

UE Programmation Unix

TP1 Gestion des Processus

Les appels système : gestion de processus



Un appel système permet à un processus utilisateur d'accéder à une ou plusieurs fonctions internes au noyau du system d'exploitation et de les exécuter.

Ainsi, un programme peut invoquer un ou plusieurs services du système d'exploitation.



Concept de Processus

Définition **Processus** – un programme en cours d'exécution

Un processus est plus que le code d'un programme, qui est parfois appelé **la section de texte**.

- Un programme par lui-même n'est pas un processus : un programme est une **entité passive**, tel que le contenu d'un fichier stocké sur disque, tandis qu'un processus est une **entité active**.
- Bien que deux processus puissent être associés au même programme, ils sont toutefois considérés comme deux séquences d'instructions séparées.
Par exemple, plusieurs utilisateurs peuvent exécuter plusieurs copies du programme de gestion de courrier électronique ou le même utilisateur peut appeler plusieurs copies du programme éditeur.
- Chacune d'elles est un processus séparé et bien que **les sections de texte** soient équivalentes, **les sections de données ne sont pas les mêmes**.



La fonction main()

Arguments de la ligne de commandes

- Langage C offre des mécanismes qui permettent d'intégrer parfaitement un programme C dans l'environnement hôte
 - environnement orienté ligne de commande (Unix, Linux)
- Programme C peut recevoir de la part de l'interpréteur de commandes qui a lancé son exécution, une liste d'arguments
 - ligne de commande qui a servi à lancer l'exécution du programme
- Liste composée
 - du nom du fichier binaire contenant le code exécutable du programme
 - des paramètres de la commande

La fonction main ()

Un processus débute par l'exécution de la fonction `main()` du programme correspondant

Definition

```
int main (int argc, char *argv[]);  
ou  
int main (int argc, char **argv);
```

- `argc`: nombre d'arguments de la ligne de commande y compris le nom du programme
- `argv[]`: tableau de pointeurs vers les arguments (paramètres) passés au programme lors de son lancement

A NOTER

- `argv[0]` pointe vers le nom du programme
- `argv[argc]`vaut `NULL`

- argc (argument count)**
 - nombre de mots qui compose la ligne de commande (y compris le nom de la commande qui a servi à lancer l'exécution du programme)
- argv (argument vector)**
 - tableau de chaînes de caractères contenant chacune un mot de la ligne de commande
 - `argv[0]` est le nom du programme exécutable

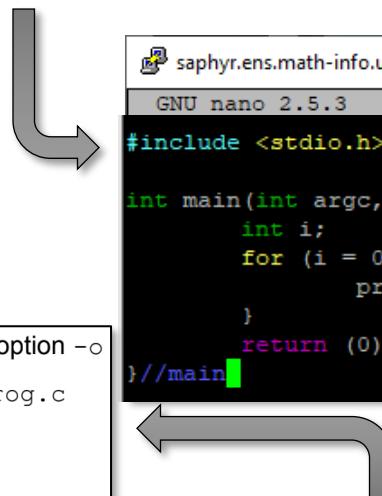
Exemple de cours

```
#include <stdio.h>

int main (int argc, char *argv[]){
    int i;
    for (i=0;i<argc;i++){
        printf ("argv[%d]: %s\n",i, argv[i]);
    }
    return (0);
}// main

>./affich_param Bonjour à tous
argv[0]: ./affich_param
argv[1]: Bonjour
argv[2]: à
argv[3]: tous
```

```
[ij04115@saphyr:~/unix_tpl]:ven. sept. 11$ nano affich_param.c
```



```
saphyr.ens.math-info.univ-paris5.fr - PuTTY
GNU nano 2.5.3          Fichier : affich_param.c
#include <stdio.h>

int main(int argc, char *argv[]){
    int i;
    for (i = 0 ; i < argc ; i++){
        printf("argv[%d]: %s\n", i, argv[i]);
    }
    return (0);
}//main
```

Spécifier le nom de l'exécutable avec l'option -o

```
ProgC > gcc -o toto premierProg.c
ProgC > ls
toto    premierProg.c
ProgC > ./toto
```

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$ gcc -o affich_param affich_param.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$ ls
affich_param  affich_param.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$
```

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$ ./affich_param Bonjour à tous
argv[0]: ./affich_param
argv[1]: Bonjour
argv[2]: à
argv[3]: tous
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$
```



Le fichier standard des erreurs stderr

Structure *FILE ** et variables *stdin*, *stdout* et *stderr*

Entête à inclure

```
#include <stdio.h>
```

Structure *FILE ** et variables *stdin*, *stdout* et *stderr*

```
FILE * stdin;
FILE * stdout;
FILE * stderr;
```

La structure FILE permet de stocker les informations relatives à la gestion d'un flux de données. Néanmoins, il est très rare que vous ayez besoin d'accéder directement à ses attributs.

Effectivement, il existe un grand nombre de fonctions qui acceptent un paramètre basé sur cette structure pour déterminer ou contrôler divers aspects.

- **stdin (Standard input)**: ce flot correspond au flux standard d'entrée de l'application. Par défaut, ce flux est associé au clavier : vous pouvez donc acquérir facilement des données en provenance du clavier. Quelques fonctions utilisent implicitement ce flux (**scanf**, par exemple).
- **stdout (Standard output)**: c'est le flux standard de sortie de votre application. Par défaut, ce flux est associé à la console d'où l'application a été lancée. Quelques fonctions utilisent implicitement ce flux (**printf**, par exemple).
- **stderr (Standard error)** : ce dernier flux est associé à la sortie standard d'erreur de votre application. Tout comme stdout, ce flux est normalement redirigé sur la console de l'application.

**fprintf it is the same as printf,
except now you are also specifying the place to print to :**

```
printf("%s", "Hello world\n"); // "Hello world" on stdout (using printf)
fprintf(stdout, "%s", "Hello world\n"); // "Hello world" on stdout (using fprintf)
fprintf(stderr, "%s", "Stack overflow!\n"); // Error message on stderr (using fprintf)
```



Messages d'erreurs affichés avec perror

C Library - <stdio.h>

C library function - perror()

Description

The C library function **void perror(const char *str)** prints a descriptive error message to stderr. First the string **str** is printed, followed by a colon then a space.

Declaration

Following is the declaration for perror() function.

```
void perror(const char *str)
```

Parameters

- **str** – This is the C string containing a custom message to be printed before the error message itself.

Return Value

This function does not return any value.

```
1 #include <stdio.h>
2
3 int main () {
4     FILE *fp;
5
6     /* first rename if there is any file */
7     rename("file.txt", "newfile.txt");
8
9     /* now let's try to open same file */
10    fp = fopen("file.txt", "r");
11    if( fp == NULL ) {
12        perror("Error: ");
13        return(-1);
14    }
15    fclose(fp);
16
17    return(0);
18 }
```

Let us compile and run the above program that will produce the following result because we are trying to open a file which does not exist –

```
Error: : No such file or directory
```

Tous les programmes devront être développés avec passage de leurs éventuels paramètres à la fonction main (int argc, char * argv [])

- Les valeurs de retour des appels aux primitives devront être testées et les messages d'erreurs affichés avec `perror`.
- Les messages d'erreurs à destination de l'utilisateur se feront sur le fichier standard des erreurs `stderr`.

1

Création de processus La primitive fork()

Création d'un processus

```
#include <unistd.h>
pid_t fork(void);
```

Crée un nouveau processus

Retourne :

- Dans le processus créé et appelé *processus fils*: ZÉRO
- Dans le processus appelant et appelé *processus père*: le PID du fils créé
- En cas d'erreur, chez le processus père: -1

Particularités

- UN APPEL RÉUSSI de cette primitive RETOURNE DEUX FOIS:
une fois chez le père et une fois chez le fils
- UN APPEL RÉUSSI retourne DEUX RÉSULTATS DIFFÉRENTS
- Un père peut avoir plusieurs fils et un fils n'a qu'un seul père
- Un fils a toujours père (adoption par init)

Soit 2 processus P1 et P2.

P1 crée le processus P2

puis exécute une boucle suffisamment longue pour permettre l'observation et dans laquelle il écrit à chaque itération :

```
PERE, mon pid est valeur du pid  
Le pid de mon fils est valeur du pid
```

P2 dès sa création,

exécute une boucle suffisamment longue pour permettre l'observation et dans laquelle il écrit à chaque itération :

```
FILS, mon pid est valeur du pid  
Le pid de mon pere est valeur du pid
```

Fonctions utiles:

- *fork()*: cf. cours.
- *getpid()* et *getppid()* retournent respectivement le pid du processus le pid de son père.

NB : on n'utilisera ni le exit ni le wait.

Identification des processus

Identificateur de processus (Process ID ou PID) :

Entier unique supérieur à zéro attribué par le système à tout processus

```
#include <unistd.h>
```

- **pid_t getppid(void);**

Renvoie le PID du père du processus appelant: *qui est mon père ?*

- **pid_t getpid(void);**

Renvoie le PID du processus appelant

Aucune de ces primitives ne retourne d'erreur

a) Écrire le programme C correspondant (respecter l'indentation des messages de P1 et P2 pour davantage de lisibilité sur l'écran au moment de l'exécution)

```
#include <stdio.h> // perror(), fprintf(), printf()
#include <stdlib.h> // atoi(), exit()
#include <unistd.h> // getppid() , getpid(), fork()

int main (int argc, char *argv[]) {

    int pid;
    int i;
    int iter_pere, iter_fils;

    if(argc != 3){

        fprintf(stderr, "Usage: %s iteration_pere iterations_fils \n", argv[0]);
        exit(1);
    }

    iter_pere = atoi(argv[1]);
    iter_fils = atoi(argv[2]);

    pid = fork();

    if (pid == -1){

        perror("fork");
        exit(1);
    }

    if (pid == 0) { /* CODE DU FILS */
        for(i = 1 ; i <= iter_fils ; i++){

            printf("\t\t\tFILS, mon pid est %d\n", getpid() ) ;
            printf("\t\t\tFILS, le pid de mon père est %d\n", getppid() ) ;
            fflush(stdout) ;

        }
    }
    else if (pid > 0) { /* CODE DU PERE */

        for(i = 1 ; i <= iter_pere ; i++){

            printf("PERE, mon pid est %d\n", getpid() ) ;
            printf("PERE, le pid de mon fils est %d\n", pid) ;

        } // for
    }
} // main
```

Entête à inclure

#include <stdlib.h> // <cstdlib> en C++

Fonction atoi

int atoi(const char * theString);

Cette fonction permet de transformer une chaîne de caractères, représentant une valeur entière, en une valeur numérique de type **int**. Le terme d'**atoi** est un acronyme signifiant : **A**SCII **T**o **I**nteger.



ATTENTION : la fonction **atoi** retourne la valeur 0 si la chaîne de caractères ne contient pas une représentation de valeur numérique. Du coup, il n'est pas possible de distinguer la chaîne "0" d'une chaîne ne contenant pas un nombre entier. Si vous avez cette difficulté, veuillez préférer l'utilisation de la fonction **strtol** qui permet bien de distinguer les deux cas.

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$ gcc -o questionl_a questionl_a.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$ ls
affich_param  affich_param.c  questionl_a  questionl_a.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$
```

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$ ./questionl_a
Usage: ./questionl_a iteration_pere iterations_fils
[ij04115@saphyr:~/unix_tpl]:sam. sept. 12$
```

- b) Dans une autre fenêtre shell, vérifiez la validité des valeurs des pid affichés par les 2 processus à l'aide de la commande ps aux | grep *votre_nom_de_login*.

```
saphyr.ens.math-info.univ-paris5.fr - PuTTY
```

```
saphyr.ens.math-info.univ-paris5.fr - PuTTY
```

```
[ij04115@saphyr:~/unix_tpl]:dim. sept. 13$ ps aux | grep ij04115
root 13156 0.0 0.1 20472 10496 ? Ss 08:17 0:00 sshd: ij04115 [priv]
ij04115 13248 0.0 0.1 20608 6868 ? S 08:17 0:00 sshd: ij04115@pts/0
ij04115 13249 0.0 0.1 21016 10172 pts/0 Ss+ 08:17 0:00 -bash2
root 13256 0.0 0.1 20472 10516 ? Ss 08:17 0:00 sshd: ij04115 [priv]
ij04115 13305 0.0 0.1 20472 6236 ? S 08:17 0:00 sshd: ij04115@pts/1
ij04115 13306 0.0 0.1 21016 10044 pts/1 Ss 08:17 0:00 -bash2
ij04115 13444 0.0 0.1 21232 8348 pts/1 R+ 08:34 0:00 ps aux
ij04115 13445 0.0 0.0 5180 944 pts/1 S+ 08:34 0:00 grep ij04115
```

```
sap saphyr.ens.math-info.univ-paris5.fr - PuTTY
```

```
[ij04115@saphyr:~/unix_tpl]:dim. sept. 13$ ./questionl_a 4 2
[ij041] PERE, mon pid est 13446
root PERE, le pid de mon fils est 13447
ij0411 PERE, mon pid est 13446
ij0411 PERE, le pid de mon fils est 13447
root PERE, mon pid est 13446
ij0411 PERE, le pid de mon fils est 13447
ij0411 PERE, mon pid est 13446
ij0411 PERE, le pid de mon fils est 13447
FILS, mon pid est 13447
FILS, le pid de mon père est 1
FILS, mon pid est 13447
FILS, le pid de mon père est 1
```

```
saphyr.ens.math-info.univ-paris5.fr - PuTTY
```

```
[ij04115@saphyr:~/unix_tpl]:dim. sept. 13$ ./questionl_a 4 2
[ij04115@saphyr:~/unix_tpl]:dim. sept. 13$ ps aux | grep ij04115
root 13156 0.0 0.1 20472 10496 ? Ss 08:17 0:00 sshd: ij04115 [priv]
ij04115 13248 0.0 0.1 20608 6868 ? S 08:17 0:00 sshd: ij04115@pts/0
ij04115 13249 0.0 0.1 21016 10172 pts/0 Ss+ 08:17 0:00 -bash2
root 13256 0.0 0.1 20472 10516 ? Ss 08:17 0:00 sshd: ij04115 [priv]
ij04115 13305 0.0 0.1 20472 6236 ? S 08:17 0:00 sshd: ij04115@pts/1
ij04115 13306 0.0 0.1 21016 10044 pts/1 Ss 08:17 0:00 -bash2
ij04115 13448 0.0 0.1 21232 8196 pts/1 R+ 08:34 0:00 ps aux
ij04115 13449 0.0 0.0 5180 864 pts/1 S+ 08:34 0:00 grep ij04115
[ij04115@saphyr:~/unix_tpl]:dim. sept. 13$
```

ps aux | grep votre_nom_de_login

La commande grep. La commande grep filtre le flux de texte qu'elle reçoit et ne laisse passer que les lignes contenant la chaîne de caractères donnée en argument.

```
$ who | grep math-info.univ-paris5.fr  
dupont pts/0 2010-08-12 12:15 (mars.math-info.univ-paris5.fr)
```

Cette ligne de commandes donne rapidement la liste des utilisateurs connecté sur la machine depuis le domaine « math-info.univ-paris5.fr ». Toute les lignes contenant le texte utilisé comme critère de filtrage s'affichent, quel que soit l'emplacement du texte sur la ligne (début, milieu ou fin).

aux, where "a" lists all processes on a [terminal](#), including those of other users, "x" lists all processes without [controlling terminals](#) and "u" adds a column for the controlling user for each process. For maximum compatibility, there is no "-" in front of the "aux". "ps auxww" provides complete information about the process, including all parameters.

- **a** and **x** control which processes are selected, and used together are explicitly described to select all processes.
- **u** outputs using the "user-oriented" format, which gives more columns, including the user id and CPU/memory usage.

a Lift the BSD-style "only yourself" restriction, which is imposed upon the set of all processes when some BSD-style (without "-") options are used or when the ps personality setting is BSD-like. The set of processes selected in this manner is in addition to the set of processes selected by other means. An alternate description is that this option causes ps to list all processes with a terminal (tty), or to list all processes when used together with the **x** option.

u Display user-oriented format.

x Lift the BSD-style "must have a tty" restriction, which is imposed upon the set of all processes when some BSD-style (without "-") options are used or when the ps personality setting is BSD-like. The set of processes selected in this manner is in addition to the set of processes selected by other means. An alternate description is that this option causes ps to list all processes owned by you (same EUID as ps), or to list all processes when used together with the **a** option.

2

Vie des processus

a) Lancez plusieurs fois le programme précédent. Quelles séquences d'exécution observez-vous, que faut-il en déduire ?

On retrouve des similitudes (similarité) entre chaque lancement du programme. Dans la plupart des cas c'est le pss père qui commence. Mais, Ceci n'est pas toujours vrai !

Les paramètres / charge du système sont tels qu'on n'a qu'une vision de l'exécution du programme.

- b) Ajustez le nombre d'itérations du processus père P1 de telle sorte qu'il se termine **AVANT** son fils P2. Que constatez-vous ?

Il suffit de mettre un plus grand nombre d'itération pour le fils que son père. Le pss init (1) prend la garde du pss fils P2 quand son père « biologique » (P1) meurt.

Processus particuliers

• Un fils a toujours père (adoption par init)

- Processus 0 : scheduler ou swapper
Aucun programme sur le disque ne correspond à ce processus. Il s'agit d'un processus système qui fait partie du noyau
- Processus 1 : init
 - Crée par le système à la fin du démarrage (bootstrap)
Le programme correspondant est /etc/init sur les anciennes versions d'Unix et sur les récentes /sbin/init
 - Il utilise les fichiers d'initialisation du système /etc/rc* ou /etc/init.d et /etc/inittab
 - Il n'appartient pas au noyau.
 - Il ne meurt jamais et devient le père de tout processus orphelin

Race condition

Dans une même application, il y a une *race condition* lorsque un résultat diffère selon l'ordre d'exécution des processus qui la composent

- Il est impossible de prévoir qui du père ou du fils s'exécutera en premier après le fork
 - Le système décide qui du père ou du fils reprendra en premier son exécution en fonction de sa charge courante et de sa politique globale d'ordonnancement
- Le système fournit des outils de communication inter-processus (signaux, sémaphores, tubes, etc) afin d'éviter les situations de race condition

c) Ajustez le nombre d'itérations du processus fils P2 de telle sorte qu'il se termine **AVANT** son père P1. Dans quel état se trouve P2 du point de vue du système ?

Le pss P2 est dans l'état Z (Zombie). Dès que le processus père a obtenu le code de fin du processus achevé au moyen des appels systèmes wait ou waitpid, le processus terminé (Zombie) est définitivement supprimé de la table des processus.

Remarques

ps -u permet de voir l'état « STAT » du pss.

R (run) signifie que le pss est en cours d'exécution.

S (sleep) signifie qu'il dort.

Z (zombie) est un pss qui est mort et que son père ne s'occupe pas de lui.

3

Effet du « clonage » sur les données

Le processus fils hérite d'une copie des données du père.

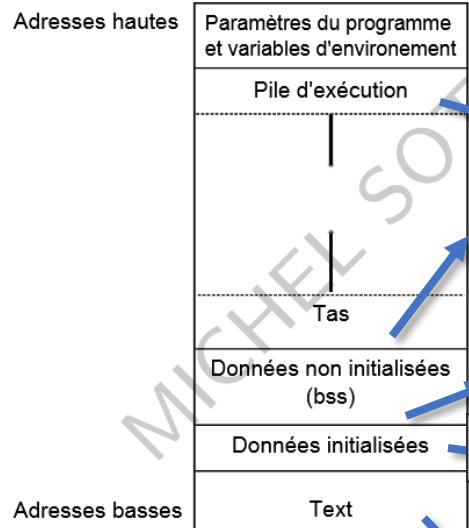
Cette copie n'est réalisée que lors d'accès en écriture (**copy on write**) et est telle que :

- L'ensemble des adresses valides des deux processus est le même,
- Les valeurs des données dans les deux processus sont les mêmes au retour de l'appel du fork(),
- Les données physiques sont différentes

- copy-on-write (COW)

- En réalité, le père et le fils partagent l'espace mémoire du père en lecture seule
- La copie n'est effectuée que lorsque le père ou le fils tentent de modifier cet espace partagé et seule la page concernée est copiée

Représentation en mémoire d'un processus



Pile d'exécution (stack).

- Variables automatiques temporaires créées à l'appel de chaque fonction
- Adresse de retour de la fonction

Tas (heap)

Utilisé pour l'allocation dynamique de mémoire (`malloc`, `calloc`, `realloc`, `free`)

Segment des variables non initialisées (uninitialized data segment)

- Les variables déclarées en dehors de toute fonction sont initialisées avec un zéro arithmétique ou à `NULL` pour les pointeurs
- Appelé historiquement `block started by symbol (bss)`

Segment des variables initialisées (data segment)

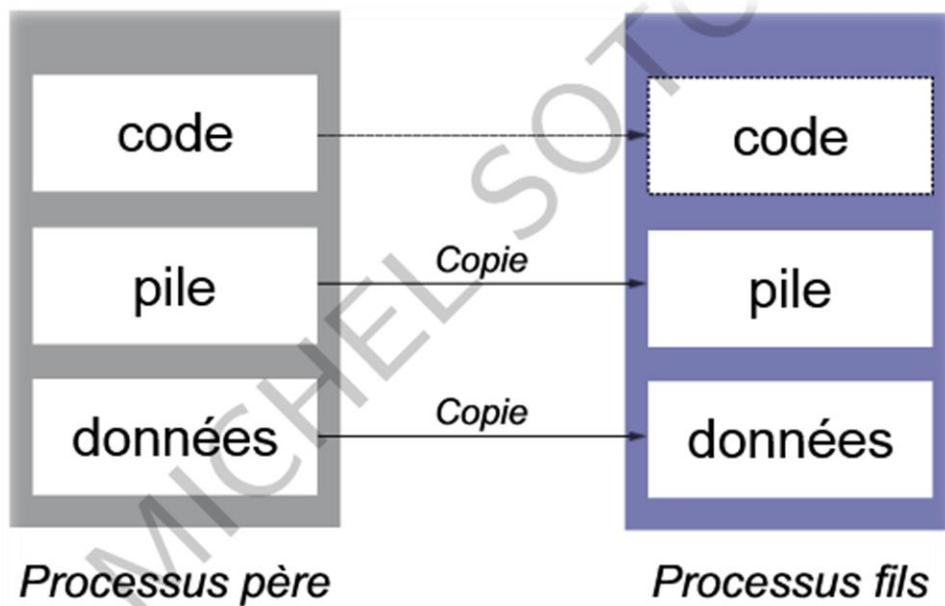
Variables en dehors de toute fonction qui ont été initialisées au moment de leur déclaration

Segment de texte (text).

- Contient les instructions en langage machine du programme
- Partageable: un seul exemplaire par programme est nécessaire en mémoire
- Accessible uniquement en lecture afin d'éviter toute modification accidentelle

Effets conceptuels du fork

- Le fils est une copie intégrale du père (clone) à l'instant du fork**
- Le père et le fils continuent leurs exécutions respectives avec l'instruction qui suit immédiatement le `fork` dans le code
- Le père est le fils ne partagent aucune zone de mémoire excepté le segment text**
 - Deux pointeurs ayant la même valeur chez le père et chez le fils désignent deux zones de mémoire physique différentes



Effet du fork

a) Ecrire le programme suivant de façon à vérifier les affirmations précédentes : 

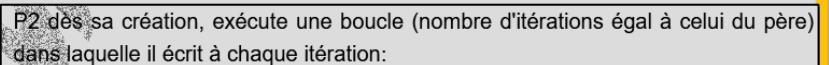
Soit 2 processus P1 et P2.

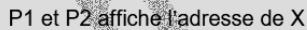
P1 crée le processus P2

PUIS exécute une boucle (suffisamment longue pour permettre l'observation)

dans laquelle il décrémente à chaque itération une variable entière X initialisée avec une valeur quelconque:

 PERE valeur de X

 P2 dès sa création, exécute une boucle (nombre d'itérations égal à celui du père) dans laquelle il écrit à chaque itération:
FILS, valeur de X

 P1 et P2 affiche l'adresse de X

Le processus fils hérite d'une copie des données du père.

Cette copie n'est réalisée que lors d'accès en écriture (**copy on write**)
et est telle que :

- L'ensemble des adresses valides des deux processus est le même,
 - Les valeurs des données dans les deux processus sont les mêmes au retour de l'appel du **fork()**,
 - Les données physiques sont différentes
- 

```
#include <stdio.h> // printf(), perror()
#include <stdlib.h> // exit()
#include <unistd.h> // fork()

int pid, x;

int main(int argc, char *argv[]) {

    x = 1;
    pid = fork();

    if(pid == -1) {
        perror("pb fork");
        exit(1);
    }

    if(pid == 0) { /* CODE DU FILS */

        printf("\t\tfils - adresse de x : %p\n", &x) ;
        printf("\t\tfils - valeur de x après fork : %d\n", x) ;
        x = x + 1 ;
        printf("\t\tfils - valeur de x = x + 1 : %d\n", x) ;
        printf("\t\tfils - adresse de x : %p\n", &x) ;

    } else { /* (pid > 0) CODE DU PERE */

        printf("pere - adresse de x : %p\n", &x) ;
        printf("père - valeur de x après fork : %d\n", x) ;
        x = x + 2 ;
        printf("père - valeur de x = x + 2 : %d\n", x) ;
        printf("père - adresse de x : %p\n", &x) ;
    } //else

    return (0) ;
} //main
```

```
[ij04115@saphyr:~/unix_tpl]:lun. sept. 14$ gcc -o question3_a question3_a.c
[ij04115@saphyr:~/unix_tpl]:lun. sept. 14$ ls
affich_param    out      question1_a.c  question3_a.c
affich_param.c  question1_a  question3_a   root
[ij04115@saphyr:~/unix_tpl]:lun. sept. 14$ ./question3_a
père - adresse de x : 0x804a02c
père - valeur de x après fork : 1
père - valeur de x = x + 2 : 3
père - adresse de x : 0x804a02c
          fils - adresse de x : 0x804a02c
          fils - valeur de x après fork : 1
          fils - valeur de x = x + 1 : 2
          fils - adresse de x : 0x804a02c
[ij04115@saphyr:~/unix_tpl]:lun. sept. 14$
```

Le processus fils hérite d'une copie des données du père.

Cette copie n'est réalisée que lors d'accès en écriture (**copy on write**) et est telle que :

- L'ensemble des adresses valides des deux processus est le même,
 - Les valeurs des données dans les deux processus sont les mêmes au retour de l'appel du fork().
 - Les données physiques sont différentes
-
- copy-on-write (COW)
 - En réalité, le père et le fils partagent l'espace mémoire du père en lecture seule
 - La copie n'est effectuée que lorsque le père ou le fils tentent de modifier cet espace partagé et seule la page concernée est copiée

4

La famille processus

Un processus **P** crée un fils **F1** et un fils **F2**.

F2 crée 1 fils **F3**.

F2 affiche :

"Je suis **w** : **F2**, le frère de **x**: **F1**"

F3 affiche :

"Je suis **y** : **F3**, le petit-fils de **p** : **P** et le neveu de **x**: **F1**"

a

Ecrire en C le programme correspondant en sachant que

- w** est la valeur du pid de **F2**,
- x** est la valeur du pid de **F1**,
- y** est la valeur du pid de **F3**
- p** la valeur du pid de **P**.

Remarque : La seule manière pour que **F2** puisse donner le pid de son frère : il faut que son frère soit créé avant et que son pid soit enregistré dans une variable pour qu'il puisse y avoir héritage du pid de **F1** par **F2**. De la même manière pour le pid de **P**.

```

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int w, // PID de F2
        y, // PID de F3
        x, // PID de F1
        p; // PID de P
    int pid;

    p = getpid(); // PID de P
    x = fork(); // CODE de P

    if (x == 0)
    {

        /* x = 0 -> code du fils, donc : CODE DE F1 */

    }

    else // x > 0, code du père, CODE DE P
    {

        pid = fork();
        if (pid == 0) // pid = 0 -> code du fils -> CODE DE F2
        {

            w = getpid(); // PID de F2
            printf("Je suis %d: F2, le frère de %d: F1\n", w,x);

            pid = fork();
            if (pid == 0) // pid = 0 -> code du fils -> CODE F3
            {
                y = getpid(); // PID de F3
                printf("Je suis %d : F3", y);
                printf(", le petit-fils de %d : P", p);
                printf(" et le neveu de %d : F1\n", x);

            } // fin du CODE de F3

        } // fin du CODE de F2

    } // fin du CODE de P

    return (0);
} // main

```

```
[ij04115@saphyr:~/unix_tpl]:lun. sept. 14$ gcc -o question4_a question4_a.c
[ij04115@saphyr:~/unix_tpl]:lun. sept. 14$ ls
affich_param    out          question1_a.c  question3_a.c  question4_a.c
affich_param.c  question1_a  question3_a  question4_a  root
[ij04115@saphyr:~/unix_tpl]:lun. sept. 14$ ./question4_a
Je suis 31627: F2, le frère de 31626: F1
[ij04115@saphyr:~/unix_tpl]:lun. sept. 14$ Je suis 31628 : F3, le petit-fils de
31625 : P et le neveu de 31626 : F1
[ij04115@saphyr:~/unix_tpl]:lun. sept. 14$
```

b On voudrait aussi que F1 affiche :

"Je suis **x: F1**, le frère de **w : F2**".

Quel problème cela pose-t-il ?

Impossible de récupérer le pid d'un processus pas encore créé.

Remarque : La seule manière pour que **F2** puisse donner le pid de son frère : il faut que son frère soit créé avant et que son pid soit enregistré dans une variable pour qu'il puisse y avoir héritage du pid de **F1** par **F2**. De la même manière pour le pid de **P**.

5

La synchronisation père/fils

Fonctions utiles:

 *fork, wait, exit: cf. cours et le man*

Synchronisation père-fils

- L'exécution d'un processus père et de ses processus fils sont totalement asynchrones
- La fin d'un fils peut se produire à n'importe quel moment de l'exécution de père
- Le système informe le père de la fin d'un de ses fils en lui envoyant le signal SIGCHLD
 - *Par défaut, ce signal n'a aucun effet sur le processus père* mais le père peut décider de capter ce signal
- Un processus père peut attendre et s'informer de la terminaison de ses fils grâce aux fonctions `wait` et `waitpid`
 - Ces primitives fonctionnent en tandem avec les fonctions `exit` et `_exit`

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- Bloque le processus père si *tous* ses fils sont en cours d'exécution
- Retourne immédiatement si au moins un des fils est terminé ou si tous les fils sont terminés
- `status` : contient la valeur du `exit` d'un fils ou une valeur élaborée par le système en cas de terminaison brutale du fils concerné
- Renvoie :
 - Le PID du fils qui s'est terminé
 - -1 en cas d'erreur

```
#include <stdlib.h>
void exit(int status);
```

- Ferment tous les descripteurs de fichiers
- Terminent le processus appelant
- Transmet au père la valeur de `status`

Soit un processus P1 qui crée d'abord n (n<5) processus et ensuite attend qu'ils terminent.

a Chaque **P_n** entre dans une boucle (suffisamment longue pour permettre l'observation) puis se termine par **exit(n)**.

Chaque fois qu'un de ces fils se termine, **P1** affiche :

- Le pid du fils terminé.
- La valeur transmise par le **exit** du fils qui se termine.
- Le nombre de fils restant à attendre.

```
#include <stdio.h> // // perror(), fprintf(), printf()
#include <stdlib.h> // atoi, exit()
#include <sys/types.h> // wait()
#include <sys/wait.h> // wait()
#include <unistd.h> // getpid(), fork(), sleep()

int main(int argc, char *argv[]) {
    int pid, i;
    int code_retour, nbre_fils, dorts_fils;

    if(argc != 3) {
        fprintf(stderr, "Usage: %s nombre_de_fils duree_sommeil_fils\n", argv[0]);
        exit(1);
    }

    nbre_fils = atoi(argv[1]);
    dorts_fils = atoi(argv[2]);

    /* un processus P1 qui crée nbre_fils processus */
    for(i = 1 ; i <= nbre_fils ; i++) {
        pid = fork();
        if(pid == -1) {
            perror("fork ");
            exit(1);
        }

        if(pid == 0){ /* CODE DU FILS */
            printf("\nFils : mon PID est %d, je mendors pour %d s\n", getpid(), dorts_fils);
            sleep(dorts_fils) ;
            exit(i);
        }
    } // for

    /* ATTENTE DES FILS */
    un processus P1 qui crée nbre_fils processus et en
    for(i = nbre_fils ; i >= 1 ; i--) {
        pid = wait(&code_retour);
        if(pid == -1) {
            perror("wait: ");
            exit(1);
        }

        printf("\nPere: mon fils %d s'est termine. \n", pid);
        printf("      : sa valeur de retour est %d\n", code_retour);
        printf("Encore %d fils\n", i - 1);
    } // for

    return (0);
} //main
```

P_n entre dans une boucle (suffisamment longue pour permettre l'observation) puis se termine par **exit(n)**.

pid_t wait(int *status);
status : contient la valeur du exit d'un fils
Renvoie : Le PID du fils qui s'est terminé

Chaque fois qu'un de ces fils se termine, **P1** affiche :

- Le pid du fils terminé.
- La valeur transmise par le **exit** du fils qui se termine.
- Le nombre de fils restant à attendre.

```
[ij04115@saphyr:~/unix_tpl]:mer. sept. 16$ gcc -o question5_a question5_a.c
[ij04115@saphyr:~/unix_tpl]:mer. sept. 16$ ls
affich_param    out          question1_a.c  question3_a.c  question4_a.c  question5_a.c
affich_param.c  question1_a  question3_a   question4_a   question5_a   root
[ij04115@saphyr:~/unix_tpl]:mer. sept. 16$ ./question5_a 4 10

Fils: mon PID est 11124, je m endors pour 10 s

Fils: mon PID est 11125, je m endors pour 10 s

Fils: mon PID est 11126, je m endors pour 10 s

Fils: mon PID est 11127, je m endors pour 10 s

Pere: mon fils 11124 s'est termine.
: sa valeur de retour est 256
Encore 3 fils

Pere: mon fils 11125 s'est termine.
: sa valeur de retour est 512
Encore 2 fils

Pere: mon fils 11126 s'est termine.
: sa valeur de retour est 768
Encore 1 fils

Pere: mon fils 11127 s'est termine.
: sa valeur de retour est 1024
Encore 0 fils
[ij04115@saphyr:~/unix_tpl]:mer. sept. 16$
```

```
[ij04115@saphyr:~/unix_tpl]:mer. sept. 16$ ./question5_a
Usage: ./question5_a nombre de fils duree_sommeil_fils
[ij04115@saphyr:~/unix_tpl]:mer. sept. 16$
```

b Modifier le programme précédent de façon que le processus père indique également si la terminaison de son fils s'est effectuée :

- Normalement et affiche la valeur transmise par le **exit** du fils qui se termine.
- Suite à la réception d'un signal non intercepté et affiche le n° du signal en cause.

```

#include <stdio.h> // // perror(), fprintf(), printf()
#include <stdlib.h> // atoi, exit()
#include <sys/types.h> // wait()
#include <sys/wait.h> // wait()
#include <unistd.h> // getpid(), fork(), sleep()

int main(int argc, char *argv[]) {
    int pid, i;
    int code_retour, nbre_fils, dors_fils;

    if(argc != 3){
        fprintf(stderr, "Usage: %s nombre_de_fils duree_sommeil_fils\n", argv[0]);
        exit(1);
    }

    nbre_fils = atoi(argv[1]);
    dors_fils = atoi(argv[2]);

    for(i = 1 ; i <= nbre_fils ; i++){
        pid = fork();
        if(pid == -1){
            perror("fork ");
            exit(1);
        }

        if(pid == 0){ /* CODE DU FILS */
            printf("\nFils : mon PID est %d, je mendors pour %d s\n", getpid(), dors_fils);
            sleep(dors_fils);
            exit(i);
        }
    } // for

    /* ATTENTE DES FILS */
    for(i = nbre_fils ; i >= 1 ; i--){
        pid = wait(&code_retour);
        if(pid == -1) {
            perror("wait: ");
            exit(1);
        } // if(pid == -1)

        printf("\nPere: mon fils %d s'est termine. \n", pid);

        if(WIFEXITED(code_retour)) {
            printf(" normalement et ma retourner %d\n", WEXITSTATUS(code_retour));
            // exit(0);
        }

        if(WIFSIGNALED(code_retour)) {
            printf(" de facon anormale\n");
            printf(" il a recu le signal %d\n", WTERMSIG(code_retour));
            // exit(0);
        }

        printf("Encore %d fils\n", i - 1);
    } // for

    return (0);
}

```

**Interprétation de valeur renvoyée par
exit dans le paramètre status**

WIFEXITED(status)	Renvoie une valeur non nulle si le fils s'est terminé normalement
WEXITSTATUS(status)	Renvoie le code de retour du processus si le processus s'est terminé normalement
WIFSIGNALED(status)	Renvoie une valeur non nulle si le fils s'est terminé à cause d'un signal
WTERMSIG(status)	Renvoie le n° du signal qui a provoqué la mort du processus fils

printf("\nPere: mon fils %d s'est termine. \n", pid);

```

if (WIFEXITED(code_retour)) {
    printf(" normalement et ma retourner %d\n", WEXITSTATUS(code_retour));
    // exit(0);
}

```

```

if (WIFSIGNALED(code_retour)) {
    printf(" de facon anormale\n");
    printf(" il a recu le signal %d\n", WTERMSIG(code_retour));
    // exit(0);
}

```

printf("Encore %d fils\n", i - 1);
} // for

return (0);

}

saphyr.ens.math-info.univ-paris5.fr - PuTTY

```
[ij04115@saphyr:~/unix_tpl]:mer. sept. 16$ gcc -o question5_b question5_b.c
[ij04115@saphyr:~/unix_tpl]:mer. sept. 16$ ls
affich_param    question1_a      question3_a.c  question5_a      question5_b.c
affich_param.c  question1_a.c   question4_a    question5_a.c  root
out            question3_a      question4_a.c  question5_b
[ij04115@saphyr:~/unix_tpl]:mer. sept. 16$ ./question5_b 4 30

Fils: mon PID est 11369, je m'endors pour 30 s

Fils: mon PID est 11370, je m'endors pour 30 s

Fils: mon PID est 11371, je m'endors pour 30 s

Fils: mon PID est 11372, je m'endors pour 30 s
```

Pere: mon fils 11369 s'est termine.
de facon anormale
il a recu le signal 9
Encore 3 fils

saphyr.ens.math-info.univ-paris5.fr - PuTTY

```
[ij04115@saphyr:~]:mer. sept. 16$ login as: ij04115
[ij04115@saphyr.ens.math-info.univ-paris5.fr's password:
Last login: Wed Sep 16 19:56:57 2020 from 88.121.6.235
[ij04115@saphyr:~]:mer. sept. 16$ kill -9 11369
[ij04115@saphyr:~]:mer. sept. 16$
```

Pere: mon fils 11370 s'est termine.
normalement et m'a retourner 2
Encore 2 fils

Pere: mon fils 11371 s'est termine.
normalement et m'a retourner 3
Encore 1 fils

Pere: mon fils 11372 s'est termine.
normalement et m'a retourner 4
Encore 0 fils

```
[ij04115@saphyr:~/unix_tpl]:mer. sept. 16$
```

6

Terminaison des processus

Terminaison d'un processus

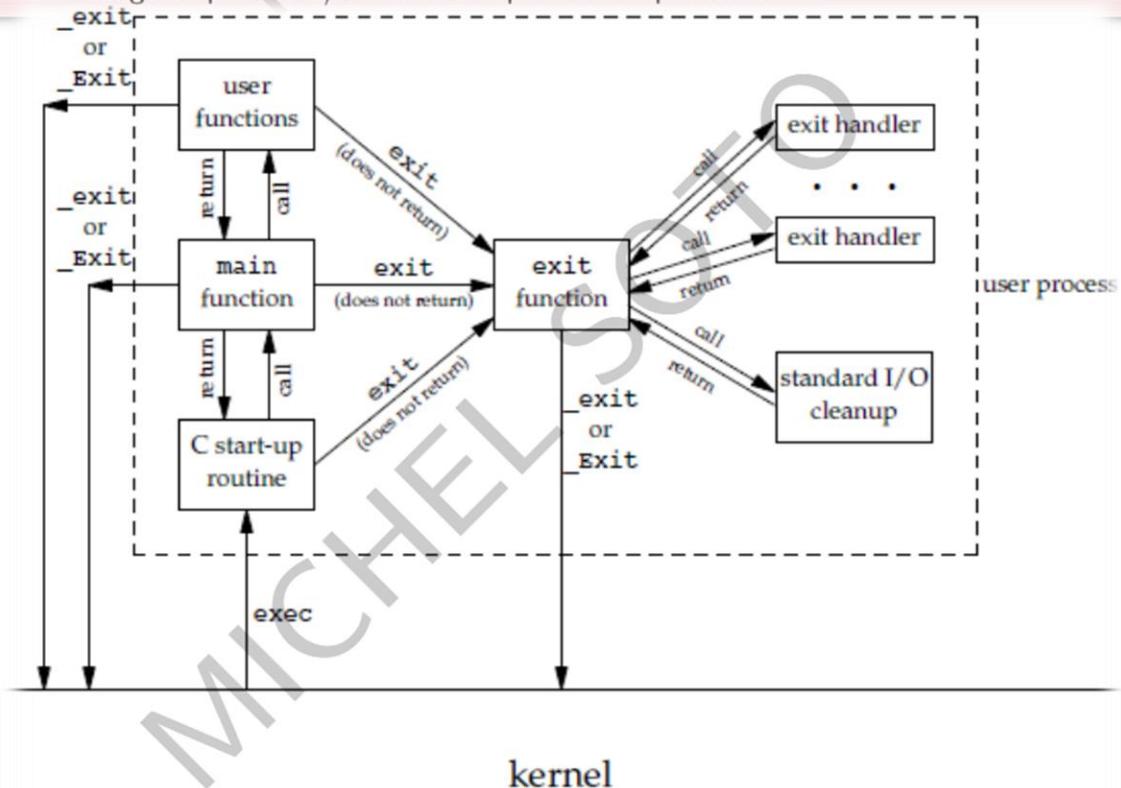
```
#include <stdlib.h>
void exit(int status);
```

```
#include <unistd.h>
void _exit(int status);
```

- ① Ferment tous les descripteurs de fichiers
- ② Terminent le processus appelant
- ③ Transmet au père la valeur de status

Differences

- `exit` exécute les gestionnaires de fin installés avec `atexit` alors que `_exit` n'exécute aucun gestionnaire de fin ou de signal, `exit` appelle `_exit`
- `exit` appartient à la librairie standard du C alors que `_exit` est une primitive du système
- La vidange ou pas des E/S standard dépend des implémentations



Les primitives `exit()` et `_exit()` entraînent la terminaison d'un processus, à la demande de ce dernier.

`exit()` diffère de `_exit()`, entre autres, par le fait qu'elle vide les tampons associés aux fichiers ouverts.

a Vérifier, par l'écriture d'un programme, l'affirmation précédente.

```
#include <stdio.h>// printf()
#include <stdlib.h>// exit(), EXIT_SUCCESS

int main(int argc, char *argv[]){
    // pas de \n donc pas de vidage de stdout
    printf(" PROGRAMME TERMINE ");

    // avec exit les buffers des fichiers ouvert sont vidés
    exit(EXIT_SUCCESS);
}
```

```
#include <stdio.h>// printf()
#include <stdlib.h>
#include <unistd.h> // _exit()

int main(int argc, char *argv[]){
    // pas de \n donc pas de vidage de stdout
    printf(" PROGRAMME TERMINE ");

    // avec exit les buffers des fichiers ouvert sont PAS vidés
    _exit(EXIT_SUCCESS);
}
```

```
[ij04115@saphyr:~/unix_tpl]:jeu. sept. 17$ cat question6_a1.c
#include <stdio.h> // printf()
#include <stdlib.h> // exit(), EXIT_SUCCESS

int main(int argc, char *argv[]){
    // pas de \n donc pas de vidage de stdout
    printf(" PROGRAMME TERMINE ");
    // avec exit les buffers des fichiers ouvert sont vidés
exit(EXIT_SUCCESS);
}

[ij04115@saphyr:~/unix_tpl]:jeu. sept. 17$ cat question6_a2.c
#include <stdio.h> // printf()
#include <stdlib.h>
#include <unistd.h> // _exit()

int main(int argc, char *argv[]){
    // pas de \n donc pas de vidage de stdout
    printf(" PROGRAMME TERMINE ");
    // avec exit les buffers des fichiers ouvert sont PAS vidés
_exit(EXIT_SUCCESS);
}

[ij04115@saphyr:~/unix_tpl]:jeu. sept. 17$ gcc -o question6_a1 question6_a1.c
[ij04115@saphyr:~/unix_tpl]:jeu. sept. 17$ gcc -o question6_a2 question6_a2.c
[ij04115@saphyr:~/unix_tpl]:jeu. sept. 17$ ./question6_a1
PROGRAMME TERMINE [ij04115@saphyr:~/unix_tpl]:jeu. sept. 17$ ./question6_a2
[ij04115@saphyr:~/unix_tpl]:jeu. sept. 17$ █
```

Qu'est-ce que le buffer en informatique ?

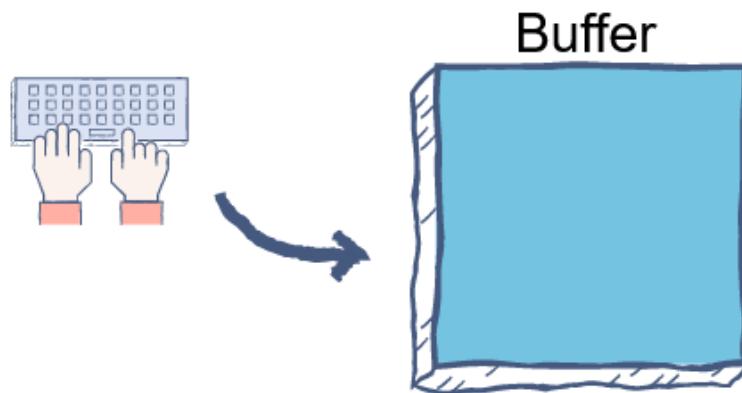
La mémoire tampon (buffer en anglais) sert à stocker temporairement des données dans la mémoire vive ou dans le disque dur d'un ordinateur. C'est en quelque sorte la "salle d'attente" des données et de toutes les informations qui transittent au sein d'un ordinateur moderne. Sans la mémoire tampon et sans les tampons, les ordinateurs fonctionneraient beaucoup moins efficacement et les temps d'attente seraient très longs. Le concept de la mémoire tampon a été mis au point pour éviter l'encombrement des données d'un port entrant à un port de transfert sortant.

Source : <https://www.journaldunet.fr/web-tech/dictionnaire-du-webmastering/1445278-buffer-definition-et-fonctionnement-pratique/>

As the name suggests, a **buffer** is temporary storage used to store input and output commands. All input and output commands are buffered in the operating system's buffer.

C language's use of a buffer

C uses a buffer to output or input variables. The buffer stores the variable that is supposed to be taken in (input) or sent out (output) of the program. A buffer needs to be cleared before the next input is taken in.



Structure *FILE ** et variables *stdin*, *stdout* et *stderr*

Entête à inclure

```
#include <stdio.h>
```

Structure *FILE ** et variables *stdin*, *stdout* et *stderr*

```
FILE * stdin;
FILE * stdout;
FILE * stderr;
```

La structure **FILE** permet de stocker les informations relatives à la gestion d'un flux de données. Néanmoins, il est très rare que vous ayez besoin d'accéder directement à ses attributs.

exit() diffère également de **_exit()** par le fait qu'elle appelle toutes les fonctions dont la demande d'exécution en cas de terminaison a été formulée au moyen de la fonction standard

```
int atexit(void (*fonction) (void));
```

C Library -
<stdlib.h>

b

Ecrire un programme qui formule la demande d'exécution de trois fonctions (func1(), func2() et func3()) en cas de terminaison.

Le processus doit se terminer sur un appel à exit().

```
#include <stdio.h> //printf()
#include <stdlib.h> //atexit(), exit(), EXIT_SUCCESS

void func1(){
    printf(" FONCTION 1 \n");
}

void func2(){
    printf(" FONCTION 2 \n");
}

void func3(){
    printf(" FONCTION 3 \n");
}

int main(int argc, char *argv[]){
    atexit(func1);
    atexit(func2);
    atexit(func3);
    exit(EXIT_SUCCESS);
}
```

[ij04115@saphyr:~/unix_tpl]:ven. sept. 18\$ gcc -o question6_b question6_b.c
[ij04115@saphyr:~/unix_tpl]:ven. sept. 18\$ ls
affich_param question1_a.c question4_a.c question5_b.c question6_a2.c
affich_param.c question3_a question5_a question6_a1 question6_b
out question3_a.c question5_a.c question6_a1.c question6_b.c
question1_a question4_a question5_b question6_a2 root
[ij04115@saphyr:~/unix_tpl]:ven. sept. 18\$./question6_b
FONCTION 3
FONCTION 2
FONCTION 1
[ij04115@saphyr:~/unix_tpl]:ven. sept. 18\$

C Vérifier, en remplaçant `exit()` par `_exit()` dans le programme précédent, que les fonctions (`func1()`, `func2()` et `func3()`) ne sont pas exécutées à la terminaison du processus

```
#include <stdio.h> //printf()
#include <stdlib.h> //atexit()
#include <unistd.h> //_exit()

void func1() {
    printf(" FONCTION 1 \n");
}

void func2() {
    printf(" FONCTION 2 \n");
}

void func3() {
    printf(" FONCTION 3 \n");
}

int main(int argc, char *argv[]){
    atexit(func1);
    atexit(func2);
    atexit(func3);

    _exit(EXIT_SUCCESS);
}
```

[ij04115@saphyr:~/unix_tpl]:ven. sept. 18\$ nano question6_c.c
[ij04115@saphyr:~/unix_tpl]:ven. sept. 18\$ gcc -o question6_c question6_c.c
[ij04115@saphyr:~/unix_tpl]:ven. sept. 18\$ ls
affich_param question3_a question5_a.c question6_a2 question6_c.c
affich_param.c question3_a.c question5_b question6_a2.c root
out question4_a question5_b.c question6_b
question1_a question4_a.c question6_a1 question6_b.c
question1_a.c question5_a question6_a1.c question6_c
[ij04115@saphyr:~/unix_tpl]:ven. sept. 18\$./question6_c
[ij04115@saphyr:~/unix_tpl]:ven. sept. 18\$



7

Les temps CPU

A **central processing unit (CPU)**, also called a central processor, main processor or just processor, is the electronic circuitry within a computer that executes instructions that make up a computer program.

La primitive times () renvoie les temps CPU consommés dans les deux modes (utilisateur et noyau) par le processus ainsi que par ses fils terminés ayant été attendus.

Les temps retournés dans les différents champs de la structure **tms** sont exprimés en nombre de clics d'horloge.

Pour les convertir en des durées exprimées en secondes, il suffit de les diviser par le nombre de clics d'horloge par seconde obtenu avec

sysconf (_SC_CLK_TCK) .

Variable	Value of Name
Clock ticks/second	_SC_CLK_TCK

sysconf - get configurable system variables

The **sysconf()** function provides a method for the application to determine the current value of a configurable system limit or option (*variable*).

```
#include <unistd.h>
long sysconf(int name);
```

The *name* argument represents the system variable to be queried.

Source : <https://pubs.opengroup.org/onlinepubs/009695399/functions/sysconf.html>

Temps CPU d'un processus

```
#include <sys/types.h>
clock_t times(struct tms *buf);

struct tms {
    clock_t    tms_utime; /* temps CPU en mode utilisateur */
    clock_t    tms_stime; /* temps CPU en mode système */
    clock_t    tms_cutime; /* temps CPU utilisateur des fils terminés et attendus */
    clock_t    tms_cstime; /* temps CPU système des fils terminés et attendus */
};

Fournit le temps CPU consommé dans les deux modes (utilisateur et noyau) par le processus et ses fils terminés.
```

Retourne :

- le temps global écoulé exprimé en nombre de tics d'horloge
- -1 en cas d'erreur

REMARQUE

Pour convertir les résultats en durées exprimées en secondes, les diviser par CLK_TCK ([`<limits.h>`](#))

Ecrire un programme qui :

- 1 affiche le nombre de clics écoulés sur la machine et le nombre clics par seconde**
- 2 effectue un certain calcul assez long (> 1 seconde)**
- 3 affiche les temps CPU utilisateur et système consommés**

```

#include <stdio.h> // printf(), fprintf(), perror()
#include <stdlib.h> // exit(), atoi()
#include <unistd.h> // sysconf()
#include <sys/times.h> //times(), struct tms
The clock_t type measures time in clock ticks

clock_t nb_tick; // le temps global écoulé exprimé en nombre de tics d'horloge
/* Les temps rentrés dans les différents champs de la structure tms sont exprimés en
nombre de clics d'horloge. */
struct tms info;

int i, p;
float u, // temps CPU en mode utilisateur
      s, // temps CPU en mode système
      tick_sec; // le nombre de clics d'horloge par seconde

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr,"Usage %s: nb iterations (> 30 000 000) \n", argv[0]);
        exit (1);
    }

    for(i = 1, p = 1 ; i <= atoi(argv[1]) ; i++, p++){
        i = i * 1;
        if (p == 100){
            printf("%d\n", i);
            p = 0;
        }
    } // for

    printf("\n");
    nb_tick = times(&info);
    if (nb_tick == -1){
        perror("pb times ");
        exit (1);
    }

    /* affiche le nombre de clics écoulés sur la machine et le nombre clics par seconde */

    printf("nb ticks écoulé : %d - nb tick/sec: %d\n", nb_tick, sysconf(_SC_CLK_TCK));
    u = info.tms_utime; // temps CPU en mode utilisateur
    s = info.tms_stime; // temps CPU en mode système
    tick_sec = sysconf(_SC_CLK_TCK); // le nombre de clics d'horloge par seconde

    printf("tms_utime: %f s \n", u/tick_sec);
    printf("tms_stime: %f s \n", s/tick_sec);
}

}

```

effectue un certain calcul assez long (> 1 seconde)

On peut imprimer un msg tout les 100 iterations

affiche les temps CPU utilisateur et système consommés

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ nano question7.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ gcc -o question7 question7.c
question7.c: In function 'main':
question7.c:32:9: warning: format '%d' expects argument of type 'int', but argument 2 has type 'cl
    printf("nb ticks écoulé : %d - nb tick/sec: %d\n", nb_tick, sysconf(_SC_CLK_TCK));
    ^
question7.c:32:9: warning: format '%d' expects argument of type 'int', but argument 3 has type 'lo
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ ./question7
Usage ./question7: nb iterations (> 30 000 000)
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ ./question7 500
100
200
300
400
500

nb ticks écoulé : 1806150997 - nb tick/sec: 100
tms_utime: 0.000000 s
tms_stime: 0.000000 s
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$
```

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ ./question7 35000000
```

```
34999700
34999800
34999900
35000000
```

```
nb ticks écoulé : 1806168847 - nb tick/sec: 100
tms_utime: 1.390000 s
tms_stime: 1.280000 s
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$
```

8

Les groupes de processus

La primitive `pid_t getpgrp(void)` renvoie le PID du leader du groupe de l'appelant.

NAME

`getpgrp` - get the process group ID of the calling process

SYNOPSIS

```
#include <unistd.h>
pid_t getpgrp(void);
```

DESCRIPTION

The `getpgrp()` function shall return the process group ID of the calling process.

RETURN VALUE

The `getpgrp()` function shall always be successful and no return value is reserved to indicate an error.

pid_t data type in C

`pid_t` data type stands for process identification and it is used to represent process ids. Whenever, we want to declare a variable that is going to be deal with the process ids we can use `pid_t` data type.

The type of `pid_t` data is a signed integer type (`signed int` or we can say `int`).

Header file:

The header file which is required to include in the program to use `pid_t` is `sys/types.h`

a Ecrire un programme qui affiche le PID du processus leader du groupe auquel il appartient.

```
#include <stdio.h> //printf()
#include <sys/types.h> // pid_t
#include <unistd.h> //getpgrp()

pid_t mon_groupe, mon_leader ;

int main(int argc, char *argv[]) {
    mon_leader = getpgrp();

    printf("Mon PID est: %d \n", getpid());
    printf("Le processus leader de mon groupe est: %d \n", mon_leader) ;

    return 0;
}
```

- #include <unistd.h>
pid_t getpid(void);
Renvoie le PID du processus appelant

Ne retourne pas d'erreur

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ nano question8_a.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ gcc -o question8_a question8_a.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ ./question8_a
Mon PID est: 10312
Le processus leader de mon groupe est: 10312
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ █
```

Un processus appartient au même groupe que celui auquel appartient son père et ceci jusqu'à ce qu'il demande à en changer.

b

Modifier le programme précédent de façon qu'il crée un processus fils qui affiche aussi le PID du processus leader du groupe auquel il appartient.

```
#include <stdio.h> //printf(), perror()
#include <stdlib.h> // exit()
#include <sys/types.h> // pid_t
#include <unistd.h> //getpgrp(), fork()

pid_t mon_groupe, mon_leader, pid ;

int main(int argc, char *argv[]) {

    pid = fork();
    if (pid == -1) {
        perror("Erreur du fork");
        exit(1);
    }

    if(pid == 0) { // Fils =====
        mon_leader = getpgrp();

        printf("FILS, Mon PID est : %d \n", getpid());
        printf("FILS, Le processus leader de mon groupe est: %d \n\n", mon_leader);
    }
    else { // Père =====

        mon_leader = getpgrp();

        printf("PERE, Mon PID est: %d \n", getpid());
        printf("PERE, Le processus leader de mon groupe est: %d \n", mon_leader);

    }
}

return 0;
}
```

C Vérifier si les deux processus précédents appartiennent au même groupe ?

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ nano question8_b.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ gcc -o question8_b question8_b.c
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ ./question8_b
PERE, Mon PID est: 10567
FILS, Mon PID est : 10568
FILS, Le processus leader de mon groupe est: 10567

PERE, Le processus leader de mon groupe est: 10567
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ ./question8_b
PERE, Mon PID est: 10569
PERE, Le processus leader de mon groupe est: 10569
FILS, Mon PID est : 10570
FILS, Le processus leader de mon groupe est: 10569

[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$
```

d Appartiennent-ils au même groupe auquel appartient le shell à partir duquel vous avez demandé l'exécution (**ps j**) ?

j BSD job control format.

```
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ ps j
  PID  PID  PGID  SID TTY      TPGID STAT   UID    TIME COMMAND
 8395  8396  8396  8396 pts/0      10725 Ss    11978  0:00 -bash2
 8396 10725 10725  8396 pts/0      10725 R+    11978  0:00 ps j
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ ./question8_b
PERE, Mon PID est: 10726
PERE, Le processus leader de mon groupe est: 10726
FILS, Mon PID est : 10727
FILS, Le processus leader de mon groupe est: 10726

[j?ij04115@saphyr:~/unix_tpl]:sam. sept. 19$ ps j
  PID  PID  PGID  SID TTY      TPGID STAT   UID    TIME COMMAND
 8395  8396  8396  8396 pts/0      10728 Ss    11978  0:00 -bash2
 8396 10728 10728  8396 pts/0      10728 R+    11978  0:00 ps j
[ij04115@saphyr:~/unix_tpl]:sam. sept. 19$
```

abcd