

Solutions des exercices du TD/TP 1

- Solution 1**
1. Un entier, un pointeur, un flottant est codé sur 4 octets, un char est codé sur un octet, une chaîne C est un pointeur. Un tableau a comme taille le produit de sa longueur par la taille d'un élément. Un tableau dynamique n'est qu'un pointeur ! K est une macro traitée par le préprocesseur, pas une variable C.
 2. Les variables locales ne sont pas initialisées, tandis que les variables globales et static le sont à 0.
 3. — les variables locales l, s, td sont dans la pile 0x22... ;
 - les variables globales KBIS, tab, f sont dans le segment de données statiques 0x40... ;
 - la variable statique j est dans le segment de données statiques 0x40... ;
 - s[0] pointe sur "un" qui est dans le segment de données statiques ; (littéral chaîne connu à la compilation)
 - tab[1] pointe sur 0 qui est dans le segment de données statiques ;
 - td[0] pointe sur 0 qui est dans le segment de données dynamiques (tas) en 0x66... ;
 4. La durée de vie des objets locaux est égale à la durée de vie de l'appel de fonction. La durée de vie des objets dynamiques dépend explicitement du programmeur : jusqu'à free(td). La durée de vie des objets statiques est égale à la durée de vie du processus.

Solution 2

```
1. #define N1 100
int main(){
    int t[N1]; // non initialisées
2.    int N2;
    printf("Entrez un entier positif SVP :");
    scanf("%d",&N2);
    int t2[N2]; // non initialisées
3.    int N3;
    int *t3;
    printf("Entrez un entier positif SVP :");
    scanf("%d",&N3);
    t3=malloc(N3*sizeof(int)); // non initialisées
4.    int** mat;
    mat=malloc(N2*sizeof(int*));
    if(!mat) exit(1); /* plus d'espace */
    for(int i=0;i<N2;i++){
        mat[i]=malloc(N3*sizeof(int));
        if(!mat[i]) exit(1); /* plus d'espace */
    }
    for(int i=0;i<N2;i++){
        for(int j=0;j<N3;j++){
            mat[i][j]=i*N3+j;
            printf("%d ",mat[i][j]);
        }
    }
    for(int i=0;i<N2;i++){ /* désallocation */
        free(mat[i]);
    }
    free(mat);
    mat=NULL;
5. Il faut retourner le pointeur sur la zone allouée dans la fonction.
6.    int matpile[N2][N3];
    for(int i=0;i<N2;i++){
        for(int j=0;j<N3;j++){
            matpile[i][j]=i*N3+j;
            printf("%d ",matpile[i][j]);
        }
    }
```

La durée de vie de l'objet matpile est égale à la durée de vie du bloc où il est défini. Il n'est pas visible en dehors.

Solution 3 Dans un système mono-tâche (ou mono-programmation), l'unité centrale est mobilisée à 100% constamment par ce processus qui boucle. Il ne peut gêner que l'utilisateur qui l'a lancé. Il continue l'exécution tant qu'on le laisse faire.

Dans un système multi-tâche (ou multi-programmation), le processus correspondant obtient un quantum de temps et il l'utilise entièrement. C'est gênant pour tous les autres utilisateurs (et processus) car il utilise la ressource CPU et

empêche les autres de prendre la main. Mais ça ne les bloque pas. Ils sont ralentis. De façon systématique ce processus reprendra la main et épuisera un quantum de temps à chaque reprise.

Pour l'arrêter, une interruption est nécessaire : par exemple Ctrl C ou Ctrl \ dans un système multi-tâche (cf. Unix) ou kill -9 1234 ou 1234 est le pid du processus infernal.

Solution 4 1. il est le seul processus utilisateur,

2. tous les autres sont en attente d'entrée-sortie,
3. il est le plus prioritaire.

Pour l'enchaînement des opérations, voir le cours.

Solution 5 Les réponses ne sont pas détaillées, dans chaque cas, il faut se demander ce qui se passerait si tout utilisateur pouvait exécuter à sa guise les instructions ou codes correspondants. Attention quand même : pour ce qui concerne la date du jour ou l'allocation mémoire, ce ne sont pas des instructions simples, mais des programmes ou des

masquer une interruption	oui	lire la date du jour	non
fonctions.			
modifier la date du jour	oui	modifier l'allocation mémoire	oui

appeler l'ordonnanceur oui appel d'une fonction en C non

Solution 6 1. afficher "Nb de paramètres (arguments) : " argc "\n"

```
Pour i=0 à argc-1
    afficher i ":" argv[i] "\n"
afficher "\nVariables d'environnement :\n"
i=0
Tq arge[i] != NULL
    afficher i ":" arge[i] "\n"
    i++
afficher "Nb de variables d'environnement :" i "\n"
```

2. #include <stdio.h>

```
int main(int argc, char** argv, char** arge){
    printf("Nb de paramètres (arguments) : %d\n", argc);
    for(int i=0; i<argc; i++){
        printf("%d : %s\n", i, argv[i]);
    }
    printf("\nVariables d'environnement :\n");
    int i=0;
    while(arge[i] != NULL){
        printf("%d : %s\n", i, arge[i]);
        i++;
    }
    printf("Nb de variables d'environnement : %d\n", i);
}
```

Solution 7 Toutes les substitutions sont faites par le shell avant de lancer l'exécution. Donc on aura successivement :

- 4 paramètres ;
- nombre des fichiers du répertoire, sauf ceux qui commencent par . (point) + 1 (\$PATH).
- nombre des fichiers du répertoire d'accueil qui commencent par . et qui possèdent au moins 2 lettres supplémentaires (ni . lui-même ni ..)

Solution 9 1. i=0

```
TROUVE=faux
Tq arge[i] != NULL et non TROUVE
    TROUVE=chercherDebut("PATH=", arge[i]);
    i++
    si non TROUVE
        afficher "Pas de PATH dans les variables d'env !"
        exit
    sinon
        TROUVE=faux
        i-
        TROUVE=rechercher(getcwd() ou ".", arge[i]+taille("PATH"))
        if TROUVE
            afficher "Ce rép. est exécutable!"
        sinon
            afficher "Ce rép. n'est pas exécutable!"
```

```

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

int main(int argc, char** argv, char** arge){
    char * motif="PATH=", *delim=":", *trouve=NULL; // UNIX delim=":"
    int i = 0 ;
    while(arge[i] != NULL && arge[i]!=(trouve=strstr(arge[i],motif)))
        i++;
    if (arge[i]==NULL){
        printf("La chaîne %s n'existe pas !\n",motif);
        return 1;
    } else {
        char * cwd=(char *)malloc(1000);
        getcwd(cwd,1000); // répertoire courant

        char * source=arge[i]+strlen(motif);
        char * tok=strtok(source,delim); // appel initial avec la source
        while(tok && strcmp(tok,cwd) && strcmp(tok,".")){
            tok=strtok(NULL,delim);
        }
        if (tok)
            printf("Ce rép. %s est exécutable grâce à : %s\n", cwd, tok);
        else
            printf("Ce rép. %s n'est pas exécutable !\n", cwd);
    }
}

```

Solution 10

```

1. #include <stdio.h>
#include <stdlib.h>
int fact(int n){
    if (n<=1)
        return 1;
    else
        return n*fact(n-1);
}
int main(int argc, char** argv, char** arge){
    if(argc!=2){
        printf("Syntaxe incorrecte : %s %d\n", argv[0]);
        return 1;
    } else {
        int n=atoi(argv[1]);
        printf("%d ! = %d\n",n,fact(n));
        return 0;
    }
}
2. fact(3)
    fact(2)
        fact(1)
        ret 1;
    ret 2*1
    ret 3*2=6

```

Solution 11 1. On choisit un tableau à 2 dimensions dynamique où la largeur de chaque ligne est égale à son indice +1.

```

2. triangle(n)
tab=malloc(n+1 entiers)
Pour i=0 à n
    tab[i]=malloc(i+1 entiers)
    tab[i][0]=1
    tab[i][i]=1
    pour j=1 à i-1
        tab[i][j]=tab[i-1][j-1]+tab[i-1][j]

```

```

3. #include <stdio.h>
#include <stdlib.h>
int** triangle(int n){ /* ret un triangle de pascal */
    int** tab=malloc((n+1)*sizeof(int));
    for(int i=0; i<=n; i++){
        tab[i]=malloc((i+1)*sizeof(int));
        tab[i][0]=1;
        tab[i][i]=1;
        for(int j=1; j<i; j++){
            tab[i][j]=tab[i-1][j-1]+tab[i-1][j];
        }
    }
    return tab;
}
void afficherTriangle(int** tab, int n){
    for(int i=0; i<=n; i++){
        for(int j=0; j<=i; j++){
            printf("%d ", tab[i][j]);
        }
        printf("\n");
    }
}
int main(int argc, char** argv, char** arge){
    if(argc!=3){
        printf("Syntaxe incorrecte : %s %d %d\n", argv[0]);
        return 1;
    } else {
        int n=atoi(argv[1]);
        int p=atoi(argv[2]);
        int **tab=triangle(n);
        afficherTriangle(tab,n);
        printf("Nombre de combinaisons C(%d,%d) = %d\n", n,p,tab[n][p]);
        return 0;
    }
}

```

Solution 12 En liaison dynamique, la taille minimale d'un exécutable est petite : il faut juste pouvoir retourner au système après exécution afin de nettoyer les segments utilisés. Les données statiques non initialisées prennent moins de place car l'espace sera alloué au chargement du programme. Idem pour la pile et le tas alloués au chargement.

Solution 13

1. La deuxième fonction affiche les mêmes valeurs que la première. En effet, le second tableau est alloué au même endroit que le premier, par conséquent, on voit encore les anciennes valeurs !
2. Il faut initialiser soit même les objets automatiques (pile) et dynamiques (tas) sous peine de grosses déconvenues.

Solution 14 idem exercice précédent

Solutions des exercices du TD/TP 2

Solution 1 Traitement par le préprocesseur

Cette étape consiste à traiter toutes les lignes commençant par le caractère # . Par exemple, `#include <jeleveux.h>` engendrera la recherche du fichier `jeleveux.h` dans un certain nombre de répertoires (par défaut, liste pouvant être complétée sur la ligne de commande) et s'il est trouvé, son contenu sera ajouté directement à l'endroit de la ligne, en tant que **source**. Noter qu'il y a beaucoup de commandes possibles au préprocesseur (`#define #if #ifdef ...`). Le résultat du préprocesseur est toujours du langage **source**. On peut exécuter le préprocesseur seul par la commande `cpp` ou par `gcc -E nomprog.c` . Attention : par défaut, la sortie est sur `stdout`.

Compilation proprement dite

Elle est souvent vue en deux étapes distinctes :

- analyse et transcription en langage d'assemblage (indépendant de la machine) du code reçu du préprocesseur ; on peut s'arrêter à cette étape par la commande `gcc -S nomprog.c` . Le résultat a le suffixe `.s` par défaut. On peut le générer et le lire pour ceux qui veulent voir de l'assembleur.
- assemblage qui traduit le code précédent en *code objet* ; c'est «presque» du langage machine, mais il lui manque toutes les références externes (voir la remarque ci-après). On peut s'arrêter à cette étape par la commande `gcc -c nomprog.c` ou `nomprog.s` si on l'a générée avant.

Remarque importante : il faut noter que jusque là, aucun autre fichier (autre que ceux inclus par `#include ...`) n'a été pris en compte. Si une fonction `viensvoir()` en cours de compilation appelle une fonction `lespectacle()` qui est définie dans un autre fichier source, tout ce que le compilateur a besoin de connaître est la signature de `lespectacle()` (et encore, pour vérifier la cohérence ; en C de base, ce n'est même pas nécessaire, mais attention à la cohérence).

Édition de liens

C'est la phase de génération de l'exécutable. C'est ici que l'on prend en compte **toutes** les références externes, c'est-à-dire **tous** les appels à des fonctions définies et déjà compilées précédemment. Il s'agit aussi bien de fonctions systèmes (les appels systèmes comme `read()`, `write()`, `exit()`, `getpid()` etc, cf. section 2 du manuel), que les fonctions de l'utilisateur, compilées dans un autre fichier source (compilation séparée), comme `lespectacle()` dans l'exemple précédent.

Donc c'est ici et seulement ici qu'on aura une erreur si `lespectacle.o` est introuvable (toutes erreurs du programmeur imaginables : le nom, la non compilation, ...).

Solution 2 1. La consultation de la bibliothèque `libcrypt.a` intervient uniquement à l'édition de liens. On utilisera `libcrypt.so` en cas d'édition de liens dynamique (par défaut) ou bien `libcrypt.a` en cas d'édition de liens statique.

2. `gcc -o jeu jeu.c alea.c es.c -static -lcrypt`
3. `gcc -o jeu jeu.c alea.c es.c -lcrypt`

4. L'inconvénient est que la moindre modification, même d'un seul source provoque la recompilation de l'ensemble. Gênant surtout si ces sources sont gros et longs à compiler. Gênant aussi, car avec un bon Makefile on peut gérer tout ça plus facilement.

D'une façon générale, l'option `-lxxx` suppose qu'il existe une bibliothèque appelée `libxxx.a` dans un des répertoires scrutés par l'éditeur de liens (`LIBPATH`).

Noter enfin qu'on peut construire soi-même une telle bibliothèque, qui n'est rien d'autre qu'un ensemble de `.o`. L'utilitaire `ar` permet de gérer les bibliothèques statiques (ajout, suppression de modules objet). On pourra essayer par exemple la commande `ar -t libcrypt.a` dans le répertoire `/usr/lib` (en principe), pour connaître l'ensemble des `.o` inclus dans cette bibliothèque.

Solution 3 makefile

```
CC=gcc
CFLAGS=-g -ansi -Wall -std=c99
OBJ = jeu.o es.o alea.o

#compilation avec ed. liens dynamique par défaut
#
jeu : $(OBJ)
$(CC) $(CFLAGS) -o jeu $(OBJ) -lcrypt

#compilation avec ed. liens statique
#
jeu.st : $(OBJ)
```

```

$(CC) $(CFLAGS) -o jeu $(OBJ) -static -lcrypt

jeu.o : jeu.c alea.h es.h
$(CC) $(CFLAGS) -c jeu.c

es.o : es.c es.h
$(CC) $(CFLAGS) -c es.c

alea.o : alea.c alea.h
$(CC) $(CFLAGS) -c alea.c

```

Solution 4 expliquer les principes de base de `make`, c'est-à-dire les dépendances entre objets et les règles permettant de (re)générer ces objets. Insister sur le fait que seule la date machine permet de savoir si un objet est postérieur ou antérieur à un autre. D'où l'importance de maintenir à jour la date et la même date sur toutes les machines d'un réseau sur lequel on partage des fichiers.

On peut aussi noter que `make -n` permet de savoir ce qui va être fait si on lance `make` (i.e. `make -n` ne fait rien d'autre que scruter les dépendances).

On exécute la commande `touch alea.c` : cette commande ne fait que changer la date du fichier en la mettant à la date courante. Donc tout se passe comme si on venait de modifier quelque chose dans le fichier. Ici, lors du `make` prochain : reconstruction de `alea.o` puis de `jeu`.

Solution 5

1. Avertissement à la compilation car les signatures de `entnb()` et `sortienb()` sont inconnues. Par défaut, elles sont supposées avoir une déclaration implicite de `extern int f()`.
2. erreur du préprocesseur : `es.h` inconnu (cherché dans les répertoires système `/usr/include` et voisins dans le cas de C).

Solution 6

1. `./mmgui&; ./mm`
2. `touch mm.o; make mm`
3. `[mm.c]`

```

#include "mm.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

mm mm_creer(){
    mm jeu=(mm) malloc(sizeof(mm));
    jeu->secret[0]='\0'; // INIT À CHAINE VIDE
    //init(jeu->secret,TAILLE);
    for(int i=0;i<TAILLE;i++){ // indice du char
        char ch;
        srand((int)time(NULL)); //init générateur aléatoire
        do{
            ch = '0'+(int)((double)10 * (rand() / (RAND_MAX + 1.0)));
            //gen chiffre (0-9)
        } while(strchr(jeu->secret,ch));
        jeu->secret[i]=ch;
        jeu->secret[i+1]='\0';
    }
    jeu->nbeffais=0;
    return jeu;
}
void mm_detruire(mm jeu){
    free(jeu);
}

int mm_test(mm jeu, char* essai){
    jeu->nbeffais++;
    if(strlen(essai)!=TAILLE)
        return -1;
    int nbBien=0; // nb lettres bien placées
    int nbMal=0; // nb lettres mal placées
    char *res;

```

```
for(int i=0;i<TAILLE;i++){ // indice du char
    if (NULL!=(res=strchr(jeu->secret,essai[i]))){
        if(res==jeu->secret+i)
            nbBien++;
        else
            nbMal++;
    }
}
return nbBien*(TAILLE+1)+nbMal;
}

int mm_nbessais(mm jeu){
    return jeu->nbessais;
}
```

Solutions des exercices du TD/TP 3

Solution 1 Au lancement de tout processus, il y a 3 fichiers ouverts, numérotés 0,1 et 2, représentant respectivement *stdin*, i.e. l'entrée standard, *stdout* i.e. la sortie standard et *stderr*, i.e. l'erreur standard. Au lancement, les trois sont affectés au «terminal» (i.e. fenêtre dans laquelle a été lancé le processus), sauf s'ils sont redirigés.

Concernant les redirections, on peut lancer un processus en mettant par exemple :

```
monprog <metro >boulot 2>dodo
```

<**metro**: les lectures faites sur l'entrée standard se feront en fait sur le fichier **metro**,
>**boulot**: les écritures sur la sortie standard se font sur le fichier **boulot**) et
2>**dodo**: les écritures sur l'erreur standard se font sur le fichier **dodo**; (syntaxe bash)

Solution 2 édition, copie, commande **cat >fichu** (copie du clavier vers le fichier avec ctrl-D qui sert de fin de fichier), commande **touch** (c'est un effet de bord de cette commande)

Solution 3 Cela dépend de la destination du fichier. Un fichier *texte* est lisible «humainement» et contient principalement des caractères ASCII compris entre 20_{16} et $7F_{16}$ (sauf en ISO-Latin-1 ou en URF-8!). Un fichier *binnaire* contient des données codées selon une forme non lisible «humainement». Exemples : un exécutable, mais aussi un fichier de données contenant des entiers codés int. Il faut distinguer selon le codage un entier lisible (format chaîne de caractères) et non lisible (format interne en Complément à 2).

Solution 4 — open, close

- read, write
- lseek
- pas d'appel mais vrai lorsque un read ne lit pas le nombre de caractères demandés;

Solution 5 1. Un tableau de 256 entiers (booléens) avec 1 case par caractère (0 absent, 1 présent). Un compteur entier.

2. Algorithme

```
f=ouvrir(argv[1])
si (f== -1)
    ret 1
int present[256] initialisé à 0
int compte=0
char c
Tantque (0<lire(f,c))
    Si non present[c]
        present[c]=1
        compte++
fermer(f)
afficher compte "caractères différents : "
Pour c=0 à 255
    Si present[c]
        afficher c ", "
```

3. compte.c

```
#include <stdio.h>
#include <sys/types.h>/ * open */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>/ * read */

int main(int argc, char** argv, char** arge){
    if(argc!=2){
        printf("Syntaxe incorrecte : %s fichier.txt\n", argv[0]);
        return 1;
    }
    int f=open(argv[1],O_RDONLY);
    if (f<0){
        printf("Impossible d'ouvrir %s !\n", argv[1]);
        return 2;
    }
    int present[256];
```

```

for(int i=0;i<256;i++) /* pas de char i sinon boucle ! */
    present[i]=0;
int compte=0;
char c;
while(0<read(f,&c,1)){
    if(!present[(int)c]){
        present[(int)c]=1;
        compte++;
    }
}
close(f);
printf("%d caractères différents : ", compte);
for(int i=0;i<256;i++)
    if(present[i])
        printf("%c, ",i);
printf("\n");
}

```

Solution 6 1. Parce que le caractère affiché est un retour ligne.

2. Un tableau de 256 entiers avec 1 case par caractère comptant le nombre d'occurrences;

3. Algorithme

```

f=ouvrir(argv[1])
si (f== -1)
    ret 1
int occ[256] initialisé à 0
char c
Tantque (0< lire(f,c))
    occ[c]++
fermer(f)
Pour c=0 à 255
    Si occ[c]
        afficher c ":" occ[c] ", "

```

4. occurrences.c

```

#include <stdio.h>
#include <sys/types.h>/ * open */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>/ * read */

int main(int argc, char** argv, char** args){
    if(argc!=2){
        printf("Syntaxe incorrecte : %s fichier.txt\n", argv[0]);
        return 1;
    }
    int f=open(argv[1],O_RDONLY);
    if (f<0){
        printf("Impossible d'ouvrir %s !\n", argv[1]);
        return 2;
    }
    int occ[256];
    for(int i=0;i<256;i++) /* pas de char i sinon boucle ! */
        occ[i]=0;
    char c;
    while(0<read(f,&c,1)){
        occ[(int)c]++;
    }
    close(f);
    for(int i=0;i<256;i++)
        if(occ[i])
            printf("%c(0x%02x):%d, ", i,i,occ[i]);
    printf("\n");
}

```

```

solution 7
1. f=ouvrir(argv[1])
    si (f== -1)
        ret 1
    char c[16]
    int position=0
    int nb
    Tantque (0<nb=lire(f,c,16))
        afficherHexa position ": "
        position=position+16
        Pour i=0 à nb-1
            afficherHexa c[i]
            Si i%2
                afficher " "
            Si nb<16
                Pour i=nb à 15
                    afficher " "
                    Si i%2
                        afficher " "
                    afficher c "\n"
                fermer(f)

2. monhexl.c
#include <stdio.h>
#include <sys/types.h>/ * open */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>/ * read */

int main(int argc, char** argv, char** arge){
    if(argc!=2){
        printf("Syntaxe incorrecte : %s fichier.txt\n", argv[0]);
        return 1;
    }
    int f=open(argv[1],O_RDONLY);
    if (f<0){
        printf("Impossible d'ouvrir %s !\n", argv[1]);
        return 2;
    }
    char c[16];
    int position=0,nb;
    while(0<(nb=read(f,&c,16))){
        printf("%8.8x: ",position); /* position sur 8 car en hexa */
        position+=16;
        for(int i=0;i<nb;i++){
            char code=c[i];
            printf("%2.2hhx",code); // hh pour indiquer que c'est un char (pas int)
            if(i%2) /* tous les 2 car, afficher un espace */
                printf(" ");
        }
        if(nb<16){ /* pour la dernière ligne, compléter */
            for(int i=nb;i<16;i++){
                printf(" ");
            }
        }
        printf(" ");
        for(int i=0;i<nb;i++){ /* format caractères */
            if(c[i]>=0x20 && c[i]<0x7F) /* ASCII affichable */
                printf("%c",c[i]);
            else
                printf(".");
        }
    }
}

```

```

        printf("\n");
    }
    close(f);
}

```

Solution 8 1. Algorithme

```

source=ouvrir(argv[1], lecture)
si (source== -1)
    ret 1
dest=ouvrir(argv[1], écriture)
si (dest== -1)
    ret 2
Tantque (0<lire(source, c))
    écrire(dest, c)
fermer(dest)
fermer(source)

2. moncp.c

#include <stdio.h>
#include <sys/types.h> /* open */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h> /* read */

```

```

int main(int argc, char** argv, char** arge){
if(argc!=3){
    printf("Syntaxe incorrecte : %s source.txt destination.txt\n", argv[0]);
    return 1;
}
int source=open(argv[1],O_RDONLY);
if (source<0){
    printf("Impossible d'ouvrir %s !\n", argv[1]);
    return 2;
}
int dest=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0x644);
if (dest<0){
    printf("Impossible d'ouvrir %s !\n", argv[2]);
    return 3;
}
char c;
while(0<read(source,&c,1)){
    write(dest,&c,1);
}
close(dest);
close(source);
}

```

Solution 9 1. lseek retourne la position du pointeur, il suffit donc de positionner celui-ci en fin de fichier SEEK_END pour connaître sa taille.

2. Algorithme

```

f=ouvrir(argv[1], lecture)
si (f== -1)
    ret 1
char x=premier(argv[2])
int debut=0;
int fin=lseek(f,0,SEEK_END)-1
si fin== -1
    ret2
repeter
    int milieu=(debut+fin)/2
    lseek(f,milieu,SEEK_SET)      // se placer au milieu
    lire(f, c)
    si c>x

```

```

        fin=milieu-1
        sinon si c<x
            debut=milieu+1
        tantQue c!=x et debut<=fin
        si c==x
            afficher "Le caractère " c " est en position " milieu
        sinon
            afficher "Le caractère " c " est introuvable !"
        fermer(f)
    }

3. dicho.c

#include <stdio.h>
#include <sys/types.h>/ * open */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>/ * read */
#include <string.h>/ * strlen */

int main(int argc, char** argv, char** arge){
    if(argc!=3 || strlen(argv[2])!=1){
        printf("Syntaxe incorrecte : %s fic.txt c\n", argv[0]);
        return 1;
    }
    int f=open(argv[1],O_RDONLY);
    if (f<0){
        printf("Impossible d'ouvrir %s !\n", argv[1]);
        return 2;
    }
    char x=argv[2][0]; //printf("x=%c; ",x);
    int debut=0;
    int fin=lseek(f,0,SEEK_END)-1; //printf("fin:%d; ",fin);
    if (fin==-2){
        printf("Impossible de se déplacer dans %s !\n", argv[1]);
        return 3;
    }else if (fin==-1){
        printf("Fichier vide : %s !\n", argv[1]);
        return 4;
    }
    int milieu,i;
    char c;
    do {
        milieu=(debut+fin)/2;      //printf("milieu=%d; ",milieu);
        i=lseek(f,milieu,SEEK_SET);      // se placer au milieu
        if (i==-1){
            printf("Impossible de se déplacer en %d dans %s !\n",milieu, argv[1]);
            return 5;
        }
        i=read(f,&c,1); //printf("c=%c; ",c);
        if (i!=1){
            printf("Impossible de lire dans %s !\n", argv[1]);
            return 6;
        }
        if(c>x){
            fin=milieu-1;
        }else if(c<x){
            debut=milieu+1;
        }
    }while(c!=x && debut<=fin);
    close(f);
    if(c==x)
        printf("Le caractère %c est en position %d !\n",x, milieu);
    else
        printf("Le caractère %c est introuvable !\n",x);
}

```

Solution 10 — fopen, fclose

- fgetc (un char), fgets (une ligne), fputc, fputs
- fscanf (lecture formattée), fprintf (écriture formattée)
- fseek
- vrai lorsque le FILE* == EOF

Solution 11 1. Les appels systèmes sont plus rapides, moins gourmands en mémoire donc plus efficaces. Mais ils sont plus simples : pas de lecture ni d'écriture formattée ! De plus, ils sont moins portables : d'un système Unix à un autre, il peut y avoir quelques différences. Ces problèmes de portabilité sont de moins en moins nombreux avec la conformité au standards tels que POSIX.

Solutions des exercices du TD/TP 4

Solution 1 genprocl.c

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

int main(){
    pid_t pid;
    switch(pid = fork()){
        case -1:{      // echec du fork
            printf("Probleme : echec du fork") ;
            break ;
        }
        case 0:{      // c'est le descendant
            printf("du descendant : valeur de retour de fork() : %d\n", pid);
            printf("du descendant : je suis %d de parent %d \n", getpid(),getppid());
            return 4;
            break ;
        }
        default:{     // c'est le parent
            printf("du parent : valeur de retour de fork() : %d\n", pid);
            printf("du parent : je suis %d de parent %d \n",getpid(), getppid());
            int s=0; /* statut (?) */
            pid_t fils=waitpid(pid,&s,0) ; //pid_t fils=wait(&s); //possible
            printf("du parent : mort annoncée de : %d avec le statut %d\n", fils,
WEXITSTATUS(s)); // 8 bits de poids faibles de l'argument s (4)
            break ;
        }
    }
}
```

Solution 2 parent

```
f0
f01
f012
f0123
f013
f02
f023
f03
f1
f12
f123
f13
f2
f23
f3
```

Au total, 16 processus sont créés. Le dernier processus vivant est le parent racine car chaque parent attend la fin de ses enfants.

Solution 3 1. Le programme devra définir une variable statique, une variable dynamique, une variable automatique (locale), ouvrir un fichier (table des descripteurs propre au processus). Ensuite il se dupliquera (fork) et chacune des parties affichera les variables, les modifiera, les réaffichera. Pour le fichier, le père pourra fermer le fichier et le fils tentera de le lire ... (compteur de pus dans la table des fichiers ouverts). Noter qu'afficher l'adresse n'est pas une bonne solution, car les adresses sont relatives au début du processus, donc on aura la même adresse relative dans les deux processus pour une «même» donnée et on pourrait déduire à tort que c'est la même, sans duplication.

2. dupseg.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h> /* malloc */
```

```

#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int statique = 1;

int main(int argc, char ** argv){
    int locale = 2;
    int *pi=malloc(sizeof(int));
    *pi=3; /* Tas */
    if(argc!=2){
        printf("Syntaxe incorrecte : %s fic.txt\n", argv[0]);
        return 1;
    }
    int f=open(argv[1],O_RDONLY);
    if (f<0){
        printf("Impossible d'ouvrir %s !\n", argv[1]);
        return 2;
    }
    printf("Avant fork - Je suis : %d\n",getpid());
    printf("Avant fork - statique=%d; Adresse de statique=%p\n",statique,&statique);
    printf("Avant fork - locale=%d; Adresse de locale=%p\n",locale,&locale);
    printf("Avant fork - *pi=%d; Adresse de *pi=%p; Adresse de pi=%p\n",*pi,pi,&pi);
    printf("Avant fork - f=%d\n",f);
    char s[6];
    strcpy(s,"xxxxx");
    int i=read(f,s,5);
    if (i!=5){
        printf("Impossible de lire dans %s !\n", argv[1]);
        return 6;
    }
    printf("Avant fork - lecture de 5 octets : %s\n",s);

    pid_t Pid = fork();
    switch(Pid){
        case -1: // echec du fork
            printf("Probleme : echec du fork");
            return 3;
            break ;
        case 0: // c'est le fils
            printf("\nFils - Je suis : %d\n", getpid());
            printf("Fils - statique=%d; Adresse de statique=%p\n",statique,&statique);
            printf("Fils - locale=%d; Adresse de locale=%p\n",locale,&locale);
            printf("Fils - *pi=%d; Adresse de *pi=%p; Adresse de pi=%p\n",*pi,pi,&pi);
            printf("Fils - f=%d\n",f);
            char s[3];
            strcpy(s,"xx");
            int i=read(f,s,2);
            if (i!=2){
                printf("Impossible de lire dans %s !\n", argv[1]);
                return 6;
            }
            printf("Fils - lecture de 2 octets : %s\n",s);

            statique+=10;locale+=10; *pi+=10;

            printf("Fils après - statique=%d; Adresse de statique=%p\n",statique,&statique);
            printf("Fils après - locale=%d; Adresse de locale=%p\n",locale,&locale);
            printf("Fils après - *pi=%d; Adresse de *pi=%p; Adresse de pi=%p\n",*pi,pi,&pi);
            printf("Fils après - f=%d\n",f);
            sleep(6) ;
            break ;
    }
}

```

```

default: // c'est le pere
statique+=100;locale+=100;
(*pi)+=100;

printf("Père après - statique=%d; Adresse de statique=%p\n",statique,&statique);
printf("Père après - locale=%d; Adresse de locale=%p\n",locale,&locale);
printf("Père après - *pi=%d; Adresse de pi=%p; Adresse de pi=%p\n",*pi,pi,&pi);
printf("Père après - f=%d\n",f);
close(f);
break ;
}
}

```

3. chaque processus a en propre une table de descripteurs qui pointe sur la table des fichiers ouverts (qui elle est unique). Ils partagent donc la même tête de lecture/écriture et lisent ou écrivent à tour de rôle. Chaque processus peut fermer le fichier indépendamment de l'autre.

Solution 4 Constater que lors du lancement en tâche de fond, le processus est toujours descendant de son parent générateur, mais si ce parent disparaît, alors c'est «init» qui devient le parent. Utiliser ps -elf | grep nomProcessus pour s'en convaincre. En tâche directe, si on tue le parent, le descendant est tué aussi.

Pour la dernière question, s'inspirer des divers genproc*.c. Il suffit de faire à des moments différents l'appel getppid(), sans avoir à jouer avec ps cette fois-ci.

Solution 5 1. recouv1.c

```

/* Attention : chemin absolu pour l'exécutable! /bin/ls */
#include <stdio.h>
#include <unistd.h> // exec

int main(int argc, char ** argv){
    printf("je vais demander mon recouvrement par ls !\n");
    int i = execl("/bin/ls","ls",NULL) ;
    if (i<0)
        printf("Erreur de recouvrement !\n");
    else
        printf("Impossible !");
}

```

2. recouv2.c

```

/* Attention : chemin absolu pour l'exécutable! /bin/ls */
#include <stdio.h>
#include <unistd.h> // exec

int main(int argc, char ** argv){
    printf("je vais demander mon recouvrement par ls -l /bin !\n");
    int i = execl("/bin/ls","ls","-l","/bin",NULL) ;
    if (i<0)
        printf("Erreur de recouvrement !\n");
    else
        printf("Impossible !");
}

```

3. recouv3.c

```

/* Attention : chemin absolu pour l'exécutable! /bin/ls */
#include <stdio.h>
#include <stdlib.h> // malloc
#include <unistd.h> // exec
#include <string.h> // str..

int main(int argc, char ** argv){
    printf("je vais demander mon recouvrement par gcc moi-même !\n");
    char *s=malloc(strlen(argv[0])+3); // .c
}

```

```

strcat(strcpy(s,argv[0]),".c");
int i = execlp("gcc","gcc","-o",argv[0],s,NULL) ;
if (i<0)
    printf("Erreur de recouvrement !\n");
else
    printf("Impossible !");
}

```

Si on ne se souvient pas de la différence dans les divers appels de `exec()`, ici il suffit de se souvenir que `execl()` doit contenir en premier paramètre le chemin complet de l'exécutables (absolu ou relatif au répertoire dans lequel est lancé le processus), et `execlp()` n'a besoin que du nom de l'exécutables, car il va le chercher selon la suite des répertoires dans PATH. Les autres paramètres sont ceux qui sont envoyés au processus généré (revoir ligne `main()`). Rappelons toutes la familles des frontaux à `execve` qui le seul appel système!

```

int execve (const char *fichier, char * const argv [], char * const envp[]);

int execl (const char *path, const char *arg, ...);
int execlp (const char *file, const char *arg, ...);
int execle (const char *path, const char *arg , ..., char * const envp[]);
int execv (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);

```

Solution 6 Différences entre un appel `system()` et `exec()` :

`system()` lance un shell qui contrôle l'exécution de ce qui a été demandé. `exec()` est un recouvrement de son propre code, sans lancement de quoi que ce soit d'autre. `system()` va attendre la fin de l'exécution à contrôler puis rendre la main au processus l'ayant lancé. `exec()` va entraîner l'exécution du recouvrant. `system()` va donc créer deux nouveaux processus. `exec()` ne créera aucun nouveau.

Nom du processus avant et après recouvrement :

Le nom du processus est celui du lancement (premier paramètre de la commande de lancement) avant le recouvrement et après le recouvrement c'est celui du recouvrant. C'est-à-dire que le même processus change de nom. Dans tous les cas, le nom est celui pointé par `argv[0]`.

Solution 7 Principe : Un programme affiche son pid, puis se recouvre par un autre qui affiche son pid (il peut se recouvrir par lui-même, mais ça fait une boucle infinie et répond à la question 2...)

```

P1 : int main(){
    printf("avant recouvrement, pid = %d\n",getpid());
    execl('/home/moi/P2','P2',NULL);
P2 : int main(){
    printf("après recouvrement, pid = %d\n",getpid());

```

Solution 8 P2 se recouvre par P1 (ajouter une ligne `exec` dans P2, ou comme vu ci-dessus, P1 s'auto-recouvre. Dans tous les cas, boucle infinie de recouvrements, avec un seul processus, puisqu'il ne change pas d'identité).

Solution 9 Principe : faire une boucle de génération avec un des deux processus qui sort de la boucle. Pour sortir de la boucle, on peut faire *bestial* en faisant un `exit(?)`, ou en recouvrant le descendant par un programme qui s'arrête (fait quelque chose et ne génère pas de nouveaux processus).

Autre solution : ne faire le `fork()` que si on est le parent, les enfants ne faisant que passer dans la boucle. Voir `nbproc.c`.

Mauvaise solution qui a été faite par quelques-uns : utiliser l'exercice précédent avec la boucle, en mettant $\log(n)$ comme limite (avec des astuces compliquées pour prendre la partie entière et tout un tas d'explications pour constater que selon n ça marche ou non...).

Remarque : si vous pensez qu'il faut rajouter des exercices sur les processus, il y en a plein dans d'anciens exams en particulier. À vos remarques.

```

Solution 10 1. prompt="$"
Répéter
    afficher(prompt)
    s=lireClavier()
    if s=="exit"
        return 0
    i=fork()
    if i==0 // fils
        execlp(s,s,NULL)

```

```

else // père
    wait()
à l'infini
2. monsh.c

#include <stdio.h>
#include <stdlib.h> // malloc
#include <unistd.h> // exec
#include <string.h> // str..
#include <sys/types.h> //wait
#include <sys/wait.h>

int main(int argc, char ** argv){
    char *prompt="$";
    char s[1024];
    do{
        printf("%s", prompt);
        scanf("%s",s);
        if (!strcmp(s,"exit"))
            return 0;
        pid_t Pid;
        switch (Pid = fork()){
            case -1: // echec du fork
                printf("Probleme : echec du fork");
                break ;
            case 0: // c'est le descendant
                execlp(s,s,NULL);
                break ;
            default: // c'est le parent
                wait(NULL);
        }
    }
    while(1);
}

```

Solutions des exercices du TD/TP 5

Solution 1 le droit en écriture sur le répertoire contenant permet de supprimer un fichier. Il n'est pas nécessaire d'avoir le droit d'écriture sur le fichier.

Donc il n'y a pas de relation entre le droit de supprimer un fichier et les droit d'accès (y compris le droit de modifier) ce fichier. Par conséquent, on peut ne pas avoir le droit de supprimer le contenu (droit w), tout en pouvant supprimer le fichier par le droit w sur le répertoire contenant.

Un propriétaire peut s'enlever les droits. Ce n'est pas grave, car il a le droit de les changer.

S'enlever ses propres droits d'accès permet à un utilisateur juste d'attirer sa propre attention sur ce fichier.

Ne pas oublier finalement, que pour modifier le contenu d'un fichier il faut pouvoir atteindre ce fichier (cf. accès aux répertoires) et avoir le droit d'écriture dessus (la lecture n'est pas strictement nécessaire, à condition de ne l'ouvrir qu'en écriture...).

Solution 2 La notion de groupe sous linux est celle de l'utilisateur seul dans son groupe dit **privé**. On peut également appartenir à de vrais groupes (constitués de plusieurs personnes) créés par les administrateurs. Dans ce cas, pour changer de groupe, il faut utiliser la commande **newgrp**: attention, penser alors à expliquer que la commande **newgrp** modifie le groupe du propriétaire uniquement dans le nouveau shell qu'elle lance. En effet, on peut constater qu'en faisant **newgrp GrpA** par exemple, l'identité est changée uniquement dans la fenêtre dans laquelle on a lancé cette commande. Du coup, on peut quitter cette configuration avec **exit** et il faudra faire deux **exit** pour sortir complètement. On peut utiliser la commande **id** qui affiche l'identité courante de l'utilisateur avec les divers groupes auxquels il appartient...

Le programme qu'on peut écrire : voir **delfic.c**, un tout petit équivalent de la commande **rm** avec une analyse de l'erreur (utilisation de **errno**). Autre solution, mais plus lourde et moins convaincante, on peut dans un même programme faire un **open()** pour annoncer si on peut ouvrir ou non, puis après la fermeture, faire le **unlink()** pour constater que les droits pour chacune des opérations, accès, modification et suppression sont différents.

delfic.c

```
/** programme supprimant le fichier passé en paramètre et affichant le texte
 * obtenu par errno; il utilise unlink().
 */
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
int main(int n, char *argv[]){
    if (n < 2){
        fprintf(stderr, "Au moins un argument svp !\n");
        exit(1);
    }
    int err=unlink(argv[1]);
    if (err < 0){
        fprintf(stderr,"Erreur à la suppression de %s\n", argv[1]);
        fprintf(stderr,"Erreur numéro : %d, signifiant : %s\n",errno,strerror(errno));
        //si on veut utiliser perror, penser à sauvgarder errno
        // car cette cochonnerie de perror la détruit !
    } else
        printf("Suppression de %s effectuée !\n",argv[1]);
}
```

Solution 3 Sur un répertoire, le droit x permet de traverser le répertoire (faire **cd**, désigner un fichier inclus), le droit r permet de le lire (de connaître le contenu, faire **ls** par exemple). Dans les manips, laisser le droit x sans r, pour constater qu'en donnant le chemin à quelqu'un il peut y aller, sinon, il ne peut pas.

Donc laisser le droit x seul, interdit la visualisation du contenu d'un répertoire, sans interdire l'accès à un fichier si on connaît son nom.

Attention : il y a des différences subtiles avec **ls** (ne fait pas le même type d'accès que **ls -l**!). En effet, lors d'un accès à un répertoire, **ls -l** demande l'accès x au répertoire, sinon il n'affiche rien (même pas une erreur...) dans le cas d'un accès en lecture seule. Ce n'est qu'ensuite (en allant un cran plus loin, vers un répertoire ou fichier inclus dans ce répertoire) qu'il affiche une erreur ! Donc le mieux est d'utiliser un programme pour comprendre ...

Solution 4 On peut y créer ce qu'on veut, par exemple un bout d'arbre. Si on a ce droit dans un répertoire appartenant à un autre utilisateur **U₁**, on peut y créer un sous-arbre que **U₁** ne pourra plus effacer !

Solution 6 Il faut vérifier si la taille est compatible avec le nombre de blocs, si un même bloc n'appartient pas à deux fichiers, si les numéros d'inodes sont uniques. Ici, c'est tout. Le système, lors du lancement vérifie la cohérence complète (voir **fsck()**), mais les seules données de l'exercice ne permettent pas de vérifier plus.

situation 1 : correcte

situation 2 : taille vs nb blocs (pas assez de blocs alloués pour une telle taille de fichier)

situation 3 : 4102 est un bloc alloué pour deux fichiers.

Solution 7 À la seule vue du tableau tout ce qu'on peut dire est que les éléments ayant 1 en nombre de liens sont certainement des fichiers. Par contre, ceux qui ont plus d'un lien peuvent être des répertoires ou des fichiers. À noter que le nombre de liens pour les répertoires a une signification variable selon divers systèmes unix. La tendance actuelle est d'y mettre le nombre de **répertoires** inclus. C'est-à-dire que lors de la création d'un répertoire il y a déjà 2 dans le nombre de liens (. et ..). Il est incrémenté à chaque création de répertoire, mais pas à chaque ajout de fichier.

Solution 8 La question semble un peu ambiguë : il faut considérer l'organisation 1 seule (comme un tableau contenant les deux premières lignes seulement) et dire si elle est cohérente avec le tableau contenant le nombre de liens. Puis recommencer la même question avec l'organisation 2.

l'Org 1 est cohérente : les inodes existent et on a deux références au 148, donc c'est inférieur à 3 et on déduit qu'il doit y avoir un autre 148 en dehors des répertoires 1 et 2.

l'Org 2 pose un problème sur le nombre de liens vers le fichier d'inode 498.

Solution 9 L'appel système a permis de créer la commande. On peut utiliser l'appel système pour modifier les droits directement dans un programme.

Solution 10 L'appel système chmod() modifie les droits mais ces modifications seront effectives pour les **processus suivants** ! Le processus qui fait l'appel garde les droits qu'il a eu à l'ouverture. Il faut donc envisager un processus qui ouvre un fichier, obtient ainsi les droits pour faire des opérations et ce processus fait un chmod(). Ainsi, on peut écrire dans un fichier ouvert avec les droits en écriture mais modifiés par chmod pour ne laisser sur le fichier que les droits en lecture ! Dans ce cas, seuls les processus qui suivent celui qui a fait la modification seront concernés par les nouveaux droits. Noter que les processus qui suivent ne sont pas nécessairement des descendants. En résumé, tous les open() suivants seront affectés par les nouveaux droits.

tstchmod.c

```
/* on vérifie qu'après open, la modification des droits ne sera effective
qu'après la fermeture. Attention, si on fait ls -l, on voit les nouveaux droits.
```

Attention : c'est même pire : si le fichier n'existe pas, on peut demander un open avec des droits contradictoires à ceux qu'on va donner au fichier. De toute façon, les droits ne seront effectifs qu'après la fermeture, à l'open suivant. On peut donc tester ce programme d'abord avec le fichier "irrel" qui n'existe pas, ensuite constater ce qui se passe lorsque le fichier existe et qu'on demande de l'ouvrir en contradiction avec les droits existants.

La conclusion : lorsque le fichier n'existe pas, c'est open qui gère les droits du processus, et ceux du fichier pour les open suivants seulement (de ce processus et des autres bien sûr).

Enfin, une dernière : si on ne teste pas les retours de open et write, on peut avoir l'impression que le programme fonctionne !! il faut vérifier le contenu du fichier pour voir qu'il ne fonctionne pas.

```
/*
#include <sys/stat.h>/pour chmod
#include<fcntl.h>/pour open
#include <unistd.h>/pour read write
#include <stdio.h>
#include <stdlib.h>

main(){
    int fd=open("nouv", O_RDWR|O_CREAT,S_IRUSR|S_IWUSR);
    if (fd<=0){
        printf("pb ouverture\n");
        exit(1);
    }
    printf("open en création et lecture-écriture sur nouv réalisé\n");
    int nbecr=write(fd,"abcdefg",7);
    printf("%d caractères écrits sur irrel !\n",nbecr);
    /* on laisse ici le temps de faire ls à la main mais on pourrait aussi lancer
       un system(), ou encore faire un acces 'à l'inode et afficher les droits. On
       fait comme si on avait plus confiance dans ls...
    */
    printf("On peut consulter les droits, ls -l ; return ensuite svp !\n");
    getchar();
    int retchmod=chmod("nouv",S_IRUSR|S_IRGRP);
```

```

if (retchmod !=0){
    printf("Ca alors, pb au chmod !!!\n");
    exit(1);
}
printf("Apres chmod sur le fichier ouvert\n");
nbecr=write(fd,"hijklmn",7);
printf("On a ajoute %d caractères\n",nbecr);
printf("on attend a nouveau un return, apres consultation\n");
getchar();
close(fd);
printf("fichier ferme\n");
printf("apres le return, on va encore modifier les droits\n");
retchmod=chmod("nouv",S_IRWXU);
if (retchmod !=0){
    printf("ca alors, pb au chmod !!!\n");
    exit(1);
}

```

Solution 11 Différence entre chmod() et fchmod() : le premier peut s'appliquer à un fichier non ouvert. Le second ne s'applique qu'à un fichier ouvert.

Solution 12 On ne peut pas le faire. Si umask vaut 222 alors on supprime volontairement toute possibilité d'écriture à toutes les catégories d'utilisateurs.

Solution 13 Oui. Oui, pour les deux questions sur le partage. Les écritures se mélangent (mais y sont toutes) ou s'effacent les unes les autres, selon le mode de partage (héritage ou écriture dans des endroits distincts du fichier dans le premier cas, écriture en même position dans le deuxième cas). Voir ci-après les explications. Une chose est certaine : comme on ne sait pas quel processus aura la main, ce n'est certainement pas voulu par les utilisateurs.

Solution 14 Attention : l'explication relève de la «table des fichiers ouverts du système». Si elle n'a pas encore été traitée en cours, laisser cette question de côté pour y revenir plus tard.

Lorsque l'obtention du descripteur provient de l'héritage du parent, ils partagent la même donnée *position* de la TFOS (table des fichiers ouverts du système). Donc l'écriture d'un des processus va modifier cette donnée. Par conséquent, une écriture par l'autre processus va se faire à la suite. Mais attention : on n'est pas maîtres de l'ordre d'activation des processus.

Si le même fichier est ouvert par chaque processus, sans héritage, alors chacun a son propre pointeur. Si ces écritures se font au même endroit (écritures concurrentes), alors le dernier a raison. Sinon, le résultat est imprévisible : on peut aussi bien trouver des données des deux processus, si l'écriture de chacun s'est faite à des endroits différents, ou n'importe quoi s'il y a chevauchement.

Solution 15 ls -i pour visualiser les numéros d'inodes ; ls -l visualise le nombre de liens.

Solution 16 À partir d'un numéro d'inode il faudra parcourir toute la partition (le système de fichiers simple) pour trouver le/les fichiers référençant cet inode. Mais pour parcourir seulement cette partition, il faut savoir au moins où est la racine du système de fichiers simple. Donc connaître les points de «montage» : commande mount ou showmount.

Solution 17 Si l'utilisateur a modifié les droits de figaro après sa création et avant sa nouvelle édition (chmod), ou s'il a modifié son umask, alors les fichiers figaroLevieux et figaro auront des droits différents.

Dans le cas d'un lien dur sur figaro avant édition, c'est figaroLevieux qui sera pointé après édition.

Dans le cas d'un lien symbolique sur figaro avant édition, c'est le nouveau figaro après édition qui reste pointé. En effet, il y a bien attribution d'un nouvel inode au nouveau figaro, mais le lien symbolique ne référence que le chemin.

Solution 18 Même si les couleurs permettent aujourd'hui de visualiser les liens symboliques pointant sur des répertoires ou fichiers existants ou non, rien n'empêche d'en créer vers des répertoires et fichiers inexistant. Lesquels d'ailleurs peuvent devenir «existants» par la suite.

Comme un lien symbolique a une vie indépendante de pointeur, indépendamment du pointé, on peut supprimer le pointeur, le pointé sans indication d'erreur. On aura une erreur si on essaie d'accéder par un lien symbolique existant, un pointé inexistant.

On ne peut connaître l'existence des liens symboliques qu'en parcourant la totalité du système de fichier cette fois-ci : ces liens peuvent traverser les systèmes de fichiers simples et il n'y a pas de référence «inverse» (du pointé vers le pointeur). Donc il faut parcourir tout ; qui plus est, il n'y a même pas un décompte permettant de s'arrêter avant la fin. Noter qu'il peut y avoir des boucles...

Solutions des exercices du TD/TP 6

Solution 1 insister sur la distinction entre les unités $K = 2^{10}$, $M = 2^{20}$, $G = 2^{30}$, $T = 2^{40}$ etc, des capacités en octets, $Ko, Mo,$,

1. nombre de blocs théoriques : $10 + 2Ko \div 4o + 2^9 * 2^9 + 2^9 * 2^9 * 2^9 \geq 2^{27}$ blocs soit 2^{38} octets i.e. $256Go$;
2. la taille de chaque fichier est codée sur $32bits$, donc au plus $4Go$;
3. Pour qu'il reste de l'espace sur une partition, sans que l'on puisse ajouter un fichier, il «suffit» que la table des inodes soit pleine. Ça se dimensionne lors de la création du système de fichiers (formatage); pour plus de détails voir `mkfs` ou `newfs`.

Solution 2 1. — ne pas s'attarder sur : `st_dev st_rdev st_blksize st_blocks`. En effet, ils concernent les caractistiques des disques et apportent peu d'informations intéressantes. En particulier pour la taille des blocs, voici un extrait du man linux :

Remarquez que `st_blocks` n'est pas toujours compté en blocs de la taille `st_blksize`, et que `st_blksize` peut à la place induire une notion de taille de bloc optimale pour des entrées/sorties efficaces.

- `st_ino`; Numéro i-noeud (`ls -l`)
- `st_mode`; type de fichier et droits : 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, et 9 correspondants aux droits d'accès. On peut déjà en déduire qu'il reste 3 bits dont on ne parle pas ici.
- `st_nlink`; Nb liens matériels (`ln`)
- `st_uid` `st_gid` prop et groupe du prop
- `st_size` Taille du fichier en octets
- `st_atime` est la date de dernier accès (lecture, écriture ou création du fichier),
- `st_mtime` est la date de dernière modif (accès en écriture ou création),
- `st_ctime` est la date de dernière modif de l'inode seul (droits, propriété, liens...)
- Attention : on constate que la date de création n'est pas conservée, car elle est écrasée dès la première écriture du fichier. C'est `st_mtime` qui est affichée par la plupart des logiciels
- remarquons que les pointeurs sur blocs de données ne sont pas accessibles.

2. on peut noter que l'appel `fstat()` permet d'obtenir le même résultat cette fois ci à partir d'un descripteur (donc sur un fichier ouvert dans un programme, alors qu'il n'est pas nécessaire de l'ouvrir pour l'appel `stat()`). Enfin `lstat` ne traverse pas les lieux symboliques et donne l'état du fichier lien.

Solution 3 1. `typeFichier.c`

```
/* test le type d'un fichier ou répertoire */
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]){
    struct stat etat;

    if(argc!=2){
        fprintf(stderr,"utilisation: %s chemin\n", argv[0]);
        exit(1);
    }
    else if (lstat(argv[1],&etat)<0){
        fprintf(stderr,"Erreur de chemin (nom de fichier) : %s\n",argv[1]);
        exit(2);
    }
    else if ((etat.st_mode&S_IFMT)==S_IFDIR){ /* ou S_ISDIR() */
        printf("%s est un répertoire !\n",argv[1]);
    } else if(S_ISREG(etat.st_mode)){
        printf("%s est un fichier régulier !\n",argv[1]);
    } else if(S_ISLNK(etat.st_mode)){ /* grâce à lstat */
        printf("%s est un lien symbolique !\n",argv[1]);
    }
    else
}
```

```

        printf("%s est d'une autre nature !\n",argv[1]);
        return 0;
    }

2. droitFichier.c
/* affiche les droits d'un fichier ou répertoire ou */
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]){
    struct stat etat;

    if(argc!=2){
        fprintf(stderr,"utilisation: %s chemin\n", argv[0]);
        exit(1);
    }
    else if (lstat(argv[1],&etat)<0){
        fprintf(stderr,"Erreur de chemin (nom de fichier) : %s\n",argv[1]);
        exit(2);
    }
    else { /* début de l'affichage */
        if ((etat.st_mode&S_IFMT)==S_IFDIR){ /* ou S_ISDIR() */
            printf("%s d",argv[1]);
        } else if(S_ISREG(etat.st_mode)){
            printf("%s -",argv[1]);
        } else if(S_ISLNK(etat.st_mode)){ /* grâce à lstat */
            printf("%s l",argv[1]);
        } else {
            printf("%s ?",argv[1]);
        }
        if(etat.st_mode & S_IRUSR) printf("r"); else printf("-");
        if(etat.st_mode & S_IWUSR) printf("w"); else printf("-");
        if(etat.st_mode & S_IXUSR) printf("x"); else printf("-");
        if(etat.st_mode & S_IRGRP) printf("r"); else printf("-");
        if(etat.st_mode & S_IWGRP) printf("w"); else printf("-");
        if(etat.st_mode & S_IXGRP) printf("x"); else printf("-");
        if(etat.st_mode & S_IROTH) printf("r"); else printf("-");
        if(etat.st_mode & S_IWOTH) printf("w"); else printf("-");
        if(etat.st_mode & S_IXOTH) printf("x"); else printf("-");
        printf("\n");
    }
    return 0;
}

```

- Solution 4**
1. Les fonctions d'accès aux répertoires permettent d'éviter un travail fastidieux de parcours d'un fichier structuré avec des éléments de longueur variable (nom de fichier). De plus, elles permettent une certaine protection vis à vis d'une programmation « hasardeuse » avec open read lseek close.
 2. La création d'un nouveau fichier (creat ou open) ne fait que créer une entrée dans la table des inodes alors que la création d'un répertoire (mkdir) nécessite l'insertion de 2 entrées • et ••.
 3. Il suffit de créer un fichier (creat ou open) dans ce répertoire.
 4. Pour supprimer un répertoire, il faut que le répertoire ne contienne plus que • et •• et qu'on ait le droit w sur ••. Pour supprimer un lien, il suffit d'avoir le droit w sur le répertoire contenant ce lien.

Solution 5 monls.c

```

/* affiche les droits d'un fichier ou répertoire ou */
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

```

```

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]){
    if(argc!=2){
        fprintf(stderr,"utilisation: %s chemin\n", argv[0]);
        exit(1);
    }
    DIR * rep=opendir(argv[1]);
    if (rep==NULL){
        fprintf(stderr,"Impossible d'ouvrir le répertoire : %s\n",argv[1]);
        exit(2);
    }else{
        struct dirent * entree; /* entrée de répertoire */
        char chemin[512]; /* chemin d'une entrée */
        struct stat etat; /* état de l'inode */
        while ((entree=readdir(rep)) != NULL){
            printf("%ld ",entree->d_ino) ; /* inode number : long en décimal */
            strncat(strcat(strcpy(chemin,argv[1]),"/"),entree->d_name);
            if (lstat(chemin,&etat)<0){
                fprintf(stderr,"Impossible d'ouvrir le fichier : %s\n",chemin);
                exit(3);
            }
            if ((etat.st_mode&S_IFMT)==S_IFDIR){ /* ou S_ISDIR() */
                printf("d");
            } else if(S_ISREG(etat.st_mode)){
                printf("-");
            } else if(S_ISLNK(etat.st_mode)){ /* grâce à lstat */
                printf("l");
            } else {
                printf("?");
            }
            if(etat.st_mode & S_IRUSR) printf("r"); else printf("-");
            if(etat.st_mode & S_IWUSR) printf("w"); else printf("-");
            if(etat.st_mode & S_IXUSR) printf("x"); else printf("-");
            if(etat.st_mode & S_IRGRP) printf("r"); else printf("-");
            if(etat.st_mode & S_IWGRP) printf("w"); else printf("-");
            if(etat.st_mode & S_IXGRP) printf("x"); else printf("-");
            if(etat.st_mode & S_IROTH) printf("r"); else printf("-");
            if(etat.st_mode & S_IWOTH) printf("w"); else printf("-");
            if(etat.st_mode & S_IXOTH) printf("x"); else printf("-");
            printf(" %s\n",entree->d_name);
        }
        closedir(rep);
    }
    return 0;
}

```

Solution 6 lsmodif.c

```

/* affiche les entrées modifiées depuis moins de n jours d'un répertoire */
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <sys/time.h>

int main(int argc, char *argv[]){
    if(argc!=3){

```

```

        fprintf(stderr,"utilisation: %s chemin nbjours\n", argv[0]);
        exit(1);
    }
    int nbjours=atoi(argv[2]); /* ascii to integer */
    DIR * rep=open(dir(argv[1]));
    if (rep==NULL){
        fprintf(stderr,"Impossible d'ouvrir le répertoire : %s\n",argv[1]);
        exit(2);
    }
    struct dirent * entree; /* entrée de répertoire */
    char chemin[512]; /* chemin d'une entrée */
    struct stat etat; /* état de l'inode */
    struct timeval now;
    if (gettimeofday(&now,NULL)<0){
        fprintf(stderr,"Impossible d'obtenir la date système :!\n");
        exit(3);
    }
    while ((entree=readdir(rep)) != NULL){
        strncat(strcat(strcpy(chemin,argv[1]),"/"),entree->d_name);
        if (lstat(chemin,&etat)<0){
            fprintf(stderr,"Impossible d'ouvrir le fichier : %s\n",chemin);
            exit(4);
        }
        if (difftime(now.tv_sec,etat.st_mtime)<nbjours*24*60*60){
            printf("%s %s",entree->d_name, ctime(&(etat.st_mtime)));
        }
    }
    closedir(rep);
    return 0;
}

```

Algorithme 1 : parcours récursif d'une arborescence de répertoires

Données : rep un nom de répertoire
Résultat : void
Fonction parcours(rep) : void ;
si (ouvrir(rep) possible) alors
| tant que (elemLu=lire(rep) possible) faire
| | si (elemLu est un répertoire) alors
| | | traitementRep(elemLu);
| | | si (elemLu != ".") et (elemLu != "..") alors
| | | | parcours(rep/ elemLu);
| | | | traitementAutre(elemLu);
| | | | //si elemLu est fichier ou autre, traitement
| | | |
sinon
| | afficher (ouverture de rep impossible);

Solution 7 1.

2. arboindent.c

```

/* parcours récursif d'un dir et affichage indenté des rep */
#include <dirent.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int parcours(char* rep, int indent){ /* retourne 0 si OK */
    DIR * repCourant=open(dir(rep)); // ptr sur obj dynam. répertoire courant
    if (repCourant==NULL){

```

```

        fprintf(stderr,"Impossible d'ouvrir le répertoire : %s\n",rep);
        return -1;
    }
    char *chemin=(char *)malloc(1024);
    struct dirent *entree;
    struct stat etat; /* pour le lstat */
    while ((entree=readdir(repCourant)) != NULL){ // pour chaque entrée
        strcat(strcat(strcpy(chemin,rep),"/"),entree->d_name);
        // lstat pour ne pas être embêté par les liens
        if(lstat(chemin,&etat)<0){
            fprintf(stderr,"Impossible d'ouvrir le fichier : %s\n",chemin);
        }
        if (S_ISDIR(etat.st_mode) && strcmp(entree->d_name, ".") && strcmp(entree->d_name, "..")){
            int i; for(i=0;i<indent;i++) printf(" ");
            printf("%s\n",chemin);
            parcours(chemin,indent+1);
        }
    }
    free(chemin);
    closedir(repCourant);
    return 0;
}
int main(int argc, char*argv[]){
    if(argc!=2){
        fprintf(stderr,"utilisation: %s repertoire\n", argv[0]);
        exit(1);
    }
    printf("%s\n",argv[1]); /* la racine */
    return parcours(argv[1],1);
}

```

Solution 8 1. Il faut :

- créer une fois pour toutes un fichier vide, dont le seul objectif sera de mémoriser la date de la sauvegarde.
Donc à la fin de la sauvegarde (toute la première fois) on va faire l'équivalent d'un `touch` (mettre à jour `st_ctime` par exemple).
- ensuite, lors de chaque sauvegarde incrémentale, on ne sauvegarde que les fichiers dont l'une des dates, `st_atime` ou `st_mtime` est postérieure à celle (`st_ctime`) du fichier vide en question.

Solution 9 Un numéro identifiant est de taille constante tandis que les noms d'utilisateurs peuvent varier et de plus prennent plus de place. Les données utilisateurs sont stockées dans des fichiers.

Quelques inconvénients :

- petit : un appel système de plus ; ici, obligation de deux appels systèmes différents pour récupérer les caractéristiques d'un fichier (`stat()`) et celles de l'utilisateur (`getpw()`),
- lorsqu'on transporte un système de fichier d'un site à un autre, on va attribuer les fichiers du site distant à un utilisateur local différent ou inexistant, mais ceci est toujours un problème : y a-t-il une bonne solution autre que de ne pas transporter l'identité (nom ou numéro) lorsqu'on transporte un système de fichiers ?

Solutions des exercices du TD/TP 7

Solution 1 deux solutions :

1. depuis le terminal d'attachement du processus : ^C génère le signal SIGINT (signal 2). ^\ (control avec backslash) génère SIGQUIT (signal 3);
2. dans une autre fenêtre récupérer le numéro du processus (ps) et faire kill -SIGINT ce_numero.

Solution 2 1. sigintmsg.c

```
/* sigintmsg.c
   On peut aussi bien faire ^C que lui expedier
   le signal SIGINT d'une autre fenetre :
   kill -SIGINT numproc ; le comportement est identique
*/
#include<stdio.h>
#include<signal.h>

void gst(int sig){ /* gestionnaire du signal */
    printf("bien recu le signal %d\n",sig);
}

int main(int n, char **argv){
    struct sigaction action;
    action.sa_handler = gst;
    sigaction(SIGINT, &action, NULL);

    printf("Le processus boucle sans fin !\n");

    while(1);
}

2. sigintignore.c
/* sigintmsg.c
   On peut aussi bien faire ^C que lui expedier
   le signal SIGINT d'une autre fenetre :
   kill -SIGINT numproc ; le comportement est identique
*/
#include<stdio.h>
#include<signal.h>

int main(int n, char **argv){
    struct sigaction action;
    action.sa_handler = SIG_IGN;
    sigaction(SIGINT, &action, NULL);

    printf("Le processus boucle sans fin !\n");

    while(1);
}

3. sigintmsgunefois.c
/* sigintmsg.c
   On peut aussi bien faire ^C que lui expedier
   le signal SIGINT d'une autre fenetre :
   kill -SIGINT numproc ; le comportement est identique
*/
#include<stdio.h>
#include<signal.h>

struct sigaction action; /* variable globale car accédée dans gst */

void gst(int sig){ /* gestionnaire du signal */
    printf("bien recu le signal %d\n",sig);
    action.sa_handler=SIG_DFL;
    sigaction(SIGINT, &action, NULL);
}
```

```

}

int main(int n, char **argv){
    action.sa_handler = gst;
    sigaction(SIGINT, &action, NULL);

    printf("Le processus boucle sans fin !\n");

    while(1);
}

```

Solution 3 1. alarm1fois.c

```

/* alarm1fois.c
   utilisation de alarm pour que l'utilisateur reponde dans les 10 sec
*/
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
#include <stdlib.h>

struct sigaction action; /* variable globale car accédée dans gst */

void gst (int sig){
    printf("Trop tard\n");
    exit(1);
}

int main(int n, char **argv){
    action.sa_handler=gst;
    action.sa_flags=SA_RESTART; /* redémarrer l'appel syst */
    sigaction(SIGALRM, &action, NULL);
    printf("Entrez un entier dans les 10 secondes : ");
    alarm(3);
    int i;
    scanf("%i", &i);
    alarm(0); /* débrancher l'alarme */
    printf("bravo pour votre célérité ! entier saisi : %d\n",i);
    return 0;
}

```

2. alarm2fois.c

```

/* alarm1fois.c
   utilisation de alarm pour que l'utilisateur reponde dans les 10 sec
*/
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
#include <stdlib.h>

struct sigaction action; /* variable globale car accédée dans gst */

void gst(int sig){
    static int cpt=1;
    switch(cpt) {
        case 1 :
            printf("Veuillez repondre plus vite !\n");
            cpt++;
            alarm(3);
            return;
            break;
        case 2 :
            printf("Trop tard\n");
            exit(1);
            break;
    }
}

```

```

    }

}

int main(int n, char **argv){
    action.sa_handler=gst;
    action.sa_flags=SA_RESTART; /* redémarrer l'appel syst */
    sigaction(SIGALRM, &action, NULL);
    printf("Entrez un entier dans les 10 secondes : ");
    alarm(3);
    int i;
    scanf("%i", &i);
    alarm(0); /* débrancher l'alarme */
    printf("bravo pour votre célérité ! entier saisi : %d\n",i);
    return 0;
}

3. alarm3fois.c
/* alarm3fois.c
   utilisation de alarm pour que l'utilisateur reponde dans les 10 sec
*/
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
#include <stdlib.h>

struct sigaction action; /* variable globale car accédée dans gst */

void gst(int sig){
    static int cpt=1;
    switch(cpt) {
    case 1 :
        printf("Veuillez repondre plus vite !\n");
        cpt++;
        alarm(3);
        return;
        break;
    case 2 :
        printf("Veuillez repondre encore plus vite !\n");
        cpt++;
        alarm(3);
        return;
        break;
    case 3 :
        printf("Trop tard\n");
        exit(1);
        break;
    }
}

int main(int n, char **argv){
    action.sa_handler=gst;
    action.sa_flags=SA_RESTART; /* redémarrer l'appel syst */
    sigaction(SIGALRM, &action, NULL);
    printf("Entrez un entier dans les 10 secondes : ");
    alarm(3);
    int i;
    scanf("%i", &i);
    alarm(0); /* débrancher l'alarme */
    printf("bravo pour votre célérité ! entier saisi : %d\n",i);
    return 0;
}

```

En particulier l'utilisation du flag SA_RESTART pour relancer les appels systèmes (entrées-sorties) interrompues. En effet, cela dépend du système d'exploitation, Linux ne relance pas automatiquement les entrées-sorties interrompues par un signal. Si on veut les relancer, il faut utiliser ce flag.

Solution 4 C'est un temps minimal, car tout processus ne peut être sûr d'être élu par l'ordonnanceur exactement après l'épuisement d'un délai. Notre système étant à temps partagé, un processus qui a demandé à s'endormir quelques temps est sûr de dormir au moins pendant ce temps. Noter que c'est bien pour ça que certaines applications de contrôle de processus par exemple, demandent un contrôle en *temps réel*; dans ce type de système, un délai doit être impérativement respecté. Linux aussi fournit une version temps réel.

Solution 5 sigsegverror.c

```
/* sigsegv.cc
   Gestion de SIGSEGV en le produisant.
   ici avec complément sur errno pour afficher
   la valeur de errno et le texte correspondant.
*/
#include<stdio.h>
#include <stdlib.h>
#include<signal.h>
#include<errno.h>
#include <string.h>

void gst (int sig){
    printf("Reçu le signal : %d\n",sig);
    printf("Valeur de errno %d; message :%s\n",errno,strerror(errno));
    exit(1);
}

int main(){
    struct sigaction action;
    action.sa_handler = gst;
    sigaction(SIGSEGV, &action, NULL);

    int *pi=NULL;
    printf("oooooops : %d\n trop tard\n",*pi);
}
```

Attention : Ne pas confondre les erreurs des appels systèmes avec les messages affichées lors de la réception de signaux : un signal n'est pas un appel système.

Solution 6 voici quelques commentaires et réponses :

Remarques :

- Erreur classique : oublier que le parent doit avoir le temps de mettre en place son gestionnaire.
- le signal SIGCHLD est ingoré par défaut;
- un processus peut aussi utiliser wait() et waitpid() pour traiter la fin des enfants.
- il manque encore quelques précisions sur les processus dits *zombies* et il y a souvent des questions dessus. En fait un processus est dans cet état, lorsqu'il est terminé, mais qu'il en reste encore une trace dans la table des processus, tant que le parent n'a pas pris en compte la terminaison. L'action par défaut est une prise en compte, mais elle n'arrive pas toujours de suite.
- Pourquoi certains processus zombies restent dans cet état longtemps, même après la fin du parent ? Car ils attendent que leur père ait lu leur état de terminaison grâce à wait().

Réception des signaux :

- si le parent reçoit les deux signaux, il a eu le temps de mettre en place son gestionnaire et ensuite de traiter chacune des deux fins, ce qui suppose que le signal de l'un est arrivé suffisamment *en retard* (relatif) sur l'autre. On a vu que des signaux pouvaient se perdre, en particulier si le processus récepteur n'a pas eu la ressource *unité centrale* et que les deux processus ont envoyé leurs signaux rapprochés l'un de l'autre.
- Donc le parent peut ne recevoir qu'un des deux.
- Enfin, le dernier cas correspond aux deux processus enfants qui se sont terminés avant que le parent n'ait eu le temps de s'exécuter, ou encore n'est pas arrivé jusqu'à la mise en place de son gestionnaire de signaux.

Remarque terminale : On peut quand même gérer toutes les fins d'enfants, si on prend la peine de stocker leur identité et ensuite de les attendre tous (voir waitpid()). Pour chaque processus qui s'est terminé avant, la sortie de la fonction est immédiate.

Solution 7 debug.h

```
#ifndef _DEBUG_H
#define _DEBUG_H
/** affiche l'état courant de la pile lorsque des signaux surviennent
 * @param signaux tableau d'entier terminé par -1
```

```

/*
void debug(int* signaux);

/** fonction gestionnaire de signal qui affiche la pile
 * @param sig le numéro du signal déclenché
 */
void debug_gst(int sig);

#endif

debug.c

#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>
#include "debug.h"

static struct sigaction action; /* variable globale cachée */

// fonction gestionnaire de signal (callback) : c'est la même pour tous les
// signaux qu'on veut déboguer
void debug_gst(int sig){
    printstack(1);
    exit(EXIT_FAILURE);
}

// fonction à appeler en début de main afin de voir l'état de la pile lorsque
// un signal survient
void debug(int* signaux){
    action.sa_handler = debug_gst;
    int* p=signaux;
    while(*p!=-1){
        sigaction(*p, &action, NULL);
        p++;
    }
}

testDebug.c

#include <stdio.h>
#include <signal.h>
#include "debug.h"

char *s;
void g(void){printf(s);} // génère signal SIGSEGV
void f(void){g();}

int main(){
    int signaux[]={SIGSEGV,SIGFPE, -1}; debug(signaux);
    f();
}

```

Solutions des exercices du TD/TP 8

Solution 1 Le plus simple : un programme qui boucle sur l'écriture d'un caractère dans le tube et affiche à chaque boucle l'indice de boucle. Lorsque le tube est plein, le programme se bloque, donc l'indice affiché donne la taille (attention à l'endroit où on met l'instruction d'affichage).

Plus rapide : on écrit 1024 octets par 1024 : si la taille est un multiple d'1Ko, tout se passe bien. Par contre, la dernière écriture est bloquante si seulement une partie du message peut être écrite ! (résultat 64 Ko)

Par conséquent, on écrit un programme qui remplit le tube par bloc de n caractères, avec n passé comme argument au programme.

tailletube.c

```
/* On remplit le tube avec 1 ou n caractère à la fois.
 */

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

int main(int n, char *argv[]){
    int bloc; /* nb car par écriture */
    if (n!=2){
        fprintf(stderr,"Syntaxe : %s n \n",argv[0]);
        exit(1);
    }else if (0>=(bloc=atoi(argv[1]))){
        fprintf(stderr,"Syntaxe : %s n ; n étant un entier naturel !\n",argv[0]);
        exit(2);
    }
    char s[bloc]; /* pas nécessaire d'init la chaîne à écrire */
    int tube[2];
    if (0>pipe(tube)){
        fprintf(stderr,"Impossible de créer le tube !\n");
        exit(3);
    }
    int i;
    for (i=1 ;i!=0 ; i++){
        write(tube[1], s,bloc);
        printf("%d octets\n",i*bloc);
    }
}
```

Solution 2 1. chacun ferme les descripteurs dont il n'a pas besoin :

A L'écrivain ne se termine pas :

- soit le lecteur ne sait pas combien de caractères il attend et il y aura blocage du lecteur, car l'écrivain existe toujours (i.e. il y a un descripteur ouvert en écriture) ;
- soit le lecteur attend exactement *n* caractères ; il les lit puis il ferme son descripteur. Rien ne se passe chez l'écrivain ! **Attention, la réponse suivante est fausse** : l'écrivain en est averti (signal SIGPIPE) et la réaction par défaut à un tel signal est "killed". En effet, si l'écrivain ne fait **pas d'écriture**, alors il ne sera jamais averti. Ce n'est que s'il demande à écrire qu'il sera averti de l'inexistence de lecteurs.

B L'écrivain se termine après 20 secondes : Lorsque l'écrivain se termine après 20 secondes de temporisation (sleep) à la fin de ses écritures dans le tube, le lecteur, s'il ne connaît pas le nombre de caractères à lire, va rester bloqué sur la lecture pendant ces 20 secondes (environ, compte tenu des moments où il a la main), puis recevoir une fin de fichier qui va le faire sortir du read, avec un retour nul (zéro) au read. Si le lecteur connaît le nombre à lire, il va s'arrêter mais comme ci-dessus, rien ne se passera chez l'écrivain.

C L'écrivain temporise puis refait une écriture et ferme le tube : le lecteur, s'il ne connaît pas le nombre de caractères à lire, va lire ce qui aura été écrit et recevoir le résultat 0 (zéro) lors de la dernière tentative de lecture. Si le lecteur connaît le nombre à lire et ne lit pas cette/ces dernière(s) écritures, alors si le lecteur s'est terminé avant ces écritures, l'écrivain recevra le signal SIGPIPE ; mais si le lecteur ne s'est pas encore terminé, alors l'écrivain ira jusqu'au bout et le lecteur aussi, ce qui fait que le tube sera abandonné (détruit par le système) avec un contenu non vide.

2. Aucun des processus ne ferme les descripteurs :

A L'écrivain ne se termine pas : lorsque le **lecteur** lit exactement *n* caractères et s'arrête, qu'il ferme son descripteur en lecture, ou que le système ferme dès que le processus s'arrête, peu importe, l'écrivain continue son travail ; si l'écrivain décide d'écrire, il ne recevra rien (pas de SIGPIPE) car il est aussi lecteur.

- B L'écrivain se termine après 20 secondes : si le lecteur ne sait pas combien de caractères sont à lire, il restera bloqué, que l'écrivain ait terminé ou non (i.e pendant les 20 secondes de temporisation ou après ou plus longtemps) ; en effet, le lecteur aura toujours deux descripteurs ouverts.
- C L'écrivain temporise puis refait une écriture et ferme le tube : peu de changements par rapport au précédent lorsque l'écrivain temporise et ajoute un caractère : le lecteur lira un caractère de plus s'il ne sait pas combien lire et sera bloqué, ou lira exactement n et abandonne l'écrivain à son sort.

Solution 3 Une lecture de plus que le nombre de caractères écrits, s'il lit les caractères un à un. En effet, lors de la lecture du dernier caractère, la fin de fichier est encore fausse.

- Solution 4**
1. Le parent, en tant que lecteur sera bloqué en attendant qu'il y ait au moins un caractère dans le tube ; donc rien de spécial à l'initialisation.
 2. On ne peut rien prévoir, car il n'y a aucune raison pour activer aussi souvent le lecteur ou l'écrivain ; de plus, si on veut lire n caractères et que seuls m sont disponibles, $m < n$ alors les m sont délivrés (par `read()`) et il n'y a pas blocage ; **Attention** : par contre, sur une demande d'écriture il y aura blocage si elle ne peut être entièrement satisfaite (i.e. la longueur totale demandée sera transférée du tampon à écrire dans le tube).
 3. En général, on peut dire que les situations de blocage à la lecture sont résumées par : le tube est vide et il y a au moins un écrivain en vie. Soit on suppose que le lecteur a fermé son descripteur en écriture alors il sera averti à la lecture lorsque le tube sera vide, avec un retour 0 sur cette lecture ; ceci ne se produit que si l'écrivain a fermé (ou s'est terminé). Soit le lecteur n'a pas fermé en écriture, et il restera bloqué lorsque le tube sera vide ; bien noter que tant qu'il y a des caractères disponibles dans le tube, le lecteur n'est averti de rien. S'il y a moins de 30 caractères, la lecture est toujours débloquée et on reçoit le nombre exact de caractères lus en résultat de la lecture. Noter enfin, du moment que l'écrivain se termine, le système prend en charge la fermeture. Donc tout se passe comme s'il avait fermé les descripteurs.

Solution 5

1. Voici le schéma des processus :

```
"2"           "3"           "5"           "7"
main ... 9 8 7 6 5 4 3 2 crible ... 9 7 5 3 crible ... 7 5 crible ... 7 crible
```

2. Le `main()` lit la borne dans le premier argument de la ligne de commande puis crée un tube initial, crée le premier fils et lui envoie tous les entiers compris entre 2 et borne. Pour écrire un entier dans un tube il suffit d'écrire ses `sizeof(int)` octets !

3. `[crible.c]`

```
/* Crible */
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <signal.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

void crible(int *in) {
    if (-1==close(in[1])){
        perror("Probleme de close");
        exit(3);
    }
    int P;
    if (read(in[0],(void *)&P,sizeof(int))!=sizeof(int)){
        close(in[0]);
        return; /* dernier pus */
    }
    printf("%d ",P); /* afficher P */
    int out[2];
    if(pipe(out)==-1) {
        perror("Probleme de pipe" );
        exit(1) ;
    }
    pid_t fils ; // Num du processus fils.
    fils=fork();
    switch(fils) {
```

```

        case -1 :
            perror("Probleme de fork");
            exit(2) ;
            break;
        case 0 : /* fils */
            close(in[0]);
            crible(out);
            exit(0);
            break;
        default : /* père */
            close(out[0]);
            int i;
            while(read(in[0],(void *)&i,sizeof(int))==sizeof(int)){
                if (i % P) {
                    write(out[1],(void *)&i,sizeof(int));
                }
            }
            close(in[0]);
            close(out[1]);
            //wait(0); // si synchro, chacun attend le suivant
            return;
            break;
        }
    }
int main(int argc,char *argv[]){
    int borne; // Borne de recherche
    if (argc!=2) {
        fprintf(stderr, "Syntaxe : crible n\n");
        exit(1);
    } else if((borne=atoi(argv[1]))<2){
        fprintf(stderr, "Syntaxe : crible n ; avec n entier > 2\n");
        exit(2);
    }
    int tube[2]; // Communication avec le processus fils
    pid_t fils ; // Num du processus fils.
    if(pipe(tube)==-1) {
        perror("Probleme de pipe" );
        exit(1) ;
    }
    switch (fils=fork()){
        case -1:
            perror("Probleme de fork" );
            exit(2) ;
            break;
        case 0 : /* Fils */
            crible(tube);
            exit(0);
            break;
        default : /* Père */
            close(tube[0]);
            for (int i=2;i<=borne;i++) {
                write(tube[1],(void *)&i,sizeof(int));
            }
            close(tube[1]);
            wait(0); // si synchro, le premier attend le second
            return 0;
            break;
    }
}

```

4. Si chaque processus attend son fils, la charge du système est plus importante, sinon, l'invite de commande réapparaît dès la terminaison du main().

5. **cribleexec.c**

/* Crible */

```

#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <signal.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc,char *argv[]){
    int borne; // Borne de recherche
    if (argc!=2) {
        fprintf(stderr, "Syntaxe : crible n\n");
        exit(1);
    } else if((borne=atoi(argv[1]))<2){
        fprintf(stderr, "Syntaxe : crible n ; avec n entier > 2\n");
        exit(2);
    }
    int tube[2]; // Communication avec le processus fils
    pid_t fils ; // Num du processus fils.
    if(pipe(tube)==-1) {
        perror("Probleme de pipe" );
        exit(1) ;
    }
    switch (fils=fork()){
    case -1:
        perror("Probleme de fork" );
        exit(2) ;
        break;
    case 0 : /* Fils */
        close(tube[1]);
        dup2(tube[0],0); /* redirige vers stdin */
        close(tube[0]);
        execl("fcrible","fcrible",NULL);
        break;
    default : /* Père */
        close(tube[0]);
        for (int i=2;i<=borne;i++) {
            write(tube[1],(void *)&i,sizeof(int));
        }
        close(tube[1]);
        //wait(0); // si synchro, le premier attend le second
        return 0;
        break;
    }
}

[fcrible.c]

/* Crible */
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <signal.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc,char *argv[]){
    int P;
    if (read(0,(void *)&P,sizeof(int))!=sizeof(int)){
        return 0; /* dernier pus */
    }
}

```

```

    }
    printf("%d ",P); /* afficher P */
    int out[2];
    if(pipe(out)==-1) {
        perror("Probleme de pipe");
        exit(1);
    }
    pid_t fils ; // Num du processus fils.
    fils=fork();
    switch(fils) {
    case -1 :
        perror("Probleme de fork");
        exit(2);
        break;
    case 0 : /* fils */
        close(out[1]);
        dup2(out[0],0); /* redirige vers stdin */
        close(out[0]);
        execl("fcrible","fcrible",NULL);
        break;
    default : /* père */
        close(out[0]);
        int i;
        while(read(0,(void *)&i,sizeof(int))==sizeof(int)){
            if (i % P) {
                write(out[1],(void *)&i,sizeof(int));
            }
        }
        close(out[1]);
        //wait(0); // si synchro, chacun attend le suivant
        break;
    }
}

```

Solution 6 1. A cause du retour à la ligne!

Algorithme : fonction compte(in)

Données : *in* : tube entrant

début

```

    fermer(in[écrire]) ;
    char C;
    si theta != (C=lire(in[lire])) alors
        /* C est le premier car */;
        int nb=1 ;
        out=créerPipe();
        f=fork();
        si f==0 alors
            /* fils */;
            fermer(in[lire]) ;
            return compte(out);
        sinon
            si f>0 alors
                /* père */;
                fermer(out[lire]);
                tant que i=lire(in[lire]) faire
                    si i == C alors
                        nb++;
                    sinon
                        écrire(out[écrire],i)
                    fermer(in[lire]) ;
                    fermer(out[écrire]);
                    afficher(C:nb );
                sinon
            afficher "Erreur du fork()";
    
```

3. **compte.c**

```

/* Compte */
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <signal.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

void compte(int *in) {
    if (-1==close(in[1])){
        perror("Probleme de close");
        exit(3);
    }
    char C;
    if (read(in[0],(void *)&C,1)==0){
        close(in[0]);
        return; /* dernier pus */
    }
    int nb=1;
    int out[2];
    if(pipe(out)==-1) {
        perror("Probleme de pipe" );
        exit(1) ;
    }
    pid_t fils ; // Num du processus fils.
    fils=fork();
    switch(fils) {
        case -1 :
    
```

```

    perror("Probleme de fork");
    exit(2) ;
    break;
case 0 : /* fils */
    close(in[0]);
    compte(out);
    exit(0);
    break;
default : /* père */
    close(out[0]);
    char i;
    while(read(in[0],(void *)&i,1)){
        if (i==C) {
            nb++;
        }else{
            write(out[1],(void *)&i,1);
        }
    }
    close(in[0]);
    close(out[1]);
    printf("%c:%d ",C,nb); /* afficher C:nb */
    return;
    break;
}
}

int main(int argc,char *argv[]){
    int fd; // file descriptor
    if (argc!=2) {
        fprintf(stderr, "Syntaxe : compte chemin\n");
        exit(1);
    } else if((fd=open(argv[1],O_RDONLY))<0){
        perror("L'argument donné ne peut être ouvert");
        exit(2);
    }
    int tube[2]; // Communication avec le processus fils
    pid_t fils ; // Num du processus fils.
    if(pipe(tube)==-1) {
        perror("Probleme de pipe" );
        exit(1) ;
    }
    switch (fils=fork()){
    case -1:
        perror("Probleme de fork" );
        exit(2) ;
        break;
    case 0 : /* Fils */
        compte(tube);
        exit(0);
        break;
    default : /* Père */
        close(tube[0]);
        char c;
        while(read(fd,&c,1)){
            write(tube[1],&c,1);
        }
        close(fd);
        close(tube[1]);
        return 0;
        break;
    }
}

```

Test : ./compte compte.c

/:25 *:19 :374 C:7 o:51 m:15 p:32 t:62 e:113

```

:92 #:9 i:84 n:48 c:48 1:45 u:37 d:39 <:10 s:52 b:22 .:11 h:19 >:9
r:71 g:6 a:32 y:5 f:33 w:7 v:8 (:55 ):55 {:14 -:5 1:20 =:15 [:16 ]:
16 ":16 P:6 ;:49 x:10 3:1 ):14 0:14 ,:15 &:5 :22 2:6 _:3 N:3 k:10
::9 é:2 +:2 %:2 !:1 S:1 \:1 0:2 R:1 D:1 L:2 Y:1 ':1 é:1 è:1 F:1

```

Solution 7 1. Il faut deux tubes nommés, un dans chaque sens. Et deux processus pour chaque utilisateur, un lecteur et un écrivain. Sinon, il faudrait mettre en place une synchronisation fastidieuse : soit pour dire à *toi*, soit annoncer la longueur de chaque chaîne, soit encore utiliser un autre élément (signal - non, car entre deux utilisateurs!!!)

2. N'importe quel processus peut créer les tubes (mkfifo). On supposera que le processus écrivain ou lecteur utilise l'argument passé à la ligne de commande comme nom de fifo et tentera de le créer (s'il n'existe pas déjà). De toute façon, à l'ouverture, il y aura rendez-vous, donc attente jusqu'à ce que le correspondant existe.
3. Protocole de terminaison : c'est le dernier processus qui doit supprimer le inode. On propose que ce soit le lecteur après avoir lu 0 octets, l'écrivain ayant décidé de fermer en envoyant une fin de fichier (control D). L'autre utilisateur pourra alors faire de même.
4. Si on tue le lecteur, l'écrivain est averti par SIGPIPE comme dans un tube simple qu'il n'y a plus de lecteur.

5. **lecteur.c**

```

/* Lecteur sur un tube nommé : réalisation de rendez-vous avec l'écrivain.
Chacun des deux essaie de créer le tube nommé par mkfifo. S'il existe déjà, on
continue bien sûr. Donc l'idée est juste de créer le tube par le premier
processus. On lit ligne par ligne.
*/
#include <sys/types.h> // tous appels
#include <sys/stat.h> // mkfifo, open
#include <stdio.h> // perror
#include <errno.h> // errno
#include <stdlib.h>
#include <string.h>
#include <unistd.h> // read, write
#include <fcntl.h> // open

int main(int n, char * argv[]){
    if (n != 2){
        fprintf(stderr, "Syntaxe : %s nomdutube\n", argv[0]);
        exit(1);
    }
    if(mkfifo(argv[1], S_IRUSR|S_IWUSR|S_IROTH|S_IWOTH)==-1 && errno!=EXIST){
        perror("Erreur mkfifo - on continue");
    } else {
        chmod(argv[1], 0666); /* pour permettre aux écrivains d'écrire */
    }
    int tube=open(argv[1], O_RDONLY);
    if (tube == -1){
        perror("Erreur ouverture tube");
        exit(2);
    }
    int K=1024,nb;
    char msg[K+1];
    while(0!=(nb=read(tube,msg,K))){
        msg[nb]='\0';
        printf(msg);
    }
    unlink(argv[1]);
    return 0;
}

```

écrivain.c

```

/* Ecrivain sur un tube nommé : réalisation de rendez-vous avec le lecteur.
Chacun des deux essaie de créer le tube nommé par mkfifo. S'il existe déjà, on
continue bien sûr. Donc l'idée est juste de créer le tube par le premier
processus. On lit ligne par ligne.
*/
#include <sys/types.h> // tous appels

```

```

#include <sys/stat.h>//mkfifo, open
#include <stdio.h>// perror
#include <errno.h>//errno
#include <stdlib.h>
#include <string.h>
#include <unistd.h>//read, write
#include <fcntl.h>//open
#include <signal.h>

char nomtube[1024]; /* var globale car accédée dans gst */

void gst(int sig){ /* gestionnaire du signal SIGPIPE */
    unlink(nomtube); /* le lecteur s'est terminé */
    exit(3);
}

int main(int n, char * argv[]){
    if (n != 2){
        fprintf(stderr,"Syntaxe : %s nomduTube\n",argv[0]);
        exit(1);
    }
    if(mkfifo(argv[1], S_IRUSR|S_IWUSR|S_IROTH|S_IWOTH)==-1 && errno!=EXIST){
        perror("Erreur mkfifo - on continue !\n");
    } else {
        chmod(argv[1],0666); /* pour permettre aux écrivains d'écrire */
    }
    int tube=open(argv[1], O_WRONLY);
    if (tube == -1){
        perror("Erreur ouverture tube");
        exit(2);
    }
    strcpy(argv[1],nomtube); /* pour le gestionnaire */
    struct sigaction action;
    action.sa_handler = gst;
    sigaction(SIGPIPE, &action, NULL); /* si signal alors supp le tube */

    int K=1024;
    char msg[K+1],*res;
    printf(">");
    res=fgets(msg,K,stdin); /* res est NULL si EOF sinon msg */
    while(res){
        write(tube, msg, strlen(msg));
        printf(">");res=fgets(msg,K,stdin);
    }
    return 0;
}

```

Solutions des exercices du TD/TP 9

Solution 1 Pas besoin de commencer avec deux processus, sauf cas de mauvaise compréhension. Si P_{init} commence par déposer un caractère dans le tube, alors le premier processus P_1 qui fera une lecture obtiendra ce caractère, qui disparaîtra du tube. Tout autre processus qui cherchera à lire sera bloqué. Il y aura déblocage d'un processus lorsque P_1 aura redéposé un caractère à nouveau. Comme un seul processus sera débloqué, il y aura bien exclusion mutuelle.

Mais il reste quelques questions embêtantes à voir plus loin : est-ce que P_{init} doit rester (oui !), ne peut-on pas faire des ouvertures moins brutales qu'en lecture et écriture (oui aussi, mais attention, etc.).

Solution 2 Catastrophe : plus personne pour redéposer un caractère dans le tube. Comme tous ont ouvert en lecture et écriture, pas moyen de les avertir qu'il n'y a plus d'écrivain.

Solution 3 Rien de spécial, il ne fait qu'attendre donc n'attendra plus, même s'il est seul dans la file d'attente des lecteurs dans le tube, puisque le système va fermer tous les descripteurs qu'il a ouverts.

Solution 4 Lorsqu'aucun processus ne veut accéder, s'il n'y avait pas P_{init} , il n'y aurait plus personne avec un descripteur ouvert sur le tube. Il serait alors détruit. Ce qui est grave est la perte du caractère inclus ! En effet, la demande d'ouverture crée la structure *tube* en mémoire, donc il serait recréé dès qu'un processus le fait, mais alors, le tube serait vide et le premier processus coincé. Ainsi, le seul rôle (important) de P_{init} est de maintenir le tube en vie lorsqu'il n'y a personne.

Solution 5 Conséquence de la réponse précédente : tant qu'il y a un processus en cours d'accès et possiblement un processus au moins en attente, tout va bien. Mais si la file d'attente devient vide, impossible de continuer.

Solution 6 But de la question : il se trouve que les processus en attente de lecture sur un tube ne sont pas réveillés dans un ordre *fifo*. Ce qui veut dire qu'il peut y avoir famine, par exemple si cette file d'attente est gérée comme une pile (ça semble être le cas dans Linux - à vérifier si c'est dans tous, mais en tout cas, ça l'était encore sur l'avant dernière version dans les salles de TP ufr), ou selon les prérogatives mal connues de l'ordonnanceur. Ce qu'il faut faire en TP : générer tout ça, lancer plusieurs fois le même processus demandeur et utiliser des entrées-sorties clavier pour bloquer le processus accédant. Ensuite, afficher l'identité de chaque accédant.

Solution 7 Le problème est : comment est-ce que ça va être géré dans la table des fichiers ouverts du système. Ce qu'on sait : chaque processus va avoir sa propre entrée dans la TFOS (Table des Fichiers Ouverts du Système), et ces entrées vont pointer toutes sur le même ensemble de blocs du fichier. Comme un seul processus modifie à la fois ces blocs, le suivant va lire ce qui a été déposé par le précédent (ou pointer ailleurs), mais en aucun cas, une lecture ou écriture ne pourra être interrompue pour effectuer une lecture ou écriture par un autre processus. Donc ils peuvent adopter une stratégie quelconque, sans se gêner. Autant ouvrir au début et garder ce descripteur ouvert jusqu'à la fin.

Remarque : la synchronisation entre l'espace mémoire et le disque physique n'a rien à voir dans tout ça : il n'y a pas synchronisation forcée lorsqu'un processus a fermé, contrairement à ce que pensent quelques uns. Elle est souvent forcée lorsque tous les processus ont fermé, car dans ce cas, le sgf décide qu'il peut récupérer l'espace des blocs en mémoire, devenu disponible.

Solution 8 Selon l'opération qu'il fait, ça peut devenir catastrophique (cas d'écriture). Cette question sert à illustrer le fait que lorsqu'on utilise un moyen d'exclusion ou de synchronisation, on suppose que tous jouent le jeu de demander l'accès avant de rentrer en section critique.

Solution 9 Certains pensent qu'il suffit que P_{init} reinjecte un caractère dans le tube. Oui, mais quand ? Comment peut-il savoir que l'accédant a disparu sans laisser de caractère ? Donc les solutions ne sont pas simples :

1. P_{init} pourrait communiquer par un autre moyen avec chacun des processus : par exemple, un autre tube nommé, dans lequel ils déposeraient leur identité. Ensuite, chaque processus se verrait allouer un quantum de temps. Lorsque le processus termine normalement les opérations sur *fissa* il reinjecte le caractère et envoie à nouveau quelque chose dans le tube d'identification pour annoncer qu'il s'est terminé. Si P_{init} constate que le quantum est dépassé, sans avoir eu de retour de l'accédant, il injecte un caractère dans le tube. Noter déjà qu'il faut qu'il fasse des entrées-sorties non-bloquantes pour le pas rester bloqué sur la lecture du tube, ou qu'il se fasse reiller par une alarme personnelle. Peut-il être sûr que l'accédant a terminé (i.e qu'il a dépassé le quantun, sans rien dire ? En consultant la table des processus actuels, oui, sinon, non, mais encore une fois, on suppose que tous jouent le jeu. Donc que le processus ne reste pas plus longtemps que le quantun alloué. L'objectif est de coopérer, non de casser.
2. Plus compliqué, bien que d'apparence plus simple : P_{init} se met lui-même en attente sur le tube nommé. Mais comment se débloquer ? par une alarme par exemple. D'accord, mais quelle décision prendre ? On ne sait pas si on a avancé dans la file d'attente. Donc la bonne question serait : est-il possible de connaître l'état de la file d'attente d'un tube ? Je n'ai pas encore trouvé. Pas de réponse dans `fcntl()`, ni les trucs qui tournent autour, ni procfs pour l'instant...

3. Enfin, avec des entrées-sorties non bloquantes, on ne s'en sort pas mieux, car lorsque le tube est vide, on saura qu'il n'y a rien, mais ça peut être parce que le processus accédant s'est anormalement terminé, ou qu'il y a eu entre temps changement de processus, donc un caractère a été déposé et repris de suite...

Solution 10 Quel est le problème sans ce processus ? le besoin de l'existence continue du tube. Or, on n'en a besoin que lorsque des processus veulent y accéder. Peut-on le créer par le premier processus ? Oui, à condition de savoir qu'on est premier, en testant l'existence du fichier de type *p* (i.e en testant le résultat de *mkfifo*). Oui, mais ça suppose que le *dernier accédant* le détruit ! Comment savoir qu'on est dernier ? En gérant le signal *SIGPIPE*, de sorte à ce que si on le capte, ça veut dire qu'il n'y a plus de lecteur (si on a pensé à fermer son propre descripteur de lecture) pour le caractère qu'on vient de déposer. Ainsi, dans la gestion de ce signal on peut détruire le fichier de type *p*. Ouf, on tient une solution.

On peut aussi proposer un comportement simplifié : *P_{init}* ouvre le tube en écriture seulement. Il sera bloqué, tant qu'il n'y a pas au moins un demandeur. Le demandeur doit demander une ouverture en lecture. Alors *P_{init}* sera débloqué, déposera un caractère dans le tube. Ensuite, le système marche, à condition que *P_{init}* ne fasse pas d'entrée-sortie. C'est-à-dire que dans le pire des cas, il y aura un processus avec un descripteur ouvert et qui ne fera rien. Le tube ne sera pas détruit et le système mis en place peut continuer.

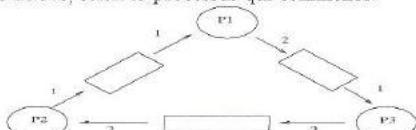
Ça vaut le coup de me mettre en œuvre en TP non ? Ne serait-ce que pour constater qu'il n'y a pas de gestion propre de la file d'attente...

Solution 11 L'idée principale ici est que l'on n'a pas besoin de savoir qui arrive premier, puisqu'on attend les deux de toute façon. *P₂* va créer ces deux tubes, *T₁* et *T₂*. Il ouvre par exemple *T₁* en écriture. *P₁* va ouvrir *T₁* en lecture lorsqu'il est lancé. Entre temps, soit *P₃* est arrivé, a ouvert *T₂* en lecture par exemple et reste bloqué, soit il n'est pas encore là. Dans le premier cas, il sera débloqué par l'ouverture par *P₂*. Dans le second cas, c'est *P₂* qui sera bloqué en attendant *P₃*.

Pour fixer les idées, on peut donc décider que *P₂* est celui qui écrit dans les deux tubes, et le rendez-vous sera réalisé lorsque chacun des deux processus aura reçu un premier caractère écrit par *P₂* dans chacun des tubes. En d'autres termes, une fois que *P₂* a été entièrement débloqué, il sait que les deux acolytes sont là et il envoie alors un caractère dans chacun des tubes. Le jeu peut commencer.

Solution 12 Non, pas de signaux entre processus appartenant à des utilisateurs différents (piège classique).

Solution 13 Oui, chaque processus ayant un tube de communication avec chacun des deux autres. Mais attention à l'ordre d'ouverture dans chacun. À part ça, avec la même remarque que ci-dessus, tant que tous les trois ne sont pas là, il y aura blocage arbitraire de ceux qui sont arrivés les premiers. Concernant l'ordre d'ouverture, il faut qu'un des trois ouvre en premier un des tubes en écriture, sachant qu'il est ouvert par l'acolyte en lecture... Après, plusieurs solutions possibles. Dans le schéma suivant on a numéroté un ordre possible pour chaque processus. Mais il y a plusieurs solutions, selon le processus qui commence.



Ici, tout ce qu'on dit est que *P₁* et *P₃* vont faire en première opération l'ouverture en lecture, *P₂* faisant celle en écriture d'abord.

Solution 14 Ça demande des développements non négligeables et comme le temps est compté, on laisse tomber pour l'instant cet exercice... En tout cas, les solutions simples sont souvent inéquitables du type autoriser au plus quatre personnes à s'installer à table à la fois (ça ne veut pas dire manger ensemble...), ou numérotter les philosophes et obliger un philosophe impair à prendre d'abord sa fourchette gauche et réciprocement. Bientôt une solution correcte, complète et lisible, demandez à Floréal...

Solution 15 On va trouver une situation et vérifier les quatre conditions :

1. Accès en *exclusion mutuelle* de chaque ressource (les ressources ne peuvent être partagées) ;
2. *tenir et attendre* : un processus au moins a réquisitionné une ressource et attend au moins une autre ;
3. *pas de préemption* des ressources (une ressource ne peut être libérée que par le processus qui la détient) ;
4. *attente circulaire* : il existe une suite P_0, P_1, \dots, P_n de processus en attente tels que P_0 attend une ressource détenue par P_1, \dots, P_i attend une ressource détenue par P_{i+1}, \dots, P_n attend une ressource détenue par P_0 .

Noter que la situation 4 implique la situation 2. En fait, Silberschatz propose de les séparer car c'est utile dans le cas où on veut faire de la prévention, mais là on peut se contenter de vérifier 4 seule.

La solution proposée consiste à ce que chaque agence obtienne la première ressource et attende la seconde, qui elle est détenue par une autre. Par exemple, on ne peut demander une réservation d'hôtel *éphone* que si personne d'autre n'est en cours de réservation.

C'est dans ce but que dans l'exercice il y a la remarque : Noter que toute agence vous garantit que toute réservation mixte demandée ne devient effective que lorsqu'elle a bien réussi à obtenir l'ensemble des réservations demandées.

Ça donne :

1. L'agence *airien* tient la ressource *train auto au départ de Montpellier vers Calais* et attend la réservation du *ferry pour la traversée de la Manche* ;
2. l'agence *elmarin* tient la ressource *ferry* et attend l'*hôtel you* ;
3. l'agence *aiplu* tient la ressource *hôtel you* et attend l'*entretien exclusif avec les frères William (Shake et Speare)* ;
4. l'agence *eignant* tient l'*entretien exclusif avec les frères William (Shake et Speare)* et attend la *semaine de relaxation à Montpellier, hôtel ypense* ;
5. l'agence *atyr* tient la *semaine de relaxation dans l'hôtel ypense* et attend le *train auto jusqu'à Calais*.

Noter que pour ce qui concerne l'agence *ottise*, elle n'a pas d'influence dans ce cas.

On constate que cette situation est indépendante du nombre de machines qui participent aux réservations. En fait, dans le cas distribué, il est beaucoup plus difficile de détecter le verrou fatal.

Enfin, pour débloquer la situation, on peut discuter des possibilités consistant à :

- garantir qu'un blocage ne peut avoir lieu (graphe d'allocation et recherche systématique avant chaque allocation qu'elle n'engendre pas un cycle par exemple),
- une détection de l'état de blocage et récupération par violation d'une contrainte comme la préemption forcée d'une ressource,
- supprimer un des processus (violent),
- compter sur les personnes qui attendent trop longtemps derrière un guichet et abandonnent la partie dans ce cas (donc un processus est tué «manuellement»), mais ce n'est adapté qu'à ce type d'exemples.