Les processus et les pipes en C

Naïm Qachri, Stéphane Fernandes Medeiros, Eythan Levy, Cédric Meuter

Université Libre de Bruxelles

2010/2011

Création de processus(I)

- La création de processus se fait à l'aide du *system call* `fork()` :

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

pid_t fork();
```

- C'est la seule fonction C qui retourne deux fois !
- Au moment de l'appel du *fork*, le processus courant est ***cloné intégralement***
- L'exécution reprend des deux côtés à partir de l'endroit où le *fork* a été appelé et l'appel à `fork()` renvoie des valeurs différentes au deux processus :
  - Le clone (le processus fils) reçoit la valeur 0
  - L'original (le processus père) reçoit **le PID du processus fils**

Exercice(I)

- Ecrivez un programme C qui crée un processus fils. Les deux processus devront ensuite afficher à l'écran un message indiquant quel processus ils sont (père ou fils).

Correction(II)

- Exemple d'exécution :

```
[cmeuter@litpc34 source]$ gcc -Wall fork.c -o fork
[cmeuter@litpc34 source]$ ./fork

Processus fils (PID = 5974, PID du père = 5973)
Processus père (PID = 5973, PID du fils = 5974)
```

- Nous avons utilisé deux *system calls* supplémentaires dans cet exercice :

```c
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

- **`getpid`** permet d'obtenir le **PID d'un processus** et **`getppid`** (parent PID) permet, quant à lui, d'obtenir **le PID du père d'un processus**.

Correction(I)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

/* Exercice I: fork.c */

int main(int argc, char *argv[]) {
    pid_t pid = fork();

    if (pid == 0) // processus fils
        printf("Processus fils (PID = %d, PID du père = %d)\n", getpid(), getppid());
    else if (pid != -1) // processus père
        printf("Processus père (PID = %d, PID du fils = %d)\n", getpid(), pid);
    else
    {
        perror("Erreur durant le fork");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Exécution d'un programme(I)

- À partir d'un programme **C**, on peut lancer un autre programme en lui passant des paramètres (comme le *shell* le fait à partir d'une ligne de commande).
- L'exécution d'un programme se fait à l'aide de la famille de *system calls* `exec`
- Lorsqu'un processus fait appel à `exec`, le processus appelant est remplacé par le programme passé en paramètre.
- Si cela fonctionne, l'appel à `exec` **ne retourne pas !**

Exécution d'un programme(II)

Pourquoi une famille de *système calls* ? Il existe en fait plusieurs variantes de `exec` :

```
int execl(const char *program, const char *arg, ...);
int execlp(const char *program, const char *arg, ...);
int execv(const char *program, const char *argv[]);
int execvp(const char *program, const char *argv[]);
```

Les versions avec un `l` passe les paramètres du programme en utilisant une liste d'arguments terminée par `NULL`.

```
execl("/bin/ls", "/bin/ls", "-l", "-a", NULL);
```

Les versions avec un `p` cherche le programme dans la variable d'environnement `PATH`. Les versions sans `p` requiert donc un chemin absolu (partant du `/`).

```
execlp("ls", "ls", "-l", "-a", NULL);
```

Exécution d'un programme(III)

- Les versions avec v passe les paramètres du programme en utilisant un vecteur d'arguments terminé par `NULL`.

```
char *arguments[4];

arguments[0] = "/bin/ls";
arguments[1] = "-l";
arguments[2] = "-a";
arguments[3] = NULL;

execv("/bin/ls", arguments);
```

Attente de processus(I)

- Il est parfois nécessaire pour le processus père d'attendre la fin du processus fils avant de continuer son traitement (attente du résultat). Pour cela, on peut utiliser les *system calls* `wait` et `waitpid`.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- Le *system call* `wait` attend la fin d'un des processus fils, retourne **son PID et stocke** (entre autre) **dans l'entier pointé par `status`** (si status est différent de NULL), **le code de retour du processus fils** (ce qui est renvoyé par la fonction `main` du processus fils).

Attente de processus(II)

- Le *system call* **`waitpid`** est très similaire à **`wait`**. Dans ce cas, le processus père attend la fin du processus fils dont **le PID est précisé en paramètre** (dans le cas où plusieurs fils ont été créés). Le paramètre entier `option` permet (entre autre) de rendre l'appel à `waitpid` **non bloquant** avec la constante `WNOHANG`. Si cela n'est pas nécessaire, on utilise la valeur 0.

- Pour extraire le code de retour du processus fils de `status`, il faut d'abord tester que le processus s'est bien terminé correctement. Pour cela, on utilise la macro `WIFEXITED(status)`, qui renvoie **vrai si le processus fils s'est terminé normalement.** Ensuite pour obtenir son code de retour, on utilise la macro `WEXITSTATUS (status)`.

Exercice (II)

- Ecrivez un programme `launch` qui exécute une commande passée via les arguments du programme dans un processus fils, qui attend sa valeur de retour et qui l'affiche à l'écran. Par exemple, `./launch ps -f` devra exécuter la commande `ps -f` et afficher sa valeur de retour.

Correction (III)

- Squelette du programme

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
   pid_t pid;
   int status, i;
   char **argument;

   pid = fork();

   if (pid == 0)
   {
        /* processus fils */
   }
   else if (pid != -1)
   {
        /* processus père */
   }
   else {
        perror("Erreur durant le fork");
        return EXIT_FAILURE;
   }

   return EXIT_SUCCESS;
}
```

Correction (IV)

- Programme du fils :

```
argument = (char **)malloc(argc * sizeof(char *));

for (i = 0; i < argc - 1; ++i)
   argument[i] = argv[i + 1];

argument[argc-1] = NULL;

if (execvp(argv[1], argument) == -1)
{
   perror("Erreur durant l'exec");
   return EXIT_FAILURE;
}
```

- Programme du père :

```
printf("En attente du fils\n");
wait(&status);

if (WIFEXITED(status))
   printf("Le code de retour du fils est %d\n", WEXITSTATUS(status));
else
   printf("Le processus fils ne s'est pas terminé correctment\n");
```

Correction (V)

- **Exemple d'exécution:**

```
[cmeuter@litpc34 source]$ ./launch ./launch  ps -f
En attente du fils
En attente du fils
UID        PID  PPID  C STIME TTY          TIME CMD
cmeuter   2954  2953  0 11:06 pts1     00:00:00 /bin/bash
cmeuter   9620  2954  0 17:25 pts1     00:00:00 ./launch ./launch ps -f
cmeuter   9621  9620  0 17:25 pts1     00:00:00 ./launch ps -f
cmeuter   9622  9621  0 17:25 pts1     00:00:00 ps -f
Le code de retour du fils est 0
Le code de retour du fils est 0


[cmeuter@litpc34 source]$ ./launch ./failure
En attente du fils
Le code de retour du fils est 1
```

Les *pipes*

- Les *pipes* permettent à deux processus tournant sur la même machine de communiquer entre eux. Ils fonctionnent comme des “tuyaux” (*pipes* en anglais) unidirectionnels entre deux processus.
- Il existe **deux** types de *pipes* :
  - Les *pipes* **anonymes** utilisés pour la communication entre parents (le père avec un fils, un fils avec un autre fils).
  - Les *pipes* **nommés** utilisés pour la communication entre deux processus indépendants (pas de lien de parenté direct entre les deux)

Les *pipes anonymes*

- Le création de *pipes* anonymes se fait à l'aide du *system call* `pipe`.

```
#include <unistd.h>

int pipe(int fd[2]);
```

- L'appel à `pipe` crée une paire de *file descriptors* stockée dans `fd`. `fd[0]` sert à la lecture du *pipe* et `fd[1]` sert à l'écriture dans le *pipe*.

- Pour établir une communication (unidirectionnelle !) à l'aide d'un pipe, il faut que le processus père crée un *pipe* avant de faire un (ou plusieurs) appel(s) à `fork`. Après le fork, le processus père et le(s) processus fils peuvent accéder aux *pipes* en utilisant `fd`.

- Pour une communication bidirectionnelle, il faut créer deux *pipes*.

Les *pipes nommés*

- Le création de *pipes* nommés se fait à l'aide du *system call* suivant:

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

- Une fois créé, le *pipe* nommé apparaîtra comme un fichier spécial dans le système de fichier, à l'endroit indiqué par `pathname`. Le paramètre `mode` quant à lui, permet de donner les permissions (lecture, écriture, ...) à ce fichier spécial (pour plus de détails sur les constantes utilisables pour `mode`, consultez les man pages de `open(2)`).

Attention!

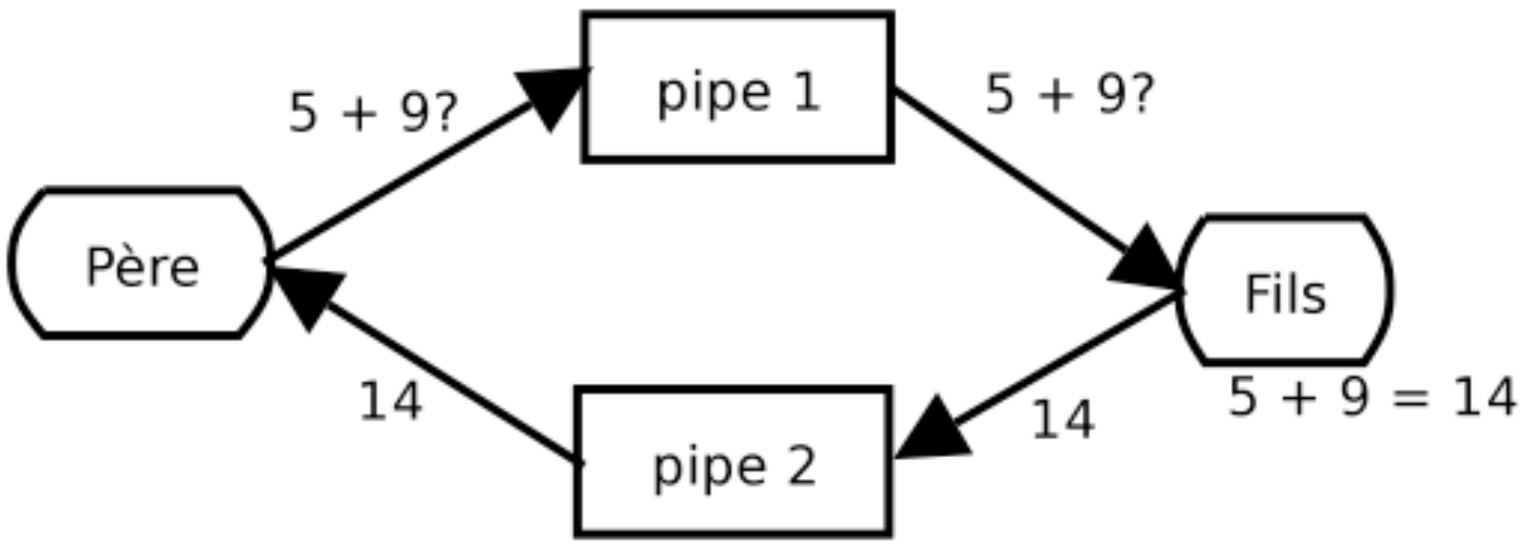
- Contrairement aux ***pipes anonymes,*** les fichiers continuent d'exister après la fermeture de ces derniers. Pour les effacer, il suffit d'utiliser le *system call* `unlink()`

```c
#include <unistd.h>

int unlink(const char *path);
```

Exercices (III)

- Ecrivez un programme qui crée un processus fils. Le processus père demandera ensuite deux nombres entiers à l'utilisateur qui seront transmis via un *pipe* au processus fils, qui en calculera la somme. Cette somme sera renvoyée au processus père via un (autre) *pipe*.



Correction (VI)

- Squelette du programme:

```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
   pid_t pid;
   int pere_fils[2], fils_pere[2], nbr1, nbr2, somme;

   if (pipe(pere_fils) == -1 || pipe(fils_pere) == -1)
   {
        perror("Erreur à la creation des pipes");
        return EXIT_FAILURE;
   }
   pid = fork();
   if (pid == 0)
   {
        /* processus fils */
   }
   else if (pid != -1)
   {
        /* processus père */
   }
   else
   {
        perror("Erreur durant le fork");
        return EXIT_FAILURE;
   }

   return EXIT_SUCCESS;
}
```

Correction (VII)

- Programme fils:

```
close(pere_fils[1]);
close(fils_pere[0]);

read(pere_fils[0], (void *)(&nbr1), sizeof(int));

read(pere_fils[0], (void *)(&nbr2), sizeof(int));

somme = nbr1 + nbr2;

write(fils_pere[1], (void *)(&somme), sizeof(int));
```

Correction (VIII)

- Programme père:

```
close(fils_pere[1]);
close(pere_fils[0]);

printf("Introduisez deux nombre : ");
scanf("%d", &nbr1);
scanf("%d", &nbr2);

write(pere_fils[1], (void *)(&nbr1), sizeof(int));
write(pere_fils[1], (void *)(&nbr2), sizeof(int));

read(fils_pere[0], (void *)(&somme), sizeof(int));
printf("La somme des deux nombres est %d\n", somme);
```

Création de threads(I)

- La création de *thread* se fait à l'aide de plusieurs *system calls auquel* faut passer en paramètre le pointeur de la fonction à exécuter en //:

```
#include <pthread.h>
pthread_t thread;

int pthread_create(pthread_t* thread, pthread_attr_t* attr, void*
(*start_routine)(void*), void* arg);
```

- Au moment de l'appel du *thread*, le processus courant est ***cloné partiellement***
- L'exécution reprend d'un côté à partir de l'endroit où le *thread* a été appelé et de l'autre dans la fonction.
- Vous devez compiler avec une directive particulière avec gcc
-

```
gcc nomSource.c -lpthread -o nomExecutable
```

Création de threads(II)

- La fonction passée en paramètre sera toujours du style

```
void* ma_fonction(void* data);
```

- Il est possible que vous deviez parfois savoir vous synchroniser avec la fin d'un des *thread,* pour cela vous devrez utiliser le *system call suivant:*

```
int pthread_join(pthread_t thread, void **thread_return);
```

thread_return est un pointeur sur la valeur de retour du *thread* **attendu**.

- Pour terminer un *thread,* on utilisera le system call suivant:

```
void pthread_exit(void *retval);
```

- Pour qu'un *thread* en achève ou coupe un autre *thread,* il sera possible d'utiliser le *system call* suivant:

```
int pthread_cancel(pthread_t thread);
```

Création de threads(III)

- La création d'un *thread* vous permet d'accéder **au même espace de données** que la fonction appelante, ce qui signifie que vous pourriez avoir **des problèmes d'accès concurrentiels** aux données.

- Pour résoudre ce problème, nous vous montrerons, dans un des chapitres suivants comment empêcher ce genre de problèmes.

- Pour le moment, vous ne initialiserez et utiliserez que les variables définies dans votre *thread*.

Exercice(I)

- Ecrivez un programme C qui crée un *thread*. Les deux processus devront ensuite afficher 100 fois à l'écran un message indiquant soit 1 si c'est le code principale, soit 2 si c'est le code du *thread*.

Correction(I)

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

void* maFonction(void* data);

int main()
{
    int i;

    pthread_t thread;// On crée un thread
        pthread_create(&thread, NULL, maFonction, NULL);// Permet d'exécuter le fonction maFonction
    en parallèle

    for(i=0 ; i<100 ; i++)
        printf("1");

    // Attend la fin du thread créé
    pthread_join(thread, NULL);

    return EXIT_SUCCESS;
}

void* maFonction(void* data)
{
    int i;

    for(i=0 ; i<100 ; i++)
        printf("2");

    return NULL;
}
```