Travaux Dirigés et Pratiques de Systèmes d'Exploitation (HAI303I)

Michel Meynard

Préambule

Les exercices de TD et de TP sont destinés à vous aider à comprendre le cours. En TP, vous devrez avoir recours aux pages du manuel Unix/Linux disponibles par la commande `man xxx`, ou par l'URL `http://manpagesfr.free.fr`. Certaines distributions utilisent aussi la commande `info` pour décrire les commandes complexes.

1 Introduction, source, objet, compilation

Exercice 1 (TD)
Quelle différence existe-t-il entre un interpréteur et un compilateur ? Citez deux exemples pour chacun d'entre eux.

Exercice 2 (TD)
Quelles similitudes et quelles différences existe-t-il entre un fichier d'en-tête (toto.h) et une bibliothèque ? Quand les utilise-t-on dans le processus de compilation ? Citez deux exemples pour chacun d'entre eux.

Exercice 3 (TD)
Quelle différence entre l'inclusion d'un fichier d'entête standard tel que `stdio.h, stdlib.h, ctype.h` et un fichier d'entête personnel `monprog.h` ?

Exercice 4 (TD)
Comment les paramètres de la ligne de commande sont-ils passés à un programme C ? Comment l'environnement du processus (variables d'environnement telles que PATH, HOME, ...) est-il atteignable par un programme C ?

Exercice 5 (TD/TP)
On souhaite écrire un programme affichant le nombre de paramètres passés à la ligne de commande, ces paramètres, ainsi que les variables d'environnement.

1. Ecrire l'algorithme de ce programme.
2. Ecrire le programme C correspondant.

Exercice 6 (TP)
A l'aide du programme précédent, testez les possibilités du compilateur `gcc` en produisant :
- le fichier prétraité ;
- le fichier assembleur ;
- le fichier objet ;

Observez ces différents fichiers.

Exercice 7 (TD/TP)
On souhaite réaliser le calcul de la moyenne d'une suite de 5 nombres flottants saisis au clavier.

1. Ecrire l'algorithme de ce programme.
2. Ecrire le programme C correspondant.

Exercice 8 (TD/TP)
En utilisant la fonction `atoi` qui convertit une chaîne en entier, réaliser le calcul de la moyenne de la suite des paramètres passés à la ligne de commande : `moy 5 8 12 3`.

1. Ecrire l'algorithme de ce programme.
2. Ecrire le programme C correspondant.

Exercice 9 (TD/TP strsplit)
Dans certains fichiers structurés, les articles sont représentés sur une ligne dont les champs sont séparés par un caractère séparateur (fichiers .csv). On souhaite écrire une fonction `char **strsplit(const char *s, const char sep)` qui découpe une chaîne de caractères correspondant à une ligne csv en un tableau de chaînes dynamiques. Par exemple, l'appel à
`strsplit("/bin:/usr/bin:/usr/local/bin",':')` doit générer le tableau suivant :

```
0 --> /bin
1 --> /usr/bin
2 --> /usr/local/bin
3 NULL
```

On écrira également une fonction `main` qui lira la chaîne et le séparateur depuis la ligne de commande.

1. Ecrire l'algorithme de ce programme.
2. Ecrire le programme C correspondant.

2 Types de données, compilation séparée

Exercice 10 (TD/TP)
Soit une chaîne de caractères C contenant un nombre entier positif en représentation décimale. On veut écrire une fonction convertissant cette chaîne en un entier.

1. Ecrire un algorithme itératif de conversion.
2. Ecrire la fonction C correspondante.

Exercice 11 (TD/TP)
On veut réaliser la conversion inverse de l'exercice précédent. Soit un nombre entier positif que l'on veut convertir en une chaîne de caractères C contenant sa représentation décimale.

1. Ecrire un algorithme de conversion.
2. Ecrire la fonction C correspondante.

Exercice 12 (TD)
Ces algorithmes de conversion (entier vers chaîne, chaîne vers flottant, ...) sont-ils fréquemment employés ? Précisez à quelles occasions.

Exercice 13 (TD/TP)
Soit la **définition** des fonctions suivantes :

— fonction pair

```c
int pair(unsigned int i){
  if (i==0)
    return 1;
  else
    return impair(i-1);
}
```

— fonction impair

```c
int impair(unsigned int i){
  if (i==0)
    return 0;
  else
    return pair(i-1);
}
```

1. Créer un fichier `pair.c` (resp. `impair.c`) qui contienne la définition de la fonction pair (resp. impair). Créer un fichier `pair.h` (resp. `impair.h`) qui contienne la **déclaration** (prototype) de la fonction pair (resp. impair). Créer un programme principal `spair.c` qui réalise l'algorithme suivant :

   ```
   lire l'entier positif n passé à la ligne de commande
   si pair(n) alors
     afficher : n est pair !
   sinon
     afficher : n est impair !
   ```

   Bien entendu, il faut réfléchir au différentes inclusions à réaliser dans les différents fichiers puis compiler les 3 sources .c et lier avec le nom spair et enfin tester ce programme.
2. Ecrire un fichier ppair.c contenant les trois fonctions pair(), impair() et main(). Compiler et lier cet unique fichier en un exécutable ppair. Tester ppair.

3 Code de Huffman

Le codage de Huffman (1952) est une technique de compression de données permettant de stocker un message (un fichier) ou de le transmettre grâce à un minimum de bits afin d'améliorer les performances. Ce codage à taille variable suppose l'indépendance des caractères du message et on doit connaître la distribution probabiliste de ceux-ci à une position quelconque dans ce message. Plus la probabilité d'occurrence d'un caractère dans un fichier est grande, plus son code doit avoir une taille réduite.

On code chaque caractère par un mot de longueur variable de façon à ce qu'aucun mot ne soit le préfixe d'un autre. La propriété principale des codes de Huffman consiste en ce que la longueur moyenne du codage d'un caractère dans un fichier soit minimale.

Pour calculer le code d'un alphabet et de sa distribution, on construit un arbre binaire étiqueté par des 0 et des 1, les feuilles de cet arbre représentant les caractères tandis que les chemins issus de la racine constituent les codes correspondants.

Exemple 1
Soit l'alphabet a,b,c,d,e associé à la distribution probabiliste de la table 1.

Caractère	a	b	c	d	e
Probabilité	0,3	0,25	0,20	0,15	0,10

Table 1 – Distribution probabiliste

On obtient l'arbre binaire de Huffman de la figure 1.

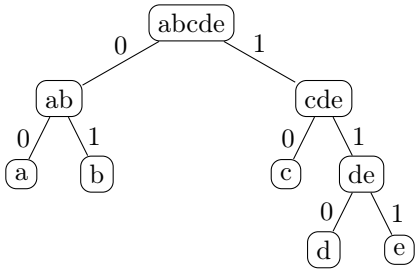

Figure 1 – Arbre de Huffman

En parcourant cet arbre de la figure 1 on aboutit au code de Huffman correspondant à la table 2.

Caractère	a	b	c	d	e
Code	00	01	10	110	111

Table 2 – Code de Huffman

Remarquons que ce code n'est pas unique (il suffit de permuter les 0 et les 1). On calcule la longueur moyenne de codage comme suit :

- 75% des caractères nécessitent 2 bits (a,b,c) ;
- 25% des caractères nécessitent 3 bits (d,e) ;
- $L_{moy} = 75\% * 2 + 25\% * 3 = 2,25$ bits

Remarquons qu'avec un code de taille fixe, chaque caractère aurait nécessité 3 bits, soit une perte de 33%.

3.1 Algorithme de Huffman

L'algorithme consiste à construire l'arbre en partant des deux feuilles "les plus basses" (plus faibles probabilités ici d et e). On leur associe un père, sorte de noeud virtuel dont la probabilité d'apparition du préfixe associé est égale à la somme des probabilités de ses deux fils. On réitère le processus en choisissant à nouveau les deux sommets sans père de plus faible probabilité jusqu'à la construction de la racine.

Exercice 14 (TD)
A l'aide de l'exemple précédent, construisez l'arbre de Huffman en précisant à chaque pas les noeuds créés.

Exercice 15 (TD/TP)
Soit la distribution probabiliste de la table 3 pour les dix caractères représentant les chiffres décimaux.

Caractère	0	1	2	3	4	5	6	7	8	9
Probabilité	0,05	0,1	0,11	0,11	0,15	0,06	0,08	0,2	0,07	0,07

TABLE 3 – Distribution probabiliste

1. Etablir manuellement un arbre de Huffman pour cette distribution en expliquant votre démarche.
2. En utilisant le résultat précédent, établir le code de Huffman pour cette distribution de la table 3 en expliquant votre démarche.
3. Donner la longueur moyenne de codage d'un chiffre du message.

Exercice 16 (TD/TP)
On souhaite calculer la distribution probabiliste des caractères d'un fichier passé en paramètre où les caractères sont stockés sur 8 bits.

1. Décrire une structure de données à base de tableau permettant de calculer puis de stocker la distribution probabiliste.
2. Ecrire l'algorithme pour calculer la distribution probabiliste.
3. Ecrire le programme C correspondant.

Exercice 17 (TD/TP)
1. Décrire une structure de données à base de tableau permettant de construire l'arbre de Huffman.
2. Ecrire l'algorithme pour construire l'arbre ;
3. Ecrire l'algorithme pour calculer le code de Huffman associé.
4. On souhaite implémenter cet algorithme sur des fichiers où les caractères sont stockés sur 8 bits. Quelle est la taille minimale et maximale d'un code ? Comment stocker le tableau des codes dans le fichier codé ?
5. (Projet) Programmer l'algorithme de compression et de décompression en C.

4 Compilation séparée et bibliothèque

Exercice 18 (TD)
Listez les avantages et inconvénients de la liaison statique et dynamique.

Exercice 19 (TP)
En reprenant l'exercice 13, on souhaite construire une bibliothèque statique (`.a`) contenant les fonctions `pair` et `impair` à l'aide de la commande `ar`.

1. Créer les fichiers objets `pair.o` et `impair.o`. Puis éditer leurs liens avec le fichier `spair.c` après compilation en construisant l'exécutable `spair2`.
2. Créer une bibliothèque statique `libpair.a` contenant les deux fichiers objets `pair.o` et `impair.o`. Vérifier le contenu de `libpair.a`.
3. Compiler spair.c et éditer ses liens avec votre bibliothèque dans un exécutable spair3. Tester spair3.

Exercice 20 (TP)
En reprenant l'exercice 13, on souhaite construire une bibliothèque dynamique (`.so`) contenant les fonctions `pair` et `impair` à l'aide de la commande .

1. Créer la bibliothèque dynamique `libpair.so` contennat les fichiers objets `pair.o` et `impair.o`.
2. Créer une bibliothèque statique `libpair.a` contenant les deux fichiers objets `pair.o` et `impair.o`. Vérifier le contenu de `libpair.a`.
3. Compiler `spair.c` dans un exécutable `spair4` puis tester ses liens dynamiques.

5 Entrées/Sorties

Exercice 21 (TD)
Quelle différence entre l'utilisation d'appels systèmes `man 2` (open, read, close) ou bien des fonctions de bibliothèque `man 3` (fopen, fprint, ...) ?

Exercice 22 (TD/TP)
On souhaite compter les caractères d'un fichier passé en argument à la ligne de commande en utilisant des appels systèmes (`wc -c`).

1. Ecrire un algorithme
2. Ecrire le programme C correspondant.

Exercice 23 (TD/TP)
On souhaite refaire l'exercice précédent en utilisant des fonctions de bibliothèque. Que faut-il faire ? Faites-le.

Exercice 24 (TD/TP)
Un filtre est un programme qui lit des données sur l'entrée standard et qui rejette dans sa sortie standard les données modifiées. On souhaite écrire un filtre `monhead` qui ne rejette que les n premières lignes d'un fichier. Par exemple :
`head -3 monfic.txt`
Cette commande permet d'afficher les 3 premières lignes du fichier `monfic.txt`.

1. Qu'est-ce qu'une ligne ? Faut-il utiliser appel système ou fonction de bibliothèque ?
2. Ecrire un algorithme
3. Ecrire le programme C correspondant.
4. l'option `-c43` de `head` permet de lire les 43 premiers caractères. Implémenter cette option.

6 Filtres et Entrées/Sorties formattées

Exercice 25 (TD/TP)
On souhaite écrire un filtre `montail` qui ne rejette que les n dernières lignes d'un fichier.

1. Quelle structure de données nécessite ce programme ? Décrivez précisément cette structure de données !
2. Ecrire un algorithme
3. Ecrire le programme C correspondant.
4. l'option `-c43` de `tail` permet de lire les 43 derniers caractères. Implémenter cette option.

Exercice 26 (TD)
Etudiez le programme suivant : **programme testaffichint.c**

```c
#include <stdio.h>                /* printf */
int main(int argc, char *argv[], char *env[]) {
  int i=12345;
  printf("%4.0d\n",i);
  fwrite(&i,sizeof(int),1,stdout);
}
```

L'affichage obtenu est le suivant :

```
$testaffichint
12345
90  $
```

1. Quelle différence esiste-t-il entre ces deux affichages ?
2. Précisez la valeur de `90__`. Quel est le "boutisme" (endianness) de cette machine ?
3. Quels avantages et inconvénients de ces types de codage dans les fichiers en matière de sauvegarde de données ?

Exercice 27 (TD/TP fichiers .csv)
En examinant certains fichiers de configuration situés dans le répertoire `/etc`, tels que `passwd`, `fstab`, `group`, on voit que ces fichiers textes sont composés d'articles d'une ligne, chaque article étant composé de champs séparés par un caractère (`:`). On souhaite écrire une commande `monselect` correspondant de manière simplifiée à l'instruction `SELECT` de SQL. Les champs étant numérotés de 1 à n, l'exemple suivant affichera chaque ligne du fichier `passwd` dont le deuxième champ est égal à "dupont" :

`monselect /etc/passwd : 2 dupont`

1. Ecrire un algorithme
2. Ecrire le programme C correspondant.

Exercice 28 (TP sauvegarde binaire)
Soit un tableau d'étudiants que l'on veut sauver (écrire) dans un fichier au format binaire. Suivent les définitions de type et de ce tableau :

```
typedef struct {
  int code;     /* code étudiant 20091234 */
  char nom[20]; /* "dupont" */
  char dnais[8]; /* 19901231 */
} etudiant;
etudiant tabetu[100]={{20091234,"dupont","19891231"},
              {20091235,"martin","19891130"},
              {0,"",""}
}; /* tableau des étudiants */
```

Le tableau est compacté vers les indices faibles et le premier article (struct)ayant un code nul indique la fin des étudiants.

1. Ecrire quelques instructions C permettant d'afficher le tableau des étudiants.
2. Ecrire un programme C permettant de sauver le tableau dans un fichier binaire passé en argument et ne contenant que les étudiants présents.
3. Visualiser le fichier sauvé en hexadécimal (`hexl` ou `od -cx`) et donner une interprétation.
4. Ecrire un programme C permettant de lire le fichier sauvé afin de le stocker dans un tableau puis afficher le résultat.

Exercice 29 (TD/TP hexl)
La commande `hexl` permet de visualiser un fichier sous sa forme hexadécimale (dump hexa) et sous sa forme texte. Chaque ligne du dump est composé de la position en hexadécimal, de 16 codes hexa, de 16 caractères affichables. L'exemple suivant illustre le propos :

```
hexl compte.c
00000000: 2369 6e63 6c75 6465 203c 7374 6469 6f2e  #include <stdio.
00000010: 683e 0909 0d0a 2369 6e63 6c75 6465 203c  h>....#include <
00000020: 7379 732f 7479 7065 732e 683e 0909 2f2a  sys/types.h>../*
00000030: 206f 7065 6e20 2a2f 0d0a 2369 6e63 6c75   open */..#inclu
```

1. Ecrire l'algorithme ;
2. Ecrire le programme C correspondant en utilisant des **appels systèmes**.

Exercice 30 (TD/TP Positionnement dans un fichier)
Soit un fichier contenant des caractères 8 bits uniques et triés dans l'ordre croissant de leur code ASCII comme dans l'exemple suivant :

```
15678ACFGJKLMXYZabcduvw
```

On souhaite tester la présence d'un char dans ce fichier en faisant une recherche dichotomique (on divise l'espace en deux à chaque pas). Par exemple :

```
dicho fic.txt 8
Le caractère 8 est en position 4
```

Questions

1. Comment connaître la taille du fichier ?
2. Ecrire l'algorithme.
3. Ecrire le programme C `dicho.c` en utilisant des **appels systèmes**.

Exercice 31 (TD/TP)
L'objectif est de comparer ce qui se passe lorsqu'on utilise des appels système et les fonctions de bibliothèque.

1. Quels sont les avantages et inconvénients d'utiliser l'une ou l'autre approche ?
2. On peut (doit) réécrire certains exercices en utilisant ces fonctions de bibliothèque.

7 Gestion des Processus

7.1 Génération de processus

Exercice 32 (TD/TP)
On veut voir un processus dit «parent» générer un autre dit «descendant», chaque processus affichant son identité (son numéro de processus) et celle de son parent. Pour obtenir les identités, utiliser les appels système ci-après.

```
NOM
       getpid, getppid - Obtenir l'identifiant d'un processus.
SYNOPSIS
       #include <unistd.h>

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

--------------------------------------------------
NOM
       fork - Créer un processus fils (child).
SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);
```

Exercice 33 (TD/TP Génération multiple)
Soit le programme C suivant : **genproc4.c**

```c
#include <stdio.h>
#include <unistd.h> //fork(), getpid(),
#include <sys/types.h> //toutes
#include <sys/wait.h>

int main(){
  for(int i=0 ; i<4 ; i++){
    pid_t nPid;
    if ((nPid = fork())==0){
      printf("un nouveau descendant %d  de parent %d ! i=%d\n",getpid(), getppid(),i);
    }else{
      int status;
      wait(&status);}//chaque parent attend la fin de ses enfants
  }
}
```

Dessinez l'arbre des processus créés par ce programme. Combien de processus existent au total ? Quel est le dernier processus vivant ?

Exercice 34 (TD/TP Duplication des segments)
Après clônage d'un processus par `fork()`, deux processus indépendants coexistent, et chacun a son propre contexte d'exécution, ses propres variables et descripteurs de fichiers. On veut mettre en évidence ce phénomène.

1. Quelles solutions peut-on proposer ? **Attention** : Il s'agit de mettre en évidence la duplication de **tous** les segments mémoire et les solutions proposées doivent en tenir compte.
2. Écrire le programme permettant d'illustrer que chaque processus possède ses propres variables.
3. En cas de lecture ou d'écritures de deux processus sur un fichier ouvert par un ancêtre commun, qu'en resulte-t-il ?

Exercice 35 (TP)
Voici quelques exercices supplémentaires :

1. Lancer un processus quelconque dans une fenêtre de type «terminal» (appelons la $fenetre_1$). Le processus doit être un programme qui ne se termine pas rapidement : un exécutable que vous avez créé, un éditeur de texte, etc, selon votre choix. Lancer ce processus en tache de fond. Dans une autre fenêtre tuer le processus «shell» de la $fenetre_1$. Que se passe-t-il pour le processus lancé ?

2. Même question si le processus n'est pas lancé en tâche de fond ? Quelles sont vos conclusions ?
3. Créer un programme permettant d'identifier les divers parents d'un processus, en fonction du moment de la disparition du générateur.

7.2 Recouvrement d'un processus

On dénote ici `exec()` la famille des appels de recouvrement de processus.

Exercice 36

1. Écrire un petit programme, affichant un texte quelconque, puis se recouvrant par `ls` grâce à `execl` ; la commande `ls` est située généralement dans `/bin`.
2. Écrire un petit programme, affichant un texte quelconque, puis se recouvrant par `ls -l /bin` ; (utilisation de `execl()`)
3. (difficile) En supposant que vous ne connaissez pas l'emplacement de `gcc`, écrire programme, affichant un texte quelconque, puis se recouvrant par sa propre compilation ! On utilisera `execlp` pour mettre en œuvre la recherche de l'exécutable selon le contenu de la variable d'environnement `PATH`.

   Pour les TP, on prévoira de modifier la variable d'environnement `PATH` pour constater que la recherche a bien lieu **seulement** selon les chemins indiqués dans cette variable.

7.3 Compléments

Exercice 37

— Quelles différences y a-t-il entre un appel `system()` et `exec()` ?
— Quel est le nom du processus avant et après recouvrement ?

7.4 Questions extraites d'examens sur le recouvrement

On a vu dans le cours qu'un recouvrement de processus n'engendrait pas un nouveau processus, bien que les segments mémoire étaient remplaçés.

Exercice 38

Pour le mettre en évidence, écrire deux très petits programmes (les plus petits possibles), montrant qu'avant et après recouvrement l'identité du processus est inchangée.

Exercice 39

Montrer en modifiant ou reécrivant vos programmes, qu'on peut avoir un nombre infini de recouvrements pour un même processus.

7.5 Questions extraites d'examens sur les processus

Exercice 40

Proposer un algorithme qui étant donné un nombre entier $n > 2$ génère exactement n processus clônes.

Exercice 41 (TD/TP Mon Shell)

On souhaite écrire un interpréteur de commande rudimentaire (sans argument).

1. Ecrire l'algorithme ;
2. Ecrire le programme correspondant.

8 Système de fichier

8.1 droits d'accès et de suppression de fichiers réguliers

Exercice 42 (TD/TP)

1. Sous unix, quels sont les droits nécessaires pour pouvoir supprimer un fichier ?
2. Quels sont les droits nécessaires pour modifier le contenu d'un fichier ?
3. Est-ce que le propriétaire d'un fichier peut s'enlever ses propres droits de lecture, écriture et exécution sur un fichier ? Est-ce grave ?

A vérifier en TP.

Exercice 43 (TD)
Selon les différents Unix et la politique de l'administrateur système, la gestion des groupes d'utilisateurs varie fortement. Sous Linux, chaque utilisateur est par défaut dans son groupe privé.

Créer un répertoire dans lequel on met deux fichiers `tempo` et `toto`. Donner les droits nécessaires à la suppression de fichiers, pour un collègue du même groupe et lui demander de supprimer `tempo`. Faire en sorte qu'il ne puisse pas supprimer `toto`. Pour changer de groupe, il faut utiliser la commande `newgrp`, à étudier. Pour connaitre les groupes où on est inscrit , utiliser la commande `id`.

Écrire un programme C permettant de mettre en évidence les droits de suppressions de fichiers. Pour ce faire, utiliser l'appel système `unlink()`, et afficher les résultats et les erreurs en fonction des droits liés tant aux fichiers qu'aux répertoires contenants.

Exercice 44 (TD/TP)
Que signifient les droits `rwx` sur un répertoire ? On veut donner à quelqu'un l'accès à un fichier en lecture et écriture, mais on ne veut pas qu'il puisse prendre connaissance du répertoire contenant. Comment faire ? Que peuvent faire tous les autres utilisateurs ?

Exercice 45 (TD/TP)
Quelles sont les possibilités offertes lorsqu'on a le droit d'écriture sur un répertoire ? En TP, on tentera la création d'un répertoire contenant un fichier régulier dans le répertoire d'un collègue. Ce dernier peut-il supprimer cette arborescence étrangère ?

Exercice 46 (TP)
Pour les TP, préparer un ensemble de manipulations, afin de mettre en évidence la signification des droits. Par exemple :

- enlever le droit de lecture sur un répertoire et chercher à modifier un fichier dans ce répertoire,
- enlever le droit d'exécution sur un répertoire et chercher à atteindre un fichier inclus,
- enlever le droit d'écriture sur un répertoire et supprimer un fichier inclus (qu'on soit propriétaire de ce fichier ou non),
- etc, à votre imagination.

8.2 Cohérences dans la table des inodes

Les situations décrites ci-après correspondent à la vision d'une partie de la table des inodes d'un système de fichiers unix. Vous devez répondre face à chaque situation si elle vous paraît cohérente ou non. Considérez chaque situation comme un cas séparé et ne pas tenir compte de la colonne *type*. N'oubliez pas de justifier votre réponse et de signaler toutes les anomalies que vous trouvez.

les blocs d'allocation ont une taille de `8 k-octets`. la notation `(0)` signifie "libre". On admettra que tous les blocs qui suivent le premier bloc libre sont libres.

situation	numéro d'inode	type	taille	blocs d'adressage direct					...
1	148		29000	4776	4786	7021	7022	(0)	(0)
	272		38000	2415	4728	4014	17012	30202	(0)
2	2405		37000	2405	4718	4004	17002	(0)	(0)
3	256		27000	4102	8204	2307	6409	(0)	(0)
	498		38000	1205	6144	4102	1025	1026	(0)

Exercice 47 (TD)
Corriger les situations incohérentes et proposer pour chaque situation un ensemble cohérent.

On regroupe l'ensemble des situations en une seule. On précise en outre les données relatives aux liens.

inode	nb. liens
148	3
272	1
2405	1
256	5
498	2

Exercice 48 (TD)
À la seule vue de ce tableau de liens, peut-on dire si une entrée est un fichier simple ou un répertoire ?

8.3 Cohérence fichiers et répertoires

Exercice 49
On propose maintenant deux organisations possibles de répertoires. Est-ce que chaque organisation est cohérente avec les données ci-dessus ?

organisation	répertoire 1		répertoire 2	
1	fibre	148	fini	272
	finioui	2405	fininon	148
2	fibonacci	498	fibonnacci	498
	figue	148	figue	148
	filon	148		
	fichtre	498		

Remplir dans le premier tableau la colonne *type* et ajouter les lignes correspondant aux répertoires.

8.4 Appels système pour les droits d'accès

Exercice 50 (TD)
La commande `chmod` et l'appel système `chmod()` existent. Quelle est l'utilité de chacun ?

Exercice 51 (TD)
On admet que les droits d'accès à un fichier peuvent être modifiés pendant l'exécution du programme (voir plus loin, on réalisera ceci en Tp). D'où la question : si on peut modifier les droits en cours d'exécution, peut-on faire des opérations en contradiction avec les droits ? En quelque sorte, quand est-ce qu'une modification de droits devient effective ?

Exercice 52 (TD)
Voici un extrait de l'entête du manuel :

```
NAME
       chmod, fchmod - change permissions of a file

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

       int chmod(const char *path, mode_t mode);
       int fchmod(int fildes, mode_t mode);
```

Quelle est la différence entre ces deux appels ?

Pour les Tp, faire un programme qui ouvre un fichier, puis qui utilise l'appel système `chmod()` pour modifier les droits en cours d'exécution. Proposer un moyen pour visualiser les droits du fichier avant puis après cet appel système. Proposer ensuite une solution permettant de distinguer et mettre en évidence les droits vus par le programme et ceux vus par l'utilisateur. Ajouter cette solution à votre programme.

8.5 Droits d'accès à la création

On peut utiliser des constantes prédéfinies ou une valeur équivalente en octal, partout où il est nécessaire de déterminer les droits d'accès aux fichiers. Tous les appels système faisant intervenir ces droits peuvent les utiliser. C'est une donnée de type `mode_t`, nécessitant l'inclusion de `<sys/types.h>` pour sa définition.

Remarque 1 : Lors de la création de fichiers, ce ne sont pas exactement ces droits qui sont mis comme droits d'accès. En fait, il est tenu compte aussi de la variable d'environnement `umask`. Les droits définitifs sont obtenus par l'opération :

`(droits demandés) et (non umask)`

Voir ci-après pour quelques détails concernant cette variable d'environnement.

Remarque 2 : Dans certains systèmes, ces macro-définitions sont faites autrement, mais on aboutit à des résultats similaires.

```
#define S_IRWXU 0000700                 /* RWX mask for owner */
#define S_IRUSR 0000400                 /* R for owner */
#define S_IWUSR 0000200                 /* W for owner */
#define S_IXUSR 0000100                 /* X for owner */

#define S_IRWXG 0000070                 /* RWX mask for group */
#define S_IRGRP 0000040                 /* R for group */
#define S_IWGRP 0000020                 /* W for group */
#define S_IXGRP 0000010                 /* X for group */

#define S_IRWXO 0000007                 /* RWX mask for other */
#define S_IROTH 0000004                 /* R for other */
#define S_IWOTH 0000002                 /* W for other */
#define S_IXOTH 0000001                 /* X for other */
```

8.6 umask

La **commande** `umask` permet de :
- connaître le masque courant
- affecter une nouvelle valeur au masque : `umask 026`

Lors de la création d'un nouveau fichier (`creat` ou `open`) le masque courant est appliqué sur les droits proposés à des fins de sécurité . L'utilisateur devra réaliser un chmod après création pour obtenir ce qu'il veut. L'opération réalisée est :

`(droits obtenus)=(droits proposés) et (non umask)`

Pour les fichiers réguliers, l'opération non est du complément à 2, c'est à dire que le droit d'exécution n'est jamais obtenu à la création. Pour les répertoires, c'est un non binaire.

Avec des droits proposés de **0777**	`umask` 022	crée les fichiers avec les droits `rw-r--r--`
	`umask` 024	crée les fichiers avec les droits `rw-r---w-`
	`umask` 066	crée les répertoires avec les droits `rwx--x--x`

Exercice 53 (TD)
Que faut-il indiquer dans les paramètres de l'appel système *open()*, pour créer un fichier accessible en écriture seulement au propriétaire, si `umask` vaut 222 ?

8.7 partage de fichiers

Cette question a été déjà abordée dans un Td précédent. On la complète. On veut faire la distinction dans l'accès aux fichiers entre des processus l'obtenant par héritage et ceux l'obtenant par une demande explicite d'ouverture.

Exercice 54 (TD)
Est-ce que deux processus peuvent partager un même fichier ouvert en lecture ? en écriture ? On sait que deux processus issus d'une même hiérarchie partagent les mêmes descripteurs de fichiers (`fork()` duplique les descripteurs). Que se passe-t-il lorsque les deux processus écrivent sur ce fichier ?

Exercice 55 (TD)
Mettre en évidence deux processus qui écrivent sur un même fichier sans hériter des descripteurs du même père. Quelle est la différence avec la situation précédente ?

8.8 numéros d'inodes et liens physiques (durs)

Exercice 56 (TD)
Quelle commande permet de visualiser le numéro d'inode d'un fichier ? Et le nombre de liens (liens *physiques* dits *durs*) ?

Exercice 57 (TD)
Partant d'un fichier et de son numéro d'inode, comment peut-on trouver l'ensemble des noms qui référencent cet inode ?

En TP : Créer des liens sur un fichier vous appartenant : un lien dans votre espace de vision (un répertoire vous appartenant), un autre lien qu'un collègue va faire, dans un répertoire lui appartenant, en utilisant si besoin les groupes pour qu'il puisse l'accéder. Constater que le nombre de liens est bien mis en évidence. Modifier alors les droits d'accès au répertoire contenant le fichier de départ. Est-ce que le fichier reste accessible au collègue ?

8.9 Une utilisation dangeureuse

Beaucoup d'éditeurs de texte prennent la peine de créer une sauvegarde du fichier édité avant de laisser l'utilisateur faire des modifications. Soit *figaro* le fichier à éditer. L'éditeur procède ainsi :

— au lancement il renomme *figaro* en *figaro.levieux*
— il laisse ensuite l'utilisateur faire son travail, jusqu'à la demande de sauvegarde,
— la demande de sauvegarde se fait en recréant *figaro*, en lui attachant les droits d'accès définis par la variable d'environnement `umask` (cf. ci-dessus).

Exercice 58 (TD)

1. Montrer comment le fichier sauvegardé et le fichier original peuvent avoir des droits différents. Les questions suivantes peuvent être traitées en admettant que cette différence de droits est possible.
2. Si un lien dur pointait sur le fichier avant édition, quel est le fichier référencé après édition ?
3. Même question que ci-dessus (question 2) avec un lien symbolique.

Exercice 59 (TP - liens symboliques)

Créer des répertoires temporaires et faire des liens symboliques sur ces répertoires, toujours par vous-même et par un collègue. Cette fois-ci peut-on savoir qu'un lien existe sur ce répertoire ? Que se passe-t-il :

— lorsqu'on supprime un lien ?
— lorsqu'on supprime le répertoire sans supprimer les liens ?
— tenter de construire un circuit et faites un `cd` sur un élément de ce circuit. Que-se passe-t-il ?

Est-il nécessaire de passer par le même groupe pour créer des liens symboliques vers les fichiers et répertoires des collègues ?

8.10 Système de fichiers : accès aux inodes et répertoires

Exercice 60 (TD Taille des fichiers)

On suppose que :

— les blocs alloués sont de `2K-octets` ;
— Les adresses de blocs sont sur 32 bits ;
— la disposition des pointeurs dans l'inode est de 10 pointeurs directs, 1 pointeur à un niveau d'indirection, 2 pointeurs à deux niveaux et 2 pointeurs à trois niveaux ;
— la taille du fichier est codée sur 32 bits ;

1. Calculer la taille maximale allouable à un fichier dans un tel système de fichier ;
2. Quelles autres informations peuvent encore modifier cette limite ?
3. Est-il possible qu'il reste des blocs disponibles sur une partition disque (système de fichiers), sans que l'on puisse ajouter un nouveau fichier ?

Voici un extrait de l'appel système d'accès à un inode ainsi que la structure d'une entrée de la table des inodes, telles qu'elles sont décrites dans le manuel :

```
NOM
  stat,  fstat, lstat - Obtenir le statut d'un fichier (file status).

SYNOPSIS
  #include <sys/stat.h>
  #include <unistd.h>

  int stat(const char *file_name, struct stat *buf);
  int fstat(int filedes, struct stat *buf);
  int lstat(const char *file_name, struct stat *buf);
  ...
Les trois fonctions modifient une structure stat déclarée ainsi

struct stat{
  dev_t         st_dev;      /* Périphérique               */
  ino_t         st_ino;      /* Numéro i-noeud             */
  umode_t       st_mode;     /* type de fichier et droits  */
  nlink_t       st_nlink;    /* Nb liens matériels         */
  uid_t         st_uid;      /* UID propriétaire           */
  gid_t         st_gid;      /* GID propriétaire           */
```

```
  dev_t         st_rdev;     /* Type périphérique           */
  off_t         st_size;     /* Taille totale en octets     */
  unsigned long st_blksize;  /* Taille de bloc pour E/S     */
  unsigned long st_blocks;   /* Nombre de blocs alloués     */
  time_t        st_atime;    /* Heure dernier accès         */
  time_t        st_mtime;    /* Heure dernière modification */
  time_t        st_ctime;    /* Heure dernier changement état*/
};
```

Ainsi, l'appel `stat()` permet d'obtenir les informations de la table des inodes concernant tout élément de cette table. Il délivre à partir du chemin d'un fichier (quel que soit son type) une structure dont la forme est donnée ci-dessus.

Exercice 61 (TD Accès à la table des inodes)

1. à quoi correspondent les divers champs de cette structure (rappel du cours) ?
2. quelle différence entre stat, fstat et lstat ?

Notons que `st_mode` contient non seulement les droits d'accès au fichier, mais aussi le type du fichier. En fait, dans ce mot de 16 bits, il y a 4 bits correspondant au type du fichier. Pour ne pas avoir à chercher où sont localisés ces divers éléments, des masques sont prédéfinis. Le masque `_S_IFMT` permet d'extraire les 4 bits correpondants au type du fichier. Ainsi,

Si`(st_mode & S_IFMT)` donne	le fichier est	et on peut tester
`S_IFREG`	régulier	`S_ISREG(buf.st_mode)`
`S_IFDIR`	un répertoire	`S_ISDIR(buf.st_mode)`
`S_IFLNK`	un lien symbolique	`S_ISLNK(buf.st_mode)`

Notons que les droits d'accès peuvent également être consultés grâce à une famille de masques tels que :

— `S_IRUSR` pour la lecture par le propriétaire ;
— `S_IWGRP` pour l'écriture par le groupe ;
— `S_IXOTH` pour l'exécution par les autres ;

Exercice 62 (TD et TP)

1. Écrire un programme `typeFichier.c` permettant d'afficher si un nom donné en paramètre est un fichier régulier, un répertoire, un lien ou est d'un autre type.
2. Écrire un programme `droitFichier.c` permettant d'afficher les droits d'accès du fichier correspondant au format de `ls -l` : `-rw-r-r-` ou `drwx-r-xr-x` ou `lr-xr-x--`.

8.11 Opérations sur les répertoires

Un répertoire est un fichier : une suite de caractères. Mais ces caractères sont structurés en articles constitué d'un couple *(numéro d'inode, nom)*. On peut utilises des entrées-sorties classiques pour les accéder. Cependant, plusieurs fonctions de bibliothèque spécifiques d'accès aux répertoires existent. Elles sont décrites dans la section 3 du manuel (fonctions de bibliothèques). Voici quelques exemples :

1. ouverture/fermeture d'un répertoire :

```
#include <dirent.h>
DIR *opendir(const char *chemin);
int closedir (DIR *pdir);
```

2. lecture de l'enregistrement suivant :

```
#include <dirent.h>
struct dirent *readdir(DIR *pdir);
```

3. création/destruction d'un répertoire :

```
#include <sys/types.h>
int mkdir (const char *nom_rep, mode_t droits);

#include <unistd.h>
int rmdir (const char *nom_rep);
```

4. positionnement, récupération de la position, et raz dans le répertoire

```
void seekdir(DIR *dir, off_t offset);
off_t telldir(DIR *dir);
void rewinddir(DIR *dir);
```

`DIR` est une structure qu'il n'est pas nécessaire de connaître en détail. C'est un bloc de contrôle permettant au système d'identifier le répertoire ouvert et il est alloué dans le tas lors de l'ouverture du répertoire. Ensuite, chaque `readdir()` fait évoluer un champ de ce bloc pointant sur l'entrée courante. Il est primordial de fermer le répertoire (`closedir()`) à la fin de son utilisation afin d'éviter une **fuite mémoire**. Enfin, chaque `readdir()` retourne un pointeur sur le même champ `dirent` de la structure DIR qui évolue donc à chaque appel ! La fonction `readdir()` n'est pas réentrante.

Exercice 63 (TD)

1. Comment peut-on justifier l'existence de fonctions spécifiques, c'est-à-dire différentes de celles sur les fichiers simples ? Pour répondre, on peut commencer par faire la liste des opérations classiques qu'on fait sur un fichier : ouverture, fermeture, lecture, écriture, positionnement, etc.
2. Comparer la création d'un nouveau fichier et celle d'un répertoire.
3. Comment écrire une nouvelle entrée dans un réperoire ?
4. Quelles vérifications sont effectuées lorsqu'on veut détruire un répertoire ? Comparer à celles faites pour un fichier.

8.12 Contenu d'un répertoire

Voici la structure `dirent` (cf. `/usr/include/dirent.h` ou manuel `readdir()`) introduite dans le paragraphe précédent :

```
struct dirent{
  long d_ino;                 /* inode number */
  off_t d_off;                /* offset to this dirent */
  unsigned short d_reclen;    /* length of this d_name */
  char d_name [NAME_MAX+1];   /* file name (null-terminated) */
}
```

Remarque : Noter que cette structure diffère entre systèmes Unix ; certains systèmes ajoutent des champs, mais il y a accord sur les quatre champs ci-dessus.

Exercice 64 (TD et TP)
Reprendre l'exercice 62, afin d'afficher (monls.c) le contenu d'un répertoire au format d'un `ls -ali` : inode type droits nomFichier

8.13 Gestion des dates

La notion de *date* pour les fichiers représente une durée (en secondes et éventuellement microsecondes) écoulées depuis le 1^{er} janvier 1970. Le type `time_t` représente cette durée écoulée et varie selon les systèmes Unix.

```
double difftime(time_t t1, time_t t0); /* nb secondes depuis t0 jusqu'à t1 */
```

Un autre type de date `struct tm` est utilisé pour l'interface avec l'homme et est composé de champs indiquant l'année, le mois, le jour, ...

```
struct tm {
  int     tm_sec;         /* seconds */
  int     tm_min;         /* minutes */
  int     tm_hour;        /* hours */
  int     tm_mday;        /* day of the month */
  int     tm_mon;         /* month */
  int     tm_year;        /* year */
  int     tm_wday;        /* day of the week */
  int     tm_yday;        /* day in the year */
  int     tm_isdst;       /* daylight saving time */
};
```

De plus, des fonctions de conversions existent afin de traduire un type dans l'autre :

```
struct tm *localtime(const time_t *timep);
time_t mktime(struct tm *tm);
```

Enfin des fonctions de formattage permettent d'obtenir des chaînes de caractères et !

```
char *ctime (const time_t *timep); /* non réentrant : "Wed Jun 30 21:49:08
1993\n" */
```

Pour obtenir la date courante, il faut utiliser la fonction suivante :

```
struct timeval {
  time_t      tv_sec;  /* secondes */
  suseconds_t tv_usec; /* microsecondes */
};
int gettimeofday(struct timeval *tv, NULL);
```

Exercice 65 (TD et TP)
Écrire un programme `lsmodif.c` permettant d'afficher la liste des fichiers et répertoires d'un répertoire donné en argument qui ont été modifiés depuis moins de n jours, n étant donné en argument. On affichera le nom du fichier et la date de modification. Par exemple, `lsmodif . 2` affichera la liste des fichiers récement modifiés (moins de 2 jours).

8.14 Parcours récursif d'un sous-arbre

Pour parcourir récursivement une arborescence issue d'un répertoire, il faut écrire une fonction récursive prenant en entrée un chemin de répertoire, qui va itérer sur toutes les entrées du répertoire et qui va s'appeler récursivement lorsque cette entrée sera un répertoire.

Exercice 66

1. Ecrire un algorithme de cette fonction récursive `parcours(rep)` en supposant que tout les appels systèmes se passent bien !
2. Écrire un programme `arboindent.c` permettant d'afficher récursivement la liste indentée des noms de répertoires d'un répertoire donné en argument.

D'autres parcours récursifs peuvent être utiles pour :
— calculer le volume nécessaire à la sauvegarde d'une arborescence : somme des tailles des fichiers et répertoires ;
— copier récursivement une arborescence ;
— vérifier que les liens symboliques ne forment pas un circuit ;
— ...

Exercice 67 (TD/TP)

1. Ecrire une fonction générique de parcours récursif d'une arborescence à partir d'un répertoire et qui exécute un certain **traitement** sur les fichiers répondant à une certaine **condition**. Le prototype de cette fonction suit.

   ```
   /**
    * fonction récursive parcourant un répertoire et appliquant un traitement
    * à tous les fichiers parcourus sous condition
    */
   int parcours(char* rep, int (*condition)(char *), void (traitement)(char *));
   ```

2. Ecrire des conditions `estRep()`, `estReg()`, `estLienSymb()` et les traitements `affiche()`, ...

8.15 Sauvegarde incrémentale

Une sauvegarde incrémentale permet de ne sauvegarder que les fichiers qui ont été créés ou modifiés depuis une date donnée, qui est souvent la date de la dernière sauvegarde.

Exercice 68

1. On se restreint à un et un seul répertoire pour l'instant. Quelle solution peut-on proposer pour qu'un tel système fonctionne automatiquement ?

8.16 Question subsidiaire : nom et numéro de propriétaire

Pour faire la liaison entre le numéro du propriétaire inscrit dans la structure d'un *inode* et le nom du propriétaire, un intermédiaire supplémentaire (un autre appel système) est nécessaire. Voici une partie de la page du manuel :

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

struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);
```

...et une partie de la structure `passwd` :

```
struct passwd {
  char    *pw_name;
  char    *pw_passwd;
  uid_t   pw_uid;
  gid_t   pw_gid;
  char    *pw_age;
  char    *pw_comment;
  char    *pw_gecos;
  char    *pw_dir;
  char    *pw_shell;
};
```

Exercice 69 (TD)
Quelles justifications peut-on fournir à une telle démarche ? Peut-on citer au moins un inconvénient ?

8.17 Autres appels noyau du système de fichiers

De la même façon que l'on peut changer de répertoire ou modifier les droits d'accès et de propriété d'un fichier par des commandes, on peut aussi manipuler les caractéristiques des fichiers par des primitives du noyau.

Par exemple, des appels comme `chdir(2)`, `chown(2)`, existent (heureusement !) ; dans un TD précédent nous avons vu `chmod(2)`. La création de liens se fait par `link(2)` ; la destruction de ces liens, donc finalement la destruction de fichiers, par `unlink(2)` (attention...). Enfin, `readlink()` permet de lire le contenu d'un lien symbolique (c'est-à-dire le pointeur, pas le pointé).

Pour les TP, on préparera des petits programmes permettant de se rendre compte de l'effet des appels cités ci-dessus. Par exemple :

- Essayer `chdir(2)` dans un répertoire inexistant et afficher l'erreur système récupérée.
- Ouvrir un fichier, faire des entrées-sorties et lancer un autre programme (dans une autre fenêtre) qui efface ce fichier. Que se passe-t'il ? Afin de ne pas ajouter au résultat recherché l'effet du réseau sur le système de fichiers, il est conseillé de faire cet exercice avec un fichier localisé dans le répertoire `/tmp`. Peut-on déduire que l'on peut dissimuler une activité sur des fichiers ?
- Constater que `unlink(2)` détruit des fichiers, mais peut-on le faire aussi sur des répertoires ? Voir `rmdir(2)` avant de se prononcer.
- Essayer d'effacer un répertoire vide dans lequel un processus a fait `chdir(2)`.
- Sans oublier `readlink()`, qui permettra de visualiser pointeur et pointé.

9 Gestion des signaux

En ligne de commande :

- `kill -l` permet de lister les signaux ;
- `ps` liste les processus en cours ;
- `kill pid` termine le pus ;

On rappelle l'appel système `sigaction` :

```
#include <signal.h>
int sigaction(int signum,
  const struct sigaction *act,
  struct sigaction *oldact);
```

Et voici la structure `sigaction` :

```
struct sigaction {
  void     (* sa_handler)   (int);
  sigset_t    sa_mask;
  int         sa_flags;
}
```

On veut gérer le signal d'arrêt de processus (signal *interrupt* ou SIGINT). Attention, il faut que l'arrêt du processus reste possible, par un autre moyen forcément.

Exercice 70 (TD)
Comment faut-il faire pour envoyer un signal SIGINT à un processus ?

Exercice 71 (TD/TP)
Écrire trois programmes permettant d'essayer toutes les possibilités de gestion d'un signal et constater qu'on peut :

1. gérer directement le signal, en affichant un texte dans le gestionnaire ;
2. ignorer l'occurence du signal,
3. traiter une fois le signal, puis revenir à la situation *par défaut*.

9.1 Configuration du terminal

En TP, après avoir fait ces exercices, modifier la séquence de caractères du terminal qui génère ce signal (souvent `^C`) et essayer une nouvelle séquence (utiliser la commande `stty intr ^G` par exemple). La nouvelle chaîne devient celle d'interruption, et `^C` perd toute signification spéciale. Faire `stty -a` pour afficher toutes les caractéristiques du terminal.

9.2 Signal SIGALRM

La primitive définie par :

```
#include<unistd.h>
unsigned int alarm(unsigned int sec);
```

permet à un processus de demander au système de lui envoyer le signal `SIGALRM` après un délai de `sec` secondes environ suivant l'exécution de l'appel. Pendant ce délai, le processus continue son exécution et un appel à la primitive avec `sec` égal à zéro, annule la demande antérieure. Le comportement par défaut d'un processus recevant le signal `SIGALRM` est la terminaison.

Utiliser cette primitive pour réaliser un temporisateur (timeout). Le programme pose une question à l'utilisateur et réalise un traitement particulier si ce dernier n'a pas répondu assez vite.

Programmer les différentes solutions suivantes :

Exercice 72 (TD/TP)

1. le programme s'interrompt automatiquement si l'utilisateur n'a pas répondu dans les 10 secondes en affichant un message "Trop tard !".
2. Le programme demande à l'utilisateur de se presser si ce dernier n'a pas répondu dans les 10 secondes, en lui donnant une nouvelle chance.
3. Au cours de l'attente le programme envoie un message de plus en plus insistant à l'utilisateur (par intervalles raccourcis : 10, 5, puis 3 secondes), puis finit par s'interrompre si ce dernier n'a toujours pas répondu.

Exercice 73 (TD)
Expliquer pourquoi le temps en `sec` passé en paramètre à la primitive ne peut être qu'approximatif.

9.3 Erreurs système

Tous les appels systèmes, en cas d'echec, modifient une variable appartenant au processus, appelée `errno`. Il ne faut pas confondre cette variable avec la valeur de retour de l'appel. En effet, lorsque l'appel échoue, la valeur de retour est souvent -1, et alors `errno` contient une valeur qui précise l'erreur.

On peut afficher le texte correspondant à cette erreur par la fonction `strerror()`. La fonction `perror()` permet aussi d'obtenir un résultat similaire.

Dans le manuel des appels système, on peut (enfin) distinguer les deux notions de *valeur renvoyée* et *erreurs*. Ces erreurs sont des constantes qui comparées à la valeur de `errno` permettent d'obtenir une précision sur l'erreur qui s'est produite.

Exercice 74 (TD/TP)
Ecrire un programme générant une erreur de segmentation. Gérer le signal correspondant en effectuant l'affichage complet de l'erreur produite. Qu'en déduisez-vous ?

9.4 Signal de fin d'un processus fils

Lorsqu'un processus se termine sous *Unix*, il envoie à son père le signal `SIGCHLD`. Ce signal a quelques particularités (voir dans le manuel ce qui se passe lors de l'appel à `exit(int)`). Mais il peut être géré par le processus qui le reçoit, comme tout autre signal gérable. On veut étudier ici plusieurs cas correspondant au traitement de ce signal.

On envisage un programme à réaliser en TP, dont le processus correspondant répond aux caractéristiques suivantes :

— il génère successivement deux processus enfants,
— il utilise son propre gestionnaire de signaux, afin de récupérer le signal `SIGCHLD`,
— puis il attend la réception des deux signaux, correspondant à la fin respective de chaque enfant.

On constate que des processus ayant ces caractéristiques reçoivent parfois les deux signaux respectifs à la fin de chaque enfant, parfois un seul de ces signaux alors que les deux enfants sont défunts et parfois pas un seul de ces signaux.

Exercice 75 (TD/TP)
Commenter chacun de ces cas et expliquer comment chaque situation décrite peut se produire.

Exercice 76 (projet)
Afin de permettre de déterminer facilement les bogues à l'exécution, on souhaite capturer certains signaux (comme le très célèbre SIGSEGV) afin d'afficher l'état de la pile au moment du crash. On saura ainsi quelle est la fonction qui a généré le signal et qui est donc fautive. On utilisera pour ce faire la fonction déclarée dans `ucontext.h` :

```
int printstack(int fd);
```

Cette dernière affiche dans un fichier de descripteur fd, la liste courante des adresses de retour dans la pile. Dans l'exemple donné ci-après, on voit ainsi que la fonction g a été appelée par la fonction qui elle même avait été appelée par le main. On pourra ainsi chercher l'erreur plus facilement dans la fonction g().

```
/auto/meynard/C/segfault'g+0x12 [0x8050d5b]
/auto/meynard/C/segfault'f+0xb [0x8050d75]
/auto/meynard/C/segfault'main+0xb [0x8050d82]
/auto/meynard/C/segfault'_start+0x83 [0x8050bc3]
```

Le code devra être écrit dans une fonction `debug(int *signaux)` qui permettra l'affichage de la pile si un des signaux passés en paramètres survient. En production, bien entendu, l'appel à debug sera commenté !

10 Tubes anonymes et nommés

10.1 Taille d'un tube

On veut connaître la taille mémoire réservée à un tube simple (généré par l'appel système `pipe()`).

Exercice 77 (TD/TP)
Quelle méthode envisagez-vous pour connaître cette taille ? Écrire le programme correspondant à la méthode préconisée pour déterminer cette capacité.

10.2 Tube et fin de fichier

Deux processus communiquent par un tube. Le processus écrivain envoie un nombre fini n de caractères dans le tube. Le lecteur lit les caractères un à un sans s'arrêter.

On envisage successivement les cas suivants :

A L'écrivain ne se termine pas (boucle sans fin, mais sans rien écrire dans le tube) et oublie de fermer le descripteur en écriture qu'il possède.

B L'écrivain se termine après 20 secondes de temporisation (sleep) à la fin de ses écritures dans le tube.

C L'écrivain fait une temporisation de 20 secondes, refait une écriture et ferme le tube.

Exercice 78 (TD)
Étudier ces **trois** cas dans **chacune** des questions suivantes :

1. On suppose que chacun ferme les descripteurs dont il n'a pas besoin. Analyser ce qui se passe selon que le lecteur connaît on non le nombre de caractères à lire.

2. Aucun des processus ne ferme les descripteurs.

Exercice 79 (TD)
Supposons un processus écrivain ayant écrit n caractères dans un tube. Supposons un processus lisant dans ce tube caractère par caractère dans une boucle infinie, combien de caractères sont lus et en combien de d'appels à la primitive `read()` ?

10.3 Synchronisation

Un processus crée un tube simple puis se duplique. On envisage le scénario suivant : le parent sera le lecteur et le descendant l'écrivain.

Exercice 80 (TD)
1. Que se passe-t-il si le parent devient le processus actif du système, avant que le descendant n'ait écrit ?
2. Le descendant écrit par blocs de 20 caractères à la fois ; le parent veut lire par blocs de 30 caractères à la fois. Rappeler d'abord le fonctionnement des appels système `read()` et `write()`. Peut-on prévoir le nombre de fois où le lecteur sera bloqué ?
3. Illustrer une situation où le lecteur va rester bloqué et une autre où il ne le sera pas. Que se passe-t-il s'il y a moins de 30 caractères disponibles dans le tube ? Que se passe-t-il s'il y a moins de 30 caractères disponibles et que tous les descripteurs en écriture sont fermés ?

10.4 Version parallèle du crible d'Eratosthène [1]

Le but du crible d'Eratostène est de, partant de l'ensemble des nombres entiers, le filtrer successivement pour ne garder que ceux qui sont premiers. Considérons les entiers à partir de 2, qui est le premier nombre premier. Pour construire le reste des nombres premiers, commençons par ôter les multiples de 2 du reste des entiers, on obtient une liste d'entiers qui commence par 3 qui est le nombre premier suivant. Éliminons maintenant les multiples de 3 de cette liste, ce qui construit une liste d'entiers commençant par 5, et ainsi de suite. Autrement dit, nous énumérons les nombres premiers par application successive de la méthode de crible suivante : Pour cribler une liste d'entiers dont le premier est premier, nous ôtons de la liste le premier élément (qui est affiché comme premier) et nous supprimons de la liste l'ensemble des multiples de ce nombre. La liste résultante sera à son tour criblée selon la même méthode, et ainsi de suite.

Nous allons étudier une version parallèle sous UNIX de ce crible où chaque crible est un processus qui lit une liste d'entiers dans un tube, qui affiche le premier p, et envoie les entiers qu'il n'a pas filtrés dans un autre tube. On aura ainsi une chaîne de processus communiquant par des tubes. On envoie la liste des entiers dans le tube initial et chaque processus `crible(i)` affiche à l'écran le premier entier qu'il reçoit, puis parmi les entiers qu'il reçoit par la suite, il élimine les multiples du premier entier reçu et écrit dans le tube suivant les autres entiers. Nous présentons les détails dans l'algorithme 1 page 20.

Exercice 81 (TD/TP)
1. Faire un schéma de fonctionnement qui illustre la génération des processus affichant les nombres premiers jusqu'à 17.
2. Que doit faire le `main()` de ce programme et comment un entier peut-il être écrit dans un tube ?
3. Ecrire le programme C correspondant et le tester avec 1500.
4. Quelle différence existe-t-il selon que chaque processus attend son fils ou non (wait) ?
5. Ecrire une version différente `criblexec.c` ou chaque processus fils exécute (recouvrement) le programme `fcrible.c` qui lit la suite des entiers sur son entrée standard.

10.5 Comptage de caractères

Certains algorithme de compression (Huffman) nécessitent de connaître le nombre d'apparition de chaque caractère présent dans un fichier. Par exemple, Si le fichier `toto.txt` possède le contenu suivant :

```
Le corbeau et le renard
Maître corbeau
```

Alors, le programme qu'on cherche à développer devra afficher ce qui suit :

```
compte toto.txt
L:1  e:7   :5  c:2  o:2  r:5  b:2  a:4  u:2  t:2  l:1  n:1  d:1
:1  M:1  î:1
```

1. Mathématicien et philosophe de l'école d'alexandrie (275-194 av. J.C.)

Algorithme 1 : Fonction crible(in)

```
Données : in : tube entrant
début
    fermer(in[ecrire]) ;
    entier P;
    si 0 != (P=lire(in[lire])) alors
        /* P est premier */;
        afficher P ;
        out=créerPipe();
        f=fork();
        si f==0 alors
            /* fils */;
            fermer(in[lire]) ;
            return crible(out);
        sinon
            si f>0 alors
                /* père */;
                fermer(out[lire]);
                tant que i=lire(in[lire]) faire
                    si i modulo P != 0 alors
                        ecrire(i, out[ecrire]);
                fermer(in[lire]) ;
                fermer(out[ecrire]);
            sinon
                afficher "Erreur du fork()";
```

En effet, ce fichier contient 7 lettres “e”, 4 “a”, ... Le codage du fichier est un codage où chaque caractère est codé sur 1 octet (ISO-Latin1).

On veut écrire une version du programme `compte` basé sur le principe suivant : un processus est créé pour chaque caractère distinct et celui-ci compte le nombre de ce caractère et transmet les autres au processus suivant grâce à un tube qu'il aura créé. Cette version reprend le principe de l'exercice du crible d'Erathostène parallèle.

Exercice 82 (TD/TP)

1. Pourquoi la deuxième ligne de l'affichage est décalé d'un cran ?
2. Ecrire l'algorithme réalisant ce comptage.
3. Ecrire le programme C `compte.c` et testez le sur le fichier source ;

10.6 Dialogue entre deux utilisateurs – tubes nommés

On veut réaliser un schéma de communication avec des tubes *nommés*.

Deux utilisateurs U_1 et U_2 veulent utiliser des tubes nommés comme support de discussion personnelle. Ils ont chacun deux fenêtres f_{exp} et f_{recp} ouvertes sur la même machine, qu'ils vont dédier à cette discussion. L'objectif est que chaque utilisateur écrive dans la fenêtre f_{exp} ce qu'il veut dire à l'autre, la réception se faisant dans la fenêtre f_{recp}. Chacun va donc lancer un processus d'écriture dans la fenêtre f_{exp} et un processus de lecture dans f_{recp}. Noter qu'on ne veut pas de synchronisation entre les utilisateurs : chacun peut à la fois lire et écrire, de sorte que chacun peut donc taper ce qu'il veut envoyer, tout en constatant qu'il reçoit du texte.

On rappelle que chaque ligne tapée sera envoyée dès la frappe de la touche *Entrée*.

Exercice 83 (TD/TP)

1. Pourrait-on n'utiliser qu'un seul tube nommé pour réaliser cette communication ?
2. Au départ, aucun inode (ou i-nœud) représentant ces tubes n'existe mais les utilisateurs sont d'accord sur les noms des tubes nécessaires : par exemple, `tube1-2` et `tube2-1`. Quel(s) processus de quel utilisateur doivent créer chacun de ces tubes ?
3. Quelle solution peut-on proposer pour fermer l'ensemble des communications en restituant parfaitement l'état initial, avant le dialogue ?
4. Que se passe-t-il si un utilisateur envoie le signal de destruction (`SIGKILL`, celui numéroté 9) à son processus lecteur ?

5. Ecrire les programmes lecteur.c et ecrivain.c puis tester les différentes situations.
6. Ecrire le programme `chat.c` qui prend 2 arguments : `nomTubeLect`, `nomTubEcrit`. Ce processus crèe deux processus enfants qui exécuteront, l'un `ecrivain` sur `nomTubEcrit` tandis que l'autre exécutera `lecteur` sur `nomTubeLect`. Testez la communication dans 2 Terminaux graphiques côte à côte qui exécuteront `chat tdg tgd` pour le terminal de gauche (tgd signifiant tube de gauche vers droit) et `chat tgd tdg` pour le terminal de droite. Examinez les différents cas d'arrêt !

en TP : Réaliser cette application et se donner les moyens de vérifier tous les éléments de synchronisation pendant le fonctionnement et lors de la terminaison des processus.

11 Threads

11.1 Crible d'Eratosthène bis

L'objectif est de reprendre l'implémentation du crible d'Eratosthène parallèle étudié dans le Td/Tp sur les tubes en utilisant des `pthreads` à la place des processus lourds (`fork`).

Exercice 84

1. En utilisant le même type de synchronisation (le processus créateur attend (`wait/join`) le processus créé, implémenter le crible d'Eratosthène parallèle en utilisant la bibliothèque `pthread`.
2. A l'aide de `time` ou de `/usr/bin/time`, comparer les temps d'éxécution des deux versions (`fork` versus `pthread`) appelées avec un argument 10000.

11.2 File d'attente partagée

Exercice 85
Terminer l'implémentation de la file d'attente partagée vue en cours.

11.3 Projet

On souhaite développer un compresseur/décompresseur de répertoire dans lequel des compressions de fichiers puissent être effectuées en parallèle grâce à des threads. Tandis que le thread main() effectuera le parcours récursif du répertoire, chaque fichier rencontré donnera lieu à la création d'un thread (dans la limite d'un nombre de threads fourni en argument) qui effectuera la compression grâce au programme externe `gzip`. Une fois le fichier compressé créé dans un répertoire temporaire, ce dernier devra être concaténé dans l'archive du répertoire. Cette archive est une ressource critique où chaque thread écrira :

— le nom complètement qualifié (i.e. `./Syst/Td01/cours.tex`) et la taille de ce nom,
— la taille du fichier compressé,
— le contenu du fichier compressé.

On s'affranchira des problèmes de droits, propriétaires, et autres attributs des fichiers.

Exercice 86
Faut-il archiver les fichiers de type répertoires ? Si oui quel thread doit le faire et que sauvegarder ?

Exercice 87
Que faire des liens durs et symboliques ?

Exercice 88
Ecrire l'algo. d'un thread compresseur.