d'une structure interne pour représenter et ajouter de la sémantique à la ligne de commande. Cette structure sera utilisée lors des appels aux autres fonctions du projet.
3. Le module `execution` (fichiers `.c` et `.h`) contient les fonctions permettant de parcourir la ligne de commande `execution_ligne_cmd` et de lancer les commandes individuelles `execution_cmd`.
4. Le module `commandes_internes` (fichiers `.c` et `.h`) contient le code des fonctions appelées lors d'un appel à une fonction interne du shell dans la ligne de commande.
5. De façon similaire, le module `commandes_externes` (fichiers `.c` et `.h`) contient le code de la fonction permettant d'exécuter une commande non reconnue comme une commande interne.
6. Le module `divers` (fichiers `.c` et `.h`) contient le code de plusieurs fonctions utilitaires. Son rôle est de factoriser les parties communes des autres modules.

3 Traitement de la ligne de commandes

Spécification de la ligne de commande

Pour des raisons de simplification du projet, on supposera que seuls les traitements suivants seront traités par le shell :

— redirection de stdout (`>` et `>>`)
— redirection de stdin (`<`)
— transfert de flux (`|`)
— exécution en séquence (`;`)
— exécution conditionnelle (`&&` et `||`)
— exécution en arrière-plan (`&`)
— substitution de variable (`%`)
— interprétation des commentaires (`#`)

Également dans une optique de simplification, on supposera que tous les éléments (en dehors de la gestion des variables d'environnement) sont séparés par des espaces. Par exemple la commande `echo␣aaabb␣b␣c;␣echo␣a␣&&echo␣b` est syntaxiquement valide, mais on utilisera l'écriture avec d'avantage d'espaces `echo␣aaabb␣b␣c␣;␣echo␣a␣&&␣echo␣b`.

Lecture de la ligne de commande

Pour lire la ligne de commande, le shell utilise la fonction `fgets` de la *libc*. La boucle de lecture utilise un tableau statique (`char cmd_line[CHAINE_MAX]`), et lit une à une plusieurs lignes dans l'entrée standard avec l'appel :

```
CHAR cmd_line[CHAINE_MAX];
while (fgets(cmd_line, sizeof(cmd_line), stdin))
{
  /* traitement de la ligne stockee dans cmd_line
   * ...
   */
}
```

Le retour de `fgets` rend faux le test pour continuer la boucle dès que le flux d'entrée est fermé.

Analyse de la ligne de commande

Le rôle de cette analyse est d'initialiser une structure de donnée interne. La description de cette structure interne est la suivante :

```
typedef struct {
  int nb_arg;
```

```
  char ligne_cmd[ARG_MAX][CHAINE_MAX];
  t_symbole modificateur[ARG_MAX];
  char entree[MAX_PATH];
  char sortie[MAX_PATH];
} parse_info;
```

Le rôle de la fonction `parse` est de découper la ligne de commande (passée en paramètre via la variable `ligne_cmd`) en mots. La fonction *POSIX* utilisée pour réaliser ce découpage est la fonction `strtok`. Cette fonction s'utilise en deux temps :

```
char *tok;
tok = strtok(ligne_cmd, SEPARATEUR);
```

Lors de ce premier appel, une structure de donnée interne à la librairie C est initialisée avec la chaîne contenue dans la variable `ligne_cmd`, et un ensemble de séparateurs de mots est déclaré via la macro `SEPARATEUR` (dans le cadre de ce projet, cette macro sera initialisée avec ␣`\n`, c'est-à-dire, les caractères espace et retour à la ligne seront considérés comme des délimiteurs de mots).

Le retour de ce premier appel donne un pointeur sur le premier mot de la chaîne.

Les appels suivants sont de la forme :

```
tok = strtok(NULL, SEPARATEUR);
```

Le premier argument à `NULL` indique de garder la chaîne d'origine, et renvoie un pointeur sur le mot suivant.

Attention, lors de ces appels, la chaîne stockée dans la variable `ligne_cmd` *est modifiée*, des caractères de fin de chaîne `\0` remplaçant les caractères de la liste de séparateurs présents dans la chaîne d'origine.

Pour chaque mot (`tok`), l'utilisation des macro `EST_EGAL` et `COMMENCE_PAR` permet de reconnaître un caractère spécial, et ainsi remplir le tableau `modificateur` de la structure de donnée `parse_info`.

On peut noter qu'il est inutile de garder l'information d'un mot égal à un caractère spécial, l'information contenue dans le tableau `modificateur` étant suffisante. À cette fin, le traitement d'un caractère spécial se résume à :

```
if (EST_EGAL(tok, "<"))
{
  info->modificateur[i] = REDIRECTION_ENTREE;
}
```

Le cas de traitement d'une variable est différent, si on considère une substituion immédiate, on écrira :

```
if (COMMENCE_PAR(tok, "%"))
{
  lire_variable(&(tok[1]), info->ligne_cmd[i], sizeof(info->ligne_cmd[i]));
  i++;
}
```

La description de la fonction `lire_variable` est faite en fin de document, le premier paramètre est une écriture permettant d'accéder au nom de la variable sans le premier caractère (`%` marquant une substitution du nom de variable à réaliser).

Dans le cas où le mot (`tok`) n'est ni un caractère spécial marquant une action à réaliser par le shell, ni un nom de variable à substituer, il suffit de le copier dans le tableau `info->ligne_cmd` de la structure de description de la ligne de commande :

```
else
{
  strcpy(info->ligne_cmd[i], tok);
  i++;
}
```

Après la phase d'analyse, le tableau de chaînes de la structure interne (`info->ligne_cmd`) contient tous les mots de la chaîne de caractères oiginale emputé des caractères spéciaux reconnus par le shell (`>`, `|`, `&`, ...). À chaque caractère spécial est associé un symbole du type énuméré `t_symbole` :

```
typedef enum{AUTRE=0,
  REDIRECTION_ENTREE,
  REDIRECTION_SORTIE_AJOUT,
  REDIRECTION_SORTIE,
  EXECUTION,
  EXECUTION_SI,
  EXECUTION_SINON,
  ARRIERE_PLAN,
  VARIABLE,
  COMMENTAIRE}
t_symbole;
```

L'ensemble des actions décrites précédement est déjà écrit dans le corps de la fonction `parse` du fichier `parse.c`.

Exécution de la ligne de commande

La fonction `execution_ligne_cmd` effectue un parcours de la ligne de commande afin d'isoler chacune des commandes à exécuter.

Il s'agit d'une double boucle, la boucle la plus externe permettant de parcourir la ligne de commande afin de traiter toutes les commandes, la boucle la plus interne permettant de délimiter une commande à exécuter. Dans la progression de la boucle interne, les champs `entree` et `sortie` seront éventuellement complétés si après l'étape d'analyse de la ligne, le tableau `modificateur` contient les informations `REDIRECTION_*`.

Une fois isolé, et après l'exécution de la commande (via un appel à `execution_cmd` décrit plus loin), le mécanisme d'enchaînement conditionnel des commandes est utilisé pour permettre l'exécution (si nécessaire) de la commande suivante. Le code correspondant à cette étape fera l'objet d'une des étapes du travail à produire.

L'exécution parallèle avec un tube `|` sera également à écrire. Contrairement à une exécution séquentielle, il ne s'agit pas de rechercher une commande unique, mais de rechercher un ensemble de commandes [2] et mettre en place les mécanismes de redirection permettant la communication entre les processus.

Exécution d'une seule commande

La fonction `execution_cmd` est complètement écrite, et se résume à une suite de comparaison afin de déterminer la fonction à lancer pour réaliser l'action demandée au shell :

```
if (EST_EGAL(info->ligne_cmd[debut], "echo"))
{
  return ActionECHO (info, debut, nb_arg);
} else if (...) {
  ...
```

Les différents mécanismes de redirection seront mis en places plus tard.

Commandes internes

Une partie des commandes internes sont intégralement écrites :

2. Pour des raisons de simplification on pourra considérer l'utilisation du tube pour lier au maximum deux commandes...

1. La commande `ActionECHO` est une version simplifiée du processus `echo` présent sur un système unix. Cette commande interne n'a pour but que d'illustrer simplement un mécanisme de redirection de la sortie pour une commande interne. Elle permettra également de tester certaines fonctionnalités du shell en l'absence d'un mécanisme d'éxécution de commande externe.
2. La commande `ActionCD` est une version simplifiée de la commande interne `cd` d'un shel unix :
   — `cd` déplace le répertoire courant vers le répertoire *USERPROFILE* de l'utilisateur (appel à la fonction `lire_variable()`, ainsi tant que cette fonction n'est pas écrite, cette action n'a pas l'effet souhaité).
   — `cd arg` déplace le répertoire courant vers le répertoire *arg*. L'argument `arg` peut être un chemin relatif ou absolu.
   — Le seul cas où la fonction `cd` aura plusieurs arguments correspond au cas où le chemin de destination contient des espaces. Ainsi `cd arg1 arg2...` déplace le répertoire courant vers le répertoire composé de la concaténation (avec insertion d'un espace entre chaque mot) de tous les arguments donnés en paramètre.
3. La commande `ActionSET` a pour rôle de permettre l'affectation d'une valeur à une variable d'environnement, ou la suppression d'une valeur à cette variable. Cette commande se base sur la fonction `ecrire_variable` qui reste à compléter.
4. Les commandes `ActionLS` et `ActionPS` sont définies et peuvent êtres écrites en tant que commande interne du shell. Cependant, quand un mécanisme d'exécution de commande externe est mis en place et pour éviter une forte complexité du code afin de prendre en compte les différentes redirection de flux possible, il est préférable de considérer l'exécution d'un programme externe [3].

Commandes externes

L'exécution de commandes externes se fait en utilisant une primitive de la famille `exec` dans un nouveau processus (à l'aide de `fork`). L'écriture du mécanisme d'exécution externe est intégralement à écrire. Les points importants porteront sur la redirection des flux en entrée ou en sortie, et sur l'exécution au premier plan (le shell ne rend pas la main) ou en arrière plan (le shell rend immédiatement la main) si le caractère `&` est présent sur la ligne de commande (dans ce cas, le tableau `modificateur` de la structure interne contient la marque `ARRIERE_PLAN`).

Fonctions utilitaires

`ecrire_variable` et `lire_variable` (définies dans le fichier `divers.c`) sont à écrire en utilisant les appels système `chdir` et `getcwd`.

`COMMENCE_PAR` (définie dans le fichier `divers.h`) est une macro du préprocesseur qui compare les n premiers caractères de deux chaînes, où n est la taille de la deuxième chaîne. Un exemple d'utilisation est `COMMENCE_PAR(s, "abc")` qui renvoie *vrai* si la chaîne `s` commence par les 3 caractères `abc`.

`EST_EGAL` (définie dans le fichier `divers.h`) est une macro du préprocesseur (basée sur la macro `COMMENCE_PAR`) qui renvoi *vrai* si les deux chaînes en paramètre sont égales.

`AfficheInvite` (définies dans le fichier `divers.c`) est une fonction qui lit la variable d'environnement `INV` et affiche une invite de commande correspondant à cette variable. Si la variable n'est pas définie, la fonction affiche uniquement les caractères `$␣`.

4 Travail à produire

Cette section décrit les différentes étapes à réaliser afin d'obtenir un shell utilisable. Pour rappel, le rapport de projet devra contenir une description des actions menées pour réaliser ces étapes, les éventuelles difficultées rencontrées, et les solutions trouvées pour résoudre ces difficultées, ainsi

3. Cette remarque est vraie pour la majorité des commandes internes, il faut considérer les commandes internes minimales comme étant cd et set.

qu'un ou plusieurs jeux de test permettant de mettre en évidence le fonctionnement et les limites des commandes internes/externes.

Variables d'environnement

Le code du shell fourni ne permet pas de définir ou de changer la valeur d'une variable d'environnement. En complétant ces deux nouvelles fonctions (via les appels systèmes `setenv` et `getenv` dans les fonctions associées du fichier `divers.c`), il sera possible de modifier (appel sur la ligne de commande à la commande interne `set`) et de lire (appel sur la ligne de commande au mécanisme de substitution de variable `%`) le contenu des variables d'environnement.

Quand cette opération est réalisée, l'invite de commande doit correctement s'afficher, et vous pouvez modifier cette invite dynamiquement :

```
set VAR = INV
set %VAR = \u\s[\p]\s
echo %OS
```

Exécution d'une commande interne

Le code du shell fourni ne contient pas les actions à réaliser pour afficher correctement le contenu d'un dossier (via l'utilisation de la bibliothèque `dirent` dans le fichier `ActionLS`). Cette commande interne est à compléter dans le fichier `commandes_internes.c`.

Par la suite, reprennez le code des commandes internes `ActionECHO`, `ActionLS` dans de nouveaux projets afin de réaliser une compilation autonome (trois programmes séparés : `echo`, `ls`) et de vérifier l'identicité des comportements dans le cas de l'exécution d'une commande interne et dans le cas de l'exécution d'une commande externe.

Exécution d'une commande externe

Le code du shell fourni ne contient pas les actions à réaliser pour exécuter une commande externe. Dans un premier temps, il est nécessaire de compléter le code de la fonction `ActionEXEC` afin de pouvoir lancer un processus externe (via l'appel système `exec` dans le fichier `commandes_externes.c`).

Quand cette opération est réalisée, vous pouvez exécuter les actions suivantes :

```
cd /usr/bin/
gedit
```

Exécution au premier plan

Par défaut, le lancement d'une commande se fait en arrière plan (le shell redonne la main sans attendre la fin de l'exécution de la commande lancée). Changez ce fonctionnement en fonction de la présence ou non, pour la commande en question, du caractère `&` (qui est repéré par le tag `ARRIERE_PLAN` placé par la boucle de parcours de la ligne de commande (fonction `execution_ligne_cmd`) sur le premier élément de la commande à exécuter).

Quand cette opération est réalisée, vous pouvez exécuter les actions suivantes :

```
cd /usr/bin/
gedit &
cal
```

Redirection en entrée et en sortie

Le code du shell fourni ne contient pas les mécanismes de redirection d'entrée ou de sortie.

Lors du parcours de la ligne de commande, et avant l'appel à la fonction `execution_cmd`, les éventuelles redirections ont été détectées. Ainsi, la structure de donnée interne contient deux champs `info->entree` et `info->sortie`. Si un champ est vide, il n'y a pas de redirection à réaliser, sinon il faut mettre en place le mécanisme de redirection depuis ou vers le fichier spécifié.

Redirection en entrée et en sortie d'une commande interne

L'exemple de la commande interne `echo` est le suivant : [4]

Dans un premier temps, la commande interne ouvre un fichier en écriture si la redirection est demandée :

```
FILE *sortie;
if (! EST_EGAL(info->sortie, ""))
{
  sortie=fopen(info->sortie,"w");
  if (sortie==NULL)
  {
    /* Traitement du cas où le fichier n'est pas accessible en écriture */
  }
}
```

Dans le cas où aucune redirection n'est demandée, il est nécessaire d'obtenir le flux de sortie par défaut (`stdout` dans la *libc*).

Dans le reste de la commande `echo`, l'affichage se fait en écrivant dans le fichier `sortie`. Cette écriture se fera sur la sortie standard s'il n'y a pas de redirection et dans le fichier spécifié par `info->sortie` s'il y en a une.

En fin de commande interne, il reste nécessaire de fermer le flux du fichier ouvert :

```
if (! EST_EGAL(info->sortie, ""))
{
  fclose(sortie);
}
```

Quand cette opération est réalisée, vous pouvez exécuter les actions suivantes :

```
echo 123 234 345 > fichier.txt
echo %OS > fichier2.txt
```

Redirection en entrée et en sortie d'une commande externe

Le traitement à réaliser, dans le cas de l'exécution d'une commande externe (avant l'appel à `exec` dans le fichier `commandes_externes.c`), est différent de celui réalisé dans le cas de l'exécution d'une commande interne. En effet, il n'est plus possible de modifier le comportement de la commande externe afin d'ouvrir un fichier si une redirection est souhaitée.

Par défaut, l'entrée standard et la sortie standard d'un processus fils sont héritées de son processus père. Dans un système de type unix, la modification des flux standard se fait via l'appel système `dup2`.

Quand cette opération est réalisée, vous pouvez exécuter les actions suivantes :

```
echo dir > fichier.txt
wc < fichier.txt
```

Exécution d'un/de plusieurs script(s)

Le code du shell fourni est exécuté par défaut, en mode interactif (les commandes sont lues sur l'entrée standard). Une utilisation du shell en mode non interactif (les instructions sont lues dans des fichiers de script) peut se faire par l'appel :

```
imrShell fichier1.imr fichier2.imr
```

4. Cette commande ne peut avoir qu'une redirection en sortie, l'exemple illustre donc ce type de redirection. Pour une redirection de l'entrée, les traitements sont similaires.

Le comportement souhaité est que le shell exécute les instructions contenues dans chacuns des fichiers passés en paramètres. Pour réaliser cette opération, il est possible de définir une variable booléenne (dans le fichier `shell.c`, dont la valeur est le résultat de l'opération `argc==1`) qui indique si des arguments ont été donnés au shell.

Si aucun argument n'a été donné, alors le comportement est interactif (et la variable booléenne a *vrai* pour valeur), si des arguments ont été donnés, alors le comportement est non interactif (et la variable booléenne a *faux* pour valeur). Bien sûr, il ne faut afficher l'invite du shell que dans le cas d'un comportement interactif, etc.

Il convient maintenant de vérifier que les variables sont correctement transmises à un processus fils, par exemple en donnant un script à interpréter par le shell :

```
set MAVAR = 12
set MAVAR2 = 15

echo echo %MAVAR > script1.imr
echo echo %MAVAR + 3 = %MAVAR2 > script2.imr

imrShell script1.imr script2.imr
```

Enchainement de commandes

Le code du shell fourni ne contient pas les actions à réaliser pour pouvoir exécuter des commandes multiples sur une même ligne, ni pour exécuter des commandes si une première commande a échoué/réussi.

La condition d'enchaînement des commandes est connue après l'analyse de la ligne de commande, et le mécanisme de sélection des actions à mener est déjà écrit dans la fonction `execution_ligne_cmd`. Il vous reste juste à *passer* la commande suivante si le résultat de l'exécution de la commande considérée est incorrect (des contrôles sont déjà positionnés dans le fichier `execution.c`, il vous reste à compléter les blocs d'instructions).

Quand cette opération est réalisée, vous pouvez exécuter les actions suivantes :

```
cd / && echo ok
cd repertoire_bidon || echo ko
```

Vous pouvez noter que la multiplication des exécutions conditionnelles sur une même ligne de commande est parfaitement correcte. Par exemple la commande suivante :

```
cd / && echo ok || echo ko
```

affiche `ok` si la commande `cd` est exécutée avec succès, et affiche `ko` si la commande retourne un code d'erreur.

Le cas de l'appel suivant :

```
cd / && echo ok && echo ok
```

affiche deux fois `ok` si la commande `cd` est exécutée avec succès, et rien en cas d'erreur.

Par contre, dans le cas de l'appel suivant :

```
cd repertoire_bidon && echo ok || echo ko && echo ok
```

l'échec de l'exécution de la commande provoque l'affichage de `ko` puis l'exécution en séquence de `echo ok` car la commande `echo ko` s'est correctement exécutée.

Tube

Le code du shell fourni ne contient pas les actions à réaliser pour mettre en place un mécanisme de communication entre deux processus.

La fonction `execution_cmd` du fichier `execution.c` n'est écrite que pour exécuter une seule commande. Il vous faut donc définir une nouvelle fonction afin de lancer deux commandes [5] liées par un tube annonyme.

Quand cette opération est réalisée, vous pouvez exécuter les actions suivantes :

```
echo ok | imrShell
```

5. La prochaine amélioration du shell pourrait être de pouvoir réaliser un enchaînement de plus de deux commandes via un tube.