

# Rapport Devoir 2



DEVOIR 2 : PROGRAMMATION SYSTÈME

# Table des matières

<b>Introduction</b>	<b>2</b>
<b>1 Exercice 1</b>	<b>3</b>
<b>2 Exercice 2</b>	<b>4</b>
2.1 Analyse du sujet . . . . .	4
2.2 Partie 1 : fonctionnement du programme . . . . .	4
2.2.1 Question 1 . . . . .	4
2.2.2 Question 2 . . . . .	4
2.2.3 Question 3 . . . . .	5
2.2.4 Question 4 . . . . .	5
2.3 Partie 2 : Implémentation du programme . . . . .	6
2.3.1 Question 1 . . . . .	6
2.3.2 Question 2 . . . . .	7
2.3.3 Question 3 . . . . .	7
2.4 Partie 3 : Modification, modularité et Makefile . . . . .	8
2.4.1 Question 1 . . . . .	8
2.4.2 Question 2 . . . . .	9
2.4.3 Question 3 . . . . .	9
2.4.4 Question 4 . . . . .	9
2.4.5 Question 5 . . . . .	9
2.4.6 Question 6 . . . . .	10
2.4.7 Question 7 . . . . .	11
<b>3 Exercice 3</b>	<b>12</b>
3.1 Analyse du sujet . . . . .	12
3.2 Préparation . . . . .	12
3.3 Récupération des informations . . . . .	12
3.3.1 Récupération dans le fichier . . . . .	12
3.3.2 Stockage de l'information . . . . .	13
3.3.3 Affichage de l'information . . . . .	13
3.4 Lancement des applications . . . . .	14
3.4.1 app_manager() . . . . .	14
3.4.2 my_system() . . . . .	14
3.5 Arrêt des applications . . . . .	14
3.5.1 Affichage du nom à la fin . . . . .	14
3.5.2 Veille et arrêt des applications infinies . . . . .	17
3.6 Envoi du signal et modification de <i>power_manager.c</i> . . . . .	18
<b>Conclusion</b>	<b>20</b>

## Introduction

Nous avons été amenés à travailler sur le premier DM de SR01 lors de la première semaine des vacances de Noël 2022. Ce second DM est une mise en pratique des notions vues en cours sur la seconde et la troisième partie. Il concerne exclusivement le langage C et les possibilités que ce langage permet pour la programmation système. Le sujet de ce second DM est disponible [ici](#).

Ce rapport contient les explications des fonctions présentes dans les différents dossiers du rendu (exercice-1, exercice-2 et exercice-3). Des commentaires ont été laissés dans le code afin de mieux l'expliquer. De plus, nous allons également expliciter nos choix vis-à-vis du sujet et notre compréhension de l'énoncé.

Les différentes fonctions commentées sont présentes dans les dossiers suivants :

- *exercice-1* : pour l'exercice 1 permettant d'afficher l'arbre demandé
- *exercice-2* : contenant les éléments de l'exercice 2, tous présents dans le dossier local du clone GitHub, sous la hiérarchie suivante :
  - le fichier initial Prog\_premiers.c et ses trois améliorations : Prog\_premiers\_m1.c, Prog\_premiers\_m2.c et Prog\_premiers\_m3.c
  - decoupage\_module : contenant les éléments de la partie 3, découpé par module et présence de CMake.mak et CMakeOpt.mak
- *exercice-3* : contenant les éléments de l'exercice 3, en particulier le fichier ApplicationManager.c

# 1 Exercice 1

Dans le cadre de cet exercice, nous avons estimé que tout écrire dans la fonction *main()* était justifié au vu de la faible quantité de code.

Le processus principal commence par afficher son propre PID et le PID de son propre processus père avec la fonction *getpid()* et *getppid()* respectivement. Ensuite, le processus principal met en pause l'exécution du programme pendant une seconde avec la fonction *sleep(1)*.

Ensuite, le programme entre dans une boucle qui itère 4 fois (on voit sur le dessin du sujet que le processus principal donne 4 fils). À chaque itération de la boucle, le processus principal crée un nouveau processus en appelant la fonction *fork()*. Si la valeur renvoyée par *fork()* est -1, cela signifie qu'une erreur est survenue lors de la création du processus et le programme affiche un message d'erreur et se termine avec *exit(1)*. Si la valeur renvoyée par *fork()* est 0, cela signifie que le processus courant est le processus fils créé par le processus père et le programme affiche son propre PID et le PID de son processus père. Si la valeur renvoyée par *fork()* est différente de 0, cela signifie que le processus courant est le processus père et il attend la fin du processus fils avec la fonction *wait(0)*.

L'astuce pour donner l'arborescence des processus est de ne pas faire de *exit()* ou de *return* à la fin des instructions du processus fils, faisant que le fils continuera lui aussi à faire des itérations de boucles. De cette façon, chaque fils fera une itération de boucle de moins que le fils précédent (car la boucle du père aura avancé d'une itération), et le premier fils fera une itération de moins que son père. Cela correspond au schéma donné.

Vous pouvez remarquer cependant que les processus ne sont pas exécutés dans le même ordre que celui des nombres indiqués, cela est en effet une tâche complexe : prenons le processus (8) suivi du (9) par exemple, comment faire en sorte que le processus créé après le (8) soit le fils du (3), et non le premier fils du (6) ? Pour cela, il faut que (6) ait un moyen d'attendre que les processus (3) et (4) aient donné leurs fils, sauf que ce processus n'a pas accès à cette information, vu qu'ils ne sont reliés que par le processus (1). Deux méthodes pourraient être envisagées pour respecter l'ordre :

- ajouter des délais (*sleep()*) de plus en plus grands en fonction des étages pour être sûr que l'étage du dessus ait fini avant de passer à l'étage suivant
- établir une communication complexe entre les processus, grâce à des pipes ou grâce à des signaux

La première solution n'a pas été implémentée, car elle est non propre et non fiable (des délais ne nous garantissent pas absolument l'ordre d'exécution).

La deuxième solution n'a pas été implémentée puisque nous n'avons pas vu en SR01 une façon de faire de la communication aussi poussée entre *n* processus. De plus, la communication entre processus n'est pas l'objet de cet exercice.

Nous en sommes donc arrivés à la conclusion que l'ordre ne devait pas avoir d'importance dans le cadre de cet exercice.

## 2 Exercice 2

### 2.1 Analyse du sujet

Les objectifs de cette partie sont multiples et permettent de couvrir l'ensemble des notions de la partie 2 du cours. En effet, cette partie aborde les points suivants :

- Git
- makefile
- notions systèmes en C : `fork()`, `system()`, `wait()`, ...
- quelques commandes shell (début de la partie 3 du cours)

### 2.2 Partie 1 : fonctionnement du programme

#### 2.2.1 Question 1

Pour représenter l'arbre des processus, nous avons choisi d'utiliser différentes couleurs, dont voici la légende :

- rouge : shell
- bleu : processus père
- violet : les fils directs du processus père
- jaune : les fils directs des processus violets
- vert : les processus créés par les processus jaunes lors de l'appel à la fonction `system()`

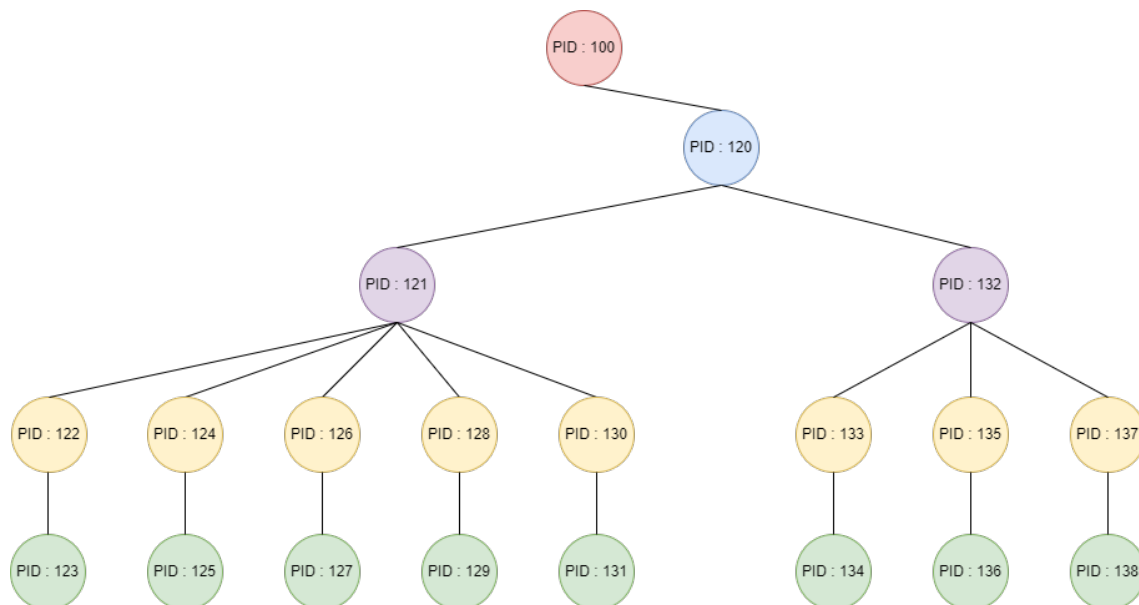


FIGURE 1 – Arbre des processus

#### 2.2.2 Question 2

Dans cette partie, nous allons expliquer pas à pas comment fonctionne le programme précédent. Le code est réparti en trois fonctions :

- `premier(int nb)` : qui vérifie si un nombre est premier
- `explorer(int debut, int fin)` : qui va tester si les nombres entre premier et fin sont premiers
- `main()` : qui va lancer l'exploration pour les nombres de 2 à 21

Le processus de pid 100 (shell) va lancer le programme (pid : 120). Au début du programme, la fonction `main` est exécutée, il affecte à `grp` la valeur 1 et va rentrer dans la boucle `while (1 <= 11 == true)`. Une fois dans la boucle `while`, la fonction `main` va appeler la fonction `explorer(1+1, 1+10)` soit `explorer(2, 11)`. Nous sommes encore pour l'instant avec le processus de pid 120.

Une fois dans cette boucle, la fonction `explorer` va effectuer un premier `fork()` qui va créer le processus fils de pid 121 (et de ppid 120). Ce processus fils va itérer entre 2 et 11 afin de chercher tous les nombres premiers. Si un nombre premier est trouvé, la fonction `explorer` va créer un nouveau processus fils (les processus jaunes du schéma précédent) qui va écrire dans le fichier `nbr_premiers.txt` à l'aide de la fonction `system`. La fonction `system` va créer un processus afin d'effectuer l'écriture dans le fichier (équivalent à la rangée verte sur le schéma). Cependant, lors de l'écriture dans le fichier, le pid du processus père va être écrit et non celui du processus qui exécute la commande (celui créé par la fonction `system`). Le processus fils va se terminer avec `exit(0)` ;

Les différents processus père attendent la fin des processus fils à l'aide de : `wait(&etat)` ;

Une fois la boucle `for` de la fonction `explorer` finie, la fonction `main` va affecter à `grp` la valeur 11 (1+10). Elle va, de ce fait, à nouveau rentrer dans la boucle `while (1 <= 11 == True)` et appeler la fonction `explorer(12, 21)` (et répéter les instructions citées ci-dessus).

### 2.2.3 Question 3

Dans le cas où les lignes 41 et 46 sont supprimées, les processus pères n'attendent plus les processus fils. Nous passons d'une exécution des processus en série à une exécution en parallèle. Ainsi, l'affichage dans le fichier `nbr_premiers.txt` peut se faire dans le désordre rendant sa lecture plus difficile. Le fait de ne pas attendre ses fils peut amener à la création de processus zombies ou orphelins.

### 2.2.4 Question 4

Ce programme garantit la synchronisation en utilisant la fonction `wait()` qui permet d'attendre la fin des processus fils pour poursuivre l'exécution du processus père.

Lorsque le processus fils exécute la commande "echo" avec la fonction `system()`, il attend que la commande se termine avant de se terminer lui-même. Le processus père utilise ensuite la fonction `wait()` pour attendre qu'il finisse.

Ainsi, chaque processus fils ne peut être exécuté que lorsque le processus fils précédent a fini. Cela garantit que chaque processus fils s'exécute de manière séquentielle et que le processus de PID `p` ne peut être exécuté qu'après la fin de l'exécution du processus de PID `p-1`. Cela est aussi valable pour les processus fils créés dans différentes itérations de la boucle `for`. On ne peut passer à l'itération suivante que lorsque le père a terminé, et comme il attend son fils, quand celui-ci a également terminé.

## 2.3 Partie 2 : Implémentation du programme

### 2.3.1 Question 1

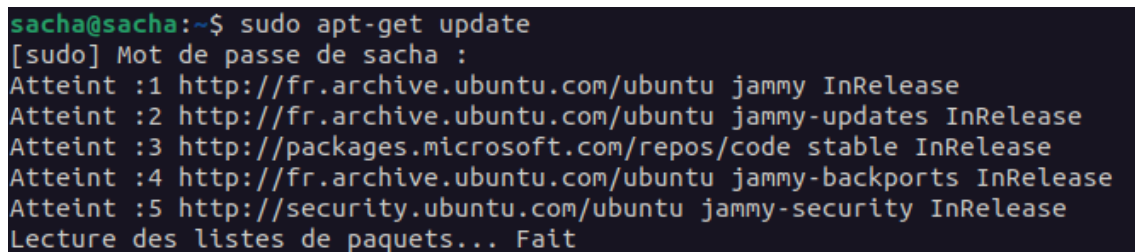
Afin d'installer Git sur Linux, il suffit de taper les commandes suivantes :

```
$ sudo apt-get update
$ sudo apt-get install git-all
$ git --version
```

La première commande va nous assurer que nous installons bien la dernière version du paquet, ainsi que toutes les dépendances qui peuvent être nécessaires. La seconde commande va installer le paquet Git. Enfin, la dernière commande va nous permettre de vérifier le bon déroulement de l'installation en nous retournant la version de Git installée.

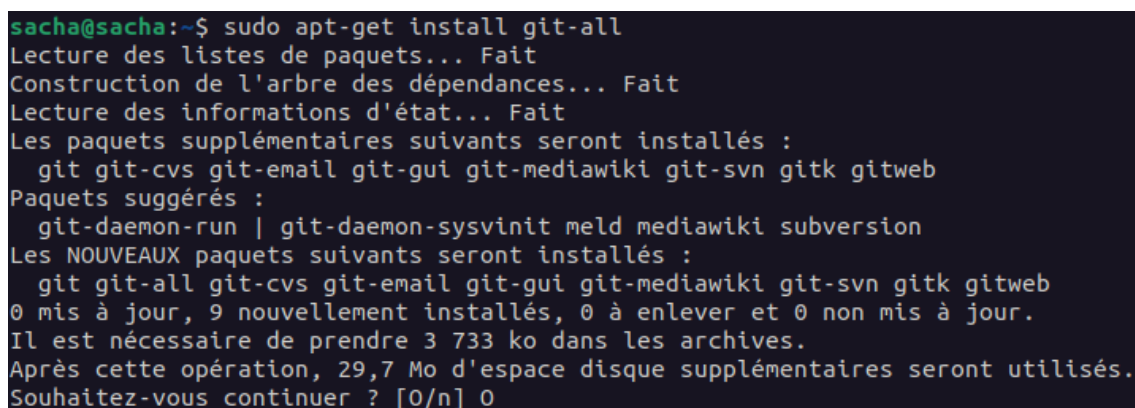
Lors de l'exécution de la première requête, le mot de passe de l'utilisateur courant est demandé afin d'installer Git, car nous utilisons le préfixe sudo qui exécute la commande en tant que root.

Voici nos résultats obtenus sur les requêtes précédentes :



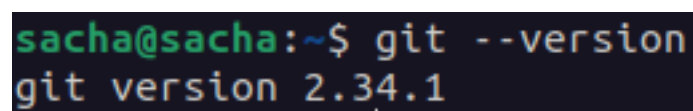
```
sacha@sacha:~$ sudo apt-get update
[sudo] Mot de passe de sacha :
Atteint :1 http://fr.archive.ubuntu.com/ubuntu jammy InRelease
Atteint :2 http://fr.archive.ubuntu.com/ubuntu jammy-updates InRelease
Atteint :3 http://packages.microsoft.com/repos/code stable InRelease
Atteint :4 http://fr.archive.ubuntu.com/ubuntu jammy-backports InRelease
Atteint :5 http://security.ubuntu.com/ubuntu jammy-security InRelease
Lecture des listes de paquets... Fait
```

FIGURE 2 – commande : sudo apt-get update



```
sacha@sacha:~$ sudo apt-get install git-all
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances... Fait
Lecture des informations d'état... Fait
Les paquets supplémentaires suivants seront installés :
  git git-cvs git-email git-gui git-mediawiki git-svn gitk gitweb
Paquets suggérés :
  git-daemon-run | git-daemon-sysvinit meld mediawiki subversion
Les NOUVEAUX paquets suivants seront installés :
  git git-all git-cvs git-email git-gui git-mediawiki git-svn gitk gitweb
0 mis à jour, 9 nouvellement installés, 0 à enlever et 0 non mis à jour.
Il est nécessaire de prendre 3 733 ko dans les archives.
Après cette opération, 29,7 Mo d'espace disque supplémentaires seront utilisés.
Souhaitez-vous continuer ? [0/n] 0
```

FIGURE 3 – commande : sudo apt-get install git-all



```
sacha@sacha:~$ git --version
git version 2.34.1
```

FIGURE 4 – commande : git --version

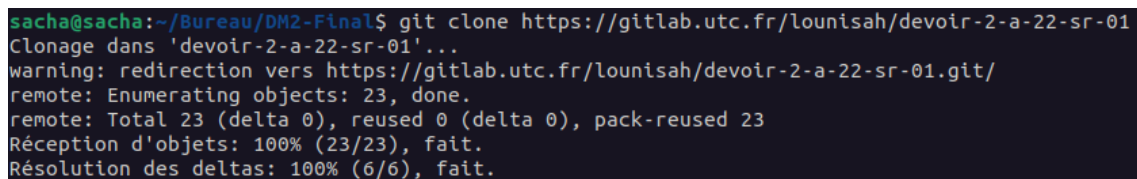
### 2.3.2 Question 2

Dans le but de cloner le dépôt distant présent sur ce [lien](#). Les commandes à exécuter sont les suivantes :

```
$ git clone https://gitlab.utc.fr/lounisah/devoir-2-a-22-sr-01
$ cd devoir-2-a-22-sr-01
$ git status
$ ls
```

La première commande va cloner le dépôt distant Git présent sur l'URL fourni en paramètre. La seconde commande va nous déplacer dans le dossier créé. Les deux dernières commandes nous permettent de nous assurer de la conformité de la copie locale. Ainsi, la troisième commande nous assure que notre copie locale est à jour avec le dossier distant et la quatrième va lister tous les fichiers présents dans le dossier.

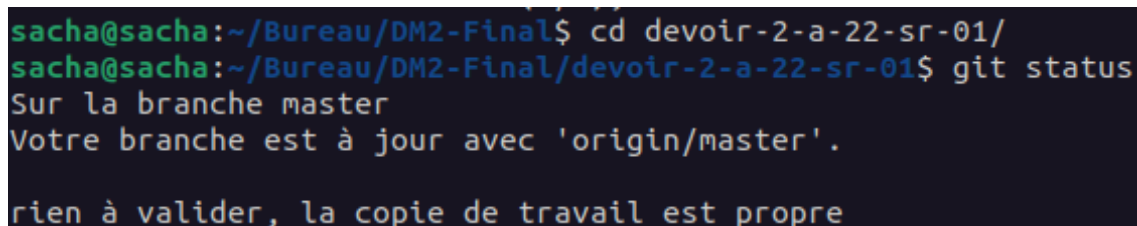
Voici le résultat que nous obtenons sur le terminal lorsque nous exécutons ces commandes :



```
sacha@sacha:~/Bureau/DM2-Final$ git clone https://gitlab.utc.fr/lounisah/devoir-2-a-22-sr-01
Clonage dans 'devoir-2-a-22-sr-01'...
warning: redirection vers https://gitlab.utc.fr/lounisah/devoir-2-a-22-sr-01.git/
remote: Enumerating objects: 23, done.
remote: Total 23 (delta 0), reused 0 (delta 0), pack-reused 23
Réception d'objets: 100% (23/23), fait.
Résolution des deltas: 100% (6/6), fait.
```

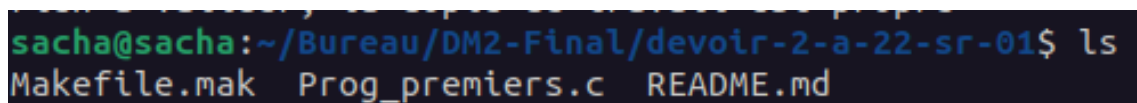
FIGURE 5 – commande : git clone

Puis, nous vérifions la bonne installation du dépôt distant :



```
sacha@sacha:~/Bureau/DM2-Final$ cd devoir-2-a-22-sr-01/
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
rien à valider, la copie de travail est propre
```

FIGURE 6 – commandes : cd + git status



```
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ ls
Makefile.mak Prog_premiers.c README.md
```

FIGURE 7 – commande : ls

### 2.3.3 Question 3

Le résultat du programme présent sur Git est présent dans le fichier `nbr_premiers.txt` qui contient tous les nombres premiers entre 2 et 21, écrit sous le format suivant :

```
2 est un nombre premier          écrit par le processus 6301
```



Le programme est le même que celui expliqué en partie 1. Nous allons, dans cette partie, étudier les différentes fonctions et non pas les PID des processus. Nous avons dans le programme les fonctions suivantes :

- `explorer(int debut, int fin)` : qui va appeler les fonctions suivantes :
  - `fork()` : permet de créer un processus fils qui va itérer sur les nombres entre `debut` et `fin`. Appeler `premier()` pour vérifier si le nombre est premier, et si c'est le cas, faire deuxième appel à `fork` pour que le processus fils écrive dans le fichier
  - `int premier(int nb)` : qui vérifie si un nombre est premier et retourne 0 (faux) ou 1 (vrai)
  - `fflush(stdout)` : va permettre de forcer l'affichage sur le flux `stdout`
  - `sprintf(char *chaine_videe, const char *chaine_format, ...)` : va permettre de mettre la chaîne formatée dans les adresses mémoires pointées par `chaine_videe`
  - `system(chaine)` : permet de lancer l'exécution d'une commande, sans recouvrement, sur le système, et va pour cela créer un processus.
  - `sleep(1)` : permet au programme de s'arrêter pendant 1 seconde
  - `exit(0)` : permet de mettre fin au programme en spécifiant un code de retour, ici 0
  - `wait()` : permet d'attendre la fin du processus fils
- `main` : qui va appeler la fonction `explorer`

Concernant la chaîne de caractère `bash` qui va permettre d'écrire dans le fichier `nbr_premiers.txt`, nous allons la décomposer :

- `echo` :
- `'%d est un nombre premier`  
   écrit par le processus `%d` : contient la chaîne dans laquelle les `%d` vont être remplacé par les valeurs de `i` et du pid du processus (celui qui appelle `system` et non pas celui de `system` qui écrit dans le fichier) à l'aide de la fonction `sprintf`
- `»` : permet d'écrire à la fin du fichier, spécifié à droite s'il existe et le créé sinon

## 2.4 Partie 3 : Modification, modularité et Makefile

### 2.4.1 Question 1

Non, ce ne sont pas les bons PID qui sont inscrits dans le fichier. En effet, les PID inscrit dans le fichier sont ceux des processus qui appellent la fonction `system` et non ceux de l'exécution de `system`.

Pour corriger cela, nous allons utiliser la commande `bash` : `$$`, qui retourne le PID du script courant. Ainsi, la chaîne créée dans la fonction `sprintf` devient :

```
sprintf(chaine,"echo ' %d  est un nombre premier \
PID processus qui appelle system : %d; \
PID processus qui exécute la boucle for : %d; \
PID processus créé par system : '$$>>nbr_premiers.txt",
i,getpid(), getppid());
```

### 2.4.2 Question 2

Cette partie est faite dans le fichier *Prog\_premiers\_m2.c*

### 2.4.3 Question 3

Cette partie est faite dans le fichier *Prog\_premiers\_m3.c*

### 2.4.4 Question 4

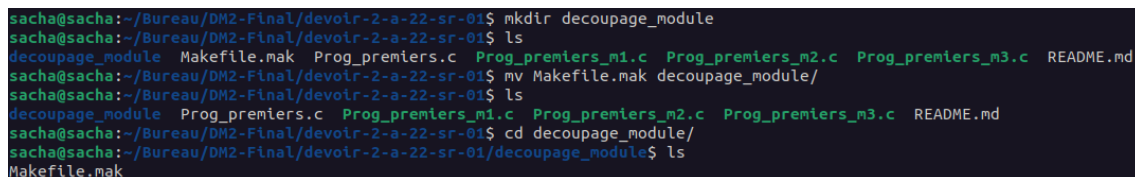
Cette partie est faite dans le dossier *decoupage\_programme*.

### 2.4.5 Question 5

Au préalable, nous avons créé le dossier *decoupage\_module* et avons déplacé le fichier *Makefile.mak*, à l'aide des commandes suivantes :

```
$ mkdir decoupage_module
$ ls
$ mv Makefile.mak decoupage_module
$ ls
$ cd decoupage_module
$ ls
```

Les appels à la commande *ls* nous permettent de nous assurer que le dossier est bien créé et le fichier bien déplacé dedans.



```
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ mkdir decoupage_module
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ ls
decoupage_module  Makefile.mak  Prog_premiers.c  Prog_premiers_m1.c  Prog_premiers_m2.c  Prog_premiers_m3.c  README.md
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ mv Makefile.mak decoupage_module/
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ ls
decoupage_module  Prog_premiers.c  Prog_premiers_m1.c  Prog_premiers_m2.c  Prog_premiers_m3.c  README.md
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ cd decoupage_module/
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01/decoupage_module$ ls
Makefile.mak
```

FIGURE 8 – initialisation dossier *decoupage\_module*

Une fois, que nous avons réalisé les commandes précédentes, nous avons exécuté les commandes suivantes afin de créer le fichier exécutable :

```
$ make -f Makefile.mak
$ ./Prog_premiers
$ make -f Makefile.mak clean
```

La commande *git make* va permettre de créer le fichier objet et le fichier exécutable *Prog\_premiers*. Nous allons pouvoir lancer cette exécutable à l'aide de la commande *./Prog\_premiers*. Enfin, nous supprimons les fichiers objets et l'exécutable à l'aide de l'appel à *clean* présent dans le *Makefile*.

```
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01/decoupage_module$ make -f Makefile.mak
gcc -c main.c
gcc -c explorer.c
gcc -c premier.c
gcc -c my_system.c
gcc main.o explorer.o premier.o my_system.o -o Prog_premiers
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01/decoupage_module$ ./Prog_premiers
Un processus fils a terminé (PID = 5999)
Un processus fils a terminé (PID = 6003)
Un processus fils a terminé (PID = 6001)
Un processus fils a terminé (PID = 6000)
Un processus fils a terminé (PID = 6004)
Un processus fils a terminé (PID = 6010)
Un processus fils a terminé (PID = 6011)
Un processus fils a terminé (PID = 6012)
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01/decoupage_module$ make -f Makefile.mak clean
rm my_system.o premier.o explorer.o main.o Prog_premiers
```

FIGURE 9 – commandes : make, ./Prog\_premiers, clean

### 2.4.6 Question 6

Cette partie est faite dans le dossier *MakefileOpt*. Les commandes sont les mêmes que pour le fichier *Makefile.mak*, à savoir :

```
$ make -f MakefileOpt.mak
$ ./Prog_premiers
$ make -f MakefileOpt.mak clean
```

Les simplifications que nous avons faites sont basées sur le makefile optimisé que nous avons vu en cours. En effet, nous avons trouvé une similitude entre les deux cas, car ici des fichiers *.c* dépendent de fichiers *.h* qui ne porte pas le même nom. Nous avons également trouvé cette version plus lisible puisqu'elle se base sur les valeurs prises par défaut par les fichiers makefiles. Ainsi, les modifications réalisées sont les suivantes :

- création des variables suivantes :
  - CC : permet de spécifier le compilateur à utiliser et de pouvoir facilement le changer
  - OBJS : transforme le *.c* des fichiers de SOURCES en *.o*
- les dépendances permettent de spécifier pour chaque fichier objet les fichier headers dont ils dépendent en plus du fichier source et header portant le même nom
- clean : permet de supprimer tous les fichiers objets (extension ".o") et le fichier exécutable (défini par la variable \$(EXEC)) dans le répertoire courant

```
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01/decoupage_module$ make -f MakefileOpt.mak
gcc -c -o main.o main.c
gcc -c -o explorer.o explorer.c
gcc -c -o premier.o premier.c
gcc -c -o my_system.o my_system.c
gcc -o Prog_premiers main.o explorer.o premier.o my_system.o
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01/decoupage_module$ ./Prog_premiers
Un processus fils a terminé (PID = 7383)
Un processus fils a terminé (PID = 7384)
Un processus fils a terminé (PID = 7385)
Un processus fils a terminé (PID = 7386)
Un processus fils a terminé (PID = 7382)
Un processus fils a terminé (PID = 7394)
Un processus fils a terminé (PID = 7395)
Un processus fils a terminé (PID = 7396)
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01/decoupage_module$ make -f MakefileOpt.mak clean
rm -rf *.o Prog_premiers
```

FIGURE 10 – commandes, version 2 : make, ./Prog\_premiers, clean

### 2.4.7 Question 7

Les commandes Git à réaliser sont les suivantes :

```
$ git add .
$ git commit -m "MON MESSAGE"
```

La commande `git add .` permet de mettre à jour de tous les fichiers (qui ne sont pas listés dans le fichier `.gitignore`, s'il existe) du dépôt entier. La commande suivante permet de commit son travail (le valider juste avant l'envoi avec `git push`) l'option `-m` permet de spécifier un message ici "MON MESSAGE".

Pour effectuer ses commandes, il faut au préalable avoir spécifié son email et son nom avec les commandes :

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Enfin, afin d'être sûr de l'ajout et de la validation de son travail, nous pouvons utiliser la commande suivante :

```
$ git status
```

Cette commande va nous permettre de voir l'état du répertoire de travail, ici le dossier `devoir-2-a-22-sr-01`.

Voici les différentes captures effectuées lors de l'application des commandes ci-dessus :

```
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ git config --global user.name "Sacha SZENDROVICS"
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ git config --global user.email sacha.szendrovics@etu.utc.fr
```

FIGURE 11 – commande : `git config`

```
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ git add .
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ git commit -m "J'ai FINI"
[master 9279818] J'ai FINI
12 files changed, 407 insertions(+)
create mode 100755 Prog_premiers_m1.c
create mode 100755 Prog_premiers_m2.c
create mode 100755 Prog_premiers_m3.c
rename Makefile.mak => decoupage_module/Makefile.mak (100%)
mode change 100644 => 100755
create mode 100755 decoupage_module/MakefileOpt.mak
create mode 100755 decoupage_module/explorer.c
create mode 100755 decoupage_module/explorer.h
create mode 100755 decoupage_module/main.c
create mode 100755 decoupage_module/my_system.c
create mode 100755 decoupage_module/my_system.h
create mode 100755 decoupage_module/premier.c
create mode 100755 decoupage_module/premier.h
sacha@sacha:~/Bureau/DM2-Final/devoir-2-a-22-sr-01$ git status
Sur la branche master
Votre branche est en avance sur 'origin/master' de 1 commit.
(utilisez "git push" pour publier vos commits locaux)

rien à valider, la copie de travail est propre
```

FIGURE 12 – commandes : `git add .` ; `git commit` ; `git status`

## 3 Exercice 3

### 3.1 Analyse du sujet

Cette partie sert à revoir des notions vues en cours, telles que :

- fork()
- exec
- gestion de signaux
- lecture dans un fichier
- structures

### 3.2 Préparation

Tout d'abord, pour *list\_appli.txt*, nous remarquons que les applications à exécuter sont sous la forme *./nom\_application*, nous en déduisons donc qu'avant de commencer l'exercice, il faut d'abord compiler les différentes applications fournies :

```

/SR01-DM2/exercice-3$ gcc -o power_manager power_manager.c
/SR01-DM2/exercice-3$ gcc -o get_time get_time.c
/SR01-DM2/exercice-3$ gcc -o network_manager network_manager.c

```

FIGURE 13 – Compilation des applications

### 3.3 Récupération des informations

#### 3.3.1 Récupération dans le fichier

La première chose à faire est de récupérer les informations contenues dans le fichier *list\_appli.txt*. Cette récupération se fait dans la fonction *parse()*. Cette fonction lit un fichier dont le nom est passé en paramètre, et attend un fichier dont la première ligne est

```
nombre_applications=n
```

Suivi de n applications séparés d'une ligne vide sous la forme :

```

name=App name
path=./app
nombre_arguments=m
arguments=
arg_1
arg_m

```

À noter que les noms doivent être écrits correctement et qu'il ne doit pas y avoir d'espace autour du "=". Il ne doit pas y avoir plusieurs ou pas de retour à la ligne

entre les applications, voici l'exemple avec lequel nous allons traiter dans le cadre de cet exercice :

```
nombre_applications=3
name=Power Manager
path=./power_manager
nombre_arguments=2
arguments=
./mise_en_veille.txt
4

name=Get Time
path=./get_time
nombre_arguments=0
arguments=

name=Network Manager
path=./network_manager
nombre_arguments=0
arguments=
```

FIGURE 14 – list\_appli.txt

### 3.3.2 Stockage de l'information

Pour stocker l'information de manière efficace, nous avons décidé de la stocker dans des structures, ainsi chaque application est représentée par une structure définie comme suit :

```
typedef struct {
    char name[MAX_LINE_LENGTH];
    char path[MAX_LINE_LENGTH];
    int num_arguments;
    char **arguments;
} application;
```

FIGURE 15 – struct application

La liste d'applications est quant à elle stockée dans un tableau d'applications dans une variable globale.

### 3.3.3 Affichage de l'information

Maintenant que nous avons récupéré l'information, nous pouvons nous assurer de l'avoir fait correctement avec la fonction *affichage()*, qui affiche chaque application se trouvant dans le tableau d'applications.

## 3.4 Lancement des applications

### 3.4.1 `app_manager()`

La gestion du lancement des applications se fait dans la fonction `app_manager()`. Cette fonction se charge d'itérer sur le tableau d'applications afin d'exécuter leur commande pour chacune d'entre elles.

Dans la boucle `for` qui lance chaque fils, on ne met pas d'instruction `wait()`, ce qui permet le lancement de chaque application en parallèle.

À la sortie de la boucle `for`, toutes les applications sont maintenant lancées. Certaines d'entre elles se terminent d'elles-mêmes comme `./get_time`, tandis que d'autres sont des boucles infinies, comme `./power_manager`, ou `./network_manager`.

À la sortie de la boucle `for`, il faut donc deux choses pour le père :

- Des instructions qui récupèrent les fils qui se terminent d'eux-mêmes
- Une boucle infinie pour s'assurer que le père ne finisse pas avant ses fils (qui pour certains sont des boucles infinies).

Pour cela, nous procédons de cette manière :

```
while (1){
    pid_t pid_fils_termine = wait(0);
}
```

Ce code permettra au père de "récupérer" chacun de ses fils qui se terminent d'eux-mêmes, et une fois que ce sera fait, cette boucle maintiendra le père en vie afin d'éviter que ses fils ne deviennent zombies. L'arrêt des fils qui sont une boucle infinie est géré par des signaux que nous allons voir juste après.

### 3.4.2 `my_system()`

Certaines applications ayant besoin de paramètres et d'autres non, nous avons codé une fonction `my_system()` afin de simplifier le lancement des applications. Cette fonction prend juste l'indice de l'application dans le tableau d'applications, et la lance avec les bons arguments en appelant `execl()` ou `execv()` selon si l'application prend ou non des arguments. À noter que cette fonction sert juste à simplifier le code, elle ne fait pas de `fork()`. Le fait de ne pas faire `fork()` est important, cela sera nécessaire plus tard pour l'arrêt des applications, il faudra que dans l'application `./power_manager`, le `ppid` du processus soit directement celui du père dans `app_manager()`.

## 3.5 Arrêt des applications

### 3.5.1 Affichage du nom à la fin

De nombreuses solutions ont été envisagées pour afficher le nom d'une application lorsqu'elle s'arrête. En voici deux que nous avons essayé, suivi de celle que nous avons finalement choisie :

### Utilisation des codes de sortie (non choisie) :

Nous avons envisagé utiliser les codes de sortie pour afficher le nom d'une application lorsqu'elle se termine, ce qui aurait donné quelque chose comme ceci :

```
// Dans app_manager() :
for (int i = 0; i < num_applications; i++) {
    pid_t pid = fork();
    if (pid == 0) {
        my_system(i);
        exit(i);
    }
}

while (1){
    int status;
    pid_t pid = wait(&status);
    printf("L'application \"%s\" s'est terminée\n", applications[
WEXITSTATUS(status)]->name);
}
```

Cette solution fonctionnait partiellement, mais posait deux problèmes :

- Cela ne fonctionne que pour les applications se fermant d'elles-mêmes, les autres seront probablement tuées par un SIGINT lors d'un ordre de mise en veille, cette solution ne fonctionne donc pas pour toutes les applications.
- Pour récupérer le code de sortie *exit(i)*, il faut que le fils atteigne cette instruction, or, dans la version de *my\_system()* que nous avons, nous appelons *execv* ou *execl* sans faire de *fork()*, ce qui fait que le programme n'atteint jamais le *exit(i)*. Pour atteindre cette instruction, il aurait fallu exécuter les applications dans un autre processus fils, soit en appelant *system()*, soit en faisant un *fork()* dans notre *my\_system()*. Cependant, comme dit tout à l'heure, nous ne pouvons pas faire de *fork()* dans l'exécution de l'application, car cela empêcherait plus tard la fermeture des applications qui sont des boucles infinies.

Cette solution n'a donc pas été conservée.

### Utilisation de *atexit()* (non choisie) :

Nous avons envisagé d'utiliser *atexit()* pour afficher le nom d'une application lorsqu'elle se termine, ce qui aurait donné quelque chose comme ceci :



```

char nom[MAX_LINE_LENGTH]; // Variable globale

// ...

void afficher_nom() {
    printf("L'application \"%s\" s'est terminée\n", nom);
}

// ...

// Dans app_manager() :
for (int i = 0; i < num_applications; i++) {
    pid_t pid = fork();
    if (pid == 0) {

        strcpy(nom, applications[i] -> name);
        atexit(afficher_nom);

        my_system(i);
    }
}

while (1) {
    pid_t pid = wait(0);
}

```

Dans chaque fils, on attribue la variable globale "nom", affectation qui sera à chaque fois uniquement faite pour le processus fils courant.

Cette solution fonctionnait bien, elle était déjà mieux compatible avec les besoins de notre fonction *my\_system()*, mais avait un problème :

Cette solution, tout comme la dernière, ne fonctionnait qu'avec les applications se terminant d'elles-mêmes. En effet, *atexit()* ne fonctionne que quand un processus se termine naturellement, en appelant la fonction *exit()*. Cependant, les applications qui sont des boucles infinies n'appellent jamais *exit()*, elles sont terminées par un SIGINT lors de la mise en veille, ce qui ne déclenche pas le *atexit()*. Nous avons tenté deux techniques pour contourner le problème, sans succès :

- Utiliser *on\_exit()* plutôt que *atexit()*, qui d'après la documentation peut se déclencher sous plus de conditions.
- Installer un gestionnaire de signal qui remplace le handler par défaut du signal SIGINT par un simple appel à *exit(0)*.

Ces solutions se sont révélées non concluantes et la méthode du *atexit()* n'a donc pas non plus été choisie.

### Stocker les pid des applications (solution choisie) :

La solution que nous avons choisie est la suivante : nous conservons en mémoire dans des variables globales les pid des applications en cours d'exécution :

```
pid_t pid_power_manager;
pid_t pid_running_process[MAX_PROCESS];
int num_running_process = 0;
```

Nous conservons également le nombre d'applications en cours d'exécution, et aussi le pid du Power manager, ce qui nous permettra par la suite de n'arrêter les applications que si le signal provient du Power manager.

Pour rendre cela possible, nous créons deux fonctions :

- *add\_running\_process()* Qui ajoute un processus à la liste des processus d'applications en cours, et qui incrémente le nombre d'applications en cours
- *remove\_running\_process()* Qui retire un processus de la liste d'applications en cours, décrémente le nombre d'applications en cours et retire l'application du tableau de structures d'applications, le tout en libérant la mémoire dédiée à cette structure. Cette fonction se charge également d'afficher le nom de l'application qui vient de se terminer.

Il reste donc à ajouter les appels de ces fonctions aux bons endroits :

```
// Dans app_manager() :
for (int i = 0; i < num_applications; i++) {
    pid_t pid_fils = fork();
    if (pid_fils == 0) {
        printf("\nLancement de l'application \"%s\"...\n\n",
applications[i]->name);
        my_system(i);
    }
    else {
        sleep(1);
        if (strcmp(applications[i]->name, "Power Manager") == 0) {
            pid_power_manager = pid_fils;
        }
        add_running_process(pid_fils);
    }
}
while (1){
    pid_t pid_fils_termine = wait(0);
    remove_running_process(pid_fils_termine);
}
```

En plus d'ajouter les appels des fonctions, on enregistre le pid du Power manager lorsqu'il est exécuté.

Ce code affiche le nom des applications qui se terminent d'elles-mêmes. Pour les autres applications, nous verrons cela dans la partie suivante.

### 3.5.2 Veille et arrêt des applications infinies

Nous commençons par installer dans la fonction *main* un gestionnaire pour le signal SIGUSR1 :

```

struct sigaction S;
S.sa_sigaction = veille_handler;
sigemptyset(&S.sa_mask);
S.sa_flags = SA_SIGINFO;
if( sigaction(SIGUSR1, &S, NULL) != 0 ){
    perror("sigaction");
    exit(EXIT_FAILURE);
}

```

Dans le handler, on peut alors récupérer le pid du processus qui a envoyé le signal :

```

void veille_handler(int signum, siginfo_t *info, void *ptr){

    pid_t pid_emetteur = info->si_pid;

    if (pid_emetteur == pid_power_manager) {

        pid_t pid_fils_termine;

        while(num_running_process > 0){
            pid_fils_termine = pid_running_process[0];
            remove_running_process(pid_fils_termine);
            kill(pid_fils_termine, SIGINT);
        }

        free(applications);
        exit(0);
    }
}

```

Pour l'arrêt de l'application, on se contente d'envoyer un SIGINT à chaque application encore en cours, qui se trouve dans le tableau *pid\_running\_process*, et on appelle aussi *remove\_running\_process* pour chacun de ces pid, ce qui permettra de le retirer de la liste, d'afficher son nom et de libérer les ressources mémoire de la structure correspondante.

Une fois toutes les applications arrêtées, on libère le tableau de pointeurs d'applications et on quitte le programme.

### 3.6 Envoi du signal et modification de *power\_manager.c*

Pour envoyer le signal lorsque l'utilisateur replace 0 par 1 dans *mise\_en\_veill.txt*, il faut inclure *signal.h* et envoyer le signal SIGUSR1 au gestionnaire :

```
// autres inclusions
#include <signal.h>

void main (int argc , char *argv[]) {
    if (c == '1') {
        kill (getppid() , SIGUSR1);
    }
}
```

À noter que dans la fonction *main* ne se trouvent actuellement que le code changé, à savoir *kill(getppid(), SIGUSR1);*.

On n'oublie pas de recompiler *power\_manager.c* afin de faire un exécutable qui corresponde à la nouvelle version du code :

```
SR01-DM2/exercice-3$ gcc -o power_manager power_manager.c
```

C'est maintenant que ne pas faire de *fork()* dans *my\_system()* prend tout son sens, car en envoyant le signal au *ppid*, on envoie le signal au processus père, le même qui exécute la boucle *for* ou encore qui est responsable de l'exécution de la fonction *main*. C'est ce même processus qui contient dans ses variables globales contenant le *pid* du Power manager, et les *pid* des applications en cours, lors de l'envoi du signal, ce processus qui le reçoit pourra donc facilement arrêter toutes les applications.

Ne pas faire de *fork()* dans *my\_system()* à également un deuxième intérêt autre que d'envoyer le signal au bon processus : celui d'arrêter les bons processus. En effet, si on avait fait un *fork()* dans le *my\_system()*, les processus qu'on aurait enregistré dans la liste des processus en cours n'auraient pas été ceux des applications, mais de ceux des processus qui lancent les applications, on aurait donc tué le père et les applications seraient devenues des processus zombie si on tuait ces processus. De plus, on aurait enregistré le mauvais processus dans la variable globale *pid\_power\_manager*, la condition

```
if (pid_emetteur == pid_power_manager)
```

dans *veille\_handler()* n'aurait donc jamais pu être remplie et on n'aurait pas pu arrêter l'application.

Cette solution permet également d'envoyer un *SIGINT* pour mettre fin aux potentielles applications qui ne sont pas des boucles infinies, mais qui ne se sont pas encore terminées lors de la mise en veille. On évite également d'envoyer un *SIGINT* aux processus s'étant déjà terminés.

## Conclusion

Le second DM nous a permis de mettre en application les notions vues en cours et en TD sur des cas d'applications plus concrets. Nous avons utilisé de nombreuses fonctions vues en cours et en TD afin de répondre aux exigences de l'énoncé. Nous avons par ailleurs fait quelques choix dans le but de simplifier le bon déroulement de nos programmes. Ce second devoir était très enrichissant et très utile dans nos révisions des différentes parties de cours. Nous avons, en effet, pu appliquer de manière concrète les notions des parties 2 et 3 du cours.

En effet, nous avons grâce à ce DM pu perfectionner les différents éléments suivants :

- compréhension des fork et des hiérarchies de processus
- utilisation d'un logiciel de gestion de versions, Git
- compréhension des notions de processus zombies et orphelins
- appréhension des appels système et des différentes fonctions pour cela : `system`, `execv`
- application et simplification des MakeFile
- révisions sur les structures
- gestion et traitement des signaux
- lecture/écriture dans des fichiers

## Table des figures

1	Arbre des processus . . . . .	4
2	commande : sudo apt-get update . . . . .	6
3	commande : sudo apt-get install git-all . . . . .	6
4	commande : git --version . . . . .	6
5	commande : git clone . . . . .	7
6	commandes : cd + git status . . . . .	7
7	commande : ls . . . . .	7
8	initialisation dossier decoupage_module . . . . .	9
9	commandes : make, ./Prog_premiers, clean . . . . .	10
10	commandes, version 2 : make, ./Prog_premiers, clean . . . . .	10
11	commande : git config . . . . .	11
12	commandes : git add .; git commit; git status . . . . .	11
13	Compilation des applications . . . . .	12
14	list_appli.txt . . . . .	13
15	struct application . . . . .	13