

SR01 : Maîtrise des systèmes informatiques

Devoir 2 : Programmation système

GAJAN Antoine & ZHENG Cristina



Table des matières

Introduction	2
1 Exercice 1 : Création de processus fils	3
2 Exercice 2 : Git et compilation de fichiers	4
2.1 Fonctionnement du programme	4
2.1.1 Arbre généalogique des processus	4
2.1.2 Explication détaillée du fonctionnement du programme	4
2.1.3 Rôle des instructions 41 et 46	5
2.1.4 Garantie de la synchronisation	5
2.2 Git et implémentation du programme	6
2.2.1 Installation de Git	6
2.2.2 Copie locale de dépôt	7
2.2.3 Explication de nbr_premiers.txt	8
2.3 Modification, modularité et Makefile	8
2.3.1 Problème dans la récupération des PID	8
2.3.2 Utilisation de execv	9
2.3.3 Exécution des fils en parallèles	11
2.3.4 Makefile optimisé	12
2.3.5 Validation des modifications sur Git	13
3 Exercice 3 : Gestion des signaux et handlers	15
3.1 Analyse du sujet	15
3.2 Structure Application	15
3.3 Pseudo code pour la lecture du fichier	15
3.4 Fonction main	16
3.4.1 Lancement des applications	16
3.4.2 Gestion de la fermeture d'une application	16
3.4.3 Fermeture du programme principal suite au signal SIGUSR1	17
3.5 Demande de mise en veille de power_manager	18
Conclusion	20

Introduction

Ce TP a pour but de nous familiariser avec les concepts de processus, programmation système et administration système.

Au cours de celui, les connaissances acquises lors des premiers cours magistraux et travaux dirigés seront mobilisées pour monter en compétences. Composé de 3 exercices, il nous permettra de comprendre la gestion des processus (exercice 1), Git et la compilation des fichiers (exercice 2) et le fonctionnement des signaux (exercice 3).

Dans chacun des exercices, nous veillerons à expliciter les choix effectués.

1 Exercice 1 : Création de processus fils

L'objectif de cet exercice est de comprendre le fonctionnement de la fonction `fork()` et de récupérer les informations associées aux identifiants de processus (PID) et leurs parents (PPID).

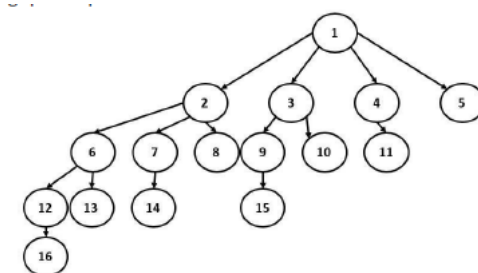


FIGURE 1 – Arbre de processus

On souhaite générer l'arbre de processus suivant et afficher pour chaque processus la ligne "Mon pid est : xx et le pid de mon père est : xx".

Notre code repose sur le fonctionnement suivant. L'appel système `fork` est utilisé pour créer un nouveau processus, appelé processus enfant, qui s'exécute en même temps que le processus qui effectue l'appel `fork()` (processus parent). Dans le processus parent, l'appel système renvoie un nombre positif correspondant au pid du fils. Dans le processus fils, l'appel système renvoie 0. Grâce à cette compréhension de `fork()`, on peut les imbriquer successivement de manière à obtenir le graphe hiérarchique des processus escomptés.

Toutefois, si on exécute le programme alors créé, un problème survient. L'affichage ne correspond pas à ce qui est souhaité. En effet, pour certains affichages, le programme affiche que le PPID du processus est 1. Cela est dû au fait que nous avons oublié de gérer la concurrence des processus. En effet, il est important que le père attende et prenne connaissance de la fin de l'exécution de son fils pour se terminer à son tour. Cela évitera qu'un père se termine avant son fils et qu'il reste des processus zombies dans le système. Afin de gérer ce problème, il faut utiliser 2 fonctions. `Exit()`, exécuté par le fils, permet d'indiquer au processus père la fin de son exécution à son père. Quant à `wait()` de la librairie `<sys/wait.h>`, exécuté par le père, permet d'attendre la fin d'exécution du fils indiqué en paramètre. Ainsi, nous avons développé ce code dans `Prog1.c`.

2 Exercice 2 : Git et compilation de fichiers

2.1 Fonctionnement du programme

2.1.1 Arbre généalogique des processus

Afin de comprendre le fonctionnement du programme, il nous a d'abord fallu établir les relations entre les différents processus. Ceci nous permettra de mieux comprendre quel processus est le père et quel processus est le fils dans chacune des relations.

Après analyse rigoureuse du code, nous avons obtenu l'arbre de hiérarchie des processus suivant :

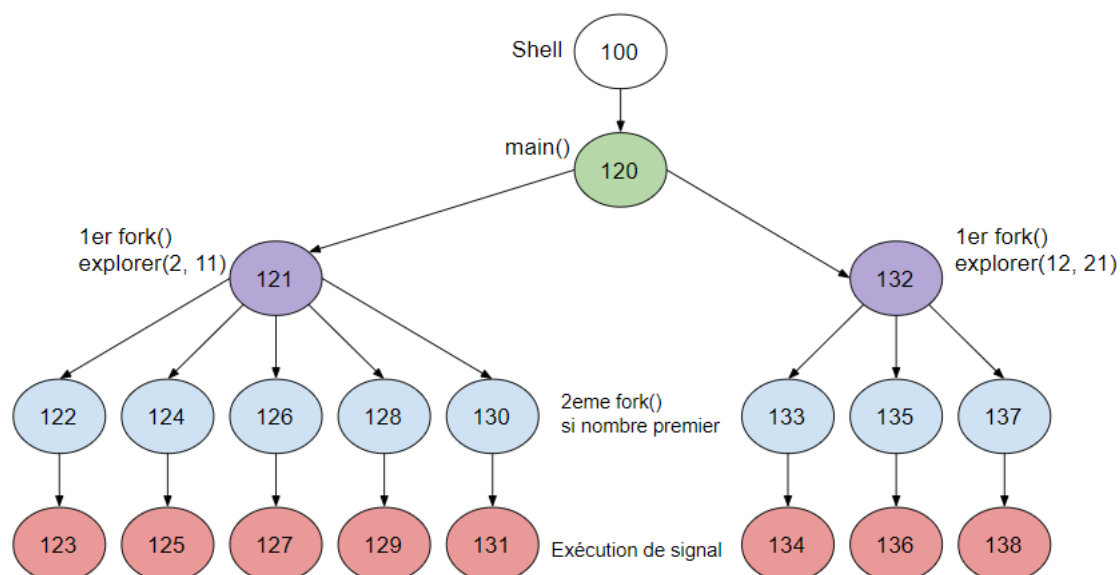


FIGURE 2 – Arbre généalogique des processus

2.1.2 Explication détaillée du fonctionnement du programme

Maintenant que nous comprenons mieux la génération des processus de ce programme, nous pouvons apporter une explication détaillée sur son fonctionnement. Tout d'abord, le shell (de PID 100) crée un processus fils de PID 120 qui exécute le programme Prog_premiers. La fonction main() se lance alors avec le PID 120. La fonction main() démarre par une initialisation de la variable grp à 1. Cette variable

s'incrémentera à chaque passage dans la boucle while jusqu'à ce que groupe soit supérieur à 11. En d'autres termes, on fera 2 appels à la fonction explorer() qui se situe dans la boucle while. On appelle explorer(2, 11). Au cours de cet appel, un appel fork est réalisé. Un processus fils (PID 121) va être créé. Les deux processus (de PID 120 et 121) vont être exécutés en concurrence. Le fils, de PID 121, va rentrer dans une boucle for tandis que son père de PID 120 va attendre la fin de son exécution pour poursuivre les instructions suivantes.

Le processus de PID 121 exécute la boucle for. Pour chaque nombre premier entre 2 et 11, le processus de PID 121 va créer un fils qui exécutera une commande Shell avec l'appel system() pour écrire dans le fichier nbr_premiers.txt le nombre premier et le PID du processus qui l'a écrit. Lors de l'entrée dans la boucle for, $i = 2$. 2 étant un nombre premier, le processus de PID 121 va créer un processus fils de PID 122 et attendre la fin de son exécution. Il va faire un appel system(), qui va impliquer la création d'un processus fils de PID 123, pour écrire dans le fichier nbr_premiers.txt "2 est un nombre premier écrit par le processus 122" (nous reviendrons sur cet affichage dans une question ultérieure). Une fois cette écriture effectuée, le processus 122 va attendre 2 secondes avant de retourner la valeur 0 pour indiquer à son père (de PID 121 qu'il a terminé son exécution). Le même mécanisme sera répété pour chacun des nombres premiers rencontrés entre 3 et 11 (à savoir 3, 5, 7, 11). En sortie de boucle for, le processus de PID 121 indique à son père (de PID 120) qu'il a terminé son exécution avec l'instruction exit(0).

Le processus de PID 120 reprend alors son exécution dans le main, et incrémente la variable grp de 10. On a alors $grp = 11$, qui va induire l'appel explorer(12, 21). L'exécution de la fonction explorer(12, 21) reposera sur le même principe qu'évoqué ci-dessus, avec une gestion des processus.

2.1.3 Rôle des instructions 41 et 46

Deux instructions ont suscité notre attention lors de la compréhension du code. Il s'agit de l'instruction wait(&etat); aux lignes 41 et 46. L'appel système wait() suspend l'exécution du processus appelant, jusqu'à ce que l'un de ses fils se termine. Dans le cas où l'on omet l'une de ces instructions, comme les différents processus s'exécutent en concurrence, il se peut que les pères n'attendent pas la fin de l'exécution de leur fils. On risque d'avoir des processus zombies dans le système. De plus, l'écriture des nombres premiers dans le fichier ne se fera plus par ordre croissant, mais dans le désordre.

2.1.4 Garantie de la synchronisation

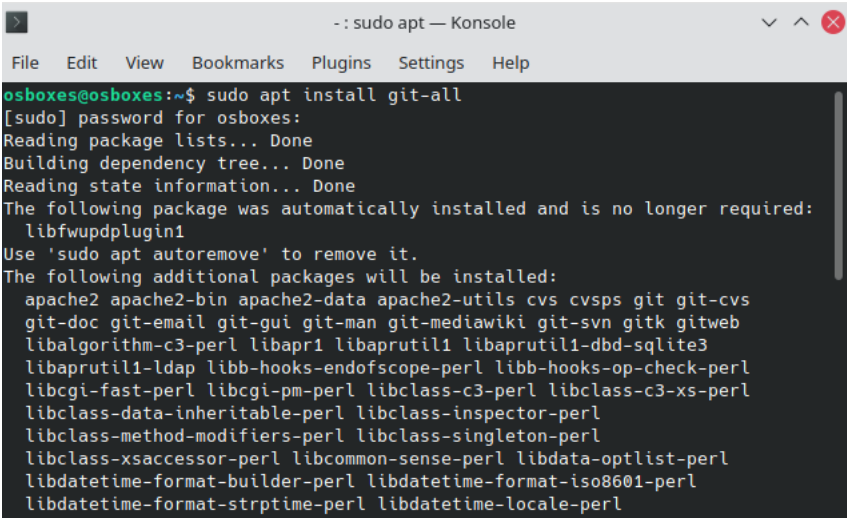
On souhaite à présent comprendre pourquoi un processus de PID p ne peut être exécuté qu'après la fin de l'exécution du processus de PID $p - 1$. Cela s'explique à nouveau par l'usage de l'appel système wait() qui suspend l'exécution du processus appelant jusqu'à ce que l'un de ses fils se termine. Cet appel est effectué par chaque

processus père, ce qui implique que chaque père va attendre la fin de l'exécution de son fils avant de poursuivre les instructions.

2.2 Git et implémentation du programme

2.2.1 Installation de Git

On souhaite installer Git sur la machine virtuelle Linux installée au début du semestre. Pour cela, il faut d'abord exécuter dans un terminal la commande `sudo apt install git-all`, qui va installer Git. Pour exécuter cette instruction, il faudra au préalable avoir configuré sur la machine virtuelle `sudo` (nous l'avons effectué au début du semestre).



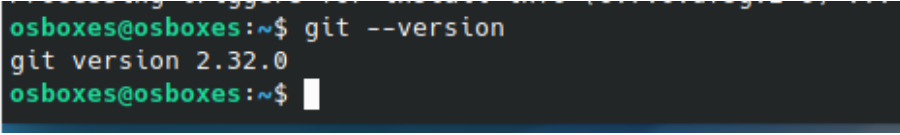
```

-: sudo apt — Konsole
File Edit View Bookmarks Plugins Settings Help
osboxes@osboxes:~$ sudo apt install git-all
[sudo] password for osboxes:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following package was automatically installed and is no longer required:
  libfwupdplugin1
Use 'sudo apt autoremove' to remove it.
The following additional packages will be installed:
  apache2 apache2-bin apache2-data apache2-utils cvs cvsps git git-cvs
  git-doc git-email git-gui git-man git-mediawiki git-svn gitk gitweb
  libalgorithm-c3-perl libapr1 libaprutil1 libaprutil1-dbd-sqlite3
  libaprutil1-ldap libb-hooks-endofscope-perl libb-hooks-op-check-perl
  libcgi-fast-perl libcgi-pm-perl libclass-c3-perl libclass-c3-xs-perl
  libclass-data-inheritable-perl libclass-inspector-perl
  libclass-method-modifiers-perl libclass-singleton-perl
  libclass-xsaccessor-perl libcommon-sense-perl libdata-optlist-perl
  libdatetime-format-builder-perl libdatetime-format-iso8601-perl
  libdatetime-format-strptime-perl libdatetime-locale-perl

```

FIGURE 3 – Installation de Git

On peut alors vérifier la version de Git présente sur la machine, à l'aide de `git --version`.



```

osboxes@osboxes:~$ git --version
git version 2.32.0
osboxes@osboxes:~$

```

FIGURE 4 – Version de Git sur la machine

Pour pouvoir ensuite faire des modifications en tant qu'utilisateur, il faut que nous configurions Git. Pour cela, nous appliquons les commandes vues en cours :

```
devoir2 : bash — Konsole
File Edit View Bookmarks Plugins Settings Help
osboxes@osboxes:~/devoir2$ git config --global user.email "antoine1.gajan@gmail.com"
osboxes@osboxes:~/devoir2$ git config --global user.name "antoine-gajan"
osboxes@osboxes:~/devoir2$
```

FIGURE 5 – Configuration de Git

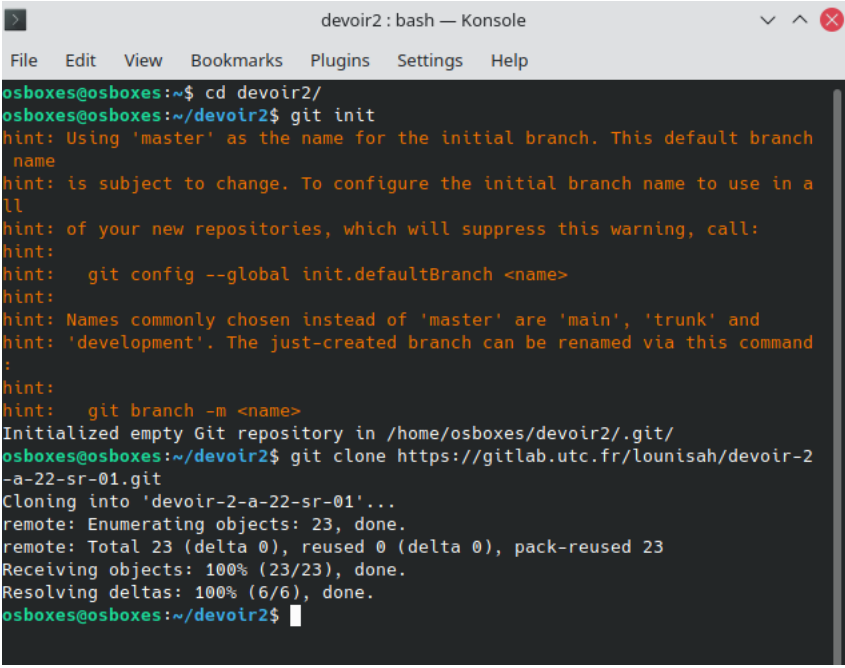
2.2.2 Copie locale de dépôt

La configuration de Git étant terminée, on peut utiliser Git de manière optimale. On veut à présent créer une copie locale du dépôt suivant : <https://gitlab.utc.fr/lounisah/devoir-2-a-22-sr-01>. Pour cela, il faut procéder comme suit. On crée un dossier à l'aide de `mkdir`.

```
~ : bash — Konsole
File Edit View Bookmarks Plugins Settings Help
osboxes@osboxes:~$ ls
Desktop Documents Downloads Music Pictures Public Templates Videos
osboxes@osboxes:~$ mkdir devoir2
osboxes@osboxes:~$ ls
Desktop Documents Music Public Videos
devoir2 Downloads Pictures Templates
osboxes@osboxes:~$
```

FIGURE 6 – Création du dossier devoir2

Ensuite, on initialise un dépôt à l'aide de la commande `git init`. Enfin, on clone le dépôt distant à l'aide du lien de clonage indiqué sur le Gitlab. On obtient alors :



```

devoir2 : bash — Konsole
File Edit View Bookmarks Plugins Settings Help
osboxes@osboxes:~$ cd devoir2/
osboxes@osboxes:~/devoir2$ git init
hint: Using 'master' as the name for the initial branch. This default branch
hint: name
hint: is subject to change. To configure the initial branch name to use in a
hint: ll
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command
hint: :
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/osboxes/devoir2/.git/
osboxes@osboxes:~/devoir2$ git clone https://gitlab.utc.fr/lounisah/devoir-2
-a-22-sr-01.git
Cloning into 'devoir-2-a-22-sr-01'...
remote: Enumerating objects: 23, done.
remote: Total 23 (delta 0), reused 0 (delta 0), pack-reused 23
Receiving objects: 100% (23/23), done.
Resolving deltas: 100% (6/6), done.
osboxes@osboxes:~/devoir2$

```

FIGURE 7 – Initialisation du dépôt et clonage

2.2.3 Explication de nbr_premiers.txt

Le fichier nbr_premiers.txt contient les nombres premiers de 2 à 21 ainsi que les PID de processus ayant demandé l’affichage avec la commande system().

On remarque que sprintf permet de créer une chaîne de caractères formatée où les valeurs %d sont remplacées par les valeurs des variables mises en paramètres. system() est un appel système qui permet de lancer l’exécution d’une commande dans le terminal à partir d’un nouveau processus et va exécuter la commande contenue dans la chaîne de caractères formatée créée avec sprintf. La commande est, dans le cas du nombre 2 : "echo '2 est un nombre premier écrit par le processus 122' » nbr_premiers.txt". Cette commande affiche le contenu entre les simples quotes dans le fichier nbr_premiers.txt grâce à la redirection de la sortie standard avec l’opérateur ».

2.3 Modification, modularité et Makefile

2.3.1 Problème dans la récupération des PID

Les PIDs des processus qui ont écrit dans le fichier nbr_premiers.txt ne sont correctement récupérés. En effet, les PIDs affichés dans le fichier correspondent aux identifiants des processus qui ont appelé la commande system(). En d’autre terme, c’est le père du processus qui a écrit le message sur le terminal.

On peut donc proposer le code Prog_premiers_m1.c. Celui-ci va afficher le PID du processus qui a créé le processus system (processus qui fait appel à la fonction system), le père de ce processus (processus qui exécute la boucle for) et le PID du processus créé dans la fonction system.

L'unique difficulté vient du fait de comprendre que le PID du processus créé dans la fonction system est une variable de type shell, qu'il faut obtenir grâce à la commande \$\$. On obtient alors le morceau de code associé suivant :

```
sprintf(chaine,"echo '%d est un nombre premier dont le processus qui
fait appel a la fonction system est %d, dont le pere qui execute la
boucle for est %d et est écrit par le processus $$' >>
nbr_premiers.txt",i,getpid(),getppid());
```

Après modification du code, on obtient alors le résultat suivant dans nbr_premiers.txt :

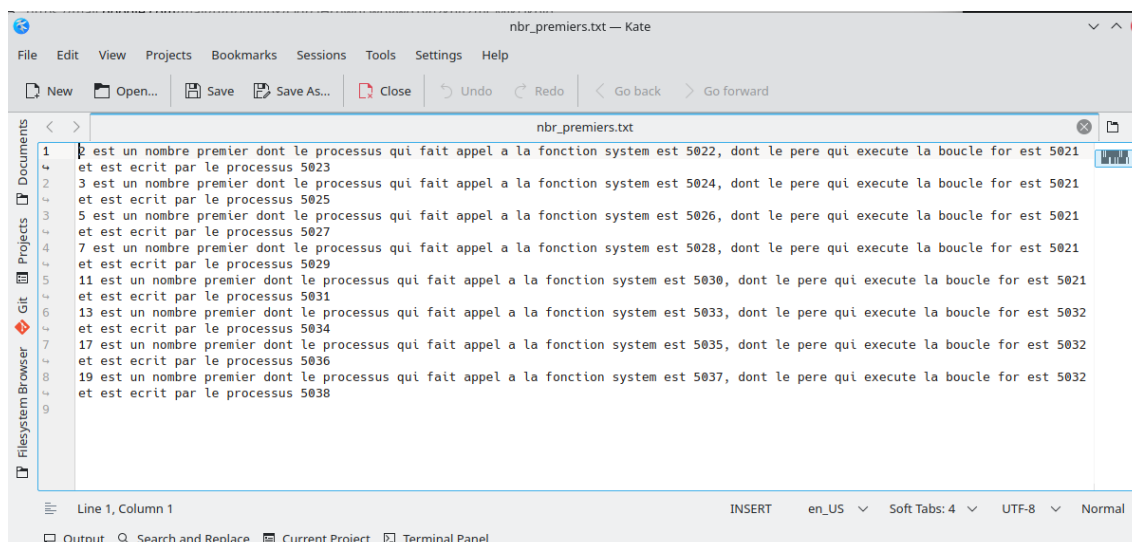


FIGURE 8 – Fichier nbr_premiers.txt avec précision des processus

2.3.2 Utilisation de execv

On souhaite à présent utiliser execv à la place de l'appel system(). En particulier, on voudrait créer une fonction my_system(), qui a la même finalité que system(). Pour cela, il est important de comprendre le fonctionnement de execv(). Les appels système execv() remplacent le processus courant par un nouveau processus construit à partir d'un fichier ordinaire exécutable. Les segments de texte et de données du processus sont remplacés par ceux du fichier exécutable. En d'autres termes, pour pouvoir reproduire le fonctionnement de system() avec la fonction execv(), il faudra dans un premier temps créer un processus fils avec fork(), et exécuter dans le processus fils la fonction execv(). Ainsi, une fois l'exécution du fils terminée, le processus

père pourra reprendre son activité sans que les instructions qu'il devait exécuter aient été remplacées par celles du fichier exécutable.

Au sein du processus fils crée, il faudra veiller à rediriger la sortie standard (qui auparavant était réalisée par `system`) vers le fichier `nbr_premiers.txt`. En particulier, il faudra utiliser les commandes `dup()` et `close()` judicieusement afin de parvenir à l'effet voulu.

Ainsi, on a la fonction `my_system` suivante :

```
void my_system(char *chaine)
{
    pid_t pid;
    int stat;
    // Creation d'un processus fils en concurrence
    pid = fork();
    if (pid == 0)
    {
        // Code a executer par le fils
        // Arguments pour l'ecriture
        char *arguments[] = {"/bin/echo", chaine, NULL};
        // Copie de la sortie standard pour la sauvegarder (utilise
        emplacement 3)
        int new_stdout = dup(1);
        // Creation d'un descripteur de fichier a l'emplacement 4
        int fd = open("nbr_premiers.txt", O_RDWR | O_APPEND | O_CREAT,
        0666);
        // Fermeture de la sortie standard stdout
        close(1);
        // On copie le descripteur vers 1 (sortie standard)
        int new_file_txt = dup(fd);
        // On ferme le descripteur qui est devenu inutile
        close(fd);
        // Ecriture dans la sortie standard (fichier nbr_premiers.txt)
        execv("/bin/echo", arguments);
        // On remet stdout comme sortie standard
        close(new_file_txt);
        dup(new_stdout);
        close(new_stdout);
        exit(0);
    }
    else
    {
        // Attente que le processus fils termine
        wait(&stat);
    }
}
```

Afin de ne pas alourdir le code présenté, le code ci-dessus ne comporte pas les vérifications et affichage d'erreur. Ces tests et affichages sont présents dans le code source et ont été réalisés au moyen de la fonction `perror()` qui utilise `errno` de la librairie `errno.h`.

Enfin, il nous a fallu changer le texte de la variable `chaine` dans la fonction `explorer`. En effet, maintenant que l'écriture et la redirection de la sortie standard

est faite manuellement dans `my_system()`, la variable chaîne de la fonction explorer peut être simplifiée comme suit :

```
sprintf(chaine, "%d est un nombre premier. Le processus qui fait appel a  
system() est %d et son pere est %d", i, getpid(), getppid());
```

2.3.3 Exécution des fils en parallèles

On voudrait exécuter les fils en parallèle sans pour autant créer des processus zombies.

Pour cela, il ne faut plus que le processus père (celui qui exécute la boucle for) attende l'exécution de son fils (lié au fork du nombre premier) pour continuer à itérer dans la boucle. Néanmoins, il faut éviter les processus zombies. Pour cela, il faudra tout de même attendre la fin des processus fils créés dans la boucle for une fois l'exécution de cette dernière entièrement terminée. Sachant que l'appel système wait attend la fin d'un des processus, il faut connaître le nombre de processus fils créés pour ne pas oublier d'en attendre un. Nous pouvons donc rajouter une variable pour cela. Nous l'incrémenterons à chaque nombre premier rencontré. Enfin, nous rajouterons une boucle for pour attendre l'ensemble des processus fils. Ainsi, les processus fils s'exécuteront en parallèle sans pour autant qu'ils deviennent des zombies à la fin de leur exécution (voir Prog_premiers_m3.c).

Cette optimisation permet un gain de temps considérable, puisque les processus s'exécutent en parallèles. Néanmoins, cela se fait au détriment de la lisibilité du fichier `nbr_premiers.txt`. En effet, comme nous n'attendons plus qu'un processus ait fini d'écrire dans le fichier pour poursuivre le programme et les prochaines écritures, certains affichages se font dans le désordre, comme en témoigne la capture d'écran ci-dessous.

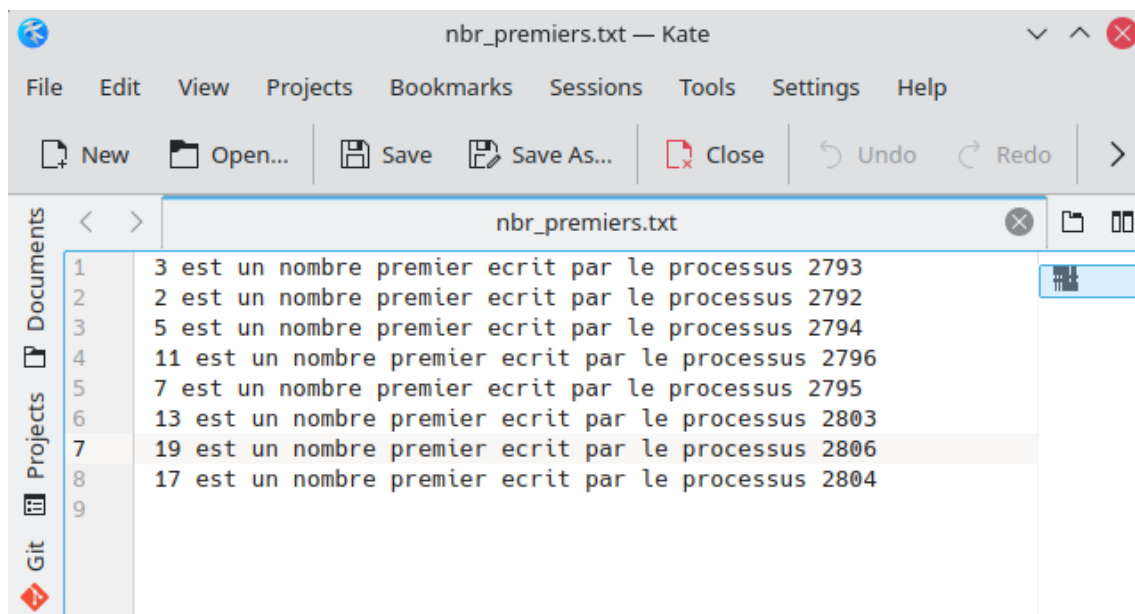


FIGURE 9 – Affichage du programme prog_premiers_m3.c

2.3.4 Makefile optimisé

Nous voulons à présent optimiser le Makefile. Pour cela, nous allons utiliser des macros (statiques et dynamiques) et ajouter des règles. A l'aide des connaissances apportées par le TD sur Makefile et le cours magistral, on obtient le Makefile suivant :

```

OBS = main.o explorer.o premier.o my_system.o
CC = gcc

Prog_premiers : $(OBS)
    $(CC) -o $@ $(OBS)

.c.o :
    $(CC) -c $<

main.o : explorer.h
explorer.o : my_system.h explorer.h premier.h
premier.o : premier.h
my_system.o : my_system.h

clean :
    rm Prog_premiers *.o

```

La création de ce Makefile optimisé nous permet de nous affranchir de nombreuses lignes de commandes. En effet, la création de l'exécutable se fait automatiquement à l'aide de la commande `make -f MakefileOpt` et la suppression des fichiers objets et de l'exécutable se fait à l'aide de `make -f MakefileOpt clean`.

Ainsi, après exécution des commandes, on obtient le résultat suivant :

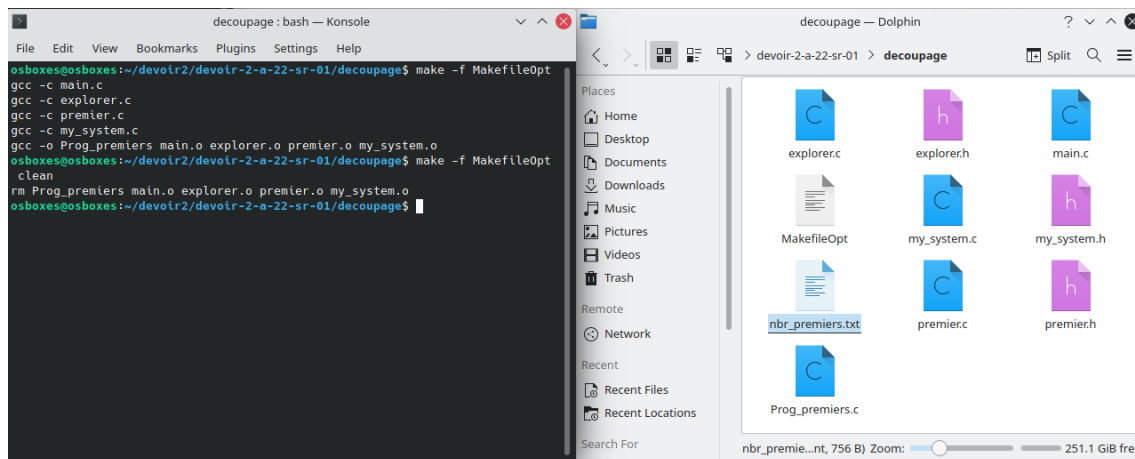


FIGURE 10 – Création et suppression de l'exécutable grâce au Makefile optimisé

2.3.5 Validation des modifications sur Git

Nous devons maintenant ajouter et valider nos modifications sur Git en local. Pour cela, nous allons utiliser 3 commandes. La première, optionnelle, est `git status`. Elle nous indique quels fichiers ont été modifiés, supprimés ou créés au cours de la session de travail. La deuxième, `git add *`, est obligatoire et indexe tous les fichiers modifiés dans leur version actuelle pour qu'il fasse partie du prochain instantané du projet. Enfin, `git commit`, valide les données et les stocke en sécurité dans notre base de données locale. L'option `-m` permet d'écrire un message afin de permettre aux collaborateurs de comprendre les modifications qui ont été apportées au projet.

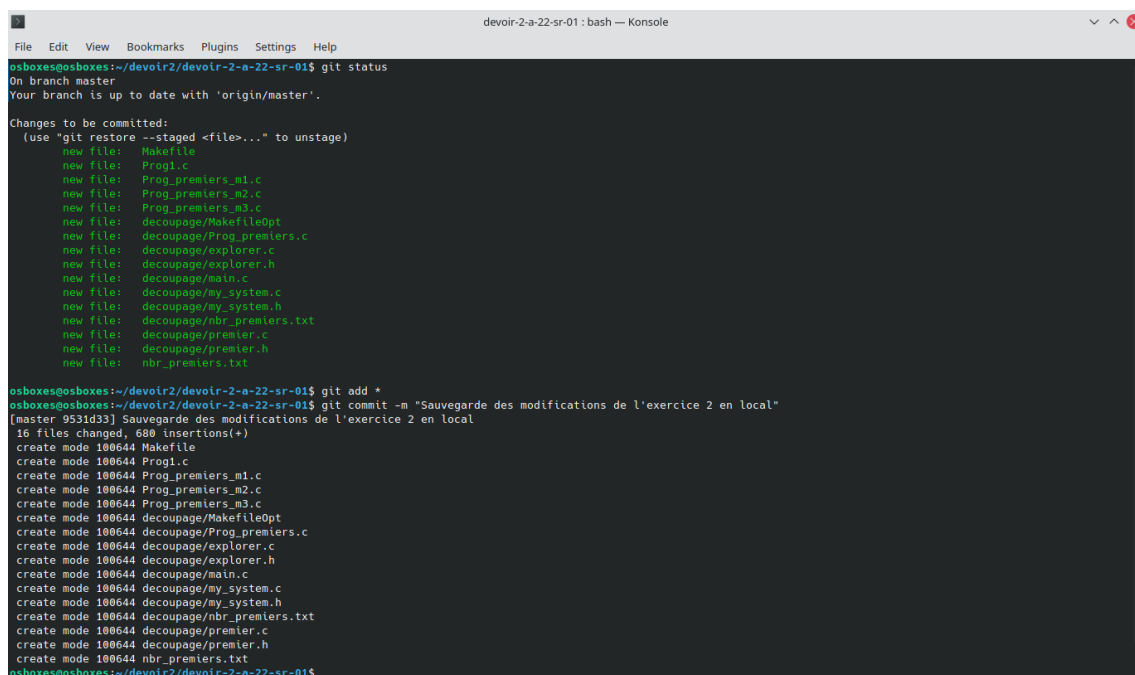


FIGURE 11 – Ajout et sauvegarde des modifications en local

Ce n'est pas demandé ici, mais dans le cas d'un projet, si l'on souhaite que les autres collaborateurs puissent voir nos modifications sur le dépôt distant pour pouvoir les récupérer, il faudra utiliser la commande `git push` pour que la sauvegarde se fasse également sur le dépôt distant.

3 Exercice 3 : Gestion des signaux et handlers

3.1 Analyse du sujet

Contrairement aux deux premiers exercices, cet exercice est moins guidé. Il nous a donc fallu analyser méticuleusement le sujet pour comprendre ce qui était réellement attendu et répondre au mieux à la problématique de départ. Nous avons d'abord commencé par écrire un pseudo algorithme pour savoir quelles seraient les fonctions dont nous aurions besoin. Nous avons d'abord réalisé qu'il faudrait une fonction pour lire dans le fichier `list_appli.txt`. Ensuite, nous avons créé une structure pour contenir les informations relatives aux applications et simplifier notre code par la suite. Enfin, nous nous sommes faits une liste des points clés du cours pour connaître les éléments à mobiliser dans cet exercice.

3.2 Structure Application

Tout d'abord, nous avons commencé par définir la structure `Application`. En effet, cette dernière sera mobilisée dans chacune des fonctions évoquées ci-dessus. Après lecture de l'énoncé, nous nous sommes aperçus qu'une application était ici définie par un nom, un path (chemin d'accès du fichier exécutable), un nombre d'arguments, une liste d'arguments, un pid (identifiant du processus qui exécute le programme) et un état de fonctionnement.

Ainsi, on a la structure suivante :

```
typedef struct Application
{
    char name[TAILLE_MAX_STRING];
    char path[TAILLE_MAX_STRING];
    int nb_arguments;
    char arguments[NB_MAX_ARG][TAILLE_MAX_STRING];
    pid_t pid;
    int en_marche;
} Application;
```

3.3 Pseudo code pour la lecture du fichier

Comme évoqué précédemment, la première étape de cet exercice est de parvenir à parser le fichier. Pour cela, nous avons défini la fonction `lecture_fichier()`. Celle-ci lit caractère par caractère le fichier, et fait appel à une fonction `convertir()` qui convertir une chaîne en `chaîne1=chaîne2`. Ainsi, on peut aisément copier la `chaîne2`, qui correspond à la valeur de l'argument, à l'espace dédié dans la structure `Application`.

Pour lire le fichier, nous avons été contraints de le faire à la main, caractère

par caractère. En effet, nous avons d'abord commencé à coder sous Windows pour bénéficier des fonctionnalités avancées de notre IDE. En raison de l'encodage du fichier, avec les fonctions de la librairie `<string.h>`, cela ne fonctionnait que sous Windows. Plus précisément, nous nous sommes rendus compte de la présence de `"\r\n"` à la fin de chaque ligne sous Linux. Ceci nous a posé un problème au début car nous ne comprenions pas pourquoi la lecture du fichier fonctionnait sous Windows mais pas sous Linux. Après recherche, nous avons pris conscience de la présence des `"\r"`, ce qui a motivé notre choix de lire à la main.

3.4 Fonction main

La fonction `main()` de `ApplicationManager.c` doit effectuer les tâches suivantes :

- créer un ensemble de processus fils chacun est responsable à l'exécution d'une application
- lors de l'arrêt d'une application, informer l'utilisateur en lui affichant le nom de l'application terminée
- s'arrêter après avoir fermé toutes les applications en cours d'exécution lors de la réception d'un ordre de mise en veille de la part de `power_manager` (signal `SIGUSR1`)

Ces 3 tâches représenteront les 3 étapes clés de notre raisonnement pour la fonction `main()`.

3.4.1 Lancement des applications

Ainsi, nous avons décidé de commencer par faire appel à `lecture_fichier()` pour stocker les informations relatives aux applications dans un tableau d'application préalablement créé. Ensuite, pour chaque application, nous utiliserons les connaissances des questions précédentes. Ainsi, nous appellerons pour chaque application la fonction `fork()` afin de créer un processus fils, et au sein de celui-ci, on appellera `execv()` pour lancer l'application.

3.4.2 Gestion de la fermeture d'une application

Ensuite, pour pouvoir indiquer quand une application se ferme, nous avons décidé d'implémenter une boucle `while`. Ainsi, tant qu'une application est toujours en état de marche, on regarde si l'état des processus fils (et donc des applications lancées) a changé à l'aide de `waitpid()`. Afin que les fils fonctionnent en parallèle, nous avons cherché quelle option nous permettrait cette fonctionnalité. C'est le cas de l'option `WNOHANG` vue en cours qui doit être appelée comme suit :

```
pid = waitpid(tab_app[i].pid, &statut, WNOHANG);
```

Si pid vaut -1, alors le programme a rencontré une erreur lors de l'attente de la fin du processus fils. Si pid vaut 0, aucun changement d'état n'a été observé. Enfin, si pid est supérieur à 0, alors cela signifie que le processus fils a changé d'état, c'est-à-dire, qu'il s'est terminé.

3.4.3 Fermeture du programme principal suite au signal SIGUSR1

Enfin, nous devons rendre notre programme capable de gérer convenablement les signaux SIGUSR1. C'est pourquoi nous avons défini de redéfinir la gestion des signaux à l'aide d'un handler. Celui-ci se fait au travers de la redéfinition de la structure sigaction standard.

Cette nouvelle structure sigaction doit être en capacité de masquer tous les signaux (à l'exception de SIGQUIT et SIGUSR1). Egaleme nt, elle doit permettre lors de la réception d'un signal SIGUSR1 de faire appel à une fonction arret(), chargée de l'arrêt complet des applications.

Ainsi, nous avons la structure suivante :

```
struct sigaction handler_app_manager;
// Masquage de tous les signaux
if (sigfillset(&handler_app_manager.sa_mask) != 0)
{
    perror("sigfillset");
    exit(EXIT_FAILURE);
}

// Sauf SIGQUIT (controle c)
if (sigdelset(&handler_app_manager.sa_mask, SIGQUIT) != 0)
{
    perror("sigdelset");
    exit(EXIT_FAILURE);
}

// Siginfo pour connaître le nom du processus émetteur d'un signal
handler_app_manager.sa_flags = SA_SIGINFO;

// Redéfinition de la fonction appliquer en cas de signal
handler_app_manager.sa_sigaction = arret;

// Redéfinition du comportement de SIGUSR1
if (sigaction(SIGUSR1, &handler_app_manager, NULL) != 0)
{
    perror("sigaction");
    exit(EXIT_FAILURE);
}
```

Nous avons également opté pour la mise en place de drapeaux (flags). La présence de SA_SIGINFO permet d'obtenir des informations, entre autres, sur l'émetteur du signal. En effet, il est demandé que ce comportement de SIGUSR1 se déclenche uniquement si c'est le programme power_manager qui lui a envoyé.

Ainsi, dans la fonction `arret()`, nous pouvons gérer ceci de la manière suivante :

```
void arret(int sig, siginfo_t *info, void *context)
{
    int i;
    // On verifie que c'est le bon signal envoye par le bon emetteur
    if (sig == SIGUSR1 && info->si_pid == pid_power_manager)
    {
        printf("Fermeture de toutes les applications\n");
        // Pour chaque application
        for (i = 0; i < nb_app; i++)
            // Si elle est en etat de marche
            if (tab_app[i].en_marche == 1)
            {
                // On met fin a son fonctionnement
                if (kill(tab_app[i].pid, SIGUSR1) != 0)
                {
                    perror("kill");
                    exit(EXIT_FAILURE);
                }
                // MAJ attributs
                printf("L'application %s, executee par le processus %d\n", tab_app[i].name, tab_app[i].pid);
                tab_app[i].en_marche = 0;
                tab_app[i].pid = -1;
            }
        exit(EXIT_SUCCESS);
    }
}
```

3.5 Demande de mise en veille de power_manager

Enfin, pour rendre notre programme opérationnel, il ne nous reste plus qu'à modifier le code de `power_manager.c` afin qu'il envoie un signal `SIGUSR1` lorsqu'il lit un "1" dans le fichier `mise_en_veille.txt`. Ainsi, on a :

```
if (c == '1')
{
    // Mise en veille
    printf("[power manager] Mise en veille en cours ...\n");
    fp = fopen(argv[1], "w");
    fputs("0", fp);
    fclose(fp);
    // Envoi d'un signal SIGUSR1 au pere
    if (kill(parent_pid, SIGUSR1) == -1) exit(EXIT_FAILURE);
}
```

Le code étant déjà en grande partie rédigée, il ne nous restait plus qu'à rajouter l'instruction `kill()`, qui permet d'envoyer le signal indiqué en paramètre au processus indiqué en paramètre. Ainsi, on envoie au père (le programme principal) un signal `SIGUSR1` dès qu'un "1" est observé dans le fichier `mise_en_veille.txt`. Nous avons veillé à gérer les potentielles erreurs à l'aide d'une structure `if` qui vérifie si le

programme est parvenu à envoyer le signal au processus demandé.

Conclusion

Ainsi, ce TP a été pour nous l'occasion de mettre en pratique les connaissances théoriques apportées par les cours magistraux. En particulier, ce TP nous a permis de maîtriser la création de processus avec l'instruction `fork()`, l'appel d'un programme avec `execv()`, le fonctionnement de Git et la gestion des signaux avec la structure `sigaction`. Ce devoir a été pour nous l'opportunité de revoir la lecture des fichiers et de comprendre l'importance des caractères non imprimables tels que les retours chariots.