IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# Programmation système Unix/Linux

UE OSAP

Printemps 2020

## Introduction

— Vous avez deux séances de TP de 3h pour faire les exercices proposés dans ce document. Le dernier exercice est facultatif.
— Pour chaque exercice il y a un listing de programme à compléter. Prenez l'habitude de commenter le code de vos programme. Organiser vos listings de programme dans votre répertoire de travail. Chacun de vos listing aura un nom préfixé par le numéro de la question correspondante (p.ex. `1_1-status.c`), et débutera par un commentaire indiquant votre nom.
— Rappel pour compiler : `gcc -Wall -o programme programme.c`
— Avant toutes choses, récupérez sur Moodle le fichier `ProgSystem.tgz`, et desarchivez-le (commande `tar xvf ProgSystem.tgz`. Il contient les programmes à compléter au cours du TP.

## 1 Appels systèmes et fonctions, pointeurs et structures

En guise de rappel sur le langage C, jouons avec les pointeurs... Mais aussi habituons-nous à consulter le manuel de référence [1] et terminons par une réflexion philosophique sur les vertus des appels systèmes comparés aux mérites des fonctions.

1. Le programme `1_1-status.c` accepte un nom de fichier en argument et, à l'aide de l'appel système `stat()`, donne pour ce fichier, le mode, la taille en octets et le numéro UID de l'utilisateur. Le résultat à la forme suivante :
   `Fichier xyz: Mode 81ED Taille: 24576 Octets Proprietaire: 821`
   Le modèle proposé comporte une erreur, cependant le compilateur accepte cette erreur et mieux, il la corrige (dans les dernières versions du compilateur), donc la compilation et même l'exécution (ce qui est nouveau) se passent bien.
   Ce n'est pas une erreur de syntaxe, mais une mauvaise utilisation de pointeur. Examinez le code pour trouver l'erreur et corriger le code afin qu'il soit plus correct.
2. Compléter le programme en remplaçant l'affichage du numéro UID par le nom en clair du propriétaire du fichier. On utilisera la fonction `getpwuid(3)`.
   Résultat attendu :
   `Fichier xyz: Mode 81ED Taille: 24576 Octets Proprietaire: dupont`
3. Complétez le programme en faisant deux appels successifs à `getpwuid()` avant de faire les affichages : une première fois "pour de vrai" (sur le véritable UID du propriétaire du fichier passé en argument), puis une seconde fois "pour de faux" (sur l'UID 1000 en sauvegardant le résultat dans une variable que l'on ignorera par la suite).
   Voyez-vous un problème ?
4. Pourquoi devons-nous passer un pointeur à `stat()` pour récupérer le résultat alors que la fonction de type `getpwuid()` nous indique l'adresse de son résultat par un pointeur en retour ?
   Notez que l'explication ici n'est pas du tout la même que celle qui oppose `getpwuid()` à `getpwuid_r()`.

## 2 Accès aux variables d'environnement dans un processus

1. Écrire un programme affichant le contenu du tableau `environ`. Ce tableau est une variable externe, connue du système, il convient donc de la déclarer avec le mot clef `extern` (voir `man environ`). Il contient des pointeurs sur des chaînes de caractères de la forme `VAR_ENV=valeur`, le nombre d'éléments contenus est variable, la fin du tableau est indiquée par un 0 (le pointeur `NULL`).

1. Par exemple : `man malloc`

2. En utilisant la fonction `getenv()`, écrire un petit programme qui affiche la valeur d'une variable d'environnement dont on passe le nom en paramètre du programme.

# 3 Utilisation de `fork()` et `exec()`

Un petit shell rien qu'à nous (mais pas très performant).

Travaillez de manière progressive, n'essayez pas de résoudre tous les problèmes à la fois, suivez consciencieusement les consignes de chaque question.

1. Travaillez sur l'ébauche de programme `3_1-petit_shell.c`. Compléter ce programme de manière à ce qu'il émule un shell très simple. Pour cela, faire un `fork()`, puis dans le processus fils, faire un `execl()` de la commande UNIX récupérée au prompt. Dans cette question on présuppose que l'utilisateur donne le chemin complet du programme à exécuter (p.ex. `/bin/ls`, `/bin/date`, `/usr/bin/last`, etc. ) sans ajouter d'option à sa ligne de commande.
2. Recommencer avec un autre appel système de la famille *exec*, différent de `execl()`, mais qui prenne en compte la variable d'environnement `PATH` pour rechercher les commandes à exécuter.
3. Lancer quelques commandes dans le `petit_shell` ci-dessus. Faire volontairement des erreurs. On constate qu'il faut autant de `[Control-D]` plus 1 que l'on a fait d'erreur pour sortir du `petit_shell`. Recommencer l'opération, sans toutefois chercher à sortir. Faire `ps -e` dans une fenêtre à part. On peut constater qu'il y a autant de processus `petit_shell` qu'il y a eu d'erreurs de commises. Pourquoi ? Modifier le programme en conséquence.
4. Utiliser maintenant le fichier `3_4-moyen_shell.c`. Ce programme *casse* la ligne de commande passée au prompt en mots, placés dans une chaîne de caractères. Les adresses de ces sous-chaînes sont regroupées dans un tableau de pointeurs qui pourra être utilisé avec un membre de la famille `exec`. On pourra alors passer des commandes avec un nombre variable d'arguments.

# 4 Le traitement des signaux

1. Produisons un `core file`. (Lisez le `man core`.) Dans votre shell, si c'est un shell tcsh tapez `limit coredumpsize unlimited` (voir le `man tcsh`), ou si c'est un shell bash tapez `ulimit -c unlimited` (voir `help ulimit`). Compilez le programme `4_1-sig.c` et exécutez-le.
   Depuis un autre terminal, avec la commande `kill`, envoyez-lui le signal `SIGILL` (*instruction illégale* ; en principe c'est plutôt le noyau qui décide d'envoyer ça à un processus). Cela produit un fichier `core` (le `man 7 signal` donne la liste des signaux qui produisent un *core dump*). [2]
2. Modifier ce programme pour qu'il ne produise plus de fichier `core` sur réception sur le signal `SIGILL`, mais affiche un simple message. Êtes-vous certain qu'il ne se termine pas lorsque l'on ré-émet `SIGILL` une seconde fois (c.a.d. est-ce que votre handler est toujours en place après avoir servi une première fois) ?
3. Modifier le même programme pour qu'il émette un *message numéro 1* ou un *message numéro 2* sur réception du signal `INT` (lorsque l'on tape `[Control-C]`). Le choix du message émis se fait par commutation commandée par le signal `QUIT` (lorsque l'on tape `[Control-\]`). On commence par le *message 1* qui est émis systématiquement jusqu'à ce que le signal `QUIT` soit envoyé, c'est alors le *message 2* qui s'affiche sur réception de `INT`. Un nouveau `QUIT` rétablit la sortie du *message 1* et ainsi de suite.
   Note : Dans cet exercice on vous demande d'utiliser la valeur retournée par la fonction `signal()`. On ne veut pas une bidouille à base de booléen en variable globale ou autre truc du genre... et il n'y a pas non plus besoin de `if`... Réfléchissez !
4. Faire la même chose avec la fonction `sigaction()`.

# 5 La terminaison des processus

1. Les processus zombies
   Copier le texte du programme `5_1-zomb.c`. Compilez, lancez en *background*, puis faites `ps -el` après 5 secondes. Vous constatez qu'il existe un processus `zomb` et que ce dernier a un fils dont l'état est marqué

2. Note pour les utilisateurs d'Ubuntu : Ubuntu a un comportement non standard et mal documenté concernant le core dump. Certains développeurs ont eu l'idée géniale de détourner la génération de core dump pour envoyer des rapports de bug aux développeurs des applications qui ont plantées (à condition que ce soit une application Ubuntu). C'est le paquet `apport` qui fait ça. Il est vaguement désactivé par défaut, mais possiblement bloque tout de même la génération des fichiers core dump. Pour retrouver un système Unix conforme à POSIX, soit supprimer ce paquet `apport`, soit faire :
`sudo sysctl kernel.core_pattern=core`
(Voir le `man core`.)

`Z` et le nom est `[defunct]`. Ce processus est un *zombie process*. On ne peut plus s'en débarrasser sans tuer le processus père (avec kill).

Modifier le programme pour qu'à la mort du processus fils, ce dernier ne reste pas en zombie. Vous utiliserez pour cela la fonction `signal()` et dans un premier temps, le `wait()` aura `NULL` comme seul argument.

2. Récupération par le père du paramètre d'`exit` d'un fils

   Quand un processus se termine, le paramètre du `exit()` de terminaison (il y a toujours un `exit()`, même si on ne le précise pas explicitement) peut être récupéré par le processus père, dans l'argument du `wait()`. Ce paramètre peut donner des indications sur la terminaison correcte ou incorrecte du fils. C'est ainsi que toutes les commandes UNIX rendent une information pouvant être testée par l'utilisateur : 0 pour une terminaison correcte (assimilé à VRAI par les shells) et différent de 0 lorsque le travail n'a pas pu être mené à bien par la commande (assimilé à FAUX). Cette valeur est accessible par `$?` en Bourne Shell ou par `$status` en C-Shell. Essayez par exemple `grep obelix /etc/passwd` puis `echo $status` (ou `echo $?`). Puis recommencez avec `root` au lieu de `obelix`.

   Prenez le modèle `5_2-zomb_exit.c`. Le paramètre du `exit` du fils est fourni au programme en premier argument de la ligne de commande (`argv[1]`). Modifiez ce modèle de manière à récupérer le code de `exit` du fils dans le père à l'aide de l'argument du `wait()`. La modification inclura aussi l'obtention par le père du numéro de signal reçu si le fils se termine sur un signal.

   Testez le code en lançant le programme avec pour argument un nombre compris entre 0 et 255.

   Modifiez la valeur du `sleep()` dans le fils en lui donnant la valeur 30s. Vous aurez ainsi le temps de retrouver avec `ps` le numéro du fils dans une autre fenêtre. Tuez le fils en lui envoyant un signal.

# 6 L'appel système `fcntl()`

1. Prendre l'ébauche `6_1-read_on_delay.c`.

   Ce programme lit simplement le terminal et donc bloque en lecture en fonctionnement normal. Si on lui passe l'argument true, il appelle `fcntl()` pour positionner le flag `O_NONBLOCK` afin de ne plus bloquer.

   Compléter l'ébauche fournie pour réaliser le positionnement du flag. (Prier d'ajouter ce flag sans écraser ceux déjà présents.) Vérifier l'effet en utilisant en parallèle le gestionnnaire graphique de processus. Vous pouvez aussi vérifier le fonctionnement en lançant l'exécution via la commande de traçage `strace`. La commande `top` donne aussi des indications intéressantes.

   Note : Avec certaines version du shell `bash` l'entrée standard n'est pas restaurée en mode non-bloquant à la terminaison de notre programme (qui a positionné stdin en mode non-bloquant sans le restaurer à la fin...). En principe, cela peut être restauré avec la commande `stty sane`. Si ce n'est pas le cas, une astuce consiste a lancer un nouveau processus bash dans le shell et d'en sortir (c.à.d. taper `bash` puis `exit`).

2. Modifier le programme pour enlever le flag (et seulement ce flag) sur réception du signal `SIGINT`. (Vérifier auparavant, avec la commande `stty -a`, quelle combinaison de touches clavier permet d'envoyer ce signal). Compléter le programme en rajoutant un gestionnaire pour le signal `SIGQUIT` qui repositionne le flag.

# 7 Les tubes de communication

Écrire un programme qui crée un tube de communication puis un fils. Le père lit le clavier (fichier standard d'entrée) et écrit dans le tube ce qu'il a lu. Le père est paramétré de telle manière que son fichier standard de sortie soit le tube. Le fils lit son standard d'entrée et il est paramétré de telle manière que celui-ci soit le tube. Il affiche à l'écran ce qu'il a lu.

Vous devez donc créer un tube, puis un fils. Dans les deux processus vous devez tout de suite faire une redirection, standard de sortie dans le père, standard d'entrée dans le fils.

# 8 Contrôle des terminaux

Nous allons développer une petite application qui chiffre un mot de passe entré au clavier. Le chiffrement utilise la fonction classique `crypt()`. Cette dernière utilise une clé de chiffrement constituée par une chaîne de 2 caractères pouvant être les lettres de l'alphabet en minuscule ou majuscule, un chiffre entre 0 et 9 ou les caractères «.» et «/», soit 64 caractères possibles. Pour calculer la clé on place les caractères dans un tableau et on fait deux fois un tirage aléatoire entre 0 et 63. On utilisera la fonction `random()` pour faire le tirage, cette fonction rend un entier long, il suffira de prendre le modulo 64 de cet entier.

Pour que le résultat soit vraiment aléatoire il est nécessaire *d'armer* le dispositif avant le tirage avec la fonction `srandom()` qui, elle aussi, a besoin d'une *graine* de départ. Un bon moyen de trouver une graine très variable est de prendre la date en microsecondes avec la fonction `gettimeofday()`.

1. Prendre l'ébauche `8_1-mypass.c` et compléter la fonction `getkey()`. Afficher le résultat de la recherche de clé.
2. Compléter la fonction `get_pass()` qui lit le mot de passe entré au clavier puis compléter la fonction `main()` afin d'effectuer le chiffrement.[3] Afficher le résultat.
3. Compléter la fonction `setsilent()` pour que l'on puisse rentrer son mot de passe au clavier sans qu'il soit affiché en echo à l'écran. On utilisera les fonctions `tcgetattr()` et `tcsetattr()` pour obtenir et positionner les paramètres du terminal et en particulier modifier le flag autorisant ou non l'écho (voir la structure `struct termios` dans la page du `man termios`).
4. Compléter le paramétrage du terminal de telle manière que les caractères entrés au clavier soient immédiatement lus (sans attendre le retour à la ligne) et afficher le caractère «*» pour chaque caractère entré (il faut placer le terminal en mode d'entrée non canonique (`~ICANON`) et jouer sur les paramètres `c_cc[VMIN]` et `c_cc[VTIME]`).
5. Pensez à restaurer le terminal dans son fonctionnement initial à la terminaison de votre programme.

3. En fait, une telle fonction existe déjà dans la libc (faire `man getpass`), mais c'est plus rigolo d'essayer de la programmer soi-même, non ?