Chapitre 1 – Processus :

Exercice 1.1 :

Le but de cet exercice est de créer un programme qui lancera un processus puis s'endormira juste après. De cette façon, on essaiera de constater que le processus fils devient un processus zombie.

Code source :

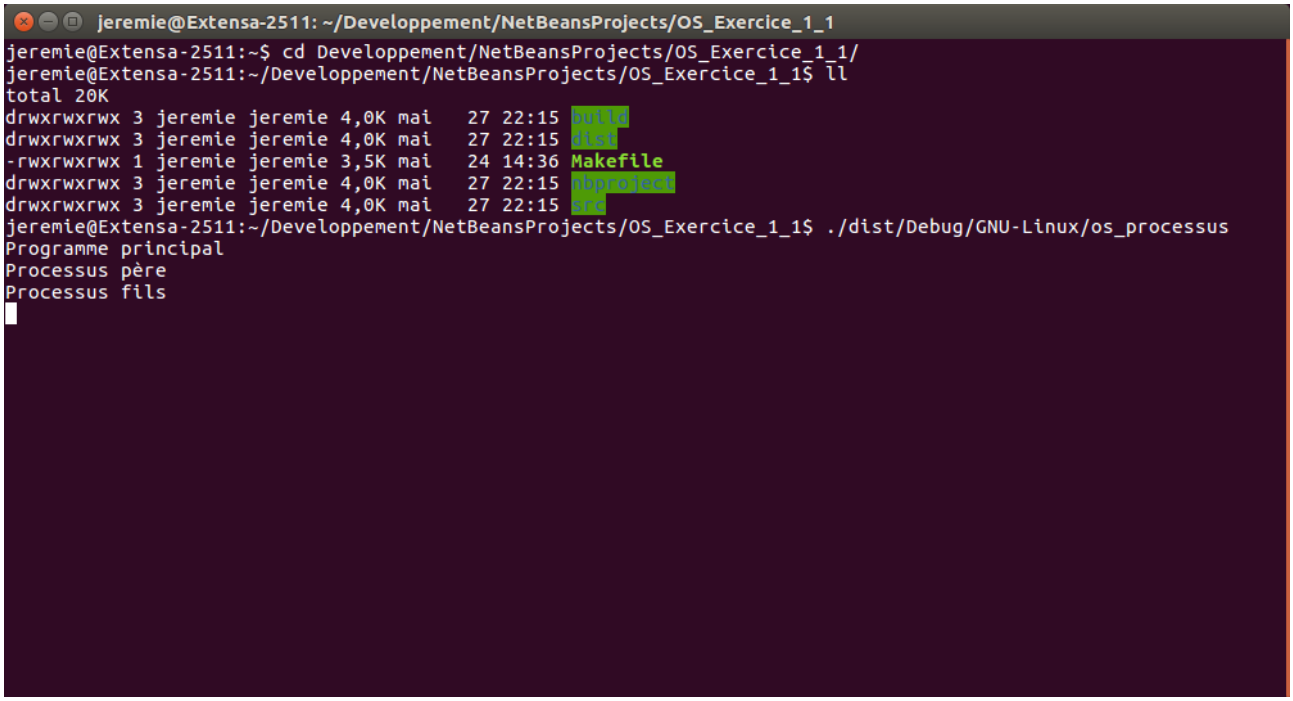
Header.h :

```
#ifndef HEADER_H
#define HEADER_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#endif      // HEADER_H
```

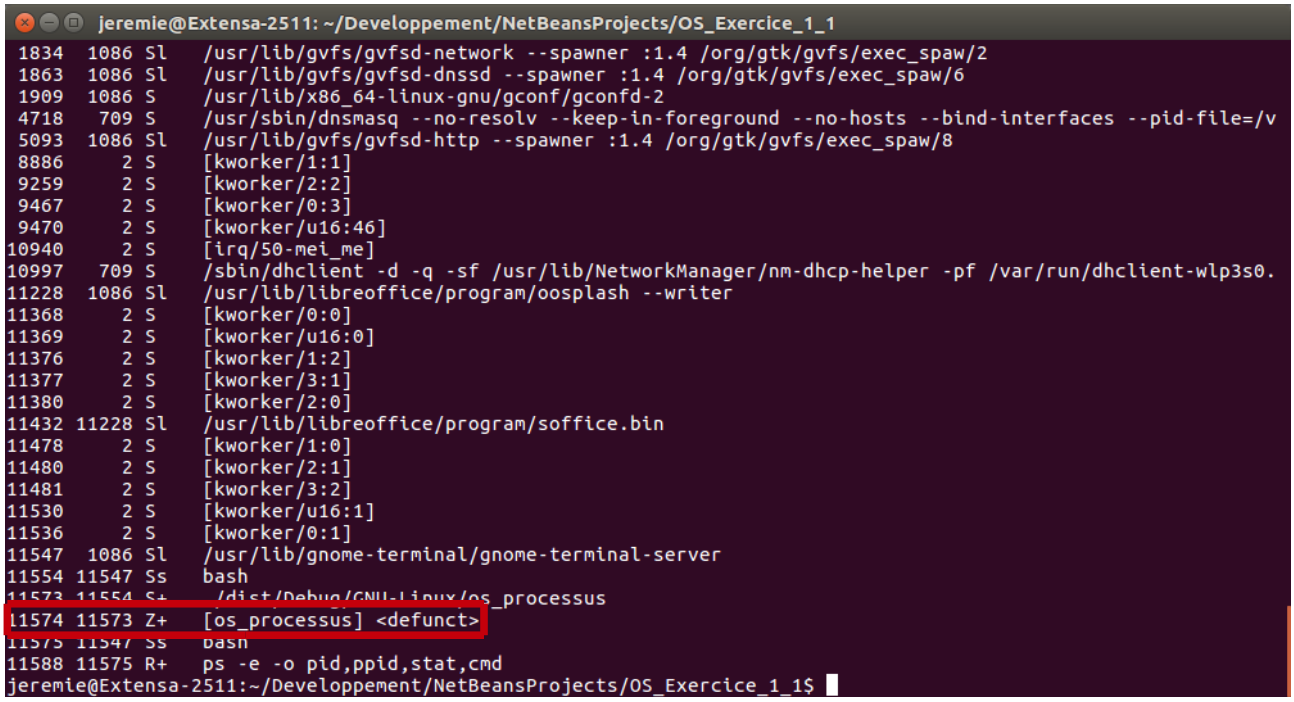
Main.c :

```
#include "headers/header.h"

int main(int argc, char** argv) {
    printf("Programme principal\n");
    int pid = fork();

    if (pid == -1) { printf("Fork impossible\n");
    } else if (pid == 0) {
        //Le processus fils désire mourir mais doit attendre que son père
        //récupérer son état pour se terminer.
        printf("Processus fils\n");
        exit(0);
    } else {
        //Le processus père ne pourra pas récupérer l'état du processus fils
        //pour que celui-ci puisse se terminer.
        printf("Processus père\n");
        pause();
    }
    return (EXIT_SUCCESS);
}
```

Trace d'exécution :

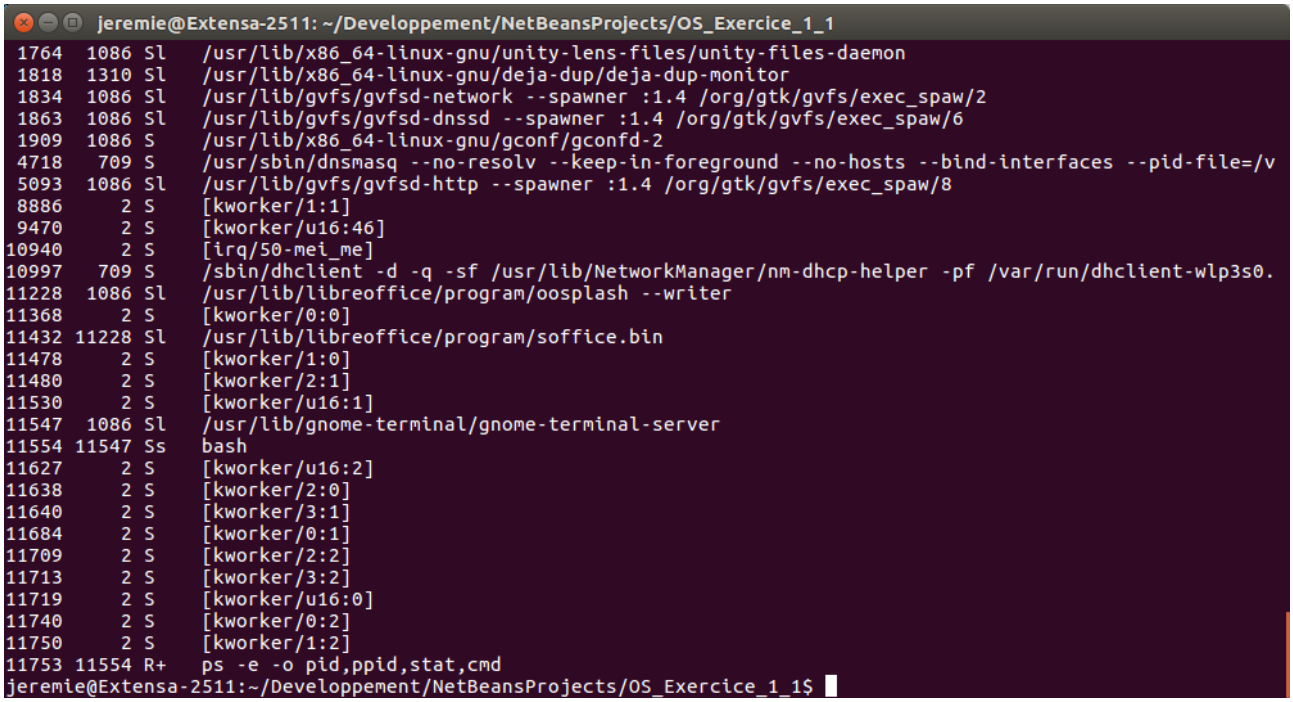


Une fois lancé, le programme crée bien deux processus, un père et un fils. En exécutant la commande « `ps -e -o pid,ppid,stat,cmd` » on obtient ceci :



On constate alors que le processus 11574, lancé par le processus11573 est en mode Zombie (Z+). En effet, le processus fils envoi un signal indiquant au père qu'il est arrivé au bout de son fil d'exécution, il cherche alors a mourir, mais le père est en mode sommeil et ne peut pas récupérer le signal envoyé par le fils. Le processus fils passe alors en mode zombie.

En interrompant le processus, on constate la disparition du processus père et du processus zombie :



Exercice 1.2 :

Le but de cet exercice est de nous faire créer deux processus concurrents qui vont partager des descripteurs de fichier. De cette façon, on pourra rediriger les flux d'entrée sortie (dans notre cas, de sortie) vers un fichier.

Code source :

Header.h

```
#ifndef HEADER_H
#define HEADER_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#endif      // HEADER_H
```

Main.c

```
#include "headers/header.h"

int main(int argc, char** argv) {
    int pid_filsDate = 0;
    int pid_filsAttente = 0;

    FILE* canal;
    canal = fopen("message.log", "wt");
    if (canal == NULL) {
        printf("%s\n", "Impossible d'ouvrir le fichier message.log");
        exit(0);
    }

    pid_filsDate = fork();
    if (pid_filsDate == 0) {
        for (int i = 0; i < 10; i++) {
            sleep(3);
            pid_filsDate = fork();
```

```
            dup2(canal->_fileno, 1);
            //Redirection de la sortie standard vers message.log.

            if (pid_filsDate == 0) {
                execl("/bin/date", "date", "-u", NULL);
                //execl ecrase le processus qui l'a lancé.
                //Les petits fils ne créeront pas d'autres processus.
            }
        }

        close(canal->_fileno);
        exit(1);
    }

    pid_filsAttente = fork();
    if (pid_filsAttente == 0) {
        for (int i = 0; i < 30; i++) {
            sleep(2);
            pid_filsAttente = fork();

            dup2(canal->_fileno, 1);
            //Redirection de la sortie standard vers message.log.

            if (pid_filsAttente == 0) {
                printf("Attendre\n");
                break; //Seul le fils parcours la boucle, pas les petits fils.
            }
        }

        close(canal->_fileno);
        exit(2);
    }

    while (wait(NULL) > -1) {

    }

    dup2(canal->_fileno, 1);
    printf("%s\n", "C'est terminé !");
    close(canal->_fileno);
    fclose(canal);
}
```

En exécutant le code ci-dessus, on remplit au fur et a mesure un fichier message.log qui contient les sorties du programme « date -u » et la sortie de notre programme. En effet, en utilisant les descripteurs de fichier, il nous est possible de rediriger le flux de sortie de printf vers un fichier (en l'occurrence, message.log).

Trace d'exécution :

Contenu de message.log :

```
Attendre
jeudi 26 mai 2016, 10:32:28 (UTC+0000)
Attendre
Attendre
jeudi 26 mai 2016, 10:32:31 (UTC+0000)
Attendre
jeudi 26 mai 2016, 10:32:34 (UTC+0000)
Attendre
jeudi 26 mai 2016, 10:32:37 (UTC+0000)
Attendre
Attendre
jeudi 26 mai 2016, 10:32:40 (UTC+0000)
Attendre
Attendre
jeudi 26 mai 2016, 10:32:43 (UTC+0000)
Attendre
jeudi 26 mai 2016, 10:32:46 (UTC+0000)
Attendre
Attendre
jeudi 26 mai 2016, 10:32:49 (UTC+0000)
Attendre
jeudi 26 mai 2016, 10:32:52 (UTC+0000)
Attendre
jeudi 26 mai 2016, 10:32:55 (UTC+0000)
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
Attendre
C'est terminé !
```

On constate bien que l'écriture des sortie se fait dans le fichier message.log. De plus, on peut remarquer qu'a chaque appel de la fonction sleep(), l'ordonnanceur passe la main à un autre processus, d'où l'enchevêtrement de l'affichage. Le timecode nous montre toutefois que les timings sont bien respectées.

Une fois tous les fils terminé, le processus père quitte en affichant « C'est terminé ! ».

Chapitre 2 - Communication inter-processus (IPC)

Exercice 2.1 :

Le but de ce programme est de permettre d'intercepter les signaux émis aux processus pour associer des actions a ces signaux. Dans notre cas, on interceptera les signaux SIGTERM (15 - envoyé par défaut grâce à la commande « `kill` ») pour incrémenter une variable.

Code source :

Header.h :

```
#ifndef HEADER_H
#define HEADER_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/signal.h>

void handler(int);

#endif      // HEADER_H
```

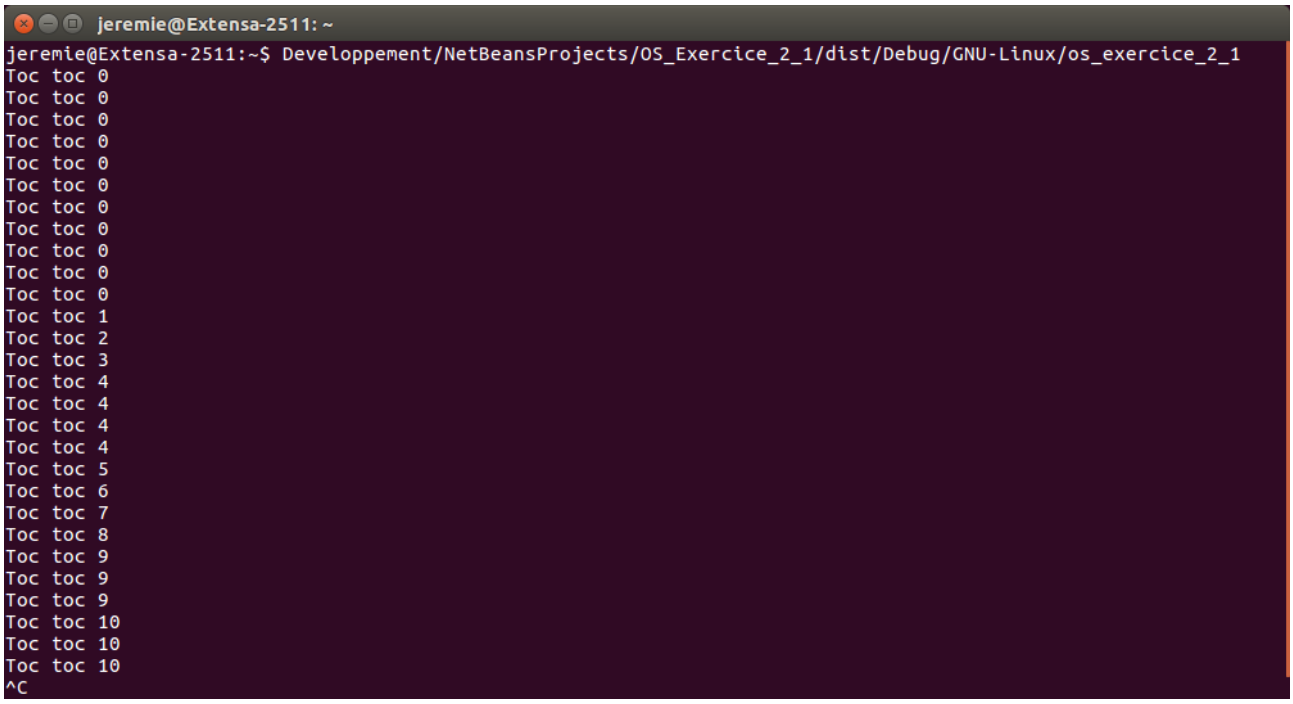
Main.c :

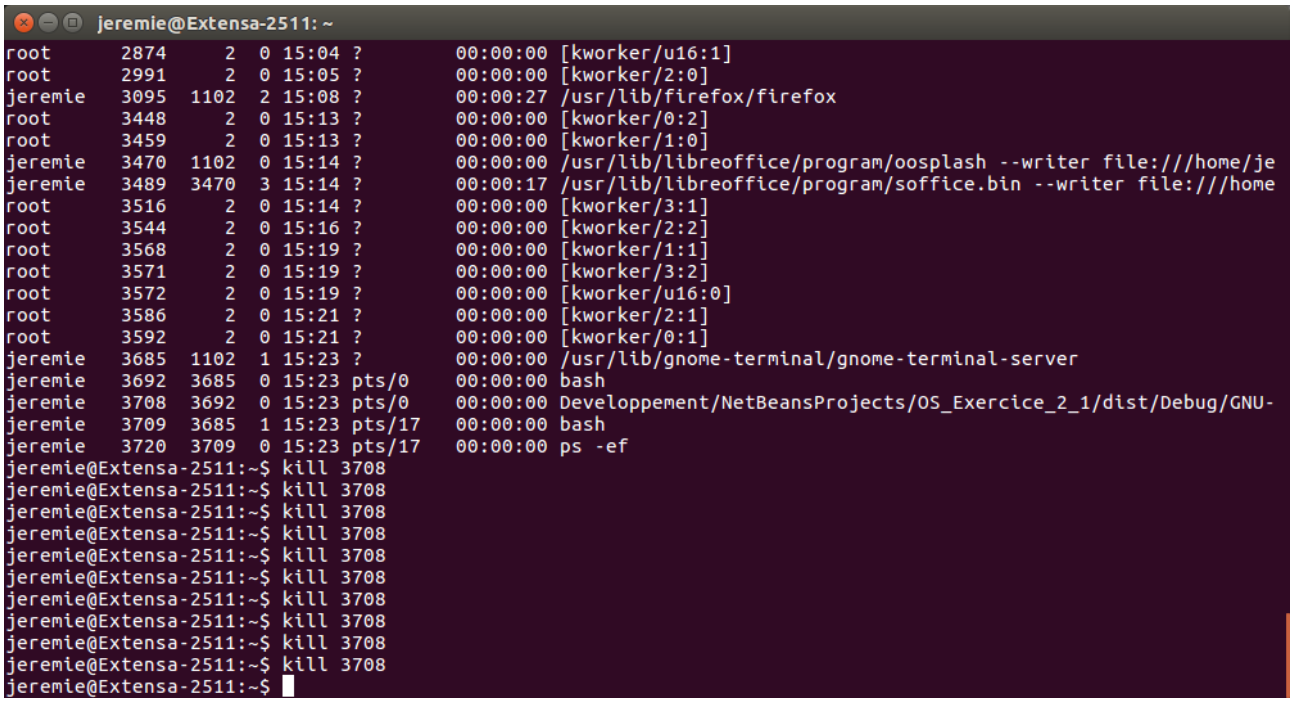
```
#include "headers/header.h"

int N = 0; //Attention, variable globale !

int main(int argc, char** argv) {
    signal(SIGTERM, handler);   //Assignation d'un handler
    while (1) {
        printf("Toc toc %d\n", N);
        sleep(2);
    }
    return (EXIT_SUCCESS);
}

void handler(int sig) {
    if (sig == SIGTERM) {
        N += 1;
        signal(SIGTERM, handler); //Sans cela le programme s'arrête au prochain
                                  //SIGTERM
    } else if (sig == SIGQUIT) { exit(0); }
}
```

Trace d'exécution :

Au lancement du programme, la variable globale N est initialisée à 0. L'affichage de la chaîne de caractère « toc toc » se fait toutes les deux secondes. Chaque fois que le programme reçoit le signal SIGTERM, la variable N est incrémentée.

Sur les deux captures ci dessous, on constate que la variable N est incrémentée autant de fois que l'on appel la commande « `kill 3708` ».





code