Cours PDS

F. Lemaire

23 février 2009

Table des matières

Chapitre 1

Présentation

1.1 Évaluation

- Note de C/Unix : NF = (3*Ecrit + 2*TP)/5
- Ecrit : Max(3*EX,2*EX+CC)/3
  - EX : note examen
  - CC : contrôle continu (mini-interros surprises/5 + interros prévenues)
- TP :
  - certains TP seront notés, contrôle TP fin de semestre
  - Projet de TP

1.2 Objectif

- maîtrise de la programmation des systèmes d'exploitation
- étude des concepts fournis par l'interface des systèmes d'exploitation
- principe d'utilisation de l'interface système
- pas de fonctionnement interne des SE

1.3 Contenu

- rappels C
  - chaîne de compilation
  - programmation modulaire/Makefile
  - gestion mémoire dynamique
  - structures auto-référencées
  - création de librairies
- utilisation bibliothèque C standard et de l'API POSIX
- entrées/sorties
  - systèmes de fichiers, manipulation de fichiers
- programmation en shell (traitement par lot)
  - commandes Unix
  - écriture de scripts en Bash
- processus
  - création, terminaison, interruptions, ordonnancement
- processus légers (threads)
  - création, synchronisation
- communication inter processus
  - signaux, tubes, sockets

1.4 Références

- "Unix, Linux et les systèmes d'exploitation", *2e édition*, Michel Divay, Dunod.
- Cours système de Philippe Marquet `www.lifl.fr/~marquet/cnl/pds`
- The Single Unix Specification `http://www.unix.org/version3/`
- Unix et vous, Raphaël Marvie
- Programmation en langage C, Anne Canteaux
- Section 2, 3 et 3p du man
- Wikipedia

Chapitre 2

Rappels

Le Langage C a été conçu en 1972 par Dennis Richie et Ken Thompson dans les laboratoires Bell. Dennis Richie et Brian Kernighan ont ensuite (informellement) spécifié le langage (K&R C).

Le C a été normalisé par American National Standards Institute (ANSI) en 89 ; c'est la norme C ANSI (ou C89). Une autre norme (très proche de ANSI) a été adoptée par International Organization for Standardization (ISO) en 90 : ISO/IEC 9899 :1990 (ou C90).

Une nouvelle norme ISO est créée en 1999 : ISO 9899 :1999 (ou C99). Les ajouts importants sont
- fonctions en ligne
- les variables peuvent être déclarées dans le code
- nouveaux types : long long int, bool et complex
- commentaire //

Nous utiliserons uniquement les normes ANSI ou C99, et POSIX (norme pour les appels systèmes vue plus tard dans ce cours).

2.1 Fonction principale main

La fonction `main` est exécutée lorsqu'on lance l'exécutable.

```
int main(int argc, char *argv[]) / int main(void)
```

- `argc` : nb d'arguments (dont le nom de l'exécutable, donc nb≥1)
- `argv` : tableau de chaînes de caractères (`argv[0]` : nom de l'exécutable, ... `argv[argc-1]` : dernier argument)
- le type de retour est int : le code renvoyé par la fonction `main` est récupéré par le système (permet de savoir si le programme s'est correctement exécuté).

Listing 2.1 – rappels/main_args.c

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int i;

  for (i=0;i<argc;i++)
    printf("Arg %d : %s\n", i, argv[i]);

  exit(EXIT_SUCCESS);
}
```

2.2 Chaîne de compilation

fichier sources .c, .h $\xrightarrow{\text{préprocesseur (cpp)}}$ .i $\xrightarrow{\text{compilateur (cc1)}}$ .s
$\xrightarrow{\text{assembleur (as)}}$ .o $\xrightarrow{\text{édition des liens(ld)}}$ exécutable

Quatre étapes de la compilation :
- préprocesseur : traitement des inclusions, macros, ... (`#include`, `#define`, ...) ;
- compilateur : transforme en langage assembleur
- assembleur : transforme en fichier objet
- éditeur des liens : crée un fichier exécutable

Exemple ▷ pour compiler `toto.c` : `gcc -o toto toto.c`. Réalise les quatre étapes automatiquement. ◁

Quelques options de gcc :
- -v (verbose) : donne des infos sur les différentes étapes étapes
- -E : uniquement le préprocesseur (`gcc -E toto.c`)

- -S : préprocesseur + compilateur
- -c : préprocesseur + compilateur + assembleur (donne un fichier objet : `gcc -c toto.c`
- -o : pour changer le nom du fichier créé (`gcc -o toto toto.c`)
- -Wall : affiche tous les avertissements (en C avertissement rime souvent avec erreur)
- -ansi : respecte la norme ISO C90 (+certaines extensions)
- -pedantic : (à utiliser avec -ansi) respect strict de la norme ISO C90
- -std= : pour choisir une norme (c89, c99, ...)

2.3 Préprocesseur

Le préprocesseur permet d'effectuer des opérations purement syntaxiques sur les fichiers sources. Le préprocesseur transforme les fichiers sources en traitant les directives de préprocesseurs, qui commencent toutes par #.

2.3.1 #include

Il y a deux syntaxes : `#include <entete.h>` (à utiliser pour les fichiers d'entêtes systèmes/librairies comme `stdlib.h`, `stdio.h`) ou `#include "entete.h"` (pour les fichiers d'entête de l'utilisateur). La différence se fera sur les chemins par défaut dans lesquels sont cherchés les fichiers (`/usr/include`).

2.3.2 #define

Permet de définir des variables ou des macros (les noms sont en majuscules par convention)

```
#define MAX 12
#define DEBUG
```

`MAX` sera remplacé partout par 12 (excepté dans les chaînes de caractères). `DEBUG` vaut la chaîne vide (DEBUG disparaît donc), mais la variable `DEBUG` est bien définie.. On peut définir une variable depuis gcc avec l'option `-DNAME[=valeur]`. Attention, cela ne redéfinit pas une variable déjà définie dans le code source.

```
#define CARRE(x) ((x)*(x))
...
x = CARRE(a+b)
```

La ligne est transformé en x = (a+b)*(a+b), d'où l'importance des parenthèses dans les définitions de macros

2.3.3 #ifdef, #ifndef

Permet d'inclure ou non certaines parties de codes (compilation conditionnelle).

```
#ifdef VARIABLE
  lignes insérées si VARIABLE a été définie
#elif
  lignes insérées sinon
#endif
```

ou

```
#ifdef VARIABLE
  lignes insérées si VARIABLE a été définie
#endif
```

Variante : `#ifndef`

Exemple ▷

```
#ifdef DEBUG
  printf("Initialisation de la librairie");
#endif
```

◁

2.4 Zones mémoire d'un processus

Un processus est un objet correspondant à l'exécution d'un programme (ou exécutable). Il y a plusieurs zones mémoires qui ont chacune une fonction précise.

Des adresses hautes vers les adresses basses :

Pile ⋮ Tas (mémoire dynamique)
Données statiques modifiables
Texte (programme), non modifiable
Reservé au système

2.4.1 Pile

La pile contient les adresses de retour des fonctions, les résultats des fonctions et les variables locales des fonctions. Lorsqu'une fonction se termine, les variables locales sont libérées. Attention, les variables locales tableaux déclarés statiquement[1] (ex : int tab[10]) comme variables locales sont alloués dans la pile.

2.4.2 Tas

Le tas permet de faire l'allocation dynamique (`malloc`, `realloc`, `free`). C'est une zone de données persistantes, entièrement à la charge de l'utilisateur.

```
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

2.4.3 Données statiques

Cette zone contient (entre autres) les variables globales (qui existent durant toute la durée de l'exécution du programme) et les variables locales statiques :

```
int f(int a) {
    static int nb_appels=0; /* initialisé une seule fois */
    nb_appels++;
    printf("%d\n", nb_appels);
    return a*a;
}

int main() {
  int res;
  res=f(1);
  res=f(2);
}
```

Affichera :

```
1
2
```

Exemple ▷

Listing 2.9 – rappels/test_mem.c

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

char *saisie1() {
  char ch[50];
  gets(ch);
  return ch;
}

char *saisie2() {
  char *ch;
  ch = (char*)malloc(50*sizeof(char));
  gets(ch);
  return ch;
}

int main() {
  char *ch;
  ch = saisie1();
  printf("%s\n", ch);

  ch = saisie2();
  printf("%s\n", ch);

}
```

saisie1 ch est un tableau statique : il est déclaré dans la pile. Il sera donc écrasé (peut-être pas tout de suite...) lorsque saisie1 se termine.

saisie2 cette fonction est correcte. Il faut juste ne pas oublier le libérer la zone (ne peut pas être fait par la fonction).

Rqe : sur mon pc, rien n'est affiché par le printf qui suit l'appel à saisie1. ◁

[1] par opposition à dynamiquement en utilisant un malloc

2.5 Pointeurs et passages de paramètres

Une variable de type pointeur contient une adresse. Les variables pointeurs sont des variables au même titre que les entiers, les caractères, ...

Définition : type * p ; // p est un pointeur sur une donnée de type type.

Opérations : p = &a (référencement), *p=a (déréférencement), p++, p=p+5, (arithmétique de pointeur)

Les pointeurs permettent de gérer la mémoire dynamique et aussi de simuler des modes de passages entrée/sortie pour les fonctions.

Les passages de paramètres en C sont toujours par valeur.

```
int f(int n) {
  n = n + 1;
}

int g(int *n) {
  *n = *n + 1;
}

int main() {
  int p;
  p=10;
  f(p);
  // p vaut 10
  g(&p);
  // p vaut 11
}
```

2.6 Exercices

2.6.1 Pointeurs

Exercice 2.6.1 :

```
  int p,q,r;
  int *a, *b;
```

Que se passe-t-il dans chacun des cas suivants (supposés indépendants) :

```
  p=1; q=2;

  a=&p; b=&q;
  *a=*b;
```

```
  p=4; q=p*p; r=p+q;

  a=&r; b=&q; r=*a+*b;
```

```
  p=4; a=&p; p=p+2;
  q=*a; p=p+2; r=*a;
```

```
  *a=23;
  a=18;
  a=p;
  a=&p; b=a+1; *b=3;
```

Exercice 2.6.2 :

Écrire une procédure qui permute 2 valeurs

Exercice 2.6.3 :

Écrire une procédure qui permet d'incrémenter un entier

2.6.2 Tableaux

- Déclaration d'un tableau int tab[100] ; // tableau de 100 entiers
- tab est un pointeur d'entiers sur une zone de 100 entiers consécutifs automatiquement allouée.
- tab est indicé de 0 à 99. Aucune fonction en C pour connaitre la taille d'un tableau.
- tab[2]=1 ; stocke 1 dans la 3eme case tab[3]==3 compare la quatrième case avec 3

```
int *p;
p=tab; // p pointe sur la premiere case du tableau
*p=2;  // stoke 2 dans la 1 ere case du tableau
p++;   // décale p de un cran vers la droite : p pointe sur le 2eme element
*p=34; // stocke 34 dans la 2eme case
*(p+1)=3 // stocke 3 dans la 3eme case
*(tab+5)=0 //stocke 0 dans la 6eme case
tab++; // Interdit (tab est un pointeur constant)
```

Exercice 2.6.4 :

Écrire une boucle en remplissant le tableau (on veut tab[i] == i)

Exercice 2.6.5 :

Écrire une procédure qui met à 0 tous les éléments d'un tableau

Chapitre 3

Programmation modulaire

Un programme C peut être découpé en différents modules. On regroupe dans un même module les fonctions et variables liés au même aspect du projet. Exemple : toutes les fonctions d'affichage dans un module, toutes les fonctions de calcul dans un autre. Au final, on compile chaque module et on les assemble pour obtenir un exécutable.

Les avantages sont nombreux :
- séparer l'interface de l'implantation ;
- développer les modules en parallèles (plusieurs développeurs) quand l'interface à été bien définie ;
- versions multiples de modules (version naïve sûre, version optimisée) ;
- gestion de gros projets ;
- réutilisation de modules ;
- compilation plus rapide : ne recompiler que les modules nécessaires.

3.1 Organisation en modules et compilation

3.1.1 Définition d'un module

Chaque module est organisé en 2 fichiers : le fichier d'entête et le fichier source. Le fichier d'entête décrit les types, variables et fonctions que l'on rendra accessibles aux autres modules (on parle de types, variables et fonctions exportées). Le fichier source contient les déclarations des variables exportées et le corps des fonctions exportées.

Une variable sera définie (et pas déclarée) par le mot clé extern. Une définition de fonction consiste à écrire son entête suivie de ;.

Exemple ▷

progmod/produit.h

```
extern int nb_appels_produits;
extern int produit(int, int);
/* mot clé extern optionnel pour les définitions de fonctions */
```

progmod/produit.c

```
#include "produit.h"

int nb_appels_produits=0;
int produit(int a, int b) {
    nb_appels_produits++;
    return a*b;
}
```

◁

3.1.2 Compilation d'un module

Se fait avec : `gcc -c modules.c`

Exemple ▷ `gcc -c produit.c` Produit le fichier objet `produit.o`

```
$ nm produit.o
0000000000000000 B nb_appels_produits
0000000000000000 T produit
$
```

◁

3.1.3 Utilisation d'un module

Tout fichier source qui inclut un fichier d'entête peut accéder aux variables et appeler les fonctions définies dans ce fichier d'entête.

Exemple ▷

progmod/prog.c

```
#include <stdlib.h>
#include <stdio.h>
#include "produit.h"

int main() {
    int a,b,c;
    scanf("%d",&a);
    scanf("%d",&b);
    c = produit(a,b);
    printf("Résultat               : %d\n", c);
    printf("Nb appels à produits : %d\n",  nb_appels_produits);
}
```

Compilation :

```
$ gcc -c prog.c
$ nm prog.o
0000000000000000 T main
                 U nb_appels_produits
                 U printf
                 U produit
                 U scanf
$
```

◁

Le `U` signifie que le symbole ne fait pas partie de l'objet.

3.1.4 Édition des liens

Lorsque que tous les fichiers source sont compilés en fichiers objet, on peut rassembler ces fichiers objets en un exécutable avec la commande : `gcc -o executable module1.o module2.o .....`

Exemple ▷

```
$ gcc -o prog prog.o
/tmp/ccyF5PD3.o: In function `main':
prog.c:(.text+0x35): undefined reference to `produit'
prog.c:(.text+0x3e): undefined reference to `nb_appels_produits'
$ gcc -o prog prog.o produit.o
$ ./prog
8
9
Résultat               : 72
Nb appels à produits : 1
$
```

◁

C'est lors de cette commande que se produit l'édition des liens. Tous les symboles (variables, fonctions) extérieurs à chaque module

3.1.5 Le mot-clé static

Une variable ou une fonction d'un fichier source peut être déclarée avec le mot clé `static`. Cette variable ou cette fonction devient alors privée et uniquement utilisable à l'intérieur du fichier source.

Exemple ▷

progmod/static.c

```
int i_public;
int f_public(int i) {
  return i;
}

static int i_prive;

static int f_prive(int i) {
  return i+1;
}
```

```
$ gcc -c static.c
$ nm static.o
000000000000000c t f_prive
0000000000000000 T f_public
0000000000000000 b i_prive
0000000000000004 C i_public
```

Les lettres minuscules (t,b) signifient privées au module, les majuscules (T,C) signifient public. ◁

3.1.6 Conventions d'écritures de modules

Règle 1 : structure du fichier d'entête Il peut arriver qu'un fichier d'entête soit inclus plusieurs fois dans un même fichier source (et ce n'est pas souhaitable) :

Exemple ▷

Listing 3.9 – entete1.h

```
#include "entete2.h"
```

Listing 3.10 – main.c

```
#include "entete1.h"
#include "entete2.h"
```

Conséquence : `entete2.h` va être inclus deux fois ◁

Pour éviter qu'un module ne soit inclus plusieurs fois lors de différents `#include`, chaque fichier d'entête est encapsulé dans un bloc

Listing 3.11 – entete.h

```
#ifndef _ENTETE_H
#define _ENTETE_H

#endif
```

Exemple ▷

Listing 3.12 – entete1.h

```
#ifndef _ENTETE1_H
#define _ENTETE1_H
...
#endif
```

Listing 3.13 – entete2.c

```
#ifndef _ENTETE2_H
#define _ENTETE2_H

#include "entete1.h"
...
#endif
```

Listing 3.14 – main.c

```
#include "entete1.h"     // définit _ENTETE1_H
#include "entete2.h"     // ne recharge pas entete1.h
```

◁

Règle 2 : quels fichiers d'entête inclure Un fichier d'entête ne doit inclure que les fichiers d'entête nécessaires aux variables et fonctions exportées par cet entête.

Exemple ▷

Listing 3.15 – produit.h

```
#ifndef _PRODUIT_H
#define _PRODUIT_H
typedef int tEntier;
tEntier produit(tEntier, tEntier);
void afficheEntier(tEntier);
#endif
```

◁

Rqe : même si `afficheEntier` fait un appel à `printf`, on ne met pas de `#include <stdio.h>` car le fichier d'entête en lui même ne fait pas référence à `printf`.

Un fichier source doit inclure son fichier d'entête (excepté le fichier source principal) et les fichiers d'entête qui définissent les variables et fonctions qu'il utilise.

Exemple ▷

Listing 3.16 – produit.c

```
#include <stdio.h>
#include "produit.h"
...
void afficheEntier(tEntier e) {
     printf("%d\n", e);
}
```

◁

Règle 3 : ordre d'inclusion des fichiers d'entête D'abord les fichiers d'entêtes systèmes, puis les les fichiers utilisateurs.

3.2 L'utilitaire `make`

`Make` est un outil permettant de recompiler de manière optimale un projet en n'effectuant que les recompilations nécessaires. Outil général fonctionnant pour tout type de projet (utile dès que certains fichiers doit être générés à partir d'autres à partir d'un ligne de commande)

GNU Make : `www.gnu.org/software/make/manual/`

3.2.1 Structure

Un Makefile simple est un fichier texte (nommé Makefile, makefile, ...) composé de une ou plusieurs règles. Une règle est du type :

```
cible : dependance1 dependance2 ...
    commande1
    commande2
      ...
```

où :

- cible : un nom de fichier ou une action
- dependance ? : un nom de fichier ou une autre cible
- commande ? : commande shell à exécuter. Attention il y a une tabulation devant chaque ligne de commande

Le découpage en règles est en fait une manière de coder le graphe de dépendance entre tous les fichiers (ex : `prog.o` dépend de `prog.c` et `produit.h`). On utilise `#` pour les commentaires.

3.2.2 Les cibles

Il y a deux sortes de cibles : les cibles actions et les cibles fichiers. Les cibles fichiers sont simplement des noms de fichiers qui seront générées par les commandes de la règle.

```
prog : prog.o produit.o
       gcc -o prog prog.o produit.o
```

À l'inverse, les cibles actions ne correspondent pas à des noms de fichiers. Elles permettent d'exécuter des commandes comme le nettoyage des fichiers générés, la création d'une archive. Les cibles actions doivent être déclarées en `.PHONY` pour éviter des problèmes avec d'éventuels fichiers ayant le même nom.

```
clean :
      rm -f prog prog.o produit.o

.PHONY : clean
```

3.2.3 Évaluation d'une règle

Pour mettre à jour une ou plusieurs cibles, on va évaluer les règles du Makefile. Pour ce faire, on utilise l'utilitiaire `make`.

Pour évaluer la première règle : `$ make`. Pour évaluer une règle de cible `cible` : `$ make cible`

Exemple ▷

```
$ make prog
$ make clean
```

◁

Lorsqu'une règle **R** est évaluée ;

- toutes les règles dont la cible est une dépendance de **R** sont évaluées.
- si la cible est une action, l'ensemble des commandes de **R** est exécuté
- si la cible est un nom de fichier, l'ensemble des commandes est exécuté si l'une (ou plusieurs) des conditions suivantes est vraie :

- la cible n'existe pas
- l'une des dépendances est une action
- l'une des dépendances est un fichier plus récent que la cible

Exemple ▷

progmod/Makefile

```
prog : prog.o produit.o
	gcc -o prog prog.o produit.o

produit.o : produit.c produit.h
	gcc -c produit.c

prog.o : prog.c produit.h
	gcc -c prog.c

clean :
	rm -f prog prog.o produit.o

.PHONY : clean
```

Scénario 1 Seuls les fichiers `.c` et `.h` existent et on tape `make`.
- la règle 1 est évaluée
- les dépendances de la règle 1 sont traitées
- `prog.o` est la cible de la règle 3
  - la règle 3 est évaluée
  - les dépendances `prog.c` et `produit.h` ne sont pas des cibles
  - la cible `prog.o` n'existe pas, on exécute la commande `gcc -c prog.c` qui crée `prog.o`
- `produit.o` est la cible de la règle 2
  - la règle 2 est évaluée
  - les dépendances `produit.c` et `produit.h` ne sont pas des cibles
  - la cible `produit.o` n'existe pas, on exécute la commande `gcc -c produit.c` qui crée `prog.o`
- la cible `prog` n'existe pas, on exécute la commande `gcc -o prog prog.o produit.o`

Scénario 2 On modifie ensuite `prog.c` et on tape **make**.
- la règle 1 est évaluée
- les dépendances de la règle 1 sont traitées
- `prog.o` est la cible de la règle 3
  - la règle 3 est évaluée
  - les dépendances `prog.c` et `produit.h` ne sont pas des cibles
  - la cible `prog.o` est plus ancienne que **prog.c**, on exécute la commande `gcc -c prog.c` qui met à jour `prog.o`
- `produit.o` est la cible de la règle 2
  - la règle 2 est évaluée
  - les dépendances `produit.c` et `produit.h` ne sont pas des cibles
  - la cible `produit.o` est plus récente que `produit.c`, on ne fait rien
- la cible `prog` est plus ancienne que `prog.o` (à cause de la règle 3 évaluée juste avant), on exécute la commande `gcc -o prog prog.o produit.o` et **prog** est à jour

On a donc évité de recompiler `produit.o`, on a recompilé de manière optimale. ◁

3.2.4 Variables

Les variables se déclarent en utilisant `nom = valeurs`. On accède au contenu de la variable avec `$(nom)` (ou bien `${nom}`). Si `nom` n'a pas été affectée, alors `$(nom)` représente la chaîne vide.

On peut déclarer et remplacer la valeur d'une variable depuis la ligne de commande en utilisant l'option `-D` de `make`. Par exemple, `make CFLAGS=-Werror -ansi -pedantic`.

progmod/Makefile–Variables

```
OBJ = prog.o produit.o
CFLAGS = -Wall -ansi -pedantic
CC = gcc

prog : $(OBJ)
	$(CC) $(CFLAGS) -o prog $(OBJ)

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

prog.o : prog.c produit.h
	$(CC) $(CFLAGS) -c prog.c

clean :
	rm -f prog $(OBJ)
```

```
.PHONY : clean
```

3.2.5 Variables automatiques

Dans la définition d'une règle, on peut utiliser les variables :
- `$@` nom de la cible
- `$<` nom de la première dépendance
- `$^` liste des dépendances
- `$?` liste des dépendances plus récentes que la cible
- `$*` nom de la cible sans suffixe

progmod/Makefile–Var–Auto

```
OBJ = prog.o produit.o
CFLAGS = -Wall -ansi -pedantic
CC = gcc

prog : $(OBJ)
	$(CC) $(CFLAGS) -o $@ $(OBJ)

produit.o : produit.c produit.h
	$(CC) $(CFLAGS) -c $<

prog.o : prog.c produit.h
	$(CC) $(CFLAGS) -c $<

clean :
	rm -f prog $(OBJ)

.PHONY : clean
```

3.2.6 Règles implicites

On peut utiliser des règles implicites pour éviter de taper des commandes identiques à plusieurs règles similaires (par exemple construire un .o à partir d'un .c).

Des règles implicites et des variables par défaut sont définies par `make`. Utiliser `make -p -f/dev/null` pour les visualiser. Pour désactiver les règles implicites (resp. les variables prédéfinies), utiliser l'option `-r` (resp. `-R`).

progmod/output.make.p.court

```
...
# default
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
...
# default
OUTPUT_OPTION = -o $@
...
%.o: %.c
#  commands to execute (built-in):
	$(COMPILE.c) $(OUTPUT_OPTION) $<
```

Exemple ▷

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

◁

On peut alors utiliser des règles sans commandes, ce qui permet de définir les dépendances.

progmod/Makefile–implicite

```
OBJ = prog.o produit.o
CFLAGS = -Wall -ansi -pedantic
CC = gcc

prog : $(OBJ)
	$(CC) $(CFLAGS) -o $@ $(OBJ)

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

produit.o : produit.c produit.h

prog.o : prog.c produit.h

clean :
	rm -f prog $(OBJ)

.PHONY : clean
```

Remarque : les règles implicites ne sont utilisées que si aucune régle ne correspond à une cible demandée.

Chapitre 4

Les librairies

4.1 Présentation

Une librairie est une collection de symboles (fonctions, variables, ...) regroupés dans un seul fichier. Une librairie se construit en regroupant un ou plusieurs fichiers objets.

Objectif d'une librairie : fournir au programmeur un ensemble de fonctions et de variables qu'il peut utiliser pour écrire des applications.

Exemple :

librairies/ficcos.c

```
#include <math.h>

int main() {
  printf("Cos de 1 vaut : %f\n", cos(1));
  return 0;
}
```

```
$ gcc -c ficcos.c
$ gcc -o ficcos ficcos.o
/tmp/cc8ojs65.o(.text+0x16): In function `main':
: undefined reference to `cos'
collect2: ld returned 1 exit status
$ gcc -o ficcos ficcos.o -lm
$
```

La fonction `cos` fait partie de la librairie mathématique, dont les routines sont dans le fichier libm.so. Le -lm signifie que le compilateur peut aller chercher des symboles dans le fichier libm.so.

Il y a deux types de librairies : statiques et dynamiques.

librairies/fic1.c

```
#include <stdio.h>
#include "fic1.h"

void mes1() {
  printf("Message 1\n");
}
```

librairies/fic2.c

```
#include <stdio.h>
#include "fic2.h"

void mes2() {
  printf("Message 2\n");
}

void mes2bis() {
  printf("Message 2 bis\n");
}
```

librairies/testmes.c

```
#include "fic1.h"
#include "fic2.h"

int main() {
  mes1();
  mes2();
}
```

fic1.h et fic2.h contiennent simplement les entêtes de mes1 et mes2.

4.2 Les librairies statiques

Une librairie statique n'est utilisée qu'au moment de la compilation. Lorsque l'on crée un exécutable à partir d'une librairie statique, les symboles de la librairies sont dupliqués dans l'exécutable.

L'avantage est que l'exécutable est indépendant de la librairie (pas de problème l'exécution à cause d'une librairie manquante). le fichier librairie est traditionnellement .a sous Unix (.lib sous win)

4.2.1 Création

On utilise la commande `ar` : `ar crs libnom.a fic1.o fic2.o ...`

- c : créer si nécessaire
- r : ajouter
- s : créer un index des symboles

```
$ ar cr libmes_static.a fic1.o fic2.o
```

Pour afficher les fichiers contenus :

```
$ ar t libmes_static.a
fic1.o
fic2.o
$
```

Pour voir les symboles contenus :

```
$ nm libmes_static.a
fic1.o:
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T mes1
                 U printf

fic2.o:
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T mes2
0000000000000017 T mes2bis
                 U printf
$
```

4.2.2 Utilisation

Pour compiler un programme se servant de libnom.a, on fait :

```
$ gcc -o main main.o -L. -lnom
```

Exemple ▷

```
$ gcc -o testmes_static testmes.o -L. -lmes_static
$ nm testmes_static
...
00000000004004e8 T main
0000000000400504 T mes1
000000000040051c T mes2
0000000000400533 T mes2bis
...
$
```

L'exécutable `testmes_static` contient le code des fonctions `mes1`, `mes2` et meme `mes2bis`.

Le résultat final est semblable à celui obtenu par :

```
gcc -o testmes_static testmes.o fic1.o fic2
```

◁

4.3 librairies dynamiques

Contrairement aux librairies statiques, les librairies dynamiques sont utilisées à la compilation et durant le lancement de l'exécutable. En effet, si on crée un exécutable utilisant une librairie dynamique, les symboles seront chargés dans la librairie pendant de l'exécution (ce qui implique que la librairie doit être accessible au moment de l'exécution). On utilise l'extension .so sous Unix (shared object) (.dll sous win dynamically linked library).

4.3.1 Création

Les fichiers objets destinés à la librairie doivent être créées avec l'option `-fPIC` (position independant code) :

```
$ gcc -fPIC -c fic1.c
$ gcc -shared -fPIC libnom.so fic1.o fic2.o ....
```

Exemple ▷

```
$ gcc -shared -fPIC -o libmes_dynamic.so fic1.o fic2.o
$ nm libmes_dynamic.so
...
0000000000000738 T mes1
0000000000000750 T mes2
0000000000000767 T mes2bis
...
```

◁

4.3.2 Utilisation

Identique aux librairies statiques : pour compiler un programme se servant de `libnom.so`, on fait :

```
$  gcc -o main main.o -L. -lnom
```

Rqe : On n'a pas besoin de `-fPIC`.

Exemple ▷

```
$  gcc -o testmes_dynamic testmes.o -L. -lmes_dynamic
$  nm testmes_dynamic
0000000000400628 T main
                 U mes1
                 U mes2
```

◁

Le programme `testmes_dynamic` ne pourra s'exécuter que si `libmes_dynamic.so` est accessible au moment de l'exécution. Pour vérifier cela :

```
$ ldd testmes_dynamic
        libmes_dynamic.so => not found
        libc.so.6 => /lib/libc.so.6 (0x00002b8baebb0000)
        /lib64/ld-linux-x86-64.so.2 (0x00002b8baea94000)
$ echo $LD_LIBRARY_PATH
$ export LD_LIBRARY_PATH=.
$ ldd testmes_dynamic
        libmes_dynamic.so => ./libmes_dynamic.so (0x00002b97860a6000)
        libc.so.6 => /lib/libc.so.6 (0x00002b97861c3000)
        /lib64/ld-linux-x86-64.so.2 (0x00002b9785f8a000)
```

Si l'on place des librairies dynamiques à des endroits spéciaux (autres que les répertoires systèmes), il faut indiquer les positions grâce à la variables `LD_LIBRARY_PATH`. C'est là qu'à l'exécution le système va chercher les libraires dynamiques non présentes dans les répertoires systèmes. Une bonne idée est de s'assurer que `LD_LIBRARY_PATH` contient bien `'.'` (le répertoire courant).

4.4 Comparaison statique/dynamique

Librairies statiques :
- \+ autonomie : l'exécutable fonctionne sans les librairies
- \- taille importante des exécutables (occupe de la place disque et mémoire)

Librairies dynamiques :
- \+ petite taille des exécutables
- \+ pas de duplication de code
- \- l'exécutable ne change pas même si on met à jour la librairie (inconvénient : l'exécutable ne fonctionne plus si l'interface de la librairie change, problème de maintenance, ...)

4.5 Chargement dynamique de fonctions

Le principe du chargement dynamique est de charger en mémoire *pendant* l'exécution d'un processus une fonction contenue dans une libraire dynamique pour ensuite l'exécuter.

Cela est pratique pour l'écriture de plugin (qui est une extension logicielle à une application). En effet, un plugin peut être écrit comme une librairie dynamique vérifiant une interface fixée à l'avance.

Chargement d'une fonction définie dans `libmes.so` Après avoir ouvert la librairie avec `dlopen`, on charge une fonction donnée par une chaîne de caractères en utilisant `dlsym`. La chaîne de caractères doit correspondre au nom de la fonction qui a été compilée dans la librairie dynamique.

librairies/ldtest.c

```
#include <dlfcn.h>
#include <stdlib.h>
#include <stdio.h>

/*

void *dlopen(const char *filename, int flag);
const char *dlerror(void);
void *dlsym(void *handle, char *symbol);
int dlclose(void *handle);

*/

typedef void (*tMessage)();

int main() {

  void *handle;
  tMessage mes;
  char *error;

  handle = dlopen ("./libmes_dynamic.so", RTLD_LAZY);
  if (handle == NULL) {
    fprintf (stderr, "%s\n", dlerror());
    exit(1);
  }

  mes = dlsym(handle, "mes1");
  if ((error = dlerror()) != NULL)  {
    fprintf (stderr, "%s\n", error);
    exit(1);
  }

  mes();
  dlclose(handle);
  return 0;

}
```

Pour compiler :

```
gcc -rdynamic -o ldtest ldtest.c -ldl
```

Chapitre 5

Unix/Shell

Une mine de renseignements sur `http://tldp.org/LDP/abs/html/`

5.1 Commandes Unix/Shell de base

Tous les processus renvoient un code de retour lorsqu'ils se terminent (en C, c'est la valeur renvoyée par `exit` ou le `return` du `main`). Par convention, la valeur 0 signifie que tout s'est bien passé, une valeur non nulle signifie qu'un problème a eu lieu (au besoin la valeur permet d'identifier le problème précisément).

5.1.1 Caractères génériques

Ils permettent de désigner un ensemble de fichiers satisfaisant un modèle. On peut construire un modèle en utilisant `*` (toute chaîne), `?` (caractère quelconque), une liste de caractères `[abc]` (l'un des 3 caractères a,b ou c), une liste de plages de caractères `[a-zA-Z]` (un caractère minuscule ou majuscule) et la négation `[!a-z]`, ou bien `[^a-z]` (tout caractère non minuscule).

On peut utilisant autant de caractères génériques que l'on veut, à l'interieur d'arborescence si on le désire.

```
#tous les fichiers tex et scripts shell
$ echo *.tex *.sh
shell.tex boucles.sh fonctions.sh simple.sh
#finissant par n
$ echo *[n]
cmd.cargen cmd.ln nocmd.man out.cargen out.ln
# commancant par s
$ echo s*
shell.aux shell.tex shell.tex~ simple.sh
# commancant par s sans tilde final
$ echo s*[^~]
shell.aux shell.tex simple.sh
```

5.1.2 Documentation

`man` Affiche les commandes du manuel.

```
man -a mot
man [section] mot
```

Touches utiles : (espace) pour avancer d'une page, (b) pour reculer d'une page, (/) pour chercher une chaîne, (q) pour quitter.

```
# liste la première section trouvée du manuel sur kill
$ man kill
# liste la section 2 sur kill
$ man 2 kill
# liste toutes les sections sur kill
$ man -a kill
```

`info` Affiche la documentation au format info.

```
info mot
```

La documentation est mieux structurée que celle du man car elle est organisée en sections (arbre). Touches utiles : (?) affiche l'aide, (l) quitte l'aide, (n) nœud suivant, (p) nœud précédent, (u) remonter d'un niveau, (s ou /) rechercher une chaîne, (return) pour accéder à un sous-menu (indiqué par *)

5.1.3 Utilisateurs/droits

Chaque utilisateur est décrit (entre autres) par un nom de login et un numéro (uid). Un utilisateur peut appartenir de plusieurs groupes (utile pour partager des droits). A tout moment, un utilisateur a un seul groupe courant, mais peut en changer quand il le veut.

`whoami` affiche le nom de login.

`groups` affiche les groupes dont l'utilisateur fait partie

`id` affiche le détail de l'utilisateur et des groupes dont il fait partie.

`newgroup` change le groupe courant

`su` change d'utilisateur courant

```
$ whoami
lemaire
$ id
uid=6009(lemaire) gid=60(calforme) groups=10(wheel),18(audio),19(cdrom),
27(video),31(grildist),60(calforme),409(games)
$ rm -f toto && touch toto
$ newgrp games
$ id
uid=6009(lemaire) gid=409(games) groups=10(wheel),18(audio),19(cdrom),
27(video),31(grildist),60(calforme),409(games)
$ rm -f titi && touch titi
$ ls -l toto titi
-rw-r--r-- 1 lemaire games    0 2007-03-01 16:14 titi
-rw-r--r-- 1 lemaire calforme 0 2007-03-01 16:14 toto
```

Les fichiers ont des droits d'accés répartis en trois catégories : utilisateur (u), groupe (g), autres (x). Chaque catégorie possède trois droits : lecture (r), écriture (w) et exécution (x). L'exécution est nécessaire pour exécuter un fichier ou rentrer dans un répertoire (avec cd). Les droits sont visibles avec `ls -l`.

`chmod` change les droits

`chgrp` change le groupe

`chown` change le propriétaire

```
$ touch toto
$ ls -l toto
-rw-r--r-- 1 lemaire calforme 0 2008-03-25 12:13 toto
$ chmod a-r toto && ls -l toto
--w------- 1 lemaire calforme 0 2008-03-25 12:13 toto
$ chmod ug+r,u+x toto && ls -l toto
-rwxr----- 1 lemaire calforme 0 2008-03-25 12:13 toto
$ chmod 644 toto && ls -l toto
-rw-r--r-- 1 lemaire calforme 0 2008-03-25 12:13 toto
$ chgrp games toto && ls -l toto
-rw-r--r-- 1 lemaire games 0 2008-03-25 12:13 toto
```

`umask` change le masque de création (fixe les droits à la création).

```
$ umask
0022
$ rm -f toto && touch toto
$ rm -rf tutu && mkdir tutu
$ ls -ld toto tutu
-rw-r--r-- 1 lemaire calforme    0 2008-03-25 12:13 toto
drwxr-xr-x 2 lemaire calforme 4096 2008-03-25 12:13 tutu
$ umask 0077
$ rm -f toto && touch toto
$ rm -rf tutu && mkdir tutu
$ ls -ld toto tutu
-rw------- 1 lemaire calforme    0 2008-03-25 12:13 toto
drwx------ 2 lemaire calforme 4096 2008-03-25 12:13 tutu
```

Pour gérer les utilisateurs et les groupes, l'administrateur dispose de `useradd`, `userdel`, `groupadd`, `groupdel`, `usermod`.

5.1.4 Répertoires

`ls` affiche le contenu de répertoires

`cd` change le répertoire courant

`pwd` affiche le répertoire courant

`mkdir`, `rmdir` crée un répertoire, supprime un répertoire supposé vide

`dirname`, `basename` affiche le répertoire du fichier/le nom sans répertoire du fichier

```
# version multicolonnes (-C est optionnel)
$ ls -C
boucles.sh          cmd.ln          genereAll              out.id
boucles.sh~         cmd.ls          genereExec             out.if
cmd.boucles         cmd.ps          genereExec.multiligne  out.ln
cmd.cargen          cmd.read        nocmd.man              out.ls
cmd.chmod           cmd.rep         out.boucles            out.ps
cmd.cmp             cmd.simple      out.cargen             out.read
cmd.comp.neg        cmd.test        out.chmod              out.rep
cmd.comp.sequence   cmd.umask       out.cmp                out.simple
cmd.du              cmd.variables   out.comp.neg           out.test
cmd.echo            cours.log       out.comp.sequence      out.umask
cmd.expr            exemples        out.du                 out.variables
cmd.file            fic1            out.echo               shell.aux
cmd.fonctions       fic2            out.expr               shell.tex
cmd.grep            fic.read        out.file               shell.tex~
cmd.head            fic.sort        out.fonctions          simple.sh
cmd.id              fic.wc          out.grep               tata
cmd.if              fonctions.sh    out.head               typescript
# format long
$ ls -l | head -n 5
total 260
-rwxr-xr-x 1 lemaire calforme   354 2008-03-25 12:11 boucles.sh
-rwxr-xr-x 1 lemaire calforme   323 2007-03-20 14:52 boucles.sh~
-rw-r--r-- 1 lemaire calforme    22 2007-03-12 17:56 cmd.boucles
-rw-r--r-- 1 lemaire calforme   158 2007-03-12 18:08 cmd.cargen
# Affichage de l'inode avec -i
$ ls -il | head -n 5
total 260
10439858 -rwxr-xr-x 1 lemaire calforme   354 2008-03-25 12:11 boucles.sh
 8428689 -rwxr-xr-x 1 lemaire calforme   323 2007-03-20 14:52 boucles.sh~
 9865428 -rw-r--r-- 1 lemaire calforme    22 2007-03-12 17:56 cmd.boucles
10581463 -rw-r--r-- 1 lemaire calforme   158 2007-03-12 18:08 cmd.cargen
```

```
$ pwd
/home/calforme/lemaire/enseign/courant/pds/cours/cours/shell
$ mkdir tutu
$ touch tutu/titi
$ rmdir tutu
rmdir: tutu: Directory not empty
$ rm tutu/titi
$ rmdir tutu
$ basename `pwd`
shell
$ dirname `pwd`
/home/calforme/lemaire/enseign/courant/pds/cours/cours
# attention dirname/basename sont purement "syntaxiques"
$ ls -l /imaginaire/inexistant
ls: cannot access /imaginaire/inexistant: No such file or directory
$ basename /imaginaire/inexistant
inexistant
$ dirname /imaginaire/inexistant
/imaginaire
```

5.1.5 Fichiers/Répertoires

`du` (disk usage) affiche la taille occupée sur le disque

`df` affiche le taux d'occupation des systèmes de fichiers montés

`wc` compte le nombre de retours chariots, de mots et d'octets

`file` devine le type d'un fichier en lisant son contenu

`head`, `tail` affiche le début, la fin d'un fichier

`cat` affiche le contenu d'un/plusieurs fichiers

`touch` change la date de modification et d'accès au fichier. si le fichier n'existe pas, il est créé (vide).

`more`, `less` affiche page par page un fichier

`cp` copie de fichiers ou répertoires

`mv` renommage/déplacement de fichiers/répertoires

`rm` suppression de fichiers/répertoires

`diff`, `cmp` comparaison de fichiers

`ln` création de liens physiques/symboliques

```
$ du .
36      ./exemples
292     .
$ du -s .
292     .
$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda6              5771468    932688   4545596  18% /
udev                     10240       144     10096   2% /dev
```

```
/dev/sda7              4814936   2745672   1824676  61% /var
/dev/sda8             38456308  15824960  20677848  44% /usr
/dev/sda9             38456308  10865428  25637380  30% /export
shm                    1025688         0   1025688   0% /dev/shm
livinus:/vol/home/calforme
                       9216000   6517216   2698784  71% /home/calforme
$ cat fic.wc
trois lignes
sept mots
quarante huit caractères
$ wc fic.wc
 3  7 48 fic.wc
```

```
$ file shell.tex
shell.tex: LaTeX document text
$ file ./genereExec
./genereExec: Bourne-Again shell script text executable
$ file /bin/zcat
/bin/zcat: Bourne shell script text executable
$ file /usr/share/pixmaps/faces/sky.jpg
/usr/share/pixmaps/faces/sky.jpg: JPEG image data, JFIF standard 1.01
```

```
$ cat cmd.chmod
touch toto
ls -l toto
chmod a-r toto && ls -l toto
chmod ug+r,u+x toto && ls -l toto
chmod 644 toto && ls -l toto
chgrp games toto && ls -l toto
$ tail -n 3 cmd.chmod
chmod ug+r,u+x toto && ls -l toto
chmod 644 toto && ls -l toto
chgrp games toto && ls -l toto
$ head -n 3 cmd.chmod
touch toto
ls -l toto
chmod a-r toto && ls -l toto
```

```
$ cat fic1
Petit fichier
avec une ereur !
$ cat fic2
Petit fichier
avec une erreur !
# $? est le code de retour de la dernière commande
$ cmp fic1 fic1 ; echo $?
0
$ echo $?
0
$ cmp fic1 fic2 ; echo $?
fic1 fic2 differ: byte 26, line 2
1
$ echo $?
0
$ diff fic1 fic2
2c2
< avec une ereur !
---
> avec une erreur !
```

```
$ echo Bonjour > toto
$ cat toto
Bonjour
$ ln -s toto titi
$ ls -l toto titi
lrwxrwxrwx 1 lemaire calforme 4 2008-03-25 12:13 titi -> toto
-rw-r--r-- 1 lemaire calforme 8 2008-03-25 12:13 toto
$ cat titi
Bonjour
$ rm titi
# un lien symbolique peut pointer n'importe où
$ ln -s inconnu titi
$ ls -l titi
lrwxrwxrwx 1 lemaire calforme 7 2008-03-25 12:13 titi -> inconnu
$ cat titi
cat: titi: No such file or directory
# sans -s, on crée un lien physique. deux fichiers avec les mêmes données.
$ ln toto tutu
$ ls -li toto tutu
10439873 -rw-r--r-- 2 lemaire calforme 8 2008-03-25 12:13 toto
10439873 -rw-r--r-- 2 lemaire calforme 8 2008-03-25 12:13 tutu
$ echo Au revoir > toto
$ cat tutu
Au revoir
$ rm toto
$ cat tutu
Au revoir
```

5.1.6 Recherche/tri

`grep` recherche un motif dans un fichier

`find` recherche un fichier

`sort` trie un fichier

`tr` remplacement/suppression de caractères

```
# grep
$ grep quarante fic.wc
quarante huit caractères
$ grep chapter shell.tex
\chapter{Unix/Shell}
$ grep chapter *tex
\chapter{Unix/Shell}
$ grep -l chapter *tex
shell.tex
$ grep -q toto *; echo $?
0
# find
$ find .. -name Makefile
../entreessorties/exemples/Makefile
../Makefile
../librairies/Makefile
../progmod/Makefile
../semaphores/exemples/Makefile
$ find .. -name Makefile\*
../entreessorties/exemples/Makefile
../Makefile
../librairies/Makefile
../progmod/Makefile
../progmod/Makefile-implicite
../progmod/Makefile-Var-Auto
../progmod/Makefile-Variables
../semaphores/exemples/Makefile
# sort
$ cat fic.sort
15 quinze
1 un
10 dix
5 cinq
20 vingt
$ sort fic.sort
10 dix
15 quinze
1 un
20 vingt
5 cinq
$ sort -n fic.sort
1 un
5 cinq
10 dix
15 quinze
20 vingt
$ sort -k 2 fic.sort
5 cinq
10 dix
15 quinze
1 un
20 vingt
# tr
$ echo "salut toi" | tr a-z A-Z
SALUT TOI
$ echo "sssaaalllluuuut ! " | tr -s sal
saluuuut !
$ echo "resalut ! " | tr -d re
salut !
```

5.1.7 Processus

`ps` affiche la liste des processus de l'utilisateur courant lancés depuis le terminal courant. Options : `x` pour inclure les processus lancés autrement que depuis le terminal, `a` pour inclure les processus lancés aussi par les autres utilisateurs, `u` affiche l'utilisateur.

`pstree` affiche la liste des processus sous forme d'arbres

`kill` envoie un signal à un processus

```
$ ps
  PID TTY          TIME CMD
 9370 pts/0    00:00:00 tcsh
10100 pts/0    00:00:07 emacs
10455 pts/0    00:00:00 xdvi-xaw.bin
10589 pts/0    00:00:00 genereAll
11323 pts/0    00:00:00 genereExec
11330 pts/0    00:00:00 ps
$ ps a | head -n 10
  PID TTY      STAT   TIME COMMAND
 5729 tty2     Ss+    0:00 /sbin/agetty 38400 tty2 linux
 5730 tty3     Ss+    0:00 /sbin/agetty 38400 tty3 linux
 5731 tty4     Ss+    0:00 /sbin/agetty 38400 tty4 linux
 5732 tty5     Ss+    0:00 /sbin/agetty 38400 tty5 linux
 5733 tty6     Ss+    0:00 /sbin/agetty 38400 tty6 linux
 9370 pts/0    Ss     0:00 -tcsh
```

```
 9487 pts/2    Ss+    0:00 mutt
 9698 pts/3    Ss+    0:00 -tcsh
10100 pts/0    S      0:07 emacs systeme/systeme.tex
$ ps x | head -n 10
  PID TTY      STAT   TIME COMMAND
 9247 ?        Ssl    0:00 gnome-session
 9272 ?        S      0:00 /usr/bin/dbus-launch --sh-syntax --exit-with-session
 9273 ?        Ss     0:00 /usr/bin/dbus-daemon --fork --print-pid 6 --print-address 9 --session
 9276 ?        Ss     0:00 /usr/bin/ssh-agent -- gnome-session
 9278 ?        S      0:00 /usr/libexec/gconfd-2 6
 9281 ?        S      0:00 /usr/bin/gnome-keyring-daemon
 9283 ?        Sl     0:00 /usr/libexec/gnome-settings-daemon
 9285 ?        S      0:02 /usr/bin/sawfish --sm-client-id 1186ce0b570001161774568000000168020009 --sm-prefix bvj7yb29y4
 9287 ?        S      0:04 gnome-panel --sm-config-prefix /gnome-panel-rEHhUb/ --sm-client-id 1186ce0b57000116177364000
$ ps u
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
lemaire    9370  0.0  0.1  22088  2080 pts/0    Ss   10:12   0:00 -tcsh
lemaire    9487  0.0  0.2  33328  4800 pts/2    Ss+  10:16   0:00 mutt
lemaire    9698  0.0  0.0  21988  2032 pts/3    Ss+  10:42   0:00 -tcsh
lemaire   10100  0.1  0.8  67612 17964 pts/0    S    11:07   0:07 emacs systeme/systeme.tex
lemaire   10333  0.0  0.0  21988  2024 pts/4    Ss+  11:42   0:00 -tcsh
lemaire   10455  0.0  0.3  45184  6976 pts/0    S    11:55   0:00 xdvi-xaw.bin -name xdvi cours.dvi
lemaire   10589  0.5  0.0  11552  1468 pts/0    S+   12:13   0:00 /bin/bash ./genereAll
lemaire   11323  0.0  0.0  11580  1444 pts/0    S+   12:13   0:00 /bin/bash ./genereExec
lemaire   11350  0.0  0.0  13836  1020 pts/0    R+   12:13   0:00 ps u
```

5.1.8 Entrées/sorties

`echo` affiche sur la sortie standard

`read` lit sur l'entrée standard

```
# le shell isole les paramètres séparés par des espaces
$ echo bonjour
bonjour
$ echo "bonjour"
bonjour
$ echo bonjour   toi
bonjour toi
$ echo "bonjour    toi"
bonjour toi
# -n : pas de retour ligne
$ echo -n Bonjour ; echo " toi"
Bonjour toi
# -e : pas d'interprétation des \
$ echo "\t" salut
\t salut
$ echo -e "\t" salut
        salut
$ echo -e
```

```
$ read a
salut
$ echo $a
salut
$ read a b
bonjour a toi !
$ echo $a
bonjour
$ echo $b
a toi !
```

```
$ cat fic.read
joe:dalton:/bin/sh:fds9z798r7ds
lucky:luke:/bin/sh:a9e7897fs8
# en redirigeant depuis un fichier
# si il n'y a qu'un paramètre, il reçoit la premier ligne
$ read a < fic.read
$ echo $a
joe:dalton:/bin/sh:fds9z798r7ds
# si il y a plusieurs paramètres, chacun reçoit un mot
# (le dernier reçoit le restant de la ligne)
$ read a b < fic.read
$ echo $a $b
joe:dalton:/bin/sh:fds9z798r7ds
# dans une boucle
$ while read ligne; do echo $ligne; done < fic.read
joe:dalton:/bin/sh:fds9z798r7ds
lucky:luke:/bin/sh:a9e7897fs8
$ OIFS="$IFS"
$ IFS=":"
$ while read nom prenom ignore; do echo "Hi $nom $prenom !"; done < fic.read
Hi joe dalton !
Hi lucky luke !
$ IFS="$OIFS"
# ne pas utiliser avec un echo et un tube !
# ce qui suit ne fonctionne pas :
$ echo toto titi | read c
$ echo $c
```

5.2 Shell

Nous n'utiliserons que le shell bash (Bourne-Again SHell) qui est une version étendue de sh. Le programme bash se situe habituellement dans /bin/bash.

5.2.1 Redirections/tubes

Redirections Chaque processus dispose (sauf exception) de trois flux ouverts qui sont
- l'entrée standard : là où sont lus les caractères
- la sortie standard : là où sont faits les affichages
- la sortie standard erreur : une deuxième sortie réservée erreurs

Par défaut, l'entrée standard est le clavier, et les deux sorties sont l'écran. On utilise `<` pour redéfinir l'entrée standard vers un fichier. `$ sort < fichier` lit les lignes dans `fichier` au lieu de les lire au clavier. On utilise `>` et `>>` pour redéfinir la sortie standard. `$ ls > fichier` va écraser ou créer `fichier` et y stocker le résultat de la sortie écran de la commande `ls`, `$ ls >> fichier` fera la même chose mais en ajoutant la sortie écran à la fin de `fichier`.

On peut aussi rediriger la sortie erreur avec `2>` Pour terminer, on peut rediriger la sortie erreur sur la sortie standard en utilisant `2>&1`. Il faut faire attention à l'ordre :
- `commande > fic 2>&1` redirige les deux sorties dans fic
- `commande 2>&1 > fic` redigire la la sortie erreur à l'écran (ne change rien donc !) et la sortie standard dans fic

Tubes On peut connecter la sortie standard d'une commande à l'entrée standard d'une autre commande à l'aide d'un tube. Moralement, le tube correspond à un reservoir (de taille finie) dans lequel peuvent transiter des caractères. `ls | sort` :
- la sortie standard de ls est redirigée vers l'entrée du tube
- l'entrée standard de more est redirigée vers la sortie du tube
- ls n'affiche donc rien (c'est sort qui produit l'affichage)
- sort ne lit rien au clavier, mais lit ses données dans le tube
- c'est le SE qui gère le tube automatiquement (la commande ls est bloquée quand le tube est rempli, et sort est bloqué quand le tube est vide)

on peut ainsi mettre plusieurs commandes de traitement de caractères (appelés aussi filtre) à la suite les unes des autres.

5.2.2 Variables

Toutes les variables de shell sont des chaînes de caractères. On les déclare par `VAR=valeur`, attention pas d'espaces autour de `=`. Une variable ne sera connue des sous-shell (et plus généralement des processus fils) que si elle est exportée en utilisant :
- soit `export VAR=valeur`
- soit `VAR=valeur` suivi de `export VAR`

Quelques variables ont une valeur par défaut :
- `HOME` : chemin de la racine du compte
- `PATH` : liste de chemins où chercher les exécutables lancés depuis le shell
- `PS1` : invite affichée à chaque ligne en mode shell interactif

On accède au contenu d'une variable en utilisant `$`. Par exemple : `echo $HOME`

On peut construire des chaînes de caractères de trois façons :

1. avec les guillemets `"..."` : les caractères spéciaux sont traités et les variables sont remplacées par leurs valeurs ;
2. avec les apostrophes `'...'` : aucun traitement n'est réalisé ;
3. avec les apostrophes inversées `` `cmd arg1 ... argN` `` : la chaîne obtenue est ce qu'afficherait à l'écran la commande entre `cmd arg1 ... argN`. On appelle cela la substitution de commandes.

```
$ echo $TOTO

$ TOTO=toto
$ TITI="Salut $TOTO"
$ echo $TITI
Salut toto
$ TITI='Salut $TOTO'
$ echo $TITI
Salut $TOTO
$ TATA=`pwd`
$ echo $TATA
/home/calforme/lemaire/enseign/courant/pds/cours/cours/shell
```

5.2.3 Écriture de scripts

Un script est un programme en shell contenant une suite de commandes. Très pratique pour écrire de petites applications. C'est un fichier au format texte qui est interprété. On peut le rendre exécutable pour le lancer comme tout autre programme, mais il faut pour cela qu'il commence par `#!/bin/sh` (ou `#!/bin/bash`).

Variables spéciales :

`*` : liste des paramètres d'appels de la commande

`@` : comme `*` mais chaque paramètre est entouré de guillemets

`#` : nombre de paramètres d'appels de la commande

`$` : numéro du processus courant

`!` : numéro du dernier processus en tâche de fond

`?` : code de retour de la dernière commande lancée

0 : nom de la commande lancée

1 ... 9 : paramètres de 1 à 9

On ne peut pas accéder au paramètres d'indice 10 ou supérieur. Il faudra soit accéder à `$*`, soit utiliser `shift` qui décale les paramètres en écrasant le contenu de `$1`.

shell/simple.sh

```
#!/bin/sh

echo "Bonjour"
echo "Paramètres ___________________________________________ : $*"
echo "Nombre paramètres ____________________________________ : $#"
echo "Nom de la commande ___________________________________ : $0"
echo "Paramètres 1 et 2 ____________________________________ : $1 / $2"
shift
echo "Paramètres 1 et 2 après un shift _____________________ : $1 / $2"
echo "Process courant ______________________________________ : $$"
echo "Dernier process  en arrière-plan _____________________ : $!"
echo "Code de retour de la dernière commande effectuée ____ : $?"
```

```
# chmod +x (à ne faire qu'une seule fois)
$ chmod +x simple.sh
$ ./simple.sh
Bonjour
Paramètres                                          :
Nombre paramètres                                   : 0
Nom de la commande                                  : ./simple.sh
Paramètres 1 et 2                                   :  /
Paramètres 1 et 2 après un shift                    :  /
Process courant                                     : 11551
Dernier process  en arrière-plan                    :
Code de retour de la dernière commande effectuée    : 0
$ ./simple.sh un deux "trois 3"   "quatre    4"
Bonjour
Paramètres                                          : un deux trois 3 quatre 4
Nombre paramètres                                   : 4
Nom de la commande                                  : ./simple.sh
Paramètres 1 et 2                                   : un / deux
Paramètres 1 et 2 après un shift                    : deux / trois 3
Process courant                                     : 11557
Dernier process  en arrière-plan                    :
Code de retour de la dernière commande effectuée    : 0
```

5.2.4 Compositions de commandes

On peut composer plusieurs commandes pour obtenir une nouvelle commande

Séquence `cmd1 ; cmd2 ; cmd3` Les trois commandes sont exécutées l'une à la suite des autres. Le code de retour est celui de la dernière commande.

```
$ echo ${HOME} ; false
/home/calforme/lemaire
$ echo $?
1
$ echo ${HOME} ; true
/home/calforme/lemaire
$ echo $?
0
```

Tube `cmd1 | cmd2 | cmd3`. Les trois commandes sont exécutées simultanément, un tube connecte la sortie de `cmd1` à l'entrée de `cmd2` (et idem entre `cmd2` et `cmd3`) Le code de retour est celui de la dernière commande.

Sous-shell `( cmd )` exécute la commande `cmd` dans un sous-shell. Utile pour ne pas perturber l'environnement courant.

Conjonction/disjonction `cmd1 && cmd2 && cmd3` Exécute chaque commande l'une après l'autre mais s'interrompt dès qu'une commande renvoie un code d'erreur non nul. `cmd1 || cmd2 || cmd3` fait la même chose mais s'arrête dès qu'une commande renvoie un code d'erreur nul. Dans les deux cas, le code de retour est celui de la dernière commande lancée. Dans le premier cas, il est nul si toutes les commandes ont renvoyé un code de retour nul. Dans le second cas, il est non nul si toutes les commandes ont renvoyé un code de retour non nul.

Négation `! cmd`. Exécute la cmd mais inverse son code de retour.

```
$ ! ls > /dev/null; echo $?
1
$ ! true ; echo $?
1
$ ! (exit 4) ; echo $?
0
$ ! (exit 0) ; echo $?
1
```

5.2.5 Tests

Il n'y a pas de notion de booléen en shell. Cette notion est remplacée par le code de retour d'une commande. Il existe plusieurs commandes permettant de faire les tests habituels (comparaisons de chaînes, d'entiers, ...).

Les commandes `test ...` et `[ ... ]` sont synonymes. La commande `[[ ... ]]` est spécifique à Bash et est une version étendue de `[ ... ]`.

```
$ ls -l genereExec cmd.test
-rw-r--r-- 1 lemaire calforme 614 2007-03-12 17:13 cmd.test
-rwxr-xr-x 1 lemaire calforme 394 2007-03-12 17:24 genereExec
$ test -e genereExec ; echo $?
0
$ test -x genereExec ; echo $?
0
$ test -e inconnu ; echo $?
1
# même chose avec l'autre syntaxe
$ [ -e genereExec ]; echo $?
0
$ [ -x genereExec ]; echo $?
0
$ [ -e inconnu ]; echo $?
1
# Pas de && ou ||
# && est celui de la conjontion de commandes
$ test -e genereExec && -e cmd.test
./genereExec: line 16: -e: command not found
# && à l'intérieur de [ ... ]
$ [ -e genereExec && -e cmd.test ]
./genereExec: line 16: [: missing `]'
# pas de mélange de style, éviter :
$ [ -e genereExec ] && [ -e cmd.test ] ; echo $?
0
# et utiliser plutôt :
$ [ -e genereExec -a -e cmd.test ] ; echo $?
0
# [[ ... ]] autorise les && et ||
$ [[ -e genereExec && -e cmd.test ]] ; echo $?
0
```

5.2.6 Alternative

On utilise au choix l'une des syntaxes suivantes :

```
if cmd; then
  commandes
fi

if cmd; then
  commandes
else
  commandes
fi
```

Si vous voulez économiser des retours chariot, ne pas oublier les points virgules ...

```
  if cmd; then commandes; else commandes; fi
```

```
$ if true; then echo vrai; else echo faux; fi
vrai
$ if false; then echo vrai; else echo faux; fi
faux
```

5.2.7 Boucles

On dispose des boucles `for`, `while` et `until`. On peut faire des sauts dans l'exécution des boucles :
- `break` sort de la boucle
- `continue` passe à l'itération suivante

shell/boucles.sh

```
#!/bin/sh

echo === 1 ===

for i in $*; do
  echo $i
done

echo === 2 ===

for i in "$*"; do
  echo $i
done

echo === 3 ===

for i in $@; do
  echo $i
done

echo === 4 ===

for i in "$@"; do
  echo $i
done

echo === 5 ===

while [ -n "$1" ]; do
    echo $1
    shift
done

echo === 6 ===

A=1
until [ $A -eq 5 ]; do
    echo $A
    A=`expr $A + 1`
done
```

```
$ ./boucles.sh "1 un" 2
=== 1 ===
1
un
2
=== 2 ===
1 un 2
=== 3 ===
1
un
2
=== 4 ===
1 un
2
=== 5 ===
1 un
2
=== 6 ===
1
2
3
4
```

5.2.8 Fonctions

Une fonction est déclarée par

```
function <nom_fonction> {
  cmds;
}

ou

<nom_fonction>() {
  cmds;
}
```

Une fonction récupère ses arguments en utilisant les variables `*`, `1` ... `9`. La variable `0` ne change pas (c'est tjrs le nom du programme lancé). Une fonction renvoie une valeur entière (qui est son code de retour) en utilisant `return`. Une fonction peut donc être vue comme une nouvelle commande du shell.

shell/fonctions.sh

```
#!/bin/sh

function affiche {
    echo -n "Params : "
    while [ -n "$1" ]; do
        echo -n $1
        shift
    done
    echo
    return 0
}

affiche un deux trois
affiche 1 2 3
```

```
$ ./fonctions.sh
Params : undeuxtrois
Params : 123
```

5.2.9 Expressions mathématiques

On peut réaliser des calculer avec `expr`. La méthode est un peu artificielle, `expr` prend en entrée une expression mathématique donnée par une suite de paramètres, calcule et affiche le résultat. Tous les paramètres doivent être séparés par des espaces. On peut utiliser les opérateurs classiques `+`, `-`, `*`, `/`, `%`. Attention, le `*` doit être protégé en utilisant `\*`.

```
$ expr 4 + 5 - 2
7
$ A=4
$ B=2
$ C=`expr $A \* $B`
$ echo $C
8
```

5.2.10 Exemples

Exemples élémentaires

Réaffichage des paramètres. Trois versions.

shell/exemples/affparam1.sh

```
#!/bin/bash

for i in $*; do
    echo "paramètre" $i
done
```

shell/exemples/affparam2.sh

```
#!/bin/bash

while [ "$1" != "" ]; do
    echo "paramètre" $1
    shift
done
```

shell/exemples/affparam3.sh

```
#!/bin/bash

while [ $# -eq 0 ]; do
    echo "paramètre" $1
    shift
done
```

Somme des paramètres

shell/exemples/addparam.sh

```
#!/bin/bash

SOMME=0
while [ "$1" != "" ]; do
    SOMME=`expr $SOMME + $1`
    shift
done
echo $SOMME
```

Affichage du contenu du répertoire courant ligne par ligne

shell/exemples/myls.sh

```
#!/bin/bash

for f in *; do
    echo $f
done

# Autre méthode :
for f in `ls `; do
    echo $f
done
```

Affichage du contenu du répertoire courrant en précédant chaque fichier de son type

shell/exemples/myls2.sh

```
#!/bin/bash

for f in *; do
    if [ -f $f ]; then
        echo "Fichier : $f"
    else
        echo "Rép     : $f"
    fi
done
```

Affichage du contenu d'un répertoire. Un script prenant un paramètre désignant un répertoire et qui affiche son contenu. On teste si l'utilisateur a passé un seul paramètre et si celui ci est bien un répertoire.

shell/exemples/affrep.sh

```
#!/bin/bash

function usage {
    echo "Usage : $0 nom_de_réperoire"
}

function erreur {
    echo "Erreur : $1"
    usage
    exit
}

if [ $# -ne 1 ]; then
    erreur "Il faut exactement un paramètre"
fi

if [ ! -d $1 ]; then
    erreur "Répertoire $1 inexistant"
fi

# Remarque : on pourrait également tester si le répertoire
# est accessible en lecture et si il est exécutable (nécessaire
# pour pouvoir rentrer dedans)

ls $1
```

Calculette en ligne de commande.Un programme qui demande en boucle une chaîne de caractères supposée être une expression mathématique. Le script calcule la valeur correspondante et l'affiche. Si l'expression est incorrecte, le programme l'indique. On arrête lorsque l'utilisateur tape quit.

shell/exemples/calcule.sh

```
#!/bin/bash

QUIT=0
while [ "$QUIT" = "0" ]; do
    read EXP
    if [ "$EXP" = "quit" ]; then
        QUIT=1
    else
        RES=`expr $EXP`
        # On lance le calcul
        # on récupère le code d'erreur
        ERR=$?
        if [ $ERR -ge 2 ]; then
            echo "Expression incorrecte : $EXP"
        else
            echo "$RES"
        fi
    fi
done
```

Lancement et trace de script

Pour rédiger ce chapitre, j'ai utilisé deux scripts dont les sources sont donnés plus bas. Le premier est `genereExec` qui lit une commande sur l'entrée standard, l'affiche, l'exécute, et continue avec les commandes suivantes. cela permet d'obtenir un affichage ressemblant à une session shell, dans laquelle on aurait tapé toutes les commandes à la main. Le script `genereAll` transforme chaque fichier de commandes `cmd.fichier` en `out.fichier` en utilisant `genereExec`.

shell/genereExec

```
#!/bin/bash

# on supprime titi, ...
rm -rf titi toto tutu

RET=0
# option -r sur read pour désactiver l'interprétation des \
while read -r line; do
    if ! echo "$line" | grep -q "^[ ]*$"; then
        if echo "$line" | grep -q "^#"; then
            echo "$line"
        else
            echo "$ $line"
            # permet de repositionner l'ancien *?
            (exit $RET)
            eval $line
            RET=$?
        fi;
    fi;
done

exit 0
```

shell/genereAll

```
#!/bin/bash

TMP=/tmp/pds

for FIC in cmd.*[^~]; do
    OUT=`echo $FIC | sed -e s/^cmd/out/`
    echo "$FIC -> $OUT"

    rm -f $OUT

    ./genereExec < $FIC > $OUT 2>&1
    if [[ "$?" -ne "0" ]]; then
        echo "Erreur sur $FIC";
        exit 1
    fi;
done
```

Exemple de fichier de commande, et de sa sortie :

shell/cmd.variables

```
echo $TOTO
TOTO=toto
TITI="Salut $TOTO"
echo $TITI
TITI='Salut $TOTO'
echo $TITI
TATA=`pwd`
echo $TATA
```

```
$ echo $TOTO

$ TOTO=toto
$ TITI="Salut $TOTO"
$ echo $TITI
Salut toto
$ TITI='Salut $TOTO'
$ echo $TITI
Salut $TOTO
$ TATA=`pwd`
$ echo $TATA
/home/calforme/lemaire/enseign/courant/pds/cours/cours/shell
```

Chapitre 6

Système d'exploitation

6.1 Présentation

Le système d'exploitation (SE, en anglais Operating System ou OS) est un ensemble de programmes responsables de la liaison entre les ressources matérielles d'un ordinateur et les applications de l'utilisateur.

Différents composants d'un SE classique :
- un noyau ;
- un système de fichiers ;
- bibliothèques dynamiques ;
- un ensemble d'outils système ;
- des programmes applicatifs de base.

Quelques systèmes d'exploitation :
- 1960 : mini-ordinateurs, le début d'Unix (Ken Thompson, Dennis Ritchie and Douglas McIlroy, Bell Labs)
- 1980 : CP/M, MS-DOS, Mac OS
- 1990-2007 : windows (98, NT, 2000, XP, ...), Linux, Mac OSX, ...

La figure 6.1 montre l'évolution des systèmes Unix.

Le développement d'applications systèmes est lié au système d'exploitation. Pour faciliter leur écriture, des standards ont été définis (voir POSIX).

6.2 Programmation système

Un appel système est une fonction fournie par le noyau d'un système d'exploitation permettant d'accéder aux ressources (au sens large) du système.

Le rôle du noyau est de gérer :
- les ressources matérielles (il contient des pilotes de périphériques) ;
- de fournir aux programmes une interface uniforme pour l'accès aux ressources ;
- l'exécution des processus (attribution mémoire, ordonnancement des processus, synchronisation, ...) ;
- les systèmes de fichiers ;
- ...

Le noyau fournit des mécanismes d'abstraction du matériel, notamment de la mémoire, du (ou des) processeur(s), et des échanges d'informations entre logiciels et périphériques matériels.

Exemples d'appels systèmes : `open`, `read`, `write` et `close` qui permettent les manipulations sur les fichiers.

La figure 6.2 montre les interactions entre les différentes parties du SE.

Un processus classique s'exécute habituellement en mode utilisateur, un processus système (faisant partie du noyau) habituellement en mode superviseur. Le mode utilisateur est limité (accès à la mémoire, aux disques, aux périphériques vérifiés, vérification du code exécutable, ...) ce qui rend le système stable. Le mode superviseur a tous les droits (un bogue dans le système est très souvent fatal). Ces deux modes correspondent classiquement chacun à deux modes du processeur.

Sur la plupart des noyaux (notamment les noyaux monolithiques comme le Noyau Linux) les appels systèmes font passer du mode utilisateur au mode superviseur.

Un système d'exploitation comme Linux a plus de 200 appels systèmes distincts (dont certains se recoupent ou offrent des fonctionnalités similaires : read, pread, ...).

6.3 Le standard POSIX

POSIX est le nom d'une famille de standards définie depuis 1988 par l'IEEE et formellement désignée IEEE 1003. Ces standards ont émergé d'un projet de standardisation des APIs des logiciels destinés à fonctionner sur des variantes du système d'exploitation UNIX. Le terme POSIX est dû à Richard Stallman : "Portable Operating System Interface", dont le X exprime l'héritage UNIX de l'API.

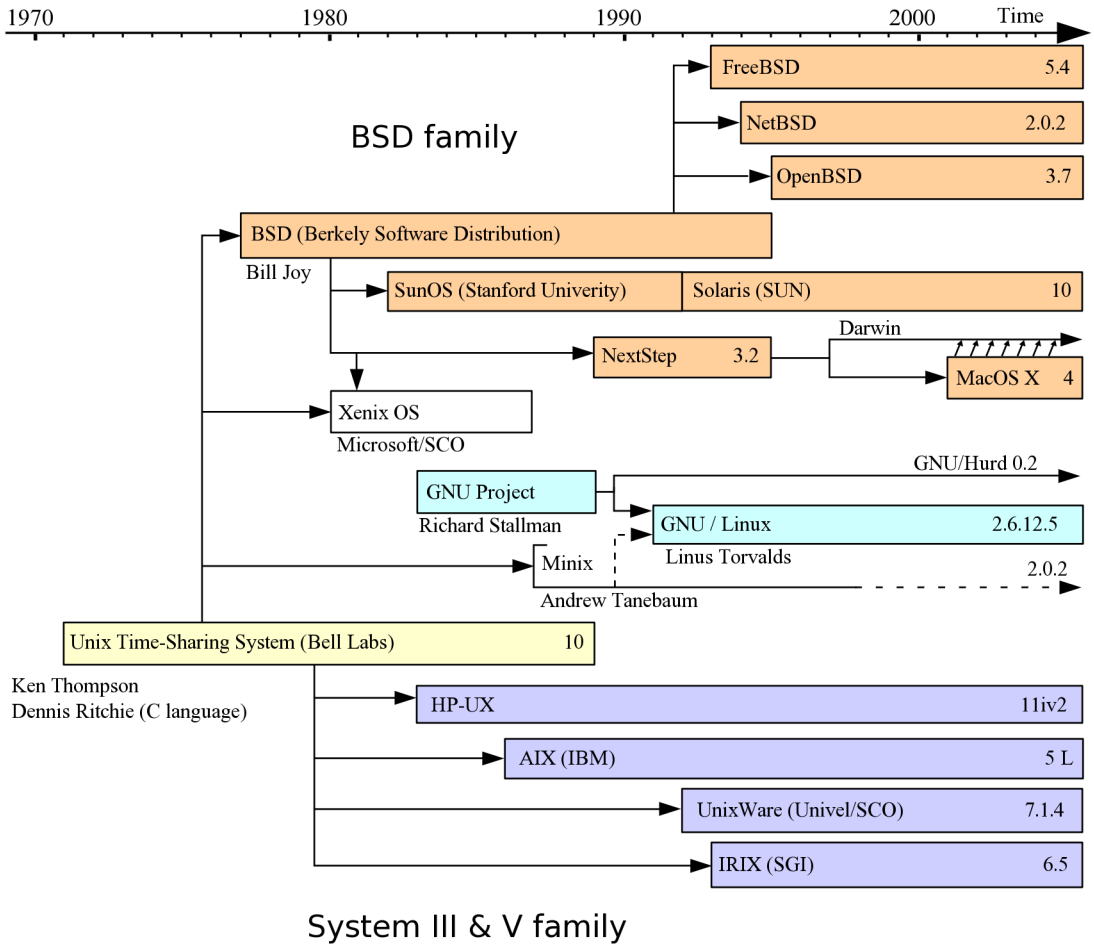


FIG. 6.1 – Histoire d'Unix

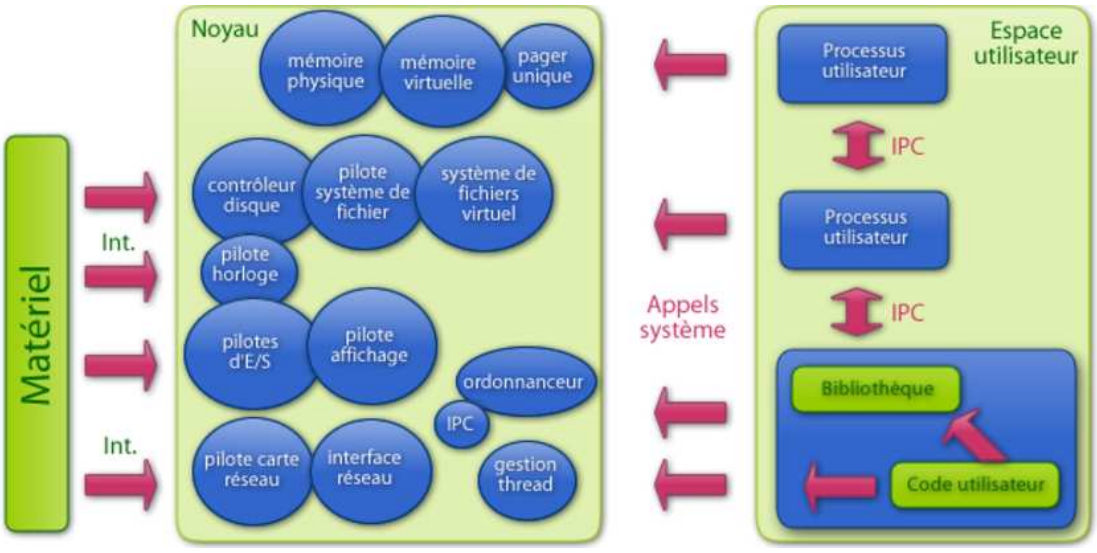


FIG. 6.2 – Mode d'exécution

La documentation est composée de trois parties :
- les API (POSIX.1 :core (inclut ANSI C), POSIX.1b (extensions temps réel), POSIX.1c (les threads, gestion des signaux)
- les commandes et utilitaires POSIX (extensions de portabilité des utilisateurs, corrections et extensions, utilitaire de protection et de contrôle, utilitaires pour le traitement par lots)
- test de conformité à POSIX.

Pour information, la iste complète des appels systèmes est disponible sur `http://www.lynuxworks.com/products/posix/function-calls.php3`

POSIX.1 : core Process Creation and Control, Signals, Floating Point Exceptions, Segmentation Violations, Illegal Instructions, Bus Errors, Timers, File and Directory Operations, Pipes, C Library (Standard C), I/O Port Interface and Control.

POSIX.1b : extensions temps-réel Priority Scheduling, Real-Time Signals,Clocks and Timers, Semaphores, Message Passing, Shared Memory, Asynch and Synch I/O, Memory Locking.

POSIX.1c : threads extensions Thread Creation, Control, and Cleanup, Thread Scheduling, Thread Synchronization, Signal Handling.

Quelques systèmes conformes à POSIX : Solarix, HP-UX, Mac OS X, Windows (NT, 2000, XP, Vista, ... sous conditions), Linux, NetBSD, OpenBsd[1], ...

La partie interface de POSIX permet la portabilité des applications.

Les interfaces de POSIX et de Unix sont (quasiment) identiques ! On parle d'interface native : une fonction POSIX correspond à un appel système.

Les autres systèmes sont rendus conformes à POSIX par le biais d'une interface POSIX (différente de l'interface système).

Pour des raisons financières (certification et documents POSIX payants et très coûteux), le Single Unix Specification a été créé dans les années 80. Dernière version en date "Single Unix Specification version 3, 2001" (`http://www.unix.org/version3/`). C'est le document de certification des systèmes Unix.

6.3.1 Bibliothèque C standard et POSIX

La bibliothèque POSIX est décrite la page 2 du man (`man 2 sbrk`). La bibliothèque C ANSI est décrite dans la page 3 du man `man 3 malloc`. Les deux spécifications sont imbriquées car elles ont un historique commun, d'où une certaine confusion.

Nous n'utiliserons que des appels faisant partie de POSIX (et C ANSI).

6.3.2 Gestion des erreurs

Les appels systèmes (ainsi que certaines fonctions de la bibliothèque C) gère les erreurs en positionnant la variable globale `errno` à une valeur spéciale. L'appel système renvoie en général -1 (ou parfois NULL) pour signifier que l'appel a échoué. On peut traiter l'erreur en analysant la valeur de `errno` ou en appelant la fonction `perror` qui affiche un message d'erreur approprié. Voir `man errno`.

Exemple ▷

```
$ man access
...
RETURN VALUE
       On  success  (all requested permissions granted), zero is returned.  On
       error (at least one bit in mode asked for a permission that is  denied,
       or  some other error occurred), −1 is returned, and errno is set appro−
       priately.
....
```

systeme/erreur.c

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

int main() {

  int out;
  out = access("/etc/shadow", R_OK);
  printf("Code retour : %d\n", out);
  if (out == -1) {
    printf("errno vaut : %d\n", errno);
    perror("Message compréhensible");
    exit(EXIT_FAILURE);
  } else {
    printf("Je peux lire le fichier de mots de passe !\n");
```

[1] Linux, NetBSD et OpenBsd ne sont pas officiellement certifiés POSIX

```
    exit(EXIT_SUCCESS);
  }

}
```

```
$ gcc -Wall -ansi -o erreur erreur.c
$ ./erreur
Code retour : -1
errno vaut : 13
Message compréhensible: Permission denied
```

◁

Chapitre 7

Système de fichiers et entrées/sorties

Un système de fichiers (FS ou FileSystem en anglais) ou système de gestion de fichiers (SGF) est une méthode permettant de stocker les informations et de les organiser dans des fichiers sur ce que l'on appelle des mémoires secondaires (disque dur, disquette, CD-ROM, clé USB, etc.). Une telle gestion des fichiers permet de traiter, de conserver des quantités importantes de données ainsi que de les partager entre plusieurs programmes informatiques. Il offre à l'utilisateur une vue abstraite sur ses données. Le système de fichier fait partie du système d'exploitation.

Un système de fichiers est présenté par une arborescence : les fichiers sont regroupés dans des répertoires (concept utilisé par la plupart des systèmes d'exploitations). Ces répertoires contiennent soit des fichiers, soit des répertoires, il y a donc un répertoire racine et des sous-répertoires. Une telle organisation permet d'obtenir une hiérarchie de répertoires et de fichiers organisés en arbres.

Quelques exemples de systèmes de fichiers :
- ext2/3/4 (Linux)
- NTFS (Windows 2000/XP,NT, Linux, Max OS X)
- ReiserFS (Linux)
- HFS+ (Mac OS X, Linux)
- ISO 9660 : en lecture seule sur tous les systèmes lisant les CDROM/DVDROM de données

7.1 Systèmes de fichiers sous Unix

7.1.1 Arborescence

On accède aux différents systèmes de fichiers par une arborescence unique, dont la racine est `/`. L'arborescence est composée de :
- noeuds terminaux (fichiers ordinaires/spéciaux, liens symboliques, ...)
- noeuds non terminaux (répertoires)

Chaque répertoire a toujours deux fils
- "." : le noeud lui même
- ".." : un lien vers le père

On désigne un noeud de l'arborescence par
- un chemin absolu : /rep1/rep2/.../[fichier—répertoire] (ex : /home/calforme/lemaire/.emacs)
- un chemin relatif : rep1/rep2/.../[fichier—répertoire] (ex : ../tp2/Makefile)

7.1.2 Montage de systèmes de fichiers

La commande `mount` affiche les partitions ainsi que leur point de montage.

```
$ mount
/dev/sda6 on / type ext3 (rw,noatime)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec)
udev on /dev type tmpfs (rw,nosuid)
devpts on /dev/pts type devpts (rw,nosuid,noexec)
/dev/sda7 on /var type ext3 (rw,noatime)
/dev/sda8 on /usr type ext3 (rw,noatime)
/dev/sda9 on /export type ext3 (rw,noatime)
shm on /dev/shm type tmpfs (rw,noexec,nosuid,nodev)
usbfs on /proc/bus/usb type usbfs (rw,noexec,nosuid,devmode=0664,devgid=85)
nfsd on /proc/fs/nfs type nfsd (rw)
rpc_pipefs on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
automount(pid5389) on /home type autofs (rw,fd=4,pgrp=5389,minproto=2,maxproto=4)
livinus:/vol/home/calforme on /home/calforme type nfs (rw,soft,addr=134.206.10.24,addr=134.206.10.24)
livinus:/vol/home/.MAIL on /home/.MAIL type nfs (rw,soft,addr=134.206.10.24,addr=134.206.10.24)
```

Exemple ▷ Pour afficher les systèmes de fichiers montés de type ext3

```
$ mount -t ext3
/dev/sda6 on / type ext3 (rw,noatime)
/dev/sda7 on /var type ext3 (rw,noatime)
/dev/sda8 on /usr type ext3 (rw,noatime)
/dev/sda9 on /export type ext3 (rw,noatime)
```

Pour afficher les systèmes de fichiers montés par nfs (Network File System)

```
$ mount -t nfs
livinus:/vol/home/calforme on /home/calforme type nfs (rw,soft,addr=134.206.10.24,addr=134.206.10.24)
livinus:/vol/home/.MAIL on /home/.MAIL type nfs (rw,soft,addr=134.206.10.24,addr=134.206.10.24)
```

Pour monter une clé Usb :

```
$ mount /dev/sdb1 /mnt/clusb
$ umount /mnt/clusb
```

◁

7.1.3 Différents types de fichiers

Sous Unix, tout est fichier !
- fichiers ordinaires ;
- répertoires ;
- liens symboliques (raccourci vers un autre fichier) ;
- tubes nommés, sockets ;
- fichiers associés à des périphériques (terminaux, cdrom, imprimantes, souris, ...), exemple : `cat /dev/input/mouse0`
- ...

Lien symbolique (soft link) C'est un fichier contenant uniquement un chemin relatif ou absolu.

```
$ ls
fichier.txt
$ cat fichier.txt
Exemple
$ ln -s fichier.txt lien.txt
$ ls -l
total 0
-rw-r--r-- 1 lemaire calforme  8 2007-02-20 14:08 fichier.txt
lrwxrwxrwx 1 lemaire calforme 11 2007-02-20 14:09 lien.txt -> fichier.txt
$ cat lien.txt
Exemple
$ rm lien.txt
$ ln -s inconnu.txt lien.txt
$ cat lien.txt
cat: lien.txt: No such file or directory
```

Lien physique (hard link) Deux fichiers peuvent partager les mêmes données.

```
$ ls -l
total 12
-rw-r--r-- 1 lemaire calforme    8 2007-02-20 14:08 fichier.txt
$ cat fichier.txt
Exemple
$ ln fichier.txt copie.txt
$ ls -l
total 0
-rw-r--r-- 2 lemaire calforme 8 2007-02-20 14:08 copie.txt
-rw-r--r-- 2 lemaire calforme 8 2007-02-20 14:08 fichier.txt
$ cat copie.txt
Exemple
$ echo Suite >> copie.txt
$ ls -l
total 0
-rw-r--r-- 2 lemaire calforme 14 2007-02-20 14:15 copie.txt
-rw-r--r-- 2 lemaire calforme 14 2007-02-20 14:15 fichier.txt
$ cat fichier.txt
Exemple
Suite
$
```

On ne peut pas créer de lien physique sur des répertoires.

Tubes nommés Ce sont des fichiers gérés comme des tubes. Permet de connecter deux processus (non apparentés) qui dialoguent par le tube nommé. On écrit d'un côté et on lit de l'autre !

```
# Dans terminal 1
$ mknod p tube
$ ls -l tube
$ prw-r--r-- 1 lemaire calforme 0 2007-03-23 17:12 tube
$ cat tube
......bloqué....
```

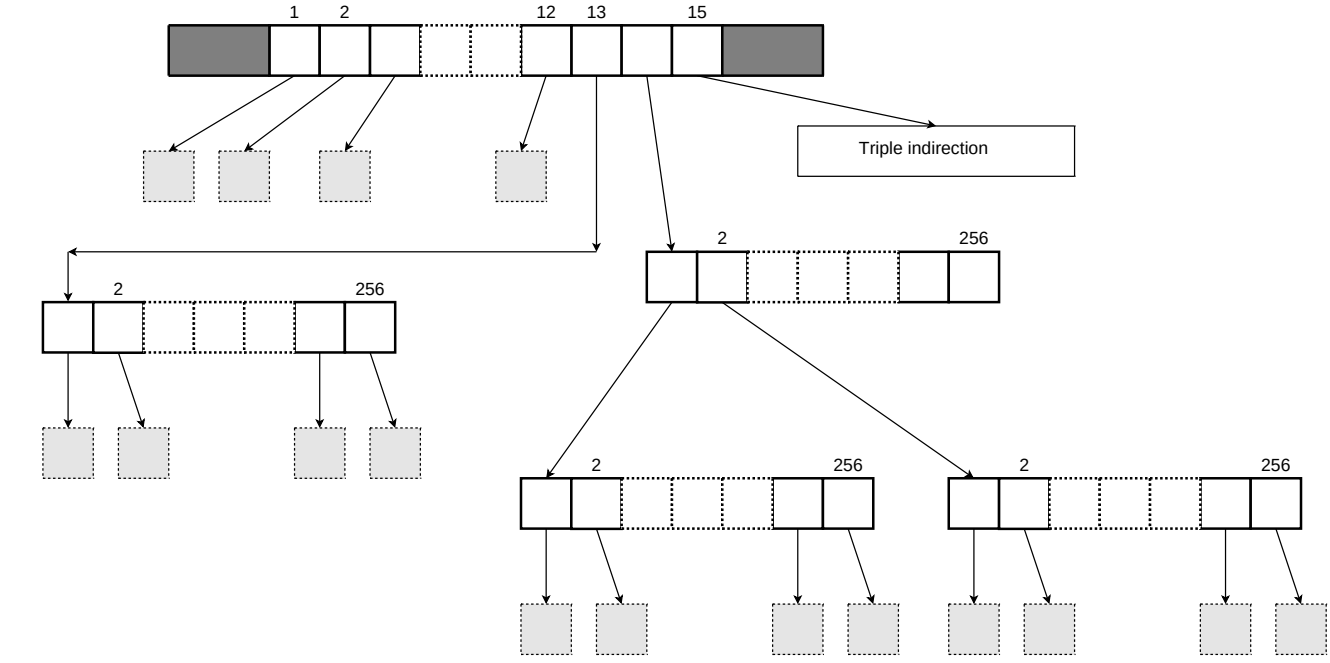
```
# Dans terminal 2
$ echo bonjour > tube
$
# De retour dans terminal 1, cat est débloqué
bonjour
$
```

Fichiers spéciaux Les périphériques sont vus comme des fichiers qui appartiennent au répertoire `/dev`. Ils sont décrits par deux nombres (visible par `ls -l`) : major répère le type de périphérique, minor repère son numéro.

```
$ ls -l /dev/mem
crw-r----- 1 root root 1, 1 2007-03-23 13:59 /dev/mem
$ ls -l /dev/sda
brw-rw---- 1 root disk 8, 0 2007-03-23 13:59 /dev/sda
$ ls -l /dev/sda5
brw-rw---- 1 root disk 8, 5 2007-03-23 13:59 /dev/sda5
$ ls -l /dev/input/mouse0
crw-r--r-- 1 root root 13, 32 2007-03-23 13:59 /dev/input/mouse0
$ cat /dev/input/mouse0 | od -N 10
0000000 177030 034000 177775 176070 034376
0000012
$ cat /dev/sound/audio | od -N 10
0000000 077577 077577 077577 077577 077577
0000012
$ cat /dev/random | od -N 10
0000000 104200 156430 044424 132751 072162
0000012
```

7.1.4 Structure d'un SF

Le SF est morcelé en blocs (habituellement 1024 octets). Le SF se compose d'un superbloc (reservé système), d'une table d'inœuds, suivi de l'espace de données (qui contient donc les blocs de données). Rqe : les chiffres donnés dans cette section peuvent changer d'un SF à un autre.



Inœud

Un inœud contient les caractèristiques d'un fichier (au sens large) comme par exemple :
- droits ;
- utilisateur et groupe propriétaires ;
- taille ;
- dates dernière modification ;
- nombre de liens physiques ;
- numéros de blocs de données ;
- ...

Rqe : le nom du fichier ne figure pas dans l'inœud.

Les données sont stockées dans l'espace de données, et sont référencées par les numéros de blocs de données. Les blocs sont référencés par leur numéro. Afin de limiter la taille des inœuds, un mécanisme de triple indirection est mis en place pour référencer tous les blocs de données d'un inœud.

Un inœud contient 15 numéros de blocs

- les 12 premiers sont les 12 premiers blocs de données
- le numéro 13 est un numéro de bloc (de données), contenant 256 numéros de blocs (en supposant qu'un numéro de bloc occupe 4 octets et qu'un bloc fait 1024 octets). Ce bloc est un bloc d'indirection.
- le numéro 14 est le numéro du bloc de double indirection : ce bloc contient 256 numéros de bloc qui sont chacun un bloc d'indirection
- le numéro 15 est le numéro de bloc de triple indirection

Cette structure favorise les petits fichiers (aucune indirection) et permet l'accès direct aux données (on trouve en temps constant le numéro d'un bloc à partir d'une position donnée dans un fichier).

On peut afficher les numéros d'inœuds avec `ls -i`. On peut même décortiquer un inœud avec la commande `debugfs` (en mode admin seulement)

```
$ ls -i /usr/lib/libc.a
2830520 /usr/lib/libc.a
$ debugfs /dev/sda8
debugfs 1.39 (29-May-2006)
debugfs: stat <2830520>

Inode: 2830520   Type: regular    Mode:  0644   Flags: 0x0   Generation: 1111443
532
User:     0   Group:     0   Size: 4069530
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 7960
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x45d1f91f -- Tue Feb 13 18:45:03 2007
atime: 0x45d1f91e -- Tue Feb 13 18:45:02 2007
mtime: 0x45d1f91e -- Tue Feb 13 18:45:02 2007
BLOCKS:
(0-11):5706851-5706862, (IND):5706863, (12-92):5706864-5706944, (93-521):5712670
-5713098, (522-532):5713100-5713110, (533-891):5713112-5713470, (892-900):571353
0-5713538, (901-927):5713597-5713623, (928-989):5713625-5713686, (990-993):57136
99-5713702
TOTAL: 995
```

Implantation d'un fichier/répertoire

Les inœuds ne possèdent pas de nom de fichier. Un répertoire possède un inœud, et les données de l'inœud sont une suite d'entrées de la forme (inœud, nom). Un répertoire occupe donc de la place disque (la commande `ls -l` affiche la taille occupée par les entrées du répertoire, pas la taille des entrées du répertoire). nom peut être :

- le nom d'un fichier ordinaire (ce qui explique qu'un inœud puisse être partagé par plusieurs fichiers ordinaires) ;
- le nom d'un répertoire ;
- ...

Par convention, le répertoire racine est situé à l'inœud numéro 2. Pour info, le père du répertoire racine est lui même.

```
$ ls -di /
2 /
$ ls -di /tmp
326401 /tmp
$ls -di /usr
2 /usr
# Pourquoi /usr a aussi l'inoeud 2 ???
```

Rqe : c'est l'association (inœud, nom) qui permet de créer des liens physiques.

7.2 Interface POSIX du système de fichiers

Lire un fichier dépend du type du fichier !

- fichier ordinaire
  - accéder aux données
- répertoire
  - accéder à la liste des entrées
- lien symbolique
  - soit on traite le fichier pointé
  - soit on manipule le lien lui-même
- fichier spécial
  - accéder à un périphérique
  - identique au cas d'un fichier ordinaire
  - limitations et opérations spécifiques possibles

L'interface POSIX permet d'abstraire les fichiers (ex : on peut ouvrir n'importe quel fichier avec `open`)

entreessorties/opendevmem.c

```
/*****************************************************************
```

```
 *opendevmem.c
 *
 *  (François lemaire)  <lemaire@lifl.fr>
 * Time-stamp: <2007-03-23 17:47:56 lemaire>
 ************************************************************************/

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

int main(int argc, char *argv[]) {
  int fd;
  char buf[1024];
  int i;
  fd = open("/dev/mem", "r");
  if (fd!=-1) {
    read(fd, buf, 1024);
    close(fd);
    for(i=0;i<1024;i++)
      printf("%hhu\n", buf[i]);
    exit(EXIT_SUCCESS);
  } else {
    perror("");
    exit(EXIT_FAILURE);
  }
}

/*
$ ./opendevmem
Permission denied
$ su
$ /opendevmem
1
0
...
$
*/
```

7.2.1 Informations

Les fonctions `stat`, `lstat` permettent de récupérer les informations sur un fichier (`lstat` déréférence le chemin dans le cas d'un lien symbolique).

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

int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);

struct stat {
              dev_t     st_dev;     /* ID of device containing file */
              ino_t     st_ino;     /* inode number */
              mode_t    st_mode;    /* protection */
              nlink_t   st_nlink;   /* number of hard links */
              uid_t     st_uid;     /* user ID of owner */
              gid_t     st_gid;     /* group ID of owner */
              dev_t     st_rdev;    /* device ID (if special file) */
              off_t     st_size;    /* total size, in bytes */
              blksize_t st_blksize; /* blocksize for filesystem I/O */
              blkcnt_t  st_blocks;  /* number of blocks allocated */
              time_t    st_atime;   /* time of last access */
              time_t    st_mtime;   /* time of last modification */
              time_t    st_ctime;   /* time of last status change */
          };
```

- numéro du périphérique et de l'inœud :

```
struct stat {
              dev_t     st_dev;     /* ID of device containing file */
              ino_t     st_ino;     /* inode number */
```

- nombre de liens physiques :

```
struct stat {
              ....
              nlink_t   st_nlink;   /* number of hard links */
```

- propriétaire, groupe, type de fichier

```
struct stat {
              ....
              uid_t     st_uid;     /* user ID of owner */
              gid_t     st_gid;     /* group ID of owner */
              mode_t    st_mode;    /* protection */
```

Le champ mode (`st_mode`) contient des informations sur le type de fichiers, les droits du propriétaire (`m` designe le champ `st_mode`) :

- `S_ISREG(m)` : fichier ordinaire
- `S_ISDIR(m)` : répertoire

- `S_ISLNK(m)` : lien
- `S_ISSOCK(m)`, `S_ISFIFO(m)`, ...
- `m & S_IFREG`, `m & S_IFDIR`, `m & S_IFLNK`
- `m & S_IRUSR` propriétaire peut lire
- `m & S_IWUSR` propriétaire peut écrire
- `m & S_IXUSR` propriétaire peut éxécuter

Voir `man stat.h` pour les macros `S_ISREG`, `S_IRUSR`, ...

- dates de modifications

```
struct stat {
              ....
              time_t     st_atime;   /* time of last access */
              time_t     st_mtime;   /* time of last modification */
              time_t     st_ctime;   /* time of last status change */
          };
```

- temps depuis EPoch = 1er Janvier 1970
- taille totale, nombre de blocs alloués, taille d'un bloc

```
struct stat {
              off_t      st_size;    /* total size, in bytes */
              blksize_t  st_blksize; /* blocksize for filesystem I/O */
              blkcnt_t   st_blocks;  /* number of blocks allocated */
          };
```

7.2.2 Droits d'accés

On peut aussi accéder aux droits avec `access` .

```
#include <unistd.h>
int access(const char *pathname, int mode);
```

- `amode` est une combinaison de `R_OK` (lecture), `W_OK` (écriture), `X_OK` (exécution) et `F_OK` (existence).
- renvoie 0 si le fichier a les droits de `mode`
- renvoie -1 si erreur, et `errno` est positionné.

On peut modifier les droits :

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

- `mode` est une combinaison de `S_IRUSR`, `S_IWUSR`, ... ou un code en octal
- renvoie 0 succès
- renvoie -1 si erreur (errno positionné)

7.2.3 Liens symboliques/physiques

Création

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
int symlink(const char *oldpath, const char *newpath);
```

- link : crée un lien physique `newpath` sur `oldpath` supposé exister
- symlink : crée un lien symbolique `newpath` sur `oldpath` qui peut ou non exister.

7.2.4 Répertoires

Création

On crée un répertoire avec `mkdir`

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

int mkdir(const char *pathname, mode_t mode);
```

- `pathname` : nom du répertoire à créer (un seul niveau d'arborescence peut être créé)
- `mode` : mode (droits) du répertoire créé

La fonction `mkdir` renvoie 0 si succès, et -1 si erreur (voir errno).

Le mode du répertoire créé est modifié par le masque courant : `mode & ~umask & 0777`. Pour fonctionner, le droit en écriture sur le répertoire père est requis.

Le masque courant peut être changé en utilisant :

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

mode_t umask(mode_t mask);
```

La fonction `umask` change le masque et renvoie l'ancien masque.

```
  int status;

  status = mkdir("/tmp/titi", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
  // ou pour le mode : 0775
```

Lecture

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

DIR *opendir(const char *name);
int closedir(DIR *dir);
```

- `opendir` renvoie un descripteur de répertoire, qui permet d'itérer toutes les entrées du répertoire
- renvoie `NULL` en cas d'erreur (voir errno)
- `closedir` ferme le descripteur

On peut itérer les entrées avec `readdir` :

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

struct dirent *readdir(DIR *dir);
```

Renvoie un pointeur sur une structure contenant les informations. Renvoie NULL si erreur ou si fin du répertoire atteinte. POSIX impose les champs `d_name` et `d_ino`.

Sous Linux :

```
struct dirent {
              ino_t          d_ino;       /* inode number */
              off_t          d_off;       /* offset to the next dirent */
              unsigned short d_reclen;    /* length of this record */
              unsigned char  d_type;      /* type of file */
              char           d_name[256]; /* filename */
          };
```

entreessorties/exemples/lirerep.c

```
/*************************************************************************
 *lirerep.c
 *
 *  (François lemaire)  <lemaire@lifl.fr>
 * Time-stamp: <2007-03-19 17:01:01 lemaire>
 *************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#include <dirent.h>
#include <errno.h>

int main() {

  DIR *dirp;
  struct dirent *dp;

  dirp = opendir(".");
  if (dirp==NULL) {
    perror("Ouverture repertoire : ");
    exit(EXIT_FAILURE);
  }

  while ( (dp=readdir(dirp)) !=NULL ) {
    printf("Noeud : %ld Entree : %s\n", dp->d_ino, dp->d_name);
  }
  if (errno!=0) {
    perror("Entree répertoire : ");
    exit(EXIT_FAILURE);
  }

  closedir(dirp);
  exit(EXIT_SUCCESS);
}
```

Suppression

Un répertoire peut se supprimer avec `rmdir`. Le répertoire doit être vide.

```
#include <unistd.h>

int rmdir(const char *pathname);
```

7.2.5 Fichiers ordinaires

Création/ouverture

On crée/ouvre un fichier avec la commande `open`.

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- renvoie un descripteur de fichier (numéro) ou -1 si erreur (voir errno) utile pour lectures/écritures/...
- crée une entrée dans la table système des fichiers ouverts
- `mode` n'est utilisé que lors de la création (et combiné avec le masque de création courant)
- `flags` doit contenir exactement un des modes `O_RDONLY`, `O_WRONLY`, `O_RDWR`
- `flags` peut contenir d'autres modes :
  - `O_APPEND` mode ajout (avant chaque écriture, le curseur est redéplacé à la fin)
  - `O_CREAT` crée le fichier si il n'existe pas
  - `O_CREAT|O_EXCL` crée le fichier mais échoue si le fichier existe déjà
  - `O_TRUNC` tronque le fichier si il existe déjà et qu'on est dans l'un des deux modes en écriture
  - `O_SYNC`, `O_NONBLOCK`

Exemple ▷

```
int fd;
// ouvre un fichier supposé existant
fd = open(path, O_RDONLY);
// ouvre un fichier en écriture, le tronque si existant, et le crée sinon
fd = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
```

◁

Un même fichier peut être ouvert plusieurs fois. Sans précautions, les lectures/écritures entrainent une corruption des données (nécessité de verrous).

entreessorties/exemples/ecrituremult.c

```
/*****************************************************************
 * ecrituremult.c
 *
 *  (François lemaire) <lemaire@lifl.fr>
 * Time-stamp: <2007-03-19 17:53:44 lemaire>
 *****************************************************************/

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

int main(int argc, char *argv[]) {
  int fd1, fd2;

  fd1 = open("tata", O_WRONLY|O_CREAT|O_TRUNC, 0666);
  fd2 = open("tata", O_WRONLY|O_CREAT|O_TRUNC, 0666);

  printf("fd1 : %d   fd2 : %d\n", fd1, fd2);

  write(fd1, "bonjour", 8);
  write(fd2, "au revoir", 10);
  write(fd1, "bonjour", 8);

  close(fd2);
  close(fd1);
  exit(EXIT_SUCCESS);
}

/*
$ ./ecrituremult
$ cat tata
au revoibonjour
*/
```

Fermeture

On utilise la fonction `close` :

```
#include <unistd.h>

int close(int fd);
```

Renvoie 0 si ok, -1 si erreur (voir errno)

Lecture/écriture

Elles se font avec les fonctions `read` et `write`

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

- `fd` : descripteur de fichier
- `count` : nombre d'octets à lire/écrire
- `buf` : zone mémoire où lire/écrire
- `read` renvoie le nombre d'octets lus (-1 si erreur, voir errno) :
  - habituellement égal à `count`
  - inférieur à `count` en cas de lecture vers la fin du fichier
  - 0 si on est sûr que plus aucune donnée ne pourra être lue (en cas de lecture et de fin de fichier, dans un tube sans écrivain, ...) ;
- `write` renvoie le nombre d'octets écrits (-1 si erreur, voir errno). Ce nombre est inférieur ou égal à `count`. On n'a pas la garantie que les données soient effectivement écrites sur le système de fichier
- les deux fonctions déplacent le curseur dans le fichier en conséquence (i.e. du nombre d'octets lus/écrits)

La norme POSIX impose qu'un `read` se déroulant après un `write` doit recevoir les nouvelles données.

Une copie de fichier avec `read` et `write`.

entreessorties/exemples/copie.c

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

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

#define BUFSIZE 8

int main(int argc, char *argv[]) {
  int fdsrc, fddst, nb;
  char buffer[BUFSIZE];

  if (argc<3) {
    fprintf(stderr, "%s f1 f2\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  fdsrc = open(argv[1], O_RDONLY);
  if (fdsrc==-1) {
    perror("Ouverture du fichier source");
    exit(EXIT_FAILURE);
  }

  /* le mode est passé en octal : mettre un 0 devant le nombre ! */
  fddst = open(argv[2], O_WRONLY|O_CREAT|O_EXCL, 0666);
  if (fddst==-1) {
    perror("Ouverture du fichier destination");
    exit(EXIT_FAILURE);
  }

  while ((nb=read(fdsrc, buffer, BUFSIZE))) {
    write(fddst, buffer, nb);
  }

  close(fdsrc);
  close(fddst);
  exit(EXIT_SUCCESS);
}
```

Attention ne pas mélanger les fonctions ANSI et les appels système : la bibliothèque ANSI s'appuie sur les appels systèmes !

7.2.6 Suppression de fichiers

```
#include <unistd.h>

int unlink(const char *pathname);
```

Fonctionne sur les fichiers ordinaires, tubes nommés, liens symboliques, sockets, ... Ne pas utiliser sur un répertoire (corruption du système de fichiers).

Si `pathname` était le dernier lien vers le fichier, et qu'aucun processus n'a ouvert le fichier, le fichier est détruit et l'espace libéré.

Si `pathname` était le dernier lien vers le fichier mais qu'un processus avait ouvert le fichier, le fichier existe jusqu'à ce que le processus le ferme (pendant ce temps, le fichier n'apparaît plus dans l'arborescence).

7.3 Interface POSIX : opérations avancées

7.3.1 Positionnement

Les écritures/lectures se font à la position courante dans le fichier, et cette position courante est modifiée par les écritures/écritures. On peut modifier la position courante avec `lseek` (attention : impossible sur certains types de fichiers comme les tubes, socket, FIFO, ...).

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

off_t lseek(int fildes, off_t offset, int whence);
```

- `offset` est un décalage en octets (positif ou négatif)
- `whence` est l'un des mots clés :
  - `SEEK_SET` : la position courante devient `offset`
  - `SEEK_CUR` : la position courante est décalée de `offset`
  - `SEEK_END` ; la position courante devient la fin du fichier plus `offset`
- renvoie la nouvelle position (en octets), ou -1 si erreur (voir errno)

On peut positionner la position courante au delà de la fin du fichier pour faire des écritures. La taille du fichier change et on crée un fichier creux : toute lecture dans le creux renvoie zéro.

entreessorties/exemples/creux.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

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

#define NZ 32768


int main(int argc, char *argv[]) {
  char debut[] = "abcd";
  char fin[] = "wxyz";

  int fddst, i;
  char zero;

  if (argc<3) {
    fprintf(stderr, "%s -a|-s fichier\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  fddst = open(argv[2], O_WRONLY|O_CREAT|O_EXCL, 0666);
  if (fddst==-1) {
    perror("Ouverture du fichier destination");
    exit(EXIT_FAILURE);
  }

  write(fddst, debut, sizeof(debut));

  if (!strcmp(argv[1], "-a")) {
    // avec un trou
    lseek(fddst, NZ, SEEK_CUR);
  } else {
    zero=0;
    for (i=0;i<NZ;i++) write(fddst, &zero, 1);
    // sans trou
  }


  write(fddst, fin, sizeof(fin));

  close(fddst);

  exit(EXIT_SUCCESS);
}
```

```
$ ./creux -a p1
$ ./creux -s p2
$ diff p1 p2
$ du p1 p2
```

```
8       p1
36      p2
```

7.3.2 Verrous

Pour assurer la cohérence d'un fichier ouvert par plusieurs processus, on utilise des verrous qui permettent de verouiller tout ou partie d'un fichier. Il y a deux types de verrous :

- verrou de lecture (verrou partagé) : interdit toute écriture
- verrou d'écriture (verrou exclusif) : interdit tout accès

On pose les verrous en utilisant `fnctl` et une structure de type `struct flock`. Cette fonction vérifie que la pose d'un verrou n'est pas incompatible avec d'éventuels verrous déjà posés.

```
struct flock {
  ...
  short l_type;    /* Type of lock: F_RDLCK,   F_WRLCK, F_UNLCK */
  short l_whence;  /* How to interpret l_start:   SEEK_SET, SEEK_CUR, SEEK_END */
  off_t l_start;   /* Starting offset for lock */
  off_t l_len;     /* Number of bytes to lock */
  pid_t l_pid;     /* PID of process blocking our lock (F_GETLK only) */
  ...
};
```

- `l_type` : verrou partagé (`F_RDLCK`), exclusif (`F_WRLCK`) ou dévérouillage (`F_UNLCK`).
- le couple `l_whence`, `l_start` permet de fixer le début de verrou
- `l_len` : si nul, le verrou s'applique jusqu'à la fin

Pour placer un verrou, on utilise :

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, struct flock *lock);
```

- `cmd` : `F_SETLK` (placement), `F_GETLK` (consultation), et `F_SETLKW` (demande bloquante)
- renvoie -1 en cas d'erreur (voir errno). ex : conflit avec autre verrou, ...

entreessorties/exemples/verrou.c

```
/*********************************************************************
 *verrou.c
 *
 *  (François lemaire)  <lemaire@lifl.fr>
 * Time-stamp: <2007-03-26 17:07:19 lemaire>
 *********************************************************************/

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

int main(int argc, char *argv[]) {
  int fd;
  struct flock fl;

  fd = open("tutu", O_WRONLY, 0666);

  fl.l_type = F_WRLCK;
  fl.l_whence = SEEK_SET;
  fl.l_start = 0;
  fl.l_len = 0;

  printf("Je demande le verrou (%d)\n", getpid());
  if (fcntl(fd, F_SETLKW, &fl)==-1) {
    perror("flock :");
    exit(EXIT_FAILURE);
  }
  printf("J'ai le verrou (%d) et j'écris : %s\n",  getpid(), argv[1]);
  write(fd, argv[1], strlen(argv[1]));
  sleep(5);

  printf("J'ai libéré le verrou (%d)\n", getpid());
  fl.l_type = F_UNLCK;
  fcntl(fd, F_SETLK, &fl);
  close(fd);
  exit(EXIT_SUCCESS);
}

/*
$ ./verrou "bonjour" & ./verrou "au revoir"
[5] 9749
Je demande le verrou (9749)
Je demande le verrou (9750)
J'ai le verrou (9749) et j'écris : bonjour
.... attente de 5 secondes ...
J'ai libéré le verrou (9749)
J'ai le verrou (9750) et j'écris : au revoir
```

```
J'ai libéré le verrou (9750)
[5]  - Done
$
*/
```

Chapitre 8

Les processus

8.1 Présentation

Un processus est un objet dynamique correspondant à l'exécution d'un programme (ou exécutable) qui est une suite d'instructions (inerte). Un processus a besoin de ressources (processeur, mémoire, périphériques, ...,) pour s'exécuter. C'est le système d'exploitation qui gère l'ensemble des processus. Les SE récents sont multi-processus ; plusieurs processus peuvent s'exécuter simultanément (ordinateur multi-processeur) et/ou séquentiellement (chaque processus s'exécute puis est endormi).

8.1.1 Cycle de vie

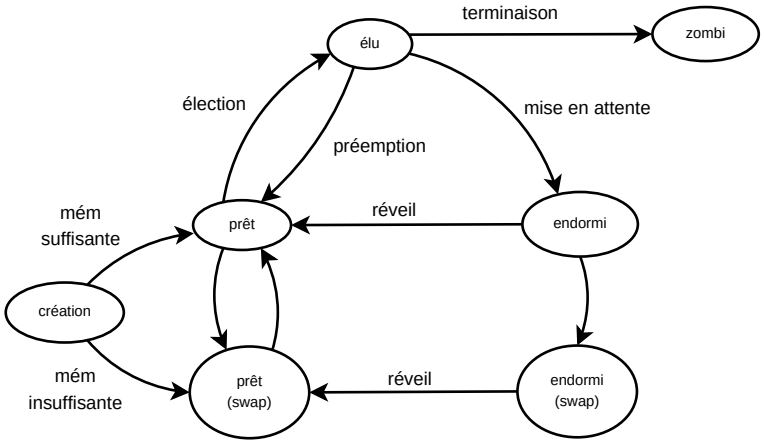


FIG. 8.1 – Cycle de vie d'un processus

élu en train d'exécuter des instructions (mode utilisateur ou noyau)

prêt en attente de cpu

endormi ne fait rien, en attente d'un événement/signal de réveil

zombi attend que son père soit notifié de la terminaison

8.1.2 Attributs

Un processus possède de nombreux attributs dont voici les principaux :
- `pid` (processus identifier) : nombre unique. `pid_t getpid(void)`
- `ppid` (parent pid) : pid du père. `pid_t getppid(void)`
- `uid`, `gid` : utilisateur/groupe propriétaire. `uid_t getuid(void)` et `uid_t geteuid(void)`
- `euid`, `egid` : utilisateur/groupe effectifs. `uid_t getuid(void)` et `uid_t geteuid(void)`. Peuvent différer de utilisateur/groupe propriétaire (cas d'un exécutable avec droits setuid/setgid). Les utilisateur/groupe effectifs déterminent les droits du processus
- `cwd` (current working directory). Le répertoire de travail du processus.
  S'obtient avec `char *getcwd(char *buf, size_t size)`. Se change avec `int chdir(const char *path)`.
- date de création du processus
- temps CPU consommé (mode noyau/utilisateur, par le processus/ses fils)
- masque de création de fichiers (voir `umask`)
- table des descripteurs de fichiers ouverts
- verrous sur les fichiers

– ...

8.2 Gestion des processus

Le lancement d'un processus s'effectue en deux étapes :
– le clonage : duplication du processus père qui crée un fils identique au père
– mutation : le processus fils change son code exécutable

Il y a donc une arboresence de processus (`pstree`). Au démarrage de l'ordinateur, un processus spécial est créé (`init`) de pid 1. C'est ce processus qui est à la racine de l'arborescence des processus.

8.2.1 Clonage

On utilise la fonction `pid_t fork(void)`.
– duplique à l'identique le processus appelant, un fils est créé
– le père et le fils continuent chacun leur exécution après le `fork`
– renvoie :
  – 0 si on est dans le processus fils
  – le pid du fils si on est dans le processus père
  – -1 si erreur (voir errno)

processus/exemples/fork.c

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

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

/*
  On crée un fils. Le père et le fils affichent chacun un message
  ainsi que les deux numéros de processus.
*/

int main() {
  pid_t pid;

  pid = fork();
  if (pid==-1) {
    perror("fork");
    exit(-1);
  }

  if (pid==0) {
     printf("Je suis le fils de pid %d de mon père de pid %d\n",
            getpid(), getppid());
  } else {
    printf("Je suis le père de pid %d de mon fils de pid %d\n",
           getpid(), pid);
    wait(NULL);
    printf("Mon fils est mort\n");
  }
  exit(EXIT_SUCCESS);
}
```

Le fils n'hérite pas de :
– pid
– temps cpu
– verrous
– signaux pendants
– priorité (remise à valeur par défaut)

Le fils obtient une copie de la mémoire du père (données statiques modifiables, tas, pile). Le SE peut travailler de façon paresseuse et ne recopier les données que lorsque le fils ou le père les modifie (économie de mémoire allouée) : stratégie copy-on-write.

Remarque : la copie de mémoire recopie donc les références à d'éventuelles structures systèmes (comme par exemple un descripteur de fichier ouvert). Ainsi, des comportements inattendus peuvent se produire :
– le père ouvre un fichier en lecture
– le père crée un fils
– le père lit une ligne dans le fichier (il récupère la première)
– le fils lit une ligne : il récupère la deuxième

Nécessité de savoir où les ressources sont stockées : dans le processus ou dans le système.

8.2.2 Terminaison

Sauf exception, le processus père est censé prendre connaissance de la terminaison de ses fils. On utilise pour cela les commandes `wait` et `waitpid`. Lorsqu'un père se termine sans avoir attendu la terminaison d'un de ses fils, ce dernier est adopté par le processus `init` (de pid 1).

`pid_t wait(int *pstatus)`
- renvoie -1 si erreur (voir errno). ex : le processus n'a aucun fils
- si le processus a des fils à l'état zombi, l'un des zombis est choisi par le système et retiré de la table des processus, et `*pstatus` est modifié si `pstatus` n'est pas `NULL`.
- sinon, le processus est bloqué jusqu'à la fin d'un fils

`pid_t waitpid(pid_t pid, int *pstatus, int options)`
- `pid` : fils à attendre. 0 pour un fils quelconque
- `options` : combinaison binaire de
  - `WNOHANG` : ne se bloque pas si aucun fils ne s'est terminé
  - `WUNTRACED`, `WCONTINUED` : voir `man ptrace`
- renvoie
  - -1 si erreur (voir errno)
  - 0 si aucun fils ne s'est terminé et si `options` contient `WNOHANG`
  - le pid du fils terminé

Il faut faire autant d'appels à `wait` (ou `waitpid`) que de fils créés.

processus/exemples/waitnonbloquant.c

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

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


int main() {
  pid_t pid;

  pid = fork();
  if (pid==-1) {
    fprintf(stderr, "Le création du fils a échoué\n");
    exit(-1);
  }

  if (pid==0) {
    printf("Je suis le fils de mon père de pid %d\n",getppid());
    printf("Je m'endors 5 secondes\n");
    sleep(5);

  } else {
    printf("Je suis le père de mon fils de pid %d\n", pid);
    while (!waitpid(-1, NULL, WNOHANG) ) {
      printf("Mon fils dort\n");
      sleep(1);
    }

    printf("Mon fils est mort\n");
    sleep(5);
  }

  return 0;
}
/*
Je suis le fils de mon père de pid 31938
Je m'endors 5 secondes
Je suis le père de mon fils de pid 31939
Mon fils dort
Mon fils dort
Mon fils dort
Mon fils dort
Mon fils dort
Mon fils est mort

*/
```

La valeur stockée dans `*pstatus` permet d'obtenir des informations sur la façon dont s'est terminé le fils. Ainsi, si on a fait un appel de la forme `wait(&status)` ou `waitpid(...,&status,...)`, l'entier `status` est modifié. Il permet de connaître la cause de la terminaison :
- `WIFEXITED(status)` : non nul si le processus a fait un `exit` (ou un return dans le main). Le code de retour est dans `WEXITSTATUS(status)`
- `WIFSIGNALED(status)` : non nul si le processus s'est terminé à cause d'un signal. Le signal qui a terminé est dans `WTERMSIG(status)`.
- ...

8.2.3 Mutation

Un processus peut changer le code qu'il est en train d'exécuter. On parle de mutation ou de recouvrement de processus. Le processus converse :
- ses numéros de processus (pid, ppid,...)
- la table des fichiers ouverts (exceptions)
- les temps CPU consommé

– ...

On utilise les appels de la famille `exec*` qui se basent tous sur `execve`.

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

– remplace le code du processus courant par celui de `filename`
– `argv` et `envp` sont des tableaux de chaînes terminés par `NULL`
– `argv` est la liste des arguments passés au programme
– `envp` est une liste de la forme `cle=valeur` qui permet de modifier des variables d'environement
– ne rend pas la main en cas de succès
– renvoie -1 si erreur (voir errno)

Attention, `argv[0]` est le nom que pense avoir le processus après mutation.

processus/exemples/execve.c

```
/*****************************************************************
 *execve.c
 *
 *  (François lemaire) <lemaire@lifl.fr>
 * Time-stamp: <2007-03-28 18:05:36 lemaire>
 *****************************************************************/

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

int main() {

  char *argv[] = { "ls", "-a", NULL };
  char *envp[] = { NULL };

  execve("/bin/ls", argv, envp);
  perror("execve");
  exit(EXIT_FAILURE);
}
```

Il y a des variantes de `execve` :

```
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[]);
```

Les arguments remplissant les ... sont un nombre variable de chaînes, la dernière devant être `NULL`.

Exemple ▷ Le `v` signifie qu'on passe un tableau, le `l` une suite d'arguments. Le `p` signifie que la commande doit être recherchée dans `$PATH`.

```
char *argv=[] = { "ls", "-l", "toto", "tutu", NULL };

execv("/bin/ls", argv);
execvp("ls", argv);
execl("/bin/ls", "ls", "-l", "toto", "tutu", NULL);
execlp("ls", "ls", "-l", "toto", "tutu", NULL);
```

◁

8.2.4 Exemple du Shell

Exécution d'une commande simple

Pour exécuter une commande simple `cmd`, le shell peut utiliser la technique suivante :
– faire un fork
– muter le fils en la commande `cmd`
– attendre le fils

Exécution d'une commande en arrière-plan

Pour exécuter une commande en arrière-plan `cmd &`, le shell peut utiliser la technique suivante :
– faire un fork
– muter le fils en la commande `cmd`
– ne pas attendre le fils

Remarque : en utilisant les commandes `bg` et `fg`, on peut commuter l'attente du shell.

L'intérêt de la distinction `fork`, `exec` est de pouvoir changer l'environnement du fils avant la mutation, comme par exemple rediriger les entrées/sorties.

8.3 Redirection des E/S

À la création d'un processus avec `fork`, trois descripteurs sont ouverts automatiquement :
- `STDIN_FILENO` (vaut 0) : entrée standard
- `STDOUT_FILENO` (vaut 1) : sortie standard
- `STDERR_FILENO` (vaut 2) : sortie erreur

Les fonctions de saisie de la librairie C lisent sur le descripteur 0, Les fonctions d'affichage de la librairie C (`printf`, ...) écrivent sur le descripteur 1, `fprintf(stderr, ...` écrit sur le descripteur 2.

Sans détournement, le descripteur 0 est le clavier, 1 et 2 sont l'écrans. On peut dupliquer un descripteur dans un autre en utilisant la commande `dup2` :

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

- force le descripteur newfd à devenir une copie de oldfd
- newfd est fermé si il était ouvert
- oldfd et intfd sont interchangeables (synonymes) :

L'intérêt de `dup2` est de changer les trois descripteurs (0,1,2) avant de faire le recouvrement avec `exec`. En effet, le processus après recouvrement conserve ses anciens descripteurs.

processus/exemples/redirection.c

```
/*************************************************************************
 *redirection.c
 *
 *  (François lemaire)  <lemaire@lifl.fr>
 * Time-stamp: <2007-04-02 12:39:17 lemaire>
 *************************************************************************/

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

int main() {
  int fd;

  fd = open("output", O_WRONLY|O_TRUNC|O_CREAT, 0644);
  if (fd==-1) { perror("open"); exit(EXIT_FAILURE); }
  dup2(fd, STDOUT_FILENO);
  close(fd);
  execl("/bin/ls", "ls", NULL);
  exit(EXIT_SUCCESS);
}

/*
$ ./redirection
$ head -n 5 output
a.out
ex0.c
ex1bis.c
ex1.c
ex2.c
*/
```

Chapitre 9

Les tubes

9.1 Présentation

Un tube est un moyen de communication unidirectionnel entre processus. C'est une FIFO de caractères (ou d'octets) : les données qui y transitent sont donc non formatées.

Un tube est manipulé par l'intermédiaire de deux descripteurs de fichiers (un pour l'entrée, un pour la sortie). On écrit dans l'entrée et on lit dans la sortie.

Un tube fait partie du système de fichiers : il est donc extérieur au processus. Ainsi, un fils aura accès au tube créé par le père.

Un tube possède un nombre quelconque de processus lecteurs et de processus écrivains, et une taille maximum (il peut donc être plein).

9.2 Manipulation

9.2.1 Création

On crée un tube avec :

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- range dans `filedes[0]` le descripteur pour la lecture et dans `filedes[1]` celui de l'écriture.
- renvoie 0 si ok, et -1 si erreur (voir errno)

Le tube créé n'est pas nommé et ne peut pas être ouvert avec `open`. Ainsi, un processus peut acquérir un descripteur sur un tube soit en le créant, soit par héritage.

Seul le processus créateur du tube et ses fils peuvent y accéder. Si un processus perd l'accès à un tube (après un close), il ne peut plus y accéder ensuite.

9.2.2 Lecture

On lit dans un tube avec la fonction `read`.

- si le tube n'est pas vide, `read` récupère les données et se termine. `read` lit `min(dispo, demandees)` où `dispo`/`demandees` sont les nombres d'octets disponibles/demandées. (read peut donc renvoyer moins d'octets que demandé)
- si le tube est vide et qu'il y a des écrivains, l'appel a `read` est bloquant (jusqu'à ce que des données soit écrites)
- si le tube est vide et qu'il n'y a plus d'écrivains, `read` renvoie 0.

Ainsi, la seule façon pour le lecteur de savoir qu'il n'y aura plus de données est d'attendre que `read` renvoie 0, ce qui implique que chaque processus doit fermer ses descripteurs non utilisés (le plus vite possible) pour que le nombre d'écrivains passe à zéro.

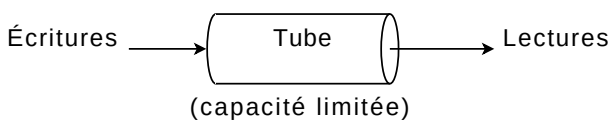


Fig. 9.1 – Tube (Fifo)

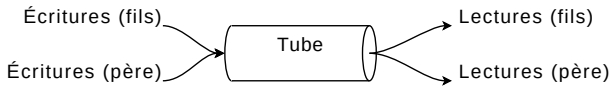


FIG. 9.2 – Tube après un fork

9.2.3 Écriture

Se fait avec `write` :
- il faut écrire moins de `PIPE_BUF` octets (de l'ordre de 4096) pour être certain de faire des écritures atomiques (écriture en une fois sans entrelacement avec d'autres écritures)
- si le nombre de lecteurs est nul, le processus écrivain reçoit le signal `SIGPIPE` (qui par défaut provoque l'arrêt du processus)
- si le nombre de lecteurs n'est pas nul et si écriture atomique, `write` attend jusqu'à pouvoir écrire d'un coup les données demandées. L'écriture est donc bloquante.

9.2.4 Exemple

tubes/exemples/tube1.c

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

int main() {
  int fd[2];
  int nb_lus;
  pid_t pid;
  char buf[100];
  char chaine[]="Salut fiston!";

  pipe(fd);
  pid = fork();

  if (pid!=0) {
    close(fd[0]);
    write(fd[1], chaine, strlen(chaine));
    // Remarque on n'a pas envoyé le 0 qui termine la chaine
    close(fd[1]);
    wait(NULL);
  } else {
    close(fd[1]);
    nb_lus = read(fd[0], buf, 99);
    // read va récupérer les données écrites sur fd[1]
    printf("Nous avons récupéré %d octets\n", nb_lus);
    buf[nb_lus]='\0';
    // On ajoute nous même le 0 à la fin de la chaîne
    printf("Chaîne reçue : %s\n", buf);
    close(fd[0]);
  }
  return 0;
}

/*
$ ./tube1
Nous avons récupéré 13 octets
Chaîne reçue : Salut fiston!
$
*/
```

Question : que se passe-t-il si le père ne fait pas `close(fd[1])` ?

tubes/exemples/tube2.c

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

int main() {
  int fd[2];
  int nb_lus;
  pid_t pid;
  char c;
  char chaine[]="Salut fiston!";

  pipe(fd);
  pid = fork();
  if (pid!=0) {
    close(fd[0]);
    write(fd[1], chaine, strlen(chaine));
    close(fd[1]);
```

```
    wait(NULL);
  } else {
    close(fd[1]);

    while ( (nb_lus=read(fd[0], &c, 1)) != 0) {
      printf("%c", c);
    }
    printf("\n");
    close(fd[0]);
  }
  return 0;
}

/*
$ ./tube2
Salut fiston!
$
*/
```

Même question.

Chapitre 10

Les signaux

Un signal est un événement asynchrone destiné à un processus et émis par un autre processus ou le système.

À la réception d'un signal, le processus s'interrompt, exécute une fonction associée au signal reçu (le handler, ou traitant de signal), et reprend là où il s'était arrêté.

Un signal est caractérisé uniquement par son numéro. Il est anonyme (on ne peut savoir qui l'a envoyé). Un signal permet de signaler un événement particulier qui nécessite souvent un traitement approprié (un processus qui a exécuté une division par 0, qui lit en dehors de sa zone mémoire, ...).

Un traitant différent est associé à chaque signal. Un traitant par défaut existe pour chaque signal.

10.1 Utilisation simple des signaux

10.1.1 Principaux signaux

Il y a de nombreux signaux, dont voici quelques uns regroupés par classe (voir `man 7 signal` pour une liste détaillée).

nom	événement	traitant par défaut
Terminaison		
SIGINT	interruption <INTR>, C-c	terminaison
SIGQUIT	interruption <QUIT>, C-\	terminaison + core
SIGHUP	fin de connection/du leader de session	terminaison
SIGKILL	terminaison immédiate	terminaison (immuable)
SIGTERM	terminaison	terminaison
Fautes		
SIGFPE	erreur arithmétique	terminaison + core
SIGILL	instruction illégale	terminaison + core
SIGSEGV	violation protection mémoire	terminaison + core
Divers		
SIGALARM	signal horloge	ignoré
SIGUSR1	signal utilisateur	terminaison
SIGUSR2	idem.	terminaison
Suspension/reprise		
SIGTSTP	suspension <susp>, C-z	suspension
SIGSTOP	suspension	suspension (immuable)
SIGCHLD	terminaison ou arrêt d'un fils	ignoré
SIGCONT	reprise d'un fils arrêté	reprise si arrêté

10.1.2 Envoi

On utilise l'appel système :

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

int kill(pid_t pid, int sig);
```

- `pid` : numéro du processus destinataire du signal (0 : tous les processus de même gid, -1 : tous les processus auxquels on peut envoyer un signal)
- `sig` : numéro du signal à envoyer
- renvoie -1 si erreur (voir errno)

Un processus ne peut envoyer un signal qu'à des processus de même `uid`.

On peut utiliser aussi la commande `kill` en shell :

```
kill -signal pid ...
```

10.1.3 Attente

On peut mettre un processus en attente jusqu'à réception d'un signal avec :

```
int pause(void);
```

Le processus s'endort jusqu'à ce qu'un signal soit reçu. Le signal est alors traité : soit le processus est arrêté, soit la fonction `pause` se termine lorsque le traitant est effectué.

10.1.4 États d'un signal

- envoyé : vient d'être envoyé (par un processus ou le système)
- reçu : reçu par le destinataire
- pendant : reçu mais pas encore traité
- delivré : pris en compte par le destinataire (traitant exécuté)

L'état pendant permet de conserver le signal jusqu'à son traitement. En effet, un signal peut être bloqué par le destinataire (par exemple si le destinataire est déjà en train d'exécuter le traitant d'un signal, si le destinataire a volontairement masqué le signal).

10.1.5 Traitement d'un signal

Pour chaque type de signal, pour chaque processus, un signal est codé par une structure :
- booléen : signal pendant
- booléen : signal bloqué
- pointeur de fonction : traitant de signal
- masque des signaux à bloquer lors de l'exécution du traitant

Traitement :
- à la réception d'un signal : pendant=vrai
- (de manière asynchrone) si signal bloqué ou signal non pendant : terminé
- sinon :
  - sauvegarde de l'ancien masque des signaux
  - exécution du traitant
  - restauration de l'ancien masque des signaux

Conséquence : si il y a une rafale de signaux identiques, certains seront perdus.

10.2 Gestion personnalisée des signaux

On peut personnaliser le comportement de chaque signal. Un signal peut être bloqué. On peut aussi modifier le traitant en :
- ignorant le signal (traitant ne faisant rien) ;
- replaçant le traitant par défaut ;
- utilisant un traitant personnalisé.

Pour bloquer et changer les traitant, on doit manipuler des ensembles de signaux.

10.2.1 Ensembles de signaux

Un ensemble de signaux se manipule par le type `sigset_t`. On utilise les primitives :
- initialisation à vide : `int sigemptyset(sigset_t *set)`
- initialisation à plein : `int sigfillset(sigset_t *set)`
- ajout/suppression :
  - `int sigaddset(sigset_t *set, int signum)`
  - `int sigdelset(sigset_t *set, int signum)`
- appartenace : `int sigismember(const sigset_t *set, int signum)`. renvoie 1 si `signum` est dans l'ensemble, 0 sinon.

Toutes ces fonctions renvoient -1 en cas d'erreur, et 0 si tout s'est bien passé (excepté `sigismember`).

10.2.2 Blocage de signaux

Pour spécifier les signaux bloqués, on utilise `sigprocmask`)

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- si `oldset` n'est pas nul, il reçoit l'ensemble courant des signaux bloqués

how	nouvel ensemble des signaux bloqués
SIG_SETMASK	*set
SIG_BLOCK	ensemble courant union *set
SIG_UNBLOCK	ensemble courant privé de *set

10.2.3 Modification du traitant

On utilise la structure `struct sigaction`

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

- `sa_handler` est soit `SIG_DFL`, `SIG_IGN` ou un pointeur de fonction.
- `sa_mask` est l'ensemble des signaux à masquer lors du traitant
- `sa_flags` : OU binaire d'options ...

On utilise la fonction `sigaction`

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- `signum` : numéro du signal à modifier
- `act` : nouveau traitant
- si `oldact` n'est pas NULL, `*oldact` reçoit l'ancien traitant
- renvoie 0 si ok, -1 si erreur

Remarque : on ne peut pas modifier les traitants de `SIGKILL` et `SIGSTOP`.

Exemple ▷

signaux/exemples/term.c

```
/*****************************************************************
 *term.c
 *
 *  (François lemaire) <lemaire@lifl.fr>
 * Time-stamp: <2007-05-02 16:45:36 lemaire>
 *****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>

void meurt(int sig) {
  printf("Argh !!\n");
  exit(0);
}

int main(int argc, char *argv[]) {

  struct sigaction sa;

  sa.sa_handler = meurt;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);

  sigaction(SIGTERM, &sa, NULL);

  printf("J'ai le pid %d\n", getpid());
  pause();
  exit(EXIT_SUCCESS);
}
```

```
$./term
J'ai le pid 10479                $ kill -TERM 10479
Argh!!                           $
$
```

◁

10.2.4 Ancienne interface signal

L'ancienne interface ANSI fournit la fonction `signal` pour modifier le traitant d'un signal. Inconvénients :
- pas de masque de signaux à bloquer lors de l'exécution du traitant
- le traitant par défaut est parfois replacé

Moralité : ne pas utiliser `signal`.

10.2.5 Alarme

On peut déclencher une minuterie qui envoie le signal `SIGALRM` au bout d'un temps déterminé en utilisant la fonction `alarm`.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

- demande l'envoi du signal `SIGALRM` au bout de `seconds` secondes
- toute nouvelle alarme annule l'alarme courante
- `seconds`=0 annule l'alarme courante sans en déclencher de nouvelle

signaux/exemples/mdp.c

```
/*****************************************************************
 *mdp.c
 *
 *  (François lemaire) <lemaire@lifl.fr>
 * Time-stamp: <2007-05-07 11:27:17 lemaire>
 *****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>

int alarmInter=0;

void arret(int sig) {
  printf("\nTemps écoulé!!!\n");
  exit(EXIT_FAILURE);
}

void avertissement(int sig) {
  struct sigaction sa;

  printf("\nIl reste 10 secondes\n");
  printf("Mot de Passe : ");

  sa.sa_handler = arret;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sigaction(SIGALRM, &sa, NULL);
  alarmInter = 1;
  alarm(10);
}

int main(int argc, char *argv[]) {
  struct sigaction sa;
  char mdp[11];
  char *out;

  /* détournement */
  sa.sa_handler = avertissement;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sigaction(SIGALRM, &sa, NULL);

  /* placement de l'alarme */
  alarm(10);

  printf("Mot de passe : ");
  fgets(mdp, 10, stdin);

  /* fgets est stoppé par au déclenchement de l'alarme.
     On relance la saisie si alarmInter vaut vrai */

  if (alarmInter) {
    fgets(mdp, 10, stdin);
  }
  printf("Mot de passe : %s\n", mdp);

  exit(EXIT_SUCCESS);
}
```

Chapitre 11

Synchronisation d'activités concurrentes

Un SE exécute différentes activités (processus, processus légers, ...) en parallèle, en les exécutant tour à tour : les activités sont chacune exécutées un certain laps de temps dans un ordre (a priori) imprévisible.

Question :

– le résultat dépend-il de l'ordonnancement (ordre dans lequel les activités sont exécutées) ?

Réponse : en général OUI !

11.1 Nécessité de la synchronisation

11.1.1 Un exemple classique

Hyp : deux activités A1 et A2 ont accès à une même variable N : N est donc une ressource partagée. A1 et A2 vont chacune à un moment donné exécuter N=N+1.

Problème : l'instruction N=N+1 se compile en langage machine en une séquence du style :

```
charger N dans le registre R1
incrémenter registre R1
charger le registre R1 dans N
```

Comme l'instruction n'est pas indivisible, les exécutions de A1 et A2 peuvent être entrelacées en

```
          A1                          A2
charger N dans R1
                              charger N dans R1
incrémenter R1
                              incrémenter R1
charger R1 dans N
                              charger R1 dans N
```

`N` ne sera donc incrémenté qu'une fois !

Pour garantir le résultat, il faut s'assurer que A1 et A2 n'exécutent pas en même temps la section de 3 instructions. Cette section est appelée section critique.

11.1.2 Section critique

Une section critique est une section de code qui doit être exécutée de manière indivisible. On dit aussi exécutée de manière atomique, ou en exclusion mutuelle.

Il faut donc modifier les trois instructions en

```
entrer dans la  section critique
charger N dans R1
incrémenter registre R1
charger R1 dans N
sortir de la section critique
```

Si A1 atteint la section critique avant A2, A2 ne pourra entrer dans la section que lorsque que A1 l'aura quittée.

Comment coder l'entrée et la sortie de la section critique ?

```
initialement dans_critique vaut faux
entrer dans la section critique :
  tant que dans_critique=vrai ne rien faire
  dans_critique=vrai

sortir de la section critique:
  dans_critique=faux
```

Cette solutions a deux inconvénients :

- l'attente est active : 100% du CPU consommé pour rien
- si A1 et A2 testent successivement en même temps si `dans_critique` vaut vrai, A1 et A2 vont simultanément rentrer dans la section critique.

Il faut donc des primitives systèmes ...

11.2 Techniques de synchronisation

- verrous
- sémaphores
- conditions/moniteurs (non vus)

11.2.1 Verrous

Un verrou est une variable à deux états : verrouillé/déverrouillé. On fait changer l'état du verrou par deux opérations :

- `mutex_lock(V)`
  - verrouille le verrou V
  - si V est déjà verrouillé, bloquant jusqu'à ce que V soit déverrouillé
- `mutex_unlock(V)`
  - déverrouille le verrou V
  - seule l'activité qui a verrouillé le verrou peut le déverrouiller

Une section critique peut donc être protégée par un bloc `mutex_lock(V)` ...`mutex_unlock(V)`

```
mutex_lock(V)
N=N+1
mutex_unlock(V)
```

La première des activités qui parvient à verrouiller V empêche les autres activités d'exécuter la section critique.

11.2.2 Sémaphores

Un sémaphore est composé :

- un nombre initial de jetons (positif ou nul)
- d'une variable entière (désignant le nombre de jetons libres)
- d'une file d'attente d'activités

On peut incrémenter et décrémenter le nombre de jetons de manière atomique. On note traditionnellement P et S ces deux opérations indivisibles (initiales de Proberen=tester et de Verhogen=incrémenter).

```
P(S) Puis-je ?
  S = S - 1
  si S < 0 alors
    l'activité exécutant P(S) est bloquée dans une file
    d'attente spécifique au sémaphore S

V(S) Vas-y !
  S = S + 1
  si S<=0 alors
    une des activités de la file d'attente est débloquée
```

Si le nombre de jetons est

- positif : c'est le nombre d'activités pouvant acquérir le jeton sans se bloquer
- négatif : c'est le nombre d'activités bloquées en attente d'un jeton

Un verrou peut être vu comme un cas particulier de sémaphore initialisé à 1 ! Les sémaphores permettent donc de protéger des sections critiques, mais permettent aussi de séquencer des activités.

```
initialiser S à 1

P(S)
N=N+1
V(S)
```

La première activité qui exécute `P(S)` provoque le blocage de l'autre sur `P(S)` jusqu'à ce que le jeton soit rendu par `V(S)`.

11.3 Exemples classiques de synchronisation

11.3.1 Attente d'activité

Problème : on veut coder le comportement suivant

```
activité 1:
<action 1>
attendre que A2 ait terminé l'action 2
...

activité 2:
<action 2>
....
```

Réponse :

```
initialiser un sémaphore S à 0

activité 1:
<action 1>
P(S)
...

activité 2:
<action 2>
V(S)
....
```

Explications :
- si A1 atteint P(S) avant que A2 n'ait atteint V(S) : S prend la valeur -1 et A1 se bloque jusqu'à ce que A2 exécute V(S)
- sinon, V(S) fait passer le sémaphore à 1, et A1 exécute P(S) sans se bloquer

11.3.2 Attente mutuelle de deux activités

Deux activités se donnent un point de rendez-vous.

```
activité 1:
<action 1>
attendre que A2 ait exécuté action 2
...


activité 2:
<action 2>
attendre que A1 ait exécuté action 1
....
```

Réponse :

```
S1=S2=0

activité 1:
<action 1>
V(S2)
P(S1)
...

activité 2:
<action 2>
V(S1)
P(S2)
....
```

11.4 Les interblocages

On peut bloquer de manière irrécupérable des activités en utilisant incorrectement les verrous/sémaphores : on parle d'étreinte fatale (ou deadlock). Cela se produit par exemple lorsque l'activité A1 attend que A2 libère une ressource, et que A2 attend que A1 en libère une autre.

```
S1 initialisé à 0
S2 initialisé à 0

    A1                  A2

    P(S1)               P(S2)
    V(S2)               V(S1)
```

Autre exemple :

```
S1 initialisé à 1
S2 initialisé à 1

    A1                  A2

    P(S1)               P(S2)
    P(S2)               P(S1)
```

Conseils pour éviter les deadlocks :

- acquérir les verrous dans le même sens
- ne pas oublier le libérer les verrous
- n'utiliser des verrous qu'entre des activités de même priorité (pour s'assurer qu'une des activités n'empêche pas une de s'exécuter)

Chapitre 12

Les processus légers ou threads

Un thread est une séquence (ou fil) d'exécution du code d'un programme au sein d'un processus. Un processus classique (lourd) ne contient qu'un seul thread. Un processus multithreadé peut contenir plusieurs flots d'exécutions simultanés.

Idée : partager les différentes ressources d'un processus par plusieurs flots d'exécution.

Référence : `http://www.llnl.gov/computing/tutorials/pthreads/`

12.1 Présentation

12.1.1 Principe

Un thread n'existe qu'au sein d'un processus lourd. Ses ressources sont partagées par les différents threads qu'il contient :

- code
- mémoire
- fichiers
- droits Unix
- environnement shell, répertoire de travail
- ...

Chaque thread possède :

- sa propre pile d'exécution
- un identificateur de thread
- un pointeur d'instruction

Cela est évidemment nécessaire pour que chaque thread ait un flot d'exécution distinct.

Un processus (Unix) peut contenir plusieurs centaines de threads.

12.1.2 Cycle de vie d'un thread

Au lancement d'un processus, un seul thread est créé (celui du processus initial). La création de chaque nouveau thread est paramétrée par une fonction à exécuter, avec éventuellement des paramètres. Le thread se termine soit explicitement, soit en étant tué par un autre thread ou par le système.

Les différents threads d'un processus se partagent le temps alloué au processus par le SE. On peut ajuster les priorité des threads au besoin.

12.1.3 Critiques des threads

Avantages

Le partage des ressources entre les threads est automatique, il évite donc des mécanismes sophistiqués de communication entre processus (mémoire partagée, tubes, ...)

Le temps de commutation entre deux threads d'un même processus est peu coûteux, car il consiste simplement à changer de flot d'exécution.

La vitesse d'exécution peut être améliorée. Supposons que l'on exécute des requêtes répondant avec un certain délai. Avec un seul thread, une attente est nécessaire à chaque requête au système. Si plusieurs threads font chacun un requête, attentes se font en parallèle.

Multiprocesseurs . Sur un SE adapté, les threads peuvent être exécutés en parallèle ... sous condition

La réactivité d'interfaces utilisateurs peut être améliorée. Un thread pour réafficher l'interface graphique (raffraichissement des images, affichage d'un curseur d'avancement), un autre thread ayant lancé un calcul lourd. L'application reste réactive.

La réactivité d'un serveur peut être améliorée. Avec un seul thread, un serveur doit gérer plusieurs requêtes : répondre aux requêtes, gérer les priorités de requêtes. Sur un serveur multithreadé, on crée un thread par requête. Le réglage de priorité (ordonnancement des threads) peut être laissé au système.

L'écriture de programmes peut être facilitée. Prenons l'exemple simplifié d'un tableur.

Pseudo code avec un thread :

```
while (1) {
  while (maj_necessaire() && ! entree_utilisateur_disponible()) {
    recalculer_une_formule();
  }
  cmd = get_entree_utilisateur();
  traiter_entree_utilisateur(cmd);
}
```

Avec un code à deux threads :

```
// thread clavier
while (1) {
  cmd = get_entree_utilisateur();
  traiter_entree_utilisateur(cmd);
}

//thread calcul
while (1) {
  attendre_un_changement();
  recalculer_toutes_les_formules();
}
```

L'expression est plus naturelle. La raison principale est ici qu'on peut bloquer chaque thread sur un événement, et le traiter quand il se produit. Chaque thread a également une tâche bien précise, et le code est mieux découpé.

Inconvénients

La concurrence entre les threads doit être correctement gérée. Risque de bogues occasionels, d'étreinte fatales (dead-lock), ... Il faut analyser finement la concurrence des threads.

La pile d'exécution peut être difficile à régler. Comme chaque thread en possède une, le réglage de sa taille dépend de l'activité du thread (appels récursifs profonds, variables locales nombreuses, ...).

Réentrance : propriété pour une fonction d'être lancée simultanément par plusieurs tâches. Si plusieurs threads appellent une même fonction non prévue pour cela, le résultat de la fonction risque d'être erroné ! Le problème se produit avec les fonctions ayant des effets de bords, utilisant des variables statiques.

Une application multi-threadée ne peut donc pas toujours appeler une librairie non prévue pour cela, sans prendre de précaution (protéger les appels par des verrous)

12.1.4 Quand choisir les threads

Il est intéressant d'utiliser les threads si un processus
- se bloque pour des délais potentiellement longs (temps perdu)
- résout des tâches distinctes en parallèle
- doit répondre à des événements asynchrones (saisie clavier, répétition d'un tâche à intervalle régulier, ...)
-

12.2 La librairie pthread

La librairie `pthread` satisfait la norme POSIX 1.c.

12.2.1 Création d'un thread

```
#include <pthread.h>

int  pthread_create(pthread_t  *  thread, pthread_attr_t * attr, void *
                    (*start_routine)(void *), void * arg);
```

- crée un thread qui exécute immédiatement la fonction `start_routine` en lui passant la valeur de `arg`

– `attr` permet de modifier le comportement du thread. Si `attr` vaut `NULL`, des paramètres par défaut sont utilisés (le thread est joignable, priorité non temps-réel, ...)
– renvoie un entier non nul si la création a échoué (nombre maximum de threads `PTHREAD_THREADS_MAX` déjà atteint, pas assez de ressources ...).
– renvoie 0 si ok. `*thread` est alors modifié (sert à identifier le thread).

La variable `arg` doit pointer sur une zone sûre (qui restera accessible, et dont la valeur ne risque pas de changer). Solution simple : passer une zone allouée par `malloc` au thread, et la `libérer` dans `start_routine`.

Un thread peut connaître son propre identifiant avec :

```
#include <pthread.h>

pthread_t pthread_self(void);
```

12.2.2 Terminaison d'un thread

Un thread peut se terminer explicitement en utilisant :

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

– `value_ptr` est la valeur de retour du thread
– l'appel à `pthread_exit` peut se faire en dehors de la fonction utilisée à la création du thread.

Un thread se termine également lorsque la fonction principale (`start_routine`) du thread se termine : la valeur de retour du thread est celle de `start_routine`.

Attention, le pointeur passée comme valeur de retour ne doit pas pointer sur une variable locale de la fonction du thread !

12.2.3 Attente de la fin d'un thread

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

– attend que le thread `thread` se termine
– quand le thread est terminé, `*value_ptr` reçoit la valeur de retour du thread.
– renvoie 0 si ok, un nombre non nul indiquant la cause de l'erreur sinon (thread incorrect, dead lock détecté, ...)
– récupère

Tant que le thread n'a pas été (re)joint, il reste comptabilisé dans la table des threads (comme pour les processus zombis).

12.2.4 Exemple

threads/exemples/bonjour.c

```
/*************************************************************************
 *bonjour.c
 *
 *  (François lemaire) <lemaire@lifl.fr>
 * Time-stamp: <2007-03-14 16:57:19 lemaire>
 *************************************************************************/

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

#include <pthread.h>
#include <unistd.h>

void* bonjour(void *arg) {
  int i;
  i = *((int *)arg);
  free(arg);
  printf("Bonjour_%d!\n", i);

  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {

  int i;
  int *arg;
  pthread_t ths[5];

  for (i=0;i<5;i++) {
    arg = (int*)malloc(sizeof(int));
    *arg = i;
    if (pthread_create(&ths[i], NULL, bonjour, (void*)arg)) {
```

```
      fprintf(stderr, "Erreur de création");
      exit(EXIT_FAILURE);
    }
  }

  for (i=0;i<5;i++) {
    pthread_join(ths[i], NULL);
  }


  exit(EXIT_SUCCESS);
}
```

Se compile avec :

```
$ gcc -Wall -D_REENTRANT -o bonjour bonjour.c -lpthread
$ ./bonjour
Bonjour 0!
Bonjour 1!
Bonjour 2!
Bonjour 3!
Bonjour 4!
$
```

12.3 Verrous

La librairie `pthread` fournit des verrous (type `pthread_mutex_t`) pour les threads.

Initialisation Se fait avec :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Voir aussi `pthread_mutex_init` pour plus de flexibilité.

Verrouillage Se fait avec :

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

L'appel est bloquant jusqu'à ce que `mutex` puisse être verrouillé. Le thread appelant devient propriétaire du verrou.

Déverrouillage Se fait avec :

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Seul le propriétaire peut déverrouiller `mutex`.

Destruction Se fait avec :

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Libère `mutex`. Le verrou doit être déverrouillé.

Les trois fonctions renvoient 0 si ok, et un code d'erreur sinon.

threads/exemples/verrou.c

```
/*****************************************************************
 *bonjour.c
 *
 *  (François lemaire) <lemaire@lifl.fr>
 * Time-stamp: <2007-03-14 16:57:19 lemaire>
 *****************************************************************/

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

#include <pthread.h>
#include <unistd.h>

int valeur;
pthread_mutex_t V = PTHREAD_MUTEX_INITIALIZER;

void inc(int *v) {
  int res;
  res = *v;
  res = res + 1;
  sleep(1);
  *v = res;
}

void* bonjour(void *arg) {
  int i;
  pthread_mutex_lock(&V);
```

```
  inc(&valeur);
  printf("%d\n", valeur);
  pthread_mutex_unlock(&V);
  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {

  int i;
  pthread_t ths[5];

  for (i=0;i<5;i++) {
    if (pthread_create(&ths[i], NULL, bonjour, NULL)) {
      fprintf(stderr, "Erreur de création");
      exit(EXIT_FAILURE);
    }
  }

  for (i=0;i<5;i++) {
    pthread_join(ths[i], NULL);
  }

  pthread_mutex_destroy(&V);
  exit(EXIT_SUCCESS);
}

/*
$ ./verrou
1
2
3
4
5

Sans le lock/unlock

$ ./verrou
1
1
1
1
1

*/
```

12.4 Sémaphores

La librairie `pthread` fournit aussi des sémaphores faciles d'emploi (les sémaphores System V sont plus ardus). C'est le type : `(sem_t`. Contrairement aux verrous, il n'y a pas de notion de propriétaire d'un sémaphore.

Initialisation Se fait avec

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- si `pshared` vaut 0, `sem` est supposé être une variable partagée entre les threads (variable globale, variable allouée dans le tas, ...). sinon, `sem` doit être située en mémoire partagée.
- `value` est la valeur initiale.

Puis-je ? se fait avec

```
int sem_wait(sem_t *sem);
```

Vas-y ! se fait avec

```
 int sem_post(sem_t *sem);
```

Destruction se fait avec

```
int sem_destroy(sem_t *sem);
```

Les quatre fonctions précédentes renvoient 0 si ok, et -1 sinon. Attention, la fonction `sem_wait()` peut-être interrompue par un signal, elle renvoie -1 dans ce cas. Parade : appeler `sem_wait` dans une boucle.

threads/exemples/sem_aff_un_deux.c

```
/*****************************************************************
 *sem_test.c
 *
 *  (François lemaire) <lemaire@lifl.fr>
 * Time-stamp: <2007-03-14 16:42:32 lemaire>
 *****************************************************************/
```

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

#include <semaphore.h>
#include <pthread.h>

sem_t S;

void* mes1(void* unused) {
  sem_wait(&S);
  printf("Et de deux !\n");
}


void* mes2(void* unused) {
  sleep(1);
  printf("Et de un, ...\n");
  sem_post(&S);
}

int main(int argc, char *argv[]) {

  pthread_t th1, th2;

  if (sem_init(&S, 0, 0)) {
    perror("sem_init");
    exit(EXIT_FAILURE);
  }

  if (pthread_create(&th1, NULL, mes1, NULL)) {
    fprintf(stderr,"thread 1");
    exit(EXIT_FAILURE);
  }

  if (pthread_create(&th2, NULL, mes2, NULL)) {
    fprintf(stderr,"thread 2");
    exit(EXIT_FAILURE);
  }


  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
  sem_destroy(&S);

  printf("Fini !\n");
  exit(EXIT_SUCCESS);
}
```

12.5 Considérations pratiques

En pratique, deux erreurs assez surprenantes et inattendues peuvent se produire. La première est liée à la manière dont les variables sont modifiées en mémoire au niveau du processeur. La seconde est lié aux optimisations du compilateur.

12.5.1 Les variables atomiques

Lorsque que deux variables sont rangées côté à côte en mémoire, il se peut que la manipulation d'une variable ait un effet de bord sur l'autre. Plus précisément, les lectures et écritures de variables ne sont pas atomiques. Cela est lié au phénomène suivant :

```c
unsigned char a,b;

/* supposons a=0, b=0 */
a=1;
```

L'affectation `a=1` risque de se faire de la manière suivante :
- chargement depuis la mémoire dans un registre du processeur des 4 octets contenant les variables `a` et `b`
- mise à 1 de la valeur de `a` dans le registre
- recopie en mémoire du registre, qui écrase l'ancienne valeur de `a` et qui remet 0 dans la variable `b`

Ainsi, les deux variables ne sont pas indépendantes. Par conséquent, si un thread ne manipule que `a`, et si un autre ne manipule que `b`, des conflits peuvent quand même se produire, même si `a` et `b` sont utilisées de manière indépendantes.

La solution consiste soit :
- soit à protéger l'accès aux variables visibles par les différents threads par un verrou ;
- soit à utiliser le mot clé `atomic_t` lors de la déclaration des variables (plus de détail dans `atomic.h`.

12.5.2 Les variables volatiles

Le mot clé `volatile` permet d'indiquer au compilateur que le contenu d'une variable risque de changer de manière intempestive.

```
int i=0;
while (i!=0);
```

Si on compile le code précédent avec optimisation (`-O2` par exemple), la variable `i` sera supprimée et le code deviendra quelque chose ressemblant à

```
while (1);
```

Si la variable `i` peut être modifiée de manière externe, cette optimisation est incorrecte.

Si on déclare une variable avec mot-clé `volatile`, le compilateur considère que le contenu de la variable peut changer à tout moment, et ne fait donc plus de suppositions sur sa valeur.

Ce mot clé est utile pour les variables

- en mémoire partagée (variables d'un controleur, ...)
- situées entre `setjmp` et `longjmp`
- partagées entre threads
- ...