Système.

McRoss & Firegrain ofc

25 octobre 2014

Introduction

bouyou ofc

Chapitre 1

Processus

1.1 La notion de processus

Un processus, c'est une *tâche en cours d'execution*. C'est en même temps du code et des données. Pour généraliser grossièrement, et pas de façon absolument vraie, un processus peut être représenté comme étant un programme en cours de fonctionnement.

1.2 Création d'un processus

Un processus est créé par le biais de l'appel système *fork( )*. *fork( )* fait la copie du processus père en un nouveau processus. Ce nouveau processus, qu'on appelle *processus fils*, exécutera le même code que le processus père [1] et *héritera* des données du père (variables par exemple) mais aussi de l'environnement du processus père. Si, par exemple, le processus père modifie des *variables d'environnement*, le fils héritera de ces variables modifiées. Bien sur, qui dit nouveau processus, dit nouveaux identifiants pour le nouveau processus.

Cependant, et c'est un détail important, les données dupliquées entre le père et le fils ne sont pas immédiatement copiées ; c'est à dire que tant que le fils et le père partagent des informations communes, cette information existe en *un seul exemplaire* dans le système. Par contre, si jamais l'un des deux décide de modifier cette information, c'est à ce moment que cette information est dupliquée. C'est la méthode de *copie sur écriture*.

Enfin, dernière note : il faut savoir qu'il n'y **pas** d'ordre d'exécution prédéfini entre un processus fils et un processus père. Cela veut dire que le fils peut prendre la main a tout moment et le père peut en faire de même.

1.3 Les états

Un processus peut avoir plusieurs états (un seul à la fois) :
— RUNNING : le processus est actuellement en train de faire son job ;
— WAITING/SLEEPING : le processus est en attente de ressources ou d'événements extérieurs ;
— STOPPED : le processus est stoppé par un *signal* et ne reprendra que par un *signal de redémarrage* ;
— TERMINATED : le processus a fini son travail. [2]

1. Bien sur, le code qui se trouve après le *fork*.
2. Dans le cas où le processus a fini son travail mais que son père n'a pas encore lu son code de retour avec *wait()*, il reste alors présent dans la table des processus tant qu'on a pas lu son code de retour ; on parle d'état ZOMBIE dans ce cas.

Chapitre 2

Exécution de programmes

2.1 Lancement d'un programme

On a vu que la création d'un processus se fait par le biais de *fork*. Pour un programme, on utilise l'appel système *exec( )* (ou du moins l'une des fonctions de la famille exec( )).
Il existe deux fonctions : execl et execlp. Ces deux fonctions fonctionnent de la manière suivante :

- execl("*chemin absolu*", "*nom de l'application*", "*argument*", ..., NULL)
- execlp("*nom de l'executable*", "*nom de l'application*", "*argument*",...., NULL)

La différence entre les deux tient dans le chemin que l'on va indiquer à la fonction : execl prend en paramètre le chemin absolu (par exemple /usr/bin/vim) alors que execlp prend en paramètre le nom direct de l'application et cherchera directement dans le $PATH.

Le lancement d'un nouveau programme *remplace totalement l'ancien*, c'est à dire que tout ce qui est *code segment, segment mémoire*, etc est réinitalisé pour le nouveau programme. Par contre, il n'y a **pas** de **nouveau processus** créé. Le processus qui lance un programme garde **le même** PID, PPID, etc.

Dans le cas où on cherche à lancer un nouveau programme sans remplacer pour autant le processus en cours, il existe 2 fonctions qu'on ne détaillera pas [1] : *system( )* et la paire *popen( )/pclose( )*. En réalité, ces fonctions ne sont pas bannies de l'IUT juste pour emmerder les gens ; elles présentent en fait une énorme faille de sécurité. [2]

2.2 Fin d'un programme

Un programme peut mettre fin à son execution soit de manière *normale* (abandon par l'utilisateur, tâche finie, return, exit) ou soit, vous l'aurez deviné, de manière *anormale* (arrêté par un signal quoi).

2.2.1 Arrêt normal

Comme vous le savez, le moyen *propre* d'arrêter un programme se fait par le biais de *exit( )* ou *return( )*. La différence entre les deux est que *exit( )* peut être utilisé partout, alors que *return( )* ne quitte le programme que si on l'appelle depuis le *main*.

1. leur utilisation amène à une lapidation par petits cailloux pointus rouillés atteints du sida
2. programme compilé en root, script *rm -r /* déguisé = boum

2.2.2 Arrêt anormal

Lorsque un programme fait une boulette (par exemple, il tente d'accéder au contenu d'un pointeur non initialisé), un *signal* est émis et arrête le programme tout en créant un dump mémoire. [3]
Mis a part ça, on peut arrêter proprement mais *anormalement* (heh) un programme en utilisant *abort( )* et *assert( )*. Ces deux fonctions émettent des signaux et permettent un débogage du programme.

2.2.3 Dans le cas d'un processus fils

Imaginons que nous avons un processus fils. Que ce processus se soit arrêté normalement ou pas, son processus père se doit de lire son code de retour, afin que le processus fils n'erre indéfiniment dans les limbes des processus.
Pour cela, on utilise *wait( )*. Cette fonction (ou l'une des fonctions de la famille wait()) permet au processus père de lire le code de retour d'un proc fils. En fait, le processus père *attend* que l'un des processus fils se termine. *wait( )* peut prendre en paramètre l'adresse d'un int afin de connaitre les circonstances de la mort du processus fils. On le fait de manière suivante :

```
int status; /* le int qui nous permet d'avoir le code retour */

/* ici on met le code avec le fork() et tout le tintamarre */

wait(&status); /*on attend la mort du fils*/

if (WIFEXITED(status)) /* on vérifie maintenant comment le fils est mort */
        printf("code %d",WEXITSTATUS(status));
else if ... /* bla bla bla */
```

Voilà. Il existe 3 macros qui permettent de savoir comment le fils est mort :

- WIFEXITED, qui est vraie si le programme s'est quitté tout seul (va avec WEXITSTATUS afin de connaitre le code retour)
- WIFSIGNALED, si le programme a été tué par un *signal* (exemple : SIGKILL aka kill -9). Utiliser WTERMSIG afin de connaitre quel signal a tué le processus.
- WIFSTOPPED, si le programme est temporairement stoppé par un signal (bien souvent SIGSTOP).

Enfin, notons qu'il faut autant de *wait( )* que de processus fils lancés !

3. On verra tout ça dans le prochain chapitre youpi super cool génial [P-REC] Bookmark.

Chapitre 3

Les signaux

3.1 Qu'est-ce qu'un signal ?

Un *signal* est une sorte de message envoyé par un processus à un autre processus. Face à un signal, un processus peut effectuer une des actions au choix :

- *Ignorer* le signal, mais attention, ce n'est pas possible pour tous les signaux (par exemple, il est impossible d'ignorer SIGKILL) ;
- *Capturer* le signal, c'est à dire exécuter une procédure particulière quand le programme reçoit un signal particulier ;
- *Laisser le kernel s'en occuper*, c'est a dire laisser l'action par défaut définie par chaque signal (exemple : se terminer anormalement pour SIGINT, ou ne rien faire pour SIGCHILD)

Un signal est émis soit :

- *Par le système*, lorsque il détecte quelque chose (instruction illégale, fin d'un processus fils, ~~rencontre avec ma GROSSE...~~)
- *Par l'utilisateur* lui même, par le biais de commandes (CTRL+C, CTRL+U, etc), par la fonction *kill( )*, ou encore avec des fonctions (*raise( ), kill( ), etc*)

3.2 Les principaux signaux

3.2.1 Signaux de terminaison (sigint, sigkill, sigquit, sigterm)

- SIGINT, code 2, appelée aussi avec CTRL+C, correspond à un SIGnal d'INTerruption, donc termine le processus. Peut être ignoré/capturé.
- SIGQUIT, code 3, appelée aussi avec CTRL+\, termine le processus mais créé un fichier core/dump. Peut être ignoré/capturé.
- SIGKILL, code 9, **tue** (pas proprement) le processus à **tout les coups**. À utiliser en dernier recours. Ne peut **pas** être ignoré/capturé.
- SIGTERM, code 15, tue **proprement** le processus. Correspond à l'action par défaut de la fonction *kill( )*. Peut être ignoré/capturé.

3.2.2 Signaux de gestion de processus (sigchld, sigstop ou sigstp, sigcont, sigalrm, sigusr1&sigusr2)

- SIGCHLD, code 17, envoyé au processus père lorsque l'un des processus fils s'est terminé. Est ignoré par défaut. Peut être ignoré/capturé.

— SIGSTOP\SIGSTP, code 19 et 20. Stoppe le processus (voir états des processus). La différence entre les deux tient dans le fait que SIGSTOP est pas ignorable ou capturable alors que SIGSTP l'est.
— SIGCONT, code 18, relance le processus si il est stoppé. Ne peut pas être ignoré/capturé.
— SIGALRM, code 14, est envoyé lorsque le temps défini par l'appel système *alarm( )* a expiré.
— SIGUSR1&SIGUSR2, code 10 et 12, sont des signaux *personnalisables*. Par défaut, ils terminent le processus, mais on peut les capturer pour faire autre chose avec.

3.2.3 Signaux d'erreur (pas très important)

Par défaut, ces signaux arrêtent le processus.
— SIGBUS & SIGSEGV : adresse mémoire invalide ou erreur de segmentation.
— SIGFPE : erreur de calcul (ex : division par 0).
— SIGXCPU & SIGXFSZ : limite d'utilisation de ressources dépassé.
— SIGILL : instruction assembleur illégale.

3.3 Les fonctions C

Il existe une ribambelle de fonctions C pour émettre/capturer/attendre/~~exploser~~ un ou des signaux.

3.3.1 Émettre un signal

Il existe ici deux fonctions, *kill( )* et *raise( )*. La différence entre les deux est que *raise( )* envoie le signal au processus qui l'appelle, alors que *kill( )* permet d'envoyer un signal à n'importe quel processus (suivant les droits bien_entendu).
kill( ) s'utilise la manière suivante :

```
kill(<pid du proc visé>, <code du signal>);
```

kill( ) retourne un int suivant sa réussite. Il faut noter qu'on peut utiliser "-1" en tant que pid pour envoyer le signal à tous les processus (a part init, of_course).

Petite note supplémentaire : il existe une fonction *alarm( )* qui prend en paramètre un nombre de secondes. Cette fonction permet d'envoyer un SIGALRM au processus en cours (c'est à dire *soi même*) après le nombre de secondes passé en paramètre.

3.3.2 Attendre un signal

La fonction *pause( )* permet de mettre le programme en attente d'un signal. Le programme se mettra en pause tant qu'il n'aura pas reçu de signal.
Note : il ne faut **pas** utiliser de *sleep( ), wait( ),* ou autres conneries pour attendre un signal. Il faut **toujours** utiliser *pause( ).* [1]
pause( ) s'utilise de la manière suivante :

```
pause( );
```

3.3.3 Gérer les signaux

Il faut d'abord créer une **procédure** qui prendra en paramètre le signal. Cette procédure s'occupera du traitement du signal.
Notez que cette procédure prend exclusivement **1 paramètre**, un entier. Pour les détails (ou pour bluffer 2/3 pequenots), cet entier représente la constante symbolique du signal. Cet entier sera rempli automatiquement

1. au risque de se faire taper dessus par CARLOS

par le système d'exploitation.
Voyons un exemple :

```
void gestSig(int sig)
{
        /* on met nos traitements par signal */
        /* par exemple, si le proc reçoit SIGINT */
        if (sig==SIGINT)
                printf("CTRL-C recu\n");
}
```

Ici, cette procédure gère le signal SIGINT. Notez qu'on est *pas obligé* de faire le test. On peut, par exemple, créer autant de procédures que de signaux que l'on souhaite traiter, ou ne créer qu'une seule procédure qui sera utilisé par tous les signaux.
Ensuite, il faut utiliser *signal( )* ou *sigaction( )*. On se penchera ici sur *signal( )*.

```
/* on imagine qu'on a notre gestionnaire de signal "gestSig" qui affiche "oui"
 * dés qu'on reçoit un SIGINT */
int main()
{
        signal(SIGINT, gestSig);
        pause();
        /* on reçoit CTRL-C aka SIGINT */
        exit 0;
}
```

Ce qui donnera à l'affichage, lorsque on execute le programme "prog" au dessus :

```
>>> prog
(on tape CTRL-C)
>>> oui
(fin du prog)
>>>
```

ez aussi non ??????
À noter : il existe aussi des constantes pour gérer les signaux, comme SIG_IGN par exemple. On utilise ces constantes de la manière suivante :

```
signal(SIGINT, SIG_IGN);
/* SIGINT sera ignoré */
```

Chapitre 4

Gestion de fichiers

"Sous UNIX, tout est fichier."

les randoms gens sur internet

4.1 Définition d'un fichier

Un fichier possède des attributs qui différent entre les systèmes de fichiers (*ext2,ext4,fat32,...*). Pour ne pas se compliquer la vie, sachez qu'un fichier possède un nom (O RLY ?), le nom de son créateur, sa taille, la date de création, ... Bon c'est pas très important.

Comme disent les *randoms gens sur internet*, **tout est fichier sous Linux/Unix**. Cela ne veut pas forcément dire que *tous* les fichiers sont de *même type*.
Ainsi, il existe plusieurs types de fichier sous Linux/Unix :

- **Les fichiers ordinaires**, c'est a dire les fichiers random comme *merci.c*, *i_et_y_se_font_defoncer.mp4*, *mcross.sh*, ...
- **Les répertoires**, c'est a dire les... répertoires et dossiers qui contiennent des fichiers.
- **Les fichiers spéciaux**, qui eux mêmes existent en deux type : les fichiers caractères, qui représentent les périphériques d'entrée-sorties (par exemple, le clavier est représenté sous la forme d'un fichier, allez voir dans */dev/*), et les fichiers bloc, qui représentent tout ce qui est disque et support de stockage.
- **Les liens**, qui ne comportent qu'un pointeur vers un fichier.
- **Les pipes**, et plus particulièrement les *pipes nommés*, qui existent sous forme de fichier sur le système.

4.2 Système de fichiers sous Unix/Linux

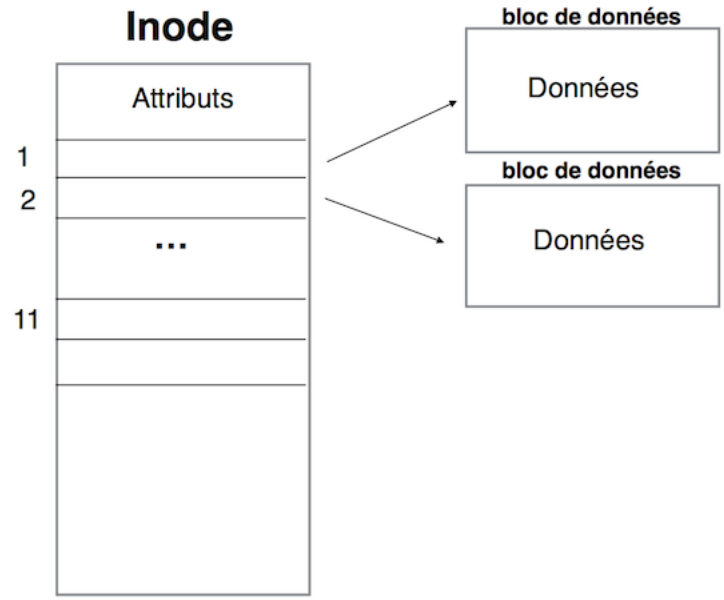
4.2.1 Les fichiers ordinaires

La gestion des fichiers sous U/L [1] repose sur la notion d'**inode** (ou i-noeud).
Un inode est un *index* avec des cases. Pour des raisons de commodité, on va dire que la première case comporte les attributs du fichier. Ensuite, il y a 10 cases (ou 12 cases, sous Linux) qui comportent chacune une adresse vers un bloc de données. Après ces 10 (ou 12 cases), il y a 3 cases *spéciales*.
Ces 3 cases sont spéciales car elles permettent ce qu'on appelle l'*adressage indirect*.

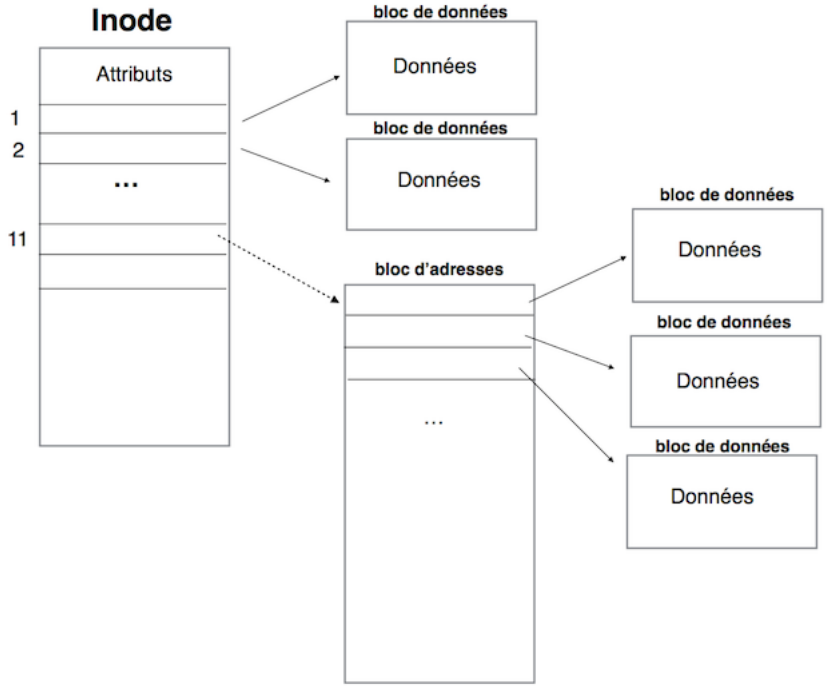
Mr. Toutlemonde : "waaaa c quoi l'adressage ??????"

1. vu que j'ai la flemme de réécrire Unix/Linux partout, je vais abréger ça par U/L

Du calme, Mr.Toutlemonde. L'adressage signifie que la case donne l'adresse d'un bloc de données :



On a ici de *l'adressage direct*. Voyons maintenant l'*adressage indirect*. :



Comme vous le voyez, l'adressage indirect s'appelle comme ça car l'adresse contenue dans la case pointe vers *un bloc d'adresses*, et non pas **directement** un bloc de données. Chacune des adresses contenue dans le bloc d'adresses pointe vers un bloc de données.
Ainsi, la case 11 pointe vers un bloc d'adresse, la case 12 pointe vers un bloc d'adresses dont chaque adresse

pointe vers **un autre** bloc d'adresses, et dans ce bloc d'adresse chaque adresse pointe vers un bloc de données (adressage indirect double), et enfin pour la case 13 on applique l'adressage indirect triple [2].

Mr. Toutlemonde : "mé là y a CARLOS ki me demande 2 calculé la taille maxi d'1 féchier, cmnt je fé ????"
Bon, j'imagine que *Mr. Toutlemonde* parle de l'exercice 1 du TD4. On va le faire ensemble avec les calculs.

Exercice 1 : Un système de gestion de fichiers sous Unix utilise des **blocs de 4096 octets** et code les **adresses sur 32 bits**.

Oui merci maintenant tg l'exercice. Tout d'abord, notez bien les informations en gras, ce sont les infos les plus importantes. Vu que 1 octet = 8 bits, une adresse tient sur **4 octets**.
Ensuite, on sait qu'un bloc tient sur 4096 octets, ce qui correspond à **4 ko**. On en déduit qu'un bloc contient donc **1024 adresses**.

Maintenant, on peut calculer la taille selon le type d'adressage :
- Direct : on a 10 blocs de 1024 adresses, donc 4096 octets
- Indirect simple : on a 1024 blocs, donc 4 Mo
- Indirect double : on a 1024*1024 blocs, donc 4 Go
- Indirect triple : on a 1024*1024*1024 blocs, donc 4 To

On fait maintenant la somme ; la taille max correspond à un fichier de 4 To et 4 Go et 4 Mo et 4096 octets.

4.2.2 Les répertoires

Les répertoires possèdent une structure différente entre UNIX et Linux.

Unix Sous Unix, un dossier est une table avec 2 entrées ; le numéro de l'inode du fichier sur 2 octets et son nom sur 14 octets. Cela se présente de la manière suivante :

N°Inode (2 oc)	Nom du fichier (14 oc)
432432	merci.c
9888889	ofc1
7893427489	sysfiche.tex
...	...

2. j'ai la flemme de détailler, mdr

Linux Sous Linux, c'est un peu plus compliqué :

Chaque case doit faire un multiple de 4 octets.					
N°Inoeud (2oc)	Taille de l'entrée (2oc)	Longeur du nom fichier (2oc)	Type du fichier (2oc)	Nom (chaque partie doit faire 4 oc)	
21	12 (8+4)	1	2 (repertoire)	. \0 \0 \0	
41	12 (8+4)	2	2 (repertoire)	. . \0 \0	
32	16 (8+8)	6	1 (fichier ordinaire)	E x o 1	. c \0 \0
39	16 (8+8)	8	1 (fichier ordinaire)	m i k u	. p n g
1337	12 (8+4)	3	2 (repertoire)	r e p \0	

La taille de l'entrée correspond à la somme des caractéristiques (8 octets) plus le nom du fichier. Vous devez vous demander à quoi correspondent les *\0* ; en fait, chaque case dans les noms de fichier doit faire **4 octets** ; si on ne remplit pas la case, on ajoute des \0. C'est tout.

4.3 Manipulation des fichiers en C

Parce oui, il faut bien jouer avec les fichiers en C.
Ce qu'il faut savoir, c'est que chaque processus possède une *File Descriptor Table*, soit, en bon français, une *Table de Descripteurs de fichiers*. En fait, un fichier est identifié au niveau système par un entier que l'on appelle *file descriptor*. La table contient donc des descripteurs de fichiers qui pointent chacune sur un fichier. Attention : les descripteurs 0, 1 et 2 sont réservés pour 3 fichiers très spéciaux, respectivement *stdin*, *stdout* et enfin *stderr*. Cela veut aussi dire que **chaque programme** possède dans sa table de descripteur de fichier les descripteurs 0, 1 et 2.
Enfin, le système d'exploitation possède une table commune à tous les processus ; c'est la *Table des fichiers ouverts*. Cette table permet au système de savoir quel fichier est ouvert par qui et avec quels droits. On ne va pas s'étendre dessus, c'est pas très intéressant.

Sur U/L, il faut passer par plusieurs étapes afin d'utiliser un fichier dans un programme C : il faut **ouvrir** le fichier, faire son petit traitement (**lecture, écriture**, etc) puis **fermer** le fichier. Tout cela se fait par le biais d'appels systèmes que nous allons détailler.

4.3.1 Ouvrir un fichier avec *open( )*

open( ) se présente de la manière suivante :

```
int open( const char *fichier, int flag );
```

Commençons par les paramètres. **fichier* représente ici le nom du fichier **et son chemin**. Le *flag* représente en fait une constante définie dans *fcntl.h* qui définit les conditions dans lesquelles on souhaite ouvrir le fichier. Il en existe 4 d'importantes :

- O_RDONLY, comme Open_ReadONLY, va ouvrir le fichier en lecture uniquement
- O_WRONLY, ouvre le fichier en écriture uniquement
- O_RDWR, ouvre le fichier en lecture et écriture
- O_APPEND, ouvre le fichier en écriture **en fin de fichier**

Enfin, *open( )* renvoie un entier (ou -1 si il se plante), qui correspond en fait au fameux **file descriptor**. Il est donc primordial de récupérer la valeur de retour d'open !

Exemple Imaginons que l'on souhaite ouvrir un fichier *fic* se trouvant à */home/mooshi/rep* en lecture et écriture, sachant que l'on se trouve dans un autre répertoire que le fichier *fic* ; on doit écrire :

```
int fd = open("/home/mooshi/rep/fic", O_RDWR);
```

Et c'est gagné.

Création d'un fichier À noter : *open()* permet aussi de créer un fichier, selon les flags que l'on met. Si on utilise *O_CREAT*, on doit rajouter un paramètre pour les droits. Regardez l'exemple, ça sera plus clair.

```
/* Je souhaite créer un fichier oui avec des droits pour tout le monde */
int fd_oui = open("oui", O_CREAT, 0777);

/* Je souhaite créer un fichier merci avec des droits que pour moi
et seulement si il existe pas déjà */
int fd_merci = open("merci", O_CREAT | O_EXCL, 0700);
```

Le 3eme paramètre est un octal qui définit les droits. Vous vous rappelez de *chmod* ? C'est la même chose.

4.3.2 Lire dans un fichier avec *read( )*

Regardons *read( )* :

```
size_t read( int fd, void *buf, size_t nbOct )
```

Mr. Toutlemonde : omg c'est quoi ce size_t ???????

size_t correspond juste à un int. C'est juste pour montrer que ce int désigne une taille en octets.
Maintenant, regardons les paramètres. En premier, on retrouve notre cher *file descriptor* que l'on a récupéré un peu plus tôt.
Pour le deuxième paramètre, il va falloir éclaircir la notion de **buffer**. Un buffer est une sorte de paquet d'une taille définie dans laquelle on va mettre des données. Très généralement, c'est un tableau de caractère qui occupe le rôle de buffer.
Enfin, le dernier paramètre correspond au nombre d'octets que l'on souhaite lire dans le fichier.
read( ) retourne le nombre d'octets effectivement lu dans le fichier.

Exemple Reprenons notre fichier fic du premier exemple. On souhaite lire son contenu sur 20 octets.

```
char buf[20]; // On cree un buffer de taille 20 octets
int nbOctetsLus = read(fd, buf, 20); //On lit 20 octets du fichier
```

Mr. Toutlemonde : ouais mais il va où le contenu du fichier ???
Ben il va ~~dans ton cul~~ ENFIN, non, mais on va voir ça tout de suite avec *write*.

4.3.3 Écrire dans un fichier avec *write( )*

En fait, les données lues vont dans notre buffer, c'est à dire, dans l'exemple précédent, *buf*. *write( )*, au lieu d'écrire dans le buffer depuis le fichier, va écrire dans le fichier depuis le buffer. Ainsi, on peut noter de nombreuses similitudes entre *write( )* et *open( )* :

```
size_t write(int fd, void *buf, size_t nbOct)
```

Ici, nbOct représente le nombre d'octet que l'on souhaite écrire dans le fichier et cette primitive renvoie ici le nombre d'octets effectivement écrits dans le fichier.
Souvent, on fait travailler *open* et *write* à la suite, puisque on utilise plusieurs fois le buffer.

Exemple On reprend l'exemple du dessus. Maintenant qu'on a lu 20 octets dans notre buffer, on va essayer d'écrire dans un nouveau fichier.

```
int newFic = open("fic2", O_CREAT, 0755); //On créé un nouveau fichier "fic2"
if (newFic != -1)
        int nbOctetsEcrits = write("fic2", buf, 20);
```

Voilà. On a écrit 20 octets dans un nouveau fichier fic2.

4.3.4 Se déplacer dans un fichier avec *lseek( )*

Pas important. On verra plus tard.

4.3.5 Fermer un fichier avec *close( )*

Là, c'est facile. *close( )* ne prend qu'un paramètre, et c'est le *file descriptor*.

Exemple Maintenant qu'on a écrit et lu nos fichiers fic et fic2, on les ferme.

```
close(fd);
close(newFic);
```

Chapitre 5

Pipes

Les processus (pour l'instant d'une même machine) peuvent communiquer entre eux par le biais de *pipes*. Un pipe, c'est un tube par lequel on fait passer des informations. Un processus peut écrire et lire dans un pipe, cependant il n'y a **qu'un seul lecteur et qu'un seul écrivain par pipe**. Si on veut que deux programmes communiquent dans les deux sens, il faut créer deux pipes. On va commencer par les **tubes anonymes**. [1]

5.1 Tubes anonymes

5.1.1 Créer un tube

Afin de créer un tube, il faut utiliser l'appel système *pipe( )*. Cet appel système prend en paramètre un tableau d'entier de taille 2. Regardez l'exemple :

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

Pourquoi un tableau d'entier de taille 2 ? Tout simplement car on aura dans ce tableau les *file descriptor* qui correspondent à l'entrée du tube (ici **desc[1]**) et la sortie du tube (ici **desc[0]**). Bien sur, je pars du principe que vous savez ce que sont les *file descriptor*, donc continuons.

5.1.2 Écrire dans un tube

Vous vous souvenez de *write( )* qu'on utilisait pour écrire dans un fichier ? C'est la même chose ici, sauf qu'on met le file descriptor de l'entrée du tube. On a donc :

```
char buffer[20]; // On a notre buffer de 20 octets
write(desc[1], buffer, 20); // On souhaite écrire dans notre pipe 20 octets de notre buffer
```

Et voilà, c'est gagné.

5.1.3 Lire depuis un tube

Là encore, ça ressemble beaucoup à *lire dans un fichier* :

```
char bufferLecture[20];
read(desc[0], bufferLecture, 20); //On veut lire 20 octets depuis le pipe dans notre buffer
```

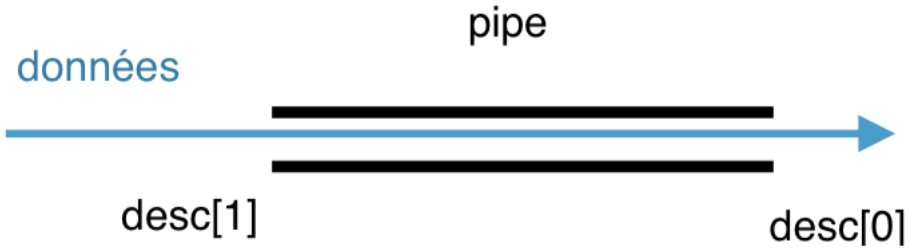
1. et on va faire que ça parce bon

5.1.4 Communiquer entre deux processus

Le problème des tubes anonymes, c'est que les deux processus qui veulent communiquer doivent forcément connaitre les file descriptor. Ainsi, il faut que l'un des processus soit créé après la création du pipe. Par exemple :

```
int main()
{
        int desc[2];
        pipe(desc);

        pid_t fils = fork();

        if(fils==0)
        {
                close(desc[0]);
                char bufferEcriture[20];
                write(desc[1], bufferEcriture, 20);
                exit(0);
        }

        wait(NULL);
        char bufferLecture[20];
        close(desc[1]);
        read(desc[0],bufferLecture,20);
        exit(0);
}
```

Il faut aussi s'assurer de **fermer** les extrémités - comprendre les *desc[]* - inutilisés par chaque processus. Dans l'exemple ci-dessus, le fils n'est qu'un écrivain ; par conséquent, on ferme l'extrémité de lecture - desc[0] -. De même pour le père, qui n'est que lecteur, on ferme d'abord l'extrémité d'écriture - desc[1] -. Et voilà, on a bien établi une communication entre 2 processus !
On peut résumer tout ça en un schéma :



5.2 Redirection de tubes aka *dup2( )*

Vous vous souvenez de *dup2( )* ? Vous savez, le cours ou vous avez ~~rien~~ tout compris ? Eh ben on va en reparler à travers un petit exercice : [2]
Bouyou-Geauchamps : *Faites un programme qui execute l'équivalent de la commande "*ls | wc*", sinon je* ***COLLOOOOOOOOOOOO-***

Comme vous êtes des gens vraiment *too stongue*, vous avez certainement les étapes en tête :

2. inspiré de http ://www.zeitoun.net/articles/communication-par-tuyau/start

— Le processus père créé un processus fils, qui exécute *ls*
— La sortie standard du processus fils est **redirigée** vers l'entrée standard du père
— Le père execute ensuite wc
— *C'est gagné*

L'étape 1 est relativement facile :

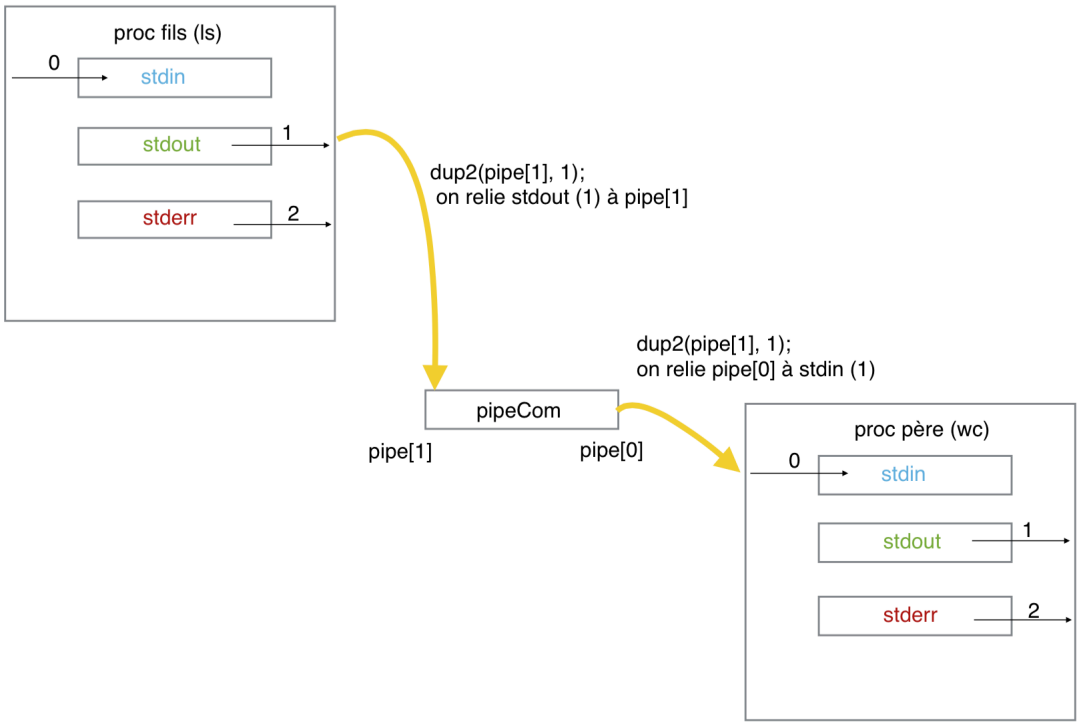
```
int main( int argc, char ** argv )
 {
   pid_t pidFils = fork();

   if (pidFils == 0)
     {
       execlp("ls", "ls", NULL);
       return 1;
     }

   wait(NULL);
   execlp("wc", "wc", NULL);
   return 2;
 }
```

La difficulté se passe évidemment sur la redirection. Il faut nécessairement créer un pipe qui fera la communication entre le père et le fils - on appellera ce pipe *pipeCom* - mais pour la suite ?
Comme vous l'avez deviné, il faut utiliser *dup2( )*.
dup2( ) se présente de la manière suivante :

```
int dup2(int fdSource, int fdCible);
```

En fait, *dup2( )* permet de copier le file descriptor en deuxième argument dans le premier.

Dans notre exercice, cela se traduit donc par une **redirection** de la sortie standard du fils sur l'entrée du pipe *pipeCom* et une redirection de la sortie du pipe *pipeCom* sur l'entrée standard du père.
Regardez le schéma pour un peu plus de clarté :



Et maintenant, voilà notre code :

```
int main( int argc, char ** argv )
 {
   pid_t pidFils = fork();
   int pipeCom[2];
   pipe(pipeCom);

   if (pidFils == 0)
     {
       close(pipeCom[0]);
       dup2(pipeCom[1],1);
       close(pipeCom[1]);
       execlp("ls", "ls", NULL);
       return 1;
     }

   wait(NULL);
   close(pipeCom[1]);
   dup2(pipeCom[0],0);
   close(pipeCom[0]);
   execlp("wc", "wc", NULL);
   return 2;
 }
```

On n'oublie pas de fermer les entrées/sorties inutiles pour éviter des embrouilles, et c'est gagné !

Chapitre 6

Threads

"JME KASS, ET C TOU"

McRoss, qui en a visiblement marre des threads

6.1 Préambule : Pointeurs et pointeurs de fonction

6.1.1 Pointeurs (lol S1)

Faisons un petit rappel sur les pointeurs (get_bellik) :

- Un pointeur se déclare sous la forme <type>* <nomDuPointeur> ; il doit obligatoirement pointer sur une variable.
- Un pointeur comporte ladresse de la variable sur laquelle il pointe.
- Afin daccéder au contenu de la variable sur laquelle le pointeur pointe, il faut utiliser *<pointeur>.
- Utiliser <pointeur> permettra daccéder à ladresse de la variable pointée.

En gros, un pointeur est une adresse, plus précisément celle de la variable pointée.

6.1.2 Pointeurs de fonction

Avant d'attaquer les threads, intéressons nous aux pointeurs de fonction. Un pointeur de fonction est donc une variable qui désigne une fonction. Comme toute variable, on peut utiliser un pointeur de fonction en tant que variable dans une fonction, mais aussi et surtout comme un paramètre dans une fonction.
Un pointeur de fonction se déclare de la manière suivante :

```
void *pointeurFonction;
```

Enfin, on peut créer directement des pointeurs de fonction avec un corps, tout comme une fonction "normale", sauf qu'on ajoute un * devant le nom de la fonction.

```
void *nomDeMaFonction([paramètres])
{
/* corps de la fonction */
}
```

6.2 La notion de thread

Vous vous souvenez des processus ? Un programme englobe des processus, un processus englobe des threads. En fait, un thread est, *une unité d'éxecution plus fine qu'un processus* ; c'est pour cela qu'on appelle

les threads des *processus légers*.
Le but des threads est de pouvoir executer plusieurs parties d'un programme en parallèle. La différence majeure avec les processus est que les threads **partagent tout** entre eux. En effet, lorsque on créé un processus, le fait de modifier quelque chose dans le processus fils n'impactait pas le processus père. Avec les threads, ce n'est pas le cas. Par exemple, si un thread modifie une variable globale, cette variable sera modifiée pour **tous les threads**. Il faut donc faire attention ! Enfin, notez que tous les threads dépendent du **thread principal** ; si le thread principal se termine, tous les autres threads se terminent aussi.

Un thread s'identifie avec un pthread_t. De la meme manière que la fonction *getpid()*, il est possible d'obtenir l'identifiant d'un thread avec la fonction *pthread_self( )*.

6.3 Création de thread et attente de terminaison

Pour créer un thread, on utilise la fonction *pthread_create( )*. Cette fonction se présente de la manière suivante :

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

Regardons tranquillement les paramètres et leur rôle :

- *thread* représente un pointeur (ou une adresse) sur un objet de type *pthread_t* qui sera l'identifiant du thread après l'execution de cette fonction ;
- *attr* représente le ou les attributs du thread, on peut voir ça comme les options de la fonction open() ;
- *fonc* représente un **pointeur de pointeur de fonction** - vous suivez ? -. Ça sera la fonction exécutée par le thread. ;
- *arg* représente un argument passé pour la fonction, il faut que ce soit un pointeur sans type (comprendre *void **), utiliser un cast si besoin. C'est donc le paramètre que va utiliser le thread pour la fonction donnée précédemment ;

Après avoir créé un thread, il faut bien le terminer. Il faut donc utiliser :

```
pthread_exit(void *valeurRetour);
```

valeurRetour est donc un pointeur non typé sur une valeur de retour. Concrètement, cela veut dire qu'il faut faire un cast sur l'adresse de la variable que l'on souhaite retourner. Enfin, sachez que *pthread_exit()* ne tue pas réellement le processus. On verra d'autres manières de le faire dans la partie qui va suivre.

Tout cela parait très flou ? Regardons alors un exemple.

```
#include<pthread.h>

/* Voici la fonction qui sera exécutée par le thread.
 *  Elle prend un paramètre int. */
void *fonction_thread(void *nombre)
{
    /* on récupère ici notre paramètre en faisant un cast.
        On cherche à récupérer le contenu d'un pointeur de type int */
    int nombreParam = *((*int) nombre);

   /* on affiche notre paramètre */
    printf("Le paramètre passé est %i \n", nombreParam);

    /* enfin, on termine le thread et on renvoie notre nombre.
        Pour cela, il faut faire un cast sur l'adresse
        de la variable que l'on veut renvoyer */
    pthread_exit( (void*) &nombreRetour);
}
```

```
int main(int argc, char **argv)
{
        pthread_t threadFils;
        int param = 5;

        /* on créé un thread qui execute fonction_thread
            on passe aussi un paramètre "param" */
        pthread_create(&threadFils, NULL, fonction_thread, (void*) &param);

        /* on déclare un void* pour lire le retour */
        void *ret1;

        /* maintenant on attend la fin du thread
           et on lit sa valeur de retour
           On oublie pas de caster ret1 selon
           le type de retour attendu */
        pthread_join(threadFils, (int*) ret1);

        /* #cterminé */
        exit(0);
}
```

6.4 Autres manières d'arrêter un thread

6.4.1 Arrêter un thread sans code de retour

Dans certains cas, on peut libérer les ressources utilisées par un thread sans se soucier de son code de retour. Pour cela, on utilise :

```
pthread_detach(pthread_t thread_a_arreter);
```

Cette fonction permet donc de tuer réellement un thread dés qu'il a fini réellement son travail, pas comme *pthread_exit( )*.

6.4.2 Envoyer un signal pour tuer un thread (cancel)

Tout comme les processus, il est possible de forcer l'arrêt d'un thread, mais aussi de bloquer le signal d'arrêt. Pour arrêter un thread, il faut utiliser :

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

avec *thread* étant l'identifiant du thread que vous avez envie d'arrêter. **Cependant**, tout comme les processus et les signaux, un thread est capable de bloquer cette demande d'arrêt.

6.4.3 Modification du comportement lors d'une demande de cancel

Il existe deux fonctions permettant de modifier ce comportement :

```
int pthread_setcancelstate(int etat, int *etat_ancien);
```

et

```
int pthread_setcaneltype(int mode, int *mode_ancien);
```

Commençons par *setcancelstate*.

pthread_setcancelstate permet de définir si le thread accepte ou pas les *cancels*. On a donc que deux modes possibles :

- *PTHREAD_CANCEL_DISABLE* pour que le thread bloque tous les cancels ;
- *PTHREAD_CANCEL_ENABLE* pour que le thread reçoive tous les cancels. C'est le mode par défaut.

Un petit exemple pour la route :

```
/* je bloque tous les cancels */
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
```

pthread_setcanceltype permet de définir **comment** le thread accepte les *cancels*. On a, là aussi, que deux modes possibles :

- *PTHREAD_CANCEL_ASYNCHRONOUS* pour que le thread traite les cancels à tous les instants ;
- *PTHREAD_CANCEL_DEFERED* pour que le thread ne traite les cancels qu'a un certain moment. C'est le mode par défaut.

Développons un peu cette histoire de *CANCEL_DEFERED* : en fait, cela veut dire que les cancels reçus par un thread ne seront traités qu'a l'exécution (par le thread) de certaines fonctions, qui sont *pthread_join( ), pthread_testcancel( )* ou *pthread_cond_wait*. Cela veut dire que les cancels reçus n'auront pas d'impact sur le thread jusqu'a que ce dernier exécute une des 3 fonctions citées auparavant.

Chapitre 7

Programmation réseau en C

7.1 Petits rappels

Bien sur, vous vous souvenez tous du réseau en S2 avec Momo ; tcp, udp, ip, etc...
Ce qui est important ici, c'est la notion de **socket**. Un(e) socket est un flux de données qui permet de communiquer via le réseau. En fait, c'est un peu comme un pipe, mais destiné pour la communication par réseau.
Un(e) socket utilise un protocole, soit **udp**, soit **tcp**. Pour résumer très simplement, le protocole udp envoie des informations très rapidement sans vérifier si elle a bien été reçue alors que le protocole tcp vérifie si le paquet a bien été reçu.
udp = rapide, sans vérif ; tcp = lent, mais vérifie.