Systèmes d'exploitation - TD/TP 1
programme et processus, environnement d'exécution

Michel Meynard

17 septembre 2009

1 Programmes et données

Soit le programme C suivant : `tailledonnees.c`

```
#include <stdio.h>/* printf */
#include <stdlib.h>

#define K 100
const int KBIS=100;
int tab[K];
static float f;

int main(int argc, char *argv[]){
  int l;
  char*s[]={"un","deux"};
  static int j=2;
  int *td=(int*)malloc(5*sizeof(int));
  printf("nom \t taille\t valeur\t adresse\n");
  printf("%s \t %d \t %d \t %p \n","l",sizeof(l),l,&l);
  printf("%s \t %d \t %p \t %p \n","s",sizeof(s),s,&s);
  printf("%s \t %d \t %p \t %p \n","s[0]",sizeof(s[0]),s[0],&(s[0]));
  printf("%s \t %d \t %d \t %p \n","argc",sizeof(argc),argc,&argc);
  printf("%s \t %d \t %p \t %p \n","tab",sizeof(tab),tab,&tab);
  printf("%s \t %d \t %d \t %p \n","tab[1]",sizeof(tab[1]),tab[1],&(tab[1]));
  printf("%s \t %d \t %-5.2f \t %p \n","f",sizeof(f),f,&f);
  printf("%s \t %d \t %d \t %p \n","j",sizeof(j),j,&j);
  printf("%s \t %d \t %p \t %p \n","td",sizeof(td),td,&td);
  printf("%s \t %d \t %d \t %p \n","td[0]",sizeof(td[0]),td[0],&(td[0]));
}
```

Soit l'affichage provoqué par l'exécution de ce programme :

```
$ ./tailledonnees
nom      taille  valeur  adresse
l        4       4199692         0x22cce4
s        8       0x22ccd8        0x22ccd8
s[0]     4       0x403004        0x22ccd8
argc     4       1       0x22ccf0
tab      400     0x404030        0x404030
tab[1]   4       0       0x404034
f        4       0.00    0x404010
j        4       2       0x402000
td       4       0x660180        0x22ccd4
td[0]    4       0       0x660180
```

Exercice 1 (TD)

1. Quelle est la taille de chaque objet sur cette machine ? Un objet est une variable ou une zone de données pointée.
2. En fonction des valeurs obtenues, que dire de l'initialisation des variables ?
3. L'espace mémoire d'un processus étant vu comme un ensemble de quatre segments (voir cours), dans quel segment est défini chaque objet ?
4. Quelle est la durée de vie de chaque objet ?

Exercice 2 (TD) Indiquez à chaque fois si les tableaux sont initialisés !

1. En supposant que $N1$ est une constante connue, comment peut-on créer un tableau d'entiers de dimension $N1$ dans la pile ?
2. En supposant que $N2$ est une variable dont on lit la valeur au clavier, comment peut-on créer un tableau d'entiers de dimension $N2$ dans la pile ?

3. En supposant que $N3$ est une variable dont on lit la valeur au clavier, comment peut-on créer un tableau d'entiers de dimension $N3$ dans le tas ?
4. Créer une matrice dans le tas de dimension $N2 \times N3$ où les valeurs de $N2$ et $N3$ sont lues au clavier. Comment accède-t-on aux éléments de la matrice ? Sont-ils initialisés ? Quelle est sa durée de vie ?
5. Si cette création se fait à partir d'une variable locale dans une fonction différente de *main()*, que faut-il faire pour préserver la matrice au delà de la fonction ?
6. Montrer comment on peut créer un tel objet dans la pile. Quelles sont les différences par rapport au même objet dans le tas (accès, durée de vie) ?

Exercice 3 (TD Processus infernal) Décrire ce qui se passe lorsqu'on lance l'exécution d'un programme qui boucle indéfiniment, dans un système mono-programmation, puis multiprogrammation. Comment peut-on l'arrêter ? Par quels états passe un tel processus dans chacun des systèmes ?

Exercice 4 (TD allocation de l'UC) Un processus qui vient de perdre la ressource *Unité Centrale* peut la réobtenir de suite, sans qu'un autre processus utilisateur soit élu entre ces deux obtentions. Décrire deux scénarios possibles dans lesquels ce phénomène peut se produire.

Exercice 5 (TD mode d'exécution) Parmi les instructions suivantes, quelles sont celles qui doivent être exécutées en mode *noyau*?

masquer une interruption	lire la date du jour	modifier la date du jour
modifier l'allocation mémoire	appeler l'ordonnanceur	appel d'une fonction en C

2 Paramètres de la ligne de commande et environnement

Exercice 6 (TD/TP) On veut écrire un programme C `argenv.c` qui affiche sur la sortie standard, le nombre et la suite des paramètres de la ligne de commande ainsi que les variables d'environnement et leur nombre.

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

Exercice 7 (TD) En utilisant le programme précédent `argenv`, quel est le nombre de paramètres affiché par les exécutions suivantes :

- `argenv un deux 3`
- `argenv * $PATH`
- `argenv ~/.??*`

3 Répertoire d'exécution

Exercice 8 (TD/TP) Dans un processus, on veut savoir si le répertoire courant fait partie des répertoires d'exécution, c'est-à-dire s'il figure dans la variable d'environnement `PATH`. On utilisera les fonctions suivantes :

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);

#include <string.h>
char *strstr(const char *meule_de_foin, const char *aiguille);
char *strtok(char *str, const char *delim);
```

1. (TD)Ecrire l'algorithme ;
2. (TP) Ecrire le programme correspondant.

4 Fonctionnement de la pile

Exercice 9 (TD/TP Factorielle) Ecrire la fonction récursive factorielle et détailler l'exécution de `fact(3)` (TRACE).

5 Travaux pratiques ...

Exercice 10 (TD/TP Triangle de Pascal) On souhaite écrire une fonction qui calcule et mémorise un triange de Pascal de taille variable n. On rappelle que ce triangle fournit le nombre de combinaisons possibles de p éléments parmi n.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...
```

Le calcul de chaque élément est la somme de ses deux supérieurs à gauche !

1. Quelle structure de données choisir pour représenter ce triangle et quel type d'allocation choisir ?
2. Ecrire l'algorithme d'une fonction à un argument n qui fabrique cette structure et qui la renvoie.
3. Ecrire un programme C `cnp.c` à deux arguments n et p, qui affiche le triangle construit puis qui indique le nombre de combinaisons possibles : `C(12,2) = 66` par exemple.

Exercice 11 (TP Taille d'un programme) Faire un premier programme vide (ne contenant aucune instruction ni inclusion de fichier), le compiler et annoncer la taille de l'exécutable.

Recommencer en ajoutant successivement :
- des données globales (ou mieux des tableaux) non initialisées,
- ces mêmes données initialisées,
- des données qui iront dans la pile,
- des données qui iront dans le tas.

Exercice 12 (TP Pile d'exécution) Ecrire un programme dont la fonction principale appelle successivement deux fonctions $f1$ et $f2$. Dans la première, définir et initialiser un tableau de 10 entiers. Faire afficher les valeurs des éléments du tableau.

Dans la seconde définir un tableau d'entiers de même taille et afficher ses éléments.

1. Que constate-t-on lors de l'exécution ? Pourquoi ?
2. Recommencer en affichant les adresses des éléments pour confirmer les observations effectuées. Qu'en concluez-vous ?

Exercice 13 (TP Pile d'exécution bis) Ecrire un programme dont la fonction principale contient deux blocs. Dans le premier, définir et initialiser un tableau de 10 entiers. Faire afficher les valeurs et les adresses des éléments du tableau.

Dans le second définir un tableau d'entiers de même taille et afficher ses éléments et leurs adresses.

Que constate-t-on lors de l'exécution ?

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 initialisée, 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 fonctions.*

masquer une interruption	*oui*	lire la date du jour	*non*
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 8 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 9
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 8 \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 10
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("%6d ", tab[i][j]);
    }
    printf("\n");
  }
}
int main(int argc, char** argv, char** arge){
  if(argc!=3){
    printf("Syntaxe incorrecte : %s 8 4\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 11 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 12
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 13 idem exercice précédent

Systèmes d'exploitation - TD/TP 2
Chaîne de développement d'une application en langages C

Michel Meynard

17 septembre 2009

1 Du source à l'exécutable

Exercice 1 (TD) Nous avons pris l'habitude de dénommer *compilation* la transformation d'un programme source en exécutable. Cette transformation comporte plusieurs étapes. Quelles sont les différentes étapes de cette transformation ? Quel est le rôle de chacune ? Quelles sont ses entrées et sorties ?

2 Application jeu

L'annexe à cet énoncé contient le listing de 5 fichiers sources constituant une application de jeu (c'est un exemple tout à fait artificiel qui n'a aucun intérêt pratique). Le jeu consiste à deviner un nombre pseudo-aléatoire. En fin de jeu, une chaîne de caractères qui pourrait être un *mot de passe* est demandée à l'utilisateur et sa version cryptée est affichée à l'écran.

Liste des fichiers :

jeu.c source C contenant la fonction `main()` de l'application.

alea.c source C contenant une fonction d'initialisation du générateur de nombres pseudo-aléatoires en fonction de l'heure système.

alea.h fichier d'entête associé.

es.c module C fournissant une fonction d'entrée d'un entier depuis le fichier d'entrée standard et une fonction de sortie depuis le fichier de sortie standard.

es.h fichier d'entête associé.

Exercice 2 (TD/TP Compilation et édition de liens) On désire que l'exécutable s'appelle *jeu*. **Attention** : la fonction `crypt()` n'existe pas dans les bibliothèques standards utiliséees implicitement. Elle est implémentée dans la bibliothèque `libcrypt.a` (ou `.so`). Il faut donc engendrer la consultation de cette bibliothèque lors de la création de l'exécutable.

1. À quelle étape intervient la consultation de la bibliothèque `libcrypt.a` (ou `.so`) ?
2. Proposez une manière de réaliser la compilation de cette application en un seul appel au compilateur avec éditions de liens statique.
3. Proposez une manière de réaliser la compilation de cette application en un seul appel au compilateur avec éditions de liens dynamique.
4. Quel est l'inconvénient de la compilation globale en un seul appel ?

Exercice 3 (TD/TP Makefile et Compilation séparée) Proposez une manière de compiler l'application par une série d'appels au(x) compilateur(s) permettant de réaliser de manière séparée la compilation des différents fichiers.

Exercice 4 (TP La commande make) Indiquez les exécutions déclenchées par un appel `make jeu`. Si après avoir construit une première fois l'application *jeu*, on exécute la commande `touch alea.c`. Quel est l'effet de `make jeu` ?

Exercice 5 (TP Modifications dans les sources de l'application) Précisez l'effet sur la construction de l'application *jeu*, des modifications de fichier source proposées ci-dessous. La compilation se terminera-t-elle sans erreur(s) ? Sinon, precisez la nature des erreurs et à quel niveau de la chaîne de développement elles doivent se produire ?

1. Suppression de la directive `#include "es.h"` dans le fichier `jeu.c`
2. Remplacement de la directive `#include "es.h"` par la directive `#include <es.h>` dans le fichier `jeu.c` ;

3 À savoir

- Les divers sources sont à votre disposition sur le site `http://www.lirmm.fr:~meynard` dans la rubrique `L3/Système`
- il faut utiliser le manuel, mais aussi `info`. Par exemple `info ar`. Mieux, sous emacs menu *help-> manuals-> browse manuals with info* (si vous connaissez) ;
- Les options de compilation de gcc utilisées sont : `CFLAGS=-g -ansi -Wall -std=c99`
- L'option -L sert à l'option -l pour inclure d'autres répertoires à scruter pour les .o c'est-à-dire que si on a des bibliothèques qu'on veut scruter dans les répertoires standard, il suffit de les mettre dans -l, alors que si on a des répertoires différents des standards dans lesquels on a des bibliothèques, alors il faut à la fois mettre le répertoire dans -L et la/les bibliothèques dans -l.
- Afficher la table des matières de la bibliothèque de base `libc.a`.

4 Sources de l'application

4.1 jeu.c

```
// jeu : le but du jeu est de deviner un nombre entre 0 et 9

#include<stdio.h>
#include <stdlib.h>//pour random
#include <crypt.h>/* pour crypt */

#include "es.h"
#include "alea.h"

int main(){
  printf("JEU : DEVINEZ UN NOMBRE ENTRE 1 ET 9\n");
  initalea();
  int nb_a_deviner=(int)random()%9;
  int nb_devine=entnb();
  if (nb_devine == nb_a_deviner) {
    printf("Bravo !!\n");
  }
  else {
    printf("Erreur, l'entier a deviner etait :");
    sortienb(nb_a_deviner);
  }
  char mot[256];
  printf("Saisie du mot de passe : ");
  scanf("%s",mot);
  printf("\nmot saisi : %s \n",  mot);
  printf("mot crypté : ");
  printf(crypt(mot, "az"));
  printf("\n");
}
```

4.2 alea.h

```
/*
 * declaration de la fonction C initialisant le generateur de nombre aleatoire
 */

#ifndef _ALEA_H_
#define _ALEA_H_

void initalea();

#endif
```

4.3 alea.c

```
/*
 * implemente une fonction C initialisant le generateur de nombre aleatoire
 * en fonction de l'heure systeme.
 */
#include<time.h>
#include<stdlib.h>
#include"alea.h"

void initalea(){
  time_t tps;
  tps=time(0);  /* lecture de l'heure systeme
   nombre de secondes depuis le 01.01.70 */
  srandom((unsigned)tps); /* initialisation */
}
```

4.4 es.h

```
// gestion des E/S standard pour l'application jeu
// declarations
//
#ifndef _ES_H_
#define _ES_H_

int entnb();
void sortienb(int i);

#endif
```

4.5 es.c

```
// gestion des E/S standard pour l'application jeu
// implementation
//

#include"es.h"


// lecture d'un nombre entier depuis le fichier d'entree std
int entnb(){
  int nb;
  printf("Entrez un nombre : ");
  scanf("%d",&nb);
  return nb;
}
// ecriture d'un entier dans le fichier de sortie std
void sortienb(int i){
  printf("entier : %d\n", i);
}
```

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 commande 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 généré 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).

Systèmes d'exploitation - TD/TP 3
Entrées-Sorties de base

Michel Meynard

17 septembre 2009

1 Entrées-Sorties fichiers

Il s'agit de se familiariser avec les différentes possibilités de faire des entrées-sorties en C. On va s'efforcer tout d'abord de faire quelques manipulations simples sur les fichiers. Cette mise au point permettra de passer à la gestion des fichiers dans le cours.

Exercice 1 (TD) Quels sont les fichiers ouverts lors du lancement de tout processus ? Comment les changer depuis la ligne de commande ?

Exercice 2 (TD) Quels sont les moyens que vous connaissez pour créer des fichiers dits fichier *texte* (contenant les caractères ASCII compris entre 20_{16} et $7F_{16}$).

Exercice 3 (TD) Qu'est-ce qui distingue un fichier *texte* d'un fichier *binaire* ?

2 Appels systèmes

Exercice 4 (TD) Connaissez-vous les appels systèmes de base suivantes de manipulation des fichiers :
- ouverture, fermeture,
- lecture, écriture,
- positionnement,
- test de fin de fichier ?

Exercice 5 (TD/TP Comptage des caractères différents) On souhaite connaître le nombre de caractères différents présents dans un fichier. Par exemple, Si le fichier `toto.txt` possède le contenu suivant : `bbbabba`

Alors, le programme `compte` devra afficher ce qui suit :

```
compte toto.txt
2 caractères différents : a, b,
```

De même, Si le fichier `titi.txt` possède le contenu suivant : `allo olla`

Alors, le programme `compte` devra afficher ce qui suit :

```
compte titi.txt
4 caractères différents :  , a, l, o,
```

Le codage du fichier est un codage où chaque caractère est codé sur 1 octet (ISO-Latin1) et l'ordre d'affichage des caractères **n'a aucune importance**.

Questions

1. Expliquer **clairement** la ou les structures de données que vous comptez utiliser pour mémoriser le nombre de caractère : schéma + définition C ou C++.
2. Ecrire l'algorithme réalisant ce comptage.
3. Ecrire le programme C compte.c ;

Exercice 6 (TD/TP Nombre d'occurrences de caractères) Certains algorithme de compression (Huffman) nécessitent de connaître le nombre d'apparition de chaque caractère présent dans un fichier. Par exemple, Si le fichier `toto.txt` possède le contenu suivant :

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

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

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

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

Questions

1. Pourquoi la deuxième ligne de l'affichage est décalé d'un cran ?
2. Expliquer la ou les structures de données que vous comptez utiliser pour mémoriser le nombre de chaque caractère.
3. Ecrire l'algorithme réalisant ce comptage d'occurrences.
4. Ecrire le programme C `occurrences.c` ;

Exercice 7 (TD/TP hexl) La commande `hexl` permet de visualiser un fichier sous sa forme hexadécimale (dump hexa) et sous sa forme texte. Chaque ligne du dump est composé de la position en hexadécimal, de 16 codes hexa, de 16 caractères affichables. L'exemple suivant illustre le propos :

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

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

Exercice 8 (TD/TP Écriture sur un fichier) On souhaite réécrire la commande `cp source dest` qui permet de copier le fichier source dans le fichier destination.

Questions

1. Ecrire l'algorithme.
2. Ecrire le programme C `moncp.c` ;

Exercice 9 (TD/TP Positionnement dans un fichier) Soit un fichier contenant des caractères 8 bits uniques et triés dans l'ordre croissant de leur code ASCII comme dans l'exemple suivant :

```
15678ACFGJKJLMXYZabcduvw
```

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

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

Questions

1. Comment connaître la taille du fichier ?
2. Ecrire l'algorithme.
3. Ecrire le programme C `dicho.c` ;

3 Fonctions de bibliothèque

Exercice 10 (TD) Connaissez-vous les appels de fonctions de bibliothèque C suivantes de manipulation des fichiers :
- ouverture, fermeture,
- lecture, écriture,
- positionnement,
- test de fin de fichier ?

Exercice 11 (TD/TP) L'objectif est de comparer ce qui se passe lorsqu'on utilise des appels système et les fonctions de bibliothèque.

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

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 *binaire* 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** 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 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%x):%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("  ");
if(i%2) /* tous les 2 car, afficher un espace */
  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("."); /* non affichable */
    }
```

```
    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)
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); //printf("fin:%d; ",fin);
  if (fin==-1){
    printf("Impossible de se déplacer dans  %s !\n", argv[1]);
    return 3;
  }else if (fin==0){
    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.

Systèmes d'exploitation - TD/TP 4
Génération et recouvrement de processus

Michel Meynard

17 septembre 2009

1 Génération de processus

1.1 Une première

Exercice 1 (TD/TP) On veut voir un processus dit «parent» générer un autre dit «descendant», chaque processus affichant son identité (son numéro de processus) et celle de son parent. Pour obtenir les identités, utiliser les appels système ci-après.

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

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

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

       pid_t fork(void);
```

Exercice 2 (TD/TP Génération multiple) Soit le programme C suivant : **genproc4.c**

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

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

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

Exercice 3 (TD/TP Duplication des segments) Après clônage d'un processus par `fork()`, deux processus indépendants coexistent, et chacun a son propre contexte d'exécution, ses propres variables et descripteurs de fichiers. On veut mettre en évidence ce phénomène.

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

Exercice 4 (TP) Voici quelques exercices supplémentaires :

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

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

2 Recouvrement

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

Exercice 5
1. Écrire un petit programme, affichant un texte quelconque, puis se recouvrant par `ls` grâce à `execl` ; la commande `ls` est située généralement dans `/bin`.
2. Écrire un petit programme, affichant un texte quelconque, puis se recouvrant par `ls -l /bin` ; (utilisation de `execl()`)
3. (difficile) En supposant que vous ne connaissez pas l'emplacement de `gcc`, écrire programme, affichant un texte quelconque, puis se recouvrant par sa propre compilation ! On utilisera `execlp` pour mettre en œuvre la recherche de l'exécutable selon le contenu de la variable d'environnement `PATH`.

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

3 Compléments

Exercice 6
- Quelles différences y a-t-il entre un appel `system()` et `exec()` ?
- Quel est le nom du processus avant et après recouvrement ?

4 Questions extraites d'examens

4.1 Sur le recouvrement

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

Exercice 7 Pour le mettre en évidence, écrire deux très petits programmes (les plus petits possibles), montrant qu'avant et après recouvrement l'identité du processus est inchangée.

Exercice 8 Montrer en modifiant ou reécrivant vos programmes, qu'on peut avoir un nombre infini de recouvrements pour un même processus.

4.2 Sur les processus

Exercice 9 Proposer un algorithme qui étant donné un nombre entier $n > 2$ génère exactement n processus clônes.

Exercice 10 (TD/TP Mon Shell) On souhaite écrire un interpréteur de commande rudimentaire (sans argument).
1. Ecrire l'algorithme ;
2. Ecrire le programme correspondant.

Solutions des exercices du TD/TP 4

Solution 1 genproc1.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()) ;
    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());
    pid_t mortAnnoncee=waitpid(Pid,NULL,0) ;
    //pid_t mortAnnoncee=wait(NULL); //possible
    printf("du parent : mort annoncee de : %d \n",mortAnnoncee);
    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. chaque

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'exécutable (absolu ou relatif au répertoire dans lequel est lancé le processus), et `execlp()` n'a besoin que du nom de l'exécutable, 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 * constargv [], char * constenvp[]);

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'exéctuion 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(){
   cout<<''avant recouvrement, pid = ''<<getpid();
   execl(''/home/moi/P2'',''P2'',NULL)}
P2 : int main(){
   cout<<''après recouvrement, pid = ''<<getpid();
```

Solution 8 P2 se recouvre par P1 (ajouter une ligne execl 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);
}
```

Systèmes d'exploitation - TD/TP 5
Système de fichiers

Michel Meynard

17 septembre 2009

1 Quelques rappels : propriétaires et droits

1.1 droits d'accès et de suppression de fichiers

Exercice 1 (TD/TP)
1. Sous unix, quels sont les droits nécessaires pour pouvoir supprimer un fichier ?
2. Y a-t-il une relation entre le droit de supprimer un fichier et les droits d'accès à ce fichier ?
3. Quels sont les droits nécessaires pour modifier le contenu d'un fichier ?
4. Est-ce que le propriétaire d'un fichier peut s'enlever ses propres droits de lecture, écriture et exécution sur un fichier ? Est-ce grave ?

Exercice 2 (TD À préparer pour les TP) Créer des répertoires dans lesquels on met des fichiers `filasoi` et `filailleurs`.On peut choisir des noms plus exotiques. Donner les droits nécessaires à la suppression de fichiers, pour un collègue du même groupe et lui demander de supprimer `filasoi`. Faire en sorte qu'il ne puisse pas supprimer `filailleurs`. Des groupes correspondants aux groupes de td/tp ont été créés et devraient permettre de faire ces essais. Il faut utiliser la commande `newgrp`, à étudier.

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

1.2 droits sur les répertoires

Exercice 3 Que signifie le droit `x` sur un répertoire ? On veut donner à quelqu'un l'accès à un fichier en lecture et écriture, mais on ne veut pas qu'il puisse prendre connaissance du répertoire contenant. Comment faire ? Que peuvent faire tous les autres utilisateurs ?

Exercice 4 Quelles sont les possibilités offertes lorsqu'on a le droit d'écriture sur un répertoire ?

Exercice 5 Pour les TP, préparer un ensemble de manipulations, afin de mettre en évidence la signification des droits. Par exemple :
- enlever le droit de lecture sur un répertoire et chercher à modifier un fichier dans ce répertoire,
- enlever le droit d'exécution sur un répertoire et chercher à atteindre un fichier inclus,
- enlever le droit d'écriture sur un répertoire et supprimer un fichier inclus (qu'on soit propriétaire de ce fichier ou non),
- etc, à votre imagination.

2 Cohérences dans le système de gestion de fichiers

2.1 Un florilège [1] de questions d'examens

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

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

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

Exercice 6 Corriger les situations incohérentes et proposer pour chaque situation un ensemble cohérent.

1. *florilège, anthologie, recueil, morceaux choisis*, voici une belle *sélection* pour relever le défi de l'anglicisme *best of*.

2.2 Cohérence fichiers et répertoires

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

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

Exercice 7 À la seule vue de ce tableau de liens, peut-on dire si une entrée est un fichier simple ou un répertoire ?

Exercice 8 On propose maintenant deux organisations possibles de répertoires. Est-ce que chaque organisation est cohérente avec les données ci-dessus ?

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

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

3 Manipulations sur les fichiers

3.1 Appels système pour les droits d'accès

Exercice 9 La commande `chmod` et l'appel système `chmod()` existent. Quelle est l'utilité de chacun ?

Exercice 10 On admet que les droits d'accès à un fichier peuvent être modifiés pendant l'exécution du programme (voir plus loin, on réalisera ceci en Tp). D'où la question : si on peut modifier les droits en cours d'exécution, peut-on faire des opérations en contradiction avec les droits ? En quelque sorte, quand est-ce qu'une modification de droits devient effective ?

Voici un extrait de l'entête du manuel :

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

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

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

Exercice 11 Quelle est la différence entre ces deux appels ?

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

3.2 Détermination des droits d'accès

Attention : umask n'est pas traité dans le cours.

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

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

`(droits demandés) et (non umask)`

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

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

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

#ifndef _POSIX_SOURCE
#define S_IREAD         S_IRUSR
#define S_IWRITE        S_IWUSR
#define S_IEXEC         S_IXUSR
#endif

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

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

3.2.1 umask

Cette commande fait partie de l'interprète du langage de commande (shell), et on rappelle son fonctionnement à la fin de cette partie. La **commande** `umask` permet de déterminer les droits par défaut avec lesquels tout fichier sera créé. C'est bien un masque qui est mis en œuvre : `umask` *valUmask* provoque la prise en compte des droits de base `rw-rw-rw-` (666 en octal), puis on fait l'opération :

`(droits de base) et (non umask)`

Ce qui revient à retirer chaque élément correspondant à un 1 binaire de *valUmask*. Par exemple :

`umask` 022 crée les fichiers avec les droits `rw-r--r--`
`umask` 024 crée les fichiers avec les droits `rw-r---w-`
`umask` 066 crée les fichiers avec les droits `rw-------`

Exercice 12 Que faut-il indiquer dans les paramètres de l'appel système *open()*, pour créer un fichier accessible en écriture seulement au propriétaire, si `umask` vaut 222 ?

3.3 partage de fichiers

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

Exercice 13 Est-ce que deux processus peuvent partager un même fichier en lecture ? en écriture ? On sait que deux processus issus d'une même hiérarchie partagent les mêmes descripteurs de fichiers (`fork()` duplique les descripteurs). Que se passe-t-il lorsque les deux processus écrivent sur ce fichier ?

Exercice 14 Mettre en évidence deux processus qui écrivent sur un même fichier sans hériter des descripteurs du même père. Quelle est la différence avec la situation précédente ?

4 Liens durs et symboliques

4.1 numéros d'inodes et liens physiques (durs)

Exercice 15 Quelle commande permet de visualiser le numéro d'inode d'un fichier ? Et le nombre de liens (liens *physiques* dits *durs*) ?

Exercice 16 Partant d'un fichier et de son numéro d'inode, comment peut-on trouver l'ensemble des noms qui référencent cet inode ?

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

4.2 Une utilisation dangeureuse

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

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

Exercice 17

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

En TP - liens symboliques :

Exercice 18 Créer des répertoires temporaires et faire des liens symboliques sur ces répertoires, toujours par vous-même et par un collègue. Cette fois-ci peut-on savoir qu'un lien existe sur ce répertoire ? Que se passe-t-il :

- lorsqu'on supprime un lien ?
- lorsqu'on supprime le répertoire sans supprimer les liens ?

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

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 à l'ufr est celle de l'utilisateur seul dans son groupe. Aux dernières nouvelles, en accord avec les administrateurs, les étudiants sont enregistrés dans un groupe correspondant à leur groupe de td/tp. Si c'est vrai, on peut les faire changer de groupe, avec 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.cc`, un tout petit équivalent de la commande `rm` avec une analyse de l'erreur (utilisation de errno) (Jean-Yves, ça a changé depuis l'an dernier, il y est cette fois-ci). Autre solution, mais plus lourde et moins convaicante, 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.cc

```
/* un programme faisant la suppression du fichier passé en paramètre, tout en
affichant le type d'erreur rencontrée lorsqu'une erreur se produit.

C'est donc un programme de fonctionnalité semblable à la
commande rm en ajoutant le texte obtenu par errno; il utilise unlink().
*/

#include <iostream> //si erreur arguments
#include <stdio.h>
#include <errno.h>

using namespace std ;

int main(int nbarg, char *argh[]){
  if (nbarg < 1){
    cout <<argh[0]<< " : au moins un argument svp" <<endl;
    exit(0);
  }
  int ox=unlink(argh[1]);
  if (ox < 0){
    cout <<"erreur à la suppression de "<< argh[1]<<endl ;
    cout <<"erreur numéro :" << errno <<" signifiant : " ;
    //si on veut utiliser perror, penser a sauvgarder errno
    // car cette cochonnerie de perror la détruit !
    //int sverr=errno;
    //perror(NULL);
    cout <<strerror(errno)<<endl;
    //exemple d'utilisation plus sophistiqué
    if (errno == EACCES)
      cout <<" pas droit écriture dans rep contenant" <<endl;
  }
}
```

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 connait 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. Voir pour l'instant dans `~aro/Cours/Syst/L3/Td/Td05/accRep.cc`.

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 U_1, on peut y créer un sous-arbre que U_1 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 répetoire 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 a 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.

Voir le programme `tstchmod.cc`.

4.3 tstchmod.cc

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

Dans cet exemple, on ouvre avec le droit d''ecriture ; on peut continuer a
ecrire sur le fichier alors qu'avec chmod on l'enleve

Attention : c'est meme pire : si le fichier n'existe pas, on peut demander un
open avec des droits contradictoires a ceux qu'on va donner au fichier. De toute
facon, les droits ne seront effectifs qu'apres la fermerture, a l'open
suivant. On peut donc tester ce programme d'abord avec le fichier ajeter 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 g'ere les droits
du processus, et ceux du fichier pour les open suivants seulement (de ce
processus et des autres bien sur).

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

```
*/
#include <iostream> //e.s c++
#include <sys/stat.h>//pour chmod
#include<fcntl.h>//pour open
#include <unistd.h>//pour read write
using namespace std;

main(){
/*  retopen=open("ajeter", O_RDWR|O_CREAT,S_IRWXU|S_IRGRP); */
  int retopen=open("ajeter", O_RDWR|O_CREAT,S_IRUSR|S_IWUSR);
  if (retopen<=0){
    cout <<"pb ouverture"<<endl;
    exit(1);
  }
  cout <<"\n apres open avec droit ecriture" <<endl;
  int nbecr=write(retopen,"abcdefg",7);
  cout<<"\n on a 'ecrit" << nbecr << "caracteres" << endl;
  /* on laisse ici le temps de faire ls a la main mais on pourrait aussi lancer
     un system(), ou encore faire un acces 'a l'inode et afficher les droits. On
     fait comme si on avait plus confiance dans ls...
  */
  cout <<"\n on peut consulter les droits ; return ensuite svp"<<endl;
  cin.get();
  int retchmod=chmod("ajeter",S_IRUSR|S_IRGRP);
  if (retchmod !=0){
    cout <<"ca alors, pb au chmod !!!"<< endl;
    exit(1);
  }
  cout<<"\n apres chmod sur le fichier ouvert"<<endl;
  nbecr=write(retopen,"hijklmn",7);
  cout<<"on a ajoute" << nbecr <<"caracteres" << endl;
  cout<<"\n on vient d'ecrire dans le fichier encore"<<endl;
  cout<<"on attend a nouveau un return, apres consultation"<<endl;
  cin.get();
  close(retopen);//en toute rigueur, il faudrait tester le retour
  cout <<"fichier ferme"<<endl;
  cout <<"apres le return, on va encore modifier les droits"<<endl;
  cin.get();
  retchmod=chmod("ajeter",S_IRWXU);
  if (retchmod !=0){
    cout <<"ca alors, pb au chmod !!!"<< endl;
    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 catégorires d'utilisateurs.

Solution 13 Oui. Oui, pour les deux questions sur le partage.Les écritures se mélangent (mais y sont toutes) ou s'éffacent les unes les autres, selon le mode de partage (héritage ou écriture dans des endroits distincts du ficher 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 systeme 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 répetoires et fichiers inexistants. 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...

Systèmes d'exploitation - TD/TP 6
Système de fichiers : accès aux inodes et répertoires

Michel Meynard

17 septembre 2009

1 Taille des fichiers

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

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

2 Accès à la table des inodes

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

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

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

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

              struct stat
              {
                  dev_t         st_dev;      /* Périphérique                */
                  ino_t         st_ino;      /* Numéro i-noeud              */
                  umode_t       st_mode;     /* type de fichier et droits   */
                  nlink_t       st_nlink;    /* Nb liens matériels          */
                  uid_t         st_uid;      /* UID propriétaire            */
                  gid_t         st_gid;      /* GID propriétaire            */
                  dev_t         st_rdev;     /* Type périphérique           */
                  off_t         st_size;     /* Taille totale en octets     */
                  unsigned long st_blksize;  /* Taille de bloc pour E/S     */
                  unsigned long st_blocks;   /* Nombre de blocs alloués     */
                  time_t        st_atime;    /* Heure dernier accès         */
                  time_t        st_mtime;    /* Heure dernière modification */
                  time_t        st_ctime;    /* Heure dernier changement état*/
              };
```

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

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

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

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

Notons que les droits d'accès peuvent également être consultés grâce à une famille de masques tels que :
- `S_IRUSR` pour la lecture par le propriétaire ;
- `S_IWGRP` pour l'écriture par le groupe ;
- `S_IXOTH` pour l'exécution par les autres ;

Exercice 3 (TD et TP)
1. Écrire un programme `typeFichier.c` permettant d'afficher si un nom donné en paramètre est un fichier régulier, un répertoire, un lien ou est d'un autre type.
2. Écrire un programme `droitFichier.c` permettant d'afficher les droits d'accès du fichier correspondant au format de `ls -l` : `-rw-r-r-` ou `drwx-r-xr-x` ou `lr-xr-x--`.

3 Opérations sur les répertoires

3.1 Fonctions spécifiques

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

1. ouverture/fermeture d'un répertoire :

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

2. lecture de l'enregistrement suivant :

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

3. création/destruction d'un répertoire :

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

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

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

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

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

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

3.2 Contenu d'un répertoire

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

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

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

Exercice 5 (TD et TP) Reprendre l'exercice 3, afin d'afficher (monls.c) le contenu d'un répertoire au format d'un `ls -ali` : inode type droits nomFichier

3.3 Gestion des dates

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

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

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

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

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

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

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

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

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

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

Exercice 6 (TD et TP) Écrire un programme `lsmodif.c` permettant d'afficher la liste des fichiers et répertoires d'un répertoire donné en argument qui ont été modifiés depuis moins de n jours, n étant donné en argument. On affichera le nom du fichier et la date de modification. Par exemple, `lsmodif . 2` affichera la liste des fichiers récement modifiés (moins de 2 jours).

3.4 Parcours récursif d'un sous-arbre

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

Exercice 7 1. Ecrire un algorithme de cette fonction récursive `parcours(rep)` en supposant que tout les appels systèmes se passent bien !

2. Écrire un programme `arboindent.c` permettant d'afficher récursivement la liste indentée des noms de répertoires d'un répertoire donné en argument.

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

3.5 Sauvegarde incrémentale

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

Exercice 8 1. On se restreint à un et un seul répertoire pour l'instant. Quelle solution peut-on proposer pour qu'un tel système fonctionne automatiquement ?

3.6 Question subsidiaire : nom et numéro de propriétaire

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

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

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

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

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

Exercice 9 (TD) Quelles justifications peut-on fournir à une telle démarche ? Peut-on citer au moins un inconvénient ?

4 Autres appels noyau du système de fichiers

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

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

Pour les TP, on préparera des petits programmes permettant de se rendre compte de l'effet des appels cités ci-dessus. Par exemple :
- Essayer `chdir(2)` dans un répertoire inexistant et afficher l'erreur système récupérée.

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

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 denière modif (accès en écriture ou création),
   - `st_ctime` est la date de dernière modif de l'inode seul (droits, proprio, 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 liens symboliques et donne l'état du fichier lien.

Solution 3 1. `typeFichier.c`

```
/*  test le type d'un fichier ou repertoire */
#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 repertoire 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 write 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 repertoire 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 */
      strcat(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 repertoire */
#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 insteger */
  DIR * rep=opendir(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){
    strcat(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 recursif  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=opendir(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 etre embete 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("  "); /* indentation */
      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 (totale 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 constant 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 ne pas transporter l'identité (nom ou numéro) lorsqu'on transporte un système de fichiers ?

Systèmes d'exploitation - TD/TP 7
Signaux

Michel Meynard

17 septembre 2009

1 Modification du gestionnaire d'un signal

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

On rappelle l'appel système `sigaction` :

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

Et voici la structure `sigaction` :

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

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

Exercice 1 (TD) Comment faut-il faire pour envoyer un signal SIGINT à un processus ?

Exercice 2 (TD/TP) Écrire trois programmes permettant d'essayer toutes les possibilités de gestion d'un signal et constater qu'on peut :

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

1.1 Configuration du terminal

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

2 Signal SIGALRM

La primitive définie par :

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

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

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

Programmer les différentes solutions suivantes :

Exercice 3 (TD/TP)
1. le programme s'interrompt automatiquement si l'utilisateur n'a pas répondu dans les 10 secondes en affichant un message "Trop tard !".
2. Le programme demande à l'utilisateur de se presser si ce dernier n'a pas répondu dans les 10 secondes, en lui donnant une nouvelle chance.

3. Au cours de l'attente le programme envoie un message de plus en plus insistant à l'utilisateur (par intervalles raccourcis : 10, 5, puis 3 secondes), puis finit par s'interrompre si ce dernier n'a toujours pas répondu.

Exercice 4 (TD) Expliquer pourquoi le temps en `sec` passé en paramètre à la primitive ne peut être qu'approximatif.

3 Erreurs système

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

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

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

Exercice 5 (TD/TP) Ecrire un programme générant une erreur de segmentation. Gérer le signal correspondant en effectuant l'affichage complet de l'erreur produite. Qu'en déduisez-vous ?

4 Signal de fin d'un processus fils

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

On envisage un programme à réaliser en TP, dont le processus correspondant répond aux caractéristiques suivantes :
- il génère successivement deux processus enfants,
- il utilise son propre gestionnaire de signaux, afin de récupérer le signal `SIGCHLD`,
- puis il attend la réception des deux signaux, correspondant à la fin respective de chaque enfant.

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

Exercice 6 (TD/TP) Commenter chacun de ces cas et expliquer comment chaque situation décrite peut se produire.

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

```
/* 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("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("Recu 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("ooouuups : %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 *zombis* 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 zombis restent dans cet état longtemps, même après la fin du parent ? Car ils attendant 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 suffisemment *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.

Systèmes d'exploitation - TD/TP 8
Tubes non nommés

Michel Meynard

17 septembre 2009

1 Taille d'un tube

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

Exercice 1 (TD/TP) Quelle méthode envisagez-vous pour connaître cette taille ? Écrire le programme correspondant à la méthode préconisée pour déterminer cette capacité.

2 Tube et fin de fichier

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

On envisage successivement les cas suivants :

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

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

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

Exercice 2 (TD) Étudier ces **trois** cas dans **chacune** des questions suivantes :

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

Exercice 3 (TD) Du côté du lecteur, si le nombre de caractères à lire est inconnu, combien de caractères sont lus et en combien de d'appels à la primitive `read()` ?

3 Synchronisation

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

Exercice 4 (TD)

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

4 Version parallèle du crible d'Eratosthène [1]

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

Nous allons étudier une version parallèle sous UNIX de ce crible où chaque crible est un processus qui lit une liste d'entiers dans un tube, qui affiche le premier p, et envoie les entiers qu'il n'a pas filtrés dans un autre tube. On aura

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

ainsi une chaîne de processus communiquant par des tubes. On envoie la liste des entiers dans le tube initial et chaque processus `crible(i)` affiche à l'écran le premier entier qu'il reçoit, puis parmi les entiers qu'il reçoit par la suite, il élimine les multiples du premier entier reçu et écrit dans le tube suivant les autres entiers. Nous présentons ci-dessous le détail de l'algorithme exécuté par le processus crible numéro *i*.

Algorithme :fonction **crible(in)**

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

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

5 Comptage de caractères

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

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

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

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

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

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

Exercice 6 (TD/TP)
1. Pourquoi la deuxième ligne de l'affichage est décalé d'un cran ?
2. Ecrire l'algorithme réalisant ce comptage.
3. Ecrire le programme C `compte.c` et testez le sur le fichier source ;

6 Dialogue entre deux utilisateurs – tubes nommés

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

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

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

Exercice 7 (TD/TP)

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

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

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 écite! (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 connait 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 connait 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 connait 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 connait 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 ; // N° 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 ; // N° 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. `criblexec.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 ; // N° 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 ; // N° 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!

2.

Algorithme :fonction **compte(in)**

Données : *in* : tube entrant

début

- fermer(*in*[*ecrire*])
- char C
- **si** *0 != (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**
          - ecrire(out[ecrire],i)
      - fermer(*in*[*lire*])
      - fermer(out[ecrire])
      - afficher(C :nb)
    - **sinon**
      - afficher "Erreur du fork()"

fin

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 ; // N° 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 ; // N° 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  l: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  ˚:2
k:10  ::9  è:2  +:2  %:2  !:1  S:1  \:1  O:2  R:1  D:1  L:2  Y:1  ’:1  é:1  ê: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!=EEXIST){
    perror("Erreur mkfifo - on continue");
  }
  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;
}
```

`ecrivain.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
```

```c
#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];

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!=EEXIST){
    perror("Erreur mkfifo - on continue");
  }
  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=gets(msg); /* res est NULL si EOF */
  while(res){
    write(tube, msg, strlen(msg));
    printf(">");res=gets(msg);
  }
  return 0;
}
```

Systèmes d'exploitation - TD/TP 9
Tubes, Exclusion et Synchronisation de Processus

Michel Meynard

17 septembre 2009

1 Exclusion Mutuelle et Tubes Nommés

Dans ce problème on veut utiliser un tube nommé et les possibilités de synchronisation sur ces tubes, comme moyen de protéger l'accès à une ressource commune. Pour fixer les idées, on admet que la ressource commune est un fichier que plusieurs processus veulent accéder en lecture et écriture.

Principe de fonctionnement :

Appelons le tube nommé *tubino* et le fichier *fissa*. Un premier processus P_{init} ouvre *tubino* en lecture **et** écriture. Dans la suite, il effectue un nombre très réduit d'opérations ; en particulier, il n'accède jamais lui-même à *fissa* ; néanmoins, il reste en vie constamment.

Tout processus (autre que P_{init}) qui prévoit d'accéder à *fissa*, ouvre au départ *tubino* en lecture **et** écriture. Ensuite, il doit lire dans le tube un caractère. Cette lecture, lui donne l'autorisation d'accéder à *fissa* en lecture et écriture. Après modification du fichier, il redépose dans le tube nommé un caractère et ferme le tube.

Noter que tout processus peut effectuer plusieurs fois des demandes d'accès à *fissa* et qu'il sera amené à suivre cette procédure à chaque demande d'accès. Donc un accès est constitué d'un ensemble de lectures et écritures sur *fissa*. Chaque processus peut effectuer un nombre indéterminé de demandes d'accès.

Remarque : Préciser ici que chaque processus peut faire plusieurs demandes d'accès pouvant comporter plusieurs lectures et écritures dans le fichier. C'est juste pour expliquer le texte. Le fait qu'il fasse un ou plusieurs accès au fichier ne change rien.

Fin Remarque.

Exercice 1 Montrer que ce principe de fonctionnement est correct, c'est-à-dire qu'il garantit l'exclusion mutuelle entre les processus demandant l'accès à *fissa*. Ne pas oublier l'initialisation. On pourra commencer avec deux processus puis généraliser.

Exercice 2 Que se passe-t-il si le processus en cours d'accès à *fissa* meurt prématurément ?

Exercice 3 Que se passe-t-il si un des processus n'ayant pas encore obtenu le droit d'accéder à *fissa* meurt ?

Exercice 4 Analyser ce qui se passe lorsqu'aucun processus ne veut accéder à *fissa* ; en déduire le rôle que remplit P_{init}.

Exercice 5 Que se passe-t-il si P_{init} meurt ?

Exercice 6 Lorsque plusieurs processus demandent à accéder à *fissa*, on va se trouver avec tous ces processus sauf un en attente. Ne connaissant pas l'ordre dans lequel ces processus en attente vont être réveillés, on se demande s'il peut y avoir famine. Donner au moins deux exemples aboutissant à une telle situation, puis, en prévision des TP, proposer une solution permettant de vérifier si le système d'exploitation utilisé est équitable ou non dans ce cas de figure.

Exercice 7 On veut décider si chaque processus peut ouvrir *fissa* au début de son lancement et ne le fermer qu'à la fin, ou si au contraire, il ne doit ouvrir le fichier que lorsqu'il a obtenu le droit d'accès et fermer avant de passer son tour. Afin d'étudier correctement cette question, il faut envisager tous les cas possibles, c'est-à-dire ceux où tous les processus adoptent une de ces deux démarches, puis le cas où certains processus utilisent la première et d'autres la deuxième démarche.

Exercice 8 Que se passe-t-il si un processus décide de ne pas passer par *tubino* pour accéder à *fissa* ?

Exercice 9 Peut-on proposer une solution au phénomène constaté dans la question 2 ?

Exercice 10 Peut-on proposer un fonctionnement sans P_{init} ?

2 Rendez-Vous Multiples

Trois processus P_1, P_2, P_3 appartenant à trois utilisateurs différents veulent se donner rendez-vous (par exemple pour démarrer un jeu).

On a vu en cours que deux processus pouvaient se donner rendez-vous en ouvrant un tube nommé, l'un en lecture, l'autre en écriture.

Pour passer de deux à trois processus, on décide d'utiliser deux tubes nommés, un pour P_1 et P_2, l'autre pour P_2 et P_3.

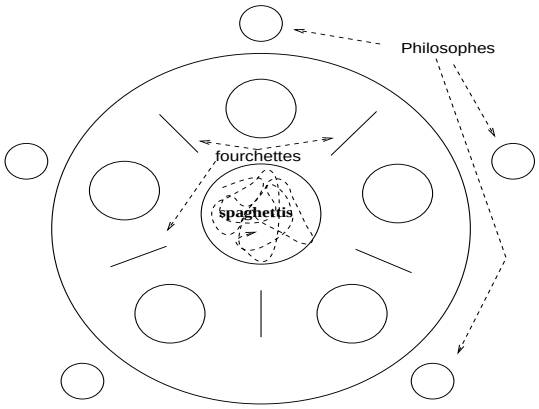
Exercice 11 P_2 va jouer le rôle de charnière commune. Dans cette question, on ne s'intéresse qu'au démarrage, c'est-à-dire tout ce qui se passe pour chaque processus, à partir de son lancement, jusqu'au moment où il peut enfin aller plus loin que le point de rendez-vous. Montrer comment une telle solution fonctionne.

Exercice 12 Est-il possible d'utiliser un seul tube nommé, entre P_1 et P_2, et un signal que P_3 enverrait à P_2 ?

Exercice 13 Si on ne veut pas qu'un processus ait le rôle central, existe-t-il une solution mettant en œuvre des tubes nommés ?

3 Les philosophes de Dijkstra

Impossible de ne pas citer ce problème tant il a marqué et marque des générations d'informaticiens :



Cinq philosophes sont assis autour d'une table ronde, une grande assiette de spaghettis au milieu, une assiette personnelle devant chaque philosophe, et une fourchette entre deux assiettes (voir dessin). Si chaque philosophe disposait de deux fourchettes, il n'y aurait plus de problème. Donc il n'y a qu'une fourchette entre deux assiettes, et chaque philosophe passe son temps à penser ce qui le conduit à avoir faim et alors il tente de s'emparer des deux fourchettes situées à gauche et à droite de son assiette, mais il ne peut les prendre que l'une après l'autre (i.e. symbolise le fait qu'on accède aux ressources une par une). S'il y arrive, il mange et une fois rassasié, repose les deux fourchettes (l'une après l'autre) et retourne à l'activité de penser.

Ce problème est une modélisation des problèmes d'allocation de ressources et permet d'illustrer des problèmes de famine et de verrou fatal. Pouvez-vous établir une correspondance entre ces cas et les philosophes ?

Exercice 14 Pouvez-vous proposer une solution évitant toute famine et tout verrou fatal ? Vous pouvez donner plusieurs solutions, mais au moins une avec des sémaphores est demandée.

4 Verrou fatal

Voici une suite d'offres de diverses agences de voyages. Il faut trouver une situation où un verrou fatal apparaît. Prouver que cette situation correspond bien à un tel verrou et proposer une solution extérieure pour débloquer la situation.

1. L'agence *airien* propose des réservations mixtes, de train auto au départ de Montpellier vers Calais, avec des réservations du ferry pour la traversée de la Manche ;
2. l'agence *aiplu* propose des réservations dans l'hôtel *you* à Londres et à la demande, un entretien exclusif avec les frères William (Shake et Speare) pendant ce séjour ;
3. l'agence *elmarin* propose des réservations au départ de Calais, dans le ferry associées à des réservations dans l'hôtel *you*, toujours à Londres ;
4. l'agence *atyr* propose des semaines de relaxation dans l'hôtel *éphone* à Montpellier, avec le retour en train auto jusqu'à Calais ;
5. l'agence *eignant* propose plutôt des entretiens exclusifs avec les frères William (Shake et Speare) suivi d'une semaine de relaxation à Montpellier, hôtel *éphone* ;

6. l'agence *ottise* propose une réservation dans l'hôtel *deville* à Montpellier, associée à un entretien exclusif du vizir Iznogoud.

Important : 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.

Exercice 15 Montrer que la situation de verrou fatal est possible que les réservations soient faites sur une machine centralisée (une machine plusieurs bases ou fichiers de données) ou de façon décentralisée, comportant plusieurs machines et des bases ou fichiers distants.

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 répones 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 prérogative 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 q'il suffit que P_{init} reinjecte un caratère 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.

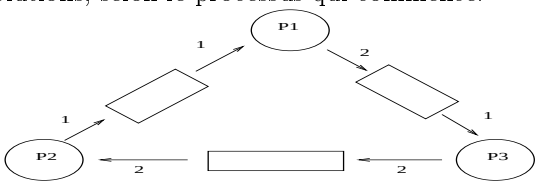
Ça vaut le coup de me mettre en œuvre en TP non ? Ne serait-ce que pour constater qu'il n'y a pas de gestion propre de la file d'attente...

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. P_2 va créer ces deux tubes, T_1 et T_2. Il ouvre par exemple T_1 en écriture. P_1 va ouvrir T_1 en lecture lorsqu'il est lancé. Entre temps, soit P_3 est arrivé, a ouvert T_2 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 P_2. Dans le second cas, c'est P_2 qui sera bloqué en attendant P_3.

Pour fixer les idées, on peut donc décider que P_2 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 P_2 dans chacun des tubes. En d'autres termes, une fois que P_2 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 numéroter les philosophes et obliger un philosophe impair à prendre d'abord sa fourchette gauche et récpiproquement. 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 $P_0, P_1, \ldots, P_n$ de processus en attente tels que P_0 attend une ressource détenue par P_1, ... P_i attend une ressource détenue par P_{i+1}, ..., 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 vérifer 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.