SYSG4 2020-2021

Cette synthèse a été réalisée avec amour par Schellekens Arnaud, Pattyn Guillaume et Devolder Thomas UwU. (C'est pas fini mais c'est déjà pas mal on a réussi avec. Bonne chance à l'oral les amis)

Chapitre 1 : Avant propos & introduction

1.1 But du cours et labos

But du cours : Compréhension du noyau linux, process etc...
Les labos permettent de comprendre des petites choses et des comportements surprenants au cours. Attention les labos sont résolus, il faut **comprendre ce qui se passe.**

1.2 Cotation (Outdated)

Interro au milieu du quadri. (espaces disque + process). 20% de l'UE
Partie commune écrite (⅔ de l'examen) et une partie labo devant 2 profs (⅓ de l'examen). (pour l'année 19-20)

1.3 Rappels

1.3.1 Commande echo et reproduction par mecho

La commande *echo* :

Faire *echo coucou* donne "coucou" sur la sortie standard.
On va essayer de reproduire cette commande avec une qu'on crée nous même. La commande Mecho. Attention echo ne lit pas au clavier car l'on fait ceci :

echo args enter
Et non :
echo enter
args enter

Le prototype de mecho peut ressembler à quelque chose comme ça :
int main (int argc, char * argv[])

où argc vaut 2 et où argv tableau de pointeurs. Le premier élément est echo, le deuxième coucou et le troisième NULL. mecho fait alors une boucle du style :

```
for(int i=1; i> argc ; i++;){
        printf("%s",argv[i]);
}
```

Mais est ce que faire *echo coucou > f* équivaut à *mecho coucou > f* ?

Faire *echo coucou > f* redirige dans un fichier, si il n'existe pas dans le répertoire courant, ce fichier est créé.
Faire *mecho coucou* > f fait un stdout grâce au shell.

Si on veut afficher * , il faut faire *echo '*'*
echo * affiche tous les fichiers du répertoire courant.
mecho * affiche aussi tous les fichiers !

Cela est bizarre, ce n'est absolument pas ce qu'on a codé.

Il va falloir interpréter nous même cette étoile sinon quelque chose le fait pour nous (le bash). Par exemple injecter dans argv ? C'est une possibilité.
Même chose pour d'autres symboles, il faut par exemple redéfinir aussi ">".

Tout cela, ce sont des process et c'est un exemple de ce que l'on va voir.
Mises à jour atomiques = un process doit terminer avant que qq d'autre utiliser les données. Bref c'est comme an ATLG4 avec les threads.
L'utilisation d'un pipe fait que le out d'un programme communique avec le in d'un autre… Il faut synchroniser tout ça.

1.3.2 Commandes *man* et *apropos*

Commande *man* :

- man 1 = Commande User
- man 2 = Appels Système
- man 3 = librairies C
- man 5 = fichier
- man 7 = livre
- man 8 = commande admin

Commande *apropos* : sert à avoir des informations.

1.3.3 Langage C

Quelques notions :

- Bien se rappeler des paramètres du main.. (voir synthèse DEV3).
- Des formats de printf.
- Que errno est une valeur numérique (définie dans errno.h)
- Un appel système qui retourne -1(erreur) met errno à jour pour indiquer au développeur la nature de l'erreur. Par exemple stdio contient la fonction perror qui utilise errno. Bien vérifier qu'il y a eu une erreur avant d'utiliser errno sinon il peut y avoir des non-sens.
- Faire attention aux strings, au zéro terminé et à ses librairies.
- strcat (avoir prévu de la place en + pour concaténer les 2 strings) sinon RIP.
- strcopy sans backslash 0, RIP.
- On préfère donc utiliser les strn, car il protège le développeurs grâce au paramètre supplémentaire précisant le nombre de char à traiter.

Variables en langage C, avec un exemple :

```
static int s2 ;
int a ;
int b = 4 ;
{
int i ;
static int s1 ;
char c * = malloc(1)
}
```

Analysons les variables :

- i est une variable locale, stockée sur la pile dont la durée de vie et la portée se limite aux accolades.
- a est une variable globale non déclarée, stockée dans .bss .
- b est une variable globale déclarée, stockée dans .data .
- s1 et s2 sont des variables statiques stockées dans le .bss .
- le contenu de c est stockée sur le tas et c sur la pile.

Le passage de paramètres en C se fait **toujours** par copie.

Exemple avec une situation :

struct stat a ;
*struct stat * b;*

stat(..,b) provoque une segmentation fault; il écrit rien dans rien.
stat(..., & a) Ok

L'affichage de printf n'est pas fiable. // fonctionne avec un tampon interne. Il est conseillé d'y ajouter un \n est primordial ou flush le tampon avec fflush(stdout).

1.4 Questions d'examens oral

Questions examens :

- **Lien entre default page et Working Set (déroulement, acteurs, enjeux)**
- **Écrire un code shell qui exécute la commande « ps > f; ls »**
- **Développer l'appel Système Down**
- **Produire un code pour ls ail | cat**
- **Produire un code pour exit x**
- **Exec, ses différents comportements lorsqu'on tombe sur une erreur ou un fichier non existant**
- **Semop expliquer différence avec Dijkstra et écrire le code d'un rdv entre 3 processus**
- **Expliquer l'algorithme de dijkstra et les différences entre dijkstra et System 5**
- **Dup2 envoyer la rep**
- **Fournir la structure d'un code qui utilise les sémaphores**
- **NFU et tout ce que ça implique**
- **Écrire un code shell pour exécuter la commande « vi f & & cat f ». Il n'y a pas de tokenisation et le shell s'arrête à l'exit**
- **L'acteur et les méthodes pour les pages victimes NFU, NRU, etc**
- **Section critique et l'alternance, implémentation du code et expliquer que deux processus ne peuvent pas rentrer en même temps sheeeee**
- **Donnez la structure du code d'un programme qui boucle à l'infini sachant que le processus survivra à la commande « kill -s SIGUSR1 pid où pid est le numéro de process du programme**
- **Expliquer la méthode AGING**
- **Algorithme de UP, DOWN et ZERO dans le contexte de Dijkstra**
- **Coder « mkdir « f » && echo zut »**
- **L'algorithme du producteur, les enjeux et fonctionnement.**
- **Écrire un code de boucle infini qui ne meurt pas à "kill -s SIGUSR1 pid"**
- **Quelles sont les causes de la mort d'un processus? Quels sont les appels systèmes provoquant cette mort? Et à la mort d'un process, quelles sont les conséquences sur lui , sur son père et sur ses fils. (aussi parler des états liés à la mort d'un process)**
- **Créer un fichier de 3 blocs avec 2 caractères ~ (lseek était le thème derrière)**
- **Expliquer l'AS exec et les différentes scénarios qui peuvent arriver lors de l'appel à cet AS**
- **Expliquer une commande qui crée des liens hardwares:softwares et comment l'OS organise ces liens avec les inodes dans le S.F.**

Chapitre 2 : Espace Disque

Un système de fichier sont des métadonnées qui définissent le formatage (ce qui nous permet d'insérer et toutes opérations de fichiers).
Il désigne en quelques sortes la façon dont on organise les fichiers au sein d'un disque. Donc le type d'allocation, la taille et nbr des fichiers, etc.
Ex : FAT, EXT, NTFS

Partition : Avant que l'espace disque puisse être utilisé, un disque doit être divisé (partitionné) en partitions. Les partitions peuvent donc être définies comme des régions dans lesquelles les systèmes d'exploitation présents sur la machine peuvent gérer leurs informations de manière séparée et privée.
Il existe deux types de partitionnement.

2.1 Partition DOS

La partition DOS permet de placer plusieurs systèmes de fichiers ou systèmes d'exploitation.

2.1.1 Première partie du DOS : le MBR

La première partie d'un système de partition DOS sur un disque est appelée MBR. Ce MBR est lui même divisé en 3 parties. Il fait 512 bytes.

1. Un programme d'amorce qui fait 446 bytes. (C'est sur quoi on boot)
2. La table des partitions. (4x16 bytes), car il peut y avoir 4 partitions.
3. Un code magique de boot mis par le BIOS, qui fait 2 bytes. Cette marque est 0x55AA. (Chargé en RAM)

2.1.2 Les partitions

Voici comment est structuré une table de partition (2ème partie du MBR), il y en a max 4 par MBR :

Indique si est bootable (1)	Adresse CHS du premier sector (2)	Type de Système de Fichiers (FAT, NTFS..) (3)	Adresse CHS du dernier sector (4)	Adresse LBA du premier sector (5)	Taille en Secteurs de la partition (6)

1 byte (0x80 si oui, sinon .0x00)	3 bytes	1 Byte	3 Bytes	4 bytes	4 bytes

Infos supplémentaires sur cette structure :

- Il y a une première information : une marque de boot. (1)
- Il peut y avoir des bytes appelés CHS, mais à priori c'est obsolète. (2) et (4).
- Dans la majorité des cas, il y a des informations sur le système de fichiers. (3)
- Vers la fin, il y a 4 bytes (5). C'est le LBA du premier secteur absolu de cette partition.
- Il y a 4 derniers bytes (6) : Le nombre de secteurs dans la partition.

Chaque partition possède un BR (Boot Record), plus précisément premier secteur qui peut également contenir un programme d'amorce.

Les 510 premiers bytes de ce BR sont des bytes de programme de chargement éventuel (donc lu par la loader), et 2 bytes magiques de signature (55AA).

Rappel LBA : logical block addressing, qui est le moyen d'adresser les secteurs d'un disque. Le LBA est donc un numéro de secteur. Le LBA est codé sur 4 bytes (32 bits)

2.1.3 La partition étendue

2.1.3.1 Description du problème du MBR.

Le MBR porte un deuxième problème (qu'on a réussi à bypass), c'est la limitation du nombre de partitions. Limité au nombre de 4, ce n'était pas assez pour certaines personnes ! On a donc créé la partition étendue qui contourne ce problème.

2.1.3.2 Caractéristiques.

Une partition étendue est une partition primaire spéciale. C'est généralement la dernière des quatre partitions. De type 0x05 ou 0x0F, elle est découpée et contient alors des partitions logiques. Chacune de ses partitions commence par un secteur EBR qui chaîne toutes les partitions logiques.

2.1.3.3 Explication du chaînage par la table EBR.

La table EBR (extended boot record) d'une partition donne une petite table (à 4 entrées mais on en utilise que 2). La première entrée donne la table actuelle et la deuxième entrée décrit la suivante (celle juste après, ce que fait une liste chaînée de partitions). Cela résulte donc en une liste chaînée.

La 2ème entrée (de 16 Bytes) est décrite comme suit :

8 Bytes	4 Bytes (1er secteur de la partition suivante)	4 Bytes
00000000	XXXX si il y a une partition après, sinon 0000	0000

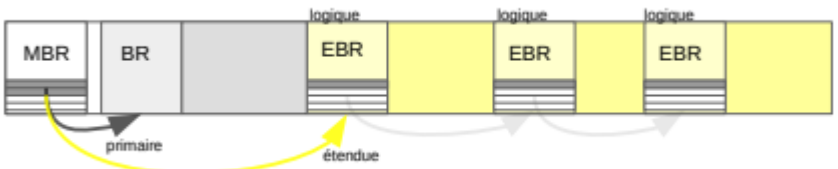
2.1.3.4 Représentation de la partition étendue.

On sait que si le maximum est 4 partitions, leur numérotation ne peut pas excéder 4,5. Pourtant, on peut voir des partitions avec le numéro 5. Si ça se produit, c'est qu'une partition primaire est étendue.

2.1.3.5 Étendue d'étendue ?

Une partition étendue n'est prévue que pour accueillir des partitions logiques.
Une partition étendue **ne peut pas** être étendue.

2.1.3.6 Schéma du cours.



2.1.4 Calculs possibles à propos des limitations

Un LBA est codé sur 4 Bytes :

Nombre Max secteurs * Taille secteur ⇔
Taille max disque 2exp32 * 512 bytes ⇔
2^32 * 2^9 ⇔ 2^41 bytes ⇔ 2 * 2exp40 bytes ⇔
2 TiB et la taille des secteurs est 512 bytes = 2 exp 9 bytes.

A compléter.

2.1.5 Outils

1. On peut exécuter la commande *cat* sur un disque, mais ce n'est pas très performant et optimal. On utilisera plus *open* que *cat*. Mais voici quand même un exemple d'une commande utilisant car et affichant le contenu d'un disque : *sudo cat /dev/da | od -tx1*

2. Autre commande faisant la même chose :
   dd if= /dev/sda (bs=512 count =1 ou bc = 1 count = 512) od -tx1
   Explication de la commande :

   dd : affiche sur la sortie standard le MBR de /dev/sda
   if : input file - le fichier lu
   bs : pour "block size" = 512 Bytes
   count : nombre de blocs de taille 512 Bytes.
   od : ça convertit la sortie de dd (un byte à la fois en hexadécimal).

   NB : Faire *of* à place de *if* détruit le MBR instantanément car commande d'admin !.

3. On peut en lire le contenu brut d'un device via les appels système open, read, close, avec les droits administrateur. (Programme C).
4. La commande *dmesg* affiche les derniers messages du noyau.
5. La commande *fdisk* permet de **partitionner**. Comme il faut sauvegarder pour appliquer, elle est plutôt safe.
   #fdisk -l /*liste les disques disponibles, leur différentes partitions et des infos sur la taille, le nbr de secteurs etc.*/
   #fdisk nomDuDisk /*permet de partitionner le disque en question. celui-ci sera trouvable normalement dans /dev ; ce dossier regroupe les fichiers de périphériques/pilotes de périphériques (device files), car pour rappel dans linux tout est fichier.*/
6. La commande *mkfs* permet de **formater** une partition avec un système de fichiers donné. Exemple (Wiki) : *mkfs -t type device*
7. La commande *mount* attache un nouveau système de fichiers à un répertoire (voir point dédié ci-dessous)

2.1.6 La commande Mount

En informatique, un **point de montage** est un répertoire à partir duquel sont accessibles les données se trouvant sous forme d'un système de fichiers sur une partition de disque dur ou un périphérique. Plus simplement, c'est le dossier qui permet d'accéder au contenu d'un disque dur, clé USB, lecteur DVD, ou autre périphérique de stockage. Lorsque les données sont accessibles à partir d'un point de montage, on dit que la partition ou le périphérique sont *montés*. (WIKI).

Plus d'infos sur :

- https://fr.wikipedia.org/wiki/Point_de_montage
- https://en.wikipedia.org/wiki/Mount_(Unix)
- https://en.wikipedia.org/wiki/Mount_(computing)
- https://youtu.be/zxxt1UxNMBo

C'est ce que la commande mount permet de faire.
Syntaxe de mount : mount (-type) périphérique pointDeMontage

Si on souhaite monter notre partition sur /mnt/data ou tout autre point de montage que vous aurez choisi (pour autant qu'il existe) :

mount /dev/sda3/ /mnt/data

C'est une commande administrateur. (Utilisation de sudo nécessaire).

Si on ne fait pas *mount*, la partition existe mais on pourra y accéder qu'avec des commandes de bas niveaux (du genre *dd*).
Cette commande Mount possède son opposé : *unmount* qui permet de démonter une partition.

2.2 Partition GPT

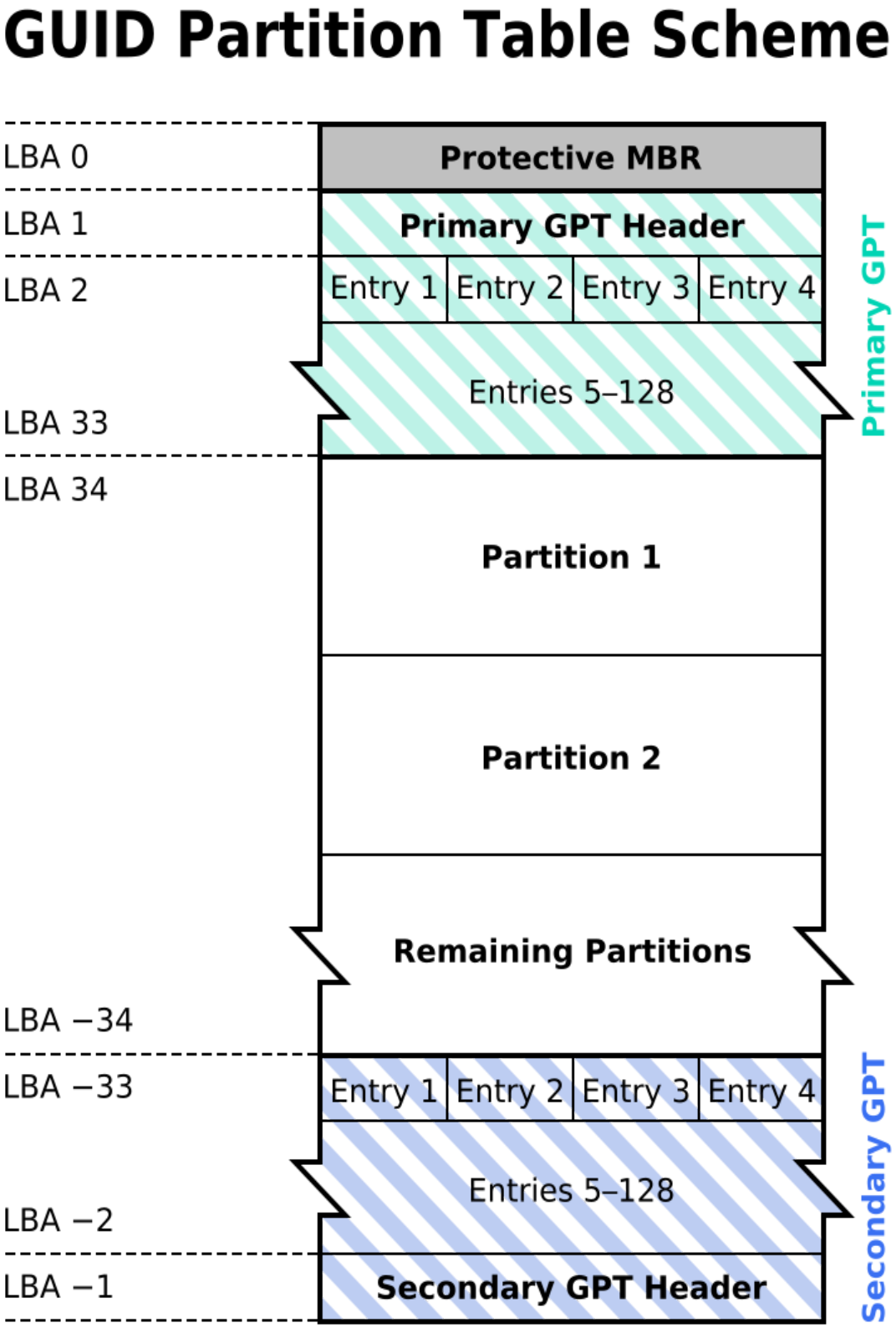
2.2.1 Pourquoi GPT ?

1. On passe de 4 à 128 partitions maximum.
2. GPT gère les disques durs et partitions jusqu'à 9,4 Zo

Pq ? car les LBA sont mtn sur 8 bytes (64 bits) et on peut aller jusqsu'à 128 partitions.
→ Nombre Max secteurs * Taille secteur
⇔ 2^64 * 512 Bytes
⇔ 2^73 Bytes
⇔ 9,4 Milliards de TB de taille max de disque que peut couvrir un partitionnement GPT.

2.2.2 Composition de GPT

2.2.2.1 Schéma de Wikipédia



2.2.2.2 Explications

“En mode GPT, les informations concernant la table de partitionnement sont stockées dans un entête GPT, mais pour garantir une compatibilité (avec les logiciels gérant MBR mais non GPT), GPT maintient une entrée MBR (dite *protectrice* car englobant la totalité du disque) suivie de l'entête d'une partition primaire, le véritable début de la table de partitionnement.”[1]

MBR protector

Au **LBA 0** se trouve un MBR appelé **MBR protecteur**. Il est donc présent pour assurer entre autres la compatibilité avec les ordinateurs utilisant le BIOS, alors que GPT fait partie de l’UEFI (nouveau BIOS si vous voulez).
Ce MBR protecteur décrit une seule partition qui recouvre tout le disque GPT. Le MBR protecteur contient aussi le programme de démarrage (bootloader). C’est donc ce qui protège les disques GPT des écritures provenant d'*utilitaires disques* qui ne (re)connaissent pas les informations de GPT.
(Voir le lien en bas de page pour + de précision)
Taille : 1 secteur (512 Bytes)

Primary GPT

Au **LBA 1** se trouve le secteur des en-têtes appelé partition **table header**. Il définit les blocs utilisables sur le disque ainsi que les nombres et la taille des descripteurs de la table de partitionnement.Donc il sait où se trouve le header secondaire (et lui inversement) et connait la taille et le nombre d’entrées dans la table des descripteurs de fichiers.
Taille : 1 secteur (512 Bytes)

Aux LBA 2 à 33 compris se trouvent les 128 descripteurs de partitions de 128 bytes chacun.
Taille : Cette table des descripteurs utilise 32 secteurs du disque
→ (128 descripteurs* 128 Bytes chacun)/512 taille secteur= 32 secteurs
+ 1 secteur (MBR protector) + 1 secteur (Table header) = 34
GPT utilise donc 34 secteurs du disque, les partitions commencent donc au secteur 34 (le 35ème TMTC).

Chacun des **descripteurs** utilise ses bytes de cette façon :

Offset	**Longueur (en bytes)**	**Contenu**
0	16	Identifiant unique global (type du F.S) **(à savoir)**

[1] https://fr.wikipedia.org/wiki/GUID_Partition_Table

16	16	Des choses sur le GUID.
32	8	Adresse du premier secteur **(à savoir)**
40	8	Adresse du dernier secteur
48	8	Flags d'attributs
56	72	Le nom **(à savoir)**

Secondary GPT

Pour assurer plus de résistance aux pannes, l'en-tête et les 128 descripteurs de partition sont dupliqués en fin de partition.

2.3 EXT

2.3.1 Présentation

Extended File System ou **EXT** est un système de fichier conçu pour Unix.
Crée en 1992, il a été remplacé par **EXT2**.

2.3.2 Composition de EXT

EXT est composé de 4 zones :

1. La zone de boot.
2. Le SuperBloc.
3. Le Tableau d'inodes.
4. Le Tableau de blocs.

2.3.2.1 Zone de boot

C'est la zone d'amorçage, tout ce qu'il y a de plus basique. Elle est située tout devant.

2.3.2.2 SuperBloc

Le SB est le premier bloc de données, il contient les information sur

1. le début et la fin des Tableaux de blocs et d'inodes.
2. Le nombre de blocs et le nombre d'inodes
3. La taille d'un bloc.
4. Si système a été démonté correctement.
5. Les emplacements des blocs libres sous forme de liste.

Le SB est marqué par un nombre magique qui sert de marque de reconnaissance. Ce numéro est à une position fixe par rapport au début du SB.
En résumé, il contient les **métadonnées du système**.

2.3.2.3 Inode & Tableau d'inodes

2.3.2.3.1 Définition d'inode

Un inode est un ensemble structuré. L'inode contient uniquement les méta-données du fichier (c'est-à-dire les informations concernant son fichier sauf les données pures). Un inode possède un numéro qui est en réalité son indice dans le tableau d'inodes.

2.3.2.3.2 Contenu d'un inode

- Taille (en bytes).
- Taille (en nombre de blocs réellement occupés).
- Type - l (pour les liens softwares) d (répertoire).
- Un compteur (Lien Hardware) à 0 veut dire que l'inode n'est pas utilisé.
- Les permissions (représentés sous un masque de 24 bits).
- Dates.
- Numéro du propriétaire.
- Liste de 10 + 3 blocs. (Les 10 premiers sont "fixes", voir liste des blocs).

2.3.2.3.3 Inodes spéciaux

Dans l'inode 1, il y a les blocs défectueux.
Dans l'inode 2, **c'est la racine**.

2.3.2.3.4 Liens entre inodes et fichiers

Chaque fichier a un seul inode.
NB : Un fichier peut avoir plusieurs noms (chacun de ceux-ci fait alors référence au même inode).Un dossier en EXT est considéré comme un fichier, il se comporte donc de la même façon.

2.3.2.4 Tableau de blocs

Ce sont les blocs qui contiennent les données des fichiers.

2.3.2.4.1 Niveaux d'(in)directions.

Ces blocs ont différents niveaux d'indirection.

- Direct : blocs contenant un pointeur directement vers les données.
- Indirect de niv 1 : Bloc qui contient un pointeur vers 256 blocs Direct.

- Indirect de niv 2 : Bloc qui contient un pointeur vers 256 blocs Indirect de niveau 1.
- Indirect de niv 3 : Bloc qui contient un pointeur vers 256 blocs Indirect de niveau 2.

2.3.2.4.2 Application

Les 10 premiers blocs sont directs.
Le 11ème bloc est Indirect de niveau 1.
Le 12ème bloc est Indirect de niveau 2.
Le 13ème bloc est Indirect de niveau 3.
La commande *debugfs* permet de découvrir en détail cette représentation.

2.3.2.4.3 Calculs

Un bloc en linux fait 8 secteurs (de 512 bytes chacun) donc 4096 bytes.
Sachant qu'un pointeur c'est 4 bytes, il est facile de calculer le nombre de blocs maximum pour un fichier.

Pour nos calculs : on considère qu'un bloc fait 8 secteurs (8*512 bytes donc).
Un pointeur fait 4 bytes.

N° du Bloc	Nombre de blocs concernés	Total des blocs concernés
1 à 10	10	10
11	1024	1034
12	1024^2	$1034 + 1024^2$
13	1024^3	$1034 + 1024^2 + 1024^3$

Cela en conclut une formule pour la taille maximale d'un fichier en bytes. :

$\mathbf{((b/4)^3 + (b/4)^2 + b/4 + 10) \times b}$ où b est la taille de bloc en bytes.

2.3.2.4.4 Outils

La commande *ls -l* permet de savoir la taille que le fichier occupe, basé sur la différence entre la fin et le début du fichier sans se soucier des blocs réellement alloués. (On parle de taille logique)
La commande *du -h* permet de savoir le nombre de blocs occupés par un fichier.
(on parle de taille réelle).
La commande *stat* permet d'avoir des infos sur le fichier en plus (il lit l'inode).

Attention l'info *blocks* sur linux affiche les secteurs.

2.3.2.5 Fichiers Creux

Un fichier creux est un fichier qui ne contient rien. Un trou dans un fichier rend ce fichier à trou. La commande lseek permet d'en créer.

2.3.2.5.1 Caractéristiques

- Les positions non écrites dans un fichier sont lues comme des 0 binaires.
- Un fichier à trou (donc qui contient un bloc ou rien n'est écrit) aura un nombre de blocs occupés virtuellement plus grand que physiquement. Il n'y a pas de blocs associés aux trous.
- Le numéro/pointeur d'un tel bloc est mis à 0 dans l'inode.
- L'AS read, renvoie des 0 pour un tel bloc.

2.3.2.5.2 Avantage par rapport à FAT & utilité

L'avantage par rapport à FAT : pas besoin de réserver les blocs.
L'une de ses utilités réside dans les tables de Hachage.
Les tables de hachage permettent un accès direct à l'information, là où en FAT, il faut suivre tout le chaînage si on veut accéder à une certaine donnée dans le fichier (ex des fichiers creux).
Mais le hachage nous ferait perdre de la place, sauf qu'en EXT, on n'en perd pas car les blocs non utilisés ne sont justement pas alloués.

2.3.3 Liens Hardwares

Un lien hardware est un nouveau nom pour le même fichier.

2.3.3.1 Caractéristiques

Pour un fichier A à l'Inode X portant le nom N, le lien hardware porte un autre nom L pour le même inode X. Le compteur de liens Hardware, qui définit le nombre de liens pointant vers l'inode X, est contenu dans l'inode X et est au minimum à 1 car sinon le fichier ne serait plus accessible par quiconque et donc libéré par l'OS.

2.3.3.2 Informations générales

1. On **ne** peut **pas** faire un lien hardware **au-delà** de la partition où on se trouve.
2. Le droit de lecture donne le droit de faire des liens Hardware.
3. Un lien hardware ne change pas le propriétaire.
4. Le fichier avec un compteur à 0 n'est supprimé que s'il n'est pas utilisé dans un process.

2.3.3.2 Problèmes d'accès possible

Supposons le cas suivant, mon fichier A de 1Gb à l'Inode X à son compteur à 2.
L'un des deux liens a été crée par moi-même, dans ma partition.
L'autre crée par mon ami, dans sa partie, je n'y ai pas accès.
Si je supprime mon lien, car je veux faire de la place, le compteur descendra à 1.
Je n'ai aucun moyen de supprimer le lien fait par mon ami, et je ne suis donc plus maître de la vie du fichier, bien que j'en suis encore propriétaire !

2.3.4 Blocs Libres

Les blocs libres sont chaînés dans le SB. Un bloc libre contient une liste de pointeurs de blocs libres. Ce dernier bloc contient le lien vers la suite ou null s'il n'y a plus rien. Si on a besoin d'un bloc, on prend le premier. Si ce même bloc ne contient plus que l'adresse A du bloc suivant, on supprime l'adresse du bloc, on prend ce dernier et le bloc libre originel du SB pointe vers l'adresse A que l'on a supprimé du bloc pris.

2.3.5 Faiblesses d'EXT

1. Nom des fichiers limité à 14 caractères (Question de confort).
2. Les déplacements de la tête et non-répartition en groupe qui correspond physiquement à un cylindre (Question de performance).
3. Les blocs libres ne sont pas représentés sous un tableau de bytes mais sous une liste : il y a de la fragmentation (Question de performance).
4. Le SB unique avec des informations vitales pour le F.S (Question de stabilité).

Il y a quelques notions, par exemple de sécurité et de copie de SB sur les slides.

2.4 EXT2

2.4.1 Améliorations

- On passe de 13 à 15 adresses de bloc. Il y a moins de redirections mais moins d'espace de stockage.
- Les blocs sont agrandis à 4 KiB.
- Il y a désormais des copies du SB.
- Noms plus long et liens software.
- Nouvelles permissions ACL.

Entrée de répertoire en EXT2 (4 exemples slide 52) :

N° d'inode (codé sur 4 bytes).
Lgr de l'entrée (2 bytes)
Lgr du nom du fichier (2 bytes)
Nom du fichier (x bytes car taille variable)

Pour le numéro 3 :

Numéro d'inode 124
0X14 pour 20 caractères occupés en tout
0x9 pour les caractères du fichier.

Les étoiles sont là pour arriver à un multiple de 4.

SI on l'efface : on supprime en agrandissant la taille de l'entrée précédente. Elle sera récupérée si une nouvelle entrée peut y aller. Exemple slide 54. Si on supprime, on supprime tout. Opendir renvoie un pointeur vers une structure opaque.

slide 55 : Ca lit tant qu'il y a des fichiers (si il n'y en a plus readdir(dp) renvoie null). Dans cet exemple de code, printf possède une spécification de taille en plus. A partir de ce code, on peut imaginer le parcours récursif et donc descendre d'un niveau. SI le programme s'arrête et qu'on ne fait pas closedir. Il sera fermé pour nous.

Liens software :

ln -s source destinations (c'est la même que lien hardware avec *-s* en plus.) Créer un fichier liens qui contient le chemin que l'on a indiqué (il s'agit donc d'un autre fichier, qui ne possède donc plus le même n° d'inode !). Si on change la cible de place ; ça ne fonctionne plus ! Contrairement au lien hardware qui lie au même inode, un lien software ne pointe pas vers le fichier cible (mais vers un fichier lien, qui possède le lien vers le fichier). Si on tar ; le lien software reste opérationnel. Si le fichier originel est supprimé, tous les liens software sont d'office useless (ils ne contiennent plus rien). On peut cependant re-créer le fichier en modifiant le fichier lien qui contenait le chemin).

Liens software : peut aller au-delà d'une autre partition / résous le problème des gros fichiers.

Readdir ne garantit pas la permanence de ce qu'elle renvoie.

2.5 Appels Système

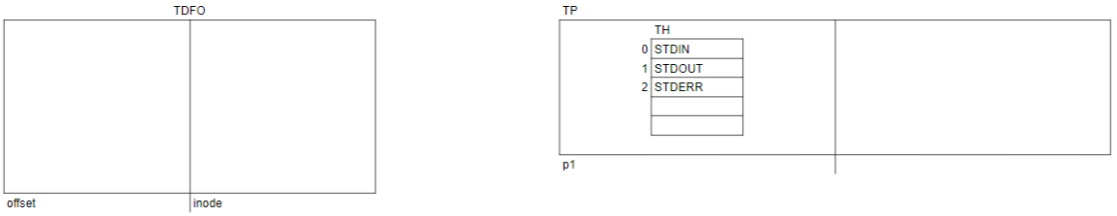
Cette partie (2.5.X) n'est pas complète.

2.5.1 OPEN

2.5.1.1 Que fait open en arrière plan ?

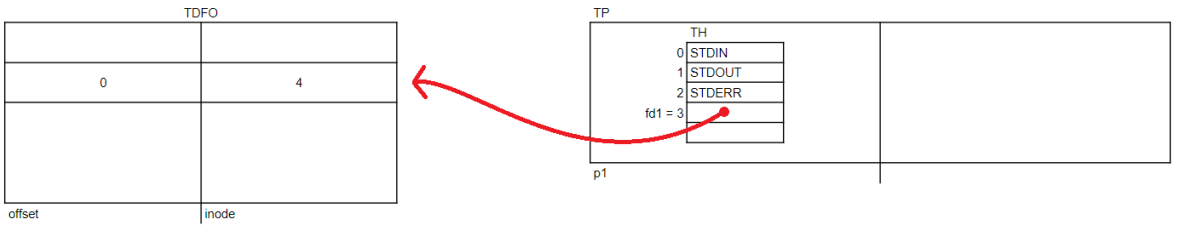
Soit un programme p avec ces 2 instructions :

1. int fd1 = open ("f", …);
2. read(fd1, 8);



Lors de l'exécution de la première instruction, l'appel système open va créer une nouvelle entrée dans la TDFO (Table des descripteurs de fichiers ouverts) GLOBALE, contenant l'inode et le offset du fichier ouvert.
Il ajoute aussi à la table des handles de notre programme l'adresse (handle) de cette nouvelle entrée de la TDFO.

Il retourne l'indice le plus petit (minimum 3 car les 3 premiers handles sont réservés) du nouvel handle dans la TH. Cet indice sera utile par après pour d'autres appels système tel que read.



Ensuite, l'appel système read lit le fichier du nombre de caractères indiqués et avance la valeur offset dans la tdfo.

Enfin, l'appel système close pour fermer le fichier qui demande aussi le file descriptor va supprimer cette adresse mais ne pas forcément supprimer l'entrée dans la TDFO car d'autres programmes pourraient se partager cette entrée. Il faut que le compteur de liens soit à zéro.

→ La TDFO est globale, et la table des handles est propre à chaque proces.

2.5.1.2 Arguments d'Open

L'appel système open à 3 arguments mais le dernier ne sert que lors de la création d'un fichier.

Paramètres :

1. le chemin du fichier.
2. Un entier (flag) sous un masque de bit. Il contient des infos sur le mode d'ouverture (READ ONLY - WRITE ONLY - READ AND WRITE) Complété par d'autres informations lesquelles sont :

- O_CREAT : crée le fichier si n'existe pas. O_CREAT sert aussi pour les droits du fichier.
- O_EXCL (ne vas pas sans o_creat): Le fichier ne doit pas exister et je le crée. Erreur s'il existe.
- O_TRUNC : vide le fichier et l'ouvre en écriture. Si on oublie O_TRUNC, cela écrase les données du fichier uniquement.
- O_APPEND : veut dire que toutes les écritures se feront à la fin, peu importe où se trouve la tete apres un positionnement par lseek.

Pour O_APPEND : Si 2 process ouvrent un fichier sans cette commande et y écrivent, chacun écrit à sa propre tête d'écriture, rien n'est perdu. L'inode est le même, l'offset est différent. **O_APPEND NE MODIFIE PAS L'OFFSET ET IL NE POSITIONNE PAS LA TÊTE D'ÉCRITURE A LA FIN** .

Retour : le descripteur de fichier, l'indice de la table des handle (où il pourra récupérer l'adresse liée à la TDFO).

2.5.1.3 Droits avec Open.

2.5.1.3.1 Droits aléatoires.

Ne pas préciser les droits alors que le fichier est créé, appose des droits aléatoires sur le fichier (gros problème). Ces nouveaux droits seront apposés dès la fermeture de ce fichier.

2.5.1.3.2 Protection automatique.

Il y a une protection par umask sur le fichier (même en précisant 0777), umask est une variable d'environnement à 0022. (Il empêche le droit d'écriture).

2.5.2 AS de statistiques

- stat :

Arg :(filename, struct stat * strstat)
la structure stat donne des informations quant au fichier en paramètre.Comme par exemple son numéro d'inode, sa taille occupée, son propriétaire, etc. Stat permet donc d'obtenir les métadonnées d'un fichier à partir de son inode.
Code c :
struct stat inode; //déclaration de la structure
stat(fileName,&inode); //on passe l'adresse de la structure en paramètre pour y stocker la structure résultante.
printf(inode.st_ino) //il ne reste plus qu'à afficher et utiliser les champs :)
Taper : man 2 stat pour plus d'infos.

- fstatp : la seule != c'est qu'au lieu de passer le chemin du fichier, on passe son descripteur de fichier en 1er param, donc int au lieu de char *.

- lstat pour les fichiers liens software :

La seule différence avec stat c'est que si le fichier est un lien symbolique, stat va en quelque sorte déréférencer ce lien et donner les infos du fichier qui est référencé. Mais si on veut les données du fichier lien lui-même, on utilisera lstat[2].

2.5.3 Dup

dup2 reçoit 2 paramètres: il copie la table des handle de f1 dans la table des handle de f2 ; copie la valeur du pointeur à un autre endroit de la table des process. Si on écrit ailleurs et qu'il y a quelque chose, ce quelque chose est fermé. C'est le noyau qui décide ou c'est copié, et le retour de l'AS qui nous donne cet endroit. (l'indice).
f1 et f2 pointeront donc tous les deux vers la même entrée de la TDFO (celle qui était donc pointée par f1), cela va donc permettre les redirections, comme rediriger la sortie standard dans un fichier.
ex : dup2(4,1) //où 1 est l'id de stdout dans la table des handle. Ce code signifie donc que lorsqu'on voudra écrire à l'écran, on écrira plutôt dans le fichier qui est référencé à l'id 4.
> brol

Et dup(f1) reçoit un seul descripteur de fichier, il ne va donc pas modifier le descripteur du second mais en créer un nouveau, donc une nouvelle entrée dans la table des descripteurs, qui aura la même référence vers l'entrée de la TDFO.
DIfférences entre cat f cat >f et cat <f :

pour cat f affiche le fichier f= 3 donc OUT = 1
cat <f i = 0 (f) out = 1
cat > in = 0 et out = 1 (f)

[2] https://stackoverflow.com/questions/32895019/difference-between-lstat-fstat-and-stat-in-c

Si cat f et cat <f donnent le même résultat ILS NE FONT PAS LA MÊME CHOSE.

2.5.4 lseek

Intérêt de p = lseek h 0 seek cur .. Savoir où on se trouve !
Modifie l'offset de lseek

Slide 87 just avant le l
c'est read qui déplace l'offset.

NB : Unlink fait comme rm supprime

C'est write 1 qui affiche à l'écran , read est éventuellement un system.in .
Slide 89 avec ce code ls donne 10036.
On a écrit 26 un énorme trou 10 carac.
Du coup la place réelle prise sur le disque. Il y a un bloc alloué mais seulement virtuellement. Ca prend 2 blocs de 4 kibi donc ca fait 2 blocs.

2.5.6 Exemple Général

Soit le code suivant

fd = open ("f", O_WROONLY | OCREATE |OCREATE, 0666) ; // Bizarre
dup2(fd,1); (et prenons que fd est à trois) // c'est ici que la redirection est faite !
printf("coucou") ;

1 et 3 pointent vers le même f.

le printf écrira dans f (on a fait une redirection de stdout !) cette ligne sera équivalente à echo coucou > f
Si on voulait récupérer stdout avant, on aurait pu le dup !.

Chapitre 3 : Process

3.1 C'est quoi un process en bref ?

C'est une instance d'un programme qui run. Il possède un PID (son identifiant) associé à un groupe de process et à un utilisateur. Un process possède toujours un père. Un process est créer par clonage, sauf le premier créé par le noyau.

L’appel système getpid retourne l'identifiant du process appelant.

NB : pid est toujours positif ; et toujours supérieur à 1 (sauf init).

3.2 Init et systemd

3.2.1 Init

Le **Process INIT** est le premier process lancée par un noyau Unix (PID 1). Il reste actif en permanence jusqu’au shutdown du système d’exploitation.
Il appartient au root, initialise le système et les services, et sera le père adoptif attitré du système. Si un process quelconque perd son père, il l'adopte. On parle d’adoption.

3.2.2 Systemd

Systemd, est un nouveau daemon qui remplace le init, est améliorant les performances en parallèle.

Explication du cours : Systemd remplace init à partir de 2010 pour accélérer le démarrage des services d’un système unix. Son utilisation est généralisée à partir de 2015. Il met en oeuvre plus de parallélisme pour les démarrages de services.

3.2.3 Plus d’infos ?

- man init
- man systemd

3.3 Autres processus créés au lancement

3.3.1 Explications

Les principaux autres processus créés au lancement sont des démons.
Le shell est un bon exemple, il est représente la connexion en mode console.

3.3.2 Exemples

- cron, crond - planification de commandes
- cupsd - impressions
- sshd - communications sécurisées
- smartd - SMART Disk Monitoring Daemon

3.4 Commandes internes, commandes externes et process

3.4.1 Commandes externes - exécution

L'exécution d'une commande externe (ls,ps,..) ou un autre programme dans le shell, nécessite la création d'un nouveau process qui sera fils du shell. Il s'agira d'une copie de ce shell, mais cette copie sera différenciée par le pid du shell originel.

Notes :
Shell $ c'est l'id du shell

le shell exécute des commandes dans des endroits bien localisées sur le disque.

commande externe = exécutable. même si which ne donne rien , ça ne veut pas dire qu'il n'y a rien.
which dépend du path. which va chercher une commande **EXTERNE**.

Une commande externe correspond à un fichier exécutable sur le disque. Donc à partir du moment où on exécute un code différent que celui qui appartient au shell, cela sera considéré comme une commande externe, et étant donc différent que le shell, cette exécution se fera dans un nouveau process, fils du shell.

3.4.2 Commandes internes - exécution

Toute commande interne est réalisée par le code même du shell. Il n'y a pas de nouveau process crée lié à l'entrée de la commande.
ex : cd, car elle change l'environnement même du shell, donc du même process, par conséquent qu'elle est réalisée par le code même du shell.
man bash pour plus d'infos.

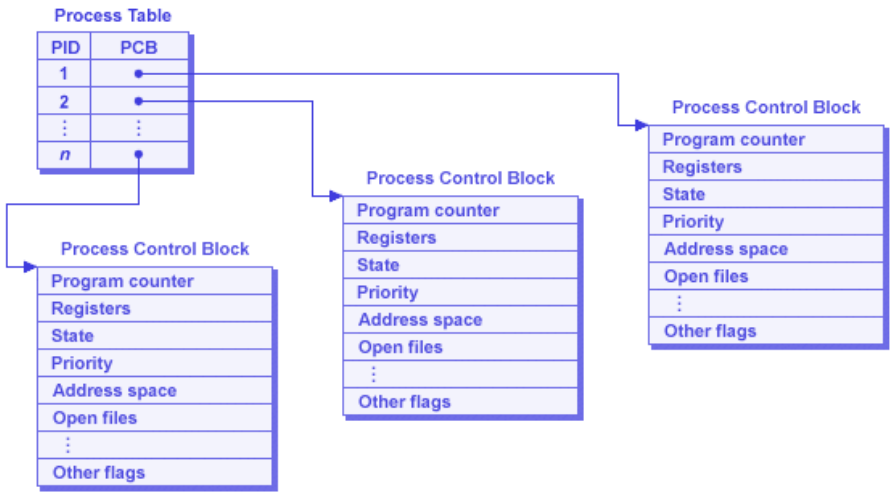
3.5 Le Process en détail

3.5.1 Le process et ses informations

Les Process existants sont tous listés dans une table qu'on appelle table des process.
On y trouve une multitude d'informations à leurs égards.
A chaque process est associé les infos suivantes :

- Son ID
- L'ID de son père
- Sa table des handles et les descripteurs

- Son état.
- Le contexte lui étant lié (Registres)
- Son niveau de privilège
- Pointeur (table des pages)
- Sa priorité
- Des statistique le concernant.
- Table des signaux.
- d'autre infos encore !



© StackOverFlow

Taper "ps" (process status) dans le shell pour obtenir des infos sur les process en cours.

3.5.2 La naissance d'un process.

3.5.2.1 Fork

La naissance d'un process (sauf init) se fait grâce à l'appel système **fork**:
Cet A.S à une particularité : **il a deux retours (le père et le fils)**.

A.S. comme défini dans le MAN : pid_t fork (void);

NB : pid_t : c'est comme size_t, un int mais adapté pour correspondre à l'id d'un process.

Fork clone le process appelant cette fonction. (Tous les concepts sont illustrés par des codes dans les slides).

3.5.2.2 Caractéristiques du process né.

Le process fils va recevoir toutes les infos de son père mais certaines sont variables. Voici un tableau récapitulant les données qui sont soient adaptées, soient conservées avec **fork**

Adaptation	**Conservation**
L'ID	rip (il est modifié via les registres de segments)
L'état	traitement de signaux (il est modifié via les registres de segments)
RAX => si 0 (process fils) sinon pid du fils (process père)	Tables des Handles
CS / DS / SS (Ce sont les registres de segments qui contiennent les sélecteurs de segments et sont utilisé dans le cas de la segmentation pure)	
CR3 (aussi un registre de segments mais intervient dans la pagination en plus de la segmentation)	
Table des signaux	

Plus d'infos sur https://fr.wikipedia.org/wiki/Segmentation_(informatique)

Parmi ces données adaptées, le nouvel id est décidé par le noyau.
Des données sont adaptés, il y a :

- Un dédoublement de l'entrée dans la table des process (une nouvelle ligne).
- Un dédoublement de l'espace d'adressage.
- Les états des 2 process sont indépendants.

NB : Si on est en segmentation pure, CS DS SS sont adaptés. (Mais ça n'arrive quasi jamais). Ce sera surtout CR3 qui sera adapté (segmentation + pagination).

3.5.2.3 L'après Fork.

L'ordonnanceur décide qui commence.
RAX est adapté chez le fils et le père. Si l'ID désigné est 0 c'est le fils sinon c'est pid du fils, donc le père.
RAX=0 pour l'enfant et RAX= PID du fils chez le père ou -1 si erreur

3.5.2.4 Code partagé

Soit le code suivant :
if((pid = fork())==0) printf("je suis le fils"); // Uniquement exécuté par le fils
printf("je suis le père");

L’affichage non déterministe; C’est l’ordonnanceur qui décide.
soit “fils père père”
soit “père fils père”

Si fork() == 0
Le code du if n’est exécuté que par le fils.
Le code d’un éventuel else n’est exécuté que par le père.
Le code en dessous du if/else est exécuté par les deux sauf si le process s’ets arrêté avant.

3.5.2.5 Si Fork n’a pas marché.

En cas d’erreur fork retourne -1.
Cela arrive quand le noyau n’a pas réussi à doubler le process., tous les cas d’erreur sont listés dans le man.

3.5.3 Exemples pratiques

3.5.3.1 Exemple simple de Fork()

```
if((pid = fork())==0){
        printf(“je suis le fils”); // Uniquement exécuté par le fils
        exit(0)
}else{
        printf(“je suis le père”); // Uniquement exécuté par le père
        exit(0)
}
```

3.5.3.2 Exemple plus complexe

```
int main(int argc, char * argv[])
{ int r;
  printf("Je␣suis␣le␣processus␣père␣%d\n",getpid());
  if ( (r=fork()) == 0) // père temporaire
  {
      if ( (r=fork()) == 0){
         usleep (1); // pourquoi ?
         printf("Je␣suis␣le␣ fils ␣non␣attendu␣:(␣%d,␣"
             "mon␣père␣est␣le␣%d\n", getpid(),getppid());
        exit (0);
      }
      else exit (0); // mon fils sera adopté
  }
  wait(0); // wait père temporaire
  while(1);
  exit (0);
}
```

Dans le code ci-dessus, Père fait un premier fils, et fils fait petit-fils. Petit-fils est mis en sleep, ce qui va l'endormir pour 1 seconde. le père lui est bloqué par wait (On verra ça plus tard mais il faut juste comprendre que cette ligne force le père à attendre que son fils meurt), c'est donc fils qui s'exécute. Fils meurt. On verra lorsqu'un process a son process parent mort, il se fera **adopter**.

3.5.3.3 Réponse du slide 26

Enoncé :

```
int main(void) {
int i ;
for ( i=0; i<2; i++) {
        printf (" hello \n");
        fork ();
        printf ("world\n");
}
exit (0);
}
```

```
Père affiche Hello
Fils1 est créé de père (avec  i = 0)
Père affiche World
i de père passe à 1
Père affiche Hello
Fils2 est créé de père (avec = i)
père affiche World et meurt
Fils1 affiche World
i de Fils1 passe à 1
Fils1 affiche Hello
Fils11 est créé de Fils1 (avec i = 1).
Fils1 affiche World et meurt
Fils2 affiche World et meurt
Fils11 afficher World et meurt
```

3.5.4 Concernant la mémoire de variables

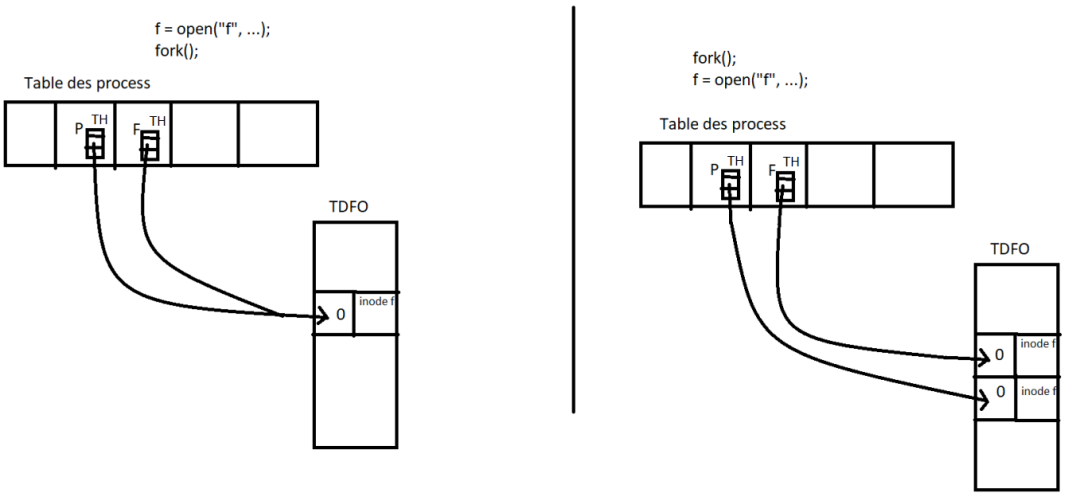
Au moment d'un fork, la mémoire(contexte) est **copiée et ensuite différenciée**. Prenons un exemple, une variable i = 2 est déclarée avant le fork., le fils la recevra tout comme le père, les deux y ont accès. Néanmoins, si le père ou le fils tente de la modifier par après, cela n'altère pas le contenu de la variable de l'autre process.
Le contexte est sauvegardé pour chaque process et rechargé lorsque c'est son tour.

En ce qui concerne les allocations dynamiques, les pointers ont les mêmes adresses logiques mais elles sont complétées par les registres cs ds ss ce qui donne deux adresses pointant vers deux endroits différents dans la mémoire et donc les variables sont indépendantes.
Si on ne fait que lire les variables , la copie de la mémoire ne sera pas faite au moment du fork mais lorsque le besoin de modifier celle-ci se présentera.

Il faut donc retenir une seule chose : une variable sera accessible par le père et le fils et cette variable est indépendante de celle de chez l'autre. Autrement dit Père garde son i, fils reçoit une variable de la même valeur avec le même nom et en fait ce qu'il veut, les process sont indépendants, l'un ne peut pas modifier l'environnement de l'autre. Ils héritent des mêmes adresses des variables mais uniquement de façon logique, leur traduction vers la vraie zone mémoire sera différente à partir du moment où on le modifie(l'espace d'adressage).

3.5.5 Fork et Open

Admettons que le père écrit "abc" dans le fichier f et que le fils écrit "def".
Le contenu à gauche sera : "abcdef" ou "defabc".
Le contenu à droite sera : "abc" ou "def".



Ceci s'explique par le moment ou l'on open. A gauche on open avant de fork, ce qui écrit avec la même "tête de lecture". A droite on open après le fork. Il y a 2" têtes de lecture" qui vont écrire l'une sur l'autre. On entend par tête de lecture une entrée dans la TDFO.

Donc, lorsqu'on fork, on recrée une table des handle pour le fils, car chaque process possède sa propre table des handle. Celle-ci a été créée à partir de la copie de celle du père (il en hérite), les valeurs à l'intérieur référencent donc la même entrée de la TDFO car

celle-ci est globale à tous les process. L'offset d'avancement du fichier va être partagé par père et fils (dans le cas où l'on open avant le fork).

3.5.6 Mort d'un process

3.5.6.1 Comment un process meurt-il ?

Il existe trois/quatre morts pour un process :

- une mort naturelle (exit(0))
- une mort accidentelle (erreur)
- une mort prématurée (exit(n≠0))
- une mort par un signal (kill, ctrl c).

3.5.6.2 EXIT et Conséquences de la mort ?

Lorsqu'un process meurt, il n'est pas directement évacué de la mémoire. Il perd son code, émet un dernier signal et ne fait ensuite plus rien mais ses statistiques sont conservées dans le heap. Cela sert au process père afin de lui permettre de savoir ce qui s'est passé. Un process qui meurt verra ses enfants adopté par init.

EXIT : void exit(int status)
Exit ferme les descripteurs de fichier, et va modifier le pid du père des fils (en les faisant adopter par init, ayant le pid 1). Il va ensuite prévenir le père du processus de cette mort (via un signal SIGCHILD) qui va se servir la valeur status pour savoir cmt c'est passé sa mort.

3.5.6.3 Adoption par init ou systemd

Si le père vient à mourir, les process fils ne sont pas repris par le père du père car le grand-père n'a aucun moyen de savoir où se trouve ses petits-fils. Il ne connaît que ses fils. Les fils sont donc adoptés par init(ou systemd) car lui sait où ils se trouvent.
C'est init qui se chargera de faire les wait si le père ne les fait pas et laisse les zombies trainer ou que ses fils sont morts et se transforment en zombies.
Si un père veut abandonner son fils, il crée un fils 1 qui crée un fils 2 , père fait mourir fils 1 et fils 2 se fait adopter par init.

3.5.6.4 La mort par signal

Les signaux sont des communications entre les process (vu plus tard), il est cependant utile de savoir qu'un signal peut dire à un process de mourir.
Avec la commande kill -signal pid

Le signal par défaut est le n°15, qui demande au process de s'arrêter.
Le n°9 tue lui le process unilatéralement. Un process peut redéfinir les signaux et changer leurs comportements, sauf un signal de stop, ni le signal 9.

3.6 La famille de fonctions Wait

3.6.1 Introduction à Wait

Les statistiques d'un process mort (peu importe la manière) sont conservées dans la table des process où il occupe toujours une place. Il est désormais considéré comme **un zombie** jusqu'à ce que le père effectue un appel à Wait().

Wait est une famille d'appels système permettant à un père de lire les statuts de ses process fils terminés, la façon dont il s'est terminé (il ne tue pas de process en cours). Le système ne libérera pas l'entrée de la table des process assignée au fils tant que cette lecture n'aura pas eu lieu. Wait synchronise en quelque sorte le père sur la mort du fils(effet secondaire).

3.6.2 Appel Bloquant, kézako ?

3.6.2.1 Explication

Dans le cas où les fils d'un process sont encore en vie, les appels système wait() sont bloquants. C'est leur comportement par défaut. Cela bloque le process jusqu'à la fin d'un fils quelconque

3.6.2.2 Exemples

Lorsque le père exécute wait() sur un process non fini, il va être en état bloqué et attendre la mort d'un de ses fils pour reprendre.
Autre exemple, lorsque le shell fait un long find, celui-ci exécute wait() et attend que son fils(find) finisse avant de reprendre la main.
Pour que le shell ne soit pas bloqué il faut rajouter "&" mais cela laissera un zombie.

3.6.3 Les différents Wait

3.6.3.1 Wait

pid_t wait(int ∗status);
wait est bloquant et wait(0) est équivalent à waitpid(-1, 0, 0)

3.6.3.2 Waitpid

pid_t waitpid(pid_t pid, int∗status , int options);

Cette fonction attend que son process fils ait fini et ne fournit pas de statistiques.
où *pid* est le pid du process fils à attendre ou -1 pour un fils quelconque.

3.6.3.3 Wait3

pid_t wait3(int *status, int options, struct rusage *rusage);

Cette fonction attend un fils quelconque.

3.6.3.4 Wait4

pid_t wait4(pid_t pid, int ∗status , int option , struct rusage ∗rusage);

- pid

C'est le process fils concerné ou -1 pour un fils quelconque. (voir les différentes significations des valeurs spécifiques de pid dans le man bien qu'on ne les utilise quasi jamais)

- status (sortie)

		n° exit (8 bits)	n° signal (8 bits)

exit(0) => status = 0
kill -3 process => status = 3
exit(1) => status = 1 * 2^8 = 256
Si status est inférieur à 255 c'est une mort par signal sinon c'est supérieur et une mort par exit.
Un process meurt soit d'un signal, soit d'un exit mais jamais les 2 en même temps ! Ce qui signifie qu'une des 2 parties de status est à 0 partout et l'autre à la valeur donnée en paramètre.
Si l'on donne 0, cela signifie que la raison de la mort ne nous intéresse pas.
Ce n'est ni pratique, ni lisible donc il existe des macros (voir man waitpid).

- option

Ajouter la valeur WNOHANG pour ne pas attendre les autres process fils et éviter le blocage.

- rusage (sortie)

La structure rusage contient les statistiques du process.
Cette structure contient entre autres deux autres structures qui représentent le temps passé sur le cpu et le noyau.

```
struct timeval {
   time_t      tv_sec;  /* secondes */
   suseconds_t tv_usec; /* microsecondes */
};
```

- retour

Cette fonction peut retourner plusieur valeurs :

- le pid du fils (quand le fils meurt) terminé.

- 0 si tous les fils sont encore en vie ou que WNOHANG est mis en option
- -1 s'il y a eu une erreur pour waitpid et 4 et -1 si il n'existe (plus) aucun fils pour wait.

Utilisation :

while(wait(0) > 0); //attendre tous les fils

3.6.4 Pas de wait ?

Le fils sera adopté par init qui deviendra responsable de l'élimination des zombies.
Techniques : double fork :
Consiste à créer d'emblée des orphelins. Le père crée un fils qui aura comme unique but de créer un fils et de mourir faisant donc le petit fils adopté par init pour laisser le père (premier) faire sa vie, car pour rappel le grand père ne sait plus rien de ses petits fils.

3.7 La Famille de fonctions Exec

3.7.1 Principe des fonctions exec

La fonction exec et ses sœurs sont des fonctions très spéciales. Si on considère que fork() à 2 retours, on peut dire que exec n'a pas de retour. En effet, exec remplace le contexte d'exécution d'un process par un nouveau contexte décrit dans un fichier exécutable. Avec un exemple simple on comprend ce que exec fait (voir les labos).
Une fois exec exécuté, on part du code de base, et on va exécuter le code d'un autre fichier. Cela se fait grâce au path.

3.7.2 Et mon code ? Mes données ? Au secours !

Après un appel à exec réussi et qu'aucun nouveau processus n'est créé, le PID ne change pas, mais le code machine, les données, le tas et la pile du processus sont remplacés par ceux du nouveau programme.

Autres effets importants (demander à MBA si ça s'applique bien **à toute la famille** d'exec) :
Effacement des signaux en attente et restauration de leurs comportements par défaut.

Conservation (par défaut) des descripteurs de fichiers.

int execve (const char * fichier, char * const argv [], char * const envp []);
fichier :chemin de l'exécutable sur le disque
argv : tableau des arguments passés au main
envp : tableau des variables d'environnement passé au main
pas de retour en fonctionnement normal.

3.7.3 Et si ça marche pas ?

Si ça ne marche pas, on revient au programme du début, exec retourne -1 et errno est mis à jour. Il y a cependant une nuance.
attention différence entre execv n'a pas réussi à fonctionner ou l'exécutable lancé par execv ne s'est pas lancé, c'est très différent !!

3.7.4 Autres fonctions exec

- Les fonctions qui contiennent la lettre *p* dans leur nom (*execvp* et *execlp*) reçoivent un nom de programme qu'elles recherchent dans le path courant; il est nécessaire de passer le chemin d'accès complet du programme aux fonctions qui ne contiennent pas de *p*.
- Les fonctions contenant la lettre *v* dans leur nom (*execv*, *execvp* et *execve*) reçoivent une liste d'arguments à passer au nouveau programme sous forme d'un tableau de pointeurs vers des chaînes terminé par *NULL*. Les fonctions contenant la lettre *l* (*execl*, *execlp* et *execle*) reçoivent la liste d'arguments via le mécanisme du nombre d'arguments variables du langage C.
- Les fonctions qui contiennent la lettre *e* dans leur nom (*execve* et *execle*) prennent un argument supplémentaire, un tableau de variables d'environnement. L'argument doit être un tableau de pointeurs vers des chaînes terminé par *NULL*. Chaque chaîne doit être de la forme *"VARIABLE=valeur"*[3]

3.7.5 Faille avec **execlp**

si un exécutable à ce bit spécial (SUID), il se peut que le programme se lance avec les droits du propriétaires du fichier. avec execv, on peut changer le path vers la

[3] https://mtodorovic.developpez.com/linux/programmation-avancee/?page=page_3

commande que execv veut initialement faire, et je me retrouve avec les droits de root sur le système du fichier.

Grâce à un programme suid où il y a

execlp(“ls”,”ls”)
exemple de programme
je crée monpetitprogramme
gcc -o ls (je l’appelle ls)

je modifie mon path :
PATH = .:$PATH

system(“ps”) uniquement pour ps.

3.7.6 Utilités de fork et exec : slide 63

Ce shell (interpréteur de commande) :

Le shell read en continu. (dans une boucle)

Quand on entre : il fait une tokenization voir s’il y a une commande externe,
Si c'est le cas, il crée un fils pour cette commande (fork + exec) afin de ne pas succomber à exec. Et on utilise wait pour la supprimer (le wait est donc bloquant) et à la toute fin, réaffiche le prompt.
Le shell fait aussi des remplacements.

3.8 SHELL :

3.8.1 Ecrire un shell :
On verra àa +en détail dans les labos mais en gros :

1. Lire une ligne de commande
2. “Tokeniser” la ligne de commande, cad un tableau qui contient chacune des chaines.
3. Si commande externe : fork+exec et attendre la fin avec wait.
4. Retour au point 1

3.8.1 & bg, fg :

Il est possible d'avoir des processus en tache de fond, on peut placer un process en foreground (fg) ou background (bg). Les commandes "normales" se font en fg et pour en mettre une en bg, il faut que la commande soit suivie de &.
Le shell n'attend alors pas sa fin pour continuer sa propre exécution.

Commandes :
jobs //donne l'état des process en bg en les numérotants.
fg n //n replace en fg le job n°n
bg n //réactive le job n°n qui était suspendu

3.8.2 Redirections :

On a déjà vu le principe des redirections, on applique cela avec exec mtn.
On modifie dans un fils sa table des handles en remplaçant stdin (par ex) à être redirigé dans un fichier pour l'exécution du programme.

3.9 Pipe :

3.9.1 concept du pipe :

un pipe permet la communication unidirectionnelle entre 2 process qui permet d'avoir une mémoire partagée, l'un écrit et l'autre y lit. Ce sont des ressources (arnaud). Le shell crée les descripteurs de fichiers. adéquats. Un pipe nommé est créé via la commande mkfifo. l'indice zéro est là l'endroit depuis lequel on lit (lecture). le 1 c'est l'écriture (à voir dans le sens du programme).

il existe un AS pipe qui sera non nommé (c'est par exemple |). Le pipe ne fonctionne qu'avec des process qui ont un ancêtre commun.
int pipe(int pipefd[2]); // tableau de 2 entrées auquel va être associé l'entré et la sortie du pipe. L'os crée deux descripteurs de fichier p[0] (lecture) et p[1] (écriture) pointant sur un inode de tube. Les données écrites sur l'extrémité écriture d'un tube sont à lire sur l'extrémité lecture du tube.

Le pipe créant 2 descripteurs, il ne faut pas oublier de les close !

L'accès à la zone partagée par deux processus nécessite une synchronisation.

3.9.2 implémentation :

mettre en oeuvre une synchronisation ! géré comme un fifo circulaire l'un écrit l'autre lit écrit lit , .. etc et tout cela synchronisé. Algorithme de producteur consommateur (a voir dans chapitre IPC).

Si pipe plein : bloque l'écrivain
Si pipe vde : bloque le lecteur

possible de partager le pipe à trois. un pipe ne se fait sûrement pas entre un fichier et un process. La fermeture des descripteurs est importante sinon, read en premier, peut bloquer le pipe. Bon exemple au slide 85.

pas de wait entre les fork, ils sont à la fin.
Le fils est le lecteur, il ferme donc son descripteur p[1] qui est pour l'écriture et le père étant l'écrivain, il ferme le descripteur p[0]
Puis quand ils terminent leur job, ils ferment leur descripteur restant.

attention aux wait avec les pipes blocages très probables. Il faut bien mettre les wait après la fermeture des pipes sinon on attend que le fils finisse mais il restera bloqué car toute tentative de lecture d'un pipe vide bloque le process tant qu'il n'y a rien à lire.

slide 90 redirection de la sortie standard vers le p[1]. les closes se font par les exits des exécutables.

slide 95 : 30 descripteurs = 3 pipes ayant 2 descripteurs, 5 process
soit : 6des/process * (4 process (fork) +père)

A retenir :

on fork selon le nombre de commandes : on fait nos choses, on close, si il faut execl et tout à la fin on ferme le père et wait.

< remplace le clavier
> remplace le std::out

3.10 Signaux :

3.10.1 Structure et fonctionnement :

Il existe 2 types de signaux : standards et réels. Ils sont soit envoyés explicitement par des actions ou générés par des erreurs. Tous les signaux sont asynchrones car entre l'appel du signal et son exécution, ça ne se fait pas directement, c'est l'ordonnanceur qui gère.

A chaque process on associe deux tableaux dans la table des process :
31 bits : masque des signaux reçus non encore traités.
31 pointeurs de fonctions (adresse de traitements associés aux signaux).

Envoyer un signal c'est mettre un bit à 1 à l'indice dans la table des signaux; Cette table permet à l'ordonnanceur de voir s'il y a des actions à effectuer, s'il s'est passé qqch pour le process. Donc Si un bit est à 1 pour un certain indice dans la table des masques des signaux, l'ordonnanceur décide quand appeler le traitement/la fonction correspondante à cet indice/à ce signal dans la table des adresses des traitements des signaux. C'est au moment où le processus passe à élu que le(s) signal(aux) sera traité, et ce traitement devra être effectué avant de redonner la main au process. Après avoir effectué le traitement, on remet à 0 le bit correspondant au signal. Étant une table, l'ordonnanceur ne connaît pas l'ordre d'arrivée des signaux, les signaux seront traités dans l'ordre des indices du tableau.

Chaque process possède ces deux tables. Et vu que le fils hérite du père, il hérite également des traitements de signaux associés aux signaux du père (donc de la table des adresses des traitements). Mais il n'est pas affecté par le traitement même des signaux du père. Si un signal est émis pour le père, ce sera uniquement pour le père. Ce qu'il hérite est uniquement la façon dont on va traiter les signaux si on les reçoit. Car oui, pour la plupart des signaux, un process peut modifier leur fonction. Sinon, les signaux ont un traitement par défaut (SIG_DFL).
Les deux signaux dont on ne peut pas modifier leur traitement sont : SIGKILL(9) et SIGSTOP(19).

sigchild est le seul signal ignoré par défaut.
l'AS sigaction permet d'associer un traitement (pas appeler ça c'est l'ordonnanceur).
int sigaction (int signum, const struct sigaction *act, struct sigaction *olact); //signum est le signal à modifier, sigaction est la structure qui va expliquer le traitement.
void sa_h (int s)
void sa_s (int sig, siginfo_t * t, void * old)
voir labo pour + de détails.

3.10.2 Signaux standard :

- 31 signaux standards
- Chaque numéro correspond à un nom de signal
- Sigchild : n°17 : fils arrêté ou terminé
- sigint n°2 : interruption clavier (ctrl-c)
- sigkill n°9 : Signal kill

Le signal 15 est un signal traitable, si reçu c'est une invitation à se fermer proprement. (shutdown par exemple)

le signal sigint est lié à un groupe de process (si je travaille à la console et qu'il y aie plein de fils/programmes attaché à ce premier) ce signal est envoyé en cascade.

Désavantage des signaux standards :

S'il y a plusieurs fois un signal, il ne sera exécuté qu'une fois car ce n'est qu'un bit à 1 qui se fait écrasé à chaque fois tant qu'il n'a pas été fait. On ne connait donc ni le nombre de signaux ni leur ordre d'arrivé.
Solution : liste de signaux au lieu d'une table. C'est ce que les signaux à temps réel feront.

3.10.3 Attendre un signal :

AS pause ; il va mettre en pause un programme jusqu'à réception d'un signal.
en code : while(1) pause(); //rajouter pause évite l'attente active.

quand un programme est en arriere plan & :
la commande fg ramène en avant plan

Quand un programme a été CtrlZ (en arrière plan et stoppé)

bg réactive
fg fait bg et ramène en avant plan.

sigchild est utilisé par les processus parents pour éliminer les zombies de la table.

Core = erreur (divide by zero)
term = ça termine?
Ign = spécial aux enfants ?

sigalarm permet une temporisation il fonctionne avec l'AS alarm on pourra demander à l'os de nous réveiller après un certain nombre de secondes. c'est quais un wait en java.

à ne pas confondre avec pause qui, s'il est activé, attend qu'on le réveille à la réception d'un signal.

si pause est utilisé avec alarm on peut faire un vrai sleep

3.10.4 SIGCHLD et le retour des zombies :

nouvelle façon d'éliminer les zombies. Grâce au signal SIGCHILD. Par défaut, il n'est pas traité, mais on peut modifier son action à effectuer lorsqu'on reçoit ce signal.
Sauf que pour revenir au défaut des signaux standards, si on utilise wait(0), on attend uniquement qu'un fils quelconque meurt, et donc on ne supprimerait pas tous les enfants si l'ordonnanceur a remis le père élu après l'exécution et la mort potentielle de plusieurs fils.
while(wait(0)>0) : Etant donné que wait(0) est bloquant, il va contineur à bloquer le père tant qu'il y a encore des fils en vie.

Solution : waipid, car il n'est pas bloquant.
supprimer des fils 2 façons les 2 façons se valent :
while(waitpid(-1,0,WNOHANG)>0); //-1 pour atteindre n'importe quel process, 0/NULL pour pas de précision, WNOHANG : pour ne pas bloquer et attendre un fils mort.
>0 : tant qu'il y a donc un pid en retour et qu'il y en ait bien 1 qui a été traité.
→ -1 pas de fils à attendre, 0 encore des fils mais aucun terminé sinon le pid du fils.
Cette condition vérifie donc que tant qu'il y a des zombies à traiter, il les traitera, sinon (si plus de fils zombies mais tj en vies ou carrément plus de fils) c'est tout.

ou sigchld qui dit explicitement que le père ignore ces fils et donc il n'est plus embêté par des zombies : sa_handler=SIG_IGN; il dit donc qu'il n'a pas besoin de savoir que ses fils soient mort pour les enterrer(ça vient de moi ça :eyes:).

3.10.5 En résumé :

- Wait(0) => ne libère pas la majorité des fils zombies avant que le père meurt
- while(wait(0)>0) => bloque le process du père jusqu'à ce que tous les fils meurent.

Solutions :

- Double fork pour faire adopter les fils à init
- Traiter le signal SIGCHLD : while(waitpid(-1,0,WNOHANG)>0)
- ignorer **explicitement** le signal SIGCHLD empêche la création des zombies :
  suppr.sa_handler=SIG_IGN;
  sigaction (SIGCHLD, &suppr, NULL);

Chapitre 4 : IPC

inter-process communication.

4.1 Sémaphores :

Les sémaphores ont un compteur :

Principe de la sémaphore : ce sont des outils voués à l'écriture de régions critiques et d'attribuer des ressources non partageables. Des ressources qui, une fois obtenues, doivent être utilisées et menées à terme. La sémaphore est donc un object particulier, une sorte de classe possédant un compteur de ressource. C'est un mécanisme qui est mis en place pour faire des fils de process en attente.
Composition d'une sémaphore :

- compteur de ressources
- file d'attente de process bloqués en attendant qu'une ressource soit disponible
- deux AS UP/DOWN : obtenir/restituer une ressource

Permet de réguler l'accès.
Il y a un troisième zéro, mais on en parle pas pour l'instant ; il sert au RDV de process

Une sémaphore est souvent représenté par un entier r :
si positif = autant de ressources dispo
0 = pas de ressources dispo
négatif = autant de process dans la file (ils réservent leurs places)

DOWN décrémente le compteur (et si la ressource n'est pas disponible, il va également block le process et le mettre en file tout en la réservant en avance.)

Up incrémente le compteur (et si il y avaient des process bloqués il l'autorise le premier qui attend à être executé).
pour r = 1

DOWN → r = 0
SC (et blocage de sautres process s'ils arrivent).
UP r = 1.

Il n'y a pas d'interruptions, les AS UP et DOWN sont rendus atomiques en bloquant les interruptions. Le noyau débloque avec CLI/STI :
CLI (**cl**ear **i**nterrupt) : instruction privilégiée (réservé au code du noyau). Il va rendre le processus interruptible, on ne le dérangera pas. Jusqu'à l'instruction STI(**S**e**t** **I**nterrupts) qui restaure alors. Pour être plus précis, ils vont set 1 ou 0 un flag (**Interrupt flag** (**IF**)).

up & down se finissent en rendant la main à l'ordonnanceur. Il n'y a pas de blocage

Avec cela on peut écrire des régions critiques.

4.2 producteur consommateur :

Un processus produit des données (cases pleines) et l'autre les consomme (cases vides).

Les deux partagent la table, le volume de données est illimité. L'échange se fait via une zone tampon fixe (buffer circulaire) FIFO circulaire.

Les deux process tournent en parallèle. Il faut faire gaffe à ne pas produire trop vite, et donc écraser les données non lues et la lecture de données inexistantes.

Solution : 2 sémaphores pour les 2 ressources. Cases pleines pour le consommateur et cases vides pour le producteur. Il y a deux indices pour la tête et la queue du tableau circulaire.

La queue sert à producteur, il met dans la case[queue] ce qu'il veut et il augmente queue de 1 modulo la table pour pouvoir y mettre sa prochaine info

L'inverse tête sert à consommateur, il consomme et il ajoute 1 pour être à jour la prochaine fois qu'il prendra.

si tete et queue ont la même valeur, ce qui est possible le comportement sera différent selon que le tableau est vide ou plein s'il est vide le consommateur est bloqué il n'y a aucune ressources pour lui en revanche si le tableau est plein c'est le producteur qui est bloqué car il n'y a plus de place pour qu'il mette ses infos.

Ainsi le **producteur** peut par cycle **down** une case **vide**, la remplir et **up** une case **pleine** (pour signifier qu'il y a une case vide en moins et une case pleine qui a été remplie en plus).
le **consommateur** va lui **down** une case **pleine**, la vider, et **up** une case **vide**.

4.3 Section critique & non solutions :

Partage de ressources par plusieurs process et que ces MàJ nécessitent une synchronisation (BDD,zone de mémoire partagée). Risque d'incohérence à cause de la concurrence entre process.

code de P1	code de P2
dernier = dernier -1 ; f[dernier] = f1 ;	dernier = dernier -1 ; f[dernier] = f2

une interruption survient avant la modification de f par P2. . .
perte d'une demande d'impression

Les changements faits par P1 à ce stade seront écrasés par P2.
Il faut éviter les mises à jour concurrentes !

Région critique : partie de code d'un processus en mise à jour d'une ressource partagée.
Il faut faire en sorte de s'assurer qu'entre la connaissance de la donnée en vue de la modifier et la modification, que cette donnée ne soit pas modifiée par qqch d'autre.
But : éviter que plusieurs proces se trouvent en même temps dans leur section critique.

les non solutions ne marchent pas.
Ne reproduisez pas cela chez vous. Ceci est fait par Sebastien

non solution 1 : Interdit toute interruption pdt la section critque. Ne marche que pour des monos processeurs et ne convient pas aux process utilisateur. trop dangereux !

non solution 2 : Variable partagée qui vaut 0 si personne en section critique et 1 sinon. Cela ne résout rien, car si on entre en section critique avant de mettre v à 1 (qui est la première instruction de la région critique) et qu'on redonne la main à un autre process, les deux se trouveront en zone critique. Le verrouillage ne marche pas. c'est le même problème à l'échelle plus petite.

non solution 3 (TSL= Test Set and Lock) : Le process prioritaire(priorité élevée) empêche le process qui bloque (basse priorité) de s'exécuter car il monopolise l'ordonnanceur (vu qu'il

est toujours prêt) alors que le processus de basse priorité a déjà set la variable avant donc blocage. => attente active

non solution 4: V=0 si P1 peut s'exécuter et 1 pour P2. Ordre imposé et attente forcée. ok tier mais vraiment pas fou.

Réglages des problèmes avec les sémaphores (réalisé en 1965):

4.4 Section critique : Sémaphores :

Avec plusieurs producteurs/consommateurs, il va falloir créer une région critique → rajout d'un mutex c'est une sémaphore à seule seule ressource et réaliser l'exclusion mutuelle (initialisé à un). Attention il faut toujours bien faire down (de pleines ou vides) avant de faire down du mutex sinon interblocage. Car si on ne vérifie pas avant de rentrer en section critique que l'on peut consommer/écrire, on est bloqué.
A quoi sert le mutex ? Il va protéger les variables de tête ou de queue peu importe chez le consom/produc mais il protège l'ensemble. surtout si c'est opaque. Il permet qu'il n'y ait qu'un seul consommateur qui rentre à la fois. Il y a un mutex pour les producteurs et un mutex pour les consommateurs.

Maintenant qu'on a vu toute la théorie, il nous faut des outils pour y parvenir (créer des sémaphores etc..) : Librairie System V

4.5 IPC system V :

3 types de ressources :

- Sémaphores
- mémoire partagée
- files de messages

4.5.1 La structure des IPC :

Créés dynamiquement et persistantes jusqu'au prochain reboot ou surpression explicite (par le créateur ou le root). Le partage est possible entre process qui n'ont aucun ancêtre commun (comparé au partage des pipes qui devaient avoir un ancêtre commun !).

Ces structures contiennent des infos : créateur, permissions, constantes symboliques, etc..
commande : *ipcs* //montre les ressources ipc accessibles du système.

Toutes les ressources sont accessibles via un id donné par le noyau, et donc à partir de lui on va pouvoir utiliser les sémaphores dans les différents process.
Pour accéder à une mémoire partagée il faut l'id (clé).
plusieurs façons de l'avoir : soit le père nous la donne avant le fork et on reçoit la variable
soit en paramètre de fonction
si c'est des process tout à fait distants et qu'on sait pas qui est le créateur ? Comment avoir cette clé?

shmget() renvoie l'identifiant du segment de mémoire partagée associé à la valeur de l'argument *clé*
premier paramètre : la clé convenue entre les différents process. (un entier)
int shmget(key_t *clé***, size_t** *size***, int** *shmflg***);**

Si on est le premier à aller dans cette mémoire partagée il faut d'abord la créer avec les flag IPC_CREAT(comme as open)
Il existe aussi le flag IPC_EXCL : erreur si je ne suis pas le créateur, la ressource ne doit pas préexister.
IPC_PRIVATE => on se passera de la clé et le groupe de process se débrouille pour trouver la clé (pipe, mémoire partagée). ça évite d'utiliser une clé qui pourrait être déjà utilisée. IPC PRIVATE est utilisé quand on ne veut pas de clé, c'est une clé privée, tous les programmes qui voudront interagir ne peuvent pas faire semget et donner l'identifiant. Ca fournit donc un nouvel id qui restera privé pour ne pas partager la ressource avec tlm, il faudra alors la partager avec ceux avec lesquels on veut partager la ressource.

Plus d'infos sur l'argument key :

`key`
One of the following:

- A key returned by the `ftok` function if the shared memory can be accessed by more than one process.
- The value IPC_PRIVATE to indicate that the shared memory cannot be accessed by other processes.

A shared memory identifier, associated data structure, and the shared memory are created for the **key** parameter if one of the following is true:

- The **key** parameter has a value of IPC_PRIVATE.
- The **key** parameter does not already have a shared memory identifier associated with it and IPC_CREAT is specified for the **shmflg** parameter.

4.5.2 Ressource 1 - SEM (Sémaphore) :

Les sémaphores IPC permettent de gérer un ensemble de sémaphores et donc de ressources.
Les 3 opérations sur un sémaphore IPC System V :
DOWN : bloque un processus tant que le compteur de ressources est < au nombre de ressources demandé. Le compteur de ressources est décrémenté en conséquence.
UP : restitue des ressources, le compteur de ressources est incrémenté en conséquence.
ZERO //attente de zéro bloque un ou +sieurs processus tant que le compteur de ressources est > à 0. N'altère pas le compteur de ressources.

semget(sem=sémaphore)
une structure anonyme décrit chaque sémaphore de l'ensemble :

semval : valeur du sem pos ou nulle (compteur de ressource)
semzcnt : en attente du zéro
semncnt : en attente de ressource
sempid : dernier processus agissant

semop : semaphore operation, (int semid, struct sembuf, , unsigned spos) :
où sembuf : sem_num //num du sem
sem_op //<0 (down), 0 (zero), >0(up)
sem_flg //flag
int semctl (semid, semno, cmd, …);
sem controll operation.

Donc on définit un sembuf qui correspond à l'action a effectuer quand on appelle semop, donc semop modifie le sémaphore.
Et semget est utilisé pour obtenir les infos par rapport à un certain sémaphore. Par ex avec l'attrribut **`GETVAL`** pour obtenir la valeur du sem.

IL faut s'attacher à la mémoire partagée et aussi se détacher
si le dernier de la mémoire se détache et personne a dit de la détruire. dès que le compteur passe à 0 la mémoire est détruite implicitement

4.5.3 Ressource 2 - SHM (shared memory) :

Une mémoire partagée par plusieurs processus est visible via l'espace d'adressage de chacun d'entre eux. Il faut l'attacher à cet espace un peu comme on monte un système de fichiers.
shmget(shm=share memory)

shmat : s'attacher
shmat(int shmid, const void * shmaddr, int shmflg)
première paramètre :identifiant de la MP
second : pointer de ou la page doit être mise (ne faite pas ça le noyau fera mieux que vous (laissez null))
3eme : droit de la page (lecture/écriture)

shmdt se détacher
seul paramètre : l'adresse d'attachement

Un compteur d'utilisation est incrémenté ou décrémenté par ces AS.
Le noyau décide d'attacher une mémoire partagée à une adresse frontière de page qui a la forme 0xXXXXX000, mutliple de 4096
shmctl contrôle et donne plus d'infos.

ça retourne un pointeur vers la mémoire donc pas besoin d'appel système pour y accéder

4.5. x Libérer un ressource :

Les ressources IPC sont permanentes, cad que si le programme qui les a créer ne les supprime pas, elles vont survivre. Créer dynamiquement par des AS, il faut donc les supprimer explicitement (la suppression n'est pas tj synchronisée avec la demande de suppression, mais plutôt quand plus personne en aura besoin).
pour les sémaphore c'est immédiat.
shmRMID attend que tous les process se détachent avant de la supprimer. Pendant ce temps d'attente personne d'autre ne peut s'attacher.
Quand y a plus de liens et que le flag (IPC_RMID marqué pour destruction) suppr est mis à vrai => suppression de la page
Quand y a plus de liens mais pas rmid, la page reste permanente.
Un détachement implicite a lieu lors d'un exit ou d'un exec. (donc le cpt d'attachement est décrémenté mais si pas de RMID, il va rester).

shmctl(shm,IPC_RMID,0);
Il y a donc 3 opérations pour les SEM

Chapitre 5 : Mémoire

1. La Pagination

La pagination, c'est pour mettre en mémoire juste une partie du code d'un programme.

Pour rappel, l'espace d'adressage est la mémoire dite "virtuelle", c'est celle que le programme voit.
La RAM c'est la mémoire "physique", là où seront vraiment les données.

Adressage physique c'est pas fou pour la multi programmation car plusieurs programmes peuvent accéder à la même case mémoire.

Donc 2 solutions :

- relocalisation statique → pendant le chargement en RAM, on révise toutes instructions avec accès mémoire en modifiant ces adresses. Ajouter un entier

à toutes les adresses pour qu' il y ai pas de collisions. Chargement plus long. (?)

- relocalisation dynamique avec le MMU. À chaque accès mémoire il y a une traduction.

Par exemple, pour :
MOV [RAX], 100
Il y aura 2 accès à la mémoire : RIP (pour l'instruction) et RAX (pour exécuter le contenu de l'instruction)

Le MMU, avant de traduire, vérifie l'adresse, ça cause un ralentissement à l'exécution mais chargement ok et on peut faire pagination et swap.

Différence entre les deux ; l'un se fait au chargement, l'autre à l'exécution.

Mais si la RAM est trop petite pour tous les programmes => swapping de programmes, c'est poop car fragmentation.

Du coup swapping de parties car tout le programme ne nécessite pas d'être en RAM et pas de fragmentation.

La partie virtuelle (Espace d'adressage) va être séparée en plusieurs "pages" de 4 KiB (2^{12} bytes).
La RAM fait pareil mais on appelle ça des "cadres" (de 4 KiB aussi).
Les pages et les cadres doivent avoir la même taille.

Mais comment fait le MMU pour identifier le numéro de page auquel correspond une adresse ?

Comme les pages font 4 KiB c'est simple pour un humain de savoir.

Pour une adresse donnée, il y aura :

- 12 bits de décalage (offset) dans la page/cadre à droite => donc 3 digits hexadécimaux (4 KiB = 2^{12} Bytes = 4096 Bytes)
- le reste des bits sera l'adresse de chaque pages/cadres

C'est donc très facile pour le MMU de traduire des adresses virtuelles en physiques, il n'a qu'à faire [adresse] DIV 4096 pour trouver l'adresse de la page et [adresse] MOD 4096 pour l'offset.

La table des pages (TP) sert à savoir dans quels cadres se trouvent les pages. Elle se trouve dans la RAM. La table contient autant d'entrées que de pages. Chaque programme aura sa table.
Chaque entrée aura un descripteur de pages de 4 Bytes.

Pour que le MMU puisse traduire une adresse virtuelle en adresse physique, il a besoin d'accéder à la TP dans la RAM. Pour cela le registre CR3 lui donne l'endroit ou se trouve la TP

=> Pour 0x0F456210 :

- l'adresse de la page sera 0x0F456
- et l'offset sera 0x210

Le MMU va chercher l'adresse du cadre correspondant à la page qu'il a trouvé. À l'indice de la page x 4 (chaque entrée fait 4 bytes), il trouvera l'adresse du cadre correspondant.
Tout cela reste simple (pour le MMU) pcq la taille de page est de 4 KiB.

C'est très important que le MMU doit être autonome, pcq il fait beaucoup d'opérations. Avec ce système, il ne fait pas appel au ALU.

En résumé, pour que le MMU traduise l'adresse, il faut :

- La partie gauche (0x0F456)
- La partie droite (0x210)
- Remplacer la partie gauche par le numéro de cadre, qui se trouve dans la TP (0x87, par exemple)
- le décalage sera le même.
- L'adresse physique sera donc, dans notre exemple : (0x00087210)

En gros, pour trouver l'adresse physique, il faut remplacer le numéro de page par le numéro de cadre.

2. PAGE FAULT

Mais que se passe-t-il si la page n'est pas en mémoire. (Si dans la TP, il y a un 0, c'est que la page n'est pas là).
=> Le MMU va demander de l'aide à l'OS (PAGE FAULT)

L'adresse virtuelle qui plante sera mise dans **CR2** avant que le MMU demande à l'OS de l'aide.
L'OS devra trouver dans la RAM, un endroit où aller mettre cette page, pour la charger. Cet endroit aura une adresse (évidement). Et devra changer les infos dans

la TP, remettre l'adresse qu'il a trouvée. RIP reprend la valeur d'avant pour réessayer l'instruction.

Si l'OS ne trouve pas de cadres (pas de place), il devra choisir une page victime (plus tard)

/!\ `sur intel RIP = CR2 pour réexécuter` dans les slides : **C'EST FAUX**

si la RAM est pleine, l'OS va dégager un cadre associé. Il fait un choix parmi plusieurs.

P = page présente
R = bit de référencement mis à 1 à chaque fois qu'on y accède (par le MMU) et à 0 à chaque interaction horloge par l'OS
M = (modification) MMu met à 1 qu'on accède la mémoire et qu'on change.

Quand une association est faite l'OS M est mis à 0 et R à 1 .
avec NRU, il préfère d'abord ceux non récemment référencée et ensuite si modifié ou non.

Not Recently Used

classe	R	M	
0	0	0	non récemment référencée, non modifiée
1	0	1	non récemment référencée, modifiée
2	1	0	récemment référencée, non modifiée
3	1	1	récemment référencée, modifiée

Il préfère dégager 0 puis 1 puis 2 puis 3.

Sinon FIFO mais ça pue

FIFO SCO mais ça pue quand même.

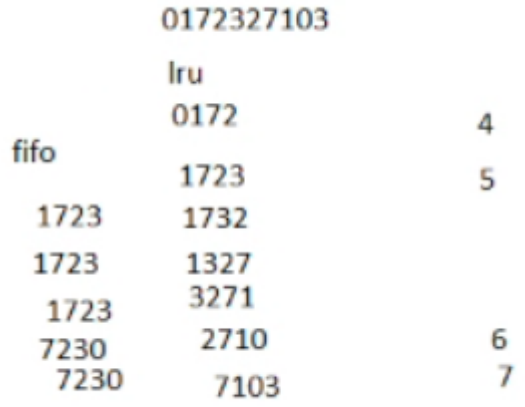
LRU mais chiant car faire une liste de stats
ou compteur mais pas pratique.

NFU - celui le moins interrompu qui clamse.

NFU amélioré AGING histoire de bit à gauche à chaque fois.

AGING c’est bien.

SLIDE 65



à refaire mais principe ok.

Si cadre et p à 0 MMU crie help.

OS prend la main si il trouve un cadre libre en RAM, il met dans table des pages le numéro du cadre, le bit p à 1 et r = 1 m à 0. décrémente le rip car il doit refaire l’instruction après que le cadre aie été associé.

SI RAM Pleine défaut de page. si Pas de cadre libre en ram.
Il va en choisir de cadre et le delete.
SI m à 1 il sauvegarde. Après ça il supprime définitivement dans la ram

Il check la table de spages à l’adresse où se trouve la cadre supprimé il met p à 0 update la nouvelle adresse avec les valeurs (pm,r) et le nouveau cadre .
rip--
Ordonnanceur.

1 instruction = 1 ns. si MMU plus lent que 1 ns c’est poop donc ça doit être rapide.

Pas ouf 1 seule table d epage

on va en faire 2 couche.

avec une première table des pages, chaque entrée pointe vers une sous table.

les 10 premiers bits servent à savoir quel sous table aller, et les 10 suivants savoir quelle page aller dans la sous table

Segmentation.

mode réel qu'au début. 2 registres combinées de 1 6 bits. bref sert qu'au début.

Mode protégé adresse logique à adresse linéaire.
si pagination on passe à adresse physique

Fonctionne avec les sélecteurs et registres cs ds ss.

GDT et LDT. notion de droits et de granularité, de privilèges

Glossaire :

Adresse = Handle
ACL = Access Control List
TDFO : Table (descripteur) des fichiers ouverts.
LBA = Logical Block Adress (C'est un numéro de secteur).
SB = SuperBloc
TI = Tableau d'inodes.
BA = Boot Area
BR = Root Record
IPC = Inter Process Communication.
MMU = Memory Management Unit.
MBR = Master Boot Record
shm = Share memory
Pile = Stack
File (ou tas) = Heap

Labos SYSL 2019-2020

Labo Intro

Non compris dans l'examen. Il s'agit principalement de makefile.

Labo ED (Espace Disque)

ED - LaboED 01-01 - Partitionnement DOS - fdisk

A retenir :

Traitement des tables primaires d'un système de fichiers DOS.

Réponses :

Adaptez le programme précédent pour afficher le type des partitions.
C'est le cinquième byte qu'il faut afficher (de fill[8]).
Que faudrait-il faire pour afficher les données de la partition logique ?

Logiquement, il faudrait montrer byte par byte, tous les bytes de tous les blocs présent dans la partition logique.
Comme décrit au dessus : *sudo cat /dev/da* | *od -tx1* permet d'afficher le contenu d'un disque. Il faudrait probablement chipoter un peu.

ED - LaboED 01-02 - Partitionnement GPT - fdisk

A retenir :

Affichage d'une table de partition GPT ainsi que son contenu.

Question :

Réaliser la restauration du descripteur de la partition 2 à partir du descripteur secondaire qui se trouve en fin de disque.
En d'autres termes on pourrait essayer de comparer que le contenu du descripteur de la partition X du GPT primaire soit le même que celui du backup du GPT secondaire.

Réponses :

En sachant que la taille de la table des descripteurs de partition utilise 32 secteurs, que le GPT header en utilise 1 et qu'il n'y a pas une seconde fois le protective MBR, le secondary GPT utilise 33 secteurs (de 512 bytes).
Si l'on veut se placer au début du descripteur de la seconde partition, il va falloir se placer dans le fichier avec la méthode
lseek (**int** descripteur de fichier, **off_t** l'offset en bytes, **int** whence (l'offset partira de là)
Et donc si l'on veut se placer dans le descripteur de la partition 2 depuis le GPT secondaire : lseek(fd, -(33*512)+128, SEEK_END)
le - car ils sont indexés depuis le LBA -34 à -1.
33 car le GPT secondaire utilise 33 secteurs et *512 pour la taille en byte de secteurs.
+128 car jusqu'à présent on s'est placé au premier secteur et donc au premier descripteur, mais on veut le deuxième et pour rappel chaque descripteur fait 128 Bytes.

ED - LaboED 01-03 - Structure F.S. - mkfs

A retenir :

- fdisk permet de partitionner.
- mkfs permet de formater.

Création d'un système de fichier.
mkfs.FSName (mkfs. et tabulez pour voir les propositions)
ex : mkfs.vfat /dev/sdb1 (où sdb 1 est une partition primaire)
mkfs.ext2 /dev/sdb5 (où sdb5 est une partition logique)
Faites bien attention que vous ne pouvez pas formater une partition étendue, elle ne fait qu'entendre les capacités de partitions ! Vous pouvez uniquement formater ses partitions logiques.

ED - LaboED 01-04 - fdisk automatisé

A retenir :

Possibilité d'écrire un script automatisant le partitionnement d'un disque.

Réponses :

Automatisation du partitionnement d'un disque.
On crée un fichier nommé script, qu'on exécutera en admin avec vi.
Corps du fichier :

Commande pour lancer le fichier : *sudo ./script*

ED - LaboED 02-01 - Inodes - stat,lstat

A retenir :

Les informations d'un inode (man 2 stat).
lien hardware : ln *cible source*
lien software : ln -s *cible source*

Réponses :

Si vous voulez les infos d'un fichier (inode, nbr liens hard, etc) → stat
Si le fihcier passé en parmaètre est un lien software, stat va déréférencer le fichier lien et donner les infos du fichier pointé.
Q : Comment obtenir les infos du fichier lien lui-même ?
→ utiliser lstat au lieu de stat :
int r = lstat(argv[1], &inode);
"lstat est identique à stat sauf que si le pathname est un lien symbolique, il va retourner les infos du fichier lien lui-même, et non pas du fichier référencé."

ED - LaboED 02-02 - Fichiers creux - lseek, ls, du -h

A retenir :

Création et remplissage d'un fichier creux, et stockage de ce fichier sur ext/fat.

Réponses :

Lien expliquant le phénomène.

De plus, ce qu'on essaie de démontrer c'est la différence entre ext et FAT pour des fichiers creux. Voir le point fichier creux en théorie.

ED - LaboED 02-03 - Structure F.S. - mkfs, debugfs

A retenir :

Description d'un système de fichier à l'aide de debugfs
Debugfs permet de lire et écrire la structure d'un FS de type ext et permet de voir le chaînage des blocs dans le détail.

Réponses :

A compléter.

ED - LaboED 03-01 - Répertoires - opendir, readdir, lstat

A retenir :

Exploration des dossiers/directories.
man opendir et man readdir pour les infos.

Question : Réaliser commande ls *

Réponses :

Readdir nous retourne un pointeur vers la structure **dirent** du prochain fichier à lire.
Cette structure possède l'attribut *d_type* qui donne le type du fichier (lien symbolique, un fichier régulier, un directory, …)

ls * : liste le contenu du dossier courant ainsi que le contenu des directory contenu dans le directory courant (à 1 niveau). Donc But : lister tous les fichiers en bouclant readdir, et si c'est un dossier, on va lister tous ses fichiers aussi.

Donc si l'attribut d_type de dirp (qui est notre structure dirent), est de type DT_DIR (directory), alors on va aussi lister ses fichiers.

NB : (il se peut que vous devriez ajouter -D_DEFAULT_SOURCE, ou qqch du style dans le makefile)

ED - LaboED 03-02 - Droits des fichiers - SUID

A retenir :

Droits sur les fichiers. On joue avec les permissions STICKY BIT, SGID et SUID.

Réponses :

A compléter

ED - LaboED 04-01 - Commande filtre, cat sans argument en c

Que ce soit le prog Mcat ou la commande cat, ils sont semblables car c'est le shell qui interprète les commandes.

ED - LaboED 04-02 - cat avec arguments en c

Question :

réécrire la commande filtre **head.** (man head)

Donc semblable à cat mais qui se limite aux 10 premières lignes du fichier.

Réponse :

Il suffit d'incrémenter un compteur à chaque fois que l'on rencontre le caractère fin de ligne.

ED - LABO AUTRE Créer une partition GPT :

Il faut principalement suivre le pdf. Suite de commande utile.
Lien utile

Labo Process

Process - LaboProcess 01-01 - fork clonage et adoption

Observation :

On observe que la valeur à l'adresse A possède une valeur initiale de 6, qu'après le fork, père et fils ont toujours 6 , mais que lors de la réassignation, père et fils ont 2 valeurs différentes, même si l'adresse est la même.

Manipulation avancée :

Rien ne change de toute façon.
Quand on fait mourir le père et qu'il se fait adopter par init, on voit que le fils possède bien la nouvelle valeur qu'il doit avoir.
Que la variable soit déclarée avant le main(statique) ou dans le main(dynamique)., rien ne change.
Jouer un peu avec getppid et des sleep pour faire survivre le fils à la mort du père et pour voir init adopter le fils sans père.
L'AS exit est obligatoire dans le code du fils sinon, celui-ci exécutera également le code du père.

Process - LaboProcess 02-01 - wait4

Observation :

Quand on fait des calculs, on ne passe pas de temps dans le noyau car on ouvre pas de fichiers, par contre avec "s" et qu'on écrit dans le fichier, la tdfo étant dans le noyau on passe du temps dans ce noyau !

Rappel : wait supprime le zombie de la table des process.

En labo, on utilisera sprintf pour transformer un nombre en chaîne de caractères.
Pour corriger le shell avec les commandes qui ne marchent pas : il faut rajouter exit(0) après perror.

Process - LaboProcess 02-02 - zombies

A retenir :

On a affaire à un simple fils qui meurt, et le père va clean le zombie.

Réponses :

Le principe est simple : père crée fils 1 , fils 1 crée fils 2, on fait attendre fils 2, fils 1 meurt, fils 2 se fait adopter par init, on clean fils 1 avec wait du père., et fils 2 mourra et se fera clean plus tard par init.

Process - LaboProcess 03-01 - exec

A retenir :

Cela présente un cas d'utilisation pratique de exec .
A noter que
wait (&j); // récupère la valeur de exit.

Process - LaboProcess 03-02 - exec et sécurité

A retenir :

Cela montre une faille que possède exec

Process - LaboProcess 04-01 - shell simple

A retenir :

-

Réponses :

Process - LaboProcess 04-02 - shell et background

A retenir :

//NOTE pas super compréhensible. Le but est : si on met une esperluette, on n'attend pas le process en mettant bg à vrai sinon à la fin on wait waitpid qui attend le process.

Réponses :

Pourquoi waitpid et pas wait ?
Le but de waitpid est de retourner les infos du zombie avant de le nettoyer.
Si on fait waitpid et qu'on ne veut pas attendre, alors quand le fils meurt ce ne sera pas nettoyé dans table.

Process - LaboProcess 04-03 - shell et redirections

A retenir :

Ce programme redirige la sortie standard dans le fichier précisé donc
si je fais ls > f, ça envoie le résultat de ls dans f.
> : redirige le résultat dans un fichier, le crée s'il n'existe pas et efface son contenu
>> : ,'écrase pas le contenu mais ajoue à la fin
< : lire le fichier et donner le résulat à une commande

Code du if de ShellBack.c
Ensuite on crée un fichier avec les commandes et on va le donner à notre shell
Code de ce fichier : m q

Process - LaboProcess 05-01 - pipe et shell

A retenir :

A compléter.

Process - LaboProcess 05-02 - pipe et shell - à corriger

Réponses :

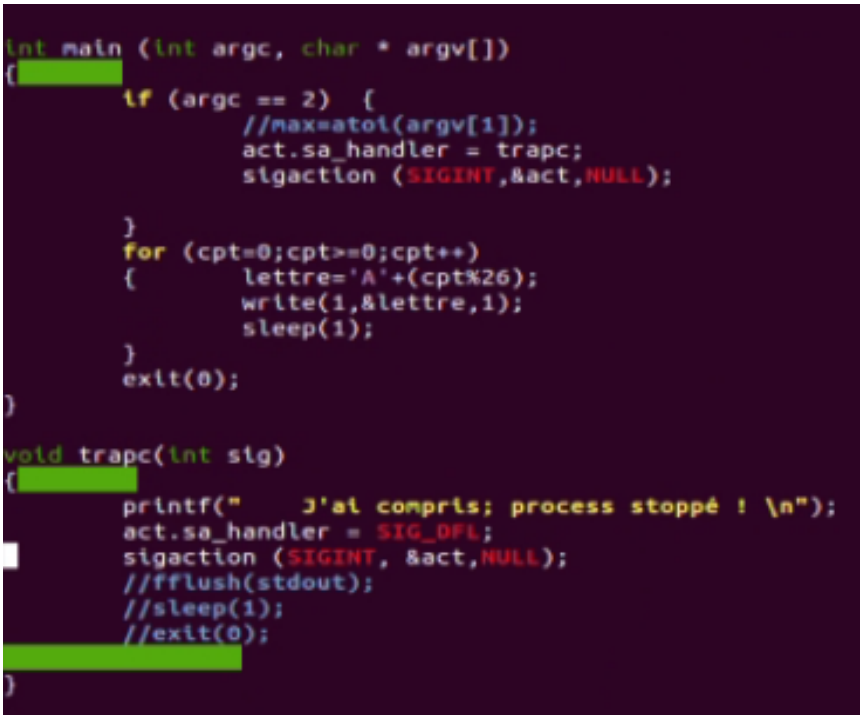
Erreurs dans le programme :

- pas de "closes" en général
- manque 1 virgule au 2ème execlp

Process - LaboProcess 06-01 - trap du ctrl-c

sigaction (SIGINT,&act,NULL) ;
Faire ctrl+c appelle sigint, on peut le redéfinir.

Dans ce code-ci, on doit faire une première fois CTRL-c pour remettre à sigint son état originel. Le deuxième CTRL-c coupe alors le programme.



```
int main (int argc, char * argv[])
{
        if (argc == 2)  {
                //max=atoi(argv[1]);
                act.sa_handler = trapc;
                sigaction (SIGINT,&act,NULL);

        }
        for (cpt=0;cpt>=0;cpt++)
        {       lettre='A'+(cpt%26);
                write(1,&lettre,1);
                sleep(1);
        }
        exit(0);
}

void trapc(int sig)
{
        printf("    J'ai compris; process stoppé ! \n");
        act.sa_handler = SIG_DFL;
        sigaction (SIGINT, &act,NULL);
        //fflush(stdout);
        //sleep(1);
        //exit(0);
}
```

Pour le code suivant :

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

#define __USE_POSIX
#define __USE_POSIX199309
#include <signal.h>

void trapall(int sig, siginfo_t * pinfo, void * pucontext);
struct sigaction act;

int main (int argc, char * argv[])
{
        int noSig;
        int retour;
        char errmessage [256];

        printf ("Bonjour, je suis %d\n", getpid());
        act.sa_flags = SA_SIGINFO;
        act.sa_sigaction = trapall;

        for (noSig=1; noSig<32; noSig++){
                char * err;
                if ((retour = sigaction (noSig,&act,NULL)) < 0) {
                        err = strerror(errno);
                        sprintf (errmessage, "sigaction %d : ", noSig);
                        strncat (errmessage, err,255);
                        fprintf (stderr, errmessage );
                        fprintf (stderr,"\n");
                }
        }

        while (1) pause();
}

void trapall(int sig, siginfo_t * pinfo, void * pucontext){
        printf("reçu le signal=%d\n", pinfo->si_signo);
}
```

Ici tous les signaux sont captés ; sauf le 9 et le 19 qui sont non redéfinissables.

Attention seul CTRL-c et CTRL-z ont un effet de cascade sur la grappe (si on a rien fait de redéfinition etc…) faire kill -2 x ne provoquent pas ce genre d'avalanche.

Process - LaboProcess 06-02 - trap de tous les signaux

```c
void trapall(int sig, siginfo_t * pinfo, void * pucontext){
        printf("reçu le signal=%d\n", pinfo->si_signo);
        act.sa_handler = SIG_IGN;
        sigaction(sig, &act, NULL);
}
```

Rappel :

Pour trouver de la documentation dans le manuel :

- man 7 signal
- man signal
- man sigaction

Process - LaboProcess Ex. Récapitulatif 002

45 78
45 78

Pourquoi ? glob et loc sont définis avant le fork avec des valeurs à 45 et 78
On s'en fout qu'elles soient statiques ou dynamiques.
Le père print.
Le fils est créé , duplique ses variables, les réinitialise mais meurt.
Le père attend son fils qui meurt puis print des valeurs chez le père.

Process - LaboProcess Ex. Récapitulatifs 004

fork() dédouble la mémoire pointée par un pointeur. L'adresse logique du pointeur sera les mêmes mais la réelle adresse mémoire du fils sera différée de part les registres cs ss ds. Pour prouver par exemple :

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


int *pi = NULL;
int main(){
        //chez le pere
        int i = 10;
        pi = &i;
        printf("Valeur du pointeur chez le pere : %d et adresse du pointeur : %x\n", *pi, pi);

        //creation d'un fils
        if(fork() == 0){
                *pi = 20;
                printf("Valeur du pointeur chez le fils : %d et adresse du pointeur : %x\n", *pi, pi);
                exit(0);
        }
        wait(0);

        printf("Valeur du pointeur chez le pere apres le fils : %d et adresse du pointeur : %x\n", *pi, pi);

        exit(0);
}
```

/!\ On fait execvp des commandes externes car les commandes internes modifient l'environnement et généralement le process veut profiter de cet environnement modifié. /!\

Process - LaboProcess Ex. Récapitulatifs 036

Voir chapitre 3.2.

Labo IPC

(nique les ipc)