Concepts et programmation système (HLIN504) - TD/TP 1
programme et processus, environnement d'exécution

Michel Meynard

6 septembre 2018

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 i=1;int j=2;i=i+j;}
  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 %lu \t %d \t %p \n","l",sizeof(l),l,&l); // %lu long unsigned
  printf("%s \t %lu \t %p \t %p \n","s",sizeof(s),s,&s);
  printf("%s \t %lu \t %p \t %p \n","s[0]",sizeof(s[0]),s[0],&(s[0]));
  printf("%s \t %lu \t %d \t %p \n","argc",sizeof(argc),argc,&argc);
  printf("%s \t %lu \t %d \t %p \n","KBIS",sizeof(KBIS),KBIS,&KBIS);
  printf("%s \t %lu \t %p \t %p \n","tab",sizeof(tab),tab,&tab);
  printf("%s \t %lu \t %d \t %p \n","tab[1]",sizeof(tab[1]),tab[1],&(tab[1]));
  printf("%s \t %lu \t %-5.2f \t %p \n","f",sizeof(f),f,&f);
  printf("%s \t %lu \t %d \t %p \n","j",sizeof(j),j,&j);
  printf("%s \t %lu \t %p \t %p \n","td",sizeof(td),td,&td);
  printf("%s \t %lu \t %d \t %p \n","td[0]",sizeof(td[0]),td[0],&(td[0]));
  int (*h)(int,char**)=main; /* pointeur sur fonction */
  printf("%s \t %lu \t %p \t %p \n","h",sizeof(h),h,&h);
  // char *ts[]={"toto"};  h(1,ts); // A NE PAS FAIRE : récursiv. infinie
}
```

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

```
$ ./tailledonnees
nom   taille  valeur         adresse
l       4       32767          0x7fff574eca5c
s       16   0x7fff574eca70 0x7fff574eca70
s[0]   8       0x108713f04  0x7fff574eca70
argc   4       1              0x7fff574eca6c
KBIS   4       100          0x10833bf00
tab   400   0x108714050  0x108714050
tab[1]   4       0              0x108714054
f       4       0.00           0x108714044
j       4       2              0x108714040
td       8       0x7fe7324039f0 0x7fff574eca50
td[0]   4       0              0x7fe7324039f0
h       8   0x10f3d4c50  0x7fff5082ba48
```

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 ?
5. Quelle est la visibilité (portée) de KBIS et de j ?

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 Déboguer une erreur de "segmentation fault"

Exercice 8 (TP)
1. Dans un éditeur de texte saisir le fichier segfault.c suivant : `segfault.c`

```
#include <stdio.h>
char *s;
void g(void){printf("%c\n",s[0]);} // génère signal SIGSEGV
void f(void){g();}
int main(){f();}
```

2. le compiler : `gcc -g -o segfault segfault.c` ; l'option g permet d'inscrire des informations de débogage dans le binaire ;
3. l'exécuter : `./segfault`
4. Le message affiché *Segmentation Fault (core dumped)* signifie :
   — une erreur d'accès à un segment dans la mémoire a eu lieu ;
   — un fichier binaire correspondant à l'image mémoire du processus a été créé sous le nom *core* ;
5. pour analyser le type de ce fichier core, lancer la commande :

   `file core`

   Si ce fichier n'existe pas, c'est que votre quota de fichier core est à 0. Vérifiez-le avec la commande : `ulimit -a`. Puis augmentez votre quota avec la commande : `ulimit -c 1000`. Passez en `sudo` si vous n'êtes pas administrateur.
6. pour savoir dans quel état était la pile au moment de l'arrêt, on lance le débogueur `gdb` sur le programme en lui fournissant également son fichier core : `gdb segfault core` ;
7. dans la session gdb, on lance `bt` (backtrace) qui affiche l'état de la pile au moment de l'erreur mémoire :

```
(gdb) bt
#0  0xfee847a9 in _ndoprnt () from /lib/libc.so.1
#1  0xfee88a6e in printf () from /lib/libc.so.1
#2  0x08050cdb in g ()
#3  0x08050ce8 in f ()
#4  0x08050cf5 in main ()
```

8. on a ainsi découvert dans quelle fonction l'erreur a eu lieu (dans printf appelée par g ...) ; si un vrai programme est découpé en petites fonctions, la connaissance de la fonction qui génère l'erreur permettra de résoudre rapidement cette erreur ;
9. si la fonction incriminée est longue et complexe avec des boucles imbriquées, etc, on utilisera le débogueur `gdb` de la façon suivante :

```
gdb segfault
(gdb) break g                 // point d'arret posé au début de g()
(gdb) run                     // lance le prg
Breakpoint 1, g () at segfault.c:3
3       void g(void){printf(s);}
(gdb) print s                 // on peut afficher des variables
$2 = 0x0
(gdb) next                    // exécuter l'instruction suivante
Program received signal SIGSEGV, Segmentation fault.
0xfee847a9 in _ndoprnt () from /lib/libc.so.1(gdb)
(gdb)
```

10. Il ne vous sera ainsi plus nécessaire de traquer les erreurs en écrivant puis supprimant des `printf()` pour traquer l'erreur ! Pour découvrir plus de commandes gdb, `man gdb` ou `info gdb` mais aussi dans gdb lui même, `help bt` vous indiquera ce que fait la commande backtrace. Découvrez donc step, list, cont, set var, watch, ...

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 ?

6 Répertoire d'exécution

Exercice 14 (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.

Concepts et programmation système - TD/TP 2
Chaîne de développement d'une application en langage C

Michel Meynard

6 septembre 2018

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 Mastermind

Le jeu de Mastermind consiste pour l'utilisateur à découvrir une combinaison secrète de 4 chiffres (0-9) choisis au hasard par l'ordinateur en répétant des

propositions de 4 chiffres et en récupérant à chaque fois un nombre de chiffres bien placés et un nombre de chiffres xdmal placés.

Deux versions de ce jeu, l'une graphique l'autre en mode console, vous sont fournies dans une archive située sur le site `www.lirmm.fr/~meynard`.

Voici le code du main : main.c

```
#include "mm.h"
#include <stdio.h>
#include <unistd.h>
/** Fonction principale de la version console du jeu Mastermind
 */
int main(){
  // printf("taille mm : %d; taille struct mm : %d\n", sizeof(mm), sizeof(struct mm));
  mm j=mm_creer(); // printf("%s\n",j->secret);
  char saisie[1024];
  int res, CONTINUER=1;
  printf("Vous devez tenter de découvrir une combinaison secrète de %d chiffres [0-9] en saisissant rép
  do {
    printf("?");
    scanf("%s",saisie);
    res=mm_test(j,saisie);
    if (res==-1)
      printf("Erreur de saisie !\n");
    else if (res==0)
      printf("Aucune lettre correcte !\n");
    else {
      printf("%d lettres bien placées, %d lettres mal placées !\n", res/(TAILLE+1),res%(TAILLE+1));
      if(res/(TAILLE+1)==TAILLE){
        printf("BRAVO ! Vous avez réussi en %d  essais !\n",mm_nbessais(j));
        CONTINUER=0;
      }
    }
  } while(CONTINUER);
  return 0;
}
```

Voici le code de mm.h : mm.h

```
#define TAILLE 4

/** Type pointeur sur une structure de jeu
 */
typedef struct mm* mm;
/** Structure de jeu contenant le mot secret ainsi que le nombre d'essais
```

```
 * effectués
 */
struct mm {
  char secret[TAILLE+1];  // mot secret stocké dans une chaine de char
  int nbessais; /* nombre d'essai */
};
/**
 * Crèe un nouveau jeu en générant aléatoirement un nouveau mot secret
 * composé de TAILLE lettres comprises entre 0 et 9.
 * @return un pointeur sur le jeu créé dans le tas (donc à detruire !)
 */
mm mm_creer();
/**
 * Supprime un jeu en désallouant la mémoire
 * @param mm un pointeur sur la structure de jeu
 */
void mm_detruire(mm);
/**
 * teste un mot essai face au mot secret stocké dans le jeu
 * @param jeu un pointeur sur la structure de jeu
 * @param essai la chaîne de caractères proposée par le joueur humain
 * @returns un entier contenant (TAILLE+1)*nb lettres bien placées + nb lettres
 * mal placées; -1 si l'essai est erroné (nb lettres, ...)
 */
int mm_test(mm jeu, char* essai);
/** Retourne le nb d'essais déjà effectués
 * @param jeu un pointeur sur la structure de jeu
 * @returns le nombre d'essais
 */
int mm_nbessais(mm jeu);
```

Exercice 2 (TD Compréhension du code)

1. Qu'est ce que `TAILLE` et pourquoi le mot `secret` contient-il `TAILLE+1` caractères ?
2. Quelle est la taille de la structure de jeu (`struct mm`) ?
3. `mm_test` devant retourner un couple d'entiers (nb de chiffres bien placés, nb de chiffres mal placés), quel est le codage utilisé pour agréger ces 2 entiers en un seul ?

Exercice 3 (TD compréhension du makefile)

Soit une partie du makefile permettant de construire l'application en mode console :

```
CC=gcc
CFLAGS=-Wall -std=c99

mm: mm.o mm.h main.o
	$(CC) $(CFLAGS) -o mm main.o mm.o

.c.o:
	$(CC) $(CFLAGS) -c $<
```

1. Qu'est ce que CC et CFLAGS ?
2. Quel est le nom de l'exécutable et comment est-il produit ?
3. Donner un cas qui justifie la présence de mm.h dans la liste des dépendances de mm.
4. Quelle signification donnez-vous à la règle implicite `.c.o:` ?
5. En cas de modification de mm.c, quelles seront les opérations réalisées lors d'un : `make mm`

Exercice 4 (TP Mastermind)

— Récupérez l'archive sur le site `www.lirmm.fr/~meynard` et décompressez-la. Lisez les différents fichiers textes : main.c, mm.h, mmgui.c, makefile ;
— Quel est le type du fichier `mm.o` (file) ?
— Utilisez la commande `nm` pour étudier la table des symboles du fichier `mm.o`. Que signifient T et U ?
— fabriquez le projet en lançant la commande make : que se passe-t-il ?

— lancez depuis la ligne de commande ./mm afin de jouer en mode console.
— Lancez `make mmgui` afin de fabriquer puis jouer au mastermind en mode graphique.
— Ecrire le module mm.c qui vérifie les spécification du fichier mm.h !
— Recherchez la signification de la commande `pkg-config` et la signification d'un encadrement par des anti-quote (‘).
— Rechercher la librairie standard C : `libc.so` et lisez sa table des symboles grâce à la commande `readelf`.

Exercice 5 (TP La commande make)
Indiquez les exécutions déclenchées par un appel `make mm`. Si après avoir construit une première fois l'application *mm*, on exécute la commande `touch main.c`. Quel est l'effet de `make mm` ?

Exercice 6 (TP Modifications dans les sources de l'application)
Précisez l'effet sur la construction de l'application *mm*, 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 "mm.h"` dans le fichier `main.c`
2. Remplacement de la directive `#include "mm.h"` par la directive `#include <mm.h>` dans le fichier `main.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, quand on utilise l'option -l, à définir un autre répertoire où chercher les bibliothèques (libX.so) qu'on aura indiqué dans l'option (`-lX`).
— Afficher la table des matières de la bibliothèque de base : `readelf -s libc.so`.

Concepts et programmation système - TD/TP 3
Entrées-Sorties de base

Michel Meynard

6 septembre 2018

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.

Concepts et programmation système - TD/TP 4
Génération et recouvrement de processus

Michel Meynard

6 septembre 2018

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.

Concepts et programmation système - TD/TP 5
Système de fichiers

Michel Meynard

6 septembre 2018

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 `tempo` et `toto`. 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 `tempo`. Faire en sorte qu'il ne puisse pas supprimer `toto`. 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.

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

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.

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 ?

Exercice 11
Voici un extrait de l'entête du manuel :

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

SYNOPSIS
```

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

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

Quelle est la différence entre ces deux appels ?

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

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 ?

Concepts et programmation système - TD/TP 6
Système de fichiers : accès aux inodes et répertoires

Michel Meynard

6 septembre 2018

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 ;
— ...

Exercice 8 (TD/TP)

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

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

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

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 9

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 10 (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.

Concepts et programmation système - TD/TP 7
Signaux

Michel Meynard

6 septembre 2018

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.

Exercice 7 (projet)
Afin de permettre de déterminer facilement les bogues à l'exécution, on souhaite capturer certains signaux (comme le très célèbre SIGSEGV) afin d'afficher l'état de la pile au moment du crash. On saura ainsi quelle est la fonction qui a généré le signal et qui est donc fautive. On utilisera pour ce faire la fonction déclarée dans `ucontext.h` :

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

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

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

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

Concepts et programmation système - TD/TP 8
Tubes non nommés

Michel Meynard

6 septembre 2018

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

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

dit, nous énumérons les nombres premiers par application successive de la méthode de crible suivante : Pour cribler une liste d'entiers dont le premier est premier, nous ôtons de la liste le premier élément (qui est affiché comme premier) et nous supprimons de la liste l'ensemble des multiples de ce nombre. La liste résultante sera à son tour criblée selon la même méthode, et ainsi de suite.

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

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.

Concepts et programmation système - TD/TP 9
Tubes, Exclusion et Synchronisation de Processus

Michel Meynard

6 septembre 2018

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 exclusivement pendant un certain temps. Avant d'accéder à la ressource commune, un processus devra obtenir un "jeton" unique matérialisant le privilège. Une fois les accès terminés, ce processus devra "rendre" le jeton afin de permettre aux autres d'accéder à leur tour à la ressource.

Principe de fonctionnement :

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

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

Noter que tout processus peut effectuer plusieurs fois des demandes d'accès à *fic* 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 *fic*. Chaque processus peut effectuer un nombre indéterminé de demandes d'accès.

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 *fic*. 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 *fic* meurt prématurément ?

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

Exercice 4
Analyser ce qui se passe lorsqu'aucun processus ne veut accéder à *fic* ; 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 *fic*, 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 *fic* 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 *tubjeton* pour accéder à *fic* ?

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.

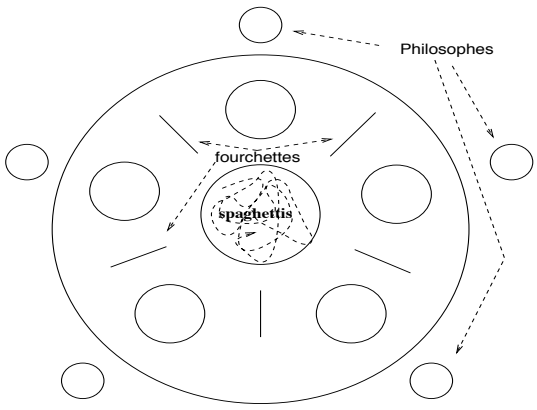
Exercice 11
P_2 va jouer le rôle de charnière commune. Dans cette question, on ne s'intéresse qu'au démarrage, c'est-à-dire tout ce qui se passe pour chaque processus, à partir de son lancement, jusqu'au moment où il peut enfin aller plus loin que le point de rendez-vous. Montrer comment une telle solution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

Concepts et programmation système - TD/TP
Threads

Michel Meynard

6 septembre 2018

1 File d'attente partagée

Exercice 1
Terminer l'implémentation de la file d'attente partagée vue en cours.

2 Projet

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

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

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

Exercice 2
Faut-il archiver les fichiers de type répertoires ? Si oui quel thread doit le faire et que sauvegarder ?

Exercice 3
Que faire des liens durs et symboliques ?

Exercice 4
Ecrire l'algo. d'un thread compresseur.