Labo 2

Objectifs : Quelques appels systèmes / fonctions pour la gestion de processus.

Introduction

Sous LINUX nous n'avons pas besoin de logiciel particulier pour pourvoir programmer en C, les outils nécessaires à l'édition et à la compilation du C sont déjà présents : VI et GCC.

Il est conseillé d'exécuter chacun des exemples ci-dessous afin de vous habituer à l'utilisation de PuTTY et des commandes.

Appels système[1] et processus

Avant de décrire ce que sont les appels système, voici quelques rappels concernant les *fonctions de bibliothèque*. *Il s'agit* de fonctions ordinaires qui se trouvent dans une bibliothèque externe au programme en court d'exécution. La plupart des fonctions utilisées jusqu'à présent se trouvent dans les bibliothèques standards du C, stdlib et stdio. Par exemple, printf et scanf en font partie.

Un appel à une fonction de bibliothèque est identique à l'appel de n'importe quelle autre fonction. Les arguments sont placés dans des registres du processeur ou sur la pile et l'exécution est transférée au début de la fonction appelée (voir dossier d'IG1 en Langage de programmation : base).

Un *appel système* est un appel à une fonction du noyau LINUX. Comme nous l'avons vu lors de la présentation de LINUX, il existe deux modes de fonctionnement : le mode noyau et le mode utilisateur. Un appel système consiste en une interruption logicielle (*system trap*) qui a pour rôle de changer de mode d'exécution pour passer du mode utilisateur au mode noyau, de récupérer les paramètres et de vérifier la validité de l'appel, de lancer l'exécution de la fonction demandée, de récupérer la (les) valeur(s) de retour et de retourner au programme appelant en passant à nouveau au mode utilisateur.

Un appel système n'est pas identique à un appel de fonction classique puisqu'une procédure spécifique est nécessaire pour transférer le contrôle au noyau. Cependant, la bibliothèque C GNU (l'implémentation de la bibliothèque standard du C fournie avec les systèmes GNU/LINUX) masque les appels système par des fonctions classiques afin qu'ils soient plus simples à utiliser. Les fonctions d'entrées-sorties de bas niveau comme `open` ou `read` font partie des appels systèmes LINUX.

Les appels système LINUX constituent l'interface de base entre les programmes et le noyau LINUX. À chaque appel correspond une opération ou une fonctionnalité de base. Certains appels sont très puissants et influent au niveau du système. Par exemple, il est possible d'éteindre le système ou d'utiliser des ressources système tout en interdisant leur accès aux autres utilisateurs. De tels appels ne sont utilisables que par des programmes s'exécutant avec les privilèges superutilisateur (lancé par `root`). Ils échouent si le programme ne dispose pas de ces droits.

Les appels système manipulent divers objets gérés par le système d'exploitation. Les processus et les fichiers sont les plus importants de ceux-ci.

Un programme est une entité **décrivant** un traitement (statique), alors qu'un processus est une entité **réalisant** un traitement (dynamique). Lors de l'exécution, son code est situé en mémoire centrale (en langage machine).

[1] Inspiré/copié du livre "Advanced Linux Programming (http://www.advancedlinuxprogramming.com) de Mark Mitchell, Jeffrey Oldham et Alex Samuel".

Afin de gérer les processus, le système d'exploitation utilise une table appelée **table des processus**. Chaque élément de cette table pointe (allocation dynamique) vers une structure communément appelée **bloc de contrôle du processus** (*Process Control Block*, souvent abrégé **PCB**).

Chaque processus, lors de sa création, se voit allouer un PCB, notamment pour permettre au système de le redémarrer avec les mêmes caractéristiques que lorsqu'il a été bloqué (voir cours de théorie : *scheduler* – gestion du CPU).

Diverses implémentations existent selon les systèmes d'exploitation, mais un PCB contient en général les informations suivantes :

- un espace d'adressage, c'est-à-dire la zone de la mémoire centrale qui contient le code exécutable (le segment de texte), les variables globales (le segment de donnes), les variables locales et la chaine d'activation des fonctions (la pile) et les données allouées dynamiquement (le tas) ;
- un contexte, c'est-à-dire l'ensemble des registres du processeur, notamment celui qui indique la prochaine instruction à exécuter et celui qui indique le sommet de la pile ;
- d'autres informations utiles, telles que…
    - le numéro du processus (pid)
    - le numéro de son processus père (ppid)
    - le propriétaire réel du processus (uid et gid) et le propriétaire effectif (euid et egid)
    - l'état du processus (endormi, en attente, en cours d'exécution...)
    - les fichiers ouverts par le processus
    - le répertoire courant du processus
    - le terminal attaché au processus
    - les duos (*flèches*, *fonction de déroutement*) dont chaque duo permet d'associer un traitement à chaque flèche reçue par le processus
    - la priorité du processus
    - la date de lancement du processus
    - le temps d'utilisation d'unité centrale écoulé
    - etc.

Les appels systèmes permettent notamment la création et l'arrêt des processus. Un processus peut créer un ou plusieurs processus fils qui, à leur tour, peuvent créer des processus fils formant une structure arborescente. Les processus peuvent se synchroniser et communiquer entre eux.

Actuellement, on utilise de plus en plus le concept de processus "légers" - thread. Les processus légers sont un moyen de raffiner et de diviser le travail normalement associé à un processus. Ceux-ci seront abordés dans le cours de Programmation orientée objet avancée (Java). Nous ne traiterons dans ce labo que des processus dits "lourd".

Vie et mort d'un processus

Naissance d'un processus

L'appel système permettant de créer un processus est `fork`.

fork	
Librairie	`#include <unistd.h>`
Prototype	`pid_t fork (void);`
Commentaires	Crée un nouveau processus (fils) semblable au processus courant (père). Le contexte des deux processus est exactement le même mis à part leur identifiant (*pid*) et l'identifiant de leur père respectif (*ppid*). La valeur renvoyée n'est pas la même pour le fils et pour le père, ce qui permet de les différencier. Trois cas sont à envisager : • le processus père, celui qui fait le `fork`, renvoie le numéro du fils (*pid*) • le processus fils qui est créé commence son exécution dès la fin du `fork` et renvoie 0 • en cas d'erreur, le processus fils n'est pas créé et le père reçoit une valeur négative

L'exemple suivant présente la structure d'un programme permettant de créer un processus fils à partir d'un processus père.

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

void fils1 (void) {
   printf("Debut du Fils 1\n");
   // Suite du Fils 1
   printf("Fin du Fils 1\n");
   exit(EXIT_SUCCESS);
}

int main (void) {
   int pid1;

   printf("Debut du Pere\n");
   pid1 = fork();
   if (pid1 < 0) {
      perror("Erreur lors de la creation (fork) du Fils 1");
      exit(EXIT_FAILURE);
   } // else inutile car exit
   if (pid1 == 0) {
      fils1();
   }
   //Suite du Père
   printf("Fin du Pere\n");
   exit(EXIT_SUCCESS);
}
```

Dans le cas où la création du processus fils **échoue** (pid1 < 0), un message d'erreur est affiché et le processus père se termine en spécifiant qu'il a rencontré un souci (voir `exit` avec comme argument `EXIT_FAILURE`).

Dans le cas où la création du processus fils **se déroule bien** (`pid1 >= 0`), comme les contextes des deux processus sont similaires (il s'agit du même programme mais les processus sont tout à fait indépendants), ils exécutent l'instruction qui suit l'appel du `fork`. C'est pourquoi il faut commencer par distinguer le processus père du processus fils en testant la valeur de retour du `fork`.

Processus père	Processus fils

`void fils1 (void) {`
`  printf("Debut du Fils 1\n");`
`  // Suite du Fils 1`
`  printf(Fin du Fils 1\n");`
`  exit(EXIT_SUCCESS);`
`}`
 	
`int main (void) {`
`  int pid1;`
 	
`  printf("Debut du Pere\n");`
 	
`0 ≠ pid1 = fork();`
`  if (pid1 == 0) {`
`  fils1();`
`  }`
 	
`  //Suite du Père`
 	
`  printf("Fin du Pere\n");`
`  exit(EXIT_SUCCESS);`
`}`
	`void fils1 (void) {`
	`  printf("Debut du Fils 1\n");`
	`  // Suite du Fils 1`
	`  printf(Fin du Fils 1\n");`
	`  exit(EXIT_SUCCESS);`
	`}`
	
	`int main (void) {`
	`  int pid1;`
	
	`  printf("Debut du Pere\n");`
	
	`0 = pid1 = fork();`
	`  if (pid1 == 0) {`
	`  fils1();`
	`  }`
	
	`  //Suite du Père`
	
	`  printf("Fin du Pere\n");`
	`  exit(EXIT_SUCCESS);`
	`}`

REMARQUE : par soucis de concision la gestion des erreurs est enlevée des deux programmes ci-dessus.

Ainsi selon qu'il s'agit du processus père ou du processus fils, les instructions adéquates seront exécutées.

Afin de tester cet exemple, connectez-vous au serveur. Ensuite, faites comme proposé ci-dessus, c'est à-dire : faites une copie du code ci-dessus, entrez la commande suivante dans PuTTY, en choisissant un nom de fichier approprié :

```
$ cat > nom_du_fichier.c
```

La commande `cat` permet de lister un fichier… Si aucun fichier ne lui est fourni, c'est le `stdin` qui est lu… Faites un *clic droit* pour coller le code/texte précédemment copié et appuyez sur [CTRL+d] pour envoyer ce code/texte au programme qui est en train de lire, en l'occurrence la commande `cat` !

Le résultat est redirigé vers le fichier grâce au chevron '>'.

Cela étant fait, il reste à compiler le programme et à l'exécuter pour obtenir le résultat présenté à la figure 4.

```
[piroc@svr24 piroc]$ gcc lab2ex1.c -o ex
[piroc@svr24 piroc]$ ./ex
Debut du Pere
Debut du Fils 1
Fin du Fils 1
Fin du Pere
[piroc@svr24 piroc]$
```

Figure 1 - Appel système fork

Voici quelques explications sur la fonction `perror`.

perror	
Librairie	`#include <unistd.h>`
Prototype	`void perror(const char * str);`
Commentaires	Affiche un message sur la sortie d'erreur standard (`stderr`), décrivant la dernière erreur rencontrée durant un appel système ou une fonction de bibliothèque. Si `str` n'est pas NULL et si `*str` n'est pas un octet nul, la chaine de caractères `str` est imprimée, suivie de ': 'ou d'un blanc, puis du message correspondant à l'erreur, et enfin d'un saut de ligne. Par convention, la chaine de caractères contient généralement le nom de la fonction qui a généré l'erreur. Le numéro d'erreur est obtenu à partir de la variable externe `errno` définie dans `<errno.h>`, qui contient le code d'erreur lorsqu'un problème survient (mais qui n'est pas effacée lorsqu'un appel est réussi). De plus, la liste globale d'erreurs `sys_errlist[]` (définie dans `<errno.h>`) peut être utilisée pour obtenir le message d'erreur en utilisant `errno` comme indice. Le nombre de messages contenus dans cette table ne peut dépasser la valeur de `sys_nerr`. Quand un appel système échoue, il renvoie habituellement -1, et place le code d'erreur dans `errno`. Beaucoup de fonctions de bibliothèque se comportent également ainsi.

L'exemple ci-dessous permet de montrer l'utilité d'un appel à `perror`.

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

int main (void) {
   FILE * pFile;
   pFile = fopen("unexist.ent","rb");
   if (pFile == NULL) {
      perror ("fopen a genere l'erreur suivante");
   } else {
      fclose (pFile);
   }
   exit(EXIT_SUCCESS);
}
```

Le résultat est visible sur la figure 5.

```
[piroc@svr24 piroc]$ ./test
fopen a genere l'erreur suivante: No such file or directory
[piroc@svr24 piroc]$
```

Figure 2 - Appel système perror

Mort du processus

Dans les exemples précédents, l'appel système `exit` est utilisé. En voici l'explication :

exit	
Librairie	`#include <stdlib.h>`
Prototype	`void exit (int status);`
Commentaires	Termine l'exécution du processus en renvoyant au père la valeur passée en paramètre. Cet appel système a les effets suivants : • tous les *file descriptors* appartenant au processus en question sont fermées. • les processus fils du processus terminé deviennent zombies et sont adoptés par le processus `init`. • le processus père reçoit un signal SIGCHLD. Le paramètre `status` (généralement appelé *exit status*) représente l'état dans lequel le processus s'est terminé (code de retour et cause de sa mort). Il est enregistré dans la table des processus afin que le processus père puisse l'intercepter au moyen d'un `wait`. L'état de terminaison du processus sera généralement précisé en utilisant une des deux constantes définies à cet effet dans `<stdlib.h>` : • EXIT_SUCCESS (= 0) • EXIT_FAILURE (≠ 0)

Le programme ci-dessus permet de comprendre en quoi l'appel à `exit` est nécessaire. En effet, si aucun appel à `exit` ne se retrouve dans la partie fils, il exécute aussi les instructions qui se situent après la structure alternative, celles qui ne doivent s'exécuter que dans le cas où il s'agit du père.

Avec `exit`

```
void fils1 (void) {
   // Suite du Fils 1
   exit(EXIT_SUCCESS);
}

int main (void) {
   int pid1;

   pid1 = fork();

   if (pid1 < 0) {
      perror("fork Fils 1");
      exit(EXIT_FAILURE);
   }
   if (pid1 == 0) {
      fils1();
   }
   //Suite du Père
   exit(EXIT_SUCCESS);
}
```

Quitte le programme...

Sans `exit`

```
void fils1 (void) {
   // Suite du Fils 1

}

int main (void) {
   int pid1;

   pid1 = fork();

   if (pid1 < 0) {
      perror("fork Fils 1");

   }
   if (pid1 == 0) {
      fils1();
   }
   //Suite du Père


}
```

Identifier un processus

Il est régulièrement nécessaire de pouvoir connaitre l'identité d'un processus et celle de son père. Les appels système permettant de les obtenir sont `getpid` et `getppid`.

getpid	
Librairie	`#include <stdlib.h>`
Prototype	`pid_t getpid (void);`
Commentaires	Renvoie l'identifiant du processus appelant. Cette fonction réussit toujours.
getppid	
Librairie	`#include <stdlib.h>`
Prototype	`pid_t getppid (void);`
Commentaires	Renvoie l'identifiant du processus père de l'appelant. Cette fonction réussit toujours.

Voici le code précèdent amélioré grâce à ces deux appels système :

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

void fils1 (void) {
   int pid1 = getpid();
   printf("PID %d - Debut du Fils 1 [PPID %d]\n", pid1, getppid());
   // Suite du fils 1
   printf("PID %d - Fin du Fils 1\n", pid1);
   exit(EXIT_SUCCESS);
}

int main (void) {
   int pid1;
   int pidP = getpid();

   printf("PID %d - Debut du Pere\n", pidP);

   pid1 = fork();
   if (pid1 < 0) {
      perror("Erreur lors de la creation (fork) du Fils 1");
      exit(EXIT_FAILURE);
   }
   if (pid1 == 0) {
      fils1();
   }
   //Suite du père
   printf("PID %d - Fin du Pere\n", pidP);
   exit(EXIT_SUCCESS);
}
```

Le résultat de l'exécution de ce nouveau programme donne ce qui est présenté en figure 6. Il est intéressant de remarquer que le processus père s'est terminé avant la fin du fils 1.

```
piroc@svr24:~$ ./ex
PID 19987 - Debut du Pere
PID 19987 - Fin du Pere
piroc@svr24:~$ PID 19988 - Debut du Fils 1 [PPID 1]
PID 19988 - Fin du Fils 1
```

Figure 3 - Appels système getpid et getppid

Attendre la fin d'un processus

L'appel système qui permet d'attendre la fin d'un processus est `wait`.

wait	
Librairie	`#include <stdlib.h>`
Prototype	`pid_t wait (int * status);`
Commentaires	Permet de synchroniser le ou les processus fils en faisant en sorte que le processus père attende qu'un des processus fils soit terminé. Elle renvoie le numéro du fils qui s'est arrêté et garnit `status` avec l'*exit status* disponible dans la table des processus.

Le programme suivant permet de voir comment cet appel est utilisé.

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

void fils1 (void) {
   int pid1 = getpid();
   int pidP = getppid();
   printf("PID %d - Debut du Fils 1 [PPID %d]\n", pid1, pidP);
   // Suite du fils 1
   printf("PID %d - Fin du Fils 1 [PPID %d]\n", pid1, pidP);
   exit(EXIT_SUCCESS);
}

int main (void) {
   int pidP = getpid();
   int pid1;
   int finishedPid;
   int status;

   printf("PID %d - Debut du Pere\n", pidP);

   pid1 = fork();
   if (pid1 < 0) {
      perror("Erreur lors de la creation (fork) du Fils 1");
      exit(EXIT_FAILURE);
   }
   if (pid1 == 0) {
      fils1();
   }
   // Suite du père.
   printf("PID %d – Le Pere attend son Fils 1 [PID %d]\n", pidP, pid1);

   finishedPid = wait(&status);
   if (finishedPid != pid1) {
      printf("Il y avait un fils inattendu [PID %d] !\n", finishedPid) ;
      exit(EXIT_FAILURE);
   }
   printf("PID %d – Le Fils 1 [PID %d] s'est termine avec le statut %04.4X\n",
            pidP, finishedPid, status);

   printf("PID %d - Fin du Pere\n", pidP);
   exit(EXIT_SUCCESS);
}
```

Si on exécute ce programme, on ne pourra pas observer que le père attend son fils étant donné que le fils se termine presqu'immédiatement ! Nous allons donc ajouter un appel système permettant d'endormir un processus…

Endormir un processus

La fonction qui permet d'endormir un processus est `sleep`.

sleep	
Librairie	`#include <unistd.h>`
Prototype	`void sleep (int time);`
Commentaires	Force un processus s'endormir pour `time` seconde(s).

Modifiez le fils afin qu'il s'endorme pendant 10 secondes.

Son code devient :

```
void fils1 (void) {
   int pid1 = getpid();
   int pidP = getppid();
   printf("PID %d - Debut du Fils 1 [PPID %d]\n", pid1, pidP);
   sleep(10);
   printf("PID %d - Fin du Fils 1 [PPID %d]\n", pid1, pidP);
   exit(EXIT_SUCCESS);
}
```

L'exécution du programme (complet) donne la sortie présentée en figure 7.

```
piroc@svr24:~$ gcc ex.c -o ex
piroc@svr24:~$ ./ex
PID 23108 - Debut du Pere
PID 23108 - Le Pere attend son Fils 1 [PID 23109]
PID 23109 - Debut du Fils 1 [PPID 23108]
PID 23109 - Fin du Fils 1 [PPID 23108]
PID 23108 - Le Fils 1 [PID 23109] s'est termine avec le statut 0000
PID 23108 - Fin du Pere
piroc@svr24:~$
```

Figure 4 - Appels système `wait` et `sleep`

REMARQUE

Un fils qui se termine mais n'a pas été attendu devient un zombie. Le noyau conserve des informations minimales sur le processus zombie (identifiant, *exit status*, informations d'utilisation des ressources) pour permettre au père de l'attendre plus tard et d'obtenir des informations sur le fils.

Tant que le zombie n'est pas effacé du système par un `wait`, il prendra un emplacement dans la table des processus du noyau, et si cette table est remplie, il sera impossible de créer de nouveaux processus. Pour rappel, si un processus père se termine, ses fils devenus zombies sont adoptés par `init`, qui les attend automatiquement pour les supprimer.

Exécuter une commande shell

Afin de pouvoir exécuter une commande shell, on utilise l'appel système `system`.

system	
Librairie	`#include <stdlib.h>`
Prototype	`int system (const char * cmd);`
Commentaires	Permet de lancer une ligne de commande (`sh`) depuis un programme. En réalité, un processus fils est créé, exécutant `sh` avec comme entrée la commande `cmd` passée en argument. Cette fonction fait donc un `fork` suivi d'un `exec` (voir cours de théorie et labo futur) dans la partie fils. Le père fait quant à lui un `wait` et renvoie la valeur de l'*exit status* du fils.

Voici un petit exemple d'utilisation de l'appel système `system` :

```
#include <stdlib.h>

int main (void) {
   int reponse;
   reponse = system("ls -l");
   printf("L'exit value de la commande est %d\n", reponse);
   exit(EXIT_SUCCESS);
}
```

Le résultat de ce programme est présenté à la figure 8.

```
piroc@svr24:~$ ./ex
ls: impossible d'accéder à -l: Aucun fichier ou dossier de ce type
L'exit value de la commande est 512
piroc@svr24:~$
```

Figure 5 - Appel système system

Espace d'adressage

Étant donné que les deux processus sont identiques mais que leurs espaces d'adressage sont distincts, une variable initialisée avant l'appel de `fork` a la même valeur dans les deux espaces d'adressage.

Puisque chaque processus a son propre espace d'adressage, toute modification sera indépendante des autres. En d'autres termes, si le père change la valeur d'une variable, la modification affectera uniquement la variable dans son espace d'adressage. L'espace d'adressage du fils créé lors de l'appel à `fork` n'est pas affecté, même s'ils comportent des noms de variables identiques. En revanche, au moment du fork, le fils hérite des valeurs de toutes les variables de son père.

Voici un programme qui illustre cet aspect de la gestion des processus…

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int i;
int myPid;

void fils1(void) {
   int reponse;

   myPid = getpid();
   printf("PID %d - Debut du Fils 1 avec i = %d [PPID = %d]\n", myPid, i, getppid());

   reponse = system("ls -f ./inexistant.txt");
   if (reponse == EXIT_SUCCESS) {
      printf("Le fichier existe !\n");
   } else {
      printf("Erreur lors de l'appel du shell (system) : %d\n", reponse);
   }
   printf("PID %d - Fin du Fils 1 avec i = %d\n", myPid, i);
   exit (EXIT_SUCCESS);
}

int main (void) {
   int finishedPid;
   int pid1;
   int status;

   myPid = getpid();
   printf("PID %d - Debut du Pere\n", myPid);
   i = 1;
   pid1 = fork();
   if (pid1 < 0) {
      perror("Erreur lors de la creation (fork) du Fils 1");
      exit(EXIT_FAILURE);
   }
   if (pid1 == 0) {
      fils1();
   }
   i = 2;
   printf ("Le Pere a modifie la variable --> i = %d\n", i);

   printf("PID %d - Le Pere attend son fils [PID %d]\n", myPid, pid1);
   finishedPid = wait(&status);
   if (finishedPid != pid1) {
      printf("Il y avait un fils inattendu [PID %d] !\n", finishedPid) ;
      exit(EXIT_FAILURE);
   }
   printf("PID %d - Le Fils 1 [PID %d] s'est termine avec le statut %04.4X\n",
            myPid, finishedPid, status);
   printf("PID %d - Fin du Pere avec i = %d\n", myPid, i);
   exit(EXIT_SUCCESS);
}
```

Le résultat de ce programme est présenté à la figure 9.

```
piroc@svr24:~$ ./ex
PID 23197 - Debut du Pere
Le Pere a modifie la variable --> i = 2
PID 23197 - Le Pere attend son fils [PID 23198]
PID 23198 - Debut du Fils 1 avec i = 1 [PPID = 23197]
ls: impossible d'accéder à ./inexistant.txt: Aucun fichier ou dossier de ce type
Erreur lors de l'appel du shell (system) : 512
PID 23198 - Fin du Fils 1 avec i = 1
PID 23197 - Le Fils 1 [PID 23198] s'est termine avec le statut 0000
PID 23197 - Fin du Pere avec i = 2
piroc@svr24:~$
```

Figure 6 - Espaces d'adressage différents...

Exemple de résolution d'un exercice

Avant de regarder la solution, essayez de la construire par vous-même !

Réfléchissez à la structure globale du programme sur base de ce que vous avez appris ci-dessus. Un DA ou quelques phrases en français pourront vous permettre de mettre vos idées au clair…

Ensuite regardez la première partie de la solution "Un peu de réflexion" et comparez-la avec vos écrits.

Rédigez le code et prenez la peine de le compiler afin d'éliminer les fautes de syntaxe !

Enfin, comparez votre solution avec celle proposée...

Énoncé

Écrire un programme où un processus père lance deux fils.

Chaque processus commence par afficher un message de début et se termine par un message de fin.

Les fils affichent un message personnel dix fois chacun. Un délai, respectivement, de 1 seconde et 2 secondes, doit être respecté entre chaque message personnel.

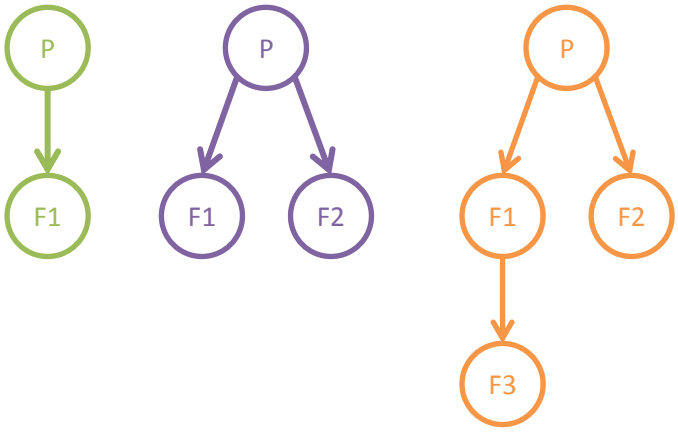
Le père attend la fin de l'exécution de ses deux fils avant d'afficher son message de fin.

Utiliser dans cet exercice les appels `fork`, `getpid`, `wait` et `sleep`.

Un peu de réflexion

Un processus peut être représenté par un nœud avec comme clé, sa position dans l'arbre (P pour père, F1 pour fils…) :

Ces nœuds, reliés sous la forme d'un arbre, représente la hiérarchie des processus :



Avant de réfléchir à l'affichage des messages, il faut établir la hiérarchie telle que demandée… Dans notre cas, c'est le deuxième exemple qui convient. Il faut donc que le père crée un premier processus, et ensuite qu'il en crée un second… Un pseudo-code, approchant le DA et sans la gestion des cas d'erreur, pourrait ressembler à ceci :

```
* père
Sortir "Début du père"

pid1 = fork()
if (pid1 = 0)
 fils1()

pid2 = fork()
if (pid2 = 0)
 fils2()

wait()
wait()

Sortir "Fin du père"
```

Ensuite il reste à ajouter les boucles pour les messages personnels de chacun des fils :

```
* fils1
Sortir "Début du fils1"

i = 1
do while (i ≤ 10)
 Sortir "Message ", i, " du fils 1"
 sleep(1)
 i++

Sortir "Fin du fils1"
```

```
* fils2
Sortir "Début du fils2"

i = 1
do while (i ≤ 10)
 Sortir "Message ", i, " du fils 2"
 sleep(2)
 i++

Sortir "Fin du fils2"
```

Une solution…

Voici le programme complet correspondant à la description ci-dessus.

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

int pidP, pid1, pid2;

void fils1(void){
/* Dans le fils 1 */
   int i;
   pid1 = getpid(); // il est à 0 quand on entre dans le fils donc il faut le mettre à jour
   printf("PID %d - Debut du premier fils [PPID %d]\n", pid1, pidP);
   for (i = 1; i <= 10; i++) {
      printf("PID %d - Message %d du premier fils -\n", pid1, i);
      sleep(1);
   }
   printf("PID %d - Fin du premier fils\n", pid1);
   exit(EXIT_SUCCESS);
}
void fils2(void){
/* Dans le fils 2 */
   int i;
   pid2 = getpid();// il est à 0 quand on entre dans le fils donc il faut le mettre à jour

   printf("PID %d - Debut du second fils [PPID %d]\n", pid2, pidP);
   for (i = 1; i <= 10; i ++) {
      printf("PID %d - Message %d du second fils --\n", pid2, i);
      sleep(2);
   }
   printf("PID %d - Fin du second fils\n", pid2);
   exit(EXIT_SUCCESS);
}

void pere(void){
/* Dans le pere */
   int finishedPid;
   int status;

   printf("PID %d - Le PID du Fils 1 est %d\n", pidP, pid1);
   printf("PID %d - Le PID du Fils 2 est %d\n", pidP, pid2);
   printf("PID %d - Le Pere attend son Fils 1 [PID %d]\n", pidP, pid1);
   finishedPid = wait(&status);
   if (finishedPid < 0) {
      perror("Erreur lors du wait\n");
      exit(EXIT_FAILURE);
   }
   printf("PID %d - Le Fils 1 [PID %d] s'est termine avec le statut %04.4X\n",
            pidP, finishedPid, status);

   printf("PID %d - Le Pere attend son Fils 2 [PID %d]\n", pidP, pid2);
   finishedPid = wait(&status);
   if (finishedPid < 0) {
      perror("Erreur lors du wait\n");
      exit(EXIT_FAILURE);
   }
   printf("PID %d - Le Fils 2 [PID %d] s'est termine avec le statut %04.4X\n",
            pidP, finishedPid, status);
   printf("PID %d - Fin du Pere\n", pidP);
   exit (EXIT_SUCCESS);
}
```

```
int main(void) {
   int status;

   pidP = getpid();
   printf("PID %d - Debut du Pere\n", pidP);

   if ((pid1 = fork()) < 0){
      perror("Erreur lors du fork du fils 1");
      exit (EXIT_FAILURE);
   }
   if (pid1 == 0) {
      // Dans le fils 1
      fils1();
   }

   if ((pid2 = fork()) < 0) {
      perror("Erreur lors du fork du fils 2");
      exit(EXIT_FAILURE);
   }
   if (pid2 == 0) {
      // Dans le fils 2
      fils2();
   }
   // Suite du père.
   // Le fils n'exécute jamais ce code grâce à son exit
   pere();
}
```

Lorsqu'on utilise la sortie standard (*stdout*), le résultat est le suivant :

```
piroc@svr24:~$ ./ex
PID 23242 - Debut du Pere
PID 23242 - Le PID du Fils 1 est 23243
PID 23242 - Le PID du Fils 2 est 23244
PID 23242 - Le Pere attend son Fils 1 [PID 23243]
PID 23244 - Debut du second fils [PPID 23242]
PID 23244 - Message 1 du second fils --
PID 23243 - Debut du premier fils [PPID 23242]
PID 23243 - Message 1 du premier fils -
PID 23243 - Message 2 du premier fils -
PID 23244 - Message 2 du second fils --
PID 23243 - Message 3 du premier fils -
PID 23243 - Message 4 du premier fils -
PID 23244 - Message 3 du second fils --
PID 23243 - Message 5 du premier fils -
PID 23243 - Message 6 du premier fils -
PID 23244 - Message 4 du second fils --
PID 23243 - Message 7 du premier fils -
PID 23243 - Message 8 du premier fils -
PID 23244 - Message 5 du second fils --
PID 23243 - Message 9 du premier fils -
PID 23243 - Message 10 du premier fils -
PID 23244 - Message 6 du second fils --
PID 23243 - Fin du premier fils
PID 23242 - Le Fils 1 [PID 23243] s'est termine avec le statut 0000
PID 23242 - Le Pere attend son Fils 2 [PID 23244]
PID 23244 - Message 7 du second fils --
PID 23244 - Message 8 du second fils --
PID 23244 - Message 9 du second fils --
PID 23244 - Message 10 du second fils --
PID 23244 - Fin du second fils
PID 23242 - Le Fils 2 [PID 23244] s'est termine avec le statut 0000
PID 23242 - Fin du Pere
piroc@svr24:~$
```

Figure 7 - Vers *stdout*

Si, par contre, on envoie la sortie standard vers un fichier au moyen du chevron '>', voici ce qui est envoyé dans le fichier :

```
piroc@svr24:~$ ./ex > lab2ex3.res
piroc@svr24:~$ cat lab2ex3.res
PID 23246 - Debut du Pere
PID 23247 - Debut du premier fils [PPID 23246]
PID 23247 - Message 1 du premier fils -
PID 23247 - Message 2 du premier fils -
PID 23247 - Message 3 du premier fils -
PID 23247 - Message 4 du premier fils -
PID 23247 - Message 5 du premier fils -
PID 23247 - Message 6 du premier fils -
PID 23247 - Message 7 du premier fils -
PID 23247 - Message 8 du premier fils -
PID 23247 - Message 9 du premier fils -
PID 23247 - Message 10 du premier fils -
PID 23247 - Fin du premier fils
PID 23246 - Debut du Pere
PID 23248 - Debut du second fils [PPID 23246]
PID 23248 - Message 1 du second fils --
PID 23248 - Message 2 du second fils --
PID 23248 - Message 3 du second fils --
PID 23248 - Message 4 du second fils --
PID 23248 - Message 5 du second fils --
PID 23248 - Message 6 du second fils --
PID 23248 - Message 7 du second fils --
PID 23248 - Message 8 du second fils --
PID 23248 - Message 9 du second fils --
PID 23248 - Message 10 du second fils --
PID 23248 - Fin du second fils
PID 23246 - Debut du Pere
PID 23246 - Le PID du Fils 1 est 23247
PID 23246 - Le PID du Fils 2 est 23248
PID 23246 - Le Pere attend son Fils 1 [PID 23247]
PID 23246 - Le Fils 1 [PID 23247] s'est termine avec le statut 0000
PID 23246 - Le Pere attend son Fils 2 [PID 23248]
PID 23246 - Le Fils 2 [PID 23248] s'est termine avec le statut 0000
PID 23246 - Fin du Pere
piroc@svr24:~$
```

Figure 8 - Vers un fichier

Quelques explications s'imposent…

Lors de l'envoi vers la sortie standard, le contenu du buffer du *stdout*, bien qu'il ait une taille de 1024 bytes, est envoyé à l'écran dès qu'il y a un '\n', qu'il est plein ou qu'on lui applique un `flush`. On parle de *line buffered* ou buffer ligne à ligne.

De plus, nous avons déjà appris que lors du `fork`, bien que le processus fils hérite des variables du père, par la suite, le père et le fils ont des espaces d'adressage distincts. En revanche, leurs descripteurs de fichiers sont les mêmes. Donc, si l'un des deux processus modifie son pointeur de position dans un fichier en écrivant, par exemple, ça se répercute également chez l'autre. Dans le cadre de cet exercice, les trois processus partagent le descripteur de fichier du *stdout*, d'où l'affichage entrelacé qui en résulte.

Par contre, lorsque le résultat est envoyé vers un fichier, le contenu du buffer du fichier (défini dans la structure `FILE` associée à ce fichier) est envoyé à l'écran quand il est plein. On parle de *block buffered* ou buffer bloc à bloc. Il est à noter que la taille du buffer associé à un fichier est de 4096 bytes... bien plus que ce que chacun des processus a envoyé. Ils ne sont donc envoyés qu'à la fin du programme au moment du traitement de l'`exit`. Au moment du `fork`, le *buffer* (la structure `FILE *`) est aussi dédoublé… d'où la répétition du message "PID 23246 - `Debut du pere`".

La question, dont la réponse se trouve dans les explications précédentes, est de savoir comment procéder pour obtenir le même résultat dans le deuxième cas que celui obtenu dans le premier ?

À vous de jouer !

Faire un programme qui lance deux fils :

- le premier processus génère un fichier '`listeC.dat`' qui contient la liste des fichiers C du répertoire courant,
- le deuxième processus générera un fichier '`listeTxt.dat`' qui contient la liste des fichiers `txt` du répertoire courant.

Le programme principal attend la fin de l'exécution des deux fils, il génère ensuite un fichier '`listes.dat`' qui est le résultat de la concaténation des deux autres fichiers.

Utiliser dans cet exercice les appels fork, `wait`, `system` et `exit`.

COUP DE POUCE…

Voici les commandes pour chacun des processus :

- Fils 1

```
char cmd[50] = "ls *.c > listeC.dat";
```

- Fils 2

```
char cmd[50] = "ls *.txt > listeTxt.dat";
```

- Père

```
char cmd[50] = "cat listeC.dat listeTxt.dat > listes.dat";
```

Diverses sources utiles…

Sans plus de précision, voici quelques sites parmi ceux visités lors de la rédaction de ce document.

http://gcc.gnu.org/

http://lia.univ-avignon.fr/chercheurs/torres/downloads/inf3600.pdf

http://notesdecours.drivehq.com/courspdf/Cours%20OS_JNinforge_V1.1.pdf

https://openclassrooms.com/courses/la-programmation-systeme-en-c-sous-unix/les-processus-1

http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html

http://man.developpez.com/